Fakultät für Informatik
Technische Universität München

Predictive Autoscaling for Multilayered Cloud Deployments

Vladimir Podolskiy

Fakultät für Informatik
Technische Universität München

Predictive Autoscaling for Multilayered Cloud Deployments

Vladimir Podolskiy

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:
Univ. Prof. Dr. Martin Bichler,

Prüfende der Dissertation:
1. Prof. Dr. Hans Michael Gerndt
2. Prof. Dr. Sven Karlsson,
   Technical University of Denmark

Die Dissertation wurde am 23.06.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 16.11.2021 angenommen.

Zusammenfassung

Für kleine und mittlere Unternehmen ist der Besitz eines Rechenzentrums mit erheblichen Kosten verbunden, die sowohl Hardware und Gebäude als auch Strom und Personal umfassen. Eine zusätzliche übergreifende Herausforderung ist die Variabilität der Last. An einem gewöhnlichen Tag kann es ein erwartetes tageszeitliches Lastmuster geben. Im Gegenteil, ein Black Friday-Ausverkauf kann zehn- bis hundertmal mehr Benutzer anlocken, die das Rechenzentrum mit der schnell ansteigenden Last überfluten. Um diese Herausforderung zu überwinden, die dynamischen Serviceanforderungen zu erfüllen und gleichzeitig die Kosten angemessen zu halten, setzen Unternehmen auf Cloud Computing. Ein typisches Szenario für die Nutzung der Cloud ist das Hosten der eigenen Anwendung in den Rechenzentren von Cloud-Anbietern. Am meistens fungieren die Cloud-Anbietern als gemeinsam genutzte Pools von hochverfügbaren Rechen- und Speicherressourcen. Dazu bieten sie ihren Kunden Garantien für den Servicegrad. Ein weiterer entscheidender Vorteil der Cloud ist ihre Elastizität, d.h. die Fähigkeit, das Ressourcenangebot je nach Nutzungsmuster dynamisch anzupassen. Der Anwendungsbesitzer zahlt also für die tatsächliche Ressourcennutzung. Der Prozess des Hinzufügens und Entfernens von Ressourcen ist in der Regel automatisiert und wird als Autoskalierung-Service bereitgestellt, d.h. ein Service, der die Nutzungsmuster überwacht und die Ressourcen automatisch auf der Grundlage einer bestimmten Verfahrensweise anpasst.

Autoskalierung-Services werden sowohl als Teil von Infrastructure-as-a-Service (IaaS)-Angeboten von Cloud-Anbietern als auch als integrierte Funktion in Container-Orchestrierungslösungen wie Kubernetes angeboten. Häufig wird die Virtualisierung auf Betriebssystemebene in Form von Containern auf Clustern virtueller Maschinen ausgeführt, die von Cloud-Anbietern angeboten werden, um mehr Flexibilität zu bieten und die Bindung an einen Anbieter zu vermeiden. Dieses Setup wird üblicherweise als mehrschichtige Virtualisierung bezeichnet. Die Autoskalierung der Anwendung in einem mehrschichtigen Setup steht vor erheblichen Herausforderungen, da die Skalierungsentscheidungen auf beiden Schichten synchronisiert werden müssen. Ein weiteres Hindernis ist die nicht zu vernachlässigende Boot- und Terminierungszeit, die bei virtuellen Maschinen mehrere Minuten erreichen kann. Täuschend klein, beginnen diese Skalierungszeiten unter einer stark variierenden Last eine große Rolle zu spielen. Der Anstieg der Benutzerlast, der die Bereitstellungsgeschwindigkeit übersteigt, ist ein bekannter Grund für hohe Reaktionszeiten. Um das Problem zu mildern, wurde vorgeschlagen, das Paradigma der automatischen Skalierung von reaktiv auf prädiktiv umzustellen, d.h. die Ressourcen im Voraus so bereitzustellen, dass die vom Endbenutzer wahrgenommene Servicequalität maximiert wird, möglicherweise auf Kosten einer Überbereitstellung.

Diese Doktorarbeit adressiert die Kombination der oben genannten Herausforderungen durch die Einführung eines flexiblen ganzheitlichen Modells der mehrschichtigen Autoskalierung für mehrschichtige Cloud-Einsätze. Der mehrschichtige Aspekt wird in dem Modell erfasst, indem die Autoskalierung als hierarchischer Prozess behandelt wird, bei dem die Skalierung auf der Ebene der Anwendung (d.h. Container) als Treiber für die Skalierung auf der Ebene der VM-Cluster fungiert, die sich wiederum an die Entscheidung auf der Anwendungsebene anpasst. Die folgenden Prozesse bilden den vorhersagbaren Aspekt der Autoskalierung nach dem vorgeschlagenen Modell: i) Lastprognose zur Abschätzung des zukünftigen Bedarfs; ii) Leistungsmodellierung zur Abschätzung, wie sich die Änderung der Ressourcenzuweisung auf die Qualitätsmetriken wie

die Reaktionszeit auswirkt; iii) Topologieanalyse zur Aufdeckung von Leistungsengpässen, die durch die Anwendungsstruktur verursacht werden; iv) Ableitung des Zeitplans zur Ausrichtung der Zeitachse von Skalierungsaktionen gemäß den Zielfunktionen auf Basis der Ergebnisse von i-iii. Jeder dieser Prozesse wird in dieser Doktorarbeit detaillierter bewertet und diskutiert, um begründete Design-Entscheidungen bei der Zusammenstellung der kundenspezifischen vorhersagbaren Autoskalierungsverfahrensweise zu ermöglichen.

Da es keine Einheitslösung für die Autoskalierung gibt, bietet diese Arbeit eine Darstellung einer Autoskalierungsverfahrensweise als eine Kombination von Bausteinen. Diese Bausteine können auf die besonderen Eigenschaften der skalierten Anwendung und die unkontrollierten Umgebungsparameter abgestimmt werden, so dass die Ziele des Anwendungsbesitzers bestmöglich erfüllt werden. Mit Hilfe einer im Rahmen dieser Arbeit entwickelten Simulations-Toolbox bewerten wir den separaten und kumulativen Einfluss der Prognose- und Leistungsmodellierung auf die Qualität der Entscheidungen durch vorhersagbare Autoskalierungsverfahrensweisen für mehrschichtige Cloud-Einsätze. Es werden auch verschiedene Arten von Metriken untersucht, um ihre Nützlichkeit als Entscheidungsmetriken für den Autoskalierungsprozess aufzudecken. Diese Experimente werden durch die diskrete Ereignissimulations-Engine Multiverse ermöglicht, die die wichtigsten Abstraktionen und Abstimmknöpfe der Autoskalierung für mehrschichtige Cloud-Einsätze genau modelliert. Sie lehnt sich in ihren Abstraktionen eng an Kubernetes an und greift auf die realen Ressourcenauslastungen und Lastspuren, Anwendungstopologien und Skalierungszeiten zurück, um die Gültigkeit der Simulationsergebnisse zu gewährleisten. Mit diesen Beiträgen hoffen wir, die Grundlagen für die Untersuchung der dynamischen Ressourcenbereitstellung über mehrere Virtualisierungsschichten hinweg zu schaffen und eine Reihe von Werkzeugen anzubieten, die diese Untersuchungen in der Laborumgebung mit begrenztem oder gar keinem Zugang zu groß angelegten Implementierungen und kostspieliger Infrastruktur ermöglichen.

Abstract

For small and medium businesses, owning a data center incurs significant cost covering hardware and building as well as power and personnel. An additional overarching challenge is the variability of load. On an ordinary day, there might be an expected diurnal load pattern whereas a Black Friday sale may lure tens to hundreds times more users overflowing the data center with the rapidly surging load. To address the challenge of meeting the dynamic service demands while keeping the costs affordable, businesses adopt cloud computing. A typical scenario for using cloud is to host one's application on premises of cloud services providers (CSPs). CSPs predominantly act as the shared pools of highly-available compute and storage resources and provide their customers with guarantees on the level of service. Another crucial benefit of the cloud is its elasticity, i.e. the capability to dynamically expand and contract the resource offering depending on the usage patterns. Thus, the application owner pays for the actual resource usage. The process of adding and removing resources is usually automated and provided as an autoscaling service, i.e. the service that monitors the usage patterns and adjusts resources automatically based on some policy.

Autoscaling services come both as a part of Infrastructure-as-a-Service (IaaS) CSPs offerings and as a built-in feature in container orchestration solutions, such as Kubernetes. Often, OS-level virtualization in form of containers runs on top of clusters of virtual machines offered by CSPs to add more flexibility and avoid vendor lock-in. This setup is commonly addressed as a multilayered virtualization. Autoscaling the application in a multilayered setup faces significant challenges due to the need to synchronize the scaling decisions on both layers. Another obstacle is the non-negligible booting and termination time reaching several minutes for virtual machines. Deceptively small, these scaling times start to play a big role under a highly variable load. The increase in the user load that outpaces the provisioning speed is a known reason for high response times. To mitigate this issue, it was proposed to shift the autoscaling paradigm from reactive to predictive, i.e., to provision the resources ahead of time such that the end user experienced quality of service is maximized maybe at the cost of overprovisioning.

This thesis addresses the combination of the above challenges by introducing a flexible holistic model of predictive autoscaling for multilayered cloud deployments. The multilayer aspect is captured in the model by treating autoscaling as a hierarchical process where the scaling on the level of application (i.e. containers) acts as a driver for scaling on the level of VM clusters, which, in turn, adjusts to the decision on the application level. The following processes constitute the predictive aspect of autoscaling according to the proposed model: i) load forecasting to estimate the future demand; ii) performance modeling to estimate how the change in the resource allocation impacts the quality metrics such as response time; iii) topology analysis to uncover the performance bottlenecks induced by the application structure; iv) scheduling plan derivation to align the timeline of scaling actions according to the goal functions based on the results of i-iii. Each of these processes is evaluated and discussed in the thesis in greater detail to allow for justified design decisions when composing the customized predictive autoscaling policy.

Due to the absence of one-size-fits-all solution for autoscaling, this thesis offers a representation of an autoscaling policy as a combination of basic building blocks. These blocks may be tuned to address the

peculiar properties of the scaled application and the uncontrolled environment parameters such that the objectives of the application owner are fulfilled in the best possible way. Via the means of a simulation toolbox developed in the scope of this thesis, we assess the separate and cumulative impact of forecasting and performance modeling on the quality of decisions by predictive autoscaling policies for multilayered cloud deployments. Different kinds of metrics are also studied to uncover their usefulness as decision metrics for the autoscaling process. These experiments are made possible by the discrete event simulation engine Multiverse which accurately models the main abstractions and tuning knobs of autoscaling for multilayered cloud deployments. It closely follows Kubernetes in its set of abstractions and draws on the real-world resource utilization and load traces, application topologies, and scaling times to ensure the validity of the simulation results. With these contributions, we hope to lay down the foundations of studying the dynamic resource provisioning across multiple virtualization layers and to offer a set of tools to enable these studies in the lab environment with limited to no access to large-scale deployments and costly infrastructure.

Acknowledgments

For me, the last year was dedicated to wrapping up the research and writing down this thesis. This was an incredibly challenging time for me, my family, and my friends. From the bottom of my heart, I thank all of you who stayed by my side and supported me!

First of all, I would like to thank my wife Anastasia, who went with me to Germany and supported me on my path towards PhD for the last five years. I'd like to thank my parents Valentina and Edward for always believing in us.

This journey would have been impossible without the support from my friends. I would like to thank Vladislav and Panos for believing in me and supporting me even in the toughest times. I would like to thank Anshul who put enormous efforts into materializing our ideas even when their feasibility was at best doubtful. I want to thank Masha whom we met in Munich for her totally different perspective on the world that challenged my own views. I thank Konstantin, Stanislav, and Dmitriy for being supportive comrades and welcoming me whenever I happen to drop by in Moscow. I'm thankful that the destiny connected me with such amazing and deep people as Sergey Koloskov, Alexander Kriebitz, and, of course, Sergey Kamolov. I want to thank Anastasia's family, especially Oleg, Anna, and Katya, for always being reachable, organizing fun trips, and engaging in marvelous discussions.

I'm deeply indepted to all the brilliant people who mentored me, supported me on my professional journey, and taught me valuable lessons. I'm grateful to Aleksey Popov for going an extra mile to give me challenging problems to solve and supporting me in my scholarship applications. I want to thank Roman Samarev and Galina Ivanova for their significant contributions to my knowledge and experience. I want to direct the words of gratitude to Ivan Beschastnikh and Animesh Trivedi for pushing me to rethink my experience and discover my blind spots. I'm thankful to Liubov Nekrasova, Anatoliy Simkin, and Dmitriy Voloshin for teaching me how to work and for offering me the opportunities for professional and personal growth. Perhaps, this journey could have ended abruptly if not for the efforts of two superstar school teachers – Olga Molokanova and Natalia Barchenkova. The success of their efforts in bringing my knowledge of physics and mathematics on par with the entry-level standards of Bauman University is something not to be underestimated.

I would like to thank my colleagues, former and current, at TUM CAPS for the technical discussions and their support, especially Martin Schulz, Dai, Andreas, Robert, and Madhura. My special thanks go to Jürgen and late Beate.

I've met countless amazing people who helped me along the way, challenged me, and spent many hours discussing various topics. I thank Sana, Roula and Antonios, Evgeniy Puzikov, Abel Souza, Felipe Oliveira Gutierrez, André Bauer, Nikolas Herbst, Sven Karlsson, Alexandru Iosup, Sybille Wahl and the DAAD Freundeskreis, Alexander Kliymuk, Yury Oleynik, Erik Elmroth, Jakub Krzywda, Johan Tordsson, Ahmed Ali Eldin, Cristian Klein, Krzysztof Rzadca, Michael Mayo, Abigail Koay, Kenneth B. Kent, Anna Kobusińska, Daniel Seybold, Jörg Domaschka, Shajulin Benedikt, Dominik Böhler, Lydia Chen, Andrey Proletarskiy,

and also my former students, especially Atakan, Yesika, Harshit, Seif, and Helge. I appreciate the hospitality of the University of Waikato and Umeå University. I'm grateful to be part of ACSOS/ICAC/SASO and Software Campus communities.

My bow to Yūki Tabata, Gabe Newell, Hideaki Anno and their teams for bringing the long-lasting inspiration into my life.

I thank German Academic Exchange Service (DAAD), the state of Bavaria and the Technical University of Munich for financially supporting my work and being welcoming hosts for me and my wife. Vielen Dank!

I want to finish this section by aknowledging the person who supported me to come abroad and work on my doctoral studies. I want to thank my supervisor, Professor Michael Gerndt, for discovering the right approach to me and cultivating my curiousity in new technologies. I'm particularly grateful for instilling the right motivation to do the research and the work ethic that is expected of a researcher.

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

1.1 Cloud Computing: the Challenge of Scaling an Interactive Application

The recorded history of cloud computing starts in 2006 with Amazon releasing its Elastic Compute Cloud (EC2) service[1]. Since then, **elasticity**, defined as the dynamic on-demand provisioning of cloud resources, became the key selling proposition of cloud services providers (CSP) [1]. The predominance of elastic services in cloud offerings is caused by the demand from businesses whose services are provided to the *changing* user base. Instantaneous drops and surges in the user load are typical for interactive cloud applications. On-demand dynamic provisioning of cloud resources allows to cope with the changing load without necessarily shouldering an enormous bill for the unused resources in times of load drought. This property of cloud is appealing to companies of all sizes and stages of maturity.

Cloud computing paradigm has the highest value for companies in two broad categories. The first comprises the established businesses with large user base of hundreds of thousands to millions. This category is captivated by the 'pay-as-you-go' model of cloud that allows to *adapt* the provisioned cloud resources to the current demand at no extra cost – one pays the same price for 1,000 virtual machines running for 1 hour and for 1 virtual machine running for 1,000 hours [2]. The second category is mainly composed of start-ups whose initial aim is to scale their user base up rapidly, on the timescale of 1-5 years. Having low initial budgets at their disposal, start-ups prefer to not overpay for serving just a handful of first customers. At the same time, they appreciate an opportunity of unbounded scaling helping them to rapidly seize the market.

Nowadays, numerous applications are brought into cloud for its dynamic resource provisioning capability. Majority of these applications falls into a couple of broad categories, viz, *transactional* (e.g. web shops, online banks) and *streaming* (e.g. audio/video streaming, gaming)[2]. The most prominent examples from each of these categories are the online payment system Paypal[3] and the online music streaming service Spotify[4],

[1] https://aws.amazon.com/ru/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/

[2] https://docs.microsoft.com/en-us/windows/win32/winsock/transactional-versus-streaming-applications-2

[3] https://cloud.google.com/customers/featured/paypal

[4] https://cloud.google.com/customers/featured/spotify

correspondingly. The distinctive characteristic of cloud applications from both categories is that they have a user-facing *frontend* delivered over the Internet and a *backend* that does all the data processing and storing. To support their customers, cloud services providers offer an abundance of optimized CSP-managed services that serve as building blocks of cloud applications[5]. Albeit not without their own drawbacks, such as vendor lock-in, these building blocks allow the cloud adopters to focus on the business logic when implementing an application.

Success of the user-facing applications heavily depends on *response time* – the most natural user experience can only be achieved at the scale of **100 ms** [3, 4]. For large online services providers such as Amazon and Google, an increase in the response time by further 100 ms converts into a significant revenue loss[6]. With the rise in response time accompanied by the diminishing throughput, users of streaming services such as Netflix and Youtube may experience instability in video and sound which is partially mitigated by buffering but not at all times [5, 6]. Multiplayer online gaming experience is even more susceptible to surges in response time since an online game has to provide the ordered video frames in real time [7, 8]. With the half of the office employees working at least 1-2 days from home due to COVID-19 pandemic at the time of writing[7], the worldwide adoption of videoconferencing tools such as Skype, Hangouts, MS Teams, and Zoom paved the path to rising response times caused by the surging load[8]. Without a doubt, the damage to user experience can divert a significant chunk of the user base in view of many new offerings [9, 10].

Appeal of cloud is in the *guaranteed level of service* offered. To a large extent, these guarantees are responsible for low response time. Cloud services providers are always willing to push these guarantees up since the application owners are always in search for better quality of cloud services at a lower price. Infrastructure ownership allows CSPs to control the delivered level of service to a considerable extent. However, it does not end there – the last mile of the application service delivery is in the custody of the application owner.

CSPs cannot provide the application-level guarantees, however, they usually offer an *autoscaling service* to improve on that frontier. The autoscaling service automates the process of dynamically provisioning cloud resources for the application owner. Instead of tracking the user load and the system resources usage and manually starting or terminating **virtual machines** (VM), the application owner can enable this service, configure it, and let it automatically provision the required cloud resources. Originally, autoscaling service was provided by CSPs on the level of VM clusters and made its decisions based on comparing the resource utilization of these clusters to the predefined threshold (cf. the original Auto Scaling service by AWS). Aside from choosing these thresholds wisely, the application owner had to carefully parallelize her application running on a single VM and include its code into the VM image. Once a VM had started, it was too cumbersome to adapt the count of application instances inside it. The old era of pure VM-based cloud ended rapidly with the rise of **containers**.

Containers are the units of software packaging that leverage the OS-level virtualization for rapid booting and portability[9]. Multiple containers can run on the same VM or physical host by time-sharing the underlying hardware. These features of containers enabled easier application deployment and scaling. The advance of container orchestration systems, with Kubernetes being the most widely adopted[10], allowed to introduce autoscaling to the application level. Now, in addition to dynamical provisioning of virtual machines to satisfy the users' demand, application owners were able to automatically adjust either the number of application containers running within any VM (horizontal scaling) or system resources allocated to them on the node

[5] https://aws.amazon.com/products/

[6] http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html

[7] https://www.economist.com/briefing/2020/09/12/covid-19-has-forced-a-radical-shift-in-working-habits

[8] https://answers.microsoft.com/en-us/msteams/forum/msteams_meet/teams-latency-problems/aeb3ac52-0146-47a5-8bbb-0da27bcf5428

[9] https://www.docker.com/resources/what-container

[10] https://www.cncf.io/wp-content/uploads/2020/08/CNCF_Survey_Report.pdf

(vertical scaling). In the early days of containerized applications, the VM-level autoscaling and application-level autoscaling did not go well together [11]. Having realized that the end goal is balancing the user satisfaction with the efficient use of already available resources, the developers of container orchestration platforms added *cluster autoscaling* responsible for scaling VM clusters through APIs offered by CSPs. With that, application-level autoscaling became a driver for the VM cluster scaling. This resolved the coordination problem across multiple virtualization and resource management layers, and CSPs started to offer the *managed Kubernetes* services or in-house container management services (e.g. Amazon Elastic Container Service). Although the application owners largely benefited from being able to adapt both the application and the underlying VM clusters, there still remained a challenge of curtailing response time which happened to sky-rocket during load spikes. Existing autoscaling services were not much of a help since they offered only the *reactive* mechanism, i.e. the application and/or VM cluster was scaled up only *after* the event demanding this scale-up happened. What autoscaling services missed at that time was being able to act in *anticipation* of the load increase.

Predictive autoscaling, or proactive dynamic provisioning of cloud resources, emerged as a response to the above challenge [12]. The core idea is simple: the historic observations of the application load or other meaningful metrics are extrapolated into the future (forecasted), and then, based on the performance model of the application, its resources are automatically adapted to accommodate the future load at what is deemed to be the right time. This mechanism can be supported with the reactive contour in case of a wrong forecast.

As it happens to any idea appealing in its simplicity, a variety of designs and implementations of predictive autoscalers emerged. Some implementations were more conceptual and illustrative of particular mathematical models and techniques [12, 13]. Other implementations tackled specific cloud resource types, certain applications, or application classes [14, 15, 16, 17]. Lastly, industry offered own predictive autoscaling services to their customers[11] and even used them in production [18]. All the proposals have one thing in common – every new research or a product takes an established theoretical apparatus, ranging from control theory and offline application profiling to deep learning, and builds on top of it. Naturally, each new predictive autoscaling solution is shown to keep the promise on particular applications/benchmarks under certain variety of workloads. What the existing body of research and the commercial services alike evade is the following question: *what stands behind the power of predictive autoscaling?* We may indeed have many solutions for the same problem, but we do not yet know what makes them do what they do and how do they fill into a bigger picture of predictive autoscaling and compare with each other.

To address the above question, this thesis offers a systematic aspect-wise study of predictive autoscaling as an inherently complex process consisting of multiple interrelated building blocks. Building on the aspect-wise study methodology, this thesis attempts to find out what exactly makes predictive autoscaling good for enabling millisecond-scale response times for containerized cloud applications at scale. As a result of this study, we introduce a theoretical generalized predictive autoscaling framework, the delta arithmetic on application and platform states that helps to describe the autoscaling process on multilayered cloud deployments in time, and the view of an autoscaling policy as a composition of primitive operations with the estimates of their impact on achieving the desired response times. The simulation toolbox devised in the scope of this thesis helps to investigate predictive autoscaling policies in-depth based on the real-world resource utilization and load traces.

1.2 Thesis Contribution

In this thesis, we make an attempt to uncover the inherent complexity of predictive autoscaling for multilayered cloud deployments. In particular, we show how its building blocks contribute to achieving the goal of

[11] `https://aws.amazon.com/blogs/aws/new-predictive-scaling-for-ec2-powered-by-machine-learning/`

improving response times for interactive transaction-based containerized applications deployed in the cloud over the reactive baseline. To achieve this goal, this thesis:

- proposes a generalized predictive autoscaling architecture that embodies several processes contributing to the dynamic resource allocation,
- assesses various load metrics forecasting methods and compares them using the novel methodology,
- proposes novel performance modeling and dynamic resource provisioning approach for vertical and horizontal autoscaling based on combination of machine learning with optimization,
- studies publicly available microservice applications to discover the underlying structural regularities beneficial for predictive autoscaling and synthesis of realistic application topologies,
- introduces an execution model that bridges the gap between autoscaling mechanisms and policies and the main entities which the autoscaling process is defined for, viz, application, services, cloud platform, virtual machines, system-level resources, and user load,
- introduces and validates an open source toolbox to simulate autoscaling policies based on real VM and resource utilization and load traces,
- quantifies the impact of decision metrics and of particular building blocks (load forecasting and performance modeling) on the quality of predictive autoscaling and devises the theoretical timeliness/relevance framework for navigating the space of autoscaling decision metrics.

1.3 Characteristics of the Thesis Contribution

The contribution of this thesis is characterized as follows:

- **Focus on horizontal autoscaling**. The most important contributions of this thesis are applicable to horizontal autoscaling, i.e. dynamic provisioning of discretized resources (VMs and containers). This allows us to focus on unbounded scaling that is not limited by the resources of an individual node.
- **No 'one-size-fits-all' solution offered**. This thesis attempts to generalize beyond a single solution by offering a generic predictive autoscaling architecture and autoscaling policies building blocks. The blocks are assessed using simulation to quanitfy their effect on the quality of autoscaling.
- **Perspective of the party that runs the application**. The thesis focuses on the application that is deployed in the cloud. The application can be deployed in the cloud in multiple ways. Infrastructure-as-a-Service (IaaS) makes the application owner responsible for deploying and managing the application whereas Function-as-a-Service (FaaS) model takes this responsibility away from the application owner. The results in this thesis are aimed at whichever party actually manages the application and runs it. This party differs depending on whether IaaS or FaaS model is adopted. The data center point of view is outside of the thesis' scope.
- **Simulation-based evaluation**. The simulation-based evaluation of autoscaling policies was adopted for several key reasons. The first one is that there are not enough real-world microservice applications and loads available for the *in vivo* evaluation of predictive autoscaling policies. The second one is that the *in vivo* evaluation struggles with separating the autoscaling-induced service level changes from the effects of particular technology stacks. The third one is the high cost of *in vivo* experiments, especially those involving deep learning-based performance models. The simulation-based evaluation is limited in the validity of the results and their immediate practical use. The first concern is addressed by assessing the validity of the simulator from different points of view. The second concern is left outside of the thesis' scope since it depends on the concrete application.

- **No focus on a particular cloud services provider**. On one hand, this is an advantage since we avoid vendor lock-in and generalize the contribution across various cloud services providers. On the other hand, this is a limitation since the results in this thesis have no immediate practical use in the production autoscaling solutions.

1.4 Structure of the Thesis

The thesis consists of 11 chapters which are briefly introduced in the following paragraphs.

Chapter 1 gives an outlook on the problem domain of predictive autoscaling for multilayered deployments in the cloud and introduces the gap that this thesis aims to fill.

Chapter 2 provides an overview of technologies that the modern cloud computing is built upon, introduces autoscaling and related terms, and discusses the examples of production-grade autoscaling services.

Chapter 3 shows limitations of the widely-adopted reactive autoscaling for multilayered cloud deployments in ensuring the millisecond-scale response times under peak loads.

Chapter 4 discusses the related work and places the thesis contribution into its context.

Chapter 5 introduces the generalized predictive autoscaler framework that aims to impose the structure on the wide variety of existing and future predictive autoscalers.

Chapter 6 assesses the usefulness of various time series forecasting methods for the task of load forecasting in predictive autoscaling. The chapter maps forecasting methods onto a two-dimensional space of interval accuracy/model fitting time to enable the selection of an appropriate method for load forecasting.

Chapter 7 lays down a compound two-stage process of performance modeling and dynamic resource allocation for vertical and horizontal autoscaling. This process consists of deriving a performance model that maps the resource utilization and the current resource allocation onto the application quality metric (e.g. throughput or response time) and using it to adjust the resource allocation such that the service level is improved and the cost of the deployment is minimized.

Chapter 8 uncovers topological regularities underpinning publicly available microservice applications and determines a random graph-based model and a set of parameters necessary to generate application topologies for realistic autoscaling simulations. The discoveries highlight the importance of resolving structural application bottlenecks (weakest links) to achieve higher quality of autoscaling.

Chapter 9 presents an execution model for multilayered cloud application scaling and introduces an open-source toolbox that builds on this model. The toolbox consists of five tools: an autoscaling policies simulator **Multiverse** that supports multilayered deployments, a simulation results visualizer **Stethoscope**, an alternative policies evaluation manager **Cruncher**, an automatic trace-based autoscaling simulations generator **Praxiteles**, and a deep networks-based performance models training tool **Training Ground**.

Chapter 10 starts by validating the simulator **Multiverse** and proceeds by using it for quantifying the importance of decision metrics as well as forecasting and performance modeling building blocks of predictive autoscaling policy for the quality of scaling actions. The reactive autoscaling policy is used as a baseline.

Chapter 11 concludes the thesis and outlines perspective directions of future work.

2

Background

2.1 Cloud Computing

2.1.1 Virtualization: an Enabling Technology for Cloud

2.1.1.1 Hardware virtualization

Virtualization is an act of creating a virtual representation of a physical resource in a computer system. The main purpose of virtualization is to create an illusion of an isolated and dedicated resource for the piece of software that uses it. In addition, virtualization offers the software secure access to the shared physical resources in the multi-tenant environments.

Virtualization concerns all the major kinds of physical resources in computer systems, i.e. compute, storage, and network. Unlike physical resources such as memory or processors, their virtualized counterparts (virtual memory and virtual CPUs, vCPUs) are not limited by the capacity of the physical devices that they represent. For instance, virtual memory might be orders of magnitude larger than the physical memory of a machine. This is achieved by using the disk space to swap rarely used memory pages out of RAM. That way, each piece of software running on the same physical host may even have its dedicated visible memory matching in size that of the physical host itself. Naturally, when the capacity of the physical memory is reached and the swapping starts, the performance of the software drops since the secondary storage has higher access latency. In case of memory, virtualization is performed by the hardware memory management unit (MMU) that translates virtual addresses into physical addresses of the host. In principle, any kind of virtualization is achieved by introducing a *new level of indirection* that manages the lower-level resources and provides their representation to the higher-level consumers in a shared manner.

Going beyond a single type of resource such as memory, one can broadly refer to the *hardware virtualization* in which all of the hardware is virtualized for the shared use by the software. Virtualization is provided by the piece of software (also hardware and firmware) called virtual machine monitor (VMM), aka hypervisor. Examples of widely-adopted hypervisors include Hyper-V[1], QEMU[2], Xen[3], and VMware ESXi[4]. Hyper-

[1] https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview
[2] https://www.qemu.org/
[3] https://xenproject.org/
[4] https://www.vmware.com/products/esxi-and-esx.html

visor runs on a host machine and provides virtualized resources to *virtual machines*, aka *guest machines*. Hypervisors belong to one of two broad categories: *bare-metal hypervisors* (type-I) and *hosted hypervisors* (type-II) [19]. As the name suggests, hypervisors in the first category run directly on the hardware which explains their superior performance when compared to type-II hypervisors. The hosted hypervisor, in contrast, runs as an application on top of host's operating system (e.g. VirtualBox[5]). Hypervisors offer one of two types of hardware virtualization, viz, full virtualization and paravirtualization.

Full virtualization emulates the underlying hardware that allows the guest operating systems to run unmodified. This type of virtualization does not introduce an additional overhead of modifying the software, but falls short of offering the performance comparable to that of paravirtualization. **Paravirtualization** offers guest OS to run in an isolated domain by being modified to substitute privileged operations of ring 0 with the calls to hypervisor (aka hypercalls)[6]. By adopting this approach, paravirtualization decreases the slowdown for applications that run on virtual machines at the cost of additional efforts required to modify the OS kernel.

Cloud services providers install hypervisors on their physical servers to offer virtual resources over the Internet. For instance, Google Compute Engine relies on KVM for virtualization[7]. Microsoft customized Hyper-V to use in their Azure cloud[8]. AWS went as far as rolling out their own AWS Nitro System that is a hypervisor assisted by the in-house built chips encapsulating network and storage resources[9].

2.1.1.2 OS-level virtualization and containers

By allowing multiple user spaces, modern operating systems enable OS-level virtualization. In this type of virtualization, programs run in the so-called *containers* that encapsulate all the dependencies required to execute them. The encapsulation results in a high portability of containers, i.e. if one wants to start the container, then there is no need to manually install the required dependencies since they are already incorporated in the *container image*.

A container image, such as a Docker image[10], is a template with instructions to create a container instance. Images describe all the dependencies and the setup activities that are required to start the container. Programs that run in a container can only access the contents of this container and the system resources that are assigned to it. Containers run on the system concurrently and are scheduled for execution by the OS scheduler, e.g. as *tasks* in Linux. With help of OS-level mechanisms such as **cgroups**, one can impose limits on system resources consumed by containers. For instance, limiting the CPU shares available to the container reduces the duration for which it can run on CPU before being interrupted by the OS scheduler.

To run a container, one needs a *container runtime*. Container runtime is a program that creates containers from their descriptions (images) and executes them. Container runtime has to be present on a virtual or a physical host in order to run the containers. There are multiple container runtimes available at the moment of writing, including the most widely-adopted ones such a Docker[11], containerd[12], runC[13], rkt[14], and

[5] https://www.virtualbox.org/

[6] https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf

[7] https://cloud.google.com/blog/products/gcp/7-ways-we-harden-our-kvm-hypervisor-at-google-cloud-security-in-plaintext

[8] https://docs.microsoft.com/en-us/azure/cloud-adoption-framework/get-started/what-is-azure

[9] https://aws.amazon.com/ec2/nitro/

[10] https://docs.docker.com/get-started/overview/

[11] https://www.docker.com/

[12] https://containerd.io/

[13] https://github.com/opencontainers/runc

[14] https://coreos.com/rkt/

lxc[15]. The diversity of container runtimes led to emergence of Open Container Initiative (OCI) that aims at standardizing the container runtime and image specifications[16].

Although containers allow to bundle the software and libraries and ship them to any Linux distribution supporting cgroups, manual management of each container does not scale well. To overcome this challenge, *container orchestration software* was introduced. Kubernetes[17] stands out among this kind of software since it is open-source and backed by Google's engineering experience in large clusters management. In addition to orchestrating containers executed using different runtimes[18], Kubernetes also manages virtual machines clusters that the containerized applications are often deployed on using APIs of public cloud services providers and open-source cloud computing platforms. Kubernetes automates management of containerized applications and ensures communication between the containers grouped into *pods* via *overlay networks*[19].

Containers are appealing to the cloud users because of their lower booting time, portability, and direct resource management capability. At the moment of writing, the cloud users can either directly deploy the containers on virtual machines or indirectly by submitting their code to the function-as-a-service (FaaS) providers. In the latter case, the provider receives the executable code and starts it in the container. This relieves the application owner of the container orchestration and resource management burden.

2.1.1.3 Beyond conventional virtualization

Virtualization is still an active research direction where most efforts are now directed at the *specialized virtualization technologies*. One of such research directions aims at supporting new cloud services models such as Function-as-a-Service (FaaS). Recently, AWS open-sourced Firecracker, a virtual machine monitor that specializes at serverless workloads to hold the promise both of strong security and the minimal virtualization overhead [20]. A whole new degree of specialization is introduced by allowing to compile the application and the configuration code into the fixed-purpose image, so-called unikernel [21]. Unikernels run directly on a hypervisor such as Xen or on the hardware. Unikernels have lower resource footprint than VMs and containers, and can be started way faster. One of the possible uses of unikernels, such as those constructed with MirageOS[20], is to start them for each new end user request. This application scenario enables an unprecedented degree of isolation.

The ever growing market of cloud services is limited in one major sense – there is only one hypervisor running on a physical server, and the cloud user is not allowed to select which hypervisor it is. There is, however, an incentive to overcome this hypervisor lock-in with *multi-hypervisor* technology that allows to run multiple compartmentalized hypervisors on the same physical host [22]. Alternatively, one can already use the *nested virtualization*. This kind of virtualization can be implemented as a chain of unmodified hypervisors running on top of each other [23]. Nested virtualization allows better flexibility in terms of choosing a particular hypervisor but at the cost of introducing additional layers of indirection which may incur unacceptable performance overheads.

With the growth of cloud market and emergence of new cloud service models, we may expect to see more purpose-built hypervisors and other virtualization tools.

[15] `https://linuxcontainers.org/`
[16] `https://opencontainers.org/`
[17] `https://kubernetes.io/`
[18] `https://kubernetes.io/docs/setup/production-environment/container-runtimes/`
[19] `https://kubernetes.io/docs/concepts/cluster-administration/networking/`
[20] `https://mirage.io/`

2.1.2 Cloud: on the Way to 'Effortless' Computing

2.1.2.1 Cloud ownership models

Cloud computing is usually categorized into private, public, hybrid, and multicloud[21].

Private cloud serves single organization and is usually allocated within the pool of resources of some public cloud services provider. In that sense, it barely differs from on-premise infrastructure since it is not shared with other parties. Private cloud is usually celebrated for its better isolation and improved security in comparison to the public cloud services. The cloud resources may still be shared among multiple departments of the same organization.

Public clouds are offered to anyone by the owners of large data centers such as Amazon and Microsoft. The distinctive property of public clouds is that the infrastructure is shared among many *tenants*. This allows public cloud services providers to leverage the economy of scale – the same physical infrastructure is shared in time and space. Consequently, the public cloud users are billed at a relatively low price. The disadvantage of public clouds is in the performance interference and instability generated by other tenants and in the potentially higher exposure to coordinated attacks.

To compromise on benefits and drawbacks of public and private clouds, some companies leverage **hybrid clouds** which allow to serve mission-critical workloads in the private part whereas the public part may accommodate unexpected surges in traffic. When the cloud user is concerned about the cloud provider lock-in, they may resort to **multicloud**, i.e. cloud offerings spanning multiple cloud providers. Multicloud is particularly useful to avoid site- or worldwide-outages of a single cloud services provider.

The following subsections focus on the public cloud offerings as having the highest diversity among all cloud ownership models discussed.

2.1.2.2 Cloud services pricing models

Public cloud services providers offer various pricing models depending on their customers needs and ability to pay as well as on their own goals. With the diverse pricing models, cloud services providers intend to maximize the *resource utilization* of their data centers such that the fixed costs for running them are recouped. Below we discuss the most widely-used cloud services pricing models.

Pay-as-you-go is the basic and most widely-known pricing model for cloud services. It emerged along with the IaaS cloud services model. Cloud services provider sets the price for a particular service per unit of time, e.g. hour, and this price does not change. It does not matter whether the service was used, the billed amount depends only on the quantity of service consumed and its price tag. If the deployed virtual machine was not used for some time, still, the cloud user is billed for occupying the cloud resources.

Pay for what you use is a fairly recent cloud services pricing model. Its wide adoption began with the FaaS cloud services model, although it can be tracked back to the cloud API services that are also billed for the actual use based on the total count of requests issued against them[22]. For example, AWS Lambda bills the cloud user both for the number of requests issued by the end users against the functions and for the duration that the memory was occupied for by the functions (measured in GB-seconds)[23]. Notably, the requests are billed in bulk, e.g. the cloud user pays for each 1 million requests made to her application. Presumably,

[21] https://www.redhat.com/en/topics/cloud-computing/public-cloud-vs-private-cloud-and-hybrid-cloud
[22] https://aws.amazon.com/cloudwatch/pricing/
[23] https://aws.amazon.com/lambda/pricing/

accounting for individual requests on the scale of Amazon and other large cloud services providers results in an unnecessarily high resource overhead in terms of storage and compute.

Spot market is a pricing model inspired by the economic model of supply and demand. To implement this model, cloud services provider has a dynamically changing fleet of unused virtual machines which currently free cloud data center resources are sliced into. Price for the virtual machines in this so-called *spot fleet* is set based on both the current supply of these machines and the current demand for them. Supply and demand are mutually related. Higher supply drives the price for spot instances down to make them more attractive for purchasing, which results in cloud users getting more of these instances. This, in turn, drives the prices up since the supply got lower. This process continues until an equilibrium point is reached, which is still just a temporary state. Naturally, cloud services provider defines the minimal spot price that is lower than the on-demand price for VM of the same type[24]. If the spot price gets higher than the price offered by the cloud user, the spot instance will be revoked. Hence, these instances are not suited for tasks requiring reliability. One may use spot instances to cover non-critical tasks such as accelerating distributed deep model training that is otherwise being handled by reserved instances equipped with GPUs.

Subscription based pricing model requires the cloud user to pay for a fixed amount of service to be available over a long continuous interval of time, usually in advance. Subscriptions are usually offered at a discount to the on-demand prices since they make the demand more predictable for cloud services providers and thus reduce risks of not selling enough services to cover the expenses. A representative example of the subscription based pricing model is a *reserved instances* offering by AWS[25]. The cloud user can commit for 1 or 3 years of using virtual machines offered by AWS. In contrast to other payment options, the cloud user may pay the costs of reserved instances upfront. This option is attractive for cloud users that have predictable minimal workload level such as internal workloads, e.g. office systems to support business continuity such as finance and supply chain management.

Different pricing models might be combined to achieve goals of the cloud user in a better way. Combination of reserved and spot instances allows to accelerate execution of lengthy tasks such as simulations and deep model training and at the same time to keep the costs low. Reserved instances may also be supported by the on-demand instances if the dynamically changing workload is more critical and there exists a more or less stable user base.

2.1.2.3 Cloud computing service models

Public cloud providers offer multiple ways of how their services can be used – these ways are conventionally grouped into the so-called *cloud computing service models*. The selection of a model depends on the business needs and business processes as well as on the software deployment processes that the company has in place. The cost and performance is yet another significant factor that impacts the decision to choose this or that service model. In the following paragraphs, we focus on several widely adopted cloud service models. By no means is this list exhaustive since new cloud service models tend to pop up every now and then, e.g. Benchmarking-as-a-Service[26]. One needs to note that there is no crisp border between these models [2], hence the following discussion can not be considered precise. It serves the purpose of simplifying the categorization and navigation across the multitude of cloud offerings.

Infrastructure-as-a-Service (IaaS) is a model that comprises offerings enabling the user to work with the *virtualized system resources* directly. In IaaS, the cloud user creates virtual machines and runs his workloads on them as though they were physical servers. IaaS is the earliest model of cloud services. This model

[24] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html
[25] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-reserved-instances.html
[26] https://idw-online.de/de/news761055

constraints the cloud user in terms of virtualized system resources that she can order. These resources are sized by the cloud services provider into various *types* (aka *flavors* in OpenStack terminology) that differ from each other by the amount of quantifiable virtualized resources offered. For example, **t3a.large** EC2 instance type (AWS) offers 2 vCPUs and 8 GBs of memory in comparison to **t3a.xlarge** instance type of the same provider that comes with 4 vCPUs and 16 GBs RAM. Cloud user is not allowed to set the resource requirements arbitrarily. This allows CSPs to make the demand for their resources more predictable. IaaS model is the most complex in use since it allows a great deal of freedom in managing cloud resources, configuring operating systems that run on VMs, deploying the application and ensuring its security. Although easier-to-use cloud services models emerged over time (see below), IaaS is still a go-to choice for the *performance-critical applications* or *legacy applications* which are not used that much to justify the expenses of switching to another model. The most prominent example of IaaS services is AWS Elastic Compute Cloud (EC2)[27].

Platform-as-a-Service (PaaS) comprises offerings that can be characterized as high-level application building blocks. Typically, they are marketed as services. By offering a selection of standard components such as authorization services, databases, and stream processing services, cloud providers establish a *platform* for their customers to build upon. The services that are offered as part of such a platform are almost always guaranteed to integrate well together. In addition, services constituting the platform offering are usually well-designed and implemented by software engineers with considerable experience. Hence, these purpose-built services tend to overshadow all possible in-house solutions that a small company might have built to solve its routine tasks such as storing the data reliably. Being part of the same platform, PaaS services are managed by the cloud services provider, which allows them to be finely tuned in terms of delivered performance and security to the underlying virtualization layers. Microsoft Azure is known to offer a wide selection of PaaS services that can be combined into the data processing pipelines[28]. Amazon Redshift data storage service by AWS might also be considered an example of PaaS model[29]. Hence, we see that it is not necessary for a cloud services provider to stick to a particular services model. Indeed, they usually diversify since the preferences of their customers differ and tend to change over time.

The main limitation of PaaS offerings is their limited flexibility. First, one has to settle with whatever the engineers of the PaaS provider deemed as necessary for the cloud customer in terms of functional requirements. This might be a good thing for standard workloads, but some workloads such as machine learning inference may require special treatment. Nowadays, this scenario is addressed with the purpose-built services[30]. Second pitfall is that by selecting PaaS offering of a particular provider, one is effectively bound to use their services thereafter since no integration or migration services are provided. Usually, PaaS is a go-to choice for companies where IT is a cost center (e.g. automotive[31] and aerospace[32] industries). Such companies are incentivized by the limited effort required to work with and maintain PaaS-based applications and a strong support from PaaS providers.

Function-as-a-Service (FaaS) hides the underlying infrastructure from the cloud user entirely – the cloud user is responsible for developing the so-called *functions* which are snippets of code or container images running as containers that are started and managed by the FaaS provider. The user of FaaS services does not have to worry neither about starting and managing the underlying containers and virtual machines nor about the scaling of the FaaS application following the changes in load. To cope with complex application

[27] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html

[28] https://azure.microsoft.com/en-us/overview/what-is-paas/

[29] https://aws.amazon.com/en/redshift/

[30] https://azure.microsoft.com/en-us/services/machine-learning/

[31] https://www.bmwgroup.com/en/company/bmw-group-news/artikel/bmw-und-microsoft.html

[32] https://www.informationweek.com/cloud/infrastructure-as-a-service/microsoft-azure-wins-boeings-cloud-business/d/d-id/1326316?

logic, FaaS providers offer their users to build function chains. Request arrives at the first function in the chain, and, when it was processed by it, the next request(s) is emitted to continue to the next function in the chain, and so on. The topology of an application composed of such functions is known to the cloud services provider thus allowing some resource management optimizations that are not possible otherwise. As an example, AWS has Lambda Functions as its FaaS offering[33]. There are also several open-source FaaS platforms with Apache OpenWhisk being the most notable industry-backed solution at the moment of writing[34].

Noting the overall vector of movement from virtual machines to containers and then to functions, one may arrive at the conclusion that the public cloud services providers aim at accumulating more responsibility for the success of their customers' applications. This way of thinking is supported by the fact that large public cloud services providers such as AWS offer their customers the services of their solutions architects whose responsibility is to ensure the customer success using the CSP's services[35]. On the other hand, this evolution of cloud computing service models follows the path of *simplification*. Indeed, with IaaS, the cloud user is responsible for managing and securing their VMs. PaaS allows one to compose an application from numerous tuned off-the-shelf building blocks (services) that are designed and developed by staff engineers of cloud providers and thus are more likely to provide higher performance, lots of functionality, high reliability and uncomplicated integration with other services of the same CSP. The amount of custom code is thus reduced for the users of PaaS model. Lastly, FaaS removes the pain of rightsizing containers with application services. With FaaS, the cloud user is able to fully focus on the application business logic. So far, FaaS demands the least effort from the cloud services user to jump-start their cloud journey and relieves them of most of the maintenance and application management burden.

The trend towards making the use of cloud effortless is likely to continue. It is expected that with the introduction of FaaS paradigm, cloud services providers will pay more attention to making certain complex aspects of cloud, such as security, simpler to their users. In addition, cloud offers will likely become more customized towards the needs of particular industries such as retail or education.

2.2 Dynamic Resource Provisioning

2.2.1 Cluster Manager: a Working Horse of Cloud Resources Provisioning

Unlike hypervisors that provision system resources of a single physical server, *cluster managers* provision virtualized resources (such as system resources and services) at the level of a cloud data center (DC) as a whole. Broadly speaking, a cluster manager is a distributed system that schedules cloud workloads on multiple nodes [24]. Among many tasks that a typical cluster manager is responsible for, it solves the task of timely provisioning of cloud services to multiple cloud users (aka *tenants*) while maximizing the resource utilization of DC. Each large cloud services provider such as Google and Microsoft has its own in-house cluster manager [24, 25]. Usually, such private cluster managers start as an effort to automate scheduling of B2C workloads (e.g. search, online retail) and later become adapted to lease spare system resources to other businesses or individuals thus becoming a cloud. Apache Mesos is an example of an open-source cluster manager[36].

[33] https://aws.amazon.com/lambda/
[34] https://openwhisk.apache.org/
[35] https://www.startupschool.org/posts/19593
[36] http://mesos.apache.org/

Cluster managers adopt an abstraction of a *task* which represents a minimal schedulable unit in the cluster; tasks may provide virtual machines [26]. *Jobs* comprise multiple tasks that have the same characteristics [24]. Conventionally, cluster managers follow the *master-worker* hierarchical organization which is proven to be more efficient than the flat peer-to-peer model in practice. For example, Google's Borg cluster manager has replicated **BorgMaster** responsible for a certain set of physical machines (*cell*), and **Borglets** (worker machines) that run on each physical server [24]. The jobs are submitted to BorgMaster, and are then scheduled for execution on physical servers by Borglets. Apache Mesos and Mercury cluster managers follow the same hierarchical organization [26, 27]. Similarly, Kubernetes, which emerged from Google's work on cluster managers such as Borg and Omega [28], has a hierarchical organization with a single master node and multiple worker nodes[37].

The main function of cluster manager is scheduling the jobs. Such jobs might both be internal to the cloud services provider (e.g. indexing the web pages) and external, i.e. submitted by the cloud users. Thus, CSPs have to balance their own workload with the external workload. This results in adoption of various workload prioritization schemes [24, 29]. These prioritization schemes aim to balance the quality of service delivered to the end user, e.g. returning the results of a search query in up to several tens of milliseconds at most, with maximizing the resource utilization of DCs. This usually forces the provider to compromise on the performance of the co-located workloads. Overall, the scheduling challenge in data centers is usually formulated as a *bin packing* problem. Having proven to be unsolvable in polynomial time, scheduling is approached using various heuristics and models that offer best solutions under some set of conditions.

Unlike internal batch analytical jobs that are also scheduled by cluster managers, cloud workloads demand a new level of *dynamism* for their provisioning. To ensure high quality of service, CSPs aim at minimizing the time required to provision their services and virtualized resources at *scale up* as well as at minimizing the time to decommission them at *scale down*. The cloud user is interested in minimizing the scale up time to ensure that the rapid surge in load will be timely served. Symmetrically, low scale down times allow the cloud user to cut the bill on paying for the services and virtual resources that are in the termination stage. This public cloud dynamism is very different when compared to what one would normally expect from the corporate data centers. Whereas in the former case one can get a new virtual machine up and running on the scale of seconds, maybe a few minutes at most, in the latter case it could take hours, maybe days, to get an approval for the resource allocation and set up the virtual machine correctly. This order of magnitude difference made up for another significant selling point of the cloud services, i.e. dynamic provisioning.

For a long time, cluster managers acted in a *reactive resource provisioning paradigm*. What this means is that the virtualized cluster resources are provided based on the internal or external user's request. For example, the internal user of the infrastructure may submit a job to the Google's Borg cluster manager to re-index all the web pages in a particular language. Borg will then allocate the resources for this job based on its requirements and the cluster resources available for it. In this paradigm, Borg will not try to anticipate such requests and will simply follow the orders given to it. Naturally, having no projections about the future load, it may take suboptimal decisions spreading the job across all the machines in the clusters equally to increase the performance and thus not leaving enough spare resources to accommodate a smaller job that might have otherwise been placed. Similarly, when accommodating the provisioning requests from the external users, the cluster manager may fail to notice a more optimal placement because it did not take into account the established patterns of the cloud users' behavior. This problem spun off a great deal of research in the *predictive resource management for clusters*.

Predictive resource management for cloud provisioning is an active area of research and engineering that builds on rich cluster-level resource utilization metrics that are collected by CSPs [29, 30]. Cloud services providers are interested in understanding how their infrastructure is utilized, what are the identifiable and

[37] `https://kubernetes.io/docs/concepts/overview/components/`

predictable *patterns* of resources usage, and how these patterns relate to the quality of service that the cloud users and the end users experience. Having control over the hardware and the whole virtualization and cluster management stack, CSPs are in the unique position to leverage the opportunities that the predictive resource management offers. They do so by exploring how the statistical and machine learning tools may capture the value of these data to meet multiple confronting goals such as offering the best quality of service to their users and maximizing the resource utilization. One of the most recent disclosed efforts in this venue is **Autopilot**, an advanced cluster manager that uses the multi-armed bandit prediction model to manage the workloads at Google scale [18]. Similar efforts are undertaken by Microsoft in making their own predictive cluster manager Resource Central [30]. The major challenge that arises on this path is to process the vast amount of machine and application data in a meaningful way and in feasible time. Even having the most powerful infrastructure will not help in advancing on this frontier since the meaningfulness of the results is something that only a seasoned expert can determine. To accelerate the advances on this direction, CSPs publish subsets of the resource utilization data that they collect [24, 29, 30, 31].

Dynamic provisioning of cloud resources does not stop at the level of cluster managers. In the end, most of the existing cloud services models, such as IaaS and PaaS, significantly limit the extent, to which CSPs may observe how the applications of their customers are organized, what internal logic do they implement, and what quality of service does the end user receive. Having limited opportunity to set the foot on this ground (e.g. FaaS model), cloud services providers opt for offering customizable dynamic provisioning services for the virtual resources that their customers use. The cloud user can set up such a service to automatically provision the virtual machines in response to the measured response time that the end user gets. If this time goes up, new instances start automatically to distribute the workload and thus drive the response time down. On the other hand, if the load drops, e.g. in the night hours, then the unneeded instances get terminated to reduce the cost. This 'userland' dynamic resource provisioning is often referred to as *autoscaling*.

2.2.2 Autoscaling: Terms and Mechanics

Autoscaling is a collective name of the software services and mechanisms to automatically, i.e. with the human interference limited to the initial configuration, adjust any type of the computer system capacity to meet the preset objectives. We stick to the broad term *system capacity* without specifying it, since there exist numerous adjustment targets for autoscaling, including node resources allocated for the container (CPU shares, main memory, ports, hardware threads, CPU sockets), nodes/virtual machines count, containers count, capacity of the service queues, application-specific characteristics such as quota on an open file descriptors per thread. Almost always, the target autoscaling metrics such as CPU utilization cannot be directly manipulated by an autoscaler. A dynamic change in the provisioned capacity can move these metrics towards the desired value, e.g. an addition of a VM to the cluster may drive the overall utilization down. Another option is that the load change will push the target metric in the desired direction. Naturally, though, the load cannot be directly manipulated as well.

All autoscaling mechanisms boil down to a feedback loop. First, the metrics describing the state of the managed entity (e.g. application) are collected. These metrics should relate to the objective that one wants to achieve with autoscaling. For instance, if the target is specified for the CPU utilization, then it makes sense to collect this metric directly. Next, based on the collected metrics and on the state of the managed entity (e.g. the number of replicated services in the application), a desired adjustment of the capacity is determined. Lastly, an attempt is made to apply this adjustment. This attempt may succeed, fail, or partially succeed. An adjustment can be made differently, e.g. incrementally, node-by-node, or in a single step. The reduction in the capacity may be delayed to allow the graceful resources deallocation by blocking the to-be-terminated nodes from scheduling new requests and waiting till the processing of already scheduled

requests is finished. When the whole round of the described operations concludes, it starts anew after some time. An unexpected event, e.g. an unprecedented surge in the traffic, can also trigger autoscaling.

The simplest autoscaling configuration includes only a single target, say, that the average CPU utilization of a cluster should stay below 80%. On crossing this mark from below, additional nodes will be scheduled proportionally to the CPU utilization overshoot. Since some metrics may oscillate wildly, smoothing and filtering are usually used, e.g. averaging over larger intervals of time (1, 5, 10 minutes) and specifying the duration of the violation that triggers an autoscaling action. An autoscaling solution falls into one of the following two categories.

CSP Managed Autoscaling comprises autoscaling services offered by a cloud services provider to remove the hurdle of managing the cloud resources that an application owner uses. This could be the capacity of VM clusters or of the functions (Function-as-a-Service). An advantage of autoscaling solutions in this category is that they are provided free of charge, i.e. not billed separately, and have the fastest and the most fine-grained access to the platform-level metrics. On the downside, there is a limited opportunity to manage these solutions. For instance, the logic of the desired resource computation is completely hidden from the application owner and he or she cannot change it.

Independent Autoscaling is usually implemented as a part of an external application container orchestration and platform management tool. For instance, Kubernetes, Nomad, and Marathon implement autoscaling functionality as part of their application and cluster management offers. Since most of such tools are open source and extendable, the autoscaling logic may be adapted to the needs of the application owner. However, when used for scaling the clusters of virtual machines, such tools may suffer from the inability to get the monitoring data timely. Either the application owner has to configure the monitoring infrastructure to collect the desired metrics and shoulder the expenses of running it or she has to opt for the paid monitoring offers from CSPs.

2.2.3 Autoscaling by Cloud Services Providers

2.2.3.1 AWS

Having captured the largest share of the world cloud computing market[38], AWS, a subsidiary of the online retailer Amazon, offers the most comprehensive autoscaling package. We are obliged to AWS for coining the term "autoscaling" to denote the elastic management of VM clusters and applications.

AWS Auto Scaling is a set of autoscaling services offered by AWS. Originally, AWS Auto Scaling was limited to providing autoscaling only to the VM clusters, collectively referred to as Amazon Elastic Compute Cloud (EC2)[39] in the AWS terminology. Now, AWS Auto Scaling also supports other AWS services such as Elastic Container Service (ECS), Spot Fleets, DynamoDB, Aurora relational database service, a fully managed non-persistent application and desktop streaming service AppStream 2.0, and Elastic MapReduce (EMR)[40]. AWS Lambda functions service is also known to support autoscaling with Application Auto Scaling and its API[41].

AWS Auto Scaling offers its users both the coarse-grained configuration opportunity via the pre-defined *strategies* and the custom autoscaling configuration option. When opting for the coarse-grained configuration, the user is allowed to choose among the optimization goals that include availability, cost, and balanced

[38] https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/
[39] https://aws.amazon.com/ec2/
[40] https://aws.amazon.com/blogs/aws/aws-auto-scaling-unified-scaling-for-your-cloud-applications/
[41] https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html

availability/cost. The custom autoscaling configuration demands to specify metrics to track and their target values. The selection of such metrics is rather flexible and allows the user to employ the application-specific metrics such as read and write capacity utilization for the database services.

Predictive scaling extends AWS Auto Scaling strategies[42] – the user can set the flag to enable it when choosing the strategy. Predictive scaling can be fine-tuned by setting its mode, e.g. whether to use the forecasts or just to log them as in dry run, by setting the behavior that is invoked when reaching the maximum capacity. It is also allowed to set the desired timing for various components of the process such as the forecast period and the forecast frequency.

Although the core offer of the AWS Auto Scaling service is horizontal autoscaling, vertical autoscaling is also supported[43]. This feature is included in AWS Ops Automator, a solution enabling the user to automatically manage her cloud resources. A time-based or event-based trigger has to be specified – such a trigger determines when instances are scaled. The vertical autoscaling can either change the size of the existing instances or replace them with new, resized ones. In both cases, a restart is required.

AWS Elastic Kubernetes Service (Amazon EKS) is a managed Kubernetes service from AWS[44]. It wraps Kubernetes containers orchestration platform that many companies are already familiar with. AWS extends Kubernetes offer with their in-house developments such as monitoring and AWS Auto Scale. As with any CSP-based Kubernetes offering, it supports all the Kubernetes autoscalers.

Elastic Load Balancing by AWS automatically distributes incoming traffic across multiple targets such as EC2 instances, containers, IP addresses, and AWS Lambda functions[45]. Three broad categories of load balancers are supported at the time of writing. *Application load balancer* operates at the level of requests, such as HTTP requests. *Network load balancer* aims to balance the TCP/UDP and TLS traffic. *Classic load balancer* operates at both the connection and requests layers and provides its services to the EC2 clusters. Elastic Load Balancing service complements AWS Auto Scaling since the distribution of traffic should adapt accordingly to the changed cloud capacity.

2.2.3.2 Microsoft Azure

Micorsoft Azure holds the second place by its market cap on the cloud market[46]. Azure tries to follow in AWS steps by offering versatile services to its customers, mostly large companies.

Azure Autoscale is the autoscaling service by Azure that targets their *Virtual Machines Scale Sets*, *Cloud Services*, *App Service - Web Apps*, and *API Management* services[47]. It is not clearly separated in a standalone service like in AWS cloud – instead, the option to enable the autoscaling is attached to various Azure resources. It can be checked, and the corresponding autoscaling rules can be configured.

The core concept of Azure Autoscale is flexibility. When autoscaling is enabled for a particular Azure resource, the user has to explicitly specify the rule- or schedule-based autoscaling rules, in contrast to the high-level scaling strategies offered by AWS Auto Scaling.

In the rule-based autoscaling, the user can specify multiple scaling rules of a simple threshold-based format. These rules can be configured by changing the time aggregation format (e.g. averaging), the metric (e.g. CPU utilization), the operator (e.g. greater than or less than), the threshold (e.g. 80%), and the time interval

[42] https://aws.amazon.com/blogs/aws/new-predictive-scaling-for-ec2-powered-by-machine-learning/
[43] https://aws.amazon.com/blogs/architecture/aws-ops-automator-v2-features-vertical-scaling-preview/
[44] https://aws.amazon.com/eks/
[45] https://aws.amazon.com/elasticloadbalancing/
[46] https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/
[47] https://docs.microsoft.com/en-us/azure/azure-monitor/platform/autoscale-get-started

during which the rules should hold in order to trigger the scaling action. The rule contains the action part as well – it specifies how the target should be scaled and by which amount (e.g. the instances count should be increased by 5).

The schedule-based autoscaling offers the user an opportunity to utilize the knowledge of load periodicity. In particular, a scaling schedule can be specified for particular days of the week and for particular intervals of time. If a load pattern such as diurnal pattern is known to occur, it can be captured in the schedule-based autoscaling. In addition, one can specify different autoscaling behavior for a range of dates.

Like AWS, Azure offers its own managed Kubernetes service – **Azure Kubernetes Service** (AKS). In addition to the core Kubernetes autoscaling mechanisms, Azure provides own Kubernetes-based Event Driven Autoscaling (KEDA)[48]. KEDA serves as *Metrics Server* in Kubernetes and allows to define autoscaling rules using custom resource definitions (CRD).

2.2.3.3 Google Cloud

Google Cloud closes the top three cloud providers by the market cap[49]. Thriving on the Google's vast experience in managing the warehouse-scale data centers, Google Cloud provides both the VM clusters-level and the application-level offers.

Google Cloud offers autoscaling service for their Managed Instances Groups (MIG)[50]. Essentially, it has the same purpose as the AWS Auto Scaling service for EC2 instances. The autoscaling service by Google Cloud supports three autoscaling policies. The first one is based on observing the average CPU utilization of the VM cluster. The second aims at balancing the HTTP serving capacity, based on either utilization or the requests per second. The third one makes use of cloud monitoring metrics.

Google's autoscaling service tries to reduce the response latency due to abrupt scale down events by allowing to set the following properties of the autoscaler: a) the *maximum allowed reduction* that limits by how much MIG can be scaled down so that the workload spike can still be served; b) the *trailing time window* is an interval of time within which the autoscaler monitors the peak workload; the autoscaler will not resize below the *maximum allowed reduction* subtracted from the peak size observed in this time interval.

Google's autoscaling service for MIG works as follows. It constantly monitors the instance group and sets its recommended size in terms of VMs count such that it would be able to serve the peak load over the last ten minutes (stabilization period)[51]. MIG can either use the recommendation as-is or use the recommendation only if it supposes the scale-out, or ignore the recommendation entirely. An autoscaler is conservative – it rounds up or down when it determines how many instances to add or remove. This behaviour is supposed to prevent the autoscaler from removing too many resources and from adding an insufficient number thereof.

Google Kubernetes Engine (GKE) is a service that provides the user with a Google-managed Kubernetes environment for a containerized application on Google's infrastructure[52]. Since this offer is based on Kubernetes, it supports all the autoscaling options provided by Kubernetes which are discussed in the next subsection.

[48] https://github.com/kedacore/keda
[49] https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/
[50] https://cloud.google.com/compute/docs/autoscaler
[51] https://cloud.google.com/compute/docs/autoscaler/understanding-autoscaler-decisions
[52] https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview

2.2.4 Autoscaling in Kubernetes

Kubernetes offers three types of autoscalers managing different *capacity representations*. Horizontal Pod Autoscaler focuses on changing the number of pods replicas. Vertical Pod Autoscaler changes pod resource requests to make it utilize more or less of the node that it is deployed on. Cluster Autoscaler changes the number of nodes if the managed application is deployed in a cloud. The technical details for each autoscaler of Kubernetes v1.18 follow.

Horizontal Pod Autoscaler (HPA) automaticlaly adjusts the count of the pod replicas that are deployed in a *ReplicaSet*[53]. HPA is one of the core Kubernetes controllers. The core component of HPA is *Replica Calculator*. It implements the pod replicas count calculation logic. Calculator is equipped with the *Metrics Client* to query the metrics relevant for its computations. It can also be adjusted with help of additional settings like tolerance that is used to stabilize the scaling. HPA runs in an infinite loop reconciling the state of the associated *ReplicaSet* periodically. On each round, it starts by extracting the current replicas count and the resources occupied by them. Next, HPA determines whether it needs to rescale controlled pod replicas, and, if yes, by how many (*desired replicas*) and with what reason (*rescale reason*). The rescaling is trivial if either the current amount of replicas is set to 0 or the current number of replicas is higher/lower than the specified max/min replicas count. In the former case HPA simply sets the desired count to 1, whereas in the latter the desired count is set to appropriate limit. In the nontrivial case, HPA has to use a more advanced logic of *Replica Calculator*.

Internally, HPA recognizes several types of metrics to use for the replicas calculation: a metric for an arbitrary object *ObjectMetricSourceType*, a metric for pods *PodsMetricSourceType*, a metric for a resource *ResourceMetricSourceType*, and an external metric *ExternalMetricSourceType*. These metrics are computed differently, which impacts how the corresponding replicas count is calculated. *ObjectMetricSourceType* describes an arbitrary Kubernetes object, e.g. hits-per-second for *Ingress* object. *PodsMetricSourceType* differs in that it represents metrics for pods and is averaged before being compared to the target value. *ResourceMetricSourceType* is specified in pods requests and limits for CPU and memory. *ExternalMetricSourceType* is introduced to allow the scaling based on the metrics for objects running outside of the Kubernetes cluster, e.g. the queue length in a cloud messaging service or the queries per second (qps) count for a load balancer.

Before calculating the desired replicas count, *Replica Calculator* collects current values for the relevant metrics via *Metrics Client*. *Metrics Client* abstracts the low-level details of the metrics retrieval which is done either via the legacy Heapster client or the REST metrics client. Its unified interface supports operations to get metric values for every metric type considered above. As an example, let us consider the calculation of pod replica count using *ObjectMetricSourceType*.

The calculation starts with computing the usage ration (in case of raw metric, i.e. not averaged):

$$usageRatio = \frac{utilization}{targetUtilization} \tag{2.1}$$

If the computed usage is within the tolerance range of the absolute usage, viz, 1.0, then the current replicas count is used as a desired count. The logic is that the utilization is close to 100%, but not higher, hence the metric is utilized. However, if the usage ratio is not within the tolerance range of the absolute usage, then the count of the running pods is determined and used to compute the desired count as follows:

$$desiredCount = \lceil usageRatio \cdot readyPodCount \rceil \tag{2.2}$$

[53] https://github.com/kubernetes/kubernetes/tree/master/pkg/controller/podautoscaler

In case of an averaged metric, *Replica Calculator* first computes the usage ratio as follows:

$$usageRatio = \frac{utilization}{targetAverageUtilization \cdot replicaCount} \tag{2.3}$$

If the discrepancy between the usage ratio and the absolute usage is larger than the tolerance, then the current replicas count (used as desired later on) is computed as follows:

$$currentReplicas = \lceil \frac{utilization}{targetAverageUtilization} \rceil \tag{2.4}$$

Lastly, the utilization is averaged over the current replicas count:

$$utilization = \lceil \frac{utilization}{currentReplicas} \rceil \tag{2.5}$$

Vertical Pod Autoscaler (VPA) sets the resource requests automatically based on usage and thus allowing the proper scheduling onto nodes so that appropriate resource amount is available for each pod. VPA employs the most elaborate logic among all the Kubernetes autoscalers. At the moment of writing, VPA was not included in the mainstream Kubernetes[54].

VPA's specification includes a target controller that manages a set of pods, e.g. *Deployment* or *StatefulSet*, the update policy that describes how the changes in resources are applied to the pods (*UpdatePolicy*), and the resource policy that computes the recommended resources for pods (*ResourcePolicy*). The update policy and the resource policy define the behaviour of VPA.

At the moment of writing, the behaviour of the update policy is determined only by the update mode that is limited to the following options: *Off* that does not allow VPA to set the pod resources, although the recommendations about the resources are still produced (valuable for the dry run); *Initial* that allows VPA to assign the resources only at the start of the pod and does not allow to perform the changes later on; *Recreate* that allows VPA to assign the resources on pod creation time and to update them during the lifetime of the pod by deleting and recreating it; *Auto* that allows VPA to assign the resources on pod creation time and to update them during the lifetime of the pod by using any available method (currently only *Recreate* is available).

The resource recomputation is governed by the *Resource Policy*. At the heart of *Resource Policy* are the policies for individual containers (*ContainerResourcePolicy*) constituting the pod with the requirement of not more than 1 policy per container. Each container resource policy can be assigned to a particular container in a pod by specifying the name of the container. If the policy is not specified for a particular container name, then the *Default Container Resource Policy* is used. Each *Container Resource Policy* is composed of the following parts: *ContainerName* that can either specify a container or be a wildcard to allow its application to all the containers in the pod; *ContainerScalingMode* that currently specifies whether the vertical autoscaling is applied to the container; *MinAllowed* – the minimal resource requirements to ensure that the container will run; *MaxAllowed* – the maximal possible resource allocation for the given container, e.g. to avoid the oversubscription for the limited resources.

VPA consists of four core components: *Recommender*, *Updater*, *Admission Controller*, and *Quality Controller*. To a large extent, *Recommender* defines the scaling behavior of VPA.

[54] https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler

Recommender computes the recommended resource requests for pods based on the current and the historical resources usage. Each pod resource recommender consists of three resource estimators – the *target estimator*, the *lower bound estimator*, and the *upper bound estimator*. These estimators allow the *Recommender* to derive the optimal, the minimal and the maximal amount of resources per pod. These estimators are initialized with specific percentiles for both the CPU and the memory utilization: target – 0.9 percentile, lower bound – 0.5 percentile, and upper bound – 0.95 percentile. The logic behind the usage of various estimators for various percentiles is to account for different resource allocation scenarios. The uncertainties in the measurements are taken into account by adding a safety margin of 15% (SMF = 0.15) on top of the resource estimate. Thus, the margin-adjusted resource estimate for the percentile X is:

$$\hat{R} = \hat{R}_{X\%} \cdot (1 + SMF) \tag{2.6}$$

An additional adjustment is applied to the lower- and upper-bound estimators. Both are multiplied by the *confidence factor* with different parameter values. This factor is computed as follows:

$$k = \left(1 + \frac{m}{d}\right)^{e} \tag{2.7}$$

where d stands for the metric history length in days, m and e are the estimator-specific parameters. With this factor, the final estimate of $\hat{R}$ for lower and upper boundaries is scaled by the corresponding k. These parameterized estimators with the safety margin of 15% are computed as follows:

$$\hat{R}_{target} = 1.15 \cdot \hat{R}_{90\%} \tag{2.8}$$

$$\hat{R}_{lowerbound} = 1.15 \cdot \hat{R}_{50\%} \cdot \left(1 + \frac{0.001}{d}\right)^{-2} \tag{2.9}$$

$$\hat{R}_{upperbound} = 1.15 \cdot \hat{R}_{95\%} \cdot \left(1 + \frac{1}{d}\right) \tag{2.10}$$

As any other Kubernetes autoscaler, VPA runs in an infinite loop. At each iteration, it executes the *Recommender* algorithm and updates the health status. The algorithm starts with the initialization of the timer to monitor the *Recommender*'s execution time and the acquisition of the context. Next, *Recommender* actualizes the state of the associated pods group and proceeds to the calculations.

The resource calculation is performed for each container. At first, *Recommender* sets the bare minimum amount of resources *minResources* for every container in the pod to be the same:

$$minResources_{CPU} = \frac{podMinCPUMillicores}{containersInPod} \tag{2.11}$$

$$minResources_{memory} = \frac{podMinMemoryMb}{containersInPod} \tag{2.12}$$

The defaults for the min values of the resources are: 250Mb for *podMinMemoryMb*, 25 mCPUs for *podMinCPUMillicores*. According to the formulas, the bare minimum of resources is equally distributed among all the containers. The computed *minResources* are used as an input to the resource estimator*WithMinResources* attached at the end of the already initialized chains of resource estimators discussed earlier (target, lower-

and upper bound: *PercentileEstimator* → *MarginEstimator* → *ConfidenceMultiplier*). With *minResources* being denoted as r, the equations 2.8, 2.9, 2.10 can be rewritten as follows:

$$\hat{R}_{target} = max\left(r, 1.15 \cdot \hat{R}_{90\%}\right) \tag{2.13}$$

$$\hat{R}_{lowerbound} = max\left(r, 1.15 \cdot \hat{R}_{50\%} \cdot \left(1 + \frac{0.001}{d}\right)^{-2}\right) \tag{2.14}$$

$$\hat{R}_{upperbound} = max\left(r, 1.15 \cdot \hat{R}_{95\%} \cdot \left(1 + \frac{1}{d}\right)\right) \tag{2.15}$$

By applying this additional safeguard estimator, *Recommender* ensures that the amount of resources allocated by VPA to a container won't fall lower than the minimum.

Cluster Autoscaler (CA) automatically adjusts the size of the Kubernetes cluster when either there are pods that failed to run due to insufficient resources or there are nodes that have been underutilized and their pods can be placed on other existing nodes[55]. CA runs on the Kubernetes master node. At the time of writing, CA supports all the major cloud services providers, viz, AWS, Google (GCE and GKE), Microsoft (Azure), Alibaba, DigitalOcean, and OpenStack Magnum.

Major functionality of CA is implemented by the following components:

- **cloud provider interfaces** that wrap cloud provider-specific APIs and the models of nodes;
- **autoscaling core** that implements the main cluster autoscaling logic, e.g. checking the new cluster size against the user-defined limits;
- **estimator** that computes the required nodes count to fit the given pods count according to the bin packing algorithm;
- **expander** that selects a node group to deploy the pods[56] according to one of the supported strategies: a random node group, a node group that fits the most pods, a node group that leaves the least fraction of CPU and memory, the most cost-effective node group, a node group based on a user-configured priorities.

CA conducts autoscaling as a scale-up followed by a scale-down during the same round. CA treats these actions a bit differently.

Scale up. At the moment of writing, Kubernetes supports only the *bin packing estimator* to allocate nodes during the scale-up. This estimator treats the task of finding the right amount of nodes to place pods as a bin packing problem [32]. Each node represents a *bin* characterized by some *volume*, or capacity in terms of resources, such as CPU, memory, ports. A pod represents an *item* taking some of this volume when placed in a bin (volume also in terms of resources). The task is to allocate items (pods) into bins (nodes) such that the number of bins (nodes) is minimized. CA employs a variation of the First Fit Decreasing (FFD) algorithm to solve the bin packing problem in $O(nlog(n))$ time [33].

To solve the bin packing problem, CA first calculates the *score* for every pod. The score is calculated as the sum of two fractions. The first one is the fraction of the CPU requested by the pod in relation to the CPU capacity given in the node template. The second one is the fraction of the memory requested by pod in relation to the memory capacity in the node template. The formula is as follows:

[55] https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler

[56] In Kubernetes, a node group corresponds to the virtual machine type (also flavor) for the cloud services provider.

$$score = \frac{CPUreq}{CPUnode} + \frac{MEMreq}{MEMnode} \tag{2.16}$$

Later, these scores are used to sort the pods in the decreasing order, i.e. higher score means the higher allocation priority. CA then loops through the sorted pods one by one and tries to place each on the available nodes. If the node cannot accommodate a new pod due to the insufficient resources, then the next node is checked, and so on until CA runs out of nodes. In the latter case, a new node is started to accommodate the pod. As a result of executing this algorithm, CA outputs the number of nodes to start.

Scale down. The asymmetry between the scale up and the scale down of the cluster is based upon risks attributed to both actions. The risk associated with the untimely scale up is not so severe since the cost is delaying the service of the incoming requests. An untimely scale down can result in abrupt cancellation of the requests being processed.

Before conducting the scale down, CA computes the resource limits that are left for the scale down. In contrast to the scale up, the intuition here is as follows. The application owner may specify the limit on the minimal amount of resources in the node group that is necessary to support the functionality of the application for the minimal anticipated load. This limit should be respected by CA, which means that it cannot remove more nodes than the amount specified by the scale down resource budget, i.e. $currentResource - minResource$ both for cores and the memory.

Next, CA considers the unneeded nodes for the scale down, i.e. the nodes whose pods can be scheduled somewhere else. The unneeded node is skipped for the scale down if it is marked with a no-scale-down annotation. Such mechanism is useful to pertain core nodes holding some unique and important resource. If the node was unneeded not long enough (*scaleDownUnneededTime*), then it is also not removed during the current round of autoscaling algorithm. Unready nodes may also be skipped for deletion if they were unready not long enough (*scaleDownUnreadyTime*).

After checking all this node removal conditions, CA computes the scale down *resource delta* and checks whether it is withing the scale down resource budget defined earlier for each resource type. If it is within the limits, then the corresponding nodes are added to the scale down candidates list.

CA starts by trying to delete the empty nodes in bulk. Such nodes do not have any pods running on them. If it turns out that there are no empty nodes, CA will try to delete the nodes that host the smallest count of pods. This results in pods recreation on other nodes while freeing the resources. Following, CA tries to find other nodes to remove by checking the scale down candidate list. During such check, CA investigates whether the candidate nodes have blocking pods on them, e.g. system pods or pods that did not yet run out of their disruption budget. If blocking pods are detected, then the nodes hosting them are excluded from the scale down – thus, the resulting scale down list may shrink. At the same time, CA tries to find a new place for pods to be deleted by draining the corresponding nodes. It starts with checking pod hints. If there is no alternative node provided in the pod's hints, then CA tries to find any node that matches the requirements of the pod. Lastly, the pods are moved and the nodes from the pruned candidates list are deleted from the cluster in the background.

Important to note that CA considers heterogeneous nodes, i.e. it takes into account whether the node has GPU or not when considering the cluster scaling.

2.2.5 Categorization of Autoscaling Approaches

To simplify the discussion of the related work on autoscaling in Chapter 4, in the following paragraphs we introduce several categories that the majority of the autoscaling approaches belong to.

Machine Learning, Forecasting and Regression based approaches. This broad category of autoscaling approaches leverages available metrics data to build the models of processes directly or indirectly related to autoscaling. Following, these models are used to predict some quantitative characteristic relevant for autoscaling, such as future load or resource utilization for particular kind of resource. If there are some sort of regularities in the collected metrics (load seasonality, relation of response time on the allocated resources and load, etc.), an appropriately tuned model fed with enough data will be able to grasp them. These regularities expressed as *regressions* allow to estimate the relevant parameters and take the optimal scaling decisions given the predicted values. The downside of autoscaling approaches in this category is that they require substantial domain knowledge as well as expertise in statistical methods, stable and pervasive monitoring infrastructure, and sometimes a considerable amount of compute and memory resources to fit models (e.g. deep neural networks). The amount of collected observations and their versatility determine the accuracy of approaches in this category to a large extent.

Optimal control based approaches. Autoscaling approaches in this category are deeply rooted in the decades old mathematical discipline of optimal control. Optimal control attempts to continuously optimize an *objective function* that quantifies the cumulative state of the dynamic system. In case of autoscaling, one may optimize for the response time or the throughput of the application. Resource utilization and cost of the cloud services may also be used as such objective functions. An objective function may also combine multiple autoscaling quality characteristics, e.g. response time and CPU utilization. Optimal control is agnostic of a particular objective function. The key idea is to discover some control law that allows to achieve the desired *optimality criterion*. The control law is expressed as a set of differential equations that map the variables (either observed or controlled) onto the value of objective function. By optimizing the cost function for the given set of differential equations describing the autoscaled system, one can find the values of control variables (e.g. the count of virtual machines or the amount of allocated resources of different kinds) that yield the desired value for the objective function, e.g. response time staying below a certaing threshold (aka set point). Most of the optimal control systems of equations for autoscaling have a *feedback form* due to scaling actions impacting not only the objective function itself but also other parameters such as resource utilization that are usually used as other inputs to the system of equations.

Pure optimization based approaches. Albeit similar to the previous category, autoscaling approaches that fall into the pure optimization domain drop the feedback loop from the consideration. Autoscaling is thus merely viewed as a constrained optimization problem replicated over discrete time scale. Optimization-based approaches are usually complemented by performance modeling approaches that derive coefficients for the objective function. The absence of the feedback loop is sometimes compensated for by infrequent updates to the performance models characterizing the relation between the control variables and the cost functional (e.g. between the count of virtual machines that the application runs on and the response time).

Algorithmic approaches. Autoscaling approaches in this category are deterministic. Although they still use the monitoring data to take the scaling decisions, the internal structure is rather rigid in a sense of a model of the problem domain. For example, a conventional ratio-based reactive horizontal autoscaling approach discussed in the prior section on Kubernetes assumes proportionality between the number of replicas and the resource utilization. This assumption might generally hold, but it may fall short of capturing accurate relations between the resource allocation and the actual resource utilization for more complex application topologies or unusual processing logic. Despite being rigid, algorithmic approaches have an advantage of being fast and easy to reason about. The downside is, however, that they often require the application owner to set a number of parameters such as resource utilization thresholds that simply do not have a general default value and that should be tuned for each particular application if not service inside the application.

3

Motivation

3.1 Limitations of Production-grade Multilayered Autoscaling

3.1.1 Approach to the Performance Evaluation of Autoscaling

To uncover the limitations of the existing autoscaling solutions, an approach devised in [34] can be applied. The approach centers at the concept of *autoscaling latency* which represents the interval of time between the moment when the desired state of the cluster/application was calculated and the moment when the current state matched it, possibly, with some discrepancies. To assess the autoscaling quality from the user's point of view, one can compute the fraction of autoscaling latency during which the service level requirements were violated; values close to 1.0 point at low autoscaling performance. *Application response time* and the *requests failure rate* can be used as service level indicators to assess the autoscaling performance.

Determining the autoscaling latency at just a single layer of virtualization (virtual machines or containers) is straightforward. In contrast, when the autoscaling is enabled at multiple layers (as in case of HPA/CA Kubernetes autoscalers combination), the autoscaling intervals on separate layers may stay apart even though they result from the same change in the load. The approach to handle such a case was introduced in [34] under the alias of *multilayered autoscaling performance evaluation*. In short, the paper proposes to solve this challenge using the notion of the *time locality*. The scaling actions across several virtualization layers are considered to be associated if two conditions hold. First, the autoscaling event on the previous layer and on the next one should have the same direction of scaling (scale up or scale down). Second, scaling on the dependent layer follows the scaling action on the layer where the scaling first occurs. This is a general approach to assess autoscaling for multilayered cloud deployments, e.g. Kubernetes offers coordinated autoscaling with the HPA/CA combination in which CA clearly follows HPA, and the application scaling is considered to be the driver to the cluster scaling. We implemented this autoscaling performance evaluation approach in an autoscaling performance measurement tool ScaleX [35].

3.1.2 Discovering Limitations of Production Autoscaling Solutions

Autoscalers face various delays from the cluster managers, hypervisors, operating systems that run on nodes, non-negligible network latencies, and the application itself. Even the autoscaler may become a source of

latency if it does not run continuously or the autoscaling logic is heavy-weight. It is natural to assume that the observed behaviour, i.e. service level degradation during autoscaling, will only be emphasized in vivo. To prove this assumption, we conducted the same autoscaling performance evaluation for three major cloud services providers, viz, AWS, Microsoft (Azure), and Google (Google Cloud). In this experiment, two autoscaling solutions were used. First, the VM cluster was autoscaled by using the native offerings of the corresponding CSP. Second, Kubernetes HPA was responsible for scaling the deployed application.

An experiment covers 4 tests, each with one of the following load patterns: linearly increasing with drop, linearly increasing with saturation and drop, random, triangular (linear increase followed by the linear decrease till 0). The total time for each test is 20 minutes with the request timeout set to 6.5 seconds. The number of the simulated concurrent clients for each load pattern was 50. For each load pattern except for random, the start value of the request rate was 1 request per second (rps), whereas the step change is set to 3 rps. The random load pattern oscillates around 50 rps. The test application computes the sum of prime numbers between 1 and 1000000 on each request received. The choice of such an application for the autoscaling evaluation has two reasons: the application is compute-intensive and thus is fast to push the autoscaling decision metric (CPU utilization) to the threshold set in autoscaling configurations; the application represents the "lower bound" for a variety of more complex applications in which the autoscaling manifests in even larger departure from the service level objectives.

The VM configurations for experiments are presented in Table 3.1. The OS image was Ubuntu 16.04 LTS. The max size of the VMs cluster was limited by 3 nodes. Kubernetes was configured with the min pods count of 1, and the max count of 10. Scale-up was invoked when the CPU utilization of the cluster was at 20% both for CSPs autoscaling services and Kubernetes HPA.

Table 3.1.: Virtual machine configurations.

Cloud services provider	Instance type	Memory	vCPUs
Google Compute Engine (Google Cloud)	-	2 GB	1 vCPU
AWS	t2.small	2 GB	1 vCPU
Micorsoft Azure	A1_V2 Standard	2 GB	1 vCPU

Following, the autoscaling performance evaluation results for the conducted experiments are discussed.

AWS Auto Scaling & Kubernetes: The scale up by the native AWS autoscaling service lags the scale-up action by Kubernetes HPA (rows **B** and **C** in Figure 3.1). The lack of coordination leads to deploying the new pods on the old VM instances with the newly added VMs hosting only a single pod. Such a disproportion results in a latency increase as shown in row **D** of Figure 3.1. For the load patterns 2-4, max and mean request latencies top 2.5 seconds directly before the scale up takes place and during the scaling action. For load patterns 1 (linearly increasing) and 3 (random), Kubernetes HPA managed to keep the request latencies and the failures count low due to the gradual incline in the load in the former case and relatively low mean request count during the whole experiment in the latter case (row **A**). In case of load pattern 1, Kubernetes HPA gave enough time to the AWS Auto Scaling service to scale up the cluster size thus avoiding the decrease in service level when the pod replicas run out of the capacity of a single node. The low efficiency of the autoscalers for the load patterns 2 and 4 is due to the steep increase in the requests rate. It is clearly visible that part of this increase occurred when the current state of the cluster did not yet catch up to the desired state. Thus, the autoscaling latency of 5 – 30 seconds was not enough accommodate the considered load generation rates, leading to the surge in tail latency during the scale up.

Microsoft Azure Autoscale & Kubernetes. Microsoft Azure Autoscale demonstrates the slowest scale up and scale down among all the evaluated autoscaling services as seen in row **C** of Figure 3.2. Plots in rows **D** and **E** allow to conclude that the performance of a single Azure VM instance is higher than that of AWS

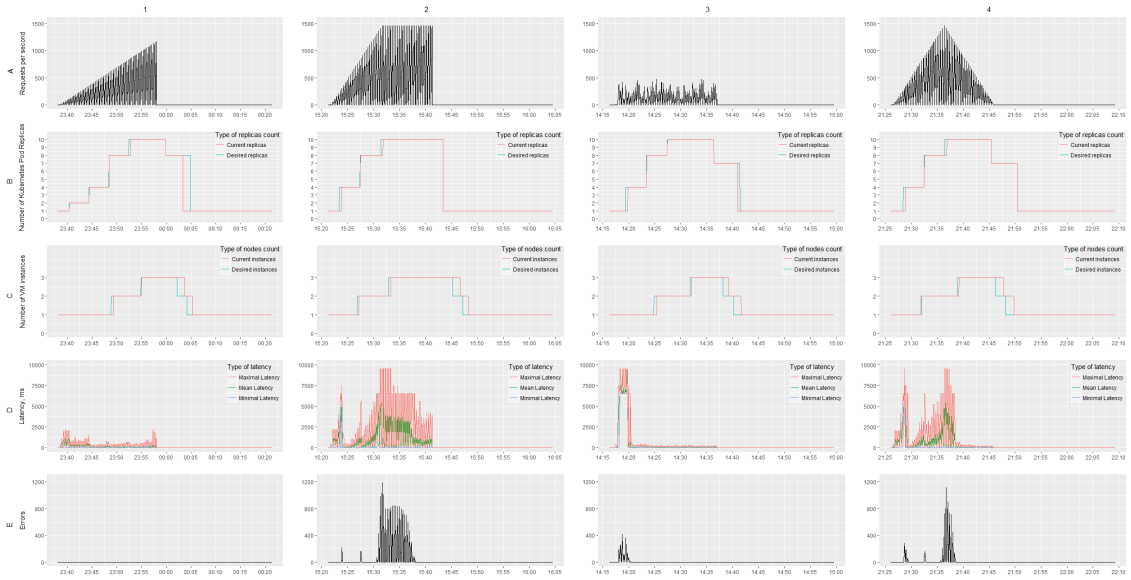


Figure 3.1.: Autoscaling performance evaluation results for the AWS Auto Scaling & Kubernetes pair [11].

and GCE since with the late scale up Azure is still able to keep the performance on par with other tested configurations. Similarly to the graphs acquired in the AWS Auto Scaling & Kubernetes experiment, row **D** of Figure 3.2 demonstrates that Kubernetes HPA allowed to dampen the arriving load in case of the gradual increase (pattern 1) and constant low mean (pattern 3), whereas the surge in the latencies and failures rate is observed in other two patterns characterized by the steep increase in the load. The scaling of the cluster is not able to keep up with such increase. Indeed, the most severe service level decrease (rows **D** and **E**) occurs when the decision to scale the cluster size up is taken, but not yet implemented (row **C**). Since in tests for both patterns 2 and 4, the scale up of the cluster size finishes too late, Kubernetes HPA places all its pods on the first VM. This leaves the fresh VMs useless – the service level does not improve upon their start. These load patterns not only increased the tail latency in the Microsoft Azure Autoscale & Kubernetes experiment, but dramatically reduced the service level over the whole spectrum of latencies.

Google Compute Engine (GCE) autoscaling & Kubernetes. Rows **D-E** in Figure 3.3 show that the GCE/Kubernetes deployment is able to cope with every offered load pattern. Row **C** reveals that the cause is that the native GCE autoscaling service quickly arrives at the scaling decision. Thus, the scaled up pod replicas are distributed over more VMs. As a result, higher load can be served. However, even with the smallest demonstrated autoscaling latencies, the load patterns 2 and 4 still result in an increased tail latency as shown in row **D**. The latency drops only when the current state matches the desired for both layers. Although this happens fast, on a scale of just a few seconds, still, for the steep load increase, some requests end up waiting longer than the majority (tail latency).

3.1.3 Conclusion

The evaluated state of art of the reactive autoscaling clearly indicates that it does not matter how low the autoscaling latency actually is – as long as it exists, a steep enough load increase will result in the tail latency. The increasing scale and the complexity of the modern cloud-native applications combined with the tight service level requirements leads to the longer tails of the response times distributions [36]. The in vivo evaluation of the state-of-art autoscalers clearly demonstrates that the reactive resource utilization-based autoscaler cannot cope with the challenge of serving user requests on the milliseconds scale.

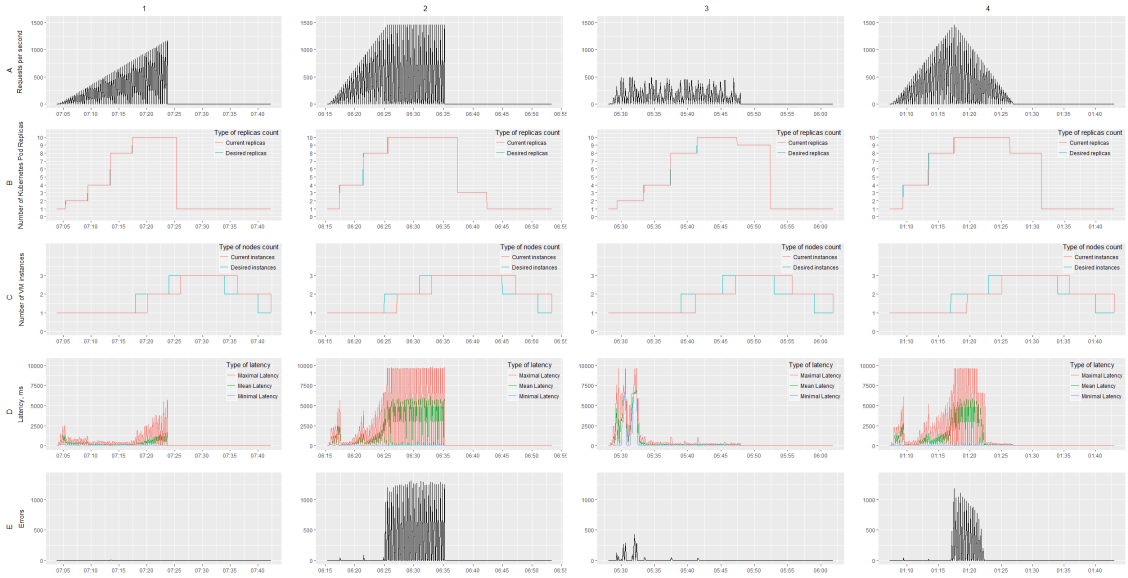


Figure 3.2.: Autoscaling performance evaluation results for the Azure Autoscale & Kubernetes pair [11].

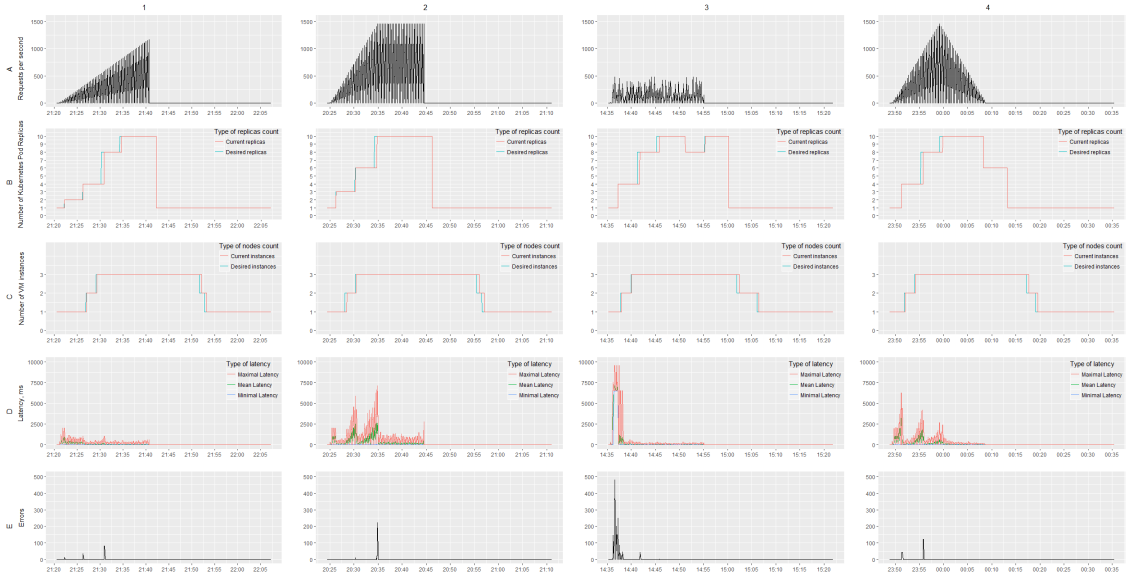


Figure 3.3.: Autoscaling performance evaluation results for the GCE autoscaling & Kubernetes pair [11].

4

Related Work

4.1 Autoscaling

Autoscaling spans multiple layers of abstraction, e.g. it can be performed on a single node, across multiple nodes, or even across multiple sites. In addition, there are many systems and applications aspects related to it, such as load balancing, state replication, fault tolerance, resource management, replica placement, virtual network topology optimization, and ensuring service level for the end user.

4.1.1 Autoscaling: General Aspects

The development of the autoscalers and autoscaling policies requires one to understand multiple facets of the autoscaling process and its impact on the service level.

In [13], authors formulated the problem of autoscaling VMs clusters as an optimal control problem with the cost function incorporating the cost of resources and the cost of deviation from the desired performance objectives. Similarly, the application of control theory to the web services scaling was investigated in [37]. A mapping of the autoscaling concepts onto the corresponding concepts of the control theory and the evaluation of the control-theoretic approach to the elastic resource management was presented in [38]. CACTOS, a framework to model applications and data centers, to simulate them for scheduling and operating purposes, and to optimize application deployment and resource use automatically based on the control theory, was proposed in [39]. The generalized optimization framework for autoscaling that conceptualizes optimizations in the cloud in multiple provisioning models such as IaaS, PaaS, and SaaS by identifying the sources of information for optimization, optimization goals, and optimization actions, was proposed in [40]. An importance of autoscaling and placement as the part of self-management for the cloud services to satisfy the service level agreements (SLA) was emphasized in [41] and [42].

Traditionally, literature represents autoscaling in a framework of the optimal control theory [43, 44]. This framework captures the key aspects of the autoscaling, i.e. the feedback loop that is at the heart of any autoscaler and the optimization target that the autoscaler tries to meet. This thesis does not depart from the optimal control theory framework and its terminology. However, the proposed autoscaling policies and mechanisms are not necessarily built using control theory modeling and optimization methodology.

4.1.2 Autoscaling for VMs and VM Clusters

IaaS offers an opportunity to scale virtual machines. Scaling of a single virtual machine can be done by expanding or shrinking resources allocated to it such as CPU cores, memory, disk space, and network bandwidth. Such scaling is often referred to as a *vertical scaling*. Scaling of the clusters of virtual machines is done by increasing or reducing the number of virtual machines in the cluster. This type of scaling is referred to as *horizontal scaling*. Historically, the autoscaling solutions appeared first for IaaS offers, in particular, for VM clusters scaling. Most of the techniques used in IaaS paradigm came directly from the data center resource management. In what follows, we proceed through the categories of autoscaling approaches introduced in Section 2.2.5.

Autoscaling for VMs and VM Clusters: Machine Learning, Forecasting and Regression based approaches. Google's Autopilot reduced the scaled jobs slack to 23% from 46% for manually-managed jobs and the number of out-of-memory (OOM) errors by a factor of 10 by using an ensemble of machine learning models that solves a multi-armed bandit form of the resource allocation problem with the resource limits computed using the moving window augmented by an exponential decay weighting [18]. A proactive-reactive VM cluster autoscaler Chameleon combines the arriving load intensity forecasting with the run-time service demand estimation using the service demand law to determine the desired amount of VMs for scaling without the application instrumentation [45]. Telescope extends the quality of autoscaling by offering a hybrid multi-step-ahead workload forecasting based on the time series decomposition combined with the K-means clustering for single time periods powering the centroids forecasting with the feedforward ANN, with the ARIMA used for trend forecasting, and XGBoost for combining the forecast components [46]. Moving ahead, authors propose to automate the selection of the forecasting model using either the oversampling and the binary classification or the recommendation based ensemble forecasting [47]. Budget-aware Performance-Feedback Autoscaler (PFA) scales VM clusters to accommodate the workflows of workloads by determining the resource profile of an application and predicting the resource demand using the token-based approximation [48]. In [49], [50], and [51], authors investigate an approach to share the virtual resources between multiple cloud tenants based on their workload profiles. Each tenant is represented by a reinforcement learning agent adapting the cloud capacity in a technical debt framework by using such attributes as amnesty and interest to improve the scaling actions. DuoScale achieves the target latency by using requests cloning on low-capacity VMs with cost savings up to 50% compared to scaling only the high-performance VMs with help of forecasting the requests arrival times and predicting the latency for overhead cancel and no cancel policies [52]. In [53], authors propose a combination of Kalman filter for preprocessing the monitoring signal with the prediction techniques such as ARIMA, feedforward neural network, and support vector machines to enable proactive provisioning of VMs on autoscaling to meet the service level objectives. Such a solution may be improved by adding a reactive autoscaling as a fallback scenario [54]. Scryer is a predictive autoscaling engine used at Netflix to scale the VM clusters in advance according to the forecasted user's demand with a fallback to the reactive algorithm[1][2]. In [55], authors propose a niche reactive cluster autoscaling algorithm that scales the cluster up only if the costs of providing a new VM can at least be covered by the profit received. [56] explores the predictive autoscaling approach for optimizing the provider's costs by using the linear regression and support vector machines techniques to predict the requests prior to making the scaling decision. [57] combines the ideas of previous two papers in a hybrid autoscaling technique that scales the VM cluster both in proactive and reactive way. In [58], authors propose a VM clusters feed-forward adaptation scheme that is based on the layered queuing network with parameters tuned by an extended Kalman filter.

[1] https://netflixtechblog.com/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270
[2] https://netflixtechblog.com/scryer-netflixs-predictive-auto-scaling-engine-part-2-bb9c4f9b9385

Autoscaling for VMs and VM Clusters: Optimal Control based approaches. Elasticity controllers for clusters of virtual machines based on the estimation of the future load managed to reduce the over-allocation of resources up to 32% and the SLA violation rate up to 2.1 times compared to a regression based solution [12]. Similarly, a hybrid reactive-adaptive controller implementing the horizontal autoscaling of the VM cluster managed to lower the delayed requests count by a factor of 3 for bursty cluster traces when compared to a pure reactive controller [59]. Automatic workload classification using K-nearest neighbors classifier allows to augment the ensembles of elasticity controllers with the ability to switch between them depending on the workload characteristics such as periodicity and burstiness [60]. In [61], authors design and evaluate a feedforward controller to predict the resource capacity requirements for VM cluster augmented by a feedback controller to dynamically adjust the parameters of the latter based on the discrepancy between the average and the reference requests waiting time. An attempt at vertical autoscaling of user-facing applications based on an inverse relationship between the average response time and the resource capacity allocated to it in an open and closed feedback control loop models is made in [62]. A dynamic hybrid memory scaling controller for Xen hypervisor that uses discrepancies both in the application response time and in the memory utilization as the measured error for the controller's input is introduced in [63]. It allowed authors to achieve memory utilization close to 83%. In [64], they proceed by proposing a fuzzy controller that dynamically scales both CPU and memory in a coordinated manner. Similarly, [65] introduces a KPI-agnostic Model Predictive Multiple Input Multiple Output (MIMO) controller to adjust the CPU and memory allocation to virtual machines hosting the application such that the desired service level objectives are met. A virtual machine repacking approach proposed in [66] demonstrates 7% – 60% resource cost saving by finding an optimal VMs count to serve the application with the desired service level while minimizing the cost.

Autoscaling for VMs and VM Clusters: Pure optimization based approaches. In [67], authors treat the VM type selection for the horizontally scaled heterogeneous cluster as an optimization problem with a cost-migration delay trade-off – the problem is then solved using the Lyapunov optimization techniques.

Autoscaling for VMs and VM Clusters: Algorithmic approaches. ElasticSFC is an autoscaling algorithm for scaling the service functions chains (SFC) in the network functions virtualization (NFV) context that compares average end-to-end latency in SFC as well as the average bandwidth utilization and the average CPU utilization against the thresholds to devise the scaling actions both for bandwidth and the network functions [68]. In [69], authors claim more than 50% cost reduction in the Giraph graph processing system deployments by dynamically repartitioning the VM cluster through merging the virtual machines and migrating the replicas depending on the memory requirements of the application. In [70], authors propose a set of algorithms unified under the name lightweight scaling that implement the combination of the fine-grained scaling by modifying VM resource configuration with an automatic reactive scaling.

In contrast to the examined papers, this thesis takes an approach-agnostic stance on autoscaling of VMs and VM clusters. This is achieved by distilling the common aspects of the autoscaling models and approaches prevalent in the literature and using them as building blocks for the autoscaling policy. Being agnostic to the approach allows to synthesize the *ad hoc* autoscaling policies, i.e. the policies that are most appropriate for the particular setting and the scaling target.

4.1.3 Autoscaling for Applications and Services

Decoupling application and service autoscaling from VM clusters autoscaling improves the dynamic resource capacity allocation. The reason is that some nodes might be able to co-locate multiple service instances. These instances share the common resources such as CPU and memory, and may also exhibit

bursting properties, i.e. the sum of the service instances resource requirements is higher than what the hosting node can offer. In addition, the co-located services or service instances may share the access to a costly resource such as GPU. Lastly, for the application owner it is easier to think in terms of application scaling.

Autoscaling for Applications and Services: Machine Learning, Forecasting and Model-based approaches. Chamulteon adapts the original approach by Chameleon [45] to autoscale the applications consisting of multiple services in a coordinated manner by forecasting the requests arrival rates only at the user-facing service with Telescope [46] and deriving the scaling action based on the results calculated for the preceding service [71]. In [72], authors propose to address the issue of DoS attacks by using the combination of Kalman filter with an OPERA layered queuing network (LQN) model to predict the performance of a multi-tier cloud application and to adapt it to the upcoming changes in workload. ATOM autoscaler is also based on an LQN application model – it generates the candidate scaling configurations based on the genetic algorithm and then searches for the solution that generates the most revenue and minimizes the total allocated CPU shares [73]. [74] proposes a cost- and profit-aware cloud applications and VM clusters autonomic management mechanism by solving the corresponding online optimization problem with help of ML-based performance models and the feedback loop that adapts the application at compute and network levels. In [75], authors introduce a model to predict the application's tail response times in case of replication without canceling. This is achieved by employing the phase-type distributions, Markovian arrival processes, and the correlated hyper-Erlang (CHE) distributions; the average prediction errors for the 99th percentile of response times are in the range 4% – 7% on MATLAB and MediaWiki benchmarks. An autoscaling framework for containerized elastic applications that scales the containerized application in three stages was proposed in [76]. This framework starts with the reactive scaling, then it predicts the demand for the CPU utilization using the tuned ARMA model. In turn, this enables the framework to scale the application proactively. MYSE is an architecture proposal for predictive autoscaling of replicated services based on queuing models and forecasting for requests inter-arrival times (using dense feedforward neural network) as well as for the service times [77]. PASCAL, an evolution of MYSE, enables predictive horizontal autoscaling of operators in the streaming applications such as Apache Storm by forecasting the future workload, profiling the performance of the application services, and combining these results to conduct the scaling actions in advance [16].

Autoscaling for Applications and Services: Optimal control based approaches. In [78], authors propose to address the challenge of the SLO-compliant multi-tier applications autoscaling by decomposing the application-level SLOs into per-tier SLOs and equipping each tier with a dedicated proportional integral derivative (PID) controller that adapts the count of nodes in the cluster.

Autoscaling for Applications and Services: Pure optimization based approaches. In [79], authors propose to solve an optimal application deployment problem by predicting application performance with the Layered Queueing Network (LQN) model that builds on results of solving individual network flow models (NFM) as linear programming subproblems that ensures the scalability of the approach.

Autoscaling for Applications and Services: Algorithmic approaches. In [80], authors propose to use the resource deflation of applications that utilize the transient cloud resources to reduce the performance degradation by up to 2 times compared to the preemption-based approaches via cascading resource reclamation technique. An alternative approach for the application autoscaling to meet SLOs based on the network bandwidth management in the overlay networks tries to improve the response time by changing the bandwidth of the application flows with a greedy hill climbing heuristic [81]. In [82], authors propose to increase web applications' fault tolerance with autoscaling policies that take into account multiple resources such as CPU, memory, network and disk usage and switch between the spot mode and the on-demand mode with the bidding strategies to improve cost-efficiency and reliability both for the cluster and the application. A partial replication system sPARE was shown to improve the tail latency of MediaWiki and Solr multi-tier

applications by a factor of 2.7x and 2.9x, respectively, by using the cross-tier variability-aware coordinated service replication via iterative search [83] and augmenting the application with a replication-aware arbiter that uses token-based dispatching of requests [84]. CAUS is an elasticity controller for containerized microservices that considers contribution by each container to make the horizontal scaling decisions – the scale down occurs only if the container's contribution is lower than the specified threshold [85].

As was already emphasized, this thesis takes an approach- and model-agnostic stance on the autoscaling policies and mechanisms. This applies to the applications and services scaling as well.

4.1.4 Autoscaling for Storage

Elastic storage is a specialized area that received a lot of attention from the autoscaling community recently. Such attention is caused by the fact that most applications are stateful. As a consequence, autoscaling of the frontend and of the application logic necessarily means that the storage should be scaled as well, otherwise it becomes a bottleneck. A special feature of scaling the storage is in the need to ensure the consistency when scaling it horizontally. In addition, elastic storage exhibits universal asymmetry property – every request is either *read* or *write*. This asymmetry allows to scale the storage differently depending on the kind of workload.

Autoscaling for Storage: Machine Learning, Forecasting and Model-based approaches. OnlineElastMan scales cloud storage services using the weighted majority workload forecast based on ARIMA models and the regression tree combined with the support vector machines-based VM classification into the service class based on the intensity of read and write requests and the requested data size [15]. ProRenaTa wraps the reactive and proactive approaches to cloud storage scaling by combining SLO violation prediction based on the storage reads/writes with the workload forecasting using the Wiener filter for the stable load, cyclic behavior and background noise and the autoregressive (AR) model for the periodic peaks [86].

Autoscaling for Storage: Optimal control based approaches. In [87], authors propose to address the challenge of managing the distributed elastic storage by deriving a state space model for the elastic storage [88] and implementing the combination of the Least Quadratic Regulation (LQR) feedback controller with the fuzzy controller. ElastMan combines proportional-integral (PI) feedback controller with the binary classification-based feedforward controller both targeting the 99th percentile read latency to manage the number of servers for the deployment of a key-value store [14].

Autoscaling for Storage: Algorithmic approaches. Elastic ephemeral storage Pocket demonstrates performance similar to ElastiCache Redis for serverless analytics while reducing the cost by almost 60% via leveraging user hints to make cost-effective resource allocations when scaling [89]. PAX employs query sampling and knowledge of the data partitioning across the cluster nodes to adapt the capacity of Cassandra clusters both in proactive and reactive ways via ensuring that the average node CPU utilization stays in the defined range [90].

4.2 Service Level & Performance Modeling

The discrepancy between the delivered and the desired service level is commonly perceived as a measure of an application autoscaler's efficiency. Such a variable can be used directly as a metric to produce the scaling action that brings the observed service level towards the desired value. Service level alone, however, is insufficient to determine the scaling action. In practice, a performance model of the autoscaled system

should complement the service level. Such model establishes the relation between the managed parameters of the systems, such as VM count and their types, and the delivered performance, e.g. in terms of service level metrics such as response time and throughput. Reversing this relation and combining it with the desired level of service yields parameterization of the scaling actions that has high chances of meeting the service level objectives. In this section, we focus at the related work that studies the ways to improve the service level delivered by the cloud applications.

In [91], authors introduce a general autonomic computing framework for data center resource allocation optimization towards measurable service and business level objectives. A microservice execution framework GrandSLAm aiming to improve the throughput of application is introduced in [92]. GrandSLAm improves the throughput of an application by up to 3 times compared to the baseline by building microservice directed acyclic graph (DAG), calculating the slack at each level of DAG, and then dynamically batching requests with reordering. In addition, it forwards the remaining slack down the microservices chain. A layered queuing network model to capture the relation between the CPU requests and requests rates and the performance parameters such as throughput, queuing delays and resource utilization is proposed in [93]. This model is used to solve the optimal resource allocation problem for cloud resource management. In [94], authors decompose service level objectives into system level thresholds by solving the constraint satisfaction problem. The problem is solved with the combinations of equations for system level thresholds constrained by the high-level application service level objectives. The initial data for solving equations is supplied by the application performance model and the performance profiles of individual application components. A generic performance model for cloud systems is proposed in [95]. This model encompasses 19 metrics divided into 3 abstraction levels: basic performance metrics, cloud capabilities, and cloud productivity. In [96], authors address the problem of QoS-aware optimization in component-based software systems, which microservice applications belong to. The types of optimization problems solved are the availability-cost problem and the three-dimensional availability-performance-cost problem. Terminus is an approach and a tool to measure the capacity of individual microservices in terms of the requests served per unit of time; it is done by exploiting the offline profiling with stress-testing the application with increasing load till the service level objectives are violated [97].

This thesis follows the state of the art in treating the service level metrics and objectives as targets guiding the autoscaling process.

4.3 Resource Management in Data Centers

Resource management is an overarching research area that encompasses autoscaling. Resource management is one of the main concerns of a cloud services provider. CSPs aim to maximize the utilization of physical servers while minimizing violations of the service level agreements (SLA) with their clients [98]. The majority of concepts and techniques used in autoscaling were borrowed from the DCs resource management research.

4.3.1 Classical Data Centers Scheduling

Scheduling in data centers is an old problem which was approached by many researchers. There exist many general-purpose solutions as well as more specialized ones.

Reconsolidating PlaceMent scheduler (RPM) is an example of a classical data center scheduling solution [99]. It assigns resources to virtual machines and solves the distributed bin packing problem to colocate VMs on physical servers. This is achieved by consolidating the VMs in a topology-aware manner,

and taking into account migration costs, resource contention, and load variability. A Peer-to-Peer (P2P) resource management framework for data centers is proposed in [100]. This framework maximizes utilization of a data center while adhering to the resource constraints of the servers. It uses three heuristics to select a neighbor which will receive the VM placement request, namely, most-utilized node with sufficient capacity, least-utilized node with sufficient capacity, and the first-fit that has sufficient capacity (including self). AdaptiveConfig is a run-time configuration solution for cluster schedulers. It first estimates jobs' performances under various configurations and scheduling scenarios, and then transforms the cluster-level search in a large configuration space into a dynamic programming problem solvable at scale [101]. Resource Prediction and Provisioning scheme (RPPS) uses time series extrapolation based on ARIMA models to estimate the required resource provisioning in a data center [102].

4.3.2 Quality of Service-Aware Schedulers

Classical schedulers work on just a single resource abstraction layer and rarely, if ever, use any application-specific information. However, recent decades witnessed the emergence of scheduling techniques that incorporate service level information (aka quality of service) into the scheduling procedure to improve the effects of scheduling decisions on the experience of end users.

In [103], authors view resource allocation in private cloud as an optimization problem. They propose a model relating the quality of service to the resources by mapping a given service level and the resource consumption to multiple profit metrics. A hybrid cloud resource manager Hcloud achieved 2.1X service level improvement compared to the fully on-demand systems and cost reduction by 46% compared to the fully reserved ones [104]. It did so by dynamically mapping jobs to the resources with the following principles: 1) reserved resources are utilized before on-demand; 2) application that can be accommodated on-demand should not delay scheduling of interference-sensitive jobs; 3) the system must adjust utilization limits of reserved instances to respond to the performance degradation. A distributed scheduler Tarcil reduced task execution time by 41% over a distributed, sampling-based scheduler [105]. These results were achieved by adjusting the sample size to satisfy statistical guarantees on the quality of allocated resources and backing off to batch sampling in case of a job requesting multiple cores. A cluster management system Quasar demonstrated an improvement in the resource utilization of the 200-server AWS EC2 cluster by 47% [106]. Quasar adopts a performance-centric approach by allowing the users of the schedulers integrated in frameworks, such as Hadoop or Spark, to express the workload performance constraints. The system uses multiple classification techniques to estimate the impact of a resource allocation on workload performance for the resource allocation. An online scalable heterogeneity and interference-aware DC scheduler Paragon enforces performance guarantees for 91% of applications [107]. Paragon classifies applications with help of collaborative filtering techniques that combine a minute's worth profiling data about the new application with the data available from previously scheduled applications. A QoS-aware resource manager PARTIES improves throughput of latency-critical applications such as Memcached and MongoDB on average by 61% [108]. This was made possible by observing the *resource fungibility* which allows to trade one type of resource such as CPU for another one such as memory. In [109], authors propose to solve the bin packing problem when allocating shared resources in a data center by using the Gaussian percentile approximation as a fitting criterion. The proposed method is evaluated on the Google trace data. In [110], authors propose a heuristic algorithm that selects an allocation strategy with the smallest number of servers required among the shared allocation (SA) and the dedicated allocation (DA) strategies. To achieve this, the algorithm considers how the queue-based performance model of both impacts the service level. A VM QoS-aware allocation controller for overbooked data centers is proposed in [111]. The controller supports two isolation levels, low QoS and high QoS. It allocates the data center capacity based either on the throughput performance model or the response time performance model.

4.3.3 DC-scale Operating Systems

Several years ago, practitioners and researchers proposed to view data centers as warehouse-scale computers and to manage the resource in the same manner as if DC was a workstation. Currently, an active research direction is devising an operating system that operates at a scale of the whole data center and manages its resources [112]. An example is a single-address space operating system Shinjuku [113]. Shinjuku demonstrated up to 6.6× throughput improvement and the 88% tail latency reduction for RocksDB server processing both point and range queries. It did so by preempting the running requests after 5 to 15 microseconds which is extremely fast compared to the Linux kernel. In addition, Shinjuku supports a single queue policy that does not differentiate between requests types and a multi queue policy where each queue is assigned to requests of a particular type read from header.

4.3.4 Scheduling Theory

Scheduling theory is a research area with a long systems formalization and modeling tradition deeply rooted in operations research field of mathematics. Scheduling theory is often applied to model scheduling decisions in data centers and to derive the theoretical guarantees for various scheduling decisions based on job shop and other models [114]. In [115], authors address the data center resource management problem for tasks of multiple types by proposing a side-effects model for co-locating tasks. They formulate a corresponding MinMaxCost with types (MCT) problem for resource allocation. For a constant number of types, a polynomial time approximation scheme is proposed to solve this NP-hard problem.

This thesis tries to question the fundamental resource management assumptions, models, and approaches from the point of view of applicability to a particular autoscaling scenario. We do not assume that the resource utilization is a key metric unlike the majority of the resource management research. Instead, any metric that is used in an autoscaling policy has to proof its objective superiority when compared to other metrics under the same conditions.

4.4 Placement

If the application to be scaled breaks topological or resource requirements symmetry in some way, the aspect of *placement* becomes prevalent. For instance, placement should be considered when some services express preferences towards particular types of resources such as VMs with GPUs or towards VMs allocated in a geographical region that is close to the end user. In addition, asymmetric resource requirements by VMs and application services point at an opportunity to co-locate entities exhibiting different resource usage patterns on the same node to maximize the resource utilization. For instance, a service that generates high memory pressure but consumes not much CPU (e.g. requests cache) can be placed on the same node with a CPU-intensive task that is not memory-hungry (e.g. computing an optimal path on a map). Such co-locations often become a subject to the performance interference [116].

REMaP mechanism for the automatic placement of the microservice applications was proposed in [117]. This mechanism is based on solving the bin packing problem on runtime by taking into account microservice affinities expressed as communication paths between the microservices and the resource usage history. A system for replica placement on heterogeneous unreliable machines in a decentralized context is proposed in [118]. First, it scores replica placements with a utility function that is proportional to the expected quality of replicating the data. Second, it maximizes the utility of the currently worst placement via the max-min

optimization. In [119], authors propose a framework for reasoning about the workload placement on NUMA machines using the scheduling concern abstraction. This abstraction represents a single hardware resource or an inseparable set of hardware resources affecting the performance of vCPU placements. To optimize the placement of containers on NUMA machines, authors propose a policy powered by multi-output Random Forest-based performance model. This policy is shown to take more optimal vCPUs placement decisions than the competing conservative, aggressive, and smart-aggressive policies. A process placement solution for clusters that uses an a posteriori genetic algorithm (MOGA) is proposed in [119]. It generates Pareto front of possible solutions for a representative workload and then allows the cluster administrator to visually select the appropriate job placements. The chosen schedules are used to calibrate the weights of the automatic solver. A tail latency-aware autoscaler Voilà integrated into Kubernetes addresses pods placement in the geo-distributed setting [120]. It does so by detecting the SLO violations and trying to fix them by moving a replica to another node and backing off to adding new replicas if there are no more free nodes or if the violation is still present. AsymSched is a thread placement algorithm that takes into account the asymmetry in NUMA systems' interconnect and optimizes memory migration with a new Linux system call [121]. The proposed system call uses different paths for migrations to reduce waiting on locks. An analysis of classification schemes for threads with the goal to determine how the threads affect each other when competing for shared resources is offered in [122]. A cloud bursting management system Seagull demonstrated to reduce the cloud costs by more than 45% [123]. It does so by placing VMs to maximize the utilization of local resources, migrating only the applications incurring the smallest cost to the cloud, and by transferring VMs incremental snapshots to the cloud to reduce the booting time. A threshold-based algorithm to scale the VM cluster based on whether the service level objectives are violated (scale up) or the utilization is small (scale down) is proposed in [124]. The algorithm integrates both the application autoscaling and the dynamic VM allocation. Its key difference from the conventional approaches in algorithmic scaling is in the implemented topology-awareness – the algorithm tries to place an application entirely within a single rack.

Like load balancing, placement is usually considered as a standalone problem in the research literature. The thesis incorporates the placement aspect into the autoscaling policy by providing the corresponding abstraction. This allows to evaluate which placement options support autoscaling policies in reducing the tail latency.

5

Generalized Predictive Autoscaler

5.1 Motivation

Predictive autoscaling aims to utilize the available information about the application and the infrastructure to model the future state of the environment and to adjust the application capacity and the system resources to this state. The term "predictive" denotes an adjustment that is made ahead of projected environment change, e.g. change in the user load. The results of predictive autoscaling can be significantly better than that of its reactive counterpart. This happens if the predicted future state of the environment either roughly matches or overestimates the real one.

The discrepancies between the projected state and the real state of the environment can occur in both directions. If predictive autoscaling overestimates the load increase, the system might end with more capacity than needed potentially increasing the costs if the VM cluster is scaled out. On the other hand, the overestimated load decrease may force the predictive autoscaler to decommission more nodes than appropriate. This may result in a lower requests processing speed. Since it is impossible to predict the future with the perfect accuracy, all the predictive autoscalers have to operate in the space of cost/quality of service tradeoff. Despite the inability of predictive autoscalers to fully hold the promise of their name, there are two basic scenarios, which they are irreplaceable in.

Long-term capacity planning. Since predictive autoscaling makes future projections about the load, the demand in resources can roughly be estimated using these predictions. In turn, this fulfills two objectives. First, one can often rely on the accuracy of the user load predictions for the cyclic and seasonal patterns. This means that most of the capacity provisioned in such a manner will be in demand and that for most of the users the service level will be appropriate. Second, the long-term load predictions (e.g. half a year and above) allow to estimate the cost of the required system resources and to plan the budget for cloud services accordingly. This, however, is only possible if enough historic data about the load was collected (usually, 1-2 years). It is desirable that the application has entered the plateau of the user demand or at least the demand does not grow exponentially. Such a service is frequently provided by CSPs, e.g. both AWS and Google extrapolate current bill for cloud services for days and months ahead[1].

[1] `https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/ce-forecast.html`

Cutting the long tail of response times distribution. As was demonstrated in Chapter 3, reactive autoscaling is prone to fail at finding a suitable trade-off between meeting tight SLOs and using as less budget as possible. Even if the budgeting condition is relaxed, there still remains a possibility of a flash crowd load. The rapid increase in load in this case usually outpaces the speed at which new virtual machines can be provisioned. In such a scenario, predictive autoscaling can help to improve higher percentiles of response time by taking into account provisioning times of virtual machines and service instances. Predictive autoscaling is still susceptible to flash crowds that cannot be predicted from the historic data. However, advanced techniques might be applied to identify the rapid load increase early enough to over-provision the capacity in a short run. This can be achieved by complementing predictive autoscaling with its reactive counterpart (which is a common technique known as *hybrid autoscaling*) or by adopting finer prediction schemes such as predictions along the request trace in the application. The latter option means that the predictions for an upstream service are made based on the dynamic information about the requests processed by one of the previous services.

The two scenarios specified above allow predictive autoscaling to increase its importance for any business providing its services online. All the predictive autoscaling policies and mechanisms are based on a set of common design principles. In the following sections we develop the design of a *generalized predictive autoscaler* that represents a modular feedback loop-based architecture adaptable to the requirements of particular deployments and load patterns.

5.2 Key Design Principles of Predictive Autoscaling

The following principles form the foundation of useful predictive autoscaling policies and mechanisms:

- **Predictive autoscaling should be organized as a feedback loop**. The literature conventionally employs the MAPE-K framework with a feedback loop at its core for designing the autoscaling solutions and policies [125]. The metrics characterizing the state of the environment and the system are collected (monitoring, **M**), then they are analyzed (**A**), next, the required adaptation to the system is scheduled (**P**), and, lastly, the adaptation is executed (**E**), or enforced, when the time comes. Some runtime information is persisted in the form of models that capture the regularities in the autoscaling process (knowledge, **K**). The new state, that the system ended up in, produces new values of the same metrics to be fed into the adaptation process again on the next cycle thus closing the system adaptation loop. The organization of predictive autoscaling naturally fits into this framework. By imposing the feedback structure on the designed predictive autoscaling policies and mechanisms, one can borrow the common models (e.g. from queuing theory and control theory) for implementing different stages of the loop. The adaptation of this feedback structure might turn out to be relatively small and specific to the use case.
- **Predictive autoscaling should be complemented by reactive autoscaling**. However accurate during tests, predictive models will always fail to provide 100% accuracy when predicting the future. Instead of trying hard to improve the prediction models accuracy by a small margin, a good practice is to add the reactive autoscaling solution in the loop. Reactive autoscaling should assist when predictive autoscaling fails to predict some future state (e.g. a load spike). The most complicated part of this design principle is to organize the cooperation between reactive and predictive autoscaling such that they do not end up interfering with each other's scaling decisions. This can be addressed by an arbitrating mechanism that gives the priority to the more generous configuration of the cluster. The invocation of the reactive autoscaler might also be postponed until the desired state by the predictive autoscaler is enforced.

- **Predictive autoscaling should always stay within the limits of the capacity/resources that it is allowed to allocate**. The resource and capacity allocation limits might be set either directly or indirectly, via budget constraints. Predictive autoscaling is required to stay within the limits such that extreme allocation scenarios such as scaling down to zero (undeploying the application) and scaling up to infinity will not be possible. The latter scenario is also related to the next design principle.
- **Predictive autoscaling should be prone to attacks stuffing the bill**. One of the purposes of autoscaling is to keep the service level high even in the face of a huge load inflow. This is done at the cost of using additional cloud resources. This approach opens up an opportunity for coordinated attacks on autoscaling that attempt to drive the cloud services bill higher without actually serving any real users. A certain kind of such an attack, called Yo-Yo attack, causes both the performance degradation and the budget loss. This is done by sending bursts of load causing the autoscaling solution to constantly switch between the scale up and scale down phases without allowing it to settle [126, 127]. The predictive autoscaling solution should be prone to such attacks by design. Anomaly detection methods might be employed to recognize malicious requests early on [128]. In addition, the predictive autoscaler might be designed to produce steady desired resource estimate for a longer interval of time by smoothing the predicted load.

Although it is rare to find all these principles implemented in a predictive autoscaler, designers are usually aware of them and might choose to drop a few since the use case might not require it or the corresponding functionality might already be implemented by another service. In the following section we will look at the design choices made by AWS when designing their predictive autoscaler.

5.3 Production Example: Predictive Scaling for EC2 in AWS

Back in 2018, AWS introduced predictive autoscaling which is based on the machine learning techniques[2]. Their predictive autoscaling solution follows the feedback loop organization. It monitors the usage of EC2 instances (virtual machines), refines the ML-based prediction model periodically (e.g. every 24 hours) based on the new utilization observations, then it produces the scaling plan (schedule) that consists of scaling actions in the future. Lastly, the actions are enforced changing the composition of the EC2 instances groups. The feedback loop encloses with the AWS solution continuing to monitor the utilization of the EC2 instances groups with the changed composition.

Predictive scaling of AWS is complemented by reactive autoscaling, called "dynamic scaling" by AWS. One is allowed to select either of these or both. The cooperation between predictive and reactive contours is implemented by allowing the predictive autoscaler to schedule the minimum resource pool and using the reactive autoscaler to provision additional resources in case of additional demand. Reactive autoscaling is allowed to adjust the decisions of the predictive autoscaler only in the upwards (scale up) direction. The reactive mechanism is threshold-based.

AWS ensures that both kinds of autoscalers stay within the desired limits for VMs clusters by providing a mandatory min/max capacity setting when configuring Auto Scaling Groups of VMs[3]. By default, both min and max are set to 1. In addition to the limits provided outside of the autoscalers immediate configuration, predictive autoscaler of AWS allows to configure its max capacity behavior. This behavior determines whether the cluster is allowed to scale up above its max limit when the predicted capacity is close to or exceeds the specified maximum. Several options are offered to the user: to enforce the max capacity setting,

[2] https://aws.amazon.com/ru/blogs/aws/new-predictive-scaling-for-ec2-powered-by-machine-learning/

[3] https://docs.aws.amazon.com/autoscaling/ec2/userguide/asg-capacity-limits.html

to set the max capacity equal to the predicted capacity, and to increase the max capacity above the predicted by a specified buffer value.

Despite following a number of previously outlined design principles, predictive autoscaler of AWS does not address attacks on autoscalers. Although AWS Shield service offers protection against DDoS attacks, it is aimed at solving the performance issues. The pricing aspect is not taken into account, hence a discussed Yo-Yo attack can devastate the cluster owner's budget if the allocation restrictions are too loose.

5.4 Constituents of Predictive Autoscaling

Predictive autoscaling process can be abstracted into the generalized predictive autoscaler architecture presented in Figure 5.1. This architecture is described in greater detail in the following chapters.

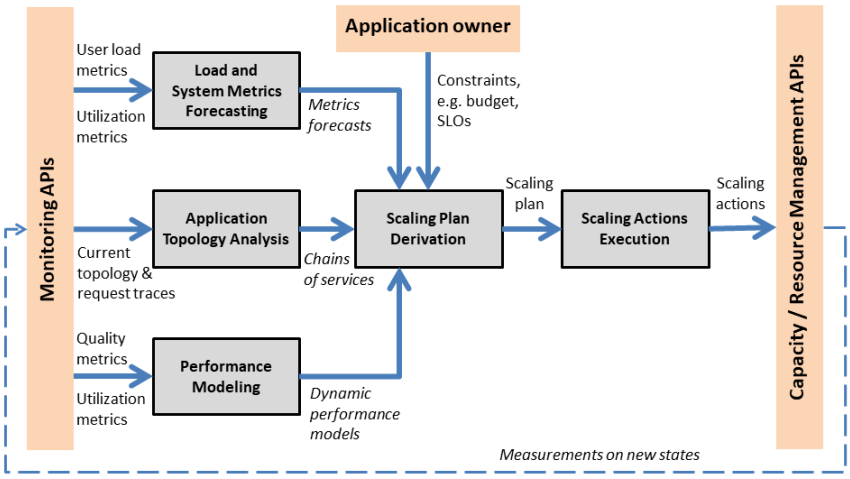


Figure 5.1.: Generalized predictive autoscaler with an explicit feedback loop.

Generalized predictive autoscaler relies on the monitoring APIs to capture various metrics dynamically. In addition to the common metrics, some monitoring solutions provide observations for the service level as well as the request traces. Usually, this is done with help of small code snippets introduced into the application codebase (instrumentation). Majority of cloud providers and resource managers/container orchestrators offers monitoring subsystems, hence monitoring is usually externalized to the predictive autoscaling solution. Collected metrics are utilized by three analytical processes: load and system metrics forecasting, application topology analysis, and performance modeling.

Load and System Metrics Forecasting process is included even into the production-grade predictive autoscaling solutions such as EC2 Predictive Scaling by AWS. It maintains one or more models to forecast the metrics represented as time series. These models are actualized periodically based on the accumulated historic data. They are used to continuously forecast future values of the monitoring metrics on some *forecasting horizon*. To improve the accuracy, separate models might be derived for different lengths of forecasting horizons, e.g. short-term (1 hour), mid-term (1 month), or long-term (6 months an more). Forecasting models are used to compute the anticipated future value of a metric, e.g. how many requests per second will arrive in 3 days (load). We discuss forecasting in Chapter 6.

Performance Modeling process is responsible for creating and tuning the performance representation of the scaled entity, e.g. a VMs cluster. The performance representation, or performance model, provides

a mapping from the monitored metric, such as utilization or incoming requests per second, to the performance measure, e.g. the response time or the task execution time. Performance modeling offers a rich set of techniques and tools that generally fall into either of two categories: offline and online. Offline performance modeling is based on performance traces and metrics collected during the specified time frame by making several application runs under different load [97]. Online performance modeling does not require full-fledged application runs in a sandbox environment. Instead, the model is continuously updated using the real-time measurements of input and performance metrics. The main limitation of online performance modeling is that it can only use the observations available under the normal conditions which may introduce bias into the model. A deeper insight into performance modeling for predictive autoscaling and its connection to the resource allocation is provided in Chapter 7.

Application Topology Analysis process is relevant for the distributed applications consisting of multiple services that are connected through the requests processing chains. Complex application topologies usually result from adopting the microservice architectural pattern. Since the functions of such applications are split into tiny maintainable pieces, there are usually many of them – the count can reach thousands in the industrial applications [129]. Relevant insights from the automated application topology analysis are about the communication load on the network and the fan-in/fan-out load[4] on particular services along the request traces. Having this information, the decisions taken by the predictive autoscaler can take into account the cascading effects, i.e. if a particular service along the request's propagation path got scaled up then the downstream services should also be scaled up to avoid creating the bottlenecks. Topology analysis for predictive autoscaling is discussed in Chapter 8.

Scaling Plan Derivation process implements some form of optimization on the provisioned capacity/resources. Capacity/resource provisioning is commonly formulated as a multi-objective constrained optimization problem. Autoscaling represents a subset of this problem, hence all the optimization techniques and methods are applicable. The main difference is that predictive autoscaling should solve the optimization problem for a sequence of future states. This complication is addressed by formulating the optimization problem for each of these states and taking into account the interval of time between them. Once the optimal or suboptimal states are found, they are organized in a *scaling plan*. A scaling plan is a schedule of scaling actions to be performed in the future. Upon changing the environment conditions (load, faults), it may be recomputed. For practicality reasons, the discussion of the **Scaling Plan Derivation** process is mixed into the **Performance Modeling** process in Chapter 7. It is not discussed at length due to abundance of literature on optimization for dynamic resource provisioning.

Lastly, execution of the planned scaling actions is performed by the **Scaling Actions Execution** process. Execution of the scaling actions is done by periodically checking the current scaling plan and enforcing the planned actions when the time comes. The enforcement is done via **Capacity / Resource Management APIs** that are provided by the container orchestrators and VMs clusters managers. These APIs differ in the functionality offered, but they unite in a basic subset of operations. For example, one can find a command to scale a pod programatically in Kubernetes[5] and also a *SetDesiredCapacity* API call to programatically change the desired capacity of Auto Scaling Group in AWS[6]. Execution of any scaling action through the Capacity / Resource Management APIs changes the state of the application and of the VM cluster resulting in the workload redistribution and the resource utilization change. Combined with the changes in the user load, this propels a new round of predictive autoscaling, or, more commonly, *state reconciliation*.

[4] Fan-in means accepting requests from multiple services. Fan-out means sending requests to multiple services.

[5] `https://kubernetes.io/docs/reference/kubectl/cheatsheet/#scaling-resources`

[6] `https://docs.aws.amazon.com/autoscaling/ec2/APIReference/API_SetDesiredCapacity.html`

6

Load and System Metrics Forecasting

6.1 Objective and the Groups of Forecasted Metrics

The objective of load and system metrics forecasting is to estimate the state which the system is anticipated to be in. Given periodic or cyclic patterns in the load or system metrics, the results might be quite accurate. Being able to project the future state, one can prepare the system to it by scaling up or down corresponding to the anticipated change. Although one can forecast different metrics at once, there is a tendency to prefer either of the following two metrics groups.

Forecasting the user load. The user load is represented with discrete observations of requests per unit of time aggregated in a relatively small window moving along the time axis, e.g. 1 minute is quite common. The user load can be forecasted for each endpoint separately. This approach yields more accurate results since the behavioral patterns differ substantially depending on the intention behind the request.

Forecasting the user load is wide-spread for the interactive cloud applications mostly for two reasons. The first reason is that the user load is by far the ultimate determinant of the resource utilization in the interactive application. In contrast, resource utilization in batch data processing applications mostly depends on the parameters of the submitted job. Moreover, the jobs do not arrive as an unbounded stream. The second reason is that the user load is an unmanaged parameter. Unlike system-level metrics that can be affected by the changes in the allocated capacity/resources with the following load redistribution, the user load cannot be capped or boosted at will of application owner. Hence, autoscalers designers adopt this way of thinking: *forecast what you cannot change and adapt what you control*.

Aggregation is an inevitable part of preparing time series for forecasting. First, predictable patterns emerge only at particular levels of granularity. If the observations are not aggregated, then the time series will most closely be represented by a random noise with the very limited forecasting opportunities following from it. On contrary, if the aggregation window is too large, say, 1 month, periodic patterns might be lost, e.g. the diurnal pattern. Second, the time window-based aggregation of raw observation allows to reduce the volume required to persist the time series for modeling.

Forecasting the system metrics. System metrics that characterize the resource utilization and the performance are generally represented similarly to the user load. The major difference lies in the mixture of

different processes underlying the values of these metrics. They depend on the user load, the capacity of the VMs clusters, the system resources allocated, the performance characteristics of the underlying physical machines, the scheduling algorithm that the hypervisor uses, the background system tasks performed by the OS, the technology stack that the services are build upon and whether it supports automatic garbage collection, etc [130, 131]. With many parameters impacting system-level metrics, the possible patterns are usually obscured by a significant amount of noise that is almost impossible to reason about. In addition, the saturation effects in the system may render the results of system metrics forecasting useless. Low variability of the signal does not allow to fit a useful extrapolation model since all the models require variability to capture the regularities in the observation sequences.

The following discussion will focus at forecasting the user load since this metric determines the behavior of interactive transaction-based applications to a large extent. For a more in-depth discussion about the motivation to forecast the user load, the reader is invited to refer to Chapter 10.

6.2 Common Time Series Forecasting Methods

Linear Regression. Linear regression is one of the simplest models that can produce meaningful results for the time series data. The regression equation relates the value of a time series variable (e.g. CPU utilization) at the given point in time to the *linear combination* of some other values. It may be similar to the autoregression models when the linear combination consists of past values of the variable. The equation usually includes an intercept and looks like follows in case of regression on the past values of the same variable:

$$z_t = \alpha_0 + \alpha_1 \cdot z_{t-1} + ... + \alpha_h \cdot z_{t-h} \tag{6.1}$$

Linear regression can be used as a baseline for comparing against more advanced forecasting methods. The coefficients are adjusted by minimizing the squared distances between the data points and the predicted result. For time series with a well-expressed trend, linear regression may produce predictions that are on par even with the most sophisticated models.

Exponential Smoothing. Exponential smoothing approximates time series using the exponential window function. Exponential functions assign decreasing weights to past observations to reflect higher importance of the recent observations when computing the predictions. A variation of this method was used by Google to implement vertical autoscaling in Autopilot [18].

Exponential smoothing is a family of models that differ in type of their components, i.e. trend, seasonality, and error. A trend could either be absent or could be of one of the following: additive, additive damped, multiplicative, or multiplicative damped. Seasonal component can either be absent, or additive, or multiplicative. R. Hyndman et al. propose to use Akaike Information Criterion (AIC) to select the most appropriate exponential smoothing model [132]. Holt's linear trend method belongs to the family of exponential smoothing models. This method computes the forecast for h steps in the future by linear equation [133]:

$$\hat{y}_{t+h|t} = l_t + hb_t \tag{6.2}$$

Exponential smoothing is hidden in l_t and b_t terms which determine the level and the trend, correspondingly. Holt-Winters extends Holt's method to support the seasonal patterns in time series [134].

Autoregressive Integrated Moving Average. AutoRegressive Integrated Moving Average (ARIMA) process models a time series in a structured way by attempting to capture both the dependence of the variable

on its past values (autoregressive component) and to model the accumulated randomness that cannot be explained with the past values (moving average component). ARIMA is a family of models. Among all others, it includes models to account for the time series seasonality (SARIMA) and for the additional signals, so-called external regressors (ARIMAX). The generalized SARIMA model for the time series z_t is conventionally represented by the generalized equation [135]:

$$\Phi_P(B^S)\phi_p(B)\nabla_S^D\nabla^d z_t = \Theta_Q(B^S)\theta_q(B)a_t \tag{6.3}$$

The equation above has two parts. The left hand side represents the autoregressive component, whereas the right hand side – the moving average component. The autoregressive component establishes the dependence of z_t on its previous values using the backward-shift operator B and the coefficient polynomials $\Phi_P(B^S)$ and $\phi_p(B)$. When applied to the value z_t at time t once, operator B yields the previous (earlier) observation, i.e. $Bz_t = z_{t-1}$. Multiple applications of this operator, deceivingly denoted as an exponentiation B^S, yield values that are *degree, S*, steps behind, e.g. $B^S z_t = z_{t-S}$. A modification $\nabla = 1 - B$ is commonly used to shorten ARIMA expressions. Random noise a_t is modeled with the moving average component with coefficients $\Theta_Q(B^S)$ and $\theta_q(B)$.

The short-hand for the generalized model is $ARIMA(p,d,q) \times (P,D,Q)_S$. Seasonal and non-seasonal components in Equation 6.3 are captured by the polynomials of B with positive integer degrees p, q, P, Q. Positive integer parameters d and D indicate how many times should the time series be differentiated in order to become stationary. ARIMA models are limited in that they require the modeled process to be stationary, i.e. no unit roots are allowed. Fractionally integrated (ARFIMA) process relaxes the requirement on differentiation parameters to be integers [136]. With this relaxation, ARFIMA is able to capture longer dependencies in the time series.

Generalized Autoregressive Conditional Heteroskedasticity. Generalized AutoRegressive Conditional Heteroskedasticity (GARCH) process relaxes the stationarity requirement to the variance of the error term which is left of the data after fitting some ARIMA model. $GARCH(p,q)$ complements ARIMA by modeling only the variance of the error term [137]:

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{q} \alpha_i a_{t-i}^2 + \sum_{i=1}^{p} \beta_i \sigma_{t-i}^2 \tag{6.4}$$

Variance is modeled as a sum of a constant term with parts describing the autoregressive behavior of the error term and its randomness (moving average). GARCH augmentation to ARIMA is required to model the time series that exhibit volatility. In practice, such time series might result from denial of service followed by a retry wave of user requests. The common way to use GARCH in combination with ARIMA is to first fit ARIMA to the time series, and then to fit the *residuals* using GARCH.

Singular Spectrum Analysis. Singular Spectrum Analysis (SSA) is a non-parametric method for time series decomposition based on the spectral information encoded in the series [138]. It has four steps:

1. embedding of original time series into the vector space of specific dimension,
2. singular value decomposition (SVD) resulting in a set of elementary matrices of rank 1,
3. eigentriple grouping to group the elementary matrices acquired on a previous step,
4. diagonal averaging to get the v of the original time series as a sum of reconstructed subseries.

SSA method is appropriate for forecasting using the linear homogeneous recurrence relation [139].

Support Vector Regression. Support vector machines (SVM) are the supervised learning models that were introduced to automate the data classification task. Support vector regression (SVR) is an extension to SVM proposed by H. Drucker et al. [140] to allow modling of continuous relations. Time series forecasting is a subclass of such modeling.

SVR does nonlinear mapping of its input into a feature space. The linear regression model in the feature space is constructed using the supervised learning approach. The linear model in the feature space is described with the following equation:

$$f(x,\omega) = \sum_{j=1}^{m} \omega_j g_j(x) + b \tag{6.5}$$

To fit the model coefficients, SVR employs optimization techniques adjusted by several meta-parameters. These parameters influence the accuracy of the regression model and the convergence speed. As with other machine learning models, poorly tuned parameters may result either in underfitting or overfitting.

6.3 Forecasting Model Selection Methodology

6.3.1 Interval Score for Forecasts Evaluation

Interval forecasts are used way more in practice in comparison to the point forecasts. The reason is that it is impossible to achieve the absolute accuracy when forecasting just a single future from an unlimited set of possibilities. Interval forecasts are based on the notion of so-called Prediction Interval (PI). This interval covers future observations with some probability. For instance, 95%-PI means that the predicted value falls with the probability of 95% into the given interval. The higher is the probability, the larger is the interval. Large intervals are useless since they fail at providing a reasonably narrow range of future values. Taken to the extreme, 100% PI spans all the values that the variable might take on.

Scoring provides a single numerical value based on the out-of-sample prediction results. The number directly relates to the forecast accuracy. Having just a single score per model allows to simplify the comparison. A *negatively oriented interval score* [141] can be used to evaluate the forecasts:

$$S_\alpha^{int}(l,u;x) = (u-l) + \frac{2}{\alpha}(l-x)\mathbf{1}\{x<l\} + \frac{2}{\alpha}(x-u)\mathbf{1}\{x>u\} \tag{6.6}$$

Interval score includes a penalty $(u-l)$ for the wide PI. α is the probability of type I error, and $\mathbf{1}\{y\}$ is an identity function that translates the result of the logical expression into the numerical space. The lower the score is, the more favorable the forecasting model is.

6.3.2 Accuracy/Fitting Time Evaluation to Select a Forecasting Method

Selection of a forecasting method to extrapolate metrics for dynamic resource provisioning cannot be based only on the accuracy measure alone. *Model fitting time* is another scoring parameter. The reason to consider this parameter when selecting a forecasting method, is to make the deployed application's state reconciliation period as small as possible.

Accuracy/model fitting time-based forecasting method selection works as follows. First, the given time series is divided into two sub-sequences. The earlier one is used to fit the parameterized model. The recent one is held out for the evaluation. The best way to divide the time series is to ensure that the first interval is the multiple of the evaluation interval (e.g. 3 weeks / 1 week). Since we are interested in extrapolating values into the future, all the observations in the evaluation sub-sequence should be newer than the observations in the training sequence. Next, the parameters of the model are tuned to ensure the best results. Usually, this is done only once, but the tuning might be repeated from time to time to ensure that the changes in patterns are well-represented. Third, the tuned model is fit to the training sub-sequence. The model fitting time is evaluated on this step. Lastly, the model with the adjusted coefficients is tested on the withheld evaluation sub-sequence. As a result, the interval accuracy score is computed. Now, the model/time series pair can be represented as a point in a two-dimensional space, where one dimension corresponds to the interval accuracy score and the other – to the model fitting duration.

In the following subsection, we provide the results of evaluating most of the discussed forecasting methods on the user load data set for the microservice application.

6.4 Load Forecasting for Microservice Apps: Comparison

Data & hardware. The experimental data set for this section was provided by a private company (now part of IBM) with in-house monitoring solution for applications deployed in the cloud. The anonymized data set was cleared prior to the experiments resulting in 261 non-zero 4 week-long time series with 673 observations in each. Missing values were interpolated. Each observation represents the aggregated count of requests per hour. Last week of data was used for evaluation. The evaluation was performed on a laptop with Intel core i7 processor, 2.7 GHz, and 16 GiB RAM under Archlinux OS (4.12.12-1-Arch).

Comparing the forecasting methods. With 10 forecasting methods evaluated, the result is represented by 2610 points in the two-dimensional space of interval accuracy score vs model fitting time. Fig. 6.1 shows a zoomed-in version of this space for all the test cases with the evaluation results near the origin $(0,0)$. The space is divided into 4 subspaces each corresponding to one artificial performance class for the forecasting methods assessed. These subspaces are determined by two median values: one for the interval scores acquired in tests (**4.5958**) – the black vertical line, and one for the model fitting duration (**0.8966 s**) – the black horizontal line. Each quadrant represents a single class:

- **Class I**: appropriate interval score and appropriate model fitting duration;
- **Class II**: appropriate interval score but non-optimal model fitting duration;
- **Class III**: appropriate model fitting duration but non-optimal interval score;
- **Class IV**: worst score and model fitting duration.

The relative attribution of the forecasting methods to the specified classes is depicted in Figure 6.2. This plot shows how many time series were attributed to one of the four aforementioned classes. Classes I and II represent the models that are the most appealing for practical use in load forecasting. Rankings inside these classes are provided in tables 6.1 and 6.2, correspondingly.

Class I has no "best fit for all" method. Both SSA and SARIMA exhibit high interval accuracy and computation efficiency. Both of them can be used to achieve accurate and fast forecasting. Despite low ranks of the forecasting methods taking outliers into account, they are still irreplaceable when dealing with load bursts. This is clearly seen in Table 6.2 where these methods have high ranks. This is caused by the presence

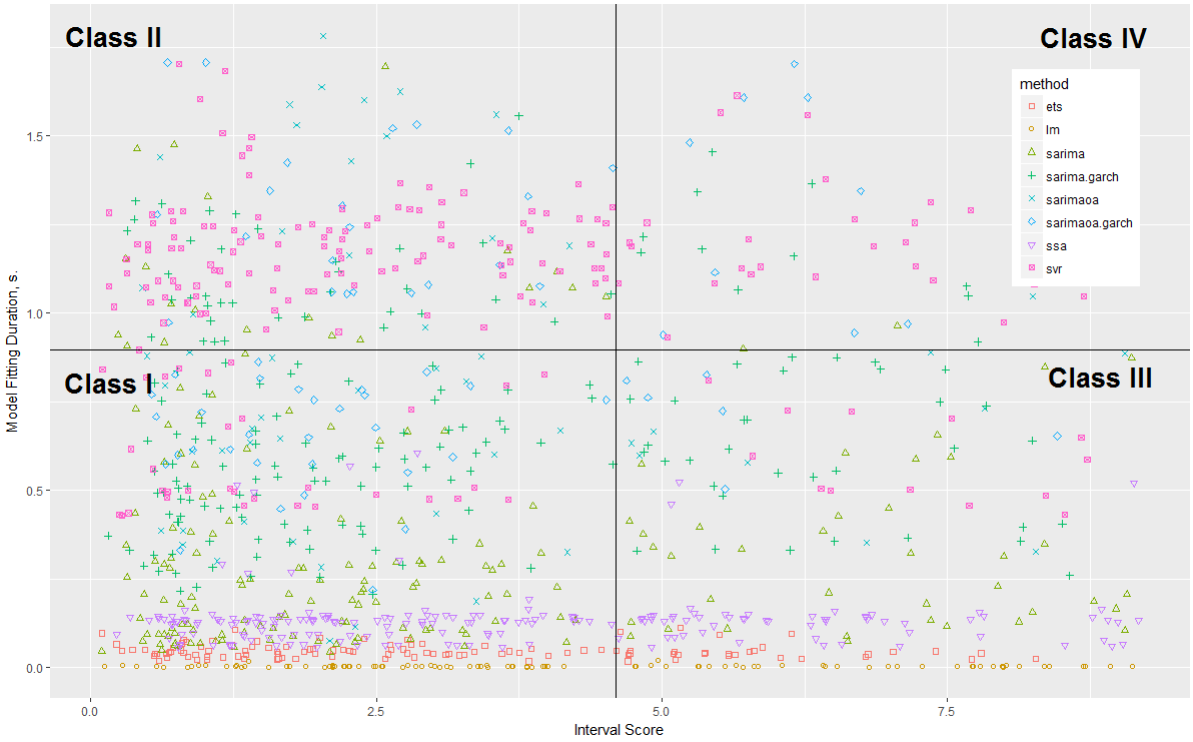


Figure 6.1.: Forecasting models in interval score/fitting duration space (zoomed-in).

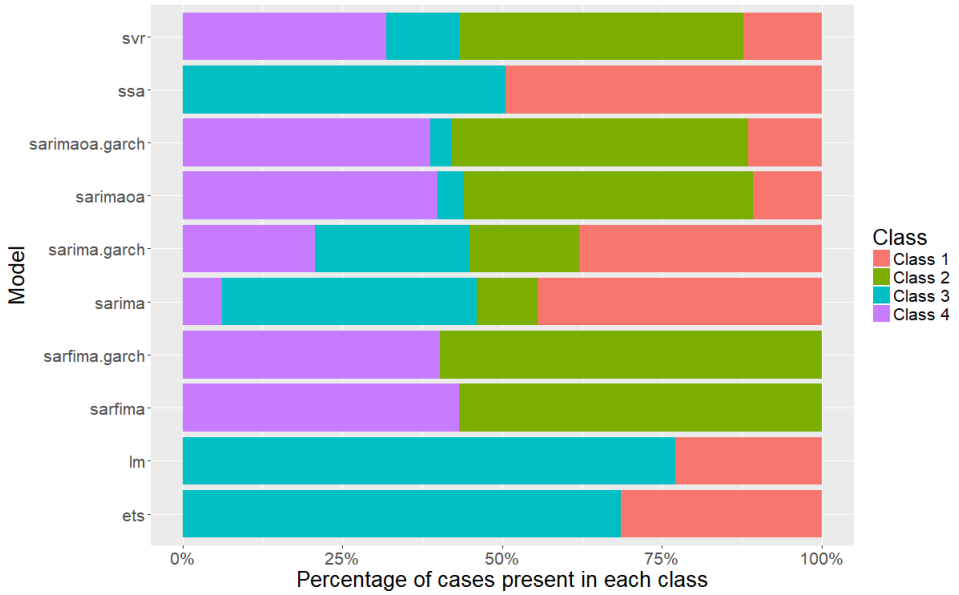


Figure 6.2.: Relative presence of the evaluated forecasting methods in each performance class.

Table 6.1.: Class I: Appropriate interval score and appropriate model fitting duration.

Rank	Method	Number of cases	% of all the cases
1	SSA	129	49
2	SARIMA	116	44
3	SARIMA with GARCH	99	38
4	Exponential Smoothing	82	31
5	Linear regression	60	23
6	SVR	32	12
7	SARIMA with outliers with GARCH	30	11
8	SARIMA with outliers	28	11

Table 6.2.: Class II: Appropriate interval score but not optimal model fitting duration.

Rank	Method	Number of cases	% of all the cases
1	SARFIMA with GARCH	156	60
2	SARFIMA	148	57
3	SARIMA with outliers with GARCH	121	46
4	SARIMA with outliers	118	45
5	SVR	116	44
6	SARIMA with GARCH	45	17
7	SARIMA	25	10

of time series with requests bursts. Due to the inclusion of the terms responsible for capturing the outliers, these methods get higher accuracy when compared to the methods that do not account for outliers.

The evaluation results reveal that none of the considered forecasting methods is equally accurate and fast. The selection of forecasting methods should be based on the anticipated usage of the forecasts and on the characteristics of the forecasted metric. For instance, the user load forecasting method may differ by the endpoint which the load is directed at. In addition, services may also get their own tailored and fine-tuned forecasting models to extrapolate the load more accurately/faster.

7

Performance Modeling and Resource Allocations Derivation for Autoscaling the Multilayered Cloud Deployments

7.1 Filling the Gap between the Deployment and the Service Level

The objective of performance modeling in predictive autoscaling is to quantify the dependence between the scaling aspects of the deployment, e.g. resource limits for vertical scaling of containers or service instances count/virtual cluster sizes for horizontal scaling, and the quality metric characterizing the deployment, e.g. 99th percentile response time. Given a performance model, one can estimate by how much does the change in resource allocation affect the service level delivered to the end user. Without such a relation, it becomes next to impossible to argue about the desired deployment state to meet the service level objectives.

Let us look at the counterexample of threshold-based resource provisioning which is a default policy in reactive autoscaling. Does knowing that the virtual cluster's memory was utilized at 80% for the last 5 minutes indicate that the cluster should be scaled up? It might be a good idea if the cluster is small and each request consumes some feasible percentage of memory. However, if the cluster is already large (say, 1000 VMs) and the memory consumption does not exhibit much variability, then provisioning another tens or hundreds of VMs might be unnecessary.

Such static thresholds on low-level metrics, that barely represent the service level, are inaccurate by their very nature. The reason is in the *ratio-based implementation* of the provisioning decisions based on static thresholds. If, say, in our example the cluster with just 3 VMs shows 95% memory utilization and the threshold is set at 80%, then it may increase by 1 VM using the threshold-based rule ($\lceil 3 \cdot \frac{95}{80} \rceil$). Adding 1 VM will not result in boosting the bill by a large sum even if there is just a handful of requests coming in after that. In contrast, a virtual cluster with 1000 VMs with the same utilization and threshold would grow by 188 VMs ($\lceil 1000 \cdot \frac{95}{80} \rceil$). This orders of magnitude difference might be justified only in case of the request rate growth significantly outpacing that of the smaller deployments. This, however, might not

happen (e.g. the request rate stabilizes after pushing the utilization to this percentage), hence such simplistic threshold-based rules tend to overprovision the resources for some period of time as the scale down upon the utilization reduction will take some time. Thus, lacking the performance model is an extreme disservice to the application owner.

Having the relation between the managed deployment parameters and the service level metrics in a form of performance model, enables one to solve the reverse task that underlies predictive resource provisioning. One can use the derived relation to pick the deployment parameters in such a way that the estimated service level stays lower (response time) or higher (throughput) than the desired threshold. Constrained optimization techniques serve this goal by attempting to minimize the goal function, i.e. the derived relationship, subject to some constraints. In the following subsections we concretize this approach for application both for the vertical and multilayered horizontal autoscaling processes.

7.2 Resource Allocation in Vertical Autoscaling

7.2.1 Deriving SLO-compliant Pod Resource Limits: General Approach

Derivation of resource limits[1] for co-located containerized applications to meet SLOs is challenging due to several reasons. First, both the application logic and the technology stack might produce sporadic artifacts in the performance data. Second, accurate resource limits might only be identified with trial-and-error on all the potential deployment settings. Third, the effects of co-location might interfere with the collected service level indicators (SLI) in unexpected ways. Finally, SLOs should be met in all the co-located applications. To address these challenges, we use machine learning techniques for performance modeling.

Performance model is determined by two kinds of variables, viz, the target (predicted or output) variables and the model features (input). As target variables for prediction we select the 99%-tile response time (ms) and the throughput (requests per second, RPS). In contrast to the target variables, the input features can be split into three groups. First, **unmanageable features** representing the user load, denoted as *Concurrency_i* for the i^{th} application and measured in RPS. Second, **manageable features** representing the resource limits allocated to application pods: $mCPUs_i$ for the i^{th} application and measured in millicores (aka mCPUs or shares of processor time). Third, **partially-manageable features** representing the SLIs of other co-located applications (useful for modeling the effects of co-location).

We address the challenges of performance modeling in three steps. First, the anomalies, unexplainable with the available features, are removed from the data. Second, the prediction models, that relate available features (resource limits, request rates, and SLIs of other applications) to the SLIs, are learned automatically. Third, the resource limits are derived by solving a constrained optimization problem to maximize the service level ensuring that the predefined SLOs are met. The following subsections detail this approach and the experimental setting that it was developed for.

7.2.2 Dataset and the Experimentation Platform

To evaluate the performance modeling and resource allocation approach upon its design continuously, we collected **8,864** observations from a private Kubernetes-based cloud. The load was generated by a physical machine, which was not part of the deployment. It also recorded the SLIs. Multiple configurations of the

[1] `https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#requests-and-limits`

deployed applications were repeated 500 times, each time randomly selecting the available CPU resources and the level of concurrency (number of parallel requests) driving their load.

The cloud was hosted on a single physical machine with an Intel(R) Xeon(R) CPU E5-2670 @ 2.60GHz with 32 virtual cores, 256GB of RAM and running Ubuntu 18.04.1 LTS (Bionic Beaver). A Kubernetes private cluster with 8 virtual nodes was deployed on it using the Vagrant and Virtual Box script from Oracle[2]. Each node was a VMBox Virtual Machine running Ubuntu with 4 VCPUs and 15GB of RAM. The single-site single-machine configuration of the private cloud reduces the influence of the network effects on the model for the co-located applications.

Three deployments were started: **Nginx**, **Liberty** and **Guestbook**. **Nginx** (*App1*) exposes a Kubernetes Load Balancer service, which forwards incoming requests to one of the available replicas of an nginx:1.15.6 container. **Liberty** (*App2*) exposes a Kubernetes Load Balancer service, which forwards incoming HTTP(s) requests to one of the available replicas of a websphere-liberty:18.0.0.3-javaee8 Docker container. Its pods run a more complex stack of applications, i.e. each pod executes the Eclipse OpenJ9 Java Virtual Machine, which is a runtime for Java. On top of OpenJ9, the Java application Liberty profile is running, which is a Java EE application server. **Guestbook** (*App3*) is a three-tier Kubernetes tutorial application. Its PHP front-end enables users to view and add comments that are stored in Redis. A Kubernetes Load Balancer service is exposed to forward incoming requests to replicas of a Guestbook container.

7.2.3 Removing the Anomalies

Variation of observations is usually considered an asset when training a machine learning model [142]. However, if a dataset lacks the features explaining the anomalous samples, then the accuracy of the model decreases. In the following paragraphs, we assess the impact of anomalous samples in the collected dataset on predicting the SLIs and envisage an approach to prune the anomalies to improve the performance model.

Preliminary analysis of distributions for the 99%-tile response time and throughput points out anomalies among the observations of the 99%-tile response time for application 2, **Liberty** (cf. Figure 7.1). Around **8.2%** of all observations of the response time are higher than **1.5**s, which in practice means failed requests. Although observations above 1.5s can be used as a threshold to identify the anomalies, there are also anomalies that might impact the model quality and still be below the threshold.

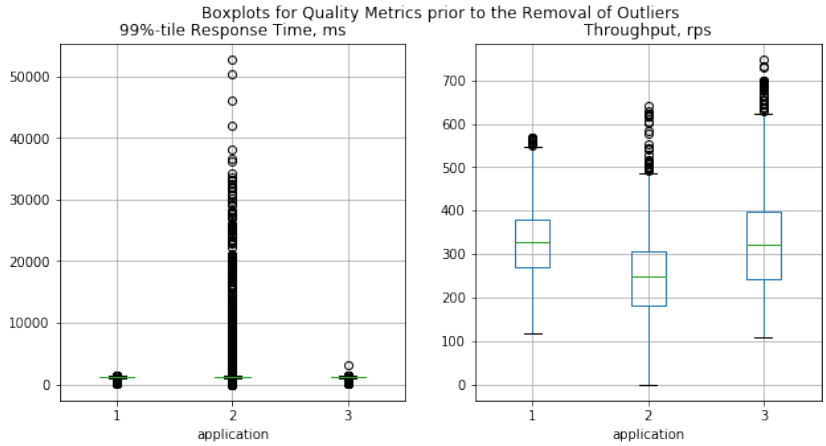


Figure 7.1.: Distribution of SLI values in the raw dataset; five number summary.

[2] `https://github.com/oracle/vagrant-boxes`

The influence of anomalies on the accuracy is tracked by removing an increasing fraction of anomalies from the dataset and calculating the R^2 score of the regression model, e.g. Lasso regression. Table 7.1 shows that the most significant improvement in 10-fold cross-validated R^2 scores for the 6-target Lasso prediction model of order 2 is for a fraction of anomalies between **0.10** and **0.12**. Anomalies are identified with the Isolation Forest algorithm [143]. Setting the fraction of removed anomalies to **0.11** causes the distribution of the 99%-tile response time for the second application (**Liberty**) to approach the normality (cf. Figure 7.2).

Fraction of removed observations	*Averaged R^2 Score*	*R^2 Score variance*
0.05	0.43	0.0127
0.06	0.51	0.0083
0.07	0.57	0.0054
0.08	0.58	0.0050
0.09	0.59	0.0044
0.10	**0.61**	**0.0036**
0.11	**0.61**	**0.0036**
0.12	**0.61**	**0.0038**
0.13	0.60	0.0037
0.14	0.59	0.0035
0.15	0.59	0.0033

Table 7.1.: Impact of the fraction of removed anomalies on the R^2 score of the 6-target Lasso regression.

Figure 7.2.: Distribution of SLI values with 11% of anomalies removed using Isolation Forest.

Anomaly detection resulted in observations of two types being labeled as anomalies: observations with very low response time for Application 2 and observations with high response time for the same application. In the dataset, there are **710** observations that took longer than **1,521** ms to complete. This threshold is the maximal response time after filtering out **11%** of anomalies found. The leftover observations cover **72.8%** of all the anomalies. These anomalies are important to model as it is not known whether low response times that they represent are due to the response being completed with error.

7.2.4 Prediction Models

The models discussed below were evaluated using the 10-fold cross-validation and max iterations number of **10,000**. The following notation is used: **RT**, 99% response time; **T**, throughput; $\overline{R^2}$, average R^2 score; and $V[R^2]$, variance of R^2. The evaluation was done on a laptop with Intel Core i7-6700HQ (2.60 GHz) and 16 GB of RAM under 64-bit OS Windows 10 HE. The scripts were run using Jupyter Notebook.

Single App-Single Target Model. This model predicts each SLI individually. Thus, for three applications with SLI of 99%-tile response time and of throughput there are six such models in total. The advantage of these models is in their high discriminatory power. The use of these models is limited because they may lead to diverging results. Let us look at two variaties of these models.

Models without target variables as predictors. Table 7.2 shows that lasso regression models have good potential for predicting the throughput of some applications. An additional advantage of these models is the short fitting time: even for the model of degree 3, the fitting time stays below 1 minute.

Application	*SLI*	*Degree = 1*		*Degree = 2*		*Degree = 3*	
		$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$
Nginx	RT	0.43	0.0022	0.56	0.0029	0.57	0.0033
	T	0.90	0.0008	0.93	0.0007	0.94	0.0007
Liberty	RT	0.60	0.0199	0.60	0.0177	0.61	0.0173
	T	0.65	0.0177	0.69	0.0182	0.69	0.0191
Guestbook	RT	0.45	0.0035	0.58	0.0037	0.59	0.0045
	T	0.92	0.0004	0.95	0.0003	0.96	0.0002

Table 7.2.: Evaluation results for the single app-single target models.

Models that include target variables as predictors. Including target variables as predictors improves the accuracy of the single app-single target models. Each model includes only those target variables as predictors that were not used as their own target. The average R^2 score of the the degree-2 model does not fall below **0.80** for any target (Table 7.3). Nevertheless, this improvement does not yield consistent resource limits. Moreover, with the introduction of target variables as predictors, the fitting time increased up to 1 min for degree 2. This rendered the use of higher-degree models unfeasible for the most of the tasks requiring fast resource allocation to co-located applications.

Application-wise Models. A single multi-SLI prediction model per application is able to produce resource allocation that is consistent for all the application SLOs. Such models will still fail in supporting consistent resource allocation across multiple applications due to not accounting for the other co-located applications in the model. Nevertheless, the use of such models might be justified since per-application prediction models can easily scale. Again, let us consider two kinds of such models.

Application	*SLI*	*Degree = 1*		*Degree = 2*	
		$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$
Nginx	Response time	0.80	0.0033	0.80	0.0073
	Throughput	0.95	0.0001	0.99	0.0000
Liberty	Response time	0.76	0.0037	0.84	0.0021
	Throughput	0.89	0.0019	0.91	0.0010
Guestbook	Response time	0.80	0.0028	0.81	0.0023
	Throughput	0.95	0.0001	0.99	0.0000

Table 7.3.: Evaluation results for the single app-single target models with target variables as predictors.

Application-wise models without target variables as predictors. Table 7.4 shows that the combined prediction of two SLIs per application loses accuracy of individual predictions for throughput. In essence, each prediction model's R^2 score is limited by the lowest R^2 score of the target SLIs, what can be inferred by comparing these results with Table 7.2.

Application	*Degree = 1*		*Degree = 2*		*Degree = 3*	
	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$
Nginx	0.47	0.0014	0.59	0.0023	0.60	0.0025
Liberty	0.74	0.0093	0.75	0.0087	0.76	0.0081
Guestbook	0.51	0.0025	0.62	0.0028	0.63	0.0033
	$\bar{t} = 1.44$ min		$\bar{t} = 6.44$ min		$\bar{t} = 68.36$ min	

Table 7.4.: Evaluation results for the application-wise models.

The joint consideration of R^2 scores with the average models fitting times for different degrees (provided at the bottom of the table) indicates that it does not make sense to increase the degree beyond 2 – the increase of R^2 score by 0.01 is barely worth an additional hour of fitting time.

Application-wise models that include target variables as predictors. Including target variables as predictors again boosts the predictive power. As shown in Table 7.5, the R^2 score stayed above **0.8** for models with degree of 2. This is never achieved for application-wise models without target variables as predictors, even though the maximal degree of a model is higher. The average fitting time higher than two hours makes these models impractical.

Application	*Degree = 1*		*Degree = 2*	
	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$
Nginx	0.81	0.0027	0.81	0.0071
Liberty	0.79	0.0046	0.83	0.0032
Guestbook	0.82	0.0021	0.83	0.0017
	$\bar{t} = 5.79$ min		$\bar{t} = 130.91$ min	

Table 7.5.: Evaluation results for the application-wise models with target variables as predictors.

SLI-wise Models can span multiple applications, hence they can be considered as globally consistent. As each such model uses only one SLI as a target, the resulting resource allocations (limits) for pods acquired using these models will be inconsistent for a single application. Such models could scale well for more SLIs in case other mechanisms to resolve the inconsistency in resource limits are in place.

SLI-wise models without target variables as predictors. Table 7.6 indicates that the 99%-tile response time is not well-explained by the features; in contrast, throughput is well-described. This follows from the long-tail distribution of response times in comparison to throughput which is closer to normal.

SLI (target)	*Degree = 1*		*Degree = 2*		*Degree = 3*	
	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$
Response time	0.41	0.0056	0.54	0.0079	0.57	0.0066
Throughput	0.83	0.0034	0.86	0.0035	0.87	0.0035
	$\bar{t} = 1.62$ min		$\bar{t} = 6.79$ min		$\bar{t} = 60.38$ min	

Table 7.6.: Evaluation results for the SLI-wise models.

SLI-wise models that include target variables as predictors. Inclusion of the target variables as predictors does not help as is demonstrated by the average R^2 score in Table 7.7 being only slightly better than that of Table 7.6. The significant overhead in fitting time renders the use of such a model impractical.

SLI (target)	*Degree = 1*		*Degree = 2*	
	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$
Response time	0.48	0.0094	0.59	0.0093
Throughput	0.88	0.0013	0.93	0.0008
	$\bar{t} = 5.71$ min		$\bar{t} = 134.70$ min	

Table 7.7.: Evaluation results for the SLI-wise models with target variables as predictors.

All-targets Model. Although the all-targets model allows to predict all targets at once (both SLIs for all three applications in our case), it has two significant drawbacks. First, this model scales poorly as new SLIs and applications will make it more complex and will require longer fitting. Second, its accuracy is sensitive to the changes even in a single parameter.

As the model under discussion already predicts all the possible target variables, there is only a single case for its evaluation whose results are presented in the Table 7.8. Poor predictive properties of the model can be explained by the presence of the 99%-tile response time for all three applications among the target variables. It was previously shown to be the problematic case for Lasso regression models when considered without other target variables as predictors.

Degree = 1		*Degree = 2*		*Degree = 3*	
$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$	$\overline{R^2}$	$V[R^2]$
0.50	0.0020	0.61	0.0036	0.63	0.0034
$\bar{t} = 2.16$ min		$\bar{t} = 7.56$ min		$\bar{t} = 62.57$ min	

Table 7.8.: Evaluation results for the all-targets model.

Performance Model Selection. Application-wise models with target variables as predictors are the most well-balanced models for the performance prediction on the collected dataset. All these models simultaneously predict both 99%-tile response time and throughput, which makes them both consistent for resource allocation on the level of the application and scalable. With the small difference in average R^2 estimates for degree 1 and 2 in Table 7.5 as well as a large difference in fitting times, one could consider using the model of degree 2 while at the same time reducing the number of iterations for the model fit and/or loosening the termination condition. This type of model was selected for the SLO-compliant resource allocation.

7.2.5 SLO-compliant Resource Allocation

Once we have the performance models that relate the resource allocation and the user load to the SLIs while being purified of anomalies, the next step is to use these models to acquire resource limits based on the required SLOs. This step requires to solve a multi-objective constrained optimization problem. It could be formulated both as an integer programming problem as well as a continuous optimization problem. In what follows, a corresponding continuous optimization problem is formulated. Its objective function captures the cumulative deviation of the predicted SLIs (response time, throughput) from their corresponding thresholds.

Consider n co-located applications as well as the throughput SLO (T_i) as $T_i \geq T_i^{(SLO)}$ and the 99%-tile response time SLO (RT_i) as $RT_i \leq RT_i^{(SLO)}$ of the i^{th} application. The limitation on the available resources: $\sum_{i=1}^{n} mCPUs_i \leq \sum_{j=1}^{m} cores_j \cdot 1000$ where m is the number of hosts (VMs or physical servers) and $cores_j$ is the number of cores available at j^{th} host. The number of cores is normalized to milliCPUs by multiplying by 1000 (homogeneous case with hosts equipped with the same cores). Resource limits on milliCPUs for all the applications under consideration can be grouped into the single vector

$\mathbf{mCPUs} = [mCPUs_1, mCPUs_2, ...mCPUs_n]^T$. SLI values predicted with the trained Lasso models for the above vector of resource limits are denoted with $\hat{RT}_i(\mathbf{mCPUs})$ and $\hat{T}_i(\mathbf{mCPUs})$ for the 99%-tile response time and throughput respectively. The following objective functions are formulated for each application:

$$f_i^{(RT)}(mCPUs) = \frac{\hat{RT}_i(\mathbf{mCPUs})}{RT_i^{(SLO)}} \tag{7.1}$$

$$f_i^{(T)}(mCPUs) = \frac{\hat{T}_i(\mathbf{mCPUs})^{-1}}{T_i^{(SLO)}} \tag{7.2}$$

As we see in Eq. 7.1, the objective function for the response time favors lower predicted response times when its value is minimized. In contrast, the minimization of object function for throughput (cf. Eq. 7.2) happens when the predicted troughput increases. We use these simple objective functions to build the multi-objective cost function for application-wise, SLI-wise, and all-targets options respectively. We do so by taking a product of their values:

$$g_i^{(App)} = f_i^{(RT)}(\mathbf{mCPUs}) \cdot f_i^{(T)}(\mathbf{mCPUs}) \tag{7.3}$$

$$g_j^{(SLI)} = \prod_{i=1}^{n} f_i^{(SLI)}(\mathbf{mCPUs}), SLI \in \{RT, T\} \tag{7.4}$$

$$g^{(All)} = \prod_{i=1}^{n} f_i^{(RT)}(\mathbf{mCPUs}) \cdot f_i^{(T)}(\mathbf{mCPUs}) \tag{7.5}$$

Thus, the problem is formulated as the following multi-objective constrained optimization problem:

$$\begin{cases} \mathbf{mCPUs}_i^* = \arg\min g_i^{(App)} \\ ||\mathbf{mCPUs}_i^*||_1 \leq \sum_{j=1}^{m} cores_j \cdot 1000 \\ \hat{RT}_i(\mathbf{mCPUs}_i^*) \leq RT_i^{(SLO)} \\ \hat{T}_i(\mathbf{mCPUs}_i^*) \geq T_i^{(SLO)} \\ mCPUs_k \geq 100 \, \forall \, mCPUs_k \in \mathbf{mCPUs}_i^* \end{cases} \tag{7.6}$$

The solution of the optimization problem as specified in Equation 7.6, yields potential discrepancies between optimal resource allocations acquired for each application. The only multi-objective cost function that would avoid the discrepancy is $g^{(All)}$, but its use is infeasible due to a linear growth in size the vector **mCPUs**, and the poor predictive properties of the all-target model (Table 7.8). Next, we introduce an approach to alleviate the discrepancy between the solutions provided by application-wise optimization.

Pure Continuous Constrained Optimization. The nonlinear integer programming formulation of the same problem is NP-hard [144]. The most straightforward and feasible approach to the solution of the optimization problem in Equation 7.6 is to solve it as a continuous constrained optimization problem. For that, one can use trust region-based methods for nonlinear constrained optimization [145]. Due to the dependence of the form of SLI prediction models on the collected data, the use of the 2-point numerical Jacobian approximation and symmetric-rank-1 (SR1) Hessian update strategy in optimization is justified. The results of the continuous constrained optimization acquired for each application are summarized (e.g. averaged) and rounded or made integer in some other way.

An initial guess of the parameters to be optimized may severely impact the optimality of the solution. To overcome this issue, we performed continuous constrained optimization multiple times for random init values $\mathbf{mCPUs_0}$; the final result is a rounded median of the results acquired in these random runs. This is an adapted version of the Basin-Hopping algorithm [146]. Preliminary tests of continuous optimization showed that **15** random runs are enough to acquire stable optimization results.

Constrained Optimization with Limited Brute Force. The accuracy of the solution provided by the pure continuous constrained optimization might be improved with a brute force search over a neighborhood of the solution. The following parameters can be specified for such an approach: 1) the search step Δ, e.g., 50 mCPUs, 2) the number of search steps ϕ in the direction of increase and decrease of the proposed solution, which can be approximated by:

$$\phi = \left\lceil \frac{\sum_{j=1}^{m} cores_j \cdot 1000}{2 \cdot n \cdot \Delta} \right\rceil \tag{7.7}$$

Considering $\mathbf{mCPUs}^*$ as the result of continuous constrained optimization (averaged and rounded over all applications), the search space is limited by: $\mathbf{mCPUs}^* \pm [(\phi \cdot \Delta)_{\times n}]$.

Evaluation. The baseline for evaluating the optimality of the solutions produced by the described optimization approaches is the result of a brute force search with a grid cell size of $10 \times 10 \times 10$, **BF-10**. Brute force with coarse-grained grid (cell size of $50 \times 50 \times 50$), **BF-50**, was used to highlight the trade-off between the quality of the acquired parameters' vector and the running time of brute force search depending on the cell size. Both brute force cases use a pre-trained all-target model (discussed before) to acquire an SLI prediction for subsequent global optimization. The results acquired with brute force were compared to the results of the continuous optimization approach, **CO**, and the continuous optimization with limited coarse-grained brute force search on top, **Hyb**.

In evaluation, the following parameters were used: $RT_i^{(SLO)} = 800$ and $T_i^{(SLO)} = 200 \forall i \in 1,2,3$. The joint limit on CPU was **3000** mCPUs. Concurrencies and SLIs were generated randomly for each test in allowed boundaries; **10** tests were conducted for both the soft and hard limit cases.

The evaluation results are presented in Table 7.9. The Euclidean distance between the optimization result for the considered approach and the result of the fine-grained brute force (**BF-10**) is in the column *Distance*. A high execution time of 2–4 minutes of the fine-grained brute force prohibits its use for dynamic reconfiguration of the co-located applications. The coarse-grained brute force search gives results that are close to that of the fine-grained approach, but significantly faster (only 1–2 seconds are required). Despite the observed small duration of the coarse-grained brute force search **BF-50**, it scales poorly for environments with a high number of co-located applications (e.g. 50-100).

Methods that are based on *continuous optimization* using trust regions show similar execution time but significantly different quality. Adding subsequent coarse-grained brute force search on a limited neighborhood improves the accuracy (the distance from the baseline decreased more than 2-fold). The main contributors to the execution time of 1–1.5 minutes for these methods were the multiple runs of the optimization for random values of the initial parameter vector.

7.2.6 Validation

Validation Test Design & Settings. Validation of SLO-compliant resource allocation includes two steps. First, a **Preliminary Validation Test (PVT)** acquires the SLIs values used as features in application-wise

Limit	*Distance (median)*			*Execution Time, s (median)*			
Type	*BF-50*	*CO*	*Hyb*	*BF-10*	*BF-50*	*CO*	*Hyb*
Soft	56.6	1229.2	515.9	209.2	1.68	69.1	71.4
Hard	40.0	1310.6	515.8	121.8	0.95	44.0	45.2

Table 7.9.: Quality of solutions by the SLO-compliant resource allocation approaches.

prediction models. In a production environment, these values should be collected dynamically and applied in the scope of a continuous resource allocation process. The test uses fixed request rates and an initial guess for resource allocation; a load test is repeated 16 times with Apache **ab**. Second, an **Evaluation Validation Test (EVT)** wraps the testing similar to **PVT** but with the following major changes: 1) SLIs prediction models are acquired by applying the application-wise lasso regression on the same concurrencies as in **PVT** with medians of SLIs values received as a result of **PVT**; 2) continuous constrained optimization with limited brute force (**Hyb**) is applied to the acquired models and the desired SLOs.

Validation test was conducted both for hard and soft pod resource limits on the Kubernetes cluster. The settings for **PVT** are given in Table 7.10. The settings for **EVT** are shown in Table 7.11 (partially acquired as the results of **PVT**). The same SLOs were set for all applications during **EVT**: $RT_i^{(SLO)} = 800$ ms and $T_i^{(SLO)} = 200$ RPS $\forall i \in \{1,2,3\}$. These SLOs were selected as realistic values based on the collected dataset.

Limit	*Concurrencies*			*Resource Limits*		
Type	*App1*	*App2*	*App3*	*App1*	*App2*	*App3*
Soft	58	61	53	200	550	120
Hard	58	63	75	300	500	100

Table 7.10.: Settings for the Preliminary Validation Test

Limit	*Concurrencies*			*Resource Limits*			*99%-tile Response Time*			*Throughput*		
Type	*App1*	*App2*	*App3*	*App1*	*App2*	*App3*	*App1*	*App2*	*App3*	*App1*	*App2*	*App3*
Soft	58	61	53	1250	500	130	414.0	466.0	421.0	266.9	254.3	241.2
Hard	58	63	75	200	550	120	562.0	627.0	571.5	231.1	227.6	292.9

Table 7.11.: Settings for the evaluation validation test. Resource limits result from the proposed approach.

Validation Results. The conducted test proved the validity of the resource allocation approach with at most two cases out of 16 trials violating SLOs as shown in Table 7.12 (marked as N_v in table). Almost all the SLO violation cases (9 out of 10) are attributed to the SLO on the 99%-tile response time, which is more unstable than the throughput. Among these violations, 4 cases were identified for the second application **Liberty** data that contained anomalies due to garbage collection invoked periodically, which is also pointed out by related work [147, 148]. For the selected test environment, soft and hard limits on Kubernetes pods' resources do not seem to significantly influence the amount of SLO violations.

The conducted test demonstrated the validity of the designed approach to SLO-compliant resource allocation. The presence of a few SLO violations points out the necessity to include additional features in the SLI prediction models that should capture unique properties of particular applications and their runtimes, such as garbage collection delays.

99%-tile Response Time					
App1		*App2*		*App3*	
Value	N_v	*Value*	N_v	*Value*	N_v
With Soft Limits					
524.9 ± 142.7	1	568.8 ± 130.2	**2**	537.9 ± 149.6	**2**
With Hard Limits					
507.9 ± 108.4	1	705.2 ± 592.8	**2**	294.5 ± 109.3	1
Throughput					
App1		*App2*		*App3*	
Value	N_v	*Value*	N_v	*Value*	N_v
With Soft Limits					
264.6 ± 5.4	0	252.2 ± 4.1	0	237.9 ± 4.6	0
With Hard Limits					
231.9 ± 5.3	0	225.1 ± 9.9	1	294.5 ± 7.2	0

Table 7.12.: Results of the validation test.

7.3 Amending Resource Allocation for Horizontal Scaling of Multilayered Deployments

7.3.1 Comparison to the approach for vertical scaling

To a large extent, devising the count of scaling entities for the purpose of horizontal scaling follows the steps of the approach laid down for vertical scaling. In particular, it again starts with deriving the performance model that maps the mix of managed and semi-managed features onto the quality metric. An important difference to the same step in the vertical scaling scenario is that this time we add the quantity of scaled virtual entities (e.g. pods or virtual machines) into the features set. This allows to get the count of desired replicas directly. The next step repeats the one for the resource allocation in the vertical scaling scenario, i.e. we search for the count of scaled entities that allows to meet the service level objective as predicted by the performance model.

One important limitation that the attentive reader might notice is that taking VMs count as an input feature severely limits the validity of the whole approach. The problem with using the VMs count is that it forces the performance model to use only the VM types that were present in the dataset. To achieve the model validity, one is either required to present it all the VM types or to build yet another model that can convert performance results of one VM type into the other. The kind of application and the load pattern should also be taken into account when devising the performance model. Combined, devising a credible performance model for the multitude of VMs types and load patterns is a Herculean task. Hence, one needs a more universal abstraction than a virtual machine to improve on perfomance modeling's feasibility frontier. We propose to use an abstraction of *service instances* to build such a model. In practice, a service instance is likely to run in a pod replica that is placed either on a virtual or on a physical node, hence we get a credible unit for horizontal scaling.

Employing another scaling unit does not free us from the need to consider VMs in the multilayered deployment. To determine the VM count in the cluster, we translate the resource requirements of service instances

into the desired VM count capable of hosting at least a single instance. This is done by monitoring the average resource requirements of services and trying to fit them into various VM types given the information on available VM resources.

7.3.2 Adjusting ML-based performance model for horizontal autoscaling

Horizontal scaling of a microservice application as a whole may result from scaling an arbitrary number of microservices. Having a performance model that maps replicas counts of all the microservices right into the delivered service level might appear to be desirable. At a closer look it has many downsides. The first disadvantage of this approach is that training such a model takes a lot of data and time. Another drawback, which is an opposite side of such model's comprehensiveness, is the lack of specificity. Instead of tracking the impact of each particular service scaling on the overall application performance, the application-wide model operates on the level of service instances combinations. Hence, instead of scaling on a per-service basis, the joint application performance model will force one to scale multiple services at the same time. Lastly, large application-wide model scales poorly when deriving the desired service instances counts for all the services at the same time with optimization techniques. The challenge is that the dimensionality of the optimization space grows with the count of services in the application.Having identified these drawbacks, we decided to pursue horizontal scaling on a *per-service basis*.

Practical systems, such as Kubernetes, approach autoscaling similarly – scaling is performed on the level of a service. Additional practical consideration in favor of this approach is that there is no single point of failure in the application. Even if an autoscaler (embodied in a per-node **kubelet** of Kubernetes) for a particular service fails, this does not impact autoscalers for other services. Having decentralized autoscaling is also beneficial through the reduction in communication. If there is a need for a particularly accurate model, one can use *sandboxing* of services to derive accurate per-service performance model which is way more costly and complicated to achieve for the whole application [97].

We adjust the performance model from the previous section on vertical autoscaling as follows. For each service s_i of a scaled application, we build a performance model that establishes the relation between the service instances count as well as some other parameters and the response time (cf. Eq. 7.8 for the performance model for the response time).

$$\hat{RT} = f(L, \mathbf{n_{s_i}}, Mem_{s_i}, vCPU_{s_i}, ...) \tag{7.8}$$

The key difference in comparison to the performance models for vertical autoscaling is in the selection of input parameters. In particular, pod resource allocation limits are substituted by the *service instances count*, $\mathbf{n_{s_i}}$. We are allowed to make this transition in the features set under the assumption that every service instance has the same resource allocation limits. We are also adding current resource utilizations to the feature set. All the other parts of this ML-based model remain the same.

The particular form of function f from Eq. 7.8 only matters for the overall accuracy of the approach and is discussed in more details in Chapter 10. Similarly to the ML-based models from the previous section, this function is a regression of several features characterizing the state of the service s_i (e.g. instances count $\mathbf{n_{s_i}}$, memory usage by all the instances of this service Mem_{s_i}, vCPU usage by all the instances of this service $vCPU_{s_i}$) and variables characterizing the load (e.g. the request rate L, RPS) onto a single application-level quality metric such as some percentile of the response time, RT. One can model this relation as being linear or nonlinear, but nonlinear models, e.g. deep learning-based ones, tend to reflect resource saturation effect better.

The optimization problem for finding the service instances count that is required to fulfill the service level objective is similar to that specified in Equation 7.5 for vertical scaling. The difference is in the parameter that we optimize for, namely, the service instances count, $\mathbf{n_{s_i}}$, and in the objective function that is simply the ratio of the predicted response time to the service level objective (commonly around 100-300 ms) as in Equation 7.1. The optimization technique used to solve this problem remains the same.

Having devised a method to find the number of service instances (pods) with the same resource allocation limits, we still need to determine the type and the number of virtual machines needed to house these service instances. In the next subsection we propose a simple technique to bridge this gap to autoscaling of multilayered cloud deployments.

7.3.3 Enabling Resource Allocation for Multilayered Deployments

In order to scale the deployment horizontally on multiple resource abstraction layers (pods with resource limits and virtual machines), one needs to *express the scaling entities on one layer in terms of scaling entities on another layer*. In our multilayered setting, a service instance in a pod replica is considered to be a scaling entity for the application layer. Virtual machine (a single instance) of some type is, in turn, a scaling entity on the virtual cluster layer underlying the deployed application.

The point of intersection for these scaling entities is in service instances consuming some amount of virtualized system resources provided by the virtual machine that the service instance is executed on. Resource utilization itself tends to be rather dynamic and is therefore a poor common ground for considering how many VMs are required to run the given number of service instances ahead of time. However, one might notice that the production-grade container orchestration systems such as Kubernetes demand to set the so-called *resource allocation limits* for the containers/pods. These limits are expressed in terms of virtual resources, e.g. the amount of RAM and the amount of vCPU in terms of millicores[3]. In turn, cloud services providers size their VM types (flavors) in the same way, i.e. using the values of virtual resources allocated to VMs instances to distinguish between them. For example, **t3.nano** by AWS offers 2 vCPUs and 0.5 GiB memory, whereas **t3.2xlarge** by the same provider offers $\times 4$ more vCPUs and $\times 64$ more RAM. Thus, one may express CPU resource limits of a pod with a service instance running in it in terms of vCPUs of a VM. With rare changes in the nomenclature of VM types, there is quite a limited space of options that one needs to consider in order to find an optimal resource allocation on the virtual infrastructure layer using the desired number of service instances and their resource limits as a guidance. Below, we propose a simplified approach to calculate the number of VMs required to accommodate the desired number of service instances. A more accurate approach would try to establish the correspondence between the VMs' sizings and the pod resource limits empirically. However, we found this path to be really tedious to pursue because of the multitude of applications and their resource consumption behaviors. Hence, the below approach is rather straightforward, but it is proven to work just fine in Kubernetes.

Building on top of the above intuition, we can now use the calculated cumulative resource demands by the number of service instances to determine the number of VM instances based on the amount of virtual resources that the VM type has to offer. Let's say that we need to accommodate n_{s_i} instances of service s_i in pods with vCPU limit set to $r_{s_i}^{(vCPU)}$. To determine the required instance count N_j of a VM of type j with $R_j^{(vCPU)}$ vCPUs allocated to it, we may use the following simple formula:

$$N_j = \left\lceil \frac{n_{s_i} \cdot r_{s_i}^{(vCPU)}}{R_j^{(vCPU)}} \right\rceil \tag{7.9}$$

[3] https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

Since the ratios of resource limits for different system resources (vCPU, memory, etc) to the VM type's resource may differ, we compute the count of VMs for each type of resource, i.e. $N_j^{(vCPU)}$, $N_j^{(mem)}$, and so on. Then, we select the maximal value. Once the similar procedure is done for all the VM types for a given cloud services provider (or multiple providers), we fix the interval of time, that the deployment is anticipated to last for, and use the price per unit of time, p_j, for each VM type to compute the cost of every allocation option:

$$C_j^{(T)} = N_j \cdot T \cdot p_j \tag{7.10}$$

Lastly, the cheapest option is selected for the deployment.

There are multiple other ways to determine the types and the number of virtual machines to accommodate the desired count of service instances with the given pod resource limits. For instance, one may attempt to compose a VM cluster of different VM types. This is particularly useful, when a VM in a homogeneous cluster has spare resources that are not taken by the pod. Then, such a VM might be substituted for another one with smaller amount of resources given to it. In addition, as a pre-step, one may try to combine several different services into an allocation group such that they balance the resource requirements of each other, e.g. a two-service allocation group with one service being CPU-intensive and another one being memory-bound. When placed on the same VM, these services will make most of the resources allocated to the VM in the data center. However, such optimizations usually require some additional runtime resource utilization information.

These and other optimizations for services placement on VMs are vastly discussed in the research literature and are not given in this thesis due to the page limit. It is only important to remember that the resource limits set by the application owner/administrator and the desired count of service instances are sufficient to find the right type and number of VMs to accommodate the application in the multilayered setting.

8

Insights into the Microservice Application Topology for better Autoscaling

8.1 Topology-Awareness for Improving Predictive Autoscaling

The role of application topology in how the application should be scaled is substantial. Applications with long request traces are more likely to exhibit unbalanced scaling, i.e. the services along the request path are scaled unevenly. Even if a service is able to process the incoming load under the strict SLOs, the downstream service could fail to scale accordingly and thus will not be able to process the output of the previous one. It is extremely likely that the independent scaling of services will create *a structural bottleneck*. The study of application topology helps in identifying such bottlenecks and designing cascading autoscaling schemes that balance the capacity along the request path. From the scaling prospective, the most important information, that could be extracted from the application topology, is as follows:

- **Presence of the bottlenecks along the request propagation path**. The bottleneck service is usually characterized either by a) insufficient system resources, or b) a high fan-in factor, i.e. a high number of incoming connections from other services. Without scaling up the bottleneck service, the scale up for the upstream services becomes useless. The discussion in this section is mostly focused at the services that become bottlenecks due to a high fan-in factor. The bottlenecks formed by the insufficient resource allocation in multi-tier applications are extensively studied in the research literature [149, 84]. In addition, the dynamic performance bottlenecks largely depend on the unique conditions that the application is in, e.g. performance of the underlying physical machines and the optimality of the business logic implementation. Studying the application topology may reveal such structural bottlenecks to scale them accordingly to minimize the requests waiting time in services' buffers.
- **Fan-out to fan-in ratio**. Application logic determines requests propagation paths. A single incoming user request may trigger multiple subsequent requests to be processed before issuing the response. Corresponding requests traces may exhibit high branching factor. The count of incoming service connections is usually called a fan-in factor. Similarly, the count of outgoing service connections is called a fan-out factor. The ratio of fan-out to fan-in of a service determines how actively is this service involved into the communication with other services. The transactional model adopted in the

interactive applications forces the incoming request to result in at most one request *on each outgoing connection* (retries are not taken into account). This means that the static application topology can be an asset in understanding the runtime requests paths. Fan-out to fan-in ratio is a fundamental indicator of how the resource consumption and the performance of a service in the request path relates to the performance and the resource consumption of the connected services.

- **Balanced clusters of services that can be scaled as a group**. Often, software architects and DevOps engineers try to balance the deployment to minimize the resource waste. Therefore, a group of services and its placement may already be composed to even out the differences in the workload on request processing. Identifying manually tuned service groups may enable coarse-grained autoscaling, when, instead of scaling a service, a group of services is scaled. Even without manual efforts to designate such service clusters, the execution-time performance data may be used to detect such groups.

Application topology thus plays an important role in how effective predictive autoscaling will end up being. For example, if the capacity and performance relations between the services are known, one can provision the capacity required to process the request in the upstream services while the request has not yet reached them. This is especially promising when meeting tight millisecond-scale SLOs.

8.2 Network Theory Essentials

Network theory offers a rich mathematical framework to analyze the complexity of real-world distributed systems like the Internet [150]. A network is a graph with labelled vertices and/or edges.

Network theory offers methods and abstractions to characterize the importance of nodes in the network (topology). The importance could be denoted differently, but the most common way is to associate the number of connections with a node's importance, i.e. the more connections the node has, the more important it is. The identification of such nodes is addressed by the *centrality indices* [151]. Degree centrality is one of the simplest centrality measures and is defined as the number of links incident upon a node v: $C_D(v) = deg(v)$. This measure characterizes an immediate importance of the node.

Degree distribution is a probability distribution of degrees in the network that describes the structural properties of the whole network in the compressed form. Degree distribution shows the frequency of encountering the nodes with a particular degree in the network – different degree distributions likely identify different topologies. For instance, if the degree distribution has a long tail for higher degrees, then the network contains only a few nodes that have connections count to other nodes. This can have significant implications in such cases as developing a network structure that scales evenly along the request propagation path.

The degree distribution of a network can be approximated with a mathematical model. Such approximations allow in-depth studies of networks' properties. For example, one of the most common types of networks is a scale-free network. It follows the following probability distribution in degrees of its vertices: $P(d) \propto d^{-\alpha}$. Here, the fraction of nodes with d connections is defined as $P(d)$. It drops exponentially with the growth of the degree (α is usually between 2 and 3).

Various models exist to describe the properties of the networks and to generate new ones. In particular, scale-free networks are best described by the Barabási-Albert (BA) model that uses a preferential attachment method to generate networks with a power-like degree distribution [152]. Random graph models such as Erdős-Rényi (ER) model and Watts-Strogatz (WS) "small world" model generate networks with other degree distributions. The degree distribution analysis yields valuable insights into the topology of cloud applications allowing one to design balanced scaling techniques.

8.3 Dataset

To conduct our study, we collected a dataset of 137 Docker Compose configuration files sampled from 107 Github repositories [153]. The collected configuration files represent a variety of applications, including web-shops and web-portals, cloud platforms for IoT, technology stacks, etc. These YAML files define the start up sequence of microservices via special keywords, such as *depends_on*, *links*, or *external_links*[1]. This formal specification allows to create a service-dependency graph. To ensure that the analysis reflects the real-world applications as much as possible, applications with three or fewer connected microservices were excluded from it. After filtering, 103 Docker Compose configuration files with the total of 826 services remained with at least 26% of the applications containing more than eight microservices.

8.4 Degree Distribution Analysis Technique for Small Networks

The dataset analysis starts with producing an adjacency matrix representation of the underlying directed graph for each the Docker Compose configuration file. These matrices are used to identify topological patterns using the degree metric. Preliminary visual inspection pointed at three candidate degree distributions in the dataset (N_d is the number of nodes/services of degree d):

- **Uniform distribution**. For degree d in a range $[a,b]$, $N_d = const$; $N_d = 0$ otherwise.
- **Power law (Pareto) distribution**. For a degree d, $N_d \propto d^{-\alpha}$.
- **Normal distribution**. For a degree d, $N_d \propto e^{-\frac{(d-\mu)^2}{2\sigma^2}}$.

With a maximum of 24 vertices in the largest microservice application, determining the form of the degree distributions with statistical tests is prone to be inaccurate [154]. A network may be attributed to several distribution types. To improve the quality of such tests, a hybrid approach is devised. It combines statistical tests with the heuristic based on the form of the distribution curve. Preliminary tests on randomly generated distributions demonstrated high inaccuracy of the statistical tests for applications with a number of services less than six. Therefore, a fallback heuristic is applied when a graph has fewer than six vertices or when the corresponding statistical test is inapplicable to the given distribution. The designed heuristics are as follows.

Uniform distribution heuristic. A small number of distinct degrees in graphs makes the direct application of uniform distribution tests impractical. It is possible to transform the data s.t. the statistical tests would provide meaningful results. First, the initial degree distribution is transformed into a histogram. Following, the Pearson's chi-squared test is applied to test the degree distribution based on a Monte Carlo test with 500 replicates [155]. The value of 500 replicates was determined by conducting multiple tests on randomly generated distributions. The fallback heuristic for uniform distribution checks the single outcome not covered by the statistical test: *whether all the vertices have the same degree*.

Power law distribution heuristic. The Kolmogorov-Smirnov test was used to determine whether the degree distribution of a graph is close to a power law (Pareto) distribution. Computed parameters of power law distribution allow to determine if the fallback test should be applied. Usually, it is necessary for borderline graphs with 6–7 vertices. The fallback heuristic computes the mean degree and checks if the number of vertices with a degree lower than the computed mean is higher than the number of vertices with a degree higher than the mean, i.e. whether the following condition holds:

[1] https://docs.docker.com/compose/compose-file/#links

$$|\{v_i|d \leq \mu\}| - |\{v_j|d > \mu\}| > T \tag{8.1}$$

Based on the threshold T, more or fewer cases can be classified as following the power law; the threshold values 1 or 2 were found to be appropriate for the dataset.

Normal distribution heuristic. To determine if the degree distribution in a graph is normal, the Shapiro-Wilk test of normality [156] is used. This test was shown to be more powerful when testing for normality in comparison to Kolmogorov-Smirnov [157]. Its associated fallback heuristic checks 1) if the most frequent degree in a graph d_f is between the minimal (d_m) and maximal (d_M) degrees, and 2) if the number of vertices with degrees higher than the most frequent degree and the number of vertices with degrees lower than the most frequent degree are almost equal (discrepancy by a threshold $T = 1$ is allowed). The conditions can be written as follows:

$$\left(d_m < d_f < d_M\right) \wedge \left(|\{v_i|d \leq d_f\}| - |\{v_j|d > d_f\}| \leq T\right) \tag{8.2}$$

8.5 Service Degree Distribution Analysis

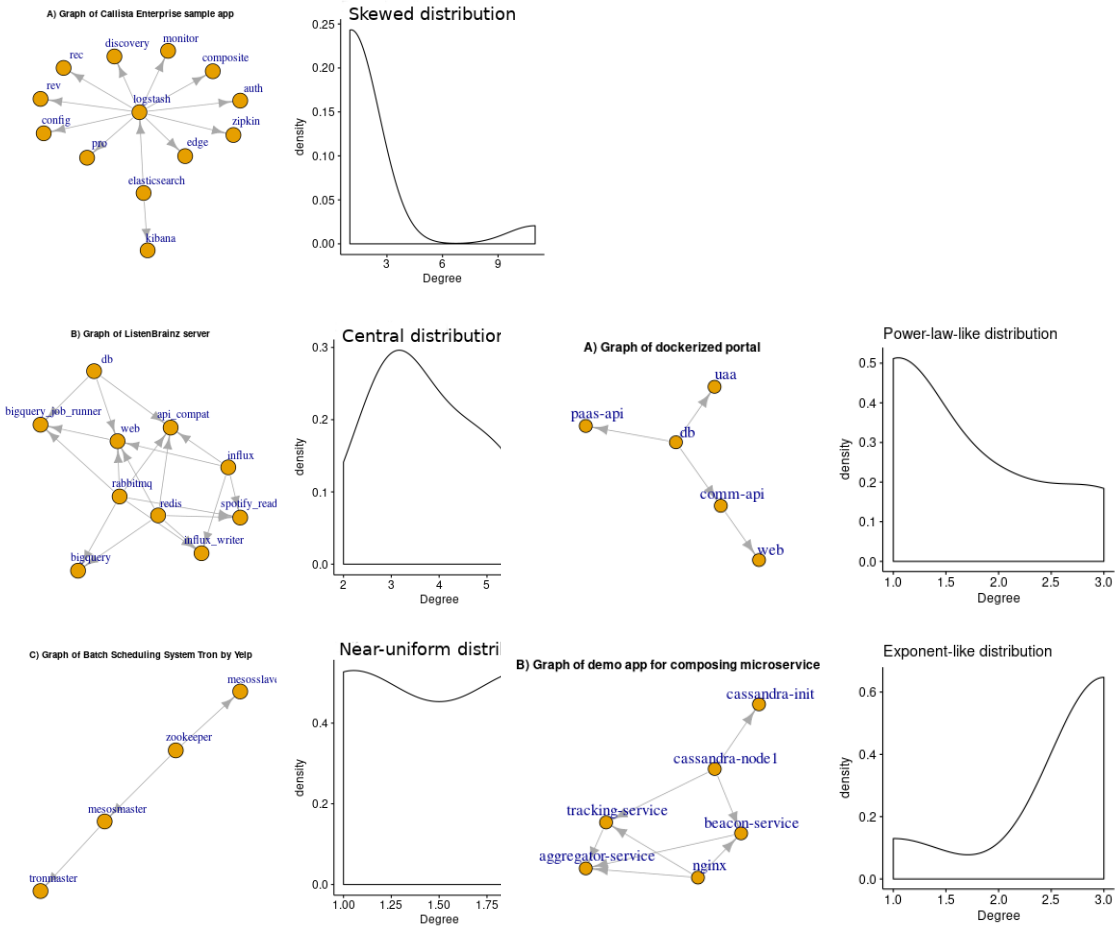
In this section, we show the results of fitting the discussed distributions to the samples from the dataset using the hybrid approach devised in the previous section. Table 8.1 shows the applications that fit in the corresponding distribution type under the parameters set for heuristic tests. Both the absolute numbers and the percentages in the dataset are reported. The *Total* column presents the count of all the applications that fall into the distribution type specified in the row. The *Pure* column shows the count of applications that fit only in single distribution type with their topologies.

Distr. Type	**Threshold = 1**[a]		**Threshold = 2**[a]	
	Total[b]	***Pure***[c]	***Total***[b]	***Pure***[c]
Skewed	90 (87.4%)	48 (46.6%)	80 (77.7%)	42 (40.8%)
Near-uniform	43 (41.8%)	11 (10.7%)	43 (41.7%)	14 (13.6%)
Central	20 (19.4%)	0 (0.0%)	20 (19.4%)	0 (0.0%)
Other	- (-%)	2 (1.9%)	- (-%)	8 (7.8%)

[a]Threshold is set for the fallback test.
[b]Positive outcomes for other types are possible.
[c]Only negative outcomes for other distribution types.

Table 8.1.: Degree distribution types for microservice applications studied.

The main observation from the table is that microservice applications with the power law degree distribution of the underlying structure graph prevail in the dataset. The applications with such a degree distribution cover roughly 87% of the whole data set with a loose threshold of 1 for the fallback heuristic and around 78% with a tighter threshold of 2. The uniform and normal distribution cases amount to only around 42% and 19% of cases, correspondingly. Considering only the cases associated with a single distribution type, a similar picture of power law distribution emerges, being the most frequent with around 47% of all the cases, and followed by the uniform distribution with around a 30%-wide gap. For the small number of unique



(a) Samples following the proposed distributions.

(b) Samples not following the proposed distributions.

Figure 8.1.: Examples of application graphs following different degree distributions.

degrees, the uniform degree distribution might be over-represented. Hence, for the examined dataset, the dominance of the power law-like distribution is even more apparent. Samples of the discussed graphs can be found in Figure 8.1a.

Table 8.1 indicates presence of several distributions that are different from those tested. The threshold increase from one to two for the power-law heuristic test yields an increase in the number of unclassified cases by six, which might be hybrids between the skewed and some other types (cf. sample **A** in Figure 8.1b). The two other cases should be quite different from the power law distribution. Indeed, these two examples show the prevalence of vertices of a higher degree in comparison to vertices of a lower degree. This type of distribution could be described as $N \propto e^d$. (cf. sample **B** in Figure 8.1b).

The main conclusion is that most microservice applications have a structure of a *scale-free network* [158]. The skewed degree distribution with a long tail implies a presence of services that have significantly more connections than the others. There are at least several services that exhibit such usage pattern, i.e. PostgreSQL, Zookeeper, RabbitMQ and Elasticsearch. This is not surprising as they implement common functions, such as logging, configuration management, message brokering, and data storage. This also means that the microservice applications *tend to form bottlenecks* and *can scale unevenly*.

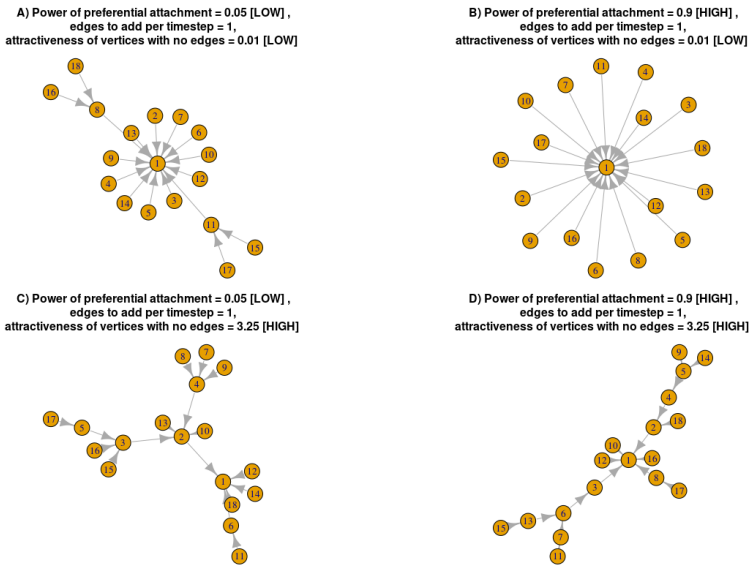


Figure 8.2.: Random graphs generated using BA-model. Number of nodes: 18.

8.6 Application Topology Generation to Assess Predictive Autoscaling Topologies

Limited public availability of the large-scale microservice applications precludes one from studying complex network effects on the autoscaling policies. Our study allows to solve this problem by generating realistic microservice applications topologies based on the generative random graphs models. In this subsection, we touch upon the application topology generation using the BA-model which turned out to be overrepresented in the microservice applications as was shown in the previous section.

We discovered that the change only in two parameters of the BA-model influences how close the generated graph is to the real topology. These parameters are the power of preferential attachment, α, and the attractiveness of vertices with no edges, a. According to BA-model generation process, a single vertex is added to the graph at each time step; a new vertex is attached to old vertices with one or more edges. The probability of i^{th} vertex to be chosen is given by $P_i = d_i^{\alpha} + a$, where d_i is the in-degree of this vertex. Higher values of α favor vertices with more connections, whereas higher a values give chance to vertices with no connections.

Studying the distributions of parameters α and a for graphs with minimal network distances (details in [159]) allowed to find two perspective intervals for each parameter: $\alpha \in [0.01; 0.10] \cup [0.80; 1.00]$, $a \in [0.00; 0.05] \cup [3.00; 3.50]$. For each interval marked either as *LOW* or *HIGH*, a value close to its middle was selected, then four possible combinations of these values were used in generating example random graphs according to the tuned BA model. Parameter *edges to add per time step* was set to 1. Generated samples are shown in Figure 8.2. Visual study shows that sample **B** corresponds to the applications that rely on the common logging service, whereas sample **C** represents an application with several auxiliary services used, e.g., to maintain configurations. Sample **D** in that sense is close to applications organized in the conventional multi-tier fashion. Sample **A** in Figure 8.2 is reflected in the dataset, since it exhibits highly-centralized hierarchical architecture with most of the services using the configuration service.

In Chapter 10, we use BA-model and the discovered parameter intervals to generate realistic microservice application topologies for the assessment of how well do the performance models in predictive autoscaling policies capture the topological information for better service level.

9

Simulation Based Design of Autoscaling Policies

9.1 Execution Model Identification

9.1.1 Execution Model Motivation and Overview

9.1.1.1 Identifying an execution model

An execution model for a distributed system[1] autoscaling is inherently complex. Below we introduce several key characteristics that shaped the autoscaling execution model introduced later.

First of all, autoscaling for multilayered deployments spans multiple *resource abstraction layers*. A high-level overview of such layered representation is provided in Figure 9.1. The topmost layer is an *application layer*. There, the system resources are indirectly represented through the requests processing *capacity* of a service, i.e. how many requests per unit of time an application service can handle without violating the SLOs [97]. Underneath, there is a *virtual cluster (platform) layer*. This layer differs from the application layer in that it a) offers limited amount of physical machines' system resources which are usually time shared, and b) is usually managed by the CSP. The platform layer offers the application layer *scalable* clusters of virtual machines. VM clusters can be scaled depending on whether the application needs more or less system resources to meet the user demand and to reduce the amount billed for using the cloud resources. At the bottom, there is a *physical infrastructure layer*. This layer cannot be autoscaled, but the autoscaling can account for its capabilities and limitations. For instance, service threads might be pinned to particular sockets s.t. the inter-process communication overhead in NUMA system is reduced.

Second, autoscaling has an inherent *time dimension*. The autoscaling process is often viewed as a sequence of application and VM cluster states on a joint timeline. Each state is distinguished by a) the allocated system resources and requests processing capacity of an application service, and by b) whether this resource/capacity state is already enforced. The latter distinction originates from scaling actions not being instant, i.e.

[1] In this thesis, the notion of distributed system encompasses both the application and the cluster of virtual nodes.

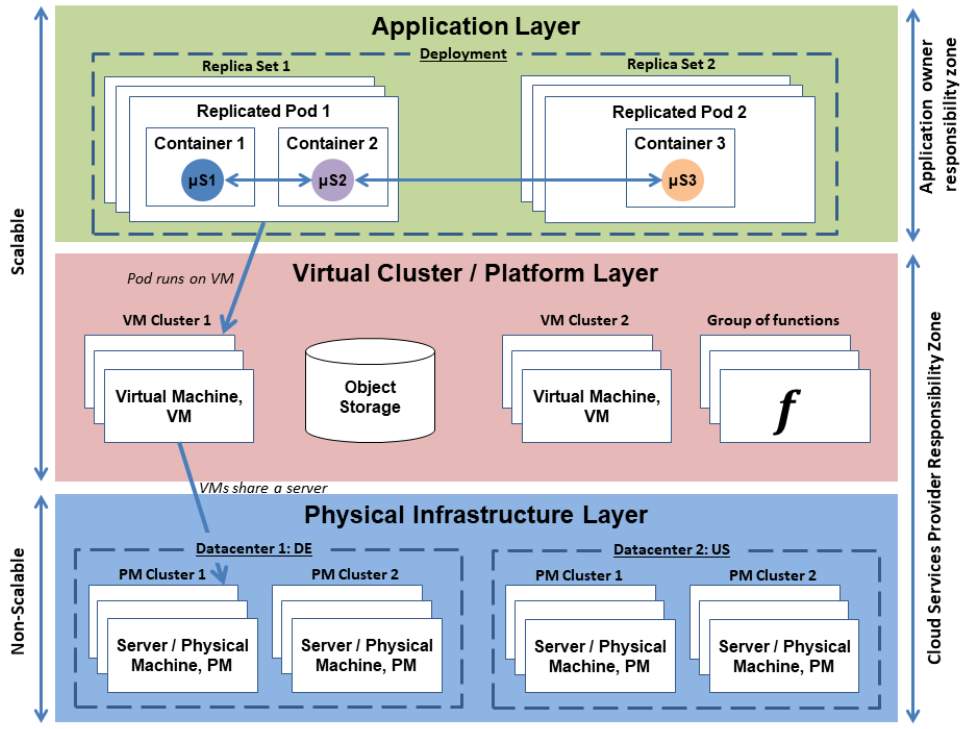


Figure 9.1.: Resource abstraction layers in Infrastructure-as-a-Service (IaaS) cloud.

some time has to pass before the *desired state* becomes the *actual state*. This lag is scattered across different abstractions and exhibits a degree of randomness. Mainly, it depends on the booting/termination time of virtual machines, on the time required to start and stop the service instance, on the immediate availability of cluster resources and budget to purchase them, on potential failures on any of the resource abstraction layers, and on the parameters of autoscaling such as *cooldown period*.

Third, higher resource abstraction layers are bound by the resources provided lower in the hierarchy. If the VM cluster size is insufficient to accommodate the desired count of service instances, then the instance may fail to start or will have to wait until the required resources become available. This dependency is asymmetric. If the cluster node scales down or fails, then all the service instances running on it will disappear.

Fourth, the applications significantly differ by how they utilize system resources and how the request paths (traces) are formed within the application. The execution model should grasp this diversity without trading its generalization capabilities. The execution model can address the versatility of request traces by modeling the *application topology* and distinguishing between the workloads produced with different requests types. For example, a request to purchase an item from a web shop is likely to be more "heavy" processing-wise than a request to just show the product details.

Fifth, there are two constituents to the autoscaling behavior – the *policy* and the *mechanism*. The policy part is responsible for *what* scaling actions should be performed. In contrast, the mechanism determines *how* these actions will be performed. The autoscaling policy determines the scaling trigger, i.e. a condition for a scaling action to get scheduled. Such triggers may be asymmetric as in Kubernetes Cluster Autoscaler where scale up and scale down actions have separate pieces of code responsible for each of them. The autoscaling mechanism is an algorithm that implements the scaling logic including collection of metrics, their processing, and the enforcement of the desired state.

The above list is not exhaustive, but these key concerns should be taken into account when devising an execution model appropriate for analyzing the autoscaling process. In what follows, we present an execution model for generalized autoscaling that prioritizes meeting SLOs relevant for the end user.

9.1.1.2 Execution model overview

To build an execution model, we first identify its key abstractions:

- **Application** represents a microservice application that processes the user requests according to the logic implemented in its services. This abstraction captures the resource utilization introduced by requests processing and allows to measure service level metrics such as response time.
- **Platform** represents scalable clusters of nodes that share physical infrastructure. Usually, these clusters are provided by CSPs to the application owner on a pay-as-you-go basis. The reason to decouple the platform abstraction from the application abstraction is due to the independence of the resource allocation goals imposed by these two layers. The application prioritizes meeting strict SLOs even at the expense of additional resources, while the platform-level goal is to reduce the expenses by minimizing the resource usage. Clearly, these objectives are in conflict.
- **Deployment** abstraction encapsulates the deployment logic. It incorporates an initial deployment configuration that includes such high-level parameters as geographical regions, which the application should be deployed into, the number of service instances as well as the types and counts of VMs to be allocated initially. Once the deployment phase is over, some parameters are bound to stay the same unless the application is re-deployed (e.g. regions bindings).
- **Load** abstraction captures the user load pattern over time allowing for some randomness on load generation. Load abstraction was added to the execution model since user load serves as a trigger for the elastic behavior of the distributed system. Having load in the execution model allows to add it into the list of controllable parameters when doing simulation-based autoscaling policies evaluations.
- **Application Scaling Policy** encapsulates autoscaling decision making, starting with the collection of monitoring data and all the way to the enforcement of the new state. Following the separation into the application and platform, the application scaling policy decides on the desired state of application to fulfill the service level objectives, whereas the platform follows the best-effort policy to accommodate this desired state (cf. platform adjustment policy below). The application scaling policy is a variation of monitoring-analyzing-planning-executing (MAPE) loop [125].
- **Platform Adjustment Policy** follows the decision taken by the application scaling policy and attempts to devise the platform state that can accommodate all the desired service instances taking into account their resource limits. At the same time, the adjustment policy is responsible for doing its best in meeting the platform adjustment goals such as cost minimization as long as the desired application state does not suffer. For example, the application owner might be interested in minimizing the amount billed from her account for the cloud resources, hence the adjustment policy tries to meet this goal, but only after the primary goal of the application scaling policy is met.
- **Scaling Process** abstraction is responsible for delaying the scaling actions like in real systems. It includes service instances and VMs start-up and termination delays. The impact of these delays on the quality of the autoscaling is substantial [11], hence they get their dedicated abstraction. Often, the overprovisioning-averse autoscaling policies fail to meet the SLOs imposed by the application owner precisely because of these delays.

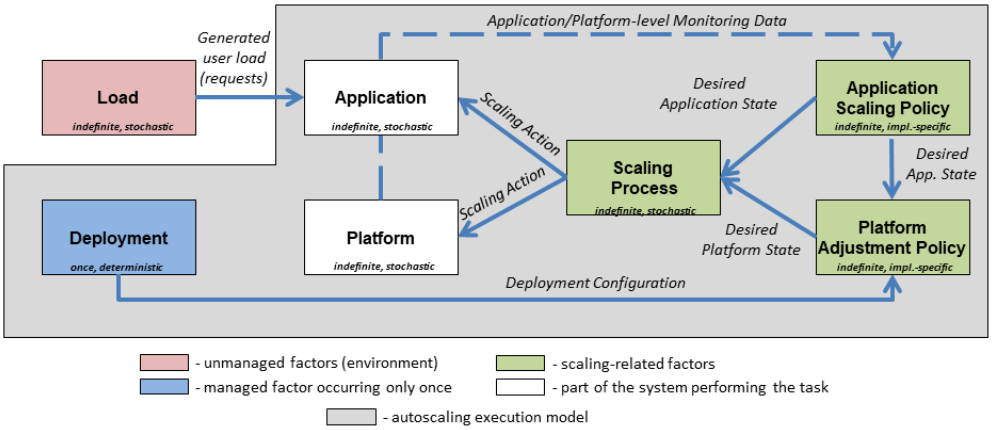


Figure 9.2.: Execution model for autoscaling of multilayered deployments.

Abstractions comprising the execution model can roughly be categorized into the *environment* and the *autoscaling execution model* itself. A bird's-eye view on the execution model is shown in Figure 9.2.

Environment groups the unmanaged aspects that the autoscaling process has to consider either directly or indirectly. Load is one of the most important such aspects (cf. the red box in Figure 9.2). Depending on load and the quality of service delivered, both the application and the platform might need to be scaled. Load has stochastic nature, and the main source of stochasticity is how the count of requests changes over time.

Autoscaling execution model groups scaled entities, i.e. the application and the platform, with the autoscaling process representation (cf. the gray area in Figure 9.2). The application and platform abstractions form the core of the execution model (cf. the white boxes in Figure 9.2). The application abstraction implements the requests processing logic. Thus, it is partially responsible for the requests processing times. The platform abstraction bears the leftover responsibility for requests processing times since it provides system resources such as processor time, network bandwidth, memory, and disk space. Other parts of the execution model impact requests processing substantially, but *indirectly*.

The deployment abstraction determines the initial state of both the platform and the application in terms of capacity and resources that they collectively offer (cf. the blue box in Figure 9.2). In respect to the application abstraction, deployment pins services to particular geographical or logical regions and indicates the desired initial capacity in terms of services instance counts and their resource requirements. Similarly, for the platform abstraction, it provides an estimate of desired initial system resources and their location in geographical or logical regions. Treating deployment as an independent abstraction is preconditioned by the set of initial constraints such as the set of regions, which autoscaling can operate in.

An autoscaling aspect in the execution model is represented with an abstractions triad – Scaling Process, Application Scaling Policy, and Platform Adjusment Policy (cf. green boxes in Figure 9.2).

The application scaling policy abstraction represents the adjustment logic for the application. It determines the quantitative parameters of the application (e.g. the number of service instances) required to meet the service level objectives such as 99th percentile response time being lower than a threshold. Application scaling policy computes the desired state based on its own parameters and the monitoring data. The monitoring data includes various metrics, e.g. resource utilization or load volume, and the current state of the application and the virtual cluster. Once the desired state is calculated, it is passed to the platform adjustment policy.

The platform adjustment policy attempts to derive the platform state such that the desired application state has enough system resources. It also attempts to meet its own objective, e.g. cost minimization. It can do

so in various ways, e.g. by placing instances of services with non-conflicting requirements to the resource kinds on the same virtual machine or by going for other cloud services provider if the offer is cheaper. Once the desired state of the platform is devised, the scaling process abstraction is used to delay the desired application and platform states accordingly.

Scaling process captures random booting and termination delays introduced both when scaling the services and the VM clusters. Naturally, these delays impact the results of the autoscaling process. The desired state that is devised by the autoscaling policy cannot be enforced immediately due to various delays such as start-up and termination delays for virtual cluster nodes and application service instances, also network delays and overbooked clusters. The scaling process abstraction amends the desired states upon their enforcement, i.e. if the desired state supposes that a new VM instance will start at a particular time, the actual starting time will be delayed by the scaling model in a stochastic way.

Presented components of the autoscaling execution model are discussed in the following subsections.

9.1.2 Application

The application abstraction is central to the autoscaling execution model. The reason is that scaling a cluster of virtual nodes in isolation, without considering application characteristics such as the response time, will likely yield poor user experience [11]. We view each application as something bigger than a simple collection of services, hence the proposed abstraction determines:

- **Interconnection of services in the application**. The communication chain is usually split across the services. A service only knows its immediate neighbors. Such an encapsulation of the communication path improves application resilience and makes it easy to modify. Any change to the communication path requires isolated adjustments made at each service. However, modeling the service communication as-is renders the holistic analysis of the application level autoscaling impossible. Therefore, the application abstraction offers an aggregate representation of communication chains in a graph form.
- **Requests processing characteristics and requests propagation paths**. Request-related information is spread between the load abstraction and the application abstraction. The load abstraction (cf. Subsection 9.1.3) represents characteristics of the aggregate user load, e.g. how many requests arrive at the given endpoint within a second. In contrast, the application abstraction captures the processing characteristics of incoming requests. Request paths in the application are also represented in the application abstraction. Each such path sets a chain of unique relations, including *service-to-service*, *entry-to-service*, and *service-to-exit*. Any request path is represented as a directed graph.
- **State of the application**. At any given point in time, the application state is distributed among its components, i.e. services and service buffers. It is impacted by load, resources available to the application services, by logic that these services implement, and by the capacity of service buffers. The application abstraction represents the state of the modeled application through the requests processing characteristics and requests propagation paths, including varying requests fan-in and fan-out factors.

An application with multiple tiers is commonly modeled as a *queuing network* [160]. The rationale behind this representation is that each application service processes the requests (*serves* in the terminology of queuing theory) by consuming the limited resources offered to it by the underlying virtual nodes. If the service resources are saturated s.t. no more incoming requests can be processed at the moment, then the incoming request ends up in a service buffer. The buffer stores the requests waiting to be processed by one of the service instances. If the buffer is full, incoming requests are dropped. When the service managed to depart

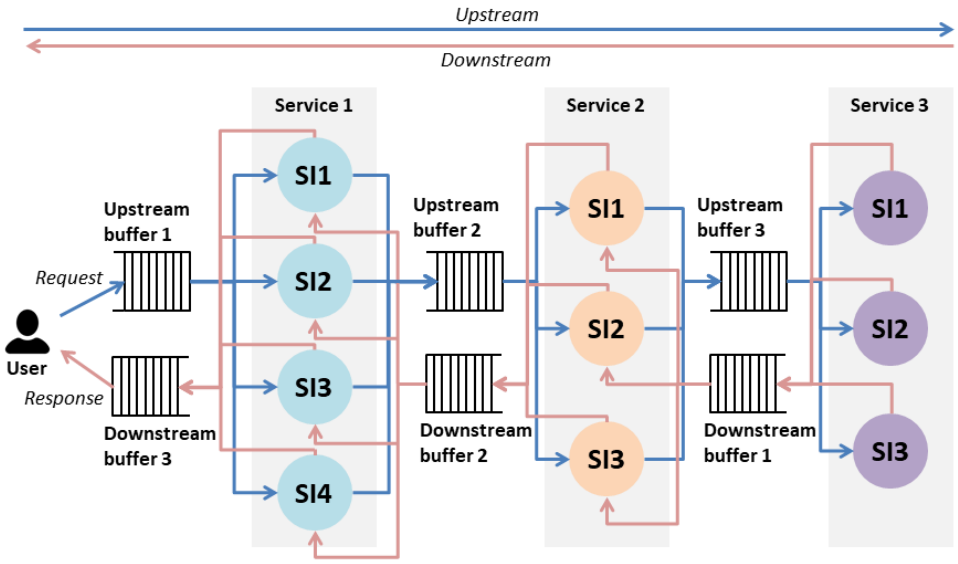


Figure 9.3.: An application represented as a queuing network. Each service is shown as a group of service instances: **SI1**, **SI2**, etc. The downstream communication represents the propagation of the requests through the services *from* the user. The upstream communication represents the propagation of the response through the service *to* the user.

from the resource saturation point, it takes on the waiting requests. Multiple policies that can be employed to manage the requests ordering in the service buffer.

Figure 9.3 shows a queuing network example corresponding to an application with three logical tiers, each having a single service. The first service has four instances, whereas the second service has three instances. Unlike conventional queuing networks, the *entry point* corresponds to the *exit point*. When the user sends a request to the application, the path of this request in the application is split into two parts.

The first part, called **upstream**, is the path that starts at the entry point and continues through the predefined service chain until the last service is reached (usually, a database). Upstream requests spend time in the dedicated *upstream buffers*[2] waiting until some service instance can process them. The processing may result in forming additional upstream requests to other services (requests fan-out). For convenience, these requests are attributed to the original one.

When the last service (e.g. database) is reached, the application should be able to form a *response*. Responses take the **downstream** path, i.e. the second part of the user request path in the application. On a way back to the user, the response may also need to wait till the service instances can process it. For that purpose, the Application Model offers the *downstream buffers*. Hence, each service is equipped with a pair of buffers: one downstream, one upstream.

9.1.3 Load

The load abstraction represents generation of the user load. The notion of request is central to this abstraction. Request carries the information relevant for application-level simulation:

[2] Queuing theory refers to buffers as 'queues'. In the context of the thesis, this term is considered misleading. The term 'queue' supposes first-in-first-out (FIFO) queuing discipline, whereas the requests and responses can be taken in different order. This order is regulated by the adopted service discipline and the practical implications, e.g. when the responses arrive out-of-order, but are all required to form the next response.

- **Region** which the request belongs to. This information enables modeling/simulation of geographically distributed applications. Load patterns may differ across regions.
- **Request type** allows to identify the request path and how much resources will it consume on its way while being processing by the service instance. This is an artificial attribute that simplifies representation of versatile combinations of endpoints with the user data.
- **Upstream/Downstream** notifies the application abstraction whether the upstream or the downstream processing logic should be applied to this specific request.
- **Processing time required per service** represents the time that the request should spend in the processing state for each application service in order to be allowed to proceed to the next service. The processing time differs by whether the processing is upstream or downstream.
- **Entry service** is the name of the first service on the request propagation path.
- **Timeout**. If the cumulative lifetime of the request reaches this value, then the request is dropped.
- **Size** represents the size of the request in bytes. This parameter is relevant for considering how much of the network bandwidth is currently occupied by the transmitted requests.
- **Request ID** is a unique request identifier. It allows to represent multiple requests and responses generated by the services upon finalizing processing of the initial request. Keeping request ID the same allows to simplify the representation of the request propagation path as a chain of services that pass the request from one to another replicating it on fan-out. The answers to the replicated requests have the same ID. They should be combined to allow the response to proceed.
- **Replies expected** carries the cumulative number of responses for the fanned out request that are expected before the downstream processing could proceed.
- **Processing time left and the transfer time left** are both set before the request is put for processing by the service or the link (cf. Subsection 9.1.5.3). The request is not allowed to proceed unless these fields hit zero.
- **Stats: cumulative time, network time, buffer time**. The per-request statistics of time spent in different states, such as being transferred over the network or waiting in the buffer, provides detailed insights into how the request lifetime was distributed. These data allow to evaluate quality of the adopted scaling policy at a fine granularity which is required to tune the autoscaling policies.

The load abstraction offers a couple of ways to configure load patterns on a *per-region* basis.

The first way is to set up a *seasonal pattern*. The seasonal pattern can be defined either by a periodic function such as sinus or by a set of values repeated with some frequency. In most cases, load patterns do not differ much day by day. Instead, they do differ between the work day and the weekend. The load abstraction supports this configuration. Load generation settings can be configured for all months without the need to write the configuration for each month separately. If either more fine- or coarse-grained measurements are available for identifying the load pattern, the resolution of the pattern can also be changed. One can configure the *ratio of each request type* in the generated load and add some randomness on top of the generated pattern by setting the type of a *probability distribution* and its parameters. The probability distribution determines by how many requests will the load at the given time step be adjusted to account for stochastic user behavior.

The second way to configure the generated load is to use *the load trace* of a real application or a website [161]. Such traces usually contain an aggregated request rate per some unit of time, e.g. an hour or a minute. In some cases, anonymized characteristics of individual requests such as size in bytes and the referenced endpoint may also be available. These parameters can be used to generate the requests that follow a realistic pattern. There are, however, three major limitations to using the real load traces: a) limited availability of such data, b) the load trace might not be sufficiently fine-granular, c) the load trace represents *just a single turn of events*, i.e. it may miss on some corner cases such as sudden load spikes.

9.1.4 Deployment

The deployment of an application is done only once at the very beginning. Hence, the deployment step is considered separately from changing the application and the VM clusters with scaling. The reason is that the deployment sets the initial configuration and thus acts as a bounding box for the forthcoming scaling actions. An example of such parameters are the regions which the application service instances are deployed in. The initial assumptions of the application owner about the resource demands are also reflected in the deployment.

To a large extent, the deployment abstraction acts as a container for the deployment settings. These settings are transformed into the delta-representation. This representation reflects the desired change to the state of both the platform and the application. Since simulation starts with no nodes and services, this change always has a plus sign and takes on values specified in the deployment configuration. The state change produced has to be delayed via the scaling process abstraction (cf. Subsection 9.1.6).

9.1.5 Platform

9.1.5.1 Main functions

The platform abstraction represents clusters of virtual nodes underlying the application. It holds the current state of the platform and the sequence of state deltas, both desired and enforced, ordered in time. Upon reaching the simulation time that corresponds to the timestamp for an enforced state change, the corresponding change is *rolled out*, i.e. the current state of the platform is updated with this change. Platform participates in two stages of the autoscaling process:

- **Adjustment**. During the adjustment stage, the resources offered by the current platform state have to be adapted to the anticipated application demand. Conceptually, the platform is viewed as a follower of the application in the context of the autoscaling process. This means that, first, the required state of the application is determined to accommodate the user load, and, second, the platform is adjusted correspondingly to fit the resource demands of the to-be-scaled application.
- **Placement**. The placement stage binds the instances of the application services to the virtual nodes. Only upon finalizing the placement, will the service instances be available for the requests processing. This stage operates on the node groups abstractions instead of individual nodes. All the nodes in a single group have the same resource amount to offer (homogeneous node groups). In special cases, when the application services require special resources to be available on the node (e.g. a GPU or an FPGA board), the node groups add another distinguishing feature – label. The labeling mechanism allows to distinguish between the node groups with non-standard resource types.

The current state of platform is represented by node groups. When scaling, we do not distinguish between the individual nodes. However, for the purpose of modeling the resource consumption and the placement, in Subsection 9.1.5.2 we consider a virtual node abstraction. Subsection 9.1.5.3 discusses a link abstraction to account for the network delays that may have significant impact on the response time [162].

9.1.5.2 Node

The node abstraction captures both the technical and the economical characteristics:

- **Provider** represents one of the cloud services providers that offers virtual nodes (VMs). Capturing this piece of information allows to model/simulate the multi-cloud scaling.
- **Type** represents a type of the virtual node specific to the provider, e.g. t2.nano for AWS.
- **vCPU** represents virtual CPUs count granted to the node of this type.
- **Memory** represents the size of the main memory allocated on the physical server for the node.
- **Network bandwidth** is a bandwidth that is provided to a single virtual node of the given type.
- **Disk** is the size of the secondary storage directly available on the virtual node.
- **Price per unit time** defines how much one will be billed for a unit of time worth of node usage.
- **CPU credits per unit time** defines how many CPU credits will the cloud services user get for using burstable performance instances that she can spend for vCPUs at some predefined level of utilization.
- **Network latency** represents the advertised or the measured network latency for the given CSP and the virtual node type.
- **Labels** is a set of strings each of which represent a particular feature of the given node type such as being equipped with GPUs to enable more precise placement in certain cases.

Both the node abstraction and the characteristics associated with it are not considered in isolation. The main purpose of this abstraction is to produce an estimate of the resources, consumed by the service instances on the virtual node. The notion of the consumed resources is thus tied to a particular type of node. To compute the resource consumption in a node group, the resource consumption for a single node is multiplied by the count of nodes in that group. When conducting the placement of service instances, it is necessary to test, whether the service instances can indeed be placed on the node, i.e. whether there are enough resources. Similarly, before a request is added for processing on the node, it is estimated whether the available resources will be sufficient for the request to be processed. These tests allow to select a node type that has sufficient resources upon conducting the adjustment phase of the scaling process.

9.1.5.3 Link

The link abstraction captures network delays when sending the requests from one node group to another node group. Any link is characterized by its latency and the available throughput. The dynamic state of the link is characterized by a) the requests that are currently in transfer on this link, and b) the throughput currently used by the requests in transfer. The dynamic state is updated upon the addition of new requests or their departure either via being transferred into the buffer (capacity available) or by being dropped if the timeout was reached. A single link simulation step is shown in Algorithm 1.

Upon performing a simulation step, the link abstraction considers each request **r** that is currently in transfer. It first modifies the time left for this request to wait on the link, the time that it spent on the network, and the cumulative time. The waiting time is initialized with the latency of the link (not shown) upon adding the request to the link. If the waiting time left for this request has hit zero, it is considered for being pushed into the service buffer **Buf**. If the buffer does not have spare capacity to hold the request, then it is dropped. The request is also dropped if the cumulative time reached the timeout value (this piece of logic is not present for clarity). If the buffer has spare capacity, then the request is pushed into the buffer. In all of these cases the used throughput of the link **Thru** is reduced by its size.

Algorithm 1. Pseudocode for the link abstraction simulation step.

input : Requests currently in transfer on this link **R**, used throughput of the link **Thru**, modeling step **step**
output: Requests to be considered for adding into the buffer **RBuf**

```
1  for request r in R do
2      RBuf ← ∅;
3      r.waitingOnLinkLeft ← r.waitingOnLinkLeft - step;
4      r.networkTime ← r.networkTime + step;
5      r.cumulativeTime ← r.cumulativeTime + step;
6      if r.waitingOnLinkLeft ≤ 0 then
7          RBuf ← RBuf ⋃ r;
8          Thru ← Thru - r.size;
9          R.remove(r);
10     end if
11 end for
12 return RBuf
```

9.1.6 Scaling Process

The scaling process abstraction defines the part of scaling behavior that depends neither on the scaling policy nor on the adjustment policy. Particularly, the scaling process abstraction is responsible for introducing the *delays* of starting and terminating the node groups and service instances. These delays are stochastic, but they are rooted in the deterministic processes such as creating a VM on a hypervisor or recreating a container upon changing its resource allocation. The scaling process abstraction consists of two sub-processes:

- **Application scaling sub-process** introduces the start-up and termination delays for every service in the application abstraction. This sub-process can only be used to delay the groups of services. The delay is issued upon enforcing the desired state of the group.
- **Platform scaling sub-process** encompasses the start-up and termination delays for every node type of every provider available. This abstraction introduces the scaling delays for a node group as a whole. The delay is issued upon enforcing the desired state of a node group.

Both the above sub-processes cooperate to introduce the coordinated delay of scaling actions across multiple resource abstraction layers. Their cooperation in delaying the scaling actions is represented by an abstraction that pairs the platform change with the change in the application – *generalized delta*. Similarly, they delay the deployment actions as well.

9.1.7 Scaling Policy and Adjustment Policy

9.1.7.1 SPACE process

The abstractions of the scaling policy and the adjustment policy are both responsible for determining the quantitative parameters and the time structure of the scaling actions on the application and on the platform levels correspondingly. Both policies are composed of building blocks (discussed later) realizing isolated

parts of autoscaling-related logic. Combined, application scaling and platform adjustment policies implement the logic of the autoscaling process. This thesis focuses on autoscaling for multilayered deployments roughly following SPACE process model, which is introduced in the following paragraphs.

SPACE process model attempts to generalize the internal behavior of coordinated autoscaling solutions such as those offered by Kubernetes (horizontal or vertical pod autoscaler + cluster autoscaler) or by the managed Kubernetes services of cloud services providers. We tried not to specialize this model to the features of particular in-house autoscaling services offerings by CSPs since these a) tend to change over time and b) do not expose internal algorithms publicly. The SPACE autoscaling process model borrows the letters in its name from the scaling phases that it unifies:

- **Scale**. During this phase, the scaling policy has to decide on the desired application state. The application can shrink and grow by removing the service instances or adding the new ones (horizontal scaling). In addition, resource allocation for the service instances might also be changed (vertical scaling). **Scale** phase is put at the very beginning of the process since, when prioritizing meeting SLOs, one needs to first determine how the application should be scaled to meet the SLOs. State-of-practice autoscaling solutions, such as Kubernetes, implement cooperative scaling across layers by starting with the application. When the desired application state is determined, one may adjust the platform resources.
- **Place**. This phase defines a set of placement constraints that determine the placement options for the service instances in the desired states. Determining the placement constraints takes both static and dynamic parameters into account. In terms of the static parameters, placement constraints derivation considers service requirements as experessed by special resources such as GPUs and FPGAs and filters out only those virtual node types that are equipped with the corresponding special resource. For example, an image recognition service may profit from being placed on a node with a tensor processing unit (TPU). Placement constraints derivation takes dynamic parameters, such as observed resource utilization, into account as well. Knowing the resource utilization by application services allows to roughly determine which virtual nodes types will be able to cover their resource demands.
- **Adjust**. The adjustment phase attempts to derive the platform state that will be able to accommodate the desired application state given the placement options. In addition, this phase tries to minimize the cost of the final deployment by choosing the cheapest placements. First, this phase checks whether the existing virtual nodes are sufficient. If there are idle nodes and no new service instances are expected to appear, these will be terminated. If some service instances are present in the desired state, then some or all the idle nodes might be spared. In case the existing nodes count is insufficient to accommodate all the desired service instances, new virtual nodes should be started.
- **Co-locate**. This phase is not obligatory but might be required by some approaches to the platform adjustment. For instance, if the platform adjustment stage is concerned with the deployment cost but at the same time values the application performance, it may attempt to co-locate the desired service instances in such a way that the resource contention is minimized. Minimization of resource contention by co-location may be achieved in multiple ways. For instance, one may attempt to minimize the network contention by placing intensely-communication services on the same node. Or, if two services have different resource usage patterns, e.g. one is CPU-bound and another one is memory-bound, then these could also be co-located to balance the resource utilization on the node, i.e. reducing the bill with limited harm to performance.
- **Enforce**. Lastly, the desired adjustments should be enforced when the time comes. Once the adjustments are enforced, there is no way to overwrite them by the subsequent state change. Only issuing the scaling action of the opposite direction (e.g. scale down for scale up) and the same size in terms

of node count will be able to cancel the effect of the enforced adjustment. However, the enforced state will likely to stay for some time until the cancelling change appears and hence will be able to somehow impact the requests processing. This differs the enforced state from the desired one.

The next sections present the phases **Scale**, **Place**, and **Adjust**. All three are obligatory for the SPACE process. The **Enforce** phase is also obligatory but it is too trivial for a dedicated discussion.

9.1.7.2 Scale

The Scale phase is responsible for scaling the application services and is thus fully captured by the scaling policy of the autoscaling execution model. The computation of the desired change for a service is organized in a sequence of fine-grained operations that converts the metrics values into the desired change in terms of service instances. To compute the desired change, one may use one or more metrics. Hence, the operations split into the *per-metric* operations and *per-metric group* operations. The per-metric operations are as follows:

- **Filtering** is applied on raw metric values to either substitute some values for other ones or to smooth it according to some filtering algorithm to get a more pronounced seasonal pattern.
- **Aggregation** is applied to the filtered metric time series to adapt its resolution by aggregating the values within the specified moving time window, e.g. one can take a maximal value in the time window or an average. This operation differs from filtering in that it changes the time series resolution.
- **Forecasting** extrapolates aggregated metric time series into the future. This operation is performed with help of an underlying forecasting model. These models may vary in how they function, but they unite in the *predict* interface. Forecasting is performed using such approaches as ARIMA, support vector regression, singular spectrum analysis, long short-term memory (LSTM) recurrent neural networks, etc (cf. Chapter 6). If the policy is reactive, then the forecasting operation is omitted.

Once the operations from the list above are performed on every metric from a group of metrics considered for scaling, the scaling phase proceeds to determine the desired state of the application. It does so by performing the following operations that make use of all the relevant metrics in a group:

- **Desired state calculation** is a key operation in determining the desired state of an application. All the metrics from the same metrics group are used in determining the state. If one or more of these metrics were forecasted, then the desired state will be in the future. The desired state calculation is done either via some analytical expression or using a black-box modeling technique such as machine learning. Despite the particular implementation, the calculated desired state should represent either how many service instances are required (horizontal scaling) or by how much should the resources allocated to them be changed (vertical scaling). Both approaches can be combined.
- **Stabilization** minimizes the oscillations in the computed desired scaling aspect using the windowing function and some aggregation in the window. For instance, in case of horizontal scaling the count of service instances may be stabilized by taking the 2 minute-wide time intervals and computing the median of service instances count in such an interval.
- **Limiting** puts the stabilized scaling aspect values into the predefined interval. Such interval is defined by the minimal and maximal value. For example, in horizontal scaling, one defines the limits on the count of service instances. The minimal value is then the count of service instances which the stabilized count cannot fall below. This may be achieved by substituting all the values that are lower than the minimal limit set by the value of this limit. Similarly, the values that are higher than

the maximal limit are substituted for it. This mechanism is similar to that of Kubernetes and IaaS autoscaling solutions of cloud services providers where one sets the min and max counts of replicas and VM instances correspondingly.

The result of the last operation is a series of desired application states to be fed into the **Place/Adjust** phases.

9.1.7.3 Place & Adjust

The combination of the **Place** and **Adjust** phases of the SPACE autoscaling process finds its representation in the platform adjustment policy abstraction. The combination of these phases aims at adapting the clusters of virtual nodes to the resource demands of the desired services state determined at the **Scale** phase.

The **Place** phase constructs services placement options for each node type. Each such option represents the maximal possible allocation of the service instances on the given type of node. Such placements are generally limited by the resource amount offered by a particular type of node. However, additional placement hints might be utilized to fulfill specific platform adjustment goals, e.g. balance the consumption of different kind of resources on a node by co-locating the services with complementary resource utilization patterns. Once the placement options are devised, they are scored based on some quantitative characteristic, e.g. the cost of the placement option. The score must adhere to the following property: higher score means better option.

The **Adjust** phase makes use of the scored placement options to determine the desired platform state. Overall, the **Adjust** phase gets its work done by proceeding through the following sequence of operations:

1. **Determining whether the resources can be freed**. This operation evaluates whether the desired change of the scaling aspect can be performed without adding new virtual nodes. As a result, this step might scale the virtual cluster down if some idle virtual nodes are available after the desired change is applied to the services running on the cluster. If there is still an unmet positive change after this step, then the following two options are considered.
2. **Computing the adjustment as expanding the resources of the platform [Option I]**. This is one of two alternatives considered to accommodate the unmet increase in the services count (horizontal scaling) or resource limits (vertical scaling). This option supposes that the virtual clusters after the previous operation stay unchanged. It adds the new nodes to these unchanged clusters, hence no service migration is required.
3. **Computing the adjustment as substituting the old platform state [Option II]**. This alternative of expanding the resource allocation on the platform level considers scraping the original virtual clusters (left after the first operation in the sequence) and substituting them for the better scored placement options. To avoid the reduction in the application availability, this option does not schedule the previous platform state for the scale-down unless the new state is ready and all the service instances from the old state have started there. This means that there is an interval of time during which both configurations exist in parallel. This results in additional costs for this so-called *shadow state*. The duration of this period largely depends on the booting times of the virtual nodes in the new platform state and the termination times of the virtual nodes in the old state.
4. **Choosing the best resource allocation option**. The above alternatives for the scale out are compared and the best one is selected. The criteria is the highest score computed as an aggregate of the placement scores for each option.
5. **Adding up the adjustments**. The computed changes are aggregated and added to the joint timeline to be enforced when their time comes.

9.1.7.4 Simulating scaling and adjustment policies

Current section introduced the abstract notions of the scaling and adjustment policies responsible for scaling the application and the underlying platform correspondingly. Combined, they constitute what is known as an *autoscaling policy*. The intention behind keeping them as abstract as possible is to leave some room for their adaptation to particular workloads and platforms. Another motivation is to organize a complex autoscaling process in a simple manner to enable its comprehensive analysis as well as the study of particular implementations of these policies.

Specific autoscaling policies can be simulated with the proposed scaling policy and adjustment policy abstractions by instantiating their components in different phases of the SPACE process. Chapter 9 discusses particular instantiations of these components, whereas Chapter 10 evaluates concrete arrangements of these building blocks into autoscaling policies.

9.2 Autoscaling Simulation and Experimentation Toolbox

9.2.1 Simulation Toolbox

A comprehensive simulation toolbox is an original contribution of this thesis. It aims at performing the fine-grained simulations of autoscaling policies. The simulation toolbox is implemented in Python. The choice of the programming language trades the performance for a higher flexibility in extending the toolbox. In addition, it enables easier reuse of the policies-specific code backed by Python's specialized analytics packages. Python also ensures wider reach of the simulation toolbox due to the language's popularity[3].

The simulation toolbox comprises five tools that enable the simulation-driven design and exploration of autoscaling policies with limited effort:

- **Multiverse** is an autoscaling simulator and the core of the toolbox. It simulates autoscaling of multilayered deployments [11]. Since the contributions in this thesis focus on the response times experienced by the cloud application user, **Multiverse** attempts to accurately simulate the application as a network of services with buffers. In a simulated application, the requests travel from one service to the other possibly waiting in the buffers till the service instances can process them. Each request is simulated individually which limits the simulator performance but allows to study the tail latency effects in detail.
- **Stethoscope** is a simulation data visualization tool. The tool leverages Python's `matplotlib` package to produce two categories of plots. The first category represents the quality of an autoscaling policy as experienced both by the user and the application owner. The second category characterizes the autoscaling behavior, i.e. an impact of an autoscaling policy on the internal deployment state.
- **Cruncher** is an autoscaling simulation automation tool that leverages the simulation capabilities of **Multiverse** and the visualization options offered by **Stethoscope**. This automation tool arranges concrete simulations to run based on the alternative configuration files provided to it. Evaluation of alternatives is allowed for any configuration file, including application, scaling process, and scaling policy. Each possible combination of provided alternatives is explored by **Cruncher** by feeding it into **Multiverse** and retrieving the results once they are ready. The evaluation results for the alternative configurations are then combined and fed into **Stethoscope** to get the comparative plots.

[3] `https://pypl.github.io/PYPL.html`

- **Praxiteles** is a tool that generates the sets of configurations files for autoscaling simulations based on meta-configuration files (aka recipes), cloud provider traces, and application topology model. The meta-configuration files determine what the generated simulation configuration files would contain. They also specify either the concrete parameters for these files (e.g. the count of services) or the probabilistic distributions over the parameter values (e.g. the memory consumption by an application is uniformly distributed in the range from 100 to 200 MBs). Some of the parameters may be derived directly from the data, i.e. from the traces that some CSPs make publicly available. These mostly include load traces and resource utilization traces. Currently, **Praxiteles** supports only two traces for Azure cloud that were recently published by Microsoft [31, 30]. Support for new traces may be added by writing custom classes implementing the same interface. Some simulation aspects can also be derived from the models. As of now, **Praxiteles** supports only the model-based application topology generation [159] and the model-based booting/termination time intervals generation. The tool's output is fed either into **Cruncher** or **Multiverse**. To sum up, **Praxiteles** attempts to simplify the tedious process of manual simulation configurations design when only the high-level aspects of simulation matter.
- **Training Ground** is a tool that helps in acquiring the justified results for short simulations that use online-learning-based models in their autoscaling policy. The challenge for these policies originates from the insufficient simulation time to achieve model accuracy that is appropriate to be used for drawing the autoscaling decisions. To overcome this challenge, **Training Ground** pre-trains the model by running the simulation over a limited set of load patterns presented to it. These 'training' simulations are repeated multiple times with each load trace being drawn randomly from the set. When the achieved accuracy is considered to be sufficient, the path to this model can be supplied as one of the parameters to the evaluated autoscaling policy. An additional advantage of this tool is that the models can be directly shipped into the production environment if it happens to use Python, Keras, and Tensorflow for cloud operations automation.

The following subsections contain the detailed explanation for each tool except for **Training Ground**. **Training Ground** is straightforward in its implementation and is thus not worthwhile to discuss.

9.2.2 Multiverse: Simulation-driven Autoscaling Policies Design Space Exploration

9.2.2.1 Key design choices

Multiverse is an event-driven autoscaling simulator that acts on the level of individual requests. This fine granularity allows the simulator to achieve high precision for exploring the tail latency effects. The following key choices were made in the simulator's design:

- **The simulation step has an order of tens of milliseconds**. The value commonly used in the experiments is 10-50 ms. Such a small simulation step allows to better capture the impact of the simulated autoscaling policy on requests with milliseconds-scale response time targets.
- **The simulated behavior, including the autoscaling policy, is governed by the abstractions presented in Section 9.1**. Each discussed abstraction captures a particular aspect of the simulated behavior. The application abstraction is at the top level. It is responsible for the requests propagation through the application services according to the simulated application topology. All the other complementary behaviors are encapsulated by other abstractions.

- **Simulated VMs form node groups that are treated as a collective entity**. Large-scale deployments may involve hundreds of virtual machines [163]. Simulating all these VMs individually would significantly delay the simulation results. Thus, in **Multiverse**, we trade the accuracy in simulating the resource utilization and other platform-specific effects for the fine-grained simulation of user-level metrics such as the response time. Each node group consists of nodes of the same type (flavor) offered by the cloud services providers. Multiple node groups with nodes of different types can coexist in the simulation for a single application.
- For the similar reason as above, **we do not simulate the effects of system software and the architectural characteristics of physical nodes (e.g. caching, NUMA)**. One may object that these effects significantly impact the performance of the software running on it. Indeed, the node architecture has enormous implications for bulk-synchronous workloads such as distributed training of deep networks. Even a single under-performing node in the cluster (so-called straggler) may delay the training in the current round [164]. However, this thesis focuses on microservice applications that do not suffer from the stragglers to such a great extent. In the end, the request may be retried by the client or rerouted through a lesser loaded service replica.
- **Focus on simulating the horizontal scaling**. This point follows from the simulation of node groups instead of individual VMs. Simulating vertical scaling (i.e. in-node scaling) requires to simulate each virtual node individually to get actionable results. Albeit studying the interplay between horizontal and vertical scaling may be rewarding for better cloud resource management, vertical scaling is bound to a node and exposes relatively little distributed systems aspects. This design choice may be considered a limitation of the **Multiverse** simulator.
- **Modularity of the modeled autoscaling policy**. First, the autoscaling policy is represented by two 'sub-policies' defining the scaling behavior on each resource abstraction level, i.e. a *service scaling policy* (aka Scaling Policy from Section 9.1) and an *infrastructure adjustment policy* (aka Adjustment Policy from Section 9.1). The next level of modularity is that both of them are composed of multiple building blocks. These building blocks act together to produce the desired state. Obeying the communication contracts imposed by the abstract interfaces, one can add new building blocks and compose a custom policy to simulate.

Next, we discuss the representation of the service scaling and infrastructure adjustment policies.

9.2.2.2 Service scaling policy implementation

As was discussed in Section 9.1, the abstraction of the service scaling policy is *decentralized*, i.e. the autoscaling decisions are taken for each service individually. Recalling the discussion from before, a centralized autoscaling authority would have been a clear bottleneck and a single point of failure for the distributed application. A well-designed distributed management system would avoid having such a drawback. For instance, Kubernetes puts HPA controllers in charge of separate pods, thus relying on the resource headroom provided to the pods such that it could catch-up with the load change at the next state reconciliation event. Nevertheless, neither Kubernetes nor the **Multiverse** simulator necessarily suffers from the 'split-brain' problem – one is free to incorporate the metrics from other services (pods) either directly or indirectly into the scaling management process for another service (pod).

The quantitative information required for scaling a service is provided by a set of *metrics groups* associated with the given service. Each metric group implements a sequence of operations to convert the raw time series data into the desired state of this service. These operations correspond to their abstract counterparts

described in the Subsection 9.1.7.2 that features the Scale phase of the SPACE process. Some operations are performed for each individual metric, whereas several final steps are done at a group level.

Metric. Let us start with the scaling-related operations implemented for an individual metric:

- **Querying metrics from the related service**. This operation allows to support the application topology-awareness in the otherwise per-service scaling policy. Currently, this operation gets metrics values from the immediate neighbors of the service on the request path. Querying can be customized to get the metric values from an arbitrary service if the researcher finds such a customization meaningful.
- **Filtering raw metric values** both from the current service and from the related services (if any). Filtering has multiple purposes. First, it removes the undefined values, e.g. if a metric was not sampled at the particular time. Second, it removes the undesirable values, e.g. outliers. Lastly, it smooths the time series to reduce the oscillations that may lead to the instability of the desired state. Although one can again provide her own filter that implements the `_internal_filter` method, currently, the implementation of **Multiverse** has a sufficient number of filters available. These filters are: *Default NA* that substitutes the undefined values for a value specified in the configuration, *Outliers Remover* that substitutes the outliers for the interpolated values, *Simple Exponential Smoother* that applies simple exponential smoothing to the time series, *Holt Smoother* with just a level and a trend, *Holt-Winters Smoother* that extends the Holt smoother by considering the seasonality, and several more advanced smoothers (*Butterworth*, *Christiano-Fitzgerald*, and *Hodrick-Prescott*).
- **Determining the correlation between the metrics**. This operation is implemented in a *stateful* way. Internally, the correlator component accumulates the filtered metrics values both from the current service and from the related one. The size of the accumulating buffer is specified in the configuration. When called to provide the correlation for two signals, it first stores them, then resamples both time series s.t. they have the common resolution, and, lastly, computes a single correlation value. The computation of the correlation value is implemented by shifting one of the time series by a *lag* taken from a finite interval along the time axis and calculating its correlation with the other unshifted series. The boundary of this symmetric interval is defined in the correlator configuration. Finally, the lag with the maximal correlation value is selected. These lags and correlations are computed for multiple related services if there are more than one and are returned to proceed with scaling. Although the high-level lagged correlation computation is 'set in stone', one can customize the correlation computation by providing a class with own implementation of the `_compute_correlation` method.

  At the moment of writing, several groups of correlation computation are supported. First, there is the *linear Pearson correlation* and *nonlinear Spearman correlation*. Second, there is a family of maximal information-based nonparametric correlators [165], i.e. *Maximal Information Coefficient* (MIC) and its generalized version (GMIC), *Maximal Asymmetry Score* (MAS), *Maximal Edge Value* (MEV), and *Total Information Coefficient* (TIC). These are provided in the `minepy` package[4]. Lastly, there is a group of estimators that output the distance correlation coefficient which measures both the linear and nonlinear association [166]: a *biased estimator for the distance correlation*, a *bias-corrected estimator for the squared distance correlation*, and an *affinely invariant distance correlation estimator*. The implementations of these estimators are provided by the `dcor` package[5].
- **Aggregating the metrics**. This operation recasts metrics values into a particular resolution by applying an aggregation operation in a rolling time window. Aggregation is done directly before forecasting to expose the time-based patterns in the given metrics (e.g. some patterns may be observed only at

[4] `https://pypi.org/project/minepy/`
[5] `https://pypi.org/project/dcor/`

particular time resolution, take for instance 1-min vs 1-hour resolution for visits to the web-shop). Another less obvious reason is that some forecasting methods tend to preserve a state which is bound on the time series resolution. Finer time series resolutions (e.g. minutes or seconds) may yield hundreds of megabytes or even couple of gigabytes of main memory taken to keep the state around. There are several aggregation operations currently supported by **Multiverse**: taking minimal or maximal value, taking average in the time window, and taking the quantile in the given time window with median being provided separately for easier accessibility. To expand the collection of aggregators, one needs to implement the `aggregate` method.

- **Forecasting the metric values**. This operation is implemented in a stateful manner. Such an implementation is especially important to the forecasting models that do not allow incremental updates, i.e. each new additional piece of data requires the model to be fit anew. Since the data accumulation for fitting a forecasting model might take some time, early in the simulation this operation is performed using the fallback *reactive* 'forecasting' model. The reactive model simply repeats the last observation of the metric for the whole forecasting horizon which is a parameter to the forecasting model. The implementation puts the reactive model under the label of 'forecasting' to ensure uniformity. First and foremost, the reactive model serves to support the reactive autoscaling policy implementation.

  Each forecasting model present in the simulator should implement two methods, namely, `_internal_fit` and `predict`. At the moment of writing, the following forecasting models can be selected in the simulator: *simple Average* which averages several last observations and presents the result as a forecast, the familiar *Holt-Winters* smoothing that can be extrapolated into the future, the classic machine learning *Support Vector Regression* (SVR) [140] model, the single-layer long short-term memory (LSTM) recurrent neural network (RNN) [167], and the *Singular Spectrum Analysis* method of signal decomposition [138, 139]. A common forecasting tool, namely, the ARIMA process is represented with several members of its family: *Autoregressive Integrated Moving Average* (ARIMA) itself, *Autoregressive Moving Average* (ARMA) process, *Autoregressive* (AR) process, *Moving Average* (MA) process, and the seasonal version of ARIMA (SARIMA). Most of the mentioned forecasting methods and models were evaluated and discussed at length in Chapter 6. Lastly, one can build an *ensemble* of forecasting models. The independent forecasts of models in the ensemble are weighted based on the model configuration and then averaged.

Although most of the resource management systems allow to leverage arbitrary metrics, in practice, only a handful of these are ever considered. The **Multiverse** simulator, in particular, supports the following quantitative metrics: resource utilization (vCPU, memory, used network bandwidth, used disk space), response time, count of requests waiting in service buffers, average time spent by requests waiting, and the user load. One may suggest numerous other metrics, but they will most likely fall in either of two categories: *application-specific* (e.g. count of database threads, DB query parameters count) or *platform-specific* (e.g. cache misses, deployment costs). Specialization is indeed a powerful concept in systems optimization, but it precludes one from discovering solutions for broad classes of applications.

Metric group. An abstraction of a metric group is introduced to enable the use of multiple metrics when determining the desired count of service instances. Once the forecasted values for every metric in a group are calculated, the following operations are performed on a group:

- **Querying current service instances count**. Since horizontal scaling results in changing the count of service instances (replicas), their current count represents the existing state. This count acts as a baseline which might or might not end up being changed. The count of service instances may be generalized to the *scaling aspect* concept. This concept represents a quantitative measure that is changed as a result of a scaling action. In vertical scaling, the scaling aspect might be the amount of

CPU shares allocated to the container running on a node. To not mislead the reader, we will focus on the count of services as a scaling aspect.

- **Calculating the desired count of service instances**. The desired count of service instances is calculated in two steps. First, one of the implemented calculators is used to get the desired count of service instances based on the current count of instances, current metric value and the forecasted metric values. Second, the calculated value is adjusted using a heuristic, e.g. the count is increased or decreased by some percentage of the calculated number. The second step might appear bogus, but there is evidence of such heuristics being used in the production resource management and container orchestration systems (cf. Vertical Pod Autoscaler of Kubernetes). **Multiverse** supports two kinds of desired instance count calculators explained in the following paragraphs.

  I. Rule-based desired service instances count calculator implements a deterministic rule to compute the desired count of service instances. Such a rule usually includes a fixed *threshold* for the metric or a group of them. The need for the scaling action is derived from the metric crossing this threshold either from top or from bottom. The current version of the simulator supports only the most prominent rule, viz, the *ratio rule*. According to the ratio rule, the desired service instances count is computed as in Equation 9.1:

$$desiredServiceInstancesCount = \frac{forecastedMetricVals}{targetMetricVal} \cdot currentServiceInstancesCount \quad (9.1)$$

  II. Learning-based desired service instances count calculator differs considerably from its rule-based counterpart. As the name suggests, such calculators attempt to learn the model of how the metrics and the service instances count map onto some quality metrics, e.g. the response time. Having this performance model allows to use the optimization techniques to find service instances count that will meet SLOs with given metric values (cf. Subsection 7.3).

  Every learning-based calculator is stateful and performs the same sequence of steps. First, upon each invocation with the data provided, it stores the provided data for micro-batch learning. Next, if there is enough data, the encapsulated model is updated by performing a training step. If the accuracy of this model is considered sufficient (as per settings), then it can be used directly in the following computation, otherwise a fallback calculator is used (e.g. the rule-based one). Assuming that the mapping model already performs well-enough, it is used by an optimizer to search for an instances count that allows to meet the quality goal expressed as a nonlinear constraint. The goal function is specified as a ratio of the quality metric as predicted by the model to the threshold (SLO, e.g. response time should be not more than 300ms). At the moment of writing, **Multiverse** supports both linear and nonlinear mapping models.

  The linear class is represented by the *Passive-Aggressive* [168] and *Stochastic Gradient Descent* online linear regression techniques. Both can also perform their duty in an offline mode, but this thesis treats the offline regression models in the context of cloud applications rather sceptically. The reason is two-fold. First, to maintain the reasonable accuracy, an ever increasing amount of data should be carefully maintained and curated which is infeasible for most practical applications. Second, the model fitting time grows with the size of the dataset kept.

  The nonlinear class of learning models is represented only by a feed-forward fully-connected neural network of an arbitrary depth. Having just a single model in this class may still be sufficient since any neural network is a universal nonlinear functions approximator [169] and since the configuration file allows to specify an arbitrary architecture of the neural network allowed by Keras[6].

[6] `https://keras.io/api/layers/`

- **Stabilizing the desired service instances count**. Despite the efforts to smooth metrics values during the filtering operation, the derived service instances count may still exhibit high variability along the timeline. Frequent scaling actions are considered bad in cloud applications management since they yield high resource waste. In addition, one cannot hope to account for all the possible load variations. Hence, some seemingly redundant capacity headroom, that will otherwise be taken away by a short dip in the future desired service instances count, might prove to be necessary to accommodate an unpredicted load spike. As of now, **Multiverse** supports only two kinds of stabilizers – min (likely under-provisioning) and max (likely over-provisioning). Both resample the given time series of desired service instances count at the given resolution and aggregate all the values in the resampled intervals according to their unique operation.
- **Limiting the desired service instances count**. In certain situations, it may be meaningful to provide bounds on the desired service instances count. For instance, the lower bound of 1 service instance disallows the autoscaler to undeploy the service entirely in the case of low load. Similarly, service instances count may be forced to stay below a certain level to not run out of budget. These two cases highlight the purpose of the last operation in the sequence, i.e. limiting the count of service instances. This operation, however, serves an additional purpose when multiple metric groups are used to determine the desired service instances counts and their outputs should be sequentially combined. More on that in the following paragraphs.

Since there could be more than a single metric group, the desired state of a service computed by each group should be aggregated. It is done either *sequentially* or in *parallel* by a *scaling aggregation rule*.

Sequential aggregation rule leverages the limiting operation of the metric group that was discussed previously. Metric groups are ordered according to the numeric priority that is assigned to each group in the configuration file. The computation of the desired service instances count proceeds from the group with the highest priority to the group with the lowest. Before handling the computation to the next in line, the current metric group updates the limits set in it based on the desired service instances count computed by itself (cf. Figure 9.4a). The upper bound corresponds to the max stabilization operation, whereas the lower bound corresponds to the min stabilization operation. The limits are the time series, not single values. The final output is produced by the lowest-priority metric group.

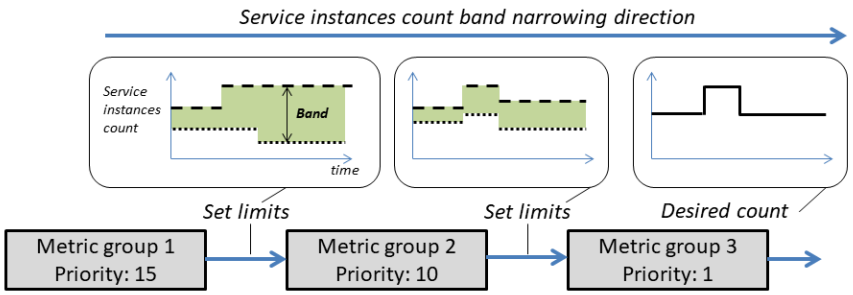
Parallel aggregation rule allows every metric group to calculate the desired service instances count independently of its peers. Later, these results are aggregated e.g. by taking min or max of the metric groups outputs which is shown in Figure 9.4b. A new parallel aggregation rule can be added by registering it with the `ScalingEffectAggregationRule` base class and overriding the initialization of the parent `ParallelScalingEffectAggregationRule` class.

9.2.2.3 Infrastructure adjustment policy implementation

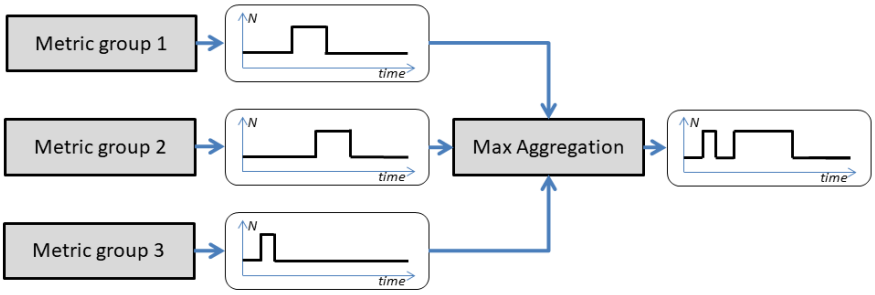
Virtual infrastructure adjustment is done after the derivation of the desired services states. The implementation of the adjustment policy in the **Multiverse** simulator follows the **Place-Score-Optimize** process outlined in the Subsection 9.1.7.3. This process is captured by *Desired Platform Adjustment Calculator*. It comprises three components:

- **Placer** forms in-node service instances placement proposals for each node type. In doing so, placer follows one of the following strategies.

  Use existing mixture of nodes. Placer attempts to use the node types that were specified in the application deployment configuration. If no options are produced with this strategy, it proceeds to the next in line with a bit more relaxed conditions.



(a) Sequential desired service instances count aggregation as a sequence of service instances counts band narrowing operations.



(b) Parallel desired service instances count aggregation by taking max value at each timestamp.

Figure 9.4.: Examples of the desired service instances counts aggregation rules.

Balance the use of the shared nodes. Placer attempts to select the node types that can accommodate a mixture of service instances with complementary resource requirements. For instance, it is desirable to put the compute-intensive and memory-intensive service instances on the same node according to this strategy. If this does not succeed, then Placer tries out simple nodes sharing defined below.

Share nodes between services. Placer attempts to bundle the instances of different services to be placed on the same node to maximize the resource utilization but without necessarily balancing the resource usage. If this is not possible due to competing resource requirements of the services, then 'one node-one service' options are considered.

Place service instances on the specialized nodes. Placer uses the information provided in the labels of the service to deduct the appropriate node types and tries to pack as many service instances as possible on such nodes. If this does not work, then placer proceeds to its last resort.

Place one service instance on a single node. Placer selects the least powerful node type that is able to host a single instance of the given service. This option is considered the least desirable in the whole placement algorithm, but is added as a fallback scenario s.t. at least one placement option would be available in the generated proposals.

The placement proposals formed by Placer are passed to Scorer.

- **Scorer** consumes the placement proposals and evaluates them according the adjustment goal specified in the policy. Currently only the *cost minimization* goal is supported by the **Multiverse** simulator, although there seems to be much promise in adding the *performance maximizing scorer* and *resource utilization maximizing scorer*. The cost-minimizing scorer assigns a score to each placement option that is computed as the inverse of the total costs required to keep the placement for the given duration of time. The scored placements are then handed over to the optimizer.

- **Optimizer** is tasked with the selection of a single most appropriate placement option based on scores set earlier. **Multiverse** currently implements only a single optimization strategy – the selection of the placement proposal with the highest score.

Each part of the adjuster discussed above might be substituted for a custom one by implementing the corresponding interface. In case of Placer, the expansion can also be done at a finer level by adding a new placing strategy and including it into the placing algorithm.

9.2.2.4 Simulation configuration files

Including the two policies discussed in greater detail in the previous subsection, there is a total of 8 mandatory JSON configuration files governing the simulation run. Below is the description of each of these files using its default name:

- **confs.json** contains paths to the configuration files below.
- **application_model.json** specifies the services that the simulated application consists of. It also specifies the application topology, the requests types with the resource requirements, and the time intervals required by services to process the request both in upstream and the downstream mode.
- **platform_model.json** specifies the resource and pricing configuration of various VM types offered by cloud services providers.
- **load_model.json** specifies the patterns of load distribution in time or the actual traces.
- **deployment_model.json** specifies the regions which the application services are deployed in, cloud providers, node types, and the initial count of nodes and service instances.
- **scaling_model.json** specifies booting and termination times both for the services and node types.
- **scaling_policy.json** specifies the building blocks of the simulated services scaling sub-policy (part of the simulated autoscaling policy).
- **adjustment_policy.json** specifies the building blocks of the simulated platform adjustment sub-policy (part of the simulated autoscaling policy).

9.2.3 Stethoscope: Understanding Quality and Behavior of Autoscaling Policies

9.2.3.1 Purpose

Stethoscope visualizes the simulation results using the underlying `matplotlib` package[7]. The plots produced by this tool are split into two broad categories. The first characterizes the quality of the simulated autoscaling policy. The second characterizes the internals of the simulated autoscaling behavior. Let us look at what plots are provided in each category.

[7] `https://matplotlib.org/`

9.2.3.2 Plots that characterize quality of an autoscaling policy

Response time is the most important user-centric metric selected to evaluate quality of the simulated autoscaling policy for the purpose of this thesis. This metric is captured by two plots. First, the *cumulative distribution of requests by response time*, aka CDF plot, that is widely used in the research literature on the topic. Second, a bit more straightforward *histogram of requests by response time*. Essentially, both convey the same information, but differ in how it is presented. The CDF plot better captures the general tendency, whereas the histogram better highlights behavioral patterns, e.g. when the requests cluster in particular response time intervals.

Response time based metrics fail to capture how many requests ended up being processed in time. Since **Multiverse** adopts dropping the requests once their lifetime crossed the timeout value, it is easy to track the count of fulfilled requests by comparing the amount of generated requests to the count of responses recorded at the point of injection. **Stethoscope** visualizes these data as a *barplot* with a bar split into two parts – one for the count (or percentage) of fulfilled requests, and another for the count (or percentage) of failed requests. A good autoscaling policy would aim both at minimizing the count of failed requests and at cutting the latency tail. Therefore, this plot and the plots from the previous paragraph provide a comprehensive picture of how well does the autoscaling policy respond to the user's expectations.

There are two more parties interested in how well does the autoscaling policy perform, namely, the application owner and the cloud services provider. However, these parties have a slightly different notion of what a good autoscaling policy means.

The aim of an application owner is indeed in maximizing the user satisfaction (which is captured by the plots discussed above), but she is also interested in *minimizing the the deployment costs*. **Stethoscope** thus provides an additional *line graph* that captures how the total cost of the cloud deployment amounts over time. This is an ever increasing value, and the last data point captures the total amount billed. To build this plot, **Multiverse** allows to include the costs of virtual nodes per unit of time into the platform model configuration file.

On the other hand, the cloud service provider perceives the high quality autoscaling as *maximizing the resource utilization of the nodes*. Since the simulated nodes are VMs, without the loss of generality we may assume that CSP offers a FaaS solution. **Stethoscope** offers *line plots* for each type of system resource utilized on deployed virtual nodes. These are the common plots that show the resource utilization over time.

9.2.3.3 Plots that characterize the autoscaling behavior

The autoscaling behavior implemented by the simulated autoscaling policy has many manifestations and can be captured on both abstract resource levels.

In order to understand how well does the simulated autoscaling policy respond to a particular load, **Stethoscope** offers a *line plot* that shows the change in the incoming load over time. The plot is broken down by the request type. Such a plot is able to capture the request rate variability over time and is thus convenient to track how well does the autoscaling policy respond to a particular *load pattern*, say, a rapid surge in the load followed by almost an instant return to the constant level. This plot is complemented by a *bar plot* that shows the count of generated requests by type. This visualization allows to better track the load mixture that the current simulation is being exposed to.

Booting and terminating delays for VMs are captured by plotting how the desired and actual count of nodes changes over time [11]. **Stethoscope** offers these *line plots* as well. The value of these plots is in providing an opportunity to estimate how different scaling delays impact the quality of the autoscaling as seen by the

user and how well are these delays addressed by the simulated autoscaling policy. Since the application services delays are almost non-existent (e.g. 30 ms in case of light containers [11]) and the application scaling actions map quite accurately onto the changes in the VMs count, the same plot for the services counts changes is redundant. In contrast, **Stethoscope** provides several other plots that far better characterize the autoscaling behavior on the application level.

The first such plot provides a bird's-eye view on how the requests proceed through the simulated application. This plot captures the distribution of the request/response lifetime among three states: being processed, being transferred over network, and waiting in a service buffer. The plot aggregates these results for all the requests of the same type and draws a *bar plot* that is similar in its organization to the fulfilled/dropped bar plot from the previous subsection.

The second application-specific plot gets into more detail by showing how much time did the request of each type spend waiting in the buffer of each application service. Such plot allows to identify the bottlenecks under the simulated autoscaling policy. By studying their characteristics, such as the count of connected services and the resource demands, one can figure out what would be the practical limitations of the designed autoscaling policy and whether and how it could be improved.

9.2.3.4 Comparative plots

The plots discussed above are provided for a single simulation. Such plots are useful when designing an autoscaling policy since they provide a rapid feedback on its qualitative and behavioral characteristics in the given setting (load, platform, application topology, and resource requirements). However, when an appropriately tuned autoscaling policy is finalized, it might be required to compare it against another tuned policy or a subversion of itself tuned for another setting. To enable this, **Stethoscope** offers a comparative version for three of the discussed plots, namely, response times distribution (CDF), fulfilled vs dropped requests (percentage only), and the distribution of the request/response lifetime among being processed/in transfer/waiting states (percentage only). In these comparative plots, the data points for every compared autoscaling policy are simply added on the same canvas to ease the visual comparison. These plots are leveraged by the **Cruncher** autoscaling simulation automation tool which is discussed next.

9.2.4 Cruncher: Evaluation of Autoscaling Policies Alternatives

When multiple alternative service scaling policies and/or platform adjustment policies or their building blocks are designed, one can use **Cruncher** to compare those in the same setting which includes the same application, the same load generation pattern and the same platform offering. In doing so, one is able to construct a design search space for these policies.

To accomplish its goals, **Cruncher** offers to distribute the simulation configuration files into two folders. The first folder contains the fixed configurations that do not change from experiment to experiment. For instance, such a folder can contain an application model configuration file if the alternatives are evaluated on the same application. The second folder shelters the alternative policies to be evaluated. This folder has a distinct structure – it requires one to put the alternative policies into the sub-folders that correspond to their respected categories, say, `scaling_policy` or `adjustment_policy`. Naturally, inside these sub-folders, the alternatives should have distinct names. This requirement is conveniently imposed by the file system itself.

When the simulation configuration files and the alternatives to be evaluated ended up in the folders, it is time to write the configuration file. The configuration file for **Cruncher** consists of two parts. The first part specifies the simulation configuration such as simulated starting time, simulation step, and the interval of

time that will be simulated. Thus, all the simulations are exposed to the same timing conditions. The second part of the file comprises the configuration of the experiments performed by **Cruncher**. One specifies the regime, the relative paths to the folders with the constant configurations and the alternative policies to be evaluated, the number of times the simulation is repeated to get the reliable results, the folder where the simulation results (plots and the report) will be stored, and whether or not one is willing to keep the final evaluated configuration mixes which are otherwise automatically deleted.

Cruncher explores all the possible combinations of alternatives by picking one configuration from each category at a time. Thus, **Cruncher** explores the Cartesian product of sets corresponding to the files put into the alternatives folders. This means that the total amount of simulations done can be computed as:

$$N = r \cdot \prod_{i=1}^{C} n_i \tag{9.2}$$

In Equation 9.2, r is the number of times each simulation is repeated, C is the count of policies categories evaluated, and n_i is the count of alternative policies in each category. Given this full exploration space looked at by **Cruncher**, we would like to discourage the reader from running it over all the possible alternatives on the policy design stage.

As was mentioned in the previous subsection, **Cruncher** leverages **Stethoscope** for building the comparative plots in several categories. Before doing so, **Cruncher** prepares the results of the simulations, in particular, it aggregates them for each alternative evaluated across all the simulation runs. In addition, the data collected for each simulation run is stored on disk to incrementally expand it upon need to ensure the reliability of the results at the analysis stage.

Based on the simulation results, **Cruncher** generates a report summarizing the core quality and behavioral metrics in the tabular form as shown in Figure 9.5. These metrics are the total cost at the end of the simulation, the count and percentage of the requests that met the service level objective, the average resource utilizations by service and by resource type, and the average nodes counts by provider and node type along with the standard deviation in these counts over time (to estimate the stability). The tabular form of the report is courtesy of the `prettytable` package[8].

Although the discussion above focused on evaluating the scaling and adjustment sub-policies alternatives, **Cruncher** supports the evaluation of alternative models (applications, deployments, loads) in the same way.

9.2.5 Praxiteles: Automatic Generation of Credible Autoscaling Simulation Configurations based on Traces

9.2.5.1 Recipes and the simulation configurations generation process

The major threat to validity of the autoscaling policy simulation is in the gap between the hand-written experimental configuration and the real applications and loads. **Praxiteles** attempts to bridge this gap by automatically generating some simulation configurations using the published load and resource utilization traces as well as the generative models that are based on empirical analysis of published applications.

The configuration generation process in **Praxiteles** is governed by the so-called *recipes*. Each recipe is a meta-configuration file that describes how the set of simulation configurations files should be generated. A recipe comprises two parts.

[8] `https://pypi.org/project/prettytable/`

```
--------------------------------------------------------------------------------------------
------------------- SUMMARY CHARACTERISTICS OF EVALUATED ALTERNATIVES -----------------------
--------------------------------------------------------------------------------------------

Alternative 1: [adjustment_policy -> cost_minimization; scaling_policy -> neural_net]

>>> COST:
+----------+--------+-----------------+
| Provider | Region | Total cost, USD |
+----------+--------+-----------------+
|   aws    |   eu   |     0.00098     |
+----------+--------+-----------------+

>>> REQUESTS THAT MET SLO:
+--------+-----------------------------------------+-----------------+-------------+
| Region |                Request type             | Total generated | Met SLO (%) |
+--------+-----------------------------------------+-----------------+-------------+
|   eu   | req-75c4a762-4da4-11eb-bb92-d8cb8af1e959 |        33       |  4 (12.12)  |
+--------+-----------------------------------------+-----------------+-------------+

>>> AVERAGE RESOURCE UTILIZATION:
+-----------------------------------------+--------+----------+--------+--------+----------------------+
|                 Service                 | Region | memory, % | disk, % | vCPU, % | network_bandwidth, % |
+-----------------------------------------+--------+----------+--------+--------+----------------------+
| service-75c4ce5c-4da4-11eb-b901-d8cb8af1e959 |   eu   |    5.36   |   0.05  |   0.81  |          0.0         |
| service-75c4ce5d-4da4-11eb-85d9-d8cb8af1e959 |   eu   |   23.47   |   0.06  |   6.48  |          0.0         |
| service-75c4ce5f-4da4-11eb-ac60-d8cb8af1e959 |   eu   |    7.7    |   0.0   |   2.13  |          0.0         |
| service-75c4ce5e-4da4-11eb-899a-d8cb8af1e959 |   eu   |   19.28   |   0.0   |   2.72  |          0.0         |
| service-75c4f5fb-4da4-11eb-9534-d8cb8af1e959 |   eu   |   15.09   |   0.1   |   3.7   |          0.0         |
| service-75c5434c-4da4-11eb-941c-d8cb8af1e959 |   eu   |   43.83   |   0.1   |  12.24  |          0.0         |
| service-75c4f5fa-4da4-11eb-a82c-d8cb8af1e959 |   eu   |    5.32   |   0.19  |   1.47  |          0.0         |
+-----------------------------------------+--------+----------+--------+--------+----------------------+

>>> NODES USAGE BY TYPE:
+----------+--------+-----------+-------------------------+-------------------------+
| Provider | Region | Node type | Desired count avg (std) | Actual count avg (std) |
+----------+--------+-----------+-------------------------+-------------------------+
|   aws    |   eu   |  t3.nano  |       10.0 (±0.0)       |       10.0 (±0.0)       |
|   aws    |   eu   |  t3.micro |       12.0 (±0.0)       |       12.0 (±0.0)       |
+----------+--------+-----------+-------------------------+-------------------------+
```

Figure 9.5.: Part of a textual report generated by **Cruncher** for an experiment with two alternative scaling policies. The results only for the first alternative are shown.

The *general* part of a recipe is filled with the manually selected parameters of the configuration files to be generated. There, one specifies either finite sets of parameter values or the interval boundaries if the parameter can take on any value from an interval uniformly. An example of system requirements recipe specification may include a list of options for the mean resource utilization as well as for the standard deviation with a unit (cf. Listing 9.1). If a concrete value is preferred, then it can also be directly specified. As one might have noticed, instead of simulating the stable resource consumption over time, we sample the normal distribution of the system resource requirements with the mean and standard deviation parameters. This allows to simulate the deviations in the resource utilization. Similarly, in the deployment recipe one may specify an interval, from which the count of the services in a particular region could be sampled as in Listing 9.2. The numeric parameter 1.0 in the providers and regions parts of the recipe denotes how the deployed instances will be distributed among the providers and regions.

```
1 "system_requirements": {
2     "memory": {
3         "mean": [1, 2, 3],
4         "std": [0.1, 0.2, 0.3],
5         "unit": "GB"
6     }
7 }
```

Listing 9.1: An example of a memory requirements generation excerpt from an application model recipe.

```
"deployment_recipe": {
    "providers": {"aws": 1.0},
    "regions": {"aws":
        { "eu": 1.0 }
    },
    "init_aspects": { "count":
        {"min": 1, "max": 10}
    }
}
```

Listing 9.2: An example of a deployment model recipe.

Another valuable component of **Praxiteles** is *Application Structure Generator*. It allows to generate a realistic application topology based on the results from Chapter 8. **Praxiteles** was extended with the Barabási-Albert random graph model (BA-model) with four sets of two parameters' values that correspond to the midpoints of intervals discovered in the study. This model allows to generate realistic microservice application topologies with an arbitrary number of services. These parameter sets with their names as seen in the **Praxiteles** are shown in Table 9.1. The graph generation itself is done using the **igraph** package[9].

Name	*Power of preferential attachment*	*Appeal of vertex with 0 edges*
tiered_with_single_center_a	0.05	0.01
single_center_b	0.9	0.01
tree_with_multiple_centers_c	0.5	3.25
pipeline_with_multiple_centers_d	0.9	3.25

Table 9.1.: Parameter sets of the BA-model for generating realistic application topology graphs. The default parameter set is in bold.

The *specific* part of the recipe fed to **Praxiteles** configures use of the published traces to make the generated simulation configurations more realistic. The specific part is simply a list of named configurations that correspond to the classes implementing the same interface. Each such class corresponds to a single dataset that it processes to extract the parameters that can be used to enrich the original recipe for **Praxiteles**. At the moment of writing, only two such datasets are supported. Both are from Microsoft: the first one is *Azure functions* [31] that provides insights into the application-level parameters as seen by a FaaS provider, and the second one is *Azure VM resource utilization data v2* with a focus on the CPU utilization [30]. The usage of both datasets to generate the realistic simulation configurations is discussed in the next subsection.

The realism of the VMs and service instances booting and termination times is ensured by the empirical observations provided in the research papers. The scaling latencies for service instances for main cloud services providers were published in [170]. VMs booting and termination times are better researched, and we used two research papers as sources for the VM booting and termination times estimates [171, 172].

The configuration generation process in **Praxiteles** is rather straightforward. First, upon discovering non-mandatory configurations of the trace-based generators, it uses them one by one to enrich the general recipe according to the logic implemented in these generators. The only requirement to such generators is that they implement the `enrich_experiment_generation_recipe` method. The method should not return anything – it simply enriches the general configuration recipe which is accessed by reference. When this is done,

[9] `https://igraph.org/python/doc/igraph.GraphBase-class.html`

the generation process continues by parsing the recipe. During this parsing process, the corresponding simulation configuration files are generated by *uniformly* sampling one of the offered parameter values when such an opportunity is offered. However, some parts of the recipe, that are generated using the trace datasets, allow to sample the parameter values such as mean and the standard deviation from the *empirical distribution*. The next subsection discusses how this option is enabled by the Azure datasets.

9.2.5.2 Trace-based generation of realistic simulation configurations

Even though the traces may share the information that they disclose about the application and/or platform and/or load, one can assign priorities to allow the higher-priority generators to overwrite the results of the lower-priority ones. Both Azure datasets considered in this subsection have little to no overlap in the information that they offer. If the datasets are not present on the machine, the generators download and unpack the files that are specified in the configuration.

Azure Functions (2020) Based Experiment Generator. The Azure Functions dataset [31] is used in **Praxiteles** to derive the *count of services* for the application recipe, the *duration of requests processing* by service instances for the requests recipe, and the *requests rate and its distribution in time* for the load recipe. Since the dataset is quite large for in-memory processing on a single machine (all the unpacked CSV files take roughly 2 GBs of disk space), one is allowed to select a subset of files to be used for the trace-based generation. The high-level applications data in the dataset differs heavily by the amount of invocations that they get from the end users (requests), hence one is free to specify the quantiles of interest to vary the scale of the simulated applications. First, a subset of applications is selected based on the invocations count belonging to the desired quantiles interval. This results in applications sharing common user load properties. The selected subset of data is then used to derive the parameters mentioned above.

The count of services is set as an α-quantile of the unique `HashFunction` field which contains hashed application function identifiers. Parameter α is specified in the generator configuration (cf. Listing 9.3 for the generator configuration example where α is 0.8).

```
{
    "name": "azurefunctions",
      "config" : {
        "data_path": "./azurefunctions/",
        "file_id": 1,
        "consider_applications_with_invocations_quantiles": {
          "left_quantile": 0.7,
          "right_quantile": 0.9
        },
        "app_size_quantile_among_selected_based_on_invocations": 0.8
    }
}
```

Listing 9.3: Recipe excerpt for the simulation configs generation using the Azure Functions dataset.

As the timestamps in the dataset are anonymized, the derived application invocation time series with a 1 minute resolution is resampled at a 1 hour resolution for a period of 24 hours and added to the load recipe for every month and day of the week. This is done to minimize our assumptions about the data and to allow one to select an arbitrary starting point and the duration of the simulation without caring to catch the right interval with at least some load.

Lastly, the requests recipe is enriched with the requests processing duration configuration that is aggregated with the percentiles summary in the dataset. In particular, the dataset offers the following duration percentiles: 1%, 25%, 50%, 75%, 99%, 100% as well as the min and max values.

Azure VM Traces v2 (2019) Based Experiment Generator. The Azure VM traces dataset v2 [30] is used in **Praxiteles** to derive the empirical distributions for vCPU and memory requirements' means and standard deviations. These resource requirements distributions are derived both for the requests and the services. The derivation process is governed by the configuration (cf. Listing 9.4), which, among all else, specifies the files that should be downloaded and used by the generator. Setting just one or a few ids will dramatically spare the disk space and the memory since the files in this dataset are big (e.g. one of 195 VM CPU readings files takes roughly 1.25 GBs on the drive). The large size of the files in the dataset forces to read records from them in batches of size set by the `csv_reading_batch_size` parameter in the configuration.

```
{
    "name": "azure-vms",
    "config": {
        "data_path": "./azuredata-resourceutil/",
        "file_ids": [1],
        "vm_category": ["Interactive"],
        "unique_vms_selected_in_each_cpu_readings_file": 200,
        "percentage_gap_to_be_considered_single_req": 3,
        "bins_for_empirical_distribution_count": 10,
        "cpu_to_memory_correlation": 0.9,
        "rescaling_factor_requests_memory_requirements": 0.1,
        "csv_reading_batch_size": 100000
    }
}
```

Listing 9.4: Recipe excerpt for the simulation configs generation using the Azure VM traces v2 dataset.

Microsoft made an initial attempt to classify the VMs into several categories based on the observed resource usage patterns [30]: delay-insensitive, interactive, and unknown. One can choose any (or all) of these categories to select the subset of the data. For the purpose of this thesis, the *interactive* category is the most appropriate one. The applications in this category are marked that way because they exhibit the diurnal cycles[10] in their behavior.

Large CPU utilization files with 5-min resolution available in the dataset are complemented by a single `vmtable.csv` file which contains the general characteristics of the VMs present in the CPU utilization files. In particular, it identifies artificial timestamps of VM creation and deletion, CPU usage for VM during its lifetime (max, average, and 95%-tile), its category as described above, the bucket of virtual cores count to which it belongs to, and the bucket of memory (in GBs). The two last pieces of information are needed to convert the relative utilization numbers into the actual resource usage using the CPU readings files.

The generator configuration allows to select a subset of VMs from the CPU readings files considered by specifying the `unique_vms_selected_in_each_cpu_readings_file` parameter. The selection criteria is applied to the CPU readings files. The reason is that the user of **Praxiteles** has no means to know which of 195 CPU ~1 GB-large readings file contains a particular VM from the `vmtable.csv`. Therefore, **Praxiteles** adopts a workaround – first, the specified count of unique VM ids is selected in the CPU readings files, and then the records for these VMs are extracted from the `vmtable.csv`. The derivation of the empirical

[10] A pattern recurring every 24 hours due to rotation of the planet Earth.

distribution for vCPU and memory requirements' means and standard deviations of services and requests is done afterwards.

The 5-minutes granularity of CPU readings does not allow to accurately determine the CPU utilization for functions (read, services) and the requests. Therefore, **Praxiteles** adopts a heuristic. It considers the minimal CPU utilization to be the service CPU utilization that does not vary much with the time. Then, for each VM, a mean and a standard deviation of the minimal CPU utilization is determined. These values are then multiplied by the cores and memory allocated to the considered VM based on the data from the `vmtable.csv` file. Combined, all the means and standard deviations form an empirical distribution with the count of samples equal to the count of the selected VMs. In a similar manner, **Praxiteles** approaches deriving an empirical distribution for CPU utilization by requests. This time, however, the difference between the maximal and average CPU utilization is used to determine the observations that potentially capture the effect of just a single request on the resource utilization. To achieve that, **Praxiteles** compares this difference with a user-specified value `percentage_gap_to_be_considered_single_req` from the configuration file. If the difference is smaller than this bound for an observation, then this observation is included into the set of observations used to compute the request CPU utilization. Finally, **Praxiteles** takes the maximal CPU utilization values of the selected observations and deducts the aggregate mean CPU utilization for the service. The mean and the standard deviation of the resulting series are then computed, multiplied by the VM's core count and memory size, and added to the set that will later on be used to produce an empirical distribution of means and standard deviations for the requests.

As the memory readings data is not present in the Azure VMs dataset, it is derived from the CPU readings by using the correlation coefficient of 0.9 which was shared in the Microsoft's results [30]. When using this correlation to determine the memory requirements of individual requests, one is at the risk of simulating the requests that consume large chunks of memory (e.g. 100-500 MBs). To account for this situation, the generator configuration offers an option to rescale the empirical distribution of the memory requirements for the requests with a `rescaling_factor_requests_memory_requirements` parameter.

Once the empirical distributions for the resource requirements are derived, they are added to the recipe. Then, they are used to sample the parameters of the *normal distribution*, viz, mean and the standard deviation. With these sampled parameters the simulator can instantiate the requests and the services such that their simulated resource usage changes randomly according to the normal law.

9.3 Advantages and Limitations of Autoscaling Simulation

9.3.1 Advantages of the Simulation Toolbox

The purpose-built simulation toolbox introduced in the previous section offers several advantages that contribute to its usefulness when designing multilayered autoscaling policies:

- **Detailed simulated execution model**. The execution model introduced in Section 9.1 represents the simulated autoscaling process for multilayered deployments very accurately. It does so by capturing the most important abstraction of application as a network of replicated services running on top of VM clusters. The accuracy is also ensured by simulating the propagation of *each individual request* through the services network. Albeit dramatically increasing the overall simulation time, this design choice ensures that the simulations results are relevant for what is observed in production.

- **Abundance of tuning knobs borrowed from production-grade autoscaling services**. The majority of tuning knobs made available in the simulator are directly related to ones present in the production-grade autoscaling solution, e.g. deployment state reconciliation time (aka sync time), cooldown period, metrics used for scaling. In addition, the simulated autoscaling policies are composed of multiple building blocks. One is free to choose one of the ready-available implementations of these building blocks such as e.g. particular forecasting method (ARIMA, support vector regression, and so on) or write their own meeting the interface requirements of the corresponding class. Inside, each building block can also be tuned by setting its specific parameters, e.g. by specifying the architecture and the count of units in the feedforward neural network used for performance modeling in the simulated predictive autoscaling policy. The simulator draws a clear line between the tuning knobs available at the container virtualization layer (application) and at the VM clusters layer.
- **Simulations powered by the real-world data**. Since the simulator draws on the resource utilization and load traces made available by cloud services providers such as Azure (cf. Subsection 9.2.5), the simulated applications demonstrate the realistic resource utilization patterns as well as the realistic requests processing times. This advantage is especially important in the light of limited availability both of the real applications and the infrastructure of appropriate scale in the academic community.
- **Support for fine-grained simulations**. The accuracy of simulations may be increased at the cost of increased execution time by setting the milliseconds-scale simulation step. Setting small simulation step makes sense in short simulations up to tens of minutes and for loads of a moderate scale (tens of RPS at most). Fine-grained simulations might come in handy in assessing how does the autoscaling policy address a particular short load pattern, e.g. a spike or a relatively short increase followed by a fall-off.
- **What-If analysis without devastating the production deployment**. Production deployments of applications and the management software (including autoscaling solutions) are usually tuned to meet the demands of the user base most of the time. The application owner would not sacrifice the running deployment for the perspective of observing how the application would behave under some esoteric combination of parameters. For example, it might be of value to test the autoscaling behavior under rapidly surging load, but one would not do that in production due to 1) not willing to sacrifice the quality of service for the existing users, and 2) high costs of running an application replica of the same scale for the sole purpose of assessing a rare scenario. In contrast, simulations allow to conduct a what-if analysis relatively cheaply. When the parameters of scenarios are specified, one may devise multiple alternative scaling policies to evaluate them under different conditions. Conditioned design of autoscaling policies might be an extremely powerful tool for production workloads since it would allow to switch between the policies based on the changes in users' behavior.

Although the listed advantages may offset the long duration of simulations on certain scenarios of interest, one still needs to be aware of the performance-related limitations of the **Multiverse** simulator.

9.3.2 Performance Limitations of the Simulator

Support for the fine-grained simulations with accurate representation of autoscaling behavior inside the **Multiverse** simulator comes at a cost of performance. Several parameters may significantly impact the wall clock time needed to get the simulation results (in order of contribution):

- **The simulation step** is the most important control parameter that defines the tradeoff between the accuracy of the simulation and the total simulation time. Naturally, smaller values of the simulation

step tend to provide more accurate results. On each simulation step, the simulator adds the amount of time defined by this step to various duration characteristics associated with each request (e.g. waiting time in buffers, total time spent in system). If the simulation step is *larger* than the time needed for the request to change its state, e.g. to pass through the link, then the resulting response time will be *overestimated*. Smaller simulation steps, however, tend to increase the total simulation time. This performance issue is discussed in more detail later in the section.

- **The scale of load**. If the simulated load is on the scale of tens of requests per second at most, then most of the simulations run in a feasible time of a few hours at most. However, the simulation of hundreds to thousands of requests per second becomes prohibitively expensive both in terms of runtime and system resources required. The reason behind this is that each request is simulated individually, e.g. on each simulation step we advance all the requests that are currently in the system. An attempt to reduce the simulation time, e.g. with batching the requests, sacrifices accuracy of the simulation results for the acceleration. Unfortunately, such changes provably render the simulation results useless to draw any conclusions from them.
- **Autoscaling policy reconciliation/sync period** introduces two sorts of delays into the simulation. First, depending on configuration of the autoscaling policy, the computation of the autoscaling action may take some time. For example, straightforward ratio-based resource allocation computation persistently outperforms feedforward neural networks combined with optimization techniques. With smaller sync periods, such recomputations are done more frequently and thus the fraction of simulation time spent computing the new state of the system increases. Second, autoscaling actions may change the count of node groups that the application services run on. Since each node group is simulated individually, the rise in their quantity naturally leads to higher simulation times. Lower sync period may result in scaling actions performed more frequently, thus potentially introducing new node groups to simulation and increasing the total simulation time as a result. From multiple simulation runs and comparison with built-in/default sync periods, we derived that an appropriate sync period is in the interval of 15-60 seconds.
- **Timeout** defines how long does the simulator keep the request in the simulated system before dropping it. Larger timeout values result in requests being present in the system for a longer time, i.e. those requests that would ordinarily have been dropped because it took too long to process them and deliver the response are still kept around and the compute and memory resources are being spent on them. Albeit the fraction of such requests is not large in real systems, setting timeout too large will result in an increased resource consumption and simulation time.

To highlight the limitations of the fine-grained simulation approach and stress the importance of simulation parameters selection, we assessed the performance of the **Multiverse** simulator as a function of the simulation step.

During the performance evaluation, we repeated the same simulation 10 times for each simulation step evaluated (from 10 to 100 ms with the step of 10 ms). Each simulation used the same configurations generated by **Praxiteles** based on the Azure data: the application had 7 services and a tree-like structure with a single entry service. It was exposed to ~15.000 requests per hour. The performance evaluation was done on the MSI GE62 6QF laptop with Intel Core i7-6700HQ CPU (2.6 GHz, 4 cores), 16 GB RAM, under OS Microsoft Windows 10 HE (version 10.0.19041). To ensure the real-world conditions for the simulations, the performance evaluation ran in Jupyter Notebook opened in the Google Chrome v87.0.4280.88 web browser with ~90 other tabs open at the same time. Other programs like MS Power Point and Atom were also running to emulate the typical research setting. The notebook used Python v3.7.3 kernel and was started in the Anaconda distribution with ipython v7.15.0.

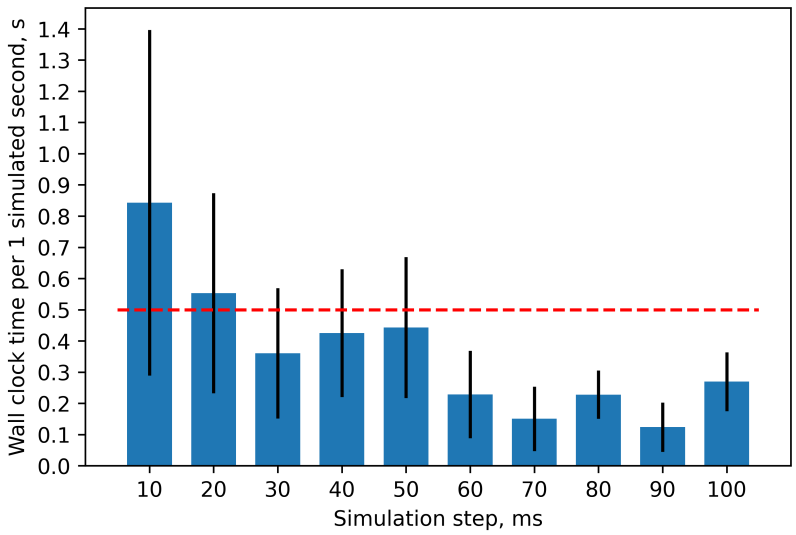


Figure 9.6.: Performance evaluation results for the **Multiverse** simulator. The simulation step took on values between 10–100 ms with 10 ms step. Each simulation was repeated 10 times. Each simulation was set to simulate 10 minutes at roughly 15k requests per hour for an application generated using Azure traces. The vertical black lines are error bars.

Performance evaluation results of the **Multiverse** simulator are shown in Figure 9.6. The red horizontal dashed line on the plot demonstrates the boundary at which one simulated minute takes 30 seconds of the wall clock time. The lower the bar is under this line, the better it is. Overall, the plot demonstrates the trend of less wall clock time needed to conduct the simulations at coarser simulation steps. The observations acquired for the lowest simulations step of 10 ms that shoot almost to 1.4 s of wall clock time for 1 simulated second signal us that such a fine resolution of the simulation step should be used with great care. In particular, it might be practical to use it to simulate rapid load surges over short intervals of time (e.g. 5-10 minutes) or in the conditions when meeting the tight response time SLO is extremely important for the autoscaling policy designed, e.g. in real-time gaming applications.

It is clear that the accuracy of the simulation results might get damaged by selecting a large simulation step. The decrease in accuracy emerges from adding more time to the joint lifetime of the request (or response) than would have otherwise passed if one would use a finer time stamp. An example of such an inaccuracy could be an addition of the 50 ms simulation step to the lifetime of the request for passing through the link whereas the link latency is set to be only 10 ms.

Based on Figure 9.6, we see that the simulation step sizes between 30 and 70 ms might offer a nice compromise between the accuracy of the simulation results and the time required to get them. On the one hand, one is likely to pay with less than half a second of the wall clock time for a single simulated second. On the other hand, the resolution is still not that coarse to address the response times SLOs of latency-critical applications fairly accurately. Increasing the simulation step beyond 70 ms does not offer much of an improvement which is explained by the diminishing reduction in the simulation steps count when increasing the step duration by 10 ms each time. For instance, when going from 10 to 20 ms the simulation steps count drops by half, whereas proceeding from 20 to 30 ms reduces it only by $\times 1.5$. We assume that the performance interference from other programs is responsible for the outliers at 80 and 100 ms since the trials for different simulation steps were not interleaved.

10

Evaluation

10.1 Simulator Validation

10.1.1 Validation Approach

Validating the simulation results is important to ensure that the realistic behavior is simulated. Validated results bring an immediate practical value to implementing an autoscaling policy. This section discusses the validity of the **Multiverse** autoscaling simulator.

The major challenge when validating a complex simulator such as **Multiverse** is in ensuring that it can simulate a general-enough behavior for the particular application domain. Since the scope of this thesis is limited to the transaction-based microservice applications, we do not discuss the validity of **Multiverse** for other broad application categories such as batch processing or streaming.

Limiting ourselves to the validation on just a single class of applications still does not allow to start off by directly comparing the simulation results against the results acquired from the runs of benchmarks and open-sourced applications. There are three strong concerns against this approach. First, there is no clear threshold that would indicate whether the simulation results correspond to the results from the real applications well-enough. Second, even a single application exhibiting a slightly different behavior from that of the simulator can refute the simulation validity. Since it is neither possible nor meaningful to validate the simulator against all the existing transaction-based microservice applications, there is no way to guarantee that the simulator will provide valid results for every such application. In addition, the vague definition of transaction-based applications allows to easily design and implement a *counterexample* for the simulation results. Third, many such applications are closed-source. Most of the available applications are either artificial benchmarks (e.g. DeathStarBench[1], Train Ticket[2], TeaStore[3]) or the toy examples (e.g. SockShop[4], Piggy Metrics[5]). Demonstrating the validity on these applications in no way guarantees that the simulation validity will also hold

[1] `https://github.com/delimitrou/DeathStarBench`
[2] `https://github.com/FudanSELab/train-ticket`
[3] `https://github.com/DescartesResearch/TeaStore`
[4] `https://github.com/microservices-demo/microservices-demo`
[5] `https://github.com/sqshq/piggymetrics`

for industry applications. Therefore, this thesis opts for discussing the validity of the **Multiverse** simulator from the following perspectives: *simulator design perspective*, *simulations configuration perspective*, and the *visual correspondence between the simulation results and the published measurements*.

10.1.2 Validation by Simulator Design

Conventionally, simulators adopt a mathematical model of the simulated process. The main validation question for such simulators is how well do these models reflect the reality. In contrast, the **Multiverse** simulator answers this question by adopting a set of abstractions and internal simulation mechanisms that are borrowed directly from the simulated subject area.

In **Multiverse**, the application is represented as a set of services that can have multiple replicas and are hosted on multiple virtual nodes (VMs). On each clock cycle, each service instance consumes a random quantity of resources that is defined by the normal distribution. The service instances may run on the same node or on the different nodes depending on the adopted placement strategy. Each request is represented as an *individual* object passed from service to service and waiting in service queues if there are not enough resources to process it upon arrival. Reconciliation of the application and platform state is invoked periodically with help of service scaling and platform adjustment policies, exactly how it is implemented in Kubernetes. Each request consumes a random amount of resources on each service where it is processed on each simulation cycle of being there (sampled from a normal distribution). Requests "processing" on VMs is implemented on the node groups shared between instances of different services. These and other abstractions discussed in Section 9.1 resulted from our hands-on experience with cloud applications deployment and resource management on public cloud services providers, in particular, AWS, and from studies of the source code and of the runtime behavior of Kubernetes.

The simulation mechanism implemented in **Multiverse** follows the footsteps of the abstractions from Section 9.1. First, the application behavior is simulated by the requests being passed from service to service and delayed there for the processing time specified in the configuration. The absence of capacity to process these requests (not enough system resources or service instances) results in them being put into a buffer. The requests are taken from that buffer according to the queuing discipline specified in the configuration (e.g. FIFO, LIFO, oldest first). Second, the scaling behavior is simulated by modifying the simulated state both for the application services and for the virtual cluster which the application runs on. The state of the application services is modified by increasing or decreasing the count of services deployed on the VM groups (node groups). The state of the platform is modified by adding, removing, or splitting the node groups according to the desired state that was derived by the adjustment policy. This mechanism corresponds to how the horizontal autoscaling works in public clouds (e.g. managed Kubernetes service and VM clusters/autoscaling groups of AWS) and in Kubernetes (combination of HPA and CA, cf. Chapter 2) at a high level. Finer details of the scaling process such as pulling VM or container images and preparing VM are considered irrelevant for the simulation and are roughly represented by the delays in the VM and services scaling that are added to the new state before it is enforced. The magnitude of these delays corresponds to that reported in the empirical studies [170, 171, 172].

The clear drawback of such simulation mechanism and set of abstractions is in that it demands more resources than the established mathematical models such as queuing networks. However, borrowing the abstractions and mechanisms from the production-grade resource managers and orchestrators allows the simulation results to offer higher accuracy in comparison to most of the mathemarical models. This is especially important when studying the methods to mitigate the 'tail at scale' effects.

10.1.3 Validation by Simulations based on Traces and Empirical Data

Simulations configurations generator **Praxiteles** was introduced not only to simplify the life of a researcher, but also to enable the generation of the simulation configurations that reflect the operation of applications and autoscaling in real settings.

First of all, **Praxiteles** leverages various cloud-related data such as VM utilization traces and FaaS applications invocations. The appeal of these trace is in that they provide a solid insight into how the production applications and VM clusters behave. The use of cluster and application traces is widely adopted in designing industry-scale solutions in an academic environment [24, 29]. Recent introduction of the Function-as-a-Service computing paradigm offered a unique opportunity for the cloud services providers to take a peek at the applications of their customers. Taking the next step, Microsoft shared the insights into the load for FaaS applications and their composition in the recent Azure Functions dataset [30]. **Praxiteles** leverages these data to generate realistic load patterns, service counts in applications, and requests processing times by each service. Hence, the use of trace-based simulation configuration generators of **Praxiteles** allows to argue about the validity of simulation results backed by the production data.

Despite all the richness of published traces, none of these datasets discloses how the applications are actually structured. Since the microservices introduced an opportunity of structuring an application in an arbitrary way as long as it implements the desired functionality, it became evident that the topology will start to play an increasing role in managing the deployments. Several studies addressed this aspect already for well-known multi-tier architecture without considering the diverse topologies offered by the microservice architectural pattern [92, 84, 149]. To fill this gap, we conducted an empirical study of a small sample of Github repositories [173] of microservice applications [159]. Among all other insights, this study revealed that the topologies of these applications are often the samples of random graphs generated by the BA-model with just four parameter sets. **Praxiteles** simulation configuration generator leverages this parameterized model to generate the realistic application topologies. Combined with the services counts derived from the Azure Functi ons dataset [30], it enforces the argument about the validity of the simulation results by **Multiverse** from the application perspective.

Thus, configuring the simulation based on production monitoring data improves the validity.

10.1.4 Validation by Visual Inspection of Scaling Behavior

Lastly, we validate the **Multiverse** simulator on a particular case of autoscaling that we studied in [11]. To make the validation concise, we focus at the multilayered deployment for AWS and an *increasing* load pattern. The corresponding load pattern and the count of virtual machines is shown in Figure 10.1. We focus at the scaling on a virtual cluster level for two main reasons. First, scaling on the virtual cluster layer follows that on the application layer. This means that similar behavior on the virtual cluster layer means similar behavior on the application layer. This allows us to keep the discussion focused. The second reason is that only the scaling on the virtual infrastructure layer impacts the application owner's bill. In contrast, scaling service instances does not affect it directly.

To conduct the validation, we ran multiple simulations in **Multiverse** slightly changing the parameters of the simulated application and of the scaling policy. This was done to ensure the stability of the simulated results. The key parameters of simulation were left unchanged. In particular, we did not change the structure of the application as well as the initial VMs type and count (one t2.small instance) and the scaling threshold of 20% vCPU utilization which were specified in the paper. To capture the specifics of AWS autoscaling, we added the default cooldown period of 5 minutes. In Figure 10.2, we provide the simulated load pattern

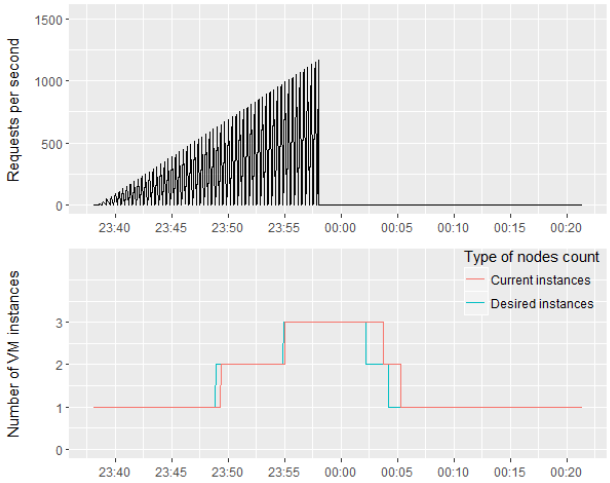


Figure 10.1.: Load and VMs scaling pattern for multilayered deployment in AWS [11].

(increase from 0 to 1200 RPS in the interval of 20 minutes) and the node count change for three selected simulations. The simulation step was 50 ms.

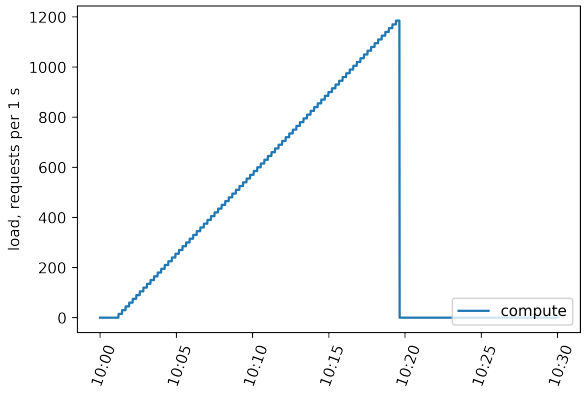
As one can notice from the node count plots in Figure 10.2, the provided patterns do not perfectly match the corresponding plot for AWS in Figure 10.1. However, what we see in these plots is that the timing of scale-ups and scale-downs is mostly right. Simulation results in Figure 10.2c even arrived at the correct count of VM instances, whereas in Figure 10.2b the simulator added one more VM for a short period of time (less than a minute), but otherwise the count of 3 VMs was reached almost synchronously to that in Figure 10.1. We assume that the parameters of simulations may be refined to capture more intricate properties both of the simulated application and infrastructure. However, there is no universal way to correctly set these parameters since they are based on a multitude of conditions and exhibit quite a large degree of randomness. Overall, we claim that the provided simulation results are satisfactory enough to claim the relevance of simulations to the real applications.

10.2 Selecting the Scale of Load for the Simulations

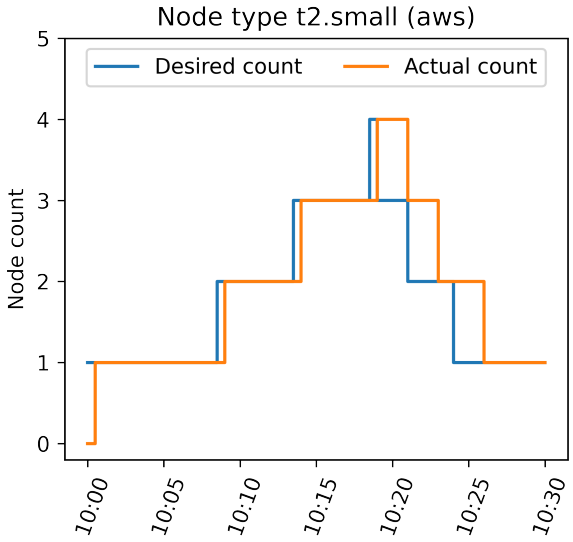
In Subsection 9.3.2, we listed the large scale of load as one of the major causes of long simulation time and proposed to decrease the scale of load to tens of requests per second at most. The main doubt that the reader may have is whether the proposed load capping still manages to offer valid simulation results. In the following paragraphs, we present two main arguments that advocate for the use of small-scale load patterns in simulations in addition to the feasible simulation time that we otherwise simply do not get.

The first argument for the use of small-scale load is that the majority of the real-world applications do not really suffer from high load. To back up this argument with the data, we continue examining the Azure functions dataset [30] to figure out what the majority of the cloud applications actually look like in terms of load. For the sake of clarity, we dropped all the zero invocations rate cases from the dataset (roughly 13 mil. data points). We averaged the data only across the provided days (14 days, each day corresponds to a single file), leaving them on a per-minute basis to minimize the impact of averaging on the end results.

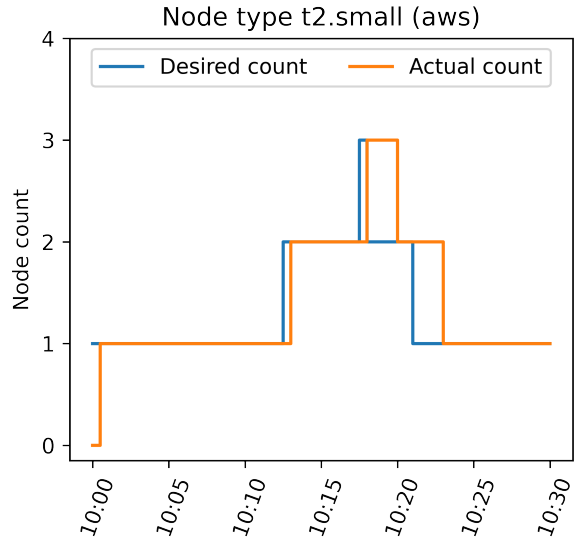
The analysis of the Azure functions dataset discloses that the vast majority of the observed per-minute requests rates (99%) are below 522 invocations per minute (ipm), or roughly under 9 invocations per second. The increase in CDF is so steep that we had to supply a zoomed-in version instead of the barely informative



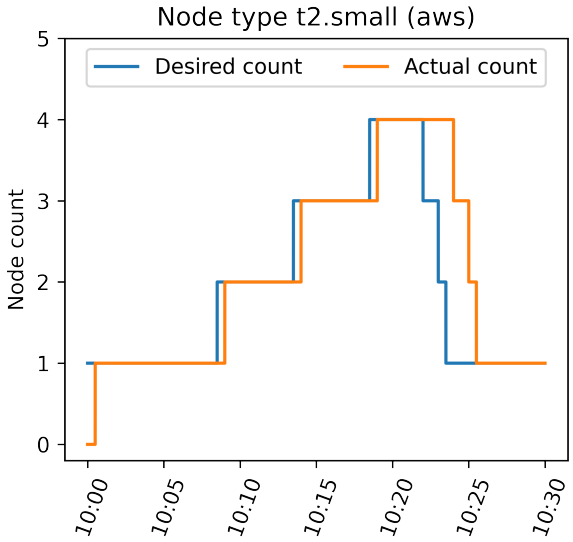
(a) Increasing load pattern generated by the **Multiverse** simulator.



(b) First example of the nodes allocation during the simulation.



(c) Second example of the nodes allocation during the simulation.



(d) Third example of the nodes allocation during the simulation.

Figure 10.2.: Simulated node count changes for the increasing load pattern.

original plot. Although Figure 10.3 zooms into an interval that is $\times 79$ smaller than the maximal invocation per minute value from the dataset, the 99th percentile is still achieved roughly in the middle of the zoomed-in plot. On top of that, 95th percentile is achieved already at less than a tenth of the zoomed-in interval (86 ipm, or roughly 1.5 invocations per second). Such a steep increase in CDF of invocations rate observations clearly indicates that mostly the applications are not faced with high load. The applications on the scale of Google search engine or Amazon web shop (40-100 kRPS according to the sources on the web) are rare exceptions rather than a rule.

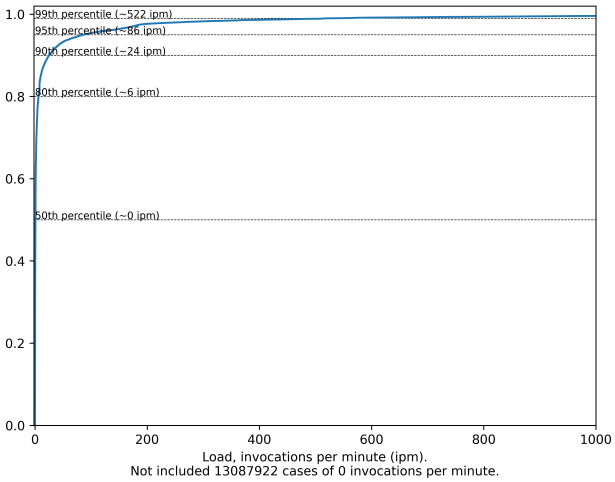


Figure 10.3.: Zoomed-in CDF of function invocations rate observations for the Azure functions dataset.

We conclude the analysis by stating that *simulating the load patterns that are capped by 10 requests per second is enough to cover 99% of the request rates observed in real cloud applications* as per Azure cloud data. Another concern that still remains is whether such small request rate can really trigger autoscaling. Our second argument addresses it.

In practice, the request rate is far from being the only parameter determining the need for scaling the application. Since the purpose of applications varies as well as the individual characteristics of requests (e.g. `SELECT * FROM data` vs `SELECT * FROM data WHERE name IN ("bob", "sam")` and the processing times that they result in, one might expect that the resources consumed during the processing of requests may vary both from application to application as well as inside a single application. Thus, two applications facing the same request rate may in fact exhibit quite different resource utilization patterns and thus will have to be scaled differently. Since there are no datasets and studies estimating the impact of individual requests on the system resources consumption, we had to derive these data from the Azure dataset based on observing the variance in the resource utilization patterns. It turned out that for most of the applications just a handful of requests was enough to saturate the resources (on the scale of 100 of requests being processed simultaneously). Thus, generating just a handful of requests per second should be enough to trigger autoscaling given the resource utilization patterns available for the derivation from the real-world datasets. In principle, one could get similar behavior by increasing the number of requests and reducing the resource consumption of an individual request, however, an increase in the requests count will result in the simulation time increase, hence we prefer to have requests with larger resource utilization that allows to reduce the rate of requests generation and still get autoscaling triggered.

10.3 Selecting the Right Metric for Autoscaling

10.3.1 Metrics Categorization

One of the most important choices in the autoscaling policy design is the selection of an appropriate scaling metric or a group thereof. Although there is an abundance of quantitative characteristics of applications and infrastructure, we make an attempt to assign them to several broader categories:

Resource utilization. Metrics in this category appeared on the autoscaling stage first. The reason is that the cloud services providers were first and foremost interested in maximizing the utilization of their data centers. Hence, the natural target for scaling would be to maximize the resource usage. First autoscalers aimed at scaling up only when the resource utilization reached a certain high threshold, usually somewhere between 70-90%. However, resource utilization was proven to be insufficient for autoscaling since it is not fully correlated to the service level delivered by the application, especially before the resource saturation point is reached. The second reason is that the metrics in this category are usually highly volatile, especially CPU utilization. Lastly, one needs to be aware of what type of system resource does the application rely the most, e.g. whether it is CPU-bound or memory-bound.

Service level as experienced by the end user. When aiming to satisfy the application users, the intuition tells us to equalize the end user experience with the autoscaling metric. In case of online shops, one can consider response time as such a metric. Response time of higher percentiles reflects quite accurately the quality of service that the user experiences. On the other hand, the satisfaction of the end users *en masse* may be characterized by the throughput of the deployed application. For a long time, CSPs did not have access to the metrics that characterize the service level that the end user experiences. FaaS paradigm allowed them to finally gain the application-level observability. In turn, this paves the path to extending the existing FaaS autoscaling mechanisms with the other ones that are based on the end user's quality of experience.

Load. Similarly to the above category, load in a sense of incoming requests was not observable by CSPs prior to FaaS. Nevertheless, they could track network bandwidth utilization even under the most restrictive (in terms of observability) IaaS paradigm. Load serves as a proxy for the end user experience. For instance, rapid increase in the application load combined with the untimely scale up will likely result in rising response time. Theoretically, using load as a metric for autoscaling makes a lot of sense from two points of view. First, load is the only metric that cannot be managed neither by the cloud services provider nor by the application owner. Second, load is the precursor to all the other metrics, e.g. new users' requests result in an increase in memory utilization and in rising response times for other requests getting less processor time on the shared infrastructure. This means that observing any other metric will essentially lead to *lagging* autoscaling decisions by definition.

Internal application behavior. Using the application internal behavior[6] as a foundation for autoscaling decisions is complicated by the diversity of building blocks/services that the application consists of, e.g. Apache Kafka, RabbitMQ, ZeroMQ, Amazon MQ and others available for the message queuing alone. Additional challenge is introduced by the custom business logic implemented in the application services. Since generalizing across applications with different functionality makes little sense, instead, we focus at the programs that implement routine functionalities such as message passing, storing the data, configuration management, authentication and so on when discussing the autoscaling decision metrics that are based on the internal application behavior. Let's consider a messaging service since the microservice applications often rely on buffers to store the queries that cannot be processed immediately. Albeit most open source and commercial solutions differ in their implementation and tuning knobs, there is a common behavioral characteristic that impacts the observable quality of service. This characteristic is *waiting time*, i.e. the time that the message/request spent in the service buffer waiting to be served. By studying this metric, one

[6] Internal behavior of an application is a behavior that is not observable by the end users.

can identify the topological/performance bottlenecks that might be resolved by changing the application architecture. From the autoscaling point of view, waiting time in these buffers is interesting as a metric that falls in-between load and response time in its relation to the end user experience.

The offered categorization of metrics aims at ordering them along two axes: the *timeliness of availability* and the *relevance to the service level as experienced by the end user*. We refer to these as timeliness and QoS-relevance, or simply relevance. An ordering of the discussed metrics categories along the axes of the timeliness/relevance space is schematically depicted in Figure 10.4.

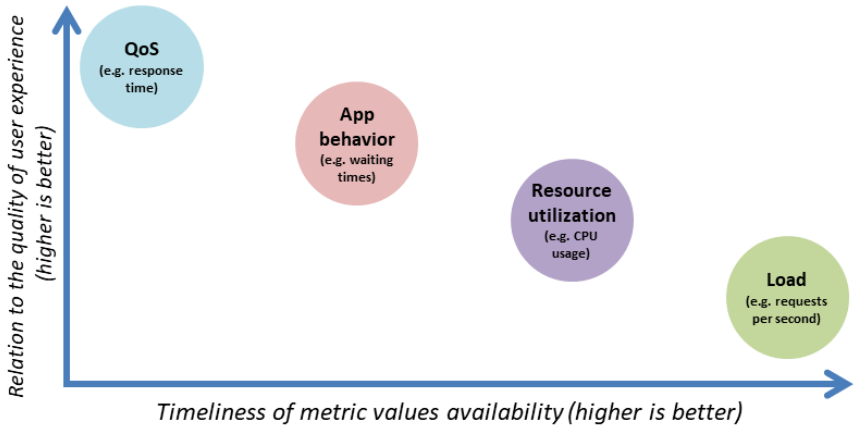


Figure 10.4.: Categories of autoscaling metrics aligned relatively to two characteristics: timeliness of availability and relevance to the quality of user experience.

Load category has the lowest relevance to the user experience. The reason is that there is also an application and infrastructure between the actual load and the quality of service. Despite that, the user load is available almost instantly upon its arrival at the entry service. If the requirements to the timeliness of the autoscaling action are high, then the load metric might be useful, although not much accurate if the user experience is considered a priority when scaling.

Resource utilization metrics come as the second preference when the timeliness of scaling actions is important. Metrics describing the application behavior were put below the resource utilization in terms of timeliness since their variability usually follows the saturation of system resources. For example, when there are not enough hardware threads to process incoming requests, a queue starts to build up thus boosting the waiting times of requests.

Although most quantitative characteristics describing the application behavior can be acquired on the time scale that is close to that of resource utilization, still, the resource contention is the major precursor to the distinctive changes in application metrics. Relevance of metrics in this category to the user experience is naturally lower than that offered by response time, etc. Unfortunately, quality of service metrics become available only post factum, i.e. either when the response was delivered or when it hit the timeout.

10.3.2 Studying the impact of metrics on autoscaling

To evaluate the impact of metrics from different categories on the autoscaling actions, we devised four load patterns. Albeit simple, these load patterns cover the most important cases encountered in practice. Representations of these patterns in time domain are shown in Figure 10.5.

The **Step-up** pattern covers 5 minutes. The load rises from 1 to 3 RPS after the first minute of simulation is over (cf. Figure 10.5a). In total, **780** requests are generated over these 5 minutes. This load pattern aims to capture how different metrics handle rapid load surge (in our case, $\times 3$ increase).

Similarly, the **Step-down** pattern covers 5 minutes. The load drops from 3 to 1 RPS after the first minute is over (cf. Figure 10.5b). In total, **420** requests are generated over 5 minutes. This load pattern aims to uncover how well do different metrics respond to the rapid load decrease ($\times 3$ decrease), i.e. how good are they for downscaling the deployed application and thus saving the budget.

Next two patterns represent the ever changing load (oscillations). Both are simulated over the period of 10 minutes. In each case, **1200** requests are generated in total. The **Slow oscillations** case (cf. Figure 10.5c) splits the whole simulation interval into two sub-intervals of 5 minutes each. Each sub-interval repeats the same step-up pattern. Between them, there is a drop to the original level of 1 request per second. The duration of these dips and spikes is chosen in such a way that it is roughly on the scale of booting and termination times of VMs simulated. In contrast, the **Rapid oscillations** pattern (cf. Figure 10.5d) has higher frequency of spikes and accommodates 5 step-ups in total. Each of them lasts only 2 minutes splitting this time equally between the dip and the spike. This load pattern helps to understand how well do different metrics categories adjust to the highly volatile load.

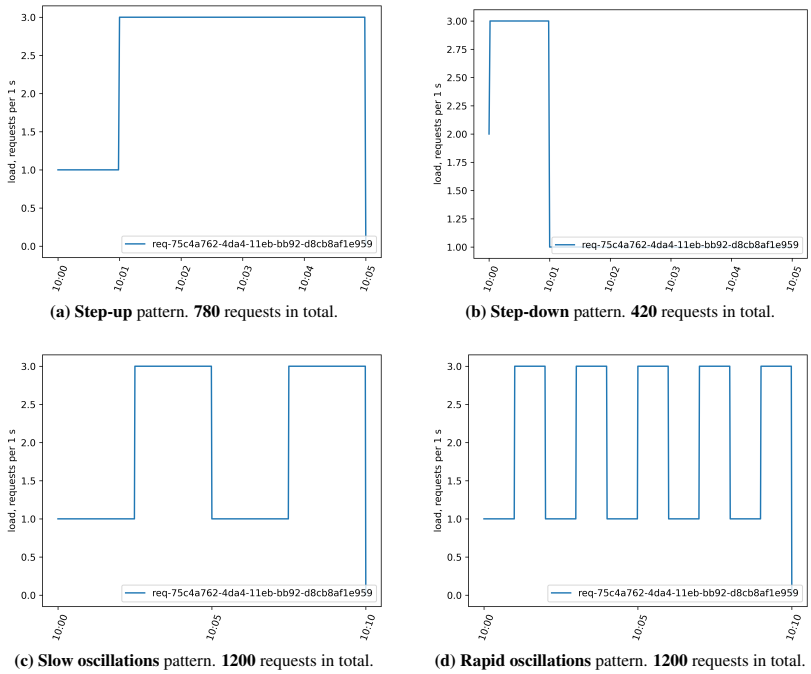
The scaling policy for this experiment is a simple rule-based policy, in which the measured metric is divided by the threshold to get the count of service instances. The result is adjusted by multiplying it by 1.15 (15% overprovisioning as in Kubernetes' VPA). The autoscaling period is set to 30 s. Each experiment was performed 10 times to ensure the reliability of the results. The following four metrics were evaluated:

- **Waiting time** (application behavior category). This metric represents how much time do the requests spend in buffers waiting to be processed. The threshold for each service is **50 ms**.
- **Response time** (user-level quality of service category). This metric represents how long does the user have to wait till the response. The threshold for each service is **200 ms**.
- **vCPU utilization** (resource utilization category). This metric represents how much node groups' vCPUs are utilized. The threshold for each service is **80%**.
- **Memory utilization** (resource utilization category). This metric represents how much of the node group's memory is consumed. The threshold for each service is **80%**.

The load metric was not evaluated since it does not encode the state neither of the application nor of the infrastructure. Hence, this metric alone is useless to argue about the quality of user experience. It requires additional *mapping* into the metric that represents quality of user experience, e.g. response time.

To a large extent, the user-centric quality of service can be quantified with just two characteristics: the *amount of timed out/unfulfilled requests* and the *response time*. The timed out requests bear higher significance than those that just took much time to be processed, but got the response. In all the conducted experiments in this section, the time out was set to 10 seconds. Figure 10.6 presents the distribution of requests between the *fulfilled* and *failed* categories.

As one can see from Figure 10.6a (**Step-up** load pattern), response time as an autoscaling metric in the reactive scenario demonstrated the highest percentage of fulfilled requests. Response time as a metric was able to add almost 10% of fulfilled requests when compared to every other metric considered. This supports the hypothesis that the response time is the best proxy of the service level. Aiming to compensate for higher response time with more service instances and nodes, an automatic reduction in percentage of failed requests will occur. Surprisingly, utilization-based metrics are on par with the waiting time. We assume that this happens since the threshold for scaling with the waiting time was set to 50 ms, which is $\times 4$ lower than



(a) Step-up pattern. **780** requests in total.

(b) Step-down pattern. **420** requests in total.

(c) Slow oscillations pattern. **1200** requests in total.

(d) Rapid oscillations pattern. **1200** requests in total.

Figure 10.5.: Load patterns used to analyze the impact of metrics category on autoscaling.

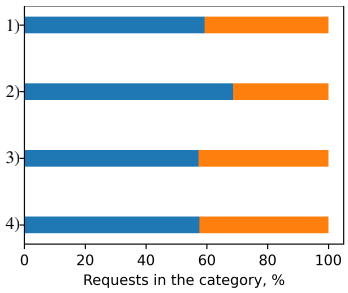
the one set for the response time. As the application consists of 7 services (with the longest path being 4 services) and each service has 2 buffers (upstream and downstream), it is very unlikely for the request to spend that much time in each buffer. The right selection of thresholds demands one to clearly understand how the application behaves internally.

The **Step-down** load pattern is served by all the considered metrics almost equally (cf. Figure 10.6b). The reason for this might be because the 3 RPS load level holds only for the first minute, dropping to 1 RPS thereafter. Since the scaling decision is taken each 30 s and the booting times for VMs are on the scale of minutes, it becomes impossible for the reactive policy to accommodate the initial level of 3 RPS. This example points at a significant issue with the reactive policies – in absence of instantaneous VM provisioning, reactive policies cannot accommodate well the changes that demand less time than the provisioning takes. Nevertheless, this case is not that grim as it appears to be. In the end, the scale-down's purpose is not to meet the service level objectives but rather *to save the budget of an application owner*. Indeed, here we observe the diversity. Both resource utilization metric-based policies demonstrated lower total cost of the experiment, **0.01319** and **0.01289** USD[7], for memory utilization- and vCPU utilization-based scaling correspondingly. The costs for the response time- and waiting time-based scaling were slightly higher, i.e. **0.01331** and **0.01386** correspondingly, which supports the statement about the timeliness of scaling actions derived based on these metrics (cf. Figure 10.4).

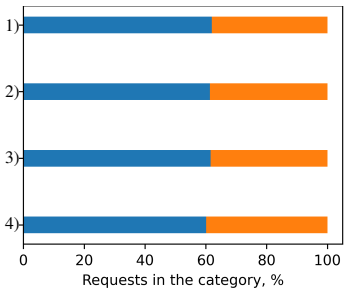
Similarly to the **Step-up** pattern, response time-based reactive autoscaling shows the best results in terms of fulfilled requests for the **Slow oscillations** load pattern (cf. Figure 10.6c). This time, however, the percentage of fulfilled requests reaches almost 80% (79.1%). This 10% improvement over the simple step-up pattern could be attributed to the dips in load and to the longer duration of simulations (10 minutes instead of 5) that recoups the scale-up. The utilization-based reactive scaling lags 12-15% behind in terms of fulfilled requests (64.4% and 67.8% for memory utilization- and vCPU utilization-based scaling correspondingly). Again, this might be because of the utilization-based metrics being poor proxies for the end user experience. Waiting time as a decision metric falls somewhere in-between with 65.5% of fulfilled requests.

[7] We use the price per hour for on-demand instances of AWS in these experiments.

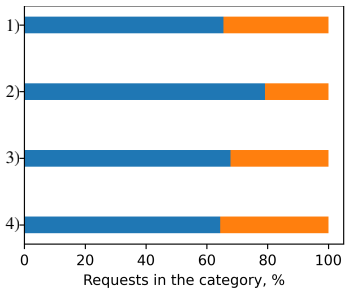
Response time manages to keep its first place as a metric of choice for fulfilling the maximal requests count also under the **Rapid oscillations** load pattern (cf. Figure 10.6d). This time, however, the resource utilization-based metrics come quite close to it demonstrating 69.4% (vCPU) and 71.6% (memory) of fulfilled requests vs 76.2% for response time. Waiting time still lags behind with 58.2% of fulfilled requests. Interestingly, all the studied metrics demonstrate the results that are at least *not worse* than that demonstrated for the **Slow oscillations** load pattern, albeit being faced with a higher frequency of changes. Considering almost identical total costs observed under both load patterns, the most plausible explanation might be that the short dips are not large enough to trigger the scale-down and thus the reactive policy ends up keeping VMs around during these dips which allows it to better accommodate the follow-up spikes. This conclusion motivates the use of *stabilizers* when computing the desired count of service instances and virtual machines which were presented in Subsection 9.1.7 as part of the scaling policy.



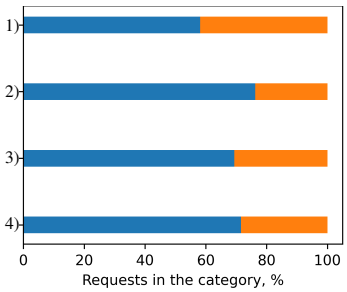
(a) Percentage of fulfilled/failed requests for the **Step-up** load pattern.



(b) Percentage of fulfilled/failed requests for the **Step-down** load pattern.



(c) Percentage of fulfilled/failed requests for the **Slow oscillations** load pattern.

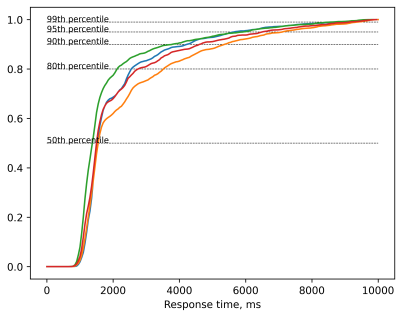


(d) Percentage of fulfilled/failed requests for the **Rapid oscillations** load pattern.

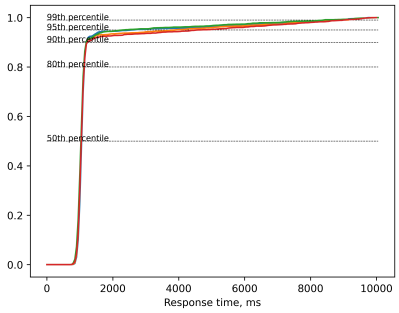
Figure 10.6.: Fulfilled vs failed requests distribution in the metrics evaluation experiment under various load patterns depicted in Figure 10.5. Bars in order from top to bottom: 1) Application behavior category (waiting time); 2) Quality of service category (response time); 3) Resource utilization category (vCPU); 4) Resource utilization category (memory). Joint legend: fulfilled requests are in blue (left part of each bar), failed requests are in orange (right part of each bar).

Appeal of the response time as a metric of choice for the reactive autoscaling policies increases after discovering that it results in the best response time CDF for all the considered load patterns (cf. the leftmost (green) curve on all plots of Figure 10.7). Again, as in the case of fulfilled requests quality metric, this is not a surprise since the response time metric directly represents the quality of service delivered to the end user. Most of these plots (cf. Figure 10.7a, Figure 10.7c, and Figure 10.7d) show relatively small difference between the use of waiting time and the memory-based scaling. In contrast, on the same plots, response times for the vCPU utilization-based reactive autoscaling saturates relatively slowly, after the 60-70th percentile is passed. This behavior can be attributed to the memory-intensive nature of the application generated for the simulations based on the Azure Functions data. It is important to keep this observation in mind, since

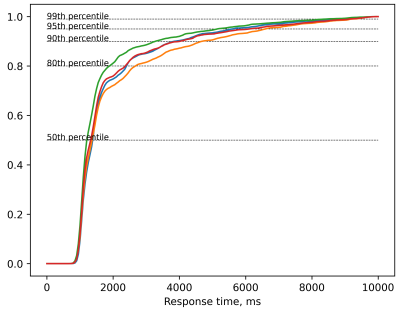
we will later use it in exploring the building blocks of predictive autoscaling policies. As we see, vCPU utilization exhibits the smallest relevance to the scaling decisions.



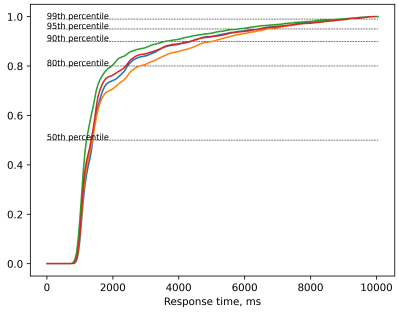
(a) Response times CDF for the **Step-up** load pattern.



(b) Response times CDF for the **Step-down** load pattern.



(c) Response times CDF for the **Slow oscillations** load pattern.



(d) Response times CDF for the **Rapid oscillations** load pattern.

Figure 10.7.: Response time CDFs in the metrics evaluation experiment under various load patterns depicted in Figure 10.5. Joint legend: Quality of Service category (response time), Internal application behavior category (waiting time), Utilization category (memory), Utilization category (vCPU).

10.3.3 Recommendations on Selecting the Decision Metric for Autoscaling

Results discussed in this section demonstrated the superiority of the service level/QoS metric category when the goal of autoscaling is to maximize the user experience. The use of response time as an autoscaling decision metric offers better results than the other metrics with lower relevance to the user experience, e.g. the resource utilization and the internal application behavior metrics. Given that it requires some time for the response time metric to be accumulated, it might make sense to combine this metric with the other ones. Upon the deployment, one can let the autoscaler take the decisions based on the utilization metrics until enough response time observations are accumulated. However, if the load pattern significantly departs from its steady state, it might be better to opt for the timely utilization metrics again till the steady state of load is recouped.

Using only the response time as a scaling metric is not enough to gain a significant improvement over the other metrics. We saw that it demonstrates only 10-15% improvement in terms of fulfilled requests in comparison to the utilization-based metrics (cf. Figure 10.6). There are two big challenges that hinder it from achieving better numbers.

The first challenge is that reactive autoscaling fails at capturing the short-lived load surges. If the duration of such a surge is lower than the combination of the autoscaler state sync period with the booting times of

VMs, then it has no chance of timely providing necessary resources. Increasing the timeout interval may help in this case but at the expense of users waiting longer for the responses and the system resources being taken from the requests that might have otherwise been processed quickly.

The second challenge is that neither the response time nor the other metrics (also load) map directly onto the system resources. For instance, by setting the threshold in the rule-based metrics, we assume that each additional instance (e.g. service or VM) bears exactly this quantity of improvement in the original metric of choice. For example, if the threshold for the response time is set to 200 ms, then it effectively imposes an assumption that each additional service instance is able to decrease the response time by 200 ms. This is obviously not true given the nonlinearity of application and infrastructure behavior topped by the unequality of various requests and services in terms of the system resources used.

The first mentioned challenge can be addressed by *forecasting* the metric values, i.e. extrapolating current observations into future and using these to provide the virtual resources ahead of time. Such an approach will likely result in over-provisioning and won't help to address the second challenge. To address the second challenge, one has to introduce some form of mapping of the observed metrics onto the quality metric such as response time. This can be done differently, e.g. by writing a formula or by training an appropriate neural network to capture this relation. In the next section we investigate how solving mentioned challenges helps in building better autoscaling policies.

10.4 Improving Timeliness and Relevance of Scaling Actions to Enable Predictive Autoscaling

10.4.1 Improving Relevance of Scaling Actions with Machine Learning

The majority of metrics collected from the applications and the infrastructure is not directly related to the quality of user experience (cf. Figure 10.4). Take resource utilization for instance. First, this metric comes in many types: CPU utilization, network utilization, memory utilization, etc. Some resource utilization metrics are less valuable when the associated type of resource is not used that much. For example, computing Fibonacci numbers might be CPU-intensive, but it is far less likely to make a case for memory utilization based scaling. Second, resource utilization is not indicative of the user experience. One can assume that the resource utilization close to 100% results in queuing new requests instead of serving them right away. However, this could also be a pointer to an accurate capacity planning complemented by the low-variability load.

As was explained in the previous section, load is *the worst metric category* in terms of QoS-relevance (cf. Figure 10.4, where the load category bubble is positioned at the very bottom). The core issue is that this metric does not reflect the state of deployment at all. Without a doubt, load impacts the service level delivered since the processing happens on the shared virtual and physical infrastructure. For load, the challenge is to bridge the wide **relevance gap** existing between this metric and the quality of service as experienced by the user. Formally, this gap can be bridged by a function f that maps the load metric onto the quality of service, e.g. the response time – $RT_{99\%} = f(L)$, with the 99%-tile response time represented as a function of load (request rate per second). One can attempt to make this crude model more accurate by introducing other parameters such as memory utilization, U_{mem}, and requests waiting time, WT, viz: $RT_{99\%} = f(L, U_{mem}, WT)$. In essence, this addition does not change the main idea of transforming a group of metrics into another metric that better represents the quality of service.

Unfortunately, pure analytical methods fail to offer such a function f for a number of reasons. First, the number of applications and load patterns is infinite. Even a minor change in the application topology and/or logic will likely result in a completely different model because of complex nonlinear interrelations between the structural characteristics of application, its resource consumption and the delivered service level. Second, the deployments in an uncontrolled environment exhibit highly volatile stochastic behavior. This precludes one from characterizing their behaviors with deterministic equations. Such sporadic events as the garbage collection may severely impact the delivered service level [174]. Third, the nonlinear nature of deployments' behavior and the abundance of parameters that characterize them renders the derivation of an all-in-one analytical model an extremely tedious task with low applicability to other conditions, viz, different load pattern, different deployment platform, and so on.

Luckily, machine learning (ML) offers a set of methods that allow to represent the mapping between the load and the quality of service as a *black box*. From an ML perspective, discovering the structure of function f is equivalent to performing a *regression*. In regression, one quantitative parameter (aka dependent parameter) is represented as a *combination* of independent parameters with different weights. Machine learning offers two broad categories of methods to approach this task.

Linear regression methods. Methods in this category are used to model the linear relations between the parameters. These methods are the go-to choice when performing an initial data analysis. These methods allow to capture the general dependencies and trends, but fail when the system under study exhibits nonlinear behavior such as e.g. performance saturation effects (cf. Roofline model [175]). Although there is an abundance of linear regression techniques, most of them tackle offline problems, i.e. an incremental adjustment of such models is not possible by design. This is one of two major limitations of the linear models (along with their inherent inability to accurately represent nonlinear processes), since one has to fit the model anew on the whole data set with the new samples added. Over time, the performance penalty will likely outgrow the accuracy benefit for adding a new sample. Although there are ways to mitigate this effect for the online models like dropping the old observations or raising the granularity by resampling, there are also more convenient online regression techniques such as the stochastic gradient descent (SGD)-based linear regression, and the passive-aggressive (PA) linear regression.

Nonlinear regression methods. This niche is dominated by the universal nonlinear approximators also known as neural networks [169]. In essence, neural networks equip one with a Swiss Army knife to devise a nonlinear model of an extreme complexity without carefully defining the relations between the parameters and distilling the meaningful features. The appeal of neural networks is in that both the features and the relations between them and the predicted variable(s) are automatically derived. The most important prerequisite is to provide enough diverse examples during the model training process, i.e. the process of gradual adjustment of network weights such that the prediction is as close as possible to the observed value. Essentially, one needs to define the input parameters space (e.g. count of service instances, load, and resource utilization), the output (e.g. response time), and the *structure* of the neural network. Conventionally, the structure of a neural network is defined in terms of *layers* each of which contains some amount of the so-called *neurons*. Recent papers rightfully depart from analogies with human brain and refer to these as the *units* instead. The count of layers determines the *depth* of the neural network. Higher number of layers and units in them means higher representative power of the model. In other words, the equation implemented by the network should encompass enough parameters to model the general relations between inputs and outputs of the modeled function f as well as the edge cases. An example of such a behavior is the linear relation between the response time and the load followed by the constant response time and increased requests drop rate when the load hits certain threshold [97]. With neural networks, one needs to be alert that 'the more the better' formula only holds true to a certain extent. If one continues to increase the representation capacity of the neural network, it may reach the point at which it will have enough parameters to memorize all the samples provided to it during the training. Then, when faced with an unobserved example, it will fail to produce

the accurate results. This problem is known as *overfitting*. Independent of the type (convolutional, recurrent, etc), neural networks can be trained either in offline or in online mode. For the latter, one accumulates the dynamically collected data in mini-batches that are later fed to the neural network to adjust its weights incrementally. This characteristic along with the capability to represent complex nonlinear behaviors makes neural networks way more appealing than the linear regression in autoscaling context. These advantages come at the expense of increased resource demands and nearly-absent explainability of the model.

Although the ML-based representation of function f does not allow to argue about the origins of the demonstrated service level accurately, later experiments will show that it fulfills the promise of more or less accurate prediction of QoS metric. The decision on appropriateness of the accuracy heavily relies on the application area under consideration. Say, models for the public infrastructure applications such as public transport ticketing system may require model to either be extremely accurate in its mapping or to err on the side of underestimating the quality of service to be delivered.

Mapping the low-relevance metrics such as load onto the quality of service alone is insufficient to determine the amount of resources to be provisioned s.t. the service level objectives are met. Upon having a prediction of the user-side quality metric, there emerges a need to compare it against an *expected* level of service, say, response time should be not higher than 200 ms, and then determine, how the service instances and the virtual infrastructure should be adapted. To avoid the need to train yet another model representing mapping function g, we adopt the approach presented in Chapter 7. Below, we recap the steps of the original resource provisioning process for the horizontal autoscaling as follows:

- **Online training of the deep model (neural network) representing the mapping function** f. This step takes a mix of managed deployment parameter (e.g. service instances count) and unmanaged metrics (e.g. load and resource utilization of some kind) as an input to train the model. The model output is the user-side quality of service, e.g. 99th percentile response time. Both input and output values are accumulated till the amount reaches the predefined mini-batch size. Then, the accumulated values are used to adjust the model weights. Due to the stochastic online nature of the process, the diversity of load patterns and likely absence of the pre-collected data sets, all the data is used for training and no validation is performed. This allows one to get an actionable model faster. Inaccuracy of the model can be compensated by over-provisioning the processing entities (service instances and VMs) with some coefficient as is done in Kubernetes. The deep model may also be pre-trained on a representative selection of load patterns and then further refined in production. It is important that the training setting resembles the production configuration as close as possible. In the evaluation, we use the pre-trained model that is refined during the simulations.
- **Optimizing for the managed deployment parameter using the trained model**. An autoscaler is limited in what it can manage. For example, Horizontal Pod Autoscaler (HPA) of Kubernetes manages only the count of replicated pods. In contrast, Kubernetes' Cluster Autoscaler (CA) manages count of underlying virtual machines based on the requirements determined by e.g. HPA. The quantified representation of this managed entity (say, pod or VM) is fed as one of the deep model inputs. Since this thesis broadens its scope by considering the autoscaling of multilayered deployments (achieved by combining HPA and CA in Kubernetes), we selected the highest level abstract entity, i.e. service instance[8], to train the model. Other inputs characterize the state of the *environment* (load) and the *deployment* (resource utilization) at a particular moment in time. Having the function f that maps this input to the response time in the form of a deep model and the threshold on the appropriate service level, we are able to use the optimization techniques to find the count of service instances minimizing the divergence between the predicted user-level quality metric and the specified threshold. Other

[8] Roughly corresponds to a pod replica in Kubernetes' terminology.

input parameters are supplied either by the current observations or are extrapolated to get the required *future value*. The selection of a particular optimization technique is not important. For instance, the **Multiverse** simulator employs slightly tuned Python **scipy**'s constrained optimization problems solver with `trust-constr` method. The application of this optimization method to the resource management problems was discussed in [174].

The above process only helps to improve the *QoS-relevance* of scaling actions by tracking the impact of deployment parameters such as VM and service instances counts on the user-side performance. In other words, it allows to determine the value of virtual entity (in case of horizontal scaling) in terms of quality of service improvement. For example, the model may provide an estimate that 5 service instances of a particular service under 3 RPS load are needed to ensure 150 ms of average response time (or any other aggregation of this metric). Having a trained model in place, we can set the response time threshold and let the optimization method search for an appropriate count of service and/or VM instances to meet this goal given other inputs such as load. Thus, we can monitor metrics with low relevance to the quality of service but still get an actionable value for the managed parameter to meet the desired SLO. For that, the deep model has to be trained on a diverse data set that covers as many cases as possible.

Employing ML helps to improve the relevance of scaling actions but fails to improve their *timeliness*. There are at least two ways to make up for the lack of improvement on the side of timeliness. The first way is to expand the deep model s.t. it would be able to account for the order of incoming observations in time. This approach would effectively result in the deep model solving two tasks at once, viz, 1) extrapolating the current values of metrics into the future, and 2) predicting response time based on these future values. Albeit possible, this yields significant complications in the structure of the deep model and its training process by introducing the recurrent neural network (RNN) part to it. In turn, this results in increased demands for the training phase both in terms of resources and samples since the model has to wait longer to account for the time-dependent behavior which may span large intervals of times, e.g. days or months.

The following subsection elaborates on an alternative way of addressing the problem of improving the timeliness of scaling actions. In particular, we offer to improve timeliness by introducing a separate *forecasting model* for metrics used as inputs to the deep model. Forecasting methods were discussed at length in Chapter 6. Below, we connect the forecasting methods to the timeliness property of scaling actions that is critical to enable the predictive autoscaling.

10.4.2 Improving Timeliness of Scaling Actions with Forecasting

Timeliness of metric collection determines when the autoscaling decision can be taken. As an autoscaling metric, load offers the best timeliness since it can be collected directly upon arrival at the application's entry point, aka frontend service (cf. Section 10.3). Although the use of load as an autoscaling decision metric can indeed improve timeliness of reactive autoscaling, there still is a limitation brought by the very nature of the reactive approach. If one wants load to be useful as the scaling metric, the minimal booting time for the scaled entity (e.g. virtual machine or container/pod, or both) should be way less than the average desired response time that the uncongested system offers, $min(T_{booting}) << RT_{SLO}$. Naturally, for small response times on the scale of hundreds of milliseconds, this condition can only be satisfied for the low-overhead virtual entities such as containers not bloated with libraries and startup scripts. However, if the virtual cluster hits its capacity under the arriving load, the provisioning of new VMs will likely take orders of magnitude more than the desired response time, e.g. 1-2 minutes. Thus, it becomes evident that the reactive paradigm is limited even for the most timely of all metrics – load.

There are certain types of dynamic load patterns (e.g. oscillating load shown in Figures 10.5d and 10.5c) that can hardly be satisfactory addressed by reactive autoscaling. These patterns are characterized either by

a) the *frequent spikes/dips* in load with the duration lower than the minimal virtual resources provisioning duration or b) the *continuous load growth* at a rate that outpaces fastest resources provisioning. Assuming that the reactive policy is well-tuned in terms of mapping the load changes onto the virtual resources, there still remains the challenge of *timely provisioning*. Monitoring load does not help to address the described situations a) and b). At the moment of spotting the load increase, it is already too late to provision additional resources since SLOs for the requests contributing to this increase will anyway be violated. Response times will surge since the booting times for many VMs are on order of multiple seconds or minutes. Seconds-scale sync times for scaling mechanisms (like in Kubernetes) add on top of that. Taking these provisioning limitations into account, the only solution to the challenge of timely resource provisioning becomes to respond not to the present, but rather to the anticipated future conditions. This can be achieved by extrapolating the scaling metrics with help of time series forecasting.

Forecasting can improve timeliness of load used as an autoscaling metric. Other metrics represented as time series (e.g. utilization) can also be forecasted. Nevertheless, this thesis focuses on load as a metric to forecast since it is not reverse-impacted by the autoscaling process and demonstrates strong seasonal patterns and trends that make it more predictable. Other metrics categories, such as utilization or response time, are largely nonlinear and are impacted by the autoscaling decisions. This significantly increases the amount of cases that the forecasting model should account for. For example, the user load exhibits weekly periodicity with a trend. It does not depend on whether the provided resources were sufficient or not. Only in the rare case of a significant load spike such as that on Black Friday experienced by the majority of online retailers, can the load exhibit its dependence on the amount of provisioned resources. The inability of a webshop to handle such a spike may result in a 'retry storm', a pattern described by Netflix[9]. For load, however, such dependence on the state of infrastructure and application is rather an exception than the rule, whereas for all the other metrics categories presented in Section 10.3, it is their intrinsic property.

Although mapping the request rate onto the quality of service metric with the follow-up optimization for the number of service instances is feasible both for the reactive and proactive (i.e. with forecasting) cases, load forecasting alone is not sufficient for predictive autoscaling. The reason for that is the same one as for not using load as a sole metric bundled with the rule-based resource provisioning. Under most of circumstances, the load alone does not encode any information on whether the deployment is saturated with the requests or not. Hence, having a fixed threshold does not help since the load can grow unbounded. If, say, there is a continuous increasing trend in the load and the load grows from 1 RPS to thousands of RPS, the fixed threshold of 1 RPS will likely result in excessive provisioning of virtual machines since the threshold was defined incorrectly and the metric itself does not reflect the current state of the application. Therefore, in the evaluation section we take this asymmetry into account by *not considering the option of standalone forecasting*.

There exist multiple options for choosing the forecasting method which were explored in Chapter 6. All the forecasting methods require prior exposure to the load to be tuned (ARIMA, Holt-Winters) or trained (LSTM RNN) appropriately. In case of the substantial change in the load pattern, e.g. from oscillations to oscillations with a trend, a forecasting model might require re-adjustment. Therefore, in production scenarios, it appears critical to offer a library of pre-tuned/pre-trained forecasting models corresponding to different load patterns that the autoscaler could choose from depending on the accuracy that they had shown recently.

[9] `https://netflixtechblog.com/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270`

10.4.3 Further Improvements with Application-specific Tuning

Deployed applications themselves offer a rich set of opportunities to improve the impact of autoscaling on user experience. Such *deterministic* optimizations are extensively targeted by systems researchers for certain application classes [92]. In this section, we suggest the optimizations that might be explored in conjunction with various predictive autoscaling policies. The proposed optimizations are not the main topic of the thesis – the goal of introducing them here is to show that complementing the predictive autoscaling by the application-specific optimizations may in fact increase the service level.

One of the most impactful optimizations for the transaction-based applications is the *requests reordering*. Reordering can be done in service buffers where requests wait to be processed. Reordering can be done according to one of many disciplines, which can vary between different services, to name a few:

- **First In-First Out** (FIFO) is a requests ordering discipline that orders the requests by the arrival time into the buffer (aka queue for FIFO). The request that arrived earlier than any other request is guaranteed to be considered for processing earlier than any other. The processing may still fail in the case when not all the responses were acquired (downstream processing).
- **Last In-First Out** (LIFO) is a requests ordering discipline that reverses the order of FIFO, i.e. the latest request will be processed first. This discipline barely has any meaning in the context of transaction-based application since it makes earlier requests wait for the processing of the latest which by definition violates the prioritization of processing requests rapidly in accordance with tight SLOs.
- **Oldest First** (OF) is a request ordering discipline that prioritizes requests that exist for the longest time, i.e. they have the largest risk of missing the response time SLO.
- **Priority-based** (PB) is a request ordering discipline that allows to assign priorities to various types of requests. Requests with the highest priority are taken from the buffer for processing at the earliest possible time. For instance, the payment transactions may be prioritized over the browsing requests. The challenge is that one has to carefully craft the prioritization scheme for potentially many different requests such that no bottlenecks are created by it and the DoS potential is not increased.

Requests can also be *batched* for processing to increase control over how the resources and the time of the deployed application are spent. Batching is usually done based on the *slack* for request processing which can roughly be defined as the difference between response time SLO and the response time itself (unused time before the deadline). If slack is positive, requests processing can be delayed till the batch of requests is filled. Such batch processing can also help improve performance of the applications since, for instance, lower number of API calls might be issued to databases. Bulk read/write operations are also enabled with batching which usually improves performance of disk I/O which is usually a bottleneck.

In addition to improving the relevance and the timeliness with deep learning and forecasting, the following section explores the effect of requests reordering on the quality of service delivered to the end user.

10.4.4 Experiments

10.4.4.1 Assessing the Impact of Timeliness and Relevance on the Service Level

Experiment goal. In this subsection, we evaluate the impact of timeliness and relevance of scaling actions devised with help of autoscaling policies on the response time.

Experiment configuration. To achieve this goal, we simulate three autoscaling policies:

- **Reactive** is a baseline autoscaling policy that provisions service instances and virtual resources based on the current metrics values compared against a threshold (like in a rule-based approach implemented in vanilla Kubernetes). The sync period is set to 30 s. We chose the **memory utilization** as a decision metric since the evaluated application is memory-intensive (we use the same synthetic application as in Subsection 10.3.2). The threshold is set at 80% of memory usage which is a conservative default for reactive autoscaling policies. The resulting number of service instances is increased by 15% following the heuristic of Kubernetes' vertical pod autoscaler. The adjustment policy for the VM clusters optimizes for the cost and allows the co-location of different services.
- **Model-based Reactive** is a reactive autoscaling policy but with the threshold scaling rule substituted for the deep learning model (cf. Subsection 10.4.1). The aim of this policy is to investigate the impact of improving the **relevance** characteristic of scaling actions on the quality of end user experience as characterized by the response time. Again, the policy uses 30 s sync period. **Model-based Reactive** policy uses a pre-trained 6-layer feedforward neural network that takes service instances count, load, and memory utilization as input and produces the average response time as the result. The network was trained on 6 load patterns, 2 of which were randomly chosen at each training repetition. Overall, roughly 15-20 repetitions were performed for the training. The concrete number is not stated since the model was allowed to further train during the experiment runs. To determine, whether the model can be used, the prediction at the current timestamp should be within the range of 75%-125% of the observed response time. The adjustment policy remains the same as in **Reactive**.
- **Predictive** is an autoscaling policy that uses both the deep learning model to improve the **QoS-relevance** and forecasting to improve the **timeliness** of scaling actions. This policy uses the same pre-trained 6-layer model as the **Model-based Reactive** policy. It adds a tuned Holt-Winters model to forecast the *load*. As such, the selection of a particular forecasting method does not matter from the standpoint of improving the timeliness of scaling actions. In the end, all the forecasting methods do the same thing – they extrapolate metric values into the future allowing to estimate the need in virtual resources and to issue or decommission them in advance. Accuracy of these techniques varies depending on parameters that are used to tune them, on the data used to fit these models, and on the forecasting horizon. Since there is no best forecasting method per se [176], we chose one of the simplest methods both in tuning and interpretation – Holt-Winters with additive trend and periodicity tuned to the load pattern. The sync time was set to 30 s and the forecasting horizon was 2 minutes to factor in the longest time for VM provisioning observed in practice [11]. The adjustment policy remains the same as in **Reactive**. A detailed discussion of the **Predictive** policy is provided in Appendix A.

The application configuration remained the same as in Section 10.3, but the application was exposed only to a single load pattern which is shown in Figure 10.8. Request timeout was set to 10 s. The overall duration of any experiment is 27 simulated minutes with the simulation step of 50 ms. During the first minute, the application was not exposed to any requests to allow it to finish its deployment. During next 5 minutes it gets 1 RPS of load constantly. For the next 20 minutes it becomes subject to load oscillating each minute between the levels of 3 and 1 RPS. In the leftover time the load drops to 0 to allow the application to finish the requests processing. Only one type of requests was used in the simulation to avoid diluting the focus of discussion. The experiment was repeated 15 times for each policy to improve the reliability of the results.

For the initial deployment, we've used 11 simulated t3.nano VM instances (AWS) and 12 t3.micro VM instances (AWS) which were randomly spread across 7 services with the instance count varying from 2 to 9.

Analysis of the results. As before, the user-experienced service level is assessed with the percentage of failed vs succeeded requests (cf. Figure 10.9a) and with the cumulative distribution of requests by response times (cf. Figure 10.9b). Before proceeding to the analysis, it is utmost important to note that the *absolute*

numbers of quality metrics are not conclusive because of two main reasons. First, the simulations step of 50 ms may result in the response time increased by a larger amount than it would otherwise have been if we used e.g. a 10 ms step. For instance, if the latency on the link is set to 10 ms, then the additional 40 ms will be added to the cumulative time of request as a result of making a simulation step. Second, the initial deployment was not fine-tuned for the experimental load which is otherwise expected to be done in the production environments. Therefore, the below analysis does not allow to argue about one particular policy being 'superior' to the others which is neither possible nor useful to the reader anyway. What it shows though is how the particular predictive components of these policies impact the user-experienced service level. We argue that this type of analysis yields deeper implications for the design of predictive autoscaling policies applicable over the broad range of predictive methods.

First thing to notice in Figure 10.9a is the continuous improvement in the percentage of fulfilled requests from **Reactive** (3) to **Model-based Reactive** (2) and then to the **Predictive** (1) policy. The major improvement of roughly 19% fulfilled requests occurs on adding the deep model and the optimization-based service instances allocation to the reactive policy. As described previously, this step improves the relevance of scaling actions. In contrast to the memory utilization-based **Reactive** policy, the **Model-based Reactive** policy is aware of the impact that the scaling actions make on the response time. The response time is a proxy for the fulfilled count of requests since the timeout is set to 10 seconds, meaning that the request is dropped if the response to it did not arrive in 10 seconds. Improving the timeliness of scaling actions by incorporating forecasting into the **Predictive** policy adds roughly 7% in fulfilled requests count on top of what the **Model-based Reactive** policy achieves. The improvement provided by adding the forecasting is less than the one added by introducing the deep model-based service instance allocation. When making conclusions about the value of forecasting for autoscaling, one needs to factor in the duration of the simulation which was just 27 minutes. Forecasting tends to pay off on the longer intervals of time, hence we expect forecasting to introduce additional improvement when running continuously over prolonged periods of time in production. Nevertheless, given the results in Figure 10.9a, we may conclude that improving the relevance outweighs improving the timeliness of the metric by roughly $\times 2.8$. Here, we also need to make a statement that the load with higher increase rate or higher amplitude of oscillations will likely render the improvement in timeliness through forecasting more valuable.

In contrast to Figure 10.9a, the CDF plot in Figure 10.9b is less conclusive. Averaging over multiple simulations yields almost identical CDF curves. However, these curves are quite different, when one takes into account that these are provided only for the *fulfilled* requests. Hence, the 95th percentile of response time that appears at roughly 6.5 seconds both for the **Model-based Reactive** policy and the **Predictive** policy, corresponds to different request counts. For **Predictive**, it is 1882 ($0.7339 \times 0.95 \times 2700$) of 2700 requests generated in the experiment, whereas for the **Model-based Reactive** policy it is 1708 ($0.6662 \times 0.95 \times 2700$). Almost 180 more requests enjoy the response time which is under 6.5 seconds with the **Predictive** policy as compared to **Model-based Reactive** one. Using the same calculation for the **Reactive** policy, we arrive at 1228 requests that are served under 6.5 s. This number roughly corresponds to the 68th percentile for the **Model-based Reactive** policy and to the 62nd percentile for the **Predictive** policy. Thus, at the 95th percentile, we get 480 requests by switching from **Reactive** to the **Model-based Reactive** policy, and almost another 180 requests by employing the **Predictive** policy. This result demonstrates the importance of improving both the *relevance* and the *timeliness* properties of scaling actions.

The combined results from figures 10.9a and 10.9b support the hypothesis that the improvements in the *relevance* and *timeliness* of scaling actions yield better user experience in terms of response time. Simulations show that improving the timeliness is less important than improving the relevance of scaling actions although it still allows to increase the count of timely served requests by roughly further 10%. Since the intrinsic asymmetry between timeliness and relevance does not allow to use the load metric forecast without its mapping onto some QoS-relevant metric, we cannot really argue whether the improvement demonstrated

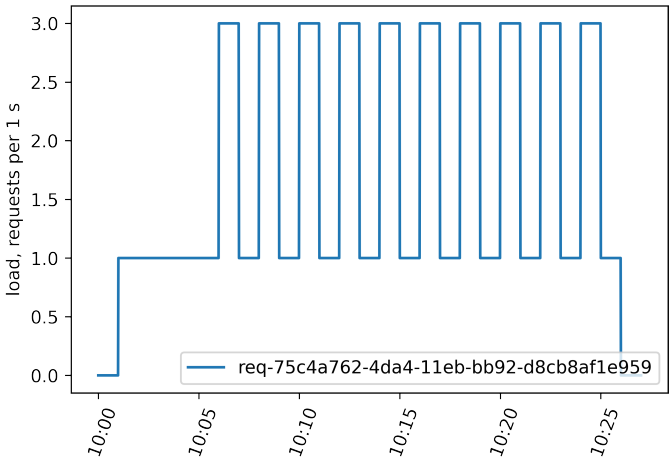
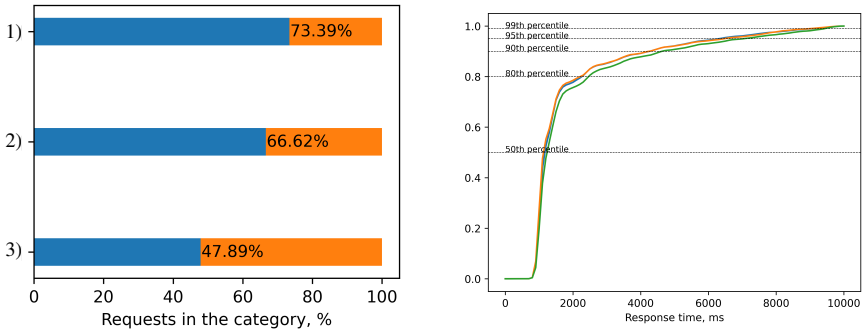


Figure 10.8.: Lagging oscillating load pattern.



(a) Requests fulfilled/failed by policy: 1) **Predictive**, 2) **Model-based Reactive**, 3) **Reactive**.

(b) CDF of response time by policy: Reactive, Model-based Reactive, Predictive

Figure 10.9.: Assessing the relevance/timeliness impact on the service level for three autoscaling policies. Experiment was repeated 15 times.

by adding forecasting should be attributed purely to it or rather to the interplay between the forecasting method and the deep model-based service instances allocation. Although CDF plots are almost identical for the evaluated policies as was seen in Figure 10.9b, accounting for the fact that these plots picture CDFs only for the fulfilled requests gives quite a different picture. Both the **Model-based Reactive** policy and the **Predictive** policy demonstrate a significant improvement in terms of amount of requests fulfilled until the deadlines specified by percentiles (e.g. we studied the case of 95th percentile of response time) in comparison to the **Reactive** policy. This nonlinear improvement becomes even clearer on a CDF plot which is based on numbers normalized by the largest amount of fulfilled requests across all three evaluated policies, i.e. that of **Predictive** (cf. Figure 10.10).

Another advantage of the **Predictive** policy is its better use of resources which manifests itself in the average cost of 0.0834 USD per simulation, which is a bit lower than 0.0899 USD for the **Model-based Reactive** policy. **Reactive** policy lags behind these two with the average cost of 0.1064 USD per simulation. We attribute the lowest cost for **Predictive** policy to its ability to forecast not only the spikes, but also the dips in load, thus issuing the down-scaling whenever appropriate.

Conclusion. To achieve the superior end user experience, the scaling policy has to improve both relevance

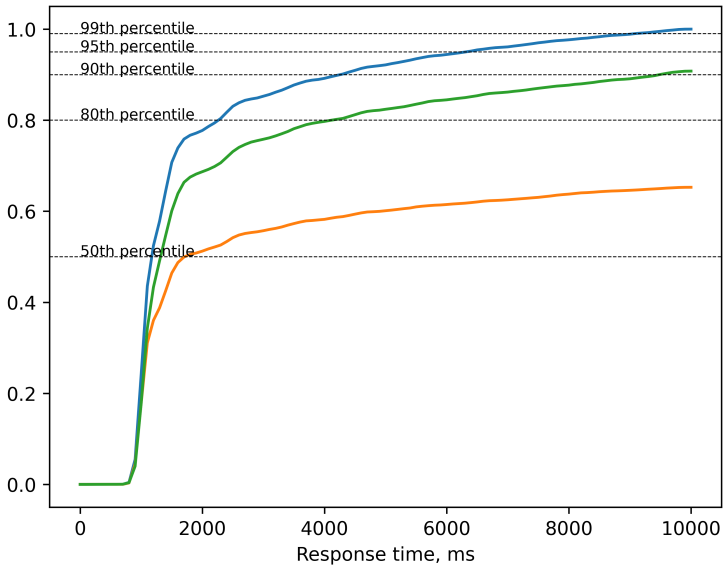


Figure 10.10.: Normalized CDFs for assessing the relevance/timeliness impact on the service level for three autoscaling policies: Reactive, Model-based Reactive, Predictive

and timeliness. We had shown that these improvements can be achieved by introducing forecasting and deep learning models into the autoscaling process, but there are also other ways to improve these characteristics. For example, this could be achieved by improving the hypervisor (to reduce the VM provisioning time) or by organizing the requests processing in a way that is optimal for the given load. The following subsection assesses the latter optimization opportunity.

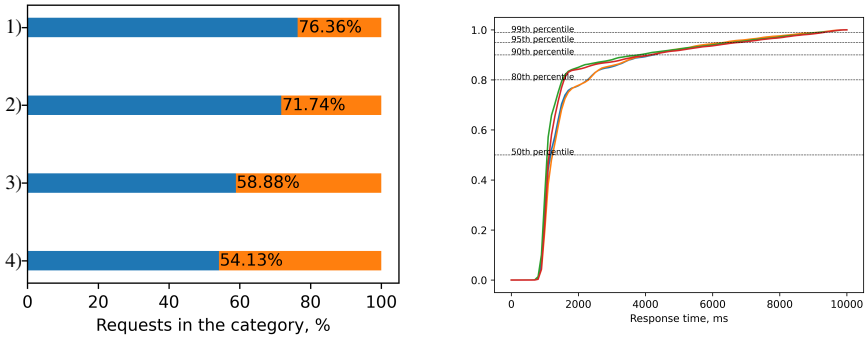
10.4.4.2 Assessing the Importance of the Application-specific Optimizations

Experiment goal. Experiment provided in this subsection aims to assess whether the application-specific optimizations such as requests reordering may complement effectively the predictive autoscaling policies.

Experiment configuration. The experiment configuration is almost entirely identical to the previous one with several exceptions. First, we do not consider the **Reactive** policy. Second, each of the **Reactive with Model** and **Predictive** policies gets two versions – one with buffers ordering the requests/responses according to the first in-first out (FIFO) discipline, and another with buffers ordering the requests/responses in such a manner that the oldest request/reponse is served first (OF).

Analysis of the results. In Figure 10.11, we observe familiar trend of improving the response time and the count of fulfilled requests by adding the forecasting to the reactive policy that uses the deep model to map the metrics to the response time. However, we are interested in another tendency here, viz, that the FIFO ordering outperforms OF ordering in terms of fulfilled requests by 17.5% in **Predictive** case and by 17.6% in **Reactive with model** case (cf. Figure 10.11a). The same improvement is observed both on non-normalized (Figure 10.11b) and normalized (Figure 10.12) response time CDF plots.

Apparently weird result for the Oldest-first (OF) policy in comparison to the FIFO policy is in reality well-explainable. By attempting to propel the oldest requests and responses through the application, we actually favor those requests that are *close to being dropped*. One might argue that this was exactly the reason why this policy should perform better than FIFO. But that is not the case when there are many old requests. Since we tend to favor them, we are trapped in a situation when an unlucky oldest request gets picked for



(a) Requests fulfilled/failed by policy: 1) **Predictive** (FIFO), 2) **Reactive with model** (FIFO), 3) **Predictive** (OF), 4) **Reactive with model** (OF)

(b) CDF of response time by policy: **Predictive** (FIFO), **Reactive with model** (FIFO), **Predictive** (OF), **Reactive with model** (OF).

Figure 10.11.: Assessing the impact of requests reordering on the service level for four autoscaling policies. Experiment was repeated 15 times.

the propagation, but still does not make it since its timer is close to expiration. This is especially important for the propagation of responses since in our application configuration there are several points where the services have to wait for multiple services to reply in order to proceed. Overall, we observe that the system-level optimizations can both improve and harm the end user quality of service delivered by the autoscaling policy.

Conclusion. This experiment demonstrated the importance of accurate application-level optimizations since a wrong optimization can dwarf all the improvements introduced by predictive autoscaling. However, when a right optimization is made, it can serve as a multiplier to the effect of predictive autoscaling.

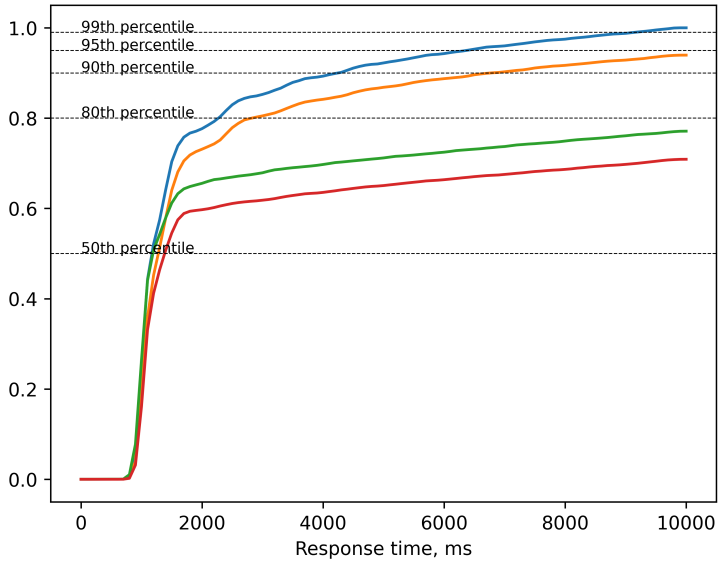


Figure 10.12.: Normalized CDFs for assessing the impact of requests reordering on the service level for four autoscaling policies: **Predictive** (FIFO), **Reactive with model** (FIFO), **Predictive** (OF), **Reactive with model** (OF).

10.5 Accountability of the Application Topology for the Success of Autoscaling

10.5.1 Experiment Motivation

Microservice applications vary in their topology depending both on the business logic and on the need of supporting services like message queuing and configuration management [159]. Often, the application topology is not the result of a concerted and well-planned effort from the architect's side, but it is rather a by-product of the natural process of application architecture evolution – new services are getting added, old ones are removed or merged. As long as the application meets both the functional and non-functional requirements, its topology often falls out of the developer's field of view.

Third-party services and frameworks play an important role in the topology evolution of the microservice application. Being responsible for the common functionality (data storage and access, configuration management, message streaming, locking, etc.), these components usually support multiple services implementing the business logic. With so many services relying on them, the third-party services are naturally 'promoted' to become the most critical components of the application. In turn, this results in the predominance of scale-free network topology among the microservice applications [159].

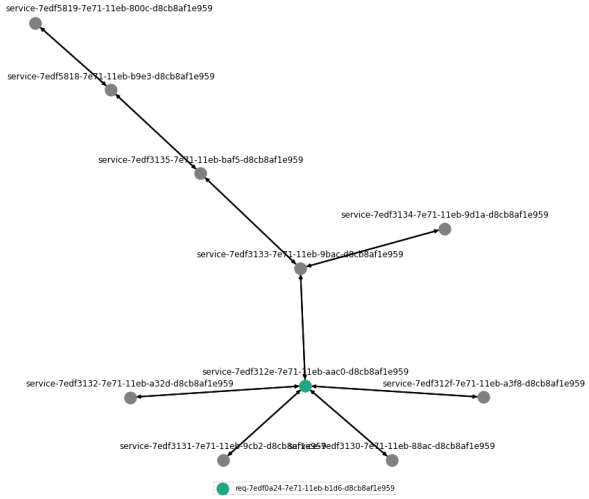
Differences in application topologies may result in different autoscaling behaviors. For instance, if one adopts the reactive utilization-based autoscaling policy for the microservice application with a structural bottleneck, it is likely that under the increasing load the resources of the bottleneck service will become saturated and the request queue will start to build up. When this service is scaled out, however, the problem is not gone. Now, its upstream services may become new bottlenecks and, again, will have to be scaled out. Thus, certain application topologies may result in an uneven resource utilization, thus paving the path to the scaling behavior that differs across the application services conditioned on the application topology.

10.5.2 Experiment

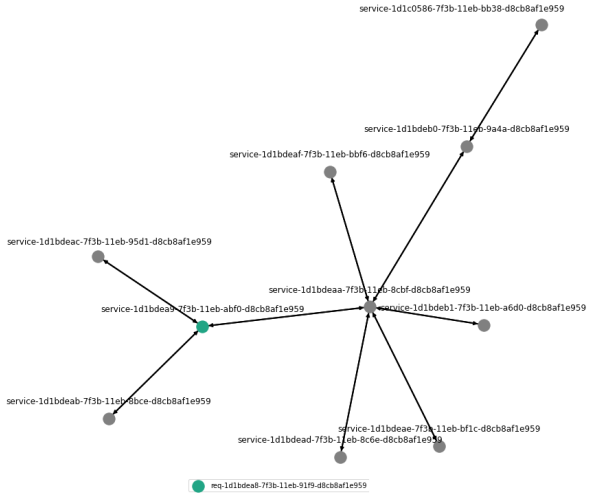
Experiment goal. In this experiment, we aim to explore how the differences in topologies of realistic applications impact the quality of autoscaling decisions taken under different policies, both predictive and reactive.

Experiment configuration. To achieve that goal, we used **Praxiteles** to synthesize four artificial applications based on Azure traces with each BA-model parameter set from Table 9.1. Each generated topology is distinguished by the last letter of the parameter set name from the table, i.e. for the set `single_center_b` the generate topology name is Topo-B. More discussion on discovering these parameter sets and the model to generate the application topologies is available in Chapter 8. The generated topologies and the concrete parameters from the sets used for their generation are shown in Figure 10.13. Each application has only 10 services and one entry point for the user requests (highlighted with a different color). Only one request type is used to make the conclusions from the experiments tractable.

Visual inspection of the application topologies in Figure 10.13 yields that they are not isomorphic to each other, and thus they represent a diverse set of topologies.Topo-A is distinguished by its branched structure, straight at the entry service – 5 edges are connected to it. However, these branches are not equal: 4 of them contain only one edge, whereas one exhibits a full 4 services in line on one of its subbranches. Topo-B clearly exhibits the scale-free property since just one of its nodes has way more connections (6) than the others (mostly 1-2, and an entry node with 3). In this sense, Topo-B is a bit different from Topo-A with the latter

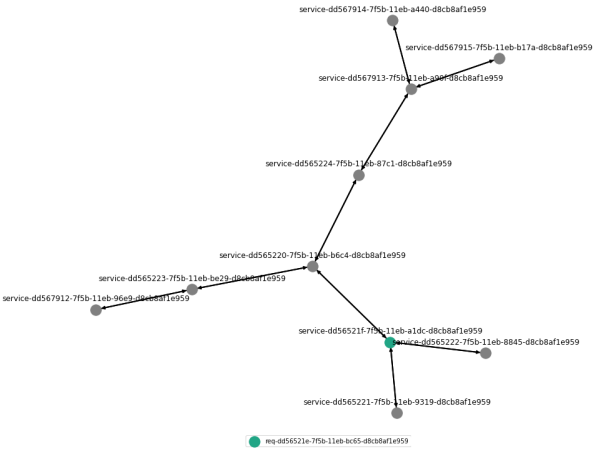


(a) Topo-A topology. Parameters used for generation: power of preferential attachment = 0.05, appeal of vertices with zero edges = 0.01.



(b) Topo-B topology. Parameters used for generation: power of preferential attachment = 0.9, appeal of vertices with zero edges = 0.01.

(c) Topo-C topology. Parameters used for generation: power of preferential attachment = 0.05, appeal of vertices with zero edges = 3.25.



(d) Topo-D topology. Parameters used for generation: power of preferential attachment = 0.9, appeal of vertices with zero edges = 3.25.

Figure 10.13.: Artificially generated topologies for the experiment showing the impact of topology on autoscaling. Each application consists of 10 services with a single entry service (highlighted). Each topology was generated by **Praxiteles** based on tuned BA-model.

being more linear. Topo-C distinguishes itself by having a relatively balanced structure – each of 4 branches connected to the entry service vertex does not have vertices with more than 2 edges. Two of these branches are equal in length, with further two being shorter by 1 and 2 edges. BA-model parameters used for Topo-D force this topology to form multiple well-connected vertices which is visible in Figure 10.13d. Introduction of vertices with more than 1 adjacent edge has an important practical implication for the response times. The more there are well-connected services with high fan-out, the longer it will usually take for the responses to be propagated to the user. The reason is that each such service has to wait for *all* the responses from its upstream neighbors. This is exactly the case for the synthesized Topo-D application topology.

Analysis of the results: fitting the performance model. As was described previously, we use the *feedforward neural nets* to regress the response times on the load and other system metrics. We used the feedforward network with the same architecture as in Section 10.4 for each service in each application. To train these models, we used the **Training Ground** tool introduced earlier. It iterated over two load patterns to train the model based on the simulation results. The data was added to the minibatch of 10 samples each 3 s of simulated time, thus the weights update was performed roughly each 30 s of the simulated time. Both load patterns are based on the SARIMA model with their configurations provided in listings 10.1 and 10.2. The main difference between these two load patterns is in the presence of trend (constant level vs increasing trend) and the parameters of the time series generation model. Both patterns are periodic with 2 m period which is barely observable either due to the increasing pattern or due to the standard deviation in the number of requests at each second set to 10. The coefficients are selected in such a way that the generated time series neither fades nor explodes. The samples of the corresponding load patterns are shown in Figure 10.14 with a 1 s resolution. We do not provide more load patterns since the load pattern sampled from SARIMA will differ at each simulation run; the load patterns in Figure 10.14 are just two such samples. However, the main structural properties, such as trend, amplitude, and periodicity, persist across these samples. We use scaled error as a loss function to dynamically assess the accuracy of predictions delivered by the performance model. It is computed as follows:

$$SE = \frac{|RT_{predicted} - RT_{current}|}{\max(RT_{predicted}, RT_{current})} \tag{10.1}$$

The value computed with the above formula reflects the percentage by which the predicted value differs from the actual one. Smaller values of the scaled error mean higher model accuracy. The advantage of using this metric is that one can intuitively set a threshold that will determine whether the trained model is ready to be used. During early stages of training, one can set this threshold in an arbitrary way since the model anyway needs some time to get to the useful state. From our experience, the values in diapazone 0.15 - 0.4 are the most practical. The seemingly high value of 0.4 is caused by the high dynamics of the system and serves to account for high variability in response times. By setting such a high value, one puts some trust into the trained model. Otherwise, if the calculated scaled error is higher than the threshold, the fallback service instances number calculator is used, e.g. a simple ratio-based calculator.

```
{
 "load_kind": "leveled",
 "batch_size": 1,
 "generation_bucket": { "value": 1, "unit": "s" },
 "regions_configs": [{
    "region_name": "eu",
    "pattern": {
        "type": "arima",
        "params": { "head_start": { "value": 1, "unit": "m" },
```

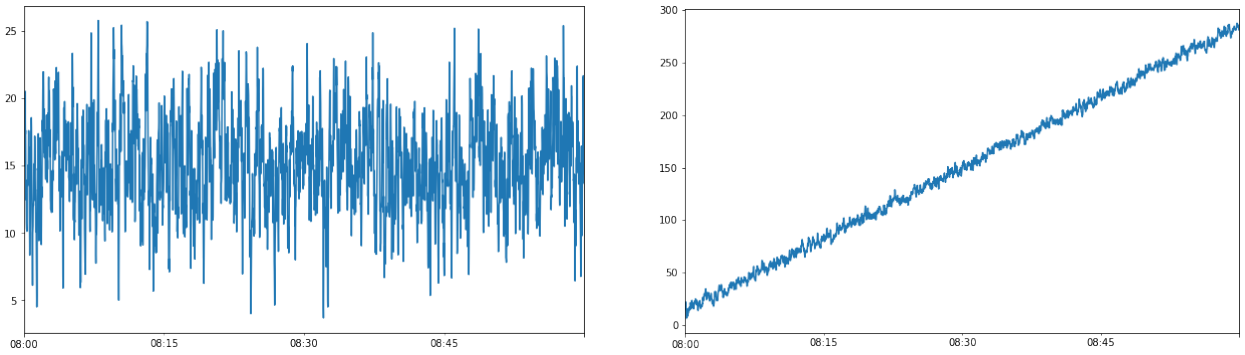
```
                    "cooldown_end": { "value": 3, "unit": "m" },
                    "duration": { "value": 6, "unit": "m" },
                    "resolution": { "value": 1, "unit": "s" },
                    "scale_per_resolution": 15,
                    "model": {"period": {"value": 2, "unit": "m"}, "trend": "c",
                              "parameters": {"p":2, "d":0, "q":1, "P":0, "D":0, "Q":1},
                              "coefficients": [0, .8, .01, .5, .01, .01]}}},
    "load_configs": [{
        "request_type": "req-7edf0a24-7e71-11eb-b1d6-d8cb8af1e959",
        "load_config": {"ratio": 1.0, "sliced_distribution":
                        {"type": "normal", "params": {"sigma": 10}}}}]
    }]
}
```

Listing 10.1: First load pattern for performance model training in the topology experiment.

```
{
 "load_kind": "leveled",
 "batch_size": 1,
 "generation_bucket": { "value": 1, "unit": "s" },
 "regions_configs": [{
    "region_name": "eu",
    "pattern": {
        "type": "arima",
        "params": { "head_start": { "value": 1, "unit": "m"},
                    "cooldown_end": { "value": 3, "unit": "m" },
                    "duration": {"value": 6, "unit": "m"},
                    "resolution": {"value": 1, "unit": "s"},
                    "scale_per_resolution": 15,
                    "model": {"period": {"value": 2, "unit": "m"}, "trend": "ct",
                              "parameters": {"p":2, "d":0, "q":2, "P":0, "D":0, "Q":1},
                              "coefficients": [.001, .001, .75, .05, .5, .01, .01, .01]}}
                                 },
    "load_configs": [{
        "request_type": "req-7edf0a24-7e71-11eb-b1d6-d8cb8af1e959",
        "load_config": {"ratio": 1.0, "sliced_distribution":
                        {"type": "normal", "params": {"sigma": 10}}}}]
    }]
}
```

Listing 10.2: Second load pattern for performance model training in the topology experiment.

The impact of various application topologies on predictive autoscaling appears already at the performance modeling phase. Figure 10.15 demonstrates that only the performance models for the simulated applications with topologies Topo-C and Topo-D were able to demonstrate appropriate accuracy below or around the selected threshold of 0.25 for the scaled error. Interestingly, despite taking roughly $\times 30\%$ less iterations to train the performance model for Topo-C (cf. Figure 10.15d vs Figure 10.15c), we observe that the models for 9 out of 10 services consistently demonstrate low scaled error at around 0.10-0.15, whereas 1 services approached the threshold after some detour right at the moment when the learning stopped (this sudden



(a) Constant load pattern.

(b) Increasing load pattern.

Figure 10.14.: Artificially generated load patterns samples to train the performance models for the experiment involving applications with different topologies.

increase in the scaled error is discussed in the following paragraphs). At the other hand, the performance model accuracy for Topo-D gets consistent early enough, at roughly 3500 iterations (cf. Figure 10.15d), albeit being lower on average. The amount of iterations spent on training the performance model for Topo-D dwarfs that of Topo-B (almost $\times 3$ more), which is explained by the amount of time spent on each simulation. Particularly, Topo-B took the longest amount of time for each simulation, whereas Topo-D took the smallest amount of time, therefore the latter managed to finish more iterations in 10 days. The lowest amount of training iterations may in fact explain the highest scaled error of the performance model for Topo-B with a model only for 1 out of 10 services exhibiting consistent downwards trend after 2600 iterations (cf. Figure 10.15b). Topo-A and Topo-C both fell somewhere in-between accomplishing almost 6000 iterations of training.

Closer look at the services with the highest scaled error in all four plots in Figure 10.15 reveals that almost all the services with more than 2 links (i.e. fan-out or fan-in is more than 1) made it into this category. In particular, we see the highest scaled error achieved for the entrance service in every topology. This certainly is not a surprise since this service accumulates the dynamics of the whole application being both the entry and the exit for the whole application. The measured response time at this service is subject to impact from all the other services that are encountered on the request path. Hence, however complex the topology is, the performance model for the entry service will be among the most sensitive, since it captures the behavior of the whole application. Figure 10.15c illustrates this finding in the clearest way by the scaled error for the entry service being catapulted at around 2000 iterations, whereas scaled error for the remaining services plummets.

Another important point of performance model error accumulation is in the services that connect to more than 2 others (i.e. request path branches). These may correspond to the entry service, but not necessarily. For example, the single such service (aside from the entry) for Topo-A (cf. Figure 10.13a) with one branch connecting it to the entry and two others connecting to the sequence of 3 services and 1 service correspondingly, consistently demonstrates scaled error above the threshold of 0.25 in Figure 10.15a. We hypothesize that the difficulties in learning the performance model for this service are connected to the complex dynamic behavior that should be captured by the model. In particular, before proceeding with its response, this service has to wait for the responses from two upstream branches that differ $\times 3$ in the number of services.

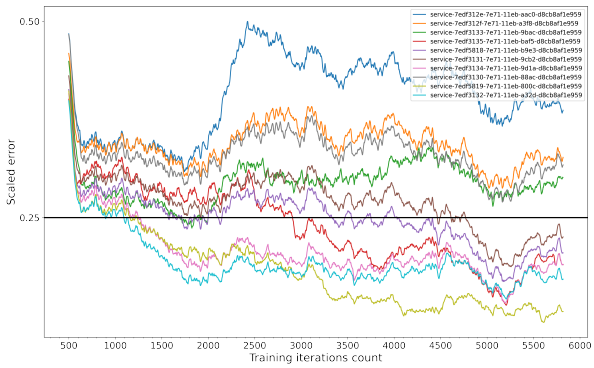
The detour in all the scaled error plots of Figure 10.15 observed at around 2000-2500 iterations, was introduced by changing some parameters of the training process. The goal was to check how long would it take for the training process to recover from that disturbance. With that small change we observe a set of different behaviors across the board. Performance models for the simulated applications with topologies Topo-A

and Topo-B (cf. Figure 10.15a and Figure 10.15b) did not seem to recover during the training process, even though Topo-A got $\times 1.53$ more iterations than Topo-B. In the case of Topo-A, this might be connected to the imbalanced structure of the request subpaths already discussed in the previous paragraph. In contrast, the similar result for Topo-B might be attributed either to the service with the highest number of connections (6) among all the application topologies studied or to the count of training iterations being lower compared to the other 3 applications. When looking at the training results for the performance models of applications with the topologies Topo-C and Topo-D, one may observe that both of them manage to recoup the training progress after some iterations. However, there is still a substantial difference in the training dynamics between these two applications after the change is introduced. Particularly, the disturbance impacts the training dynamics of the majority of services (9 out of 10) for Topo-C positively rather than negatively (cf. Figure 10.15c). However, it is not the case for the entry service whose scaled error almost doubles in the scope of 500 iterations. It takes further 3000 iterations for its performance model to get to the 0.25 scaled error threshold whereas the performance models for all the other services of this application seem to stabilize in their learning progress for the same 3000 iterations. In contrast, in the application with the Topo-D topology, only one service gets dramatically affected by the change in the training parameters – it increases 2-fold in the interval of 500 iterations (like in Topo-C), but then rapidly regains its scaled error level of 0.25 in the next 500 iterations (cf. Figure 10.15d). Other services for the Topo-D topology do not seem to be significantly affected by the disturbance. The relative insensitivity to the abrupt change in the experiment configuration might follow from a realtively well-balanced application structure for Topo-D, viz, the two branches that are attached to the second center (the first one is the entry service) differ in length only by 1 hop.

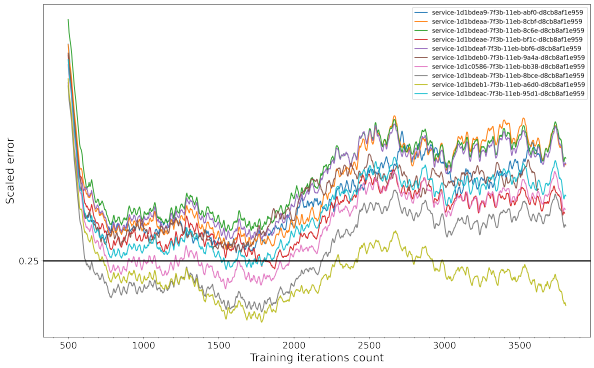
Paying more attention to Figure 10.15, one may notice that the scaled errors for different services tend to form clusters for topologies **Topo-A**, **Topo-B**, and **Topo-C**. However, this a spurious pattern. Identification of services from the same cluster in topologies depicted in Figure 10.13 reveals that most often the services that have close scaled error curves may in fact be on different branches in respect to the entry service. Hence, we conclude that relative groupings of services imposed by the topology have no visible effect on the training speed trend and its direction.

Despite not having enough evidence on the reasons for the observed training behaviors (due to deep leraning models being opaque), we can certainly state that the training progress depends on the application topology. A particular topological feature that appears to matter in the training progress is the count of services with more than 2 links and the balance in the length of requests paths.

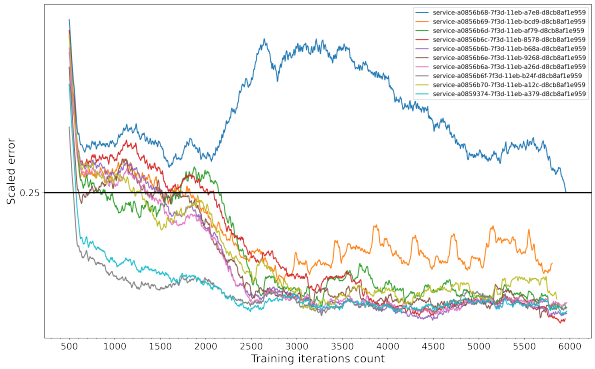
Analysis of the results: evaluating the impact of the application topology on the efficiency of autoscaling. To get the credible simulation results for assessing the impact of topologies on autoscaling with different policies, we repeated each experiment 5 times. The load pattern was sampled using the configuration in Listing 10.1 (cf. Figure 10.14a for the graphical representation). We have used two out of three policies introduced in Section 10.4, viz, **Predictive** and **Reactive**. The main difference is that the load forecasting model used in the **Predictive** policy this time was SARIMA with the same parameters as the ones used for generating the load. Important to note that having the same parameters in no way guarantees the perfect forecasts due to two key reasons: 1) upon each sampling, we get a different load pattern due to stochasticity of the SARIMA process; 2) using the SARIMA-generated value as a mean for the normal distribution with some standard deviation (10 in our experiments) adds another source of stochasticity to the load generation. Having discussed the adjustments to the autoscaling policies, we would like to stress our argument from the previous sections – *the absolute numbers acquired in the experiment do not matter*. In the following paragraphs, we *focus on the difference between the results for the two evaluated policies*. This allows us to draw general conclusions about the role that the application topology plays in autoscaling with different approaches, i.e. reactive vs predictive.



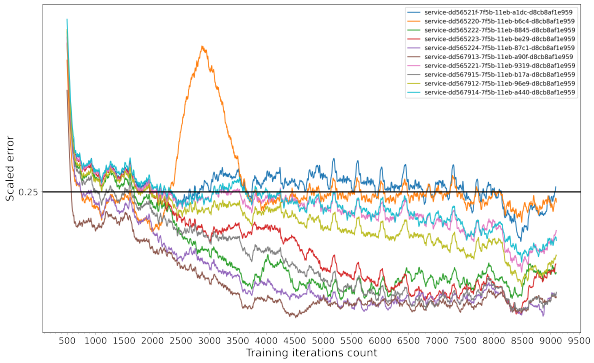
(a) Application topology Topo-A.



(b) Application topology Topo-B.

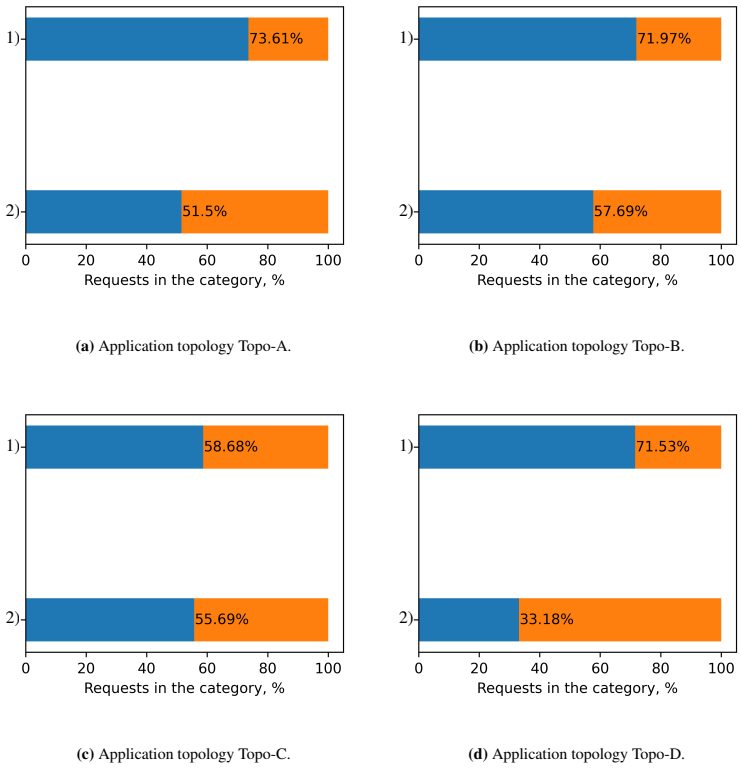


(c) Application topology Topo-C.



(d) Application topology Topo-D.

Figure 10.15.: Training progress for the feedforward network used in performance modeling of applications with different topologies.



(a) Application topology Topo-A.

(b) Application topology Topo-B.

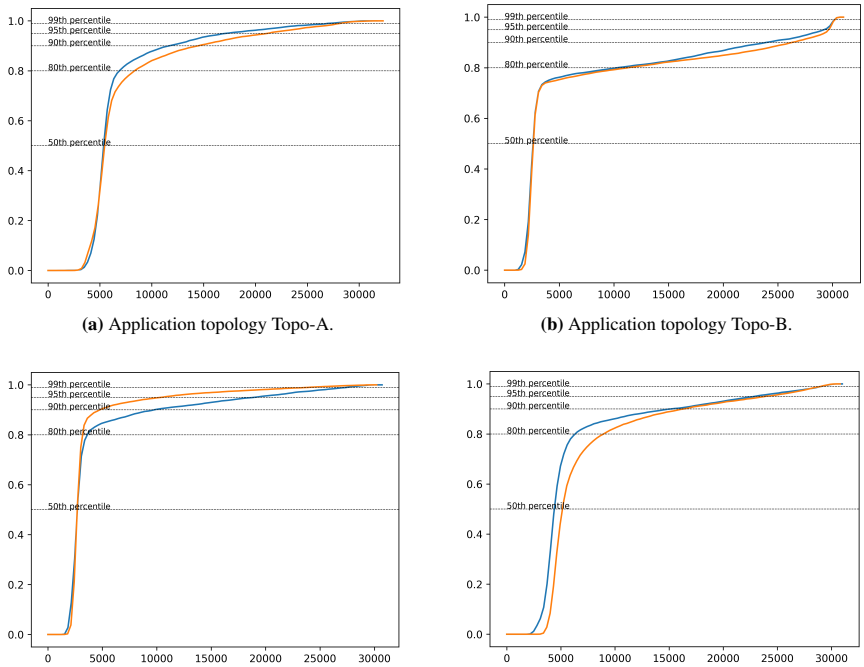
(c) Application topology Topo-C.

(d) Application topology Topo-D.

Figure 10.16.: Fulfilled vs failed requests distribution in the experiment to assess the impact of application topologies on autoscaling for topologies depicted in Figure 10.13. Bars in order from top to bottom: 1) **Predictive** autoscaling policy; 2) **Reactive** autoscaling policy. Joint legend: fulfilled requests are in blue (left part), failed requests are in orange (right part).

As in the evaluation in Section 10.4, **Predictive** autoscaling policy demonstrates the superior results in comparison to the **Reactive** one (cf. Figure 10.6). Nevertheless, the difference between the results for these two policies across the evaluated application topologies is substantial. In Figure 10.16a, we observe that employing the **Predictive** policy gives additional 22.11% of fulfilled requests for the application topology Topo-A. In stark contrast, the very same **Predictive** policy gives a mere 2.99% rise in the number of fulfilled requests for Topo-C (cf. Figure 10.16c). Interestingly, the performance models for both topologies were trained for the same count of iterations, and overall the per-service performance models for Topo-A demonstrated worse accuracy on predicting the response times than the ones trained for Topo-C (cf. Figure 10.15a vs Figure 10.15c). From that alone, we may conclude that the application topology has multiple ways to influence the autoscaling behavior. Analyzing the impact of topologies on the performance models alone is not sufficient to draw the conclusions about the overall effect that the topology will have on autoscaling.

Another important observation is that the lowest number of training iterations among all the topologies did not preclude the **Predictive** policy for Topo-B to achieve the results on par with that of Topo-A and Topo-D (cf. Figures 10.16a, 10.16b, 10.16d). Lower number of training iterations still seemingly manifests itself in the lowest gap between the percentage of fulfilled requests for **Predictive** vs **Reactive** policy in the Topo-B case among these three topologies, viz, 14.28% vs 22.11% for Topo-A and 38.35% for Topo-D. The latter number is also interesting – the difference of 38.35% in fulfilled requests between the **Predictive** and **Reactive** policies for Topo-D should be attributed not to the **Predictive** policy being superior to the **Reactive** one, but rather to the **Reactive** policy being surprisingly bad for this particular topology. Thus, we have another bit of evidence that the impact of topology on autoscaling is not limited to the performance models.



(a) Application topology Topo-A.

(b) Application topology Topo-B.

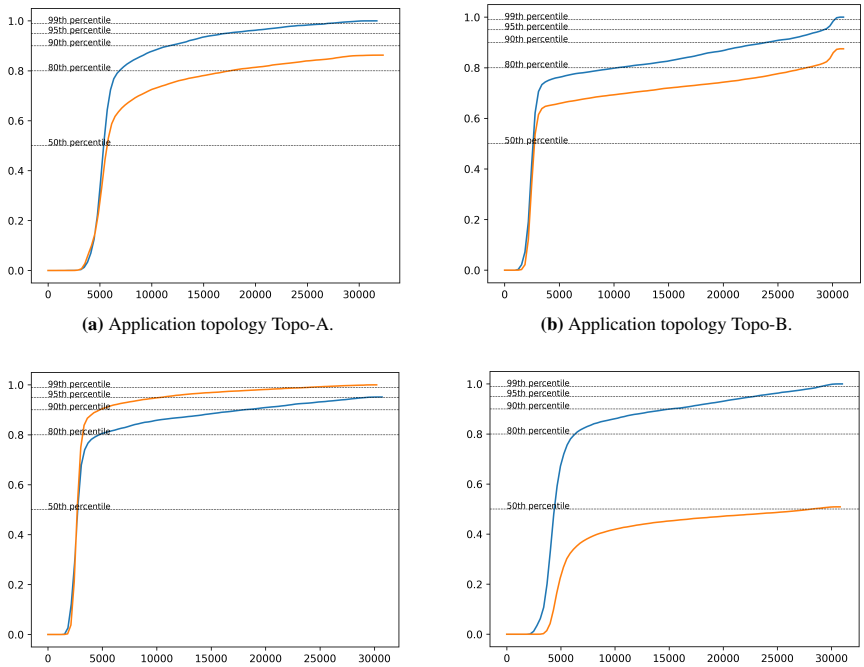
(c) Application topology Topo-C.

(d) Application topology Topo-D.

Figure 10.17.: Response time CDFs in the experiment to assess the impact of application topologies on autoscaling for topologies depicted in Figure 10.13. Joint legend: Predictive autoscaling policy, Reactive autoscaling policy.

Similarly to the plots in Figure 10.16, response time CDFs in Figure 10.17 clearly show differences both between the policies and between the topologies. In particular, we observe that the **Predictive** policy demonstrates better response times distribution for the topologies Topo-A, Topo-B, and Topo-D. Despite this similarity, the shape of CDFs differs across the topologies. For instance, Topo-A and Topo-B demonstrate similar behavior on the first part of the curve – CDFs for both policies on both plots (cf. Figure 10.17a and Figure 10.17b) rise rapidly without much difference between them. Following, we observe a clear difference when CDFs saturate – CDF for Topo-A reaches higher percentiles (80th, 90th, 95th) sooner than Topo-B and in general demonstrates better CDF. CDF for Topo-D is quite different from Topo-A and Topo-B in a sense that the **Reactive** policy demonstrates quite a bad CDF up till the 90th percentile where it catches up with the CDF of the **Predictive** policy. Similarly to this observation, we already observed the substantial difference between these policies for Topo-D in Figure 10.16. Lastly, in the case of Topo-C, we observe better response time CDF for the **Reactive** autoscaling policy both in the raw and in the normalized form (cf. Figures 10.17c and 10.18c). Overall, all four CDFs in both forms exhibit clear difference in the amount of requests that gets served in the same time. For instance, if we take 5 seconds, then the **Predictive** policy for the application with the Topo-A topology can serve roughly 50% of requests in that time (the same as in the **Reactive** policy for Topo-D), whereas Topo-B is able to take on 25% more in the same time, but fails at maintaining this dynamics when proceeding to higher percentiles. In stark contrast, Topo-C requires the same amount of time with the **Predictive** policy to serve roughly 85% of all the requests; Topo-D is just 5% lower than this result (cf. Figure 10.17d).

Conclusion. There are two major conclusions that we may draw from the experiment in this section. First, the application topology indeed impacts autoscaling, including the performance modeling stage and the service level achieved with the help of the autoscaling policies. Second, the inherently complex system dynamics imposed by the interplay of the application topology with other parameters of the deployed application precludes us from capturing all the influences of application topology in the performance model.



(a) Application topology Topo-A.

(b) Application topology Topo-B.

(c) Application topology Topo-C.

(d) Application topology Topo-D.

Figure 10.18.: Normalized response time CDFs in the experiment to assess the impact of application topologies on autoscaling for topologies depicted in Figure 10.13. Joint legend: Predictive autoscaling policy, Reactive autoscaling policy.

Careful selection of the load patterns combined with the prolonged training of the performance model may help in improving the generalizability of the performance model. However, one needs to be aware that performance modeling still needs to be done for each application topology anew as its results do not generalize across applications with different topologies.

10.6 Impact of Resource Limits on VMs Allocation Efficiency in Predictive Autoscaling for Multilayered Deployments

10.6.1 Experiment Motivation

Scaling multilayered deployments raises concerns on whether the dynamic adaptation of the virtual infrastructure layer (VMs) is done efficiently based on the planned scaling actions for the application layer. To recap, the approach to scaling the multilayered deployments in this thesis is similar to the one implemented in Kubernetes. First, the autoscaler derives the number of services that are will be needed to process the load. Second, the autoscaler figures out the cumulative resource requirements of the scaled services using the resource limits set for them in the configuration. Third, these cumulative resource requirements are compared against what various types of VMs could offer in terms of system resources, and the count of VMs needed to host these services instances is computed. Lastly, the cost of each placement is computed and the cheapest one is selected[10]. Thus, the most important tuning knob that bridges these two abstract resource provisioning layers is the *resource limit*.

[10] Subsection 7.3.3 provides a bit more details on the whole process

In this thesis, we treat the resource limits in the *reactive* sense, i.e. crossing the resource limit is allowed, but once the violation is observed by the system, it attempts to fix this[11]. The resource limit might be set for any type of system resource that is available for the service's use, e.g. vCPU or memory. For the convenience of the researcher, the **Multiverse** simulator supports two options to set the limits in the application configuration file: 1) resource limit in the units of the resource (e.g. vCPUs); 2) resource limit as percentage of the **mean** resource utilization of the given service (e.g. 150% of the mean RAM utilization of 200 MB, i.e. 300 MB).

Although setting the resource limits as a percentage of the actual mean resource utilization makes designing the experiments easier, the complex system dynamics still gets in a way of arguing for the appropriateness of smaller vs larger resource limits. This challenge is best illustrated with an example. If the resource limits are too large, then it may happen that either most of the VM types would end up not being able to offer enough resource to place the service instances or the resources will end up being wasted. However, it may also turn up as a positive thing in that the incoming requests will get enough resources to be processed. If that's the case, then the buffers won't build up and the response times will likely be low. On the other hand, if the resource limits are large, only a handful of service instances will be allocated, hence limiting the number of requests that can be processed in parallel. Lower resource limits may help to reduce the resource waste and thus the overall cost of running the deployment, but this will likely result in a smaller resource headroom for processing the requests. In turn, this will propel rising response times.

To sum this discussion up, finding the appropriate resource limits to set up for scaling the multilayered deployment is not an easy task. It depends on many factors, to name a few: a) application owner's goals, i.e. balance of the deployment cost vs the end user satisfaction; b) complex internal application's dynamics dictated by the resource utilization patterns and the topology; c) variability of load and its resource demands.

This section attempts to highlight the complexity of finding the right resource limits for the application to make the VMs allocations with the predictive autoscaling policy efficient for the multilayered deployments.

10.6.2 Experiment

Experiment goal. The experiment in this section aims to provide an insight into the complexity of tuning the resource limits to achieve efficient predictive autoscaling for the multilayered cloud deployments.

Experiment configuration. The base configuration for this experiment is exactly the same as in the previous section (cf. Section 10.5). This experiment was run with the same four applications with the topologies provided in Figure 10.13, hence we will refer to them in a similar manner: Topo-A, Topo-B, Topo-C, and Topo-D. The two major differences in comparison to the previous experiment are as follows: 1) instead of using the same resource limit coefficient (was 150% of mean resource utilization in Section 10.5), each experiment configuration uses a different percentage of mean resource utilization from this set – $\{50\%, 100\%, 150\%, 200\%, 250\%\}$; 2) the experiment is configured to use only the **Predictive** autoscaling policy to keep the focus on the impact of resource limits on the efficiency of *predictive* autoscaling. We repeated the experiment for each resource limit 5 times.

Analysis of the results. Comparison of the fulfilled requests percentages across different resource allocation limits supports the hypothesis that the impact of resource limits on the efficiency of predictive autoscaling is substantial, although highly nonlinear (cf. Figure 10.19). For example, we observe that setting up the resource limits to 250% of the average resource utilization of services has an opposite effect on the percentage of fulfilled requests for Topo-A (cf. Figure 10.19a) and Topo-B (cf. Figure 10.19b). In the case of the application topology Topo-A, setting the resource limits to be $\times 2.5$ higher than the average resource utilization

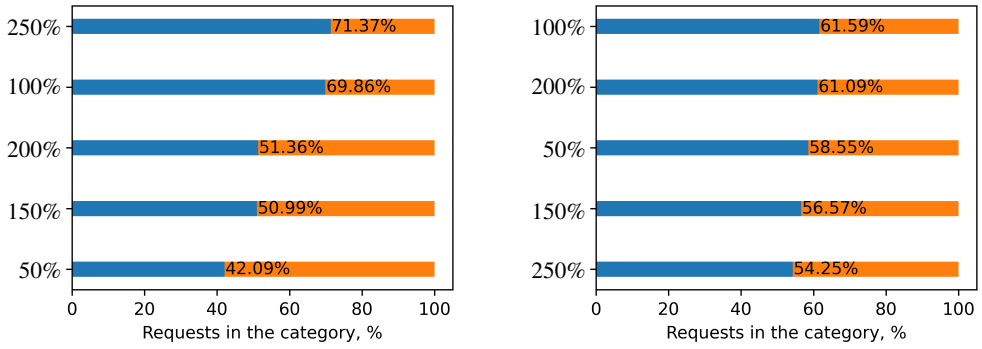
[11] `https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#requests-and-limits`

yields the highest fulfillment ratio of generated requests, whereas the same setting for Topo-B ends up at the very bottom in terms of fulfilled requests. Broadening our view to the topologies Topo-C and Topo-D yields similar results. For instance, the resource limit of 200% of the average resource utilization offers the best requests fulfillment percentage for the topology Topo-C, whereas the same resource limit ends up being next to worst for Topo-D for the same autoscaling quality metric. The reasons for such a behavior were discussed in the previous subsection.

Another interesting observation from Figure 10.19 is that the sensitivity of the requests fulfillment quality metric to the change in resource limits differs substantially across the topologies. Concretely, we observe that the given range of resource limits does not result in high variance of requests fulfillment ratio for the topologies Topo-B and Topo-D. In the case of Topo-B, the difference between the best and the worst resource limits is 7.34%, and for Topo-D it is even lower – 5.22%. These numbers are dwarfed by the 29.28% of difference demonstrated by Topo-A. Overall, the plot for Topo-A (cf. Figure 10.19a) clearly indicates that the efficiency of predictive autoscaling for the simulated application with the topology Topo-A is very sensitive even to the the small sample of the resource limits studied. Again, this supports our point about the relation between the application and virtual resources layers being a significant contributor to the quality of predictive autoscaling for multilayered cloud deployments. We omit the response time CDFs from the discussion since they do not provide any additional insights into the impact of resource limits on the efficiency of predictive autoscaling.

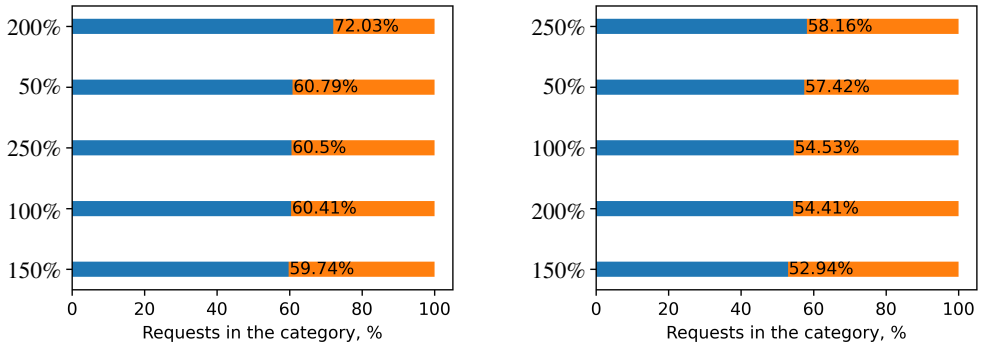
Figure 10.20 demonstrates that the impact of resource limits on the actual resource allocation during the autoscaling process is also far from being linear. This figure employs the cumulative deployment cost as an aggregate estimate of cloud resources (VMs) that were allocated during the experiment. First thing to notice in this plot is that there is no substantial difference in the cost across the resource limits. For the experiment, all the costs fall in the same interval 0.18 – 0.28 USD. This is especially clear for Topo-A, Topo-B, and Topo-D, whose cumulative deployment costs have almost no variance across the resource limits studied. Another important observation is the nonlinarity in the deployment costs for Topo-C. When increasing the resource limits from 50% to 100% of average resource utilization, the total cost drops a bit, but then it starts to surge at 150%, peaking at 200%. Surprisingly, at 250% of average resource utilization, the cost drops roughly to the level observed at 150%. Recalling that each experiment was conducted 5 times, we cannot attribute this result to any anomaly. In contrast, it represents the complex dynamics of the simulated system. A possible explanation is that the resource limits of 250% of average resource utilization resulted in allocating larger amount of resources available for processing the requests. This allowed to respond to the requests faster, and, in turn, the unneeded resources (expensive t3.large instances) were deallocated later on at a larger scale than for 200% resource limits. Apparently, 200% resource limits failed to offer the sufficient amount of resources to keep up with the request pace. All in all, this behavior resulted in a deployment with higher resource limits consuming *less* resources than that with lower resource limits. It is hardly possible to come up with a simple heuristic model that captures the diversity of behaviors induced by predictive autoscaling on multiple resource virtualization layers of cloud deployments.

Conclusion. This section demonstrated the impact of selecting the resource limits on the efficiency of predictive autoscaling for multilayered cloud deployments. The revealed nonlinearity in the behavior of predictive autoscaling when changing the way how the service instances are translated into the VMs, undermines the common intuition that the higher resource limits for services mean higher cost. Since the degree of this nonlinearity highly depends on the application and the load as well, the design of the predictive autoscaling policy should be tailored to the particular conditions that the autoscaling policy will be used in. Otherwise, efficiency of autoscaling might be quite unpredictable.



(a) Application topology Topo-A.

(b) Application topology Topo-B.



(c) Application topology Topo-C.

(d) Application topology Topo-D.

Figure 10.19.: Fulfilled vs failed requests distribution in the experiment to assess the impact of resource limits on the efficiency of predictive autoscaling for multilayered deployments for topologies depicted in Figure 10.13.

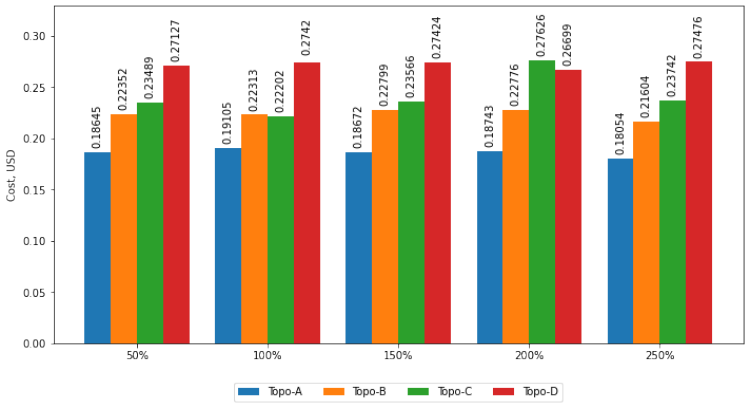


Figure 10.20.: Averaged cost of the deployment in the experiment to assess the impact of resource limits on the efficiency of predictive autoscaling for multilayered deployments.

11

Conclusion & Outlook

11.1 Conclusion

This thesis presented a systematic study of predictive autoscaling, a technology that aims to provision and decommission virtual resources ahead of time to meet the service level objectives and minimize the cost of deployment. This was achieved by splitting the autoscaling mechanism from the policy and independently investigating the most significant enablers of predictive autoscaling, viz, load forecasting and model-driven dynamic resource allocation.

This thesis contributes the architecture of generic predictive autoscaler (cf. Chapter 5) to the body of knowledge on autoscaling. In comparison to the existing works, this framework structures the most important processes of predictive autoscaling, s.a. forecasting, topology analysis, metrics mapping onto the quality of service, in a single feedback loop. Second, this thesis distills fundamental abstract entities that constitute the autoscaling process or otherwise relate to it, and their interrelations (cf. Section 9.1). These entities resulted from the work on **Multiverse** autoscaling simulator. Third, this thesis explores the topologies of microservice applications from Github and discovers the random graph model (BA-model) and its parameters that allow to synthesize applications with realistic topology for the purposes of simulation (cf. Chapter 8). Lastly, this thesis introduces the notions of timeliness and QoS-relevance for scaling actions and analyzes how do the building blocks of autoscaling policies allow to improve on them (cf. Chapter 10). By improving the QoS-relevance and timeliness of scaling actions, an autoscaling policy is able to improve the service level experienced by the end user, which was shown in the chapter.

This thesis advances the state of practice in predictive autoscaling in multiple ways. First, it showcases how the reactive autoscaling approach fails to provide an appropriate quality of service for the multilayered scaling for 3 large cloud services providers (cf. Chapter 3). The inherent limitations on timeliness of scaling actions in reactive approach do not allow it to cope well with the sharp load increases. The second practical contribution of the thesis is the systematic evaluation of different approaches to load forecasting on production load patterns (cf. Section 6). The forecasting serves the purpose of mitigating the timeliness limitations of the reactive autoscaling approach as was discussed in Chapter 10. The third practical contribution establishes an approach to automatic virtual resources allocation based on two steps: 1) establishing a mapping between the deployment metrics and the delivered quality of service, and 2) searching

for an optimal amount of virtual resources (service instances or resource shares) by solving an optimization problem with constraints expressed as SLOs (cf. Chapter 7). As the fourth practical contribution, this thesis offers an autoscaling simulation and experimentation toolbox comprising autoscaling simulator **Multiverse**, simulations results visualization tool **Stethoscope**, autoscaling simulation automation tool **Cruncher**, and data-driven simulations configurations generation tool **Praxiteles** (cf. Chapter 9). This toolbox allows to experiment with diverse tuning knobs of autoscaling policies and evaluate them on different resource consumption profiles and load patterns. Most importantly, the toolbox allows to train the models to be used in predictive autoscaling policies in advance on different load patterns and configurations to enable better generalization using another purpose-built tool **Training Ground**. The fifth practical contribution of the thesis is the sample predictive autoscaling policy that shows how the diverse tuning knobs might be set to get actionable results (cf. Appendix A).

In addition to these contributions, this thesis aimed to show how empirical techniques can be useful to derive theoretical knowledge about the studied systems. In that sense, this thesis departs from the conventional approach of producing a system that solves a concrete problem. With this work, we tried to expand both the theoretical understanding of the predictive autoscaling technology and to provide a set of tools that can help both the researchers and the practitioners to dive into the gory details of designing the predictive autoscaling policies and mechanisms. The diversity of applications, load patterns and platforms did not allow us to envisage a "one size-fits all" solution. Nevertheless, we humbly believe that the contributions of this thesis provide a foundation to proceed with designing custom predictive autoscaling policies to fit the needs of versatile applications.

11.2 Future Outlook

The following autoscaling research directions appear as the most promising for the years to come:

- **Automated design of autoscaling policies**. A great deal of effort is currently devoted to making the autoscaling policies "smart" by predicting suitable scaling actions. Unfortunately, this approach often fails to offer the *determinism* and *robustness*, as well as the *explainability* of the scaling results. We propose to utilize the same or other related techniques from the domain of artificial intelligence and deep learning to automate the design of autoscaling policies. The transfer of the stochasticity into the design process from the policy allows to synthesize deterministic autoscaling policies tailored for the specific application/load.
- **Workload-specific autoscaling policies**. Devising application-agnostic autoscaling policies deems to be a challenging theoretical exercise which tends to produce impractical results. Focus at a particular type of workload and the user load pattern, i.e. the specialization of autoscaling policies, can provide more useful scaling actions. With the tendency to compose applications of off-the-shelf building blocks such as databases and frameworks, one can bring this specialization a step further by introducing the *service-specific autoscaling policies*.
- **Autoscaling with novel resource abstractions and in novel deployment setting**. To a large extent, research in autoscaling and dynamic resource provisioning focuses on scaling VM clusters. Recently, the resource abstraction of container/pod paved its path into the autoscaling research. More recent abstractions such as functions (as in FaaS paradigm) are almost absent from the research literature. The existing body of research predominantly focuses at autoscaling in the centralized setting. However, with the decrease in the price and increase in the capabilities of embedded devices and single board computers, federated/distributed elastic applications start to emerge [120, 177].

- **Physical infrastructure-assisted autoscaling policies**. Autoscaling decisions in non CSP-assisted scenario are limited by the information provided on the layer of virtual clusters. The architecture of the physical machine hosting VM or a container, the number and the resource usage of other tenants, as well as the performance counters are unavailable. Disclosure of this information might significantly improve the scaling decisions taken regarding the application and the cluster of virtual nodes. For instance, it was demonstrated that the best quality of reactive autoscaling for multilayered deployments was achieved by combining the provider-managed autoscaling of Google Cloud with Kubernetes that is also maintained and to a large extend developed by Google [11]. One can speculate that such a fruitful cooperation becomes possible only when the physical infrastructures assists the autoscaling service.
- **Increasing the accuracy and reducing the runtime of autoscaling simulations**. Simulations of the autoscaling scenarios for large applications on the scale of Amazon and Google demand a significant amount of computational resources. Accurate simulations require to push the simulation step as low as 1-10 ms which increases simulation runtime by a large margin depending on the scale of the application. To increase the number of research works in the field as well as their practical value and the replicability, one needs to improve the design of the autoscaling simulators in two aspects – accuracy and performance. To a certain extent, compiled languages may assist with both.
- **Automatically generating realistic applications and infrastructure configurations**. A corollary to the previous point is that the simulated (or, in general, evaluated) workloads and virtual cluster configurations should be diversified. At the moment of writing, the set of benchmark microservice applications is relatively small and does not exhibit the complexity of the real-world applications that can incorporate as many as hundreds to thousands of microservices [129]. To overcome this challenge, an effort is required to generate artificial applications with realistic workload patterns and topologies. Realistic configurations of clusters of virtual nodes should provide an enormous value to the autoscaling policies research – currently, CSPs prefer not to disclose the physical hardware and the provisioning logic to anyone, which forces researchers to adopt unvalidated assumptions.
- **Multi-cost model & cross-cloud autoscaling**. The cloud services cost models other than the typical pay-per-use are relatively little considered in the autoscaling research literature. Other cost models are frequently considered in isolation, i.e. no combination of differently-billed cloud services is usually foreseen. Similarly, the cross-provider autoscaling is a relatively novel area. Although both of these aspects might deem to be of a negligible theoretical value, they are important in practice.
- **Autoscaling for changing applications**. Modern cloud applications relatively seldom stay unchanged for a long time. For instance, otto.de reaches 500 live deployments per week [178]. This means that the changes to the application logic are rather common, and when devising an autoscaling policy, one need to take into account that the application logic, its topology, and the resource usage patterns may change over time. In particular, such changes to the application challenge the "smart" autoscaling policies the most since they render the data-centric models fully or partially obsolete.
- **Comprehensive autoscaling evaluation techniques and metrics**. Autoscaling algorithms are evaluated either with the common performance metrics such as execution time, resource utilization or response time or with a set of autoscaling-specific metrics [179]. Although useful insights are usually collected in such way, still, there is a lack of methodology and models to evaluate the *autoscaling goal improvement provided in relation to the resources consumed by the autoscaling process*. The major challenge here arises from the lack of clarity of which processes should be attributed to the autoscaling.
- **User modeling**. A large body of work in predictive autoscaling is devoted to forecasting the user load [46, 176, 180]. Although this direction is proven to be useful to plan the resource allocations on

large time intervals, the representation of the user load as time series of requests per unit of time is too high-level for accurate capacity provisioning at the finer granularity. To become an appropriate choice for shorter intervals of time, predictive autoscalers might need to consider behavioral models for individuals or user segments [181, 182, 183, 184]. Integrating methods from the user modeling research into the user load prediction has a potential for improving the service level and making the availability and the response time a personalized experience.

Lastly, we would like to bring a point about *democratizing the access to the user load and configurations of real applications*. Companies with large data processing and storing infrastructure facilities, such as Google and Alibaba, aim to boost the innovation in dynamic resource management by offering the physical cluster traces publicly [185, 186, 24, 29]. However, these efforts are limited to the resource utilization of the data centers. What the research community currently lacks are the efforts of cloud-native application owners to disclose the user loads and application topologies/configurations/workload patterns. The advance of the application autoscaling research substantially relies on the access to such data. Although it is obviously complicated to provide public access to this information both from the point of trade secret and the organizational efforts, we would still like to emphasize that the advancement and the practical relevance of the autoscaling research heavily relies on the open access to the data of real applications. We hope that these data will be more available in the future.

Appendices

APPENDIX A

Example Predictive Autoscaling Policy

The components of an autoscaling policy were presented and discussed in Section 9.1. Here we focus at the combination and configuration of these components to build the **Predictive** autoscaling policy.

First, for any autoscaling policy we specify its `sync_period` and the `warm_up` time as shown in Listing A.1. The first parameter is responsible for how often is the *desired* state recalculated. In this example, the period is set to 30 s. Usually, this period is in range of seconds (e.g. Kubernetes sets it by default to 15 s) to avoid frequent changes to the application state but at the same time not to miss a major load change. For the purpose of experiments conducted in this chapter, the value of 30 s was appropriate. However, it should be reduced if the frequency of load oscillations gets higher or if the load increase/decrease rate gets higher.

The `warm_up` parameter allows to skip computing the desired state for the time period specified by it. This is relevant for cases when the initial deployment is not yet done, hence it does not make sense to scale the undeployed application. As the reader might recall, we've used a delayed load pattern in the experiments in the previous section to allow some time for the application deployment. The logic behind the `warm_up` parameter is the same.

```
"policy": {
    "sync_period": {
      "value": 30,
      "unit": "s"
    },
    "warm_up": {
      "value": 1,
      "unit": "m"
    }
}
```

Listing A.1: General settings of an autoscaling policy.

The details of autoscaling process are specified in the `application` section of the configuration file as shown in Listing A.2.

Since scaling is performed on the level of services, one has to specify the configuration of autoscaling for each service. For the **Predictive** policy, all the services had the same configuration of autoscaling policy. To avoid configuration duplication, we used the keyword `default` instead of a service name (cf. line 4 in Listing A.2). This will result in using the same autoscaling configuration for every service of an application.

Predictive policy implements *horizontal autoscaling* which is pointed out by the `scaled_aspect_name` set to `count` (cf. line 5 of Listing A.2). Scaling aspect is a quantifiable value that characterizes the capability of the deployed application to process the load, e.g. CPU shares or count of service instances. The considered policy uses `maxScale` scaling aggregation rule which first prompts every specified metric group to compute the desired service instances count according to its own data and configuration and then selects the maximal proposal. This is basically a type of parallel aggregation rule presented in Section 9.2 (cf. Figure 9.4b).

Predictive policy has only one metrics group which is specified in lines 9-35 of Listing A.2. Metric group is a collection of metrics that are used to calculate the desired state for performing the scaling actions. There are several group-level configurations that are set in lines 10-13. First, we set the limit on the count of service instances that this metric group is allowed to propose. The min limit is set to 1 (line 11) to avoid completely undeploying the service. The max limit is set to 100 (line 10) to allow more room for accommodating larger loads but without running into a risk of costly useless overprovisioning under concerted DDoS attack [126]. Entry `desired_aspect_value_calculator_conf` (line 12) configures how exactly is the scaling aspect (in our case, service instances count) is computed based on the metrics in this group. On the same level, there is a `stabilizer_conf` which sets how the computed scaling aspect is stabilized in time to avoid oscillations in the provisioned resources. The discussion of these two configuration pieces is postponed till later paragraphs.

```
1  "application": {
2      "services": [
3        {
4          "service_name": "default",
5          "scaled_aspect_name": "count",
6          "scaling_effect_aggregation_rule_name": "maxScale",
7          "metrics_groups": [
8                {
9                    "name": "group1",
10                   "initial_max_limit": 100,
11                   "initial_min_limit": 1,
12                   "desired_aspect_value_calculator_conf": {...},
13                   "stabilizer_conf": {...},
14                   "metrics": [
15                       {
16                         "metric_source_name": "Load",
17                         "metric_name": "load",
18                         "submetric_name": "*",
19                         "metric_params": {
20                           "sampling_interval": {
21                             "value": 1,
22                             "unit": "s"
23                           }
24                         },
25                         "values_aggregator_conf": {...},
26                         "forecaster_conf": {...}
```

```
27                      },
28                      {
29                        "metric_source_name": "Service",
30                        "metric_name": "memory",
31                        "values_aggregator_conf": {...},
32                        "forecaster_conf": {...}
33                      }
34                  ],
35                  "default_metric_config" : { "values_filter_conf": {...} }
36              }
37          ]
38        }
39      ]
40    }
```

Listing A.2: High-level settings of metric groups to scale application services.

Section `metrics` (lines 14-34) lists all the metrics that are used for computing the desired scaling aspect in this metric group. **Predictive** policy uses load (lines 16-27) and memory utilization (lines 28-33) for that. For both metric we specify their *source*, i.e. the component that is responsible for collecting metric values (defined in the **Multiverse** simulator). For load, it is the **Load model** (denoted as `Load`), and for memory it is **Service** (denoted as `Service`). In the latter case, we use capitalized keyword `Service` to refer to the current service that the metric group is associated with. Otherwise, one can provide an explicit service identifier if it is desirable to consider the metric of another service to take the scaling decision (in case of some topology-aware policies). For some metrics, one can also specify the sub-metrics which is shown in line 18. Here, we specify that all the load should be considered when scaling, regardless of region. For load, it is also important to specify its `sampling_interval` (cf. lines 20-23 in Listing A.2). In the considered policy, we sample the load *per second*, i.e. getting requests per second value. In general, metric-specific configurations for collection/representation appear in the `metric_params` section (cf. line 19).

Each metric gets its own filter, aggregator, and forecaster. In case of considered **Predictive** policy, the filter configuration is the same for both metrics, hence it appears in the dedicated `default_metric_config` section (line 35). Both the aggregator and the forecaster configuration could also go there, but these components required customization for the **Predictive** policy, hence they ended up in the entries of corresponding metrics under keys `values_aggregator_conf` and `forecaster_conf` (lines 25, 31 and lines 26, 32 correspondingly). The discussion of concrete configurations for these building blocks of **Predictive** policy follows in the next paragraphs.

Upon collection of individual metrics, they first need to be filtered. Both metrics of **Predictive** policy share the filtering configuration which is shown in Listing A.3. This `defaultNA` filter simply substitutes every NA value for 0. When the metric is filtered, we proceed to aggregation according to the policy.

```
1 "values_filter_conf": {
2     "name": "defaultNA",
3     "config": { "default_value": 0 }
4 }
```

Listing A.3: Configuration of the filter component for both metrics of the single metric group of **Predictive** autoscaling policy.

As was mentioned earlier, load metric and memory metric have own aggregators. The configuration of the aggregator for load metric is shown in Listing A.4. This `sumAggregator` aggregator calculates the sum of the requests produced in the interval of time windowed according to the specified resolution (1 minute in our case). Similarly, the `medianAggregator` for memory utilization metric (cf. Listing A.5) uses a time window of 1 minute, but instead of summing all the observation in this interval, it takes the median of these observations.

```
"values_aggregator_conf": {
   "name": "sumAggregator",
   "config": { "resolution": { "value": 1, "unit": "m" } }
}
```

Listing A.4: Configuration of the aggregator component for the load metric of **Predictive** autoscaling policy.

```
"values_aggregator_conf": {
    "name": "medianAggregator",
    "config": { "resolution": { "value": 1, "unit": "m" } }
}
```

Listing A.5: Configuration of the aggregator component for the memory metric of **Predictive** autoscaling policy.

Similarly to aggregators, each metric has its own forecaster. In case of memory utilization metric it does not implement actual forecasting but rather uses the last value available for this metric as in reactive autoscaling (cf. Listing A.6). The load metric uses pre-trained Holt-Winters forecasting model under path specified in the parameter `dir_with_pretrained_models` in Listing A.7. The folder contains multiple models, each for the unique combination of service name, region, and metric. This structuring allows high flexibility in case the forecasting model has to take into account some service/region specifics to devise more accurate forecasts. Lastly, both forecaster configurations share two quantitative parameters, namely, `forecast_frequency` and `horizon_in_steps`. The first one specifies the resolution at which the forecasts are provided. The second one specifies how far into the future does the forecaster extrapolate the metric into in terms of the frequency unit. In both considered listings, the forecast is 2 minutes into the future with resolution of 1 minute, which means 2 forecasted values. This, however, is not entirely accurate for the design of the **Multiverse** simulator. The simulator forecasts double the `horizon_in_steps` values. These are then shifted closer to the present moment in time, by one step (equals to 1 minute for **Predictive** policy). This is done to force the desired values to appear *in advance*. Ideally, the size of this shift should roughly correspond to the booting/termination times of virtual entities (VMs/containers).

```
"forecaster_conf": {
    "name": "reactive",
    "forecast_frequency": { "value": 1, "unit": "m" },
    "horizon_in_steps": 2
}
```

Listing A.6: Configuration of the forecasting component for the memory metric of **Predictive** autoscaling policy.

```
"forecaster_conf": {
    "name": "holt_winters",
```

```
    "dir_with_pretrained_models": "/home/ubuntu/autoscaling-simulator/trained_models/
        forecasting/holtwinters",
    "do_not_adjust_model": true,
    "forecast_frequency": { "value": 1, "unit": "m" },
    "horizon_in_steps": 2
}
```

Listing A.7: Configuration of the *pre-trained* forecasting component for the load metric of **Predictive** autoscaling policy.

To obtain a trained forecasting model for the purpose of experiments in the previous sections, we modified the scaling policy to allow the forecaster to be dynamically fitted based on the incoming load. The configuration for it is shown in Listing A.8. In contrast to previous excerpts of forecaster configurations, the `config` section in this listing contains the parameters of the Holt-Winters model. One can see that we specify an additive seasonal component with a period of 2 (minutes) for the Holt-Winters model. Other parameters set up smoothing for the level and seasonal component. In this setting, the seasonal component is sharp (1.0) which is to better relate to the generated oscillating load pattern thus both improving the response time and reducing the deployment costs. Lastly, the parameters `history_data_buffer_size` and `fit_model_when_history_amount_reached` control the maximal and minimal size of the data used to fit the model. Here, the buffer can store only last 10 minutes of observations, and the model can first be fit only when the amount of buffered observations reaches 2 minutes.

```
"forecaster_conf": {
    "name": "holt_winters",
    "config": {
        "seasonal": "add",
        "smoothing_seasonal": 1.0,
        "seasonal_periods": 2,
        "smoothing_level": 0.7
    },
    "dir_to_store_models": "/home/ubuntu/autoscaling-simulator/trained_models/forecasting
        /holtwinters",
    "forecast_frequency": { "value": 1, "unit": "m" },
    "history_data_buffer_size": { "value": 10, "unit": "m" },
    "fit_model_when_history_amount_reached": { "value": 2, "unit": "m" },
    "horizon_in_steps": 2
}
```

Listing A.8: Configuration of the forecasting component to train for the load metric of **Predictive** autoscaling policy.

In the considered autoscaling policy, the desired scaling aspect value calculator is responsible for calculating the desired service instances count thus implementing the horizontal autoscaling. The calculator does so by combining the learned "metrics-to-response time" mapping model with the optimizer that attempts to find such service instances count that the response time stays under the desired SLO. This approach was discussed at length in Section 7, here we focus on particular settings that are shown in listings A.9 and A.10.

Listing A.9 presents the configuration of pre-trained ML-based desired scaling aspect value calculator. It has its category explicitly specified in line 2. Other currently supported option is the calculator in the `rule` category. Such calculator implements a deterministic procedure to compute the desired scaling aspect value.

An example of rule-based calculator's configuration is present in the same listing as a *fallback calculator* (cf. lines 4-13). The fallback calculator is used when the value predicted with the learning-based model is off from the observed by some predefined factor, in our case it is 25% (cf. lines 28-31 in Listing A.9). Hence, if for **Predictive** policy the predicted response time falls out of this 75%-125% band, then the desired count of instances of *i*th service is computed as a ratio of the current value of observed memory utilization metric to the target value specified as 0.8 in line 8 of the listing with added 15% as per adjustment heuristic in lines 9-12. The following formula summarizes the work of the fallback calculator: $cntInstances_i = 1.15 \cdot \lceil \frac{U_{mem}}{0.8} \rceil$. If the requirement on the accuracy of prediction is satisfied, the scaling aspect value calculator will use the model that is specified in the configuration in section `model` (lines 15-20).

In case of **Predictive** policy, we use pre-trained neural networks. This is pointed at by the parameters specifying the folder where the trained models reside (`model_root_folder`) and the name of particular model that is used for mapping metrics onto the response time (`model_file_name`). We use pre-trained neural networks since it was proven to be impossible to reach an acceptable accuracy in scope of short simulation runs that were performed (15-30 simulated minutes).

Although there appears to be just a single model in the scaling policy, in reality, the `model` section represents a set of trained neural networks that differ by service, region, and metric group. Each such model resides in a separate folder. This has to do with how Keras stores the models on disk. The input to the model is specified by metrics constituting the metric group. The output of the model is specified in the `performance_metric` section (lines 21-27). For the considered example of **Predictive** autoscaling policy, we select the response time as an output metric, i.e. the model predicts the response time based on the forecasted requests count and the current memory utilization. The most interesting part that is specified in `performance_metric` config part is the `threshold` parameter (line 26). This parameter is set to 100 ms and is used to solve the constrained optimization problem to find the service instances count that are needed to meet this goal. Since the critical path in the simulated application is anyway longer than 100 ms in time equivalent, one can safely state that the considered policy aims to find the service instances count that simply minimizes the predicted response time.

Since we've discussed the configuration that uses the pre-trained model, let us now switch to the example of scaling policy that was used to train this model. A relevant excerpt of this policy is shown in Listing A.10. As one can see, Listing A.10 shows the configuration section under the same `model` key, but with the contents substituted for the specific configuration of the model. The model contains 6 layers with the first layer having 50 units (next to input), the second – 30, the third – 25, the fourth – 15, the fifth – 5, and the sixth – 1. The configuration of the network was devised as a result of preliminary runs. We intended to keep it small but at the same time to allow it to learn multiple different behaviors by training it on simulations with different load patterns. It was observed that smaller number of layers barely copes with that task. To increase generalization capability of the model, we apply a dropout rate of 0.2 for units in all layers. The loss function is set to `mean_squared_error` and the optimizer is set to ADAM, both of which are the go-to defaults in the ML community.

A purpose-built tool, called **Training ground**, used this scaling policy to train the model on a selected sample of load patterns. The tool ran the randomized simulations and collected the observations every 3 seconds. Then, once a batch of 10 observations was filled, the tool used these observations to perform a training step. This means that a training step corresponds to incorporating another 30 seconds of observations into the model trained. Figure A.1 shows how the scaled error model accuracy metric improves over time for all 7 services constituting the synthesized application. Turns out that roughly 450 minutes worth of simulations data ($900 \times 30 \times \frac{1}{60}$) are sufficient to adequately train the designed 6-layer neural network for all the services.

```
1 "desired_aspect_value_calculator_conf": {
```

```
    "category": "learning",
    "config": {
        "fallback_calculator": {
            "category": "rule",
            "config": {
                "name": "ratio",
                "target": { "metric_name": "memory", "value": 0.8 },
                "adjustment_heuristic_conf": {
                    "name": "rescale",
                    "scaling_factor": 1.15
                }
            }
        },
        "model": {
            "name": "neural_net",
            "kind": "online",
            "model_root_folder": "/home/ubuntu/autoscaling-simulator/nn",
            "model_file_name": "dav_model.mdl"
        },
        "performance_metric": {
            "metric_source_name": "response_stats",
            "metric_name": "response_time",
            "submetric_name": "*",
            "metric_type": "duration",
            "threshold": { "value": 100, "unit": "ms" }
        },
        "model_quality_metric": {
            "name": "scaled_error",
            "threshold": 0.25
        },
        "minibatch_size": 2,
        "optimizer_config": {
            "method": "trust-constr",
            "jac": "2-point",
            "hess": "SR1",
            "verbose": 0,
            "maxiter": 100,
            "xtol": 0.1,
            "initial_tr_radius": 10
        }
    }
}
```

Listing A.9: Configuration of the pre-trained desired state calculator component of **Predictive** autoscaling policy.

```
"model": {
    "name": "neural_net",
```

```
    "layers": [
        { "type": "Dense", "units": 50 },
        { "type": "Dropout", "rate": 0.2 },
        { "type": "Dense", "units": 30 },
        { "type": "Dropout", "rate": 0.2 },
        { "type": "Dense", "units": 25 },
        { "type": "Dropout", "rate": 0.2 },
        { "type": "Dense", "units": 15 },
        { "type": "Dropout", "rate": 0.2 },
        { "type": "Dense", "units": 5 },
        { "type": "Dropout", "rate": 0.2 },
        { "type": "Dense", "units": 1 }
    ],
    "model_params": {
        "learning": {
            "loss": "mean_squared_error",
            "optimizer": "adam"
        }
   }
}
```

Listing A.10: Neural network configuration for training the model of desired service instance count calculator of **Predictive** autoscaling policy.

When the desired service instances count was calculated for the future interval, **Predictive** autoscaling policy attempts to stabilize this count to avoid frequent changes that are proven to work poorly for minor spontaneous changes in load [179]. The stabilization happens by slicing the forecasting interval into 2 minute windows and taking the maximal value present in this interval (cf. Listing A.11). This value is then provided at the very beginning of the interval such that this count of service instances could be used to adjust the virtual infrastructure in advance.

```
"stabilizer_conf": {
    "name": "maxStabilizer",
    "config": { "resolution": { "value": 2, "unit": "m" } }
}
```

Listing A.11: Configuration of the stabilizing component for the computed service instances count of **Predictive** autoscaling policy.

After the desired stabilized service instances count is received for each metric group, the maximal proposal gets selected as was explained earlier. Since **Predictive** policy only uses one group, its output is effectively used as the resulting desired services instances count proposal. Next, this proposal will be fed into the virtual infrastructure adjuster which will attempt to minimize the provisioning of virtual machines by trying to batch service instances on the same VM. In production setting, this piece of behavior would correspond to the Cluster Autoscaler (CA) of Kubernetes or to the IaaS autoscaling solutions of cloud services providers. We do not discuss the adjustment here further since it does not exhibit any advanced predictive characteristics but simply optimizes for the given goal in deterministic manner.

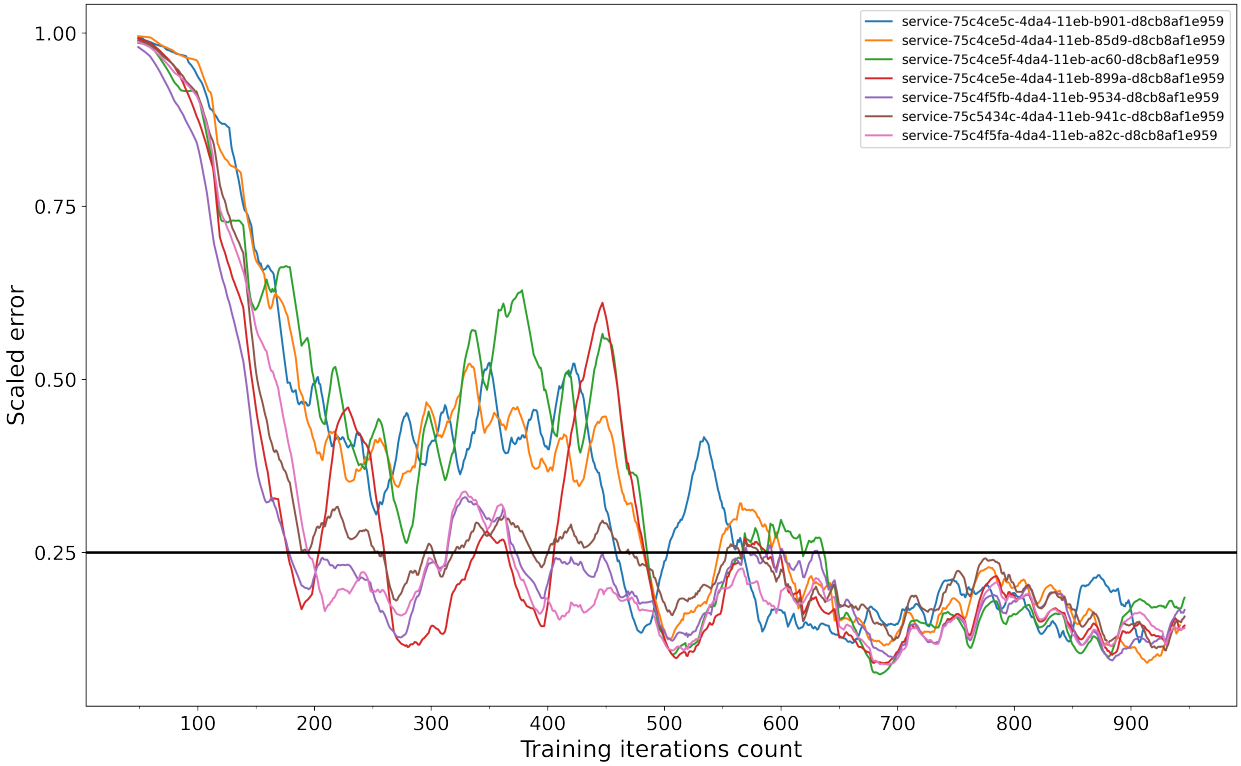


Figure A.1.: Improvement in the scaled error quality metric for neural network used for **Predictive** autoscaling policy with iterations. Each iteration correspond to roughly 30 simulations seconds worth of data added. Presented curves are averaged in window of 50 iterations to show the trend clearer. The horizontal line corresponds to the threshold value of the scaled error quality metric specified in Listing A.9.

APPENDIX B

Availability

The source code of the simulation toolbox (cf. Chapter 9) is publicly available via the following link: `https://github.com/Remit/autoscaling-simulator`.

The source code used for the evaluation of different forecasting methods (cf. Chapter 6) is publicly available via the following link: `https://github.com/Remit/ForecastingMethodsEvaluation`.

The data set used for the topology analysis in Chapter 8 is publicly available via the following link: `https://zenodo.org/record/3573846`.

The source code of ScaleX tool used to evaluate the performance of production autoscaling solutions for multilayered deployments (cf. Chapter 3) is publicly available via the following link: `https://github.com/ansjin/ScaleX`.

In case of problems running the code, please, do not hesitate to contact me via **v.e.podolskiy@gmail.com**.

APPENDIX C

List of Authored and Co-authored Publications on the Thesis Topic

Journal publications:

- Vladimir Podolskiy, Anshul Jindal, Michael Gerndt. "Multilayered Autoscaling Performance Evaluation: Can Virtual Machines and Containers Co-Scale?" Int. Journal of Applied Mathematics and Computer Science (AMCS), 2019, Vol. 29, No. 2, pp. 227-244.

Conference publications:

- Vladimir Podolskiy, Maria Patrou, Panos Patros, Michael Gerndt, and Kenneth B. Kent. 2020. The weakest link: revealing and modeling the architectural patterns of microservice applications. In Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering (CASCON '20). IBM Corp., USA, 113–122.
- H. Dickel, V. Podolskiy and M. Gerndt, "Evaluation of Autoscaling Metrics for (stateful) IoT Gateways," 2019 IEEE 12th Conference on Service-Oriented Computing and Applications (SOCA), 2019, pp. 17-24.
- Y. M. Ramirez, V. Podolskiy and M. Gerndt, "Capacity-Driven Scaling Schedules Derivation for Coordinated Elasticity of Containers and Virtual Machines," 2019 IEEE International Conference on Autonomic Computing (ICAC), 2019, pp. 177-186.
- V. Podolskiy, M. Mayo, A. Koay, M. Gerndt and P. Patros, "Maintaining SLOs of Cloud-Native Applications Via Self-Adaptive Resource Sharing," 2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), 2019, pp. 72-81.
- Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. 2019. Performance Modeling for Cloud Microservice Applications. In Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE '19). Association for Computing Machinery, New York, NY, USA, 25–32.
- V. Podolskiy, A. Jindal, M. Gerndt and Y. Oleynik, "Forecasting Models for Self-Adaptive Cloud Applications: A Comparative Study," 2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), 2018, pp. 40-49.

- V. Podolskiy, A. Jindal and M. Gerndt, "IaaS Reactive Autoscaling Performance Challenges," 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), 2018, pp. 954-957.
- A. Jindal, V. Podolskiy and M. Gerndt, "Multilayered Cloud Applications Autoscaling Performance Estimation," 2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2), 2017, pp. 24-31.

Workshop publications:

- Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. 2018. Autoscaling Performance Measurement Tool. In Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18). Association for Computing Machinery, New York, NY, USA, 91–92.
- Vladimir Podolskiy, Hans Michael Gerndt, and Shajulin Benedict. 2017. QoS-based Cloud Application Management: Approach and Architecture. In Proceedings of the 4th Workshop on CrossCloud Infrastructures & Platforms (Crosscloud'17). Association for Computing Machinery, New York, NY, USA, Article 7, 1–2.

Bibliography

[1] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, San Jose, CA, June 2013. USENIX Association.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[3] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, page 181–186, New York, NY, USA, 1991. Association for Computing Machinery.

[4] George A. Miller. Psychology and information. *American Documentation*, 19(3):286–289, 1968.

[5] Tobias Hoßfeld, Raimund Schatz, Ernst Biersack, and Louis Plissonneau. *Internet Video Delivery in YouTube: From Traffic Measurements to Quality of Experience*, pages 264–301. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[6] J. Martin, Yunhui Fu, N. Wourms, and T. Shaw. Characterizing netflix bandwidth consumption. In *2013 IEEE 10th Consumer Communications and Networking Conference (CCNC)*, pages 230–235, 2013.

[7] M. Manzano, J. A. Hernández, M. Urueña, and E. Calle. An empirical study of cloud gaming. In *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–2, 2012.

[8] I. Slivar, M. Suznjevic, L. Skorin-Kapov, and M. Matijasevic. Empirical qoe study of in-home streaming of online games. In *2014 13th Annual Workshop on Network and Systems Support for Games*, pages 1–6, 2014.

[9] Y. Xu, C. Yu, J. Li, and Y. Liu. Video telephony for end-consumers: Measurement study of google+, ichat, and skype. *IEEE/ACM Transactions on Networking*, 22(3):826–839, 2014.

[10] X. Zhang, Y. Xu, H. Hu, Y. Liu, Z. Guo, and Y. Wang. Modeling and analysis of skype video calls: Rate control and video quality. *IEEE Transactions on Multimedia*, 15(6):1446–1457, 2013.

[11] Vladimir Podolskiy, Anshul Jindal, and Michael Gerndt. Multilayered autoscaling performance evaluation: Can virtual machines and containers co–scale? *International Journal of Applied Mathematics and Computer Science*, 29(2):227 – 244, 01 Jun. 2019.

[12] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *2012 IEEE Network Operations and Management Symposium*, pages 204–212, April 2012.

[13] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, Cornel Barna, and Gabriel Iszlai. Optimal autoscaling in a iaas cloud. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, page 173–178, New York, NY, USA, 2012. Association for Computing Machinery.

[14] Ahmad Al-Shishtawy and Vladimir Vlassov. Elastman: Elasticity manager for elastic key-value stores in the cloud. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[15] Ying Liu, Daharewa Gureya, Ahmad Al-Shishtawy, and Vladimir Vlassov. Onlineelastman: self-trained proactive elasticity manager for cloud-based storage services. *Cluster Computing*, 20(3):1977–1994, Sep 2017.

[16] Federico Lombardi, Andrea Muti, Leonardo Aniello, Roberto Baldoni, Silvia Bonomi, and Leonardo Querzoni. Pascal: An architecture for proactive auto-scaling of distributed services. *Future Generation Computer Systems*, 98:342 – 361, 2019.

[17] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. Optimal operator deployment and replication for elastic distributed data stream processing. *Concurrency and Computation: Practice and Experience*, 30(9):e4334, 2018. e4334 cpe.4334.

[18] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[20] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[21] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the virtual library operating system: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework? *Queue*, 11(11):30–44, December 2013.

[22] Kartik Gopalan, Rohit Kugve, Hardik Bagdi, Yaohui Hu, Daniel Williams, and Nilton Bila. Multi-hypervisor virtual machines: Enabling an ecosystem of hypervisor-level services. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 235–249, Santa Clara, CA, July 2017. USENIX Association.

[23] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, 2010. USENIX Association.

[24] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[25] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 261–276, New York, NY, USA, 2009. Association for Computing Machinery.

[26] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 295–308, USA, 2011. USENIX Association.

[27] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 485–497, Santa Clara, CA, July 2015. USENIX Association.

[28] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.

[29] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, New York, NY, USA, 2017. Association for Computing Machinery.

[31] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[32] *Bin-Packing*, pages 426–441. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[33] György Dósa. The tight bound of first fit decreasing bin-packing algorithm is ffd(i) $\leq$ 11/9opt(i) + 6/9. In Bo Chen, Mike Paterson, and Guochuan Zhang, editors, *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 1–11, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[34] A. Jindal, V. Podolskiy, and M. Gerndt. Multilayered cloud applications autoscaling performance estimation. In *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*, pages 24–31, 2017.

[35] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Autoscaling performance measurement tool. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 91–92, New York, NY, USA, 2018. Association for Computing Machinery.

[36] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[37] Marin Litoiu, Mircea Mihaescu, Dan Ionescu, and Bogdan Solomon. Scalable adaptive web services. In *Proceedings of the 2nd International Workshop on Systems Development in SOA Environments*, SDSOA '08, page 47–52, New York, NY, USA, 2008. Association for Computing Machinery.

[38] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai. Exploring alternative approaches to implement an elasticity policy. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 716–723, 2011.

[39] P. Östberg, H. Groenda, S. Wesner, J. Byrne, D. S. Nikolopoulos, C. Sheridan, J. Krzywda, A. Ali-Eldin, J. Tordsson, E. Elmroth, C. Stier, K. Krogmann, J. Domaschka, C. B. Hauser, P. J. Byrne, S. Svorobej, B. Mccollum, Z. Papazachos, D. Whigham, S. Rüth, and D. Paurevic. The cactos vision of context-aware cloud topology optimization and simulation. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 26–31, 2014.

[40] Marin Litoiu, Murray Woodside, Johnny Wong, Joanna Ng, and Gabriel Iszlai. A business driven cloud optimization architecture. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, page 380–385, New York, NY, USA, 2010. Association for Computing Machinery.

[41] Ana Juan Ferrer, Francisco Hernández, Johan Tordsson, Erik Elmroth, Ahmed Ali-Eldin, Csilla Zsigri, Raül Sirvent, Jordi Guitart, Rosa M. Badia, Karim Djemame, Wolfgang Ziegler, Theo Dimitrakos, Srijith K. Nair, George Kousiouris, Kleopatra Konstanteli, Theodora Varvarigou, Benoit Hudzia, Alexander Kipp, Stefan Wesner, Marcelo Corrales, Nikolaus Forgó, Tabassum Sharif, and Craig Sheridan. Optimis: A holistic approach to cloud service provisioning. *Future Generation Computer Systems*, 28(1):66 – 77, 2012.

[42] Erik Elmroth, Johan Tordsson, Francisco Hernández, Ahmed Ali-Eldin, Petter Svärd, Mina Sedaghat, and Wubin Li. Self-management challenges for multi-cloud architectures. In Witold Abramowicz, Ignacio M. Llorente, Mike Surridge, Andrea Zisman, and Julien Vayssière, editors, *Towards a Service-Based Internet*, pages 38–49, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[43] *OPTIMAL CONTROL OF DISCRETE-TIME SYSTEMS*, chapter 2, pages 19–109. John Wiley & Sons, Ltd, 2012.

[44] *OPTIMAL CONTROL OF CONTINUOUS-TIME SYSTEMS*, chapter 3, pages 110–176. John Wiley & Sons, Ltd, 2012.

[45] A. Bauer, N. Herbst, S. Spinner, A. Ali-eldin, and S. Kounev. Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2018.

[46] Marwin Züfle, André Bauer, Nikolas Herbst, Valentin Curtef, and Samuel Kounev. Telescope: a hybrid forecast method for univariate time series. In *International work-conference on Time Series (ITISE 2017)*, 2017.

[47] M. Zuefle, A. Bauer, V. Lesch, C. Krupitzer, N. Herbst, S. Kounev, and V. Curtef. Autonomic forecasting method selection: Examination and ways ahead. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 167–176, 2019.

[48] Alexey Ilyushkin, André Bauer, Alessandro V. Papadopoulos, Ewa Deelman, and Alexandru Iosup. Performance-feedback autoscaling with budget constraints for cloud-based workloads of workflows, 2019.

[49] C. Mera-Gómez, F. Ramírez, R. Bahsoon, and R. Buyya. A multi-agent elasticity management based on multi-tenant debt exchanges. In *2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 30–39, Sep. 2018.

[50] Carlos Mera-Gómez, Francisco Ramírez, Rami Bahsoon, and Rajkumar Buyya. A debt-aware learning approach for resource adaptations in cloud elasticity management. In Michael Maximilien, Antonio Vallecillo, Jianmin Wang, and Marc Oriol, editors, *Service-Oriented Computing*, pages 367–382, Cham, 2017. Springer International Publishing.

[51] Carlos Mera-Gómez, Rami Bahsoon, Rajkumar Buyya, and Escuela Superior Politécnica. Elasticity debt: A debt-aware approach to reason about elasticity decisions in the cloud. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, UCC '16, page 79–88, New York, NY, USA, 2016. Association for Computing Machinery.

[52] J. F. Pérez, R. Birke, M. Björkqvist, and L. Y. Chen. Dual scaling vms and queries: Cost-effective latency curtailment. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 988–998, 2017.

[53] Yazhou Hu, Bo Deng, and Fuyang Peng. Autoscaling prediction models for cloud resource provisioning. In *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, pages 1364–1369, Oct 2016.

[54] Laura R. Moore, Kathryn Bean, and Tariq Ellahi. A coordinated reactive and predictive approach to cloud elasticity. 2013.

[55] A. Biswas, S. Majumdar, B. Nandy, and A. El-Haraki. An auto-scaling framework for controlling enterprise resources on clouds. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 971–980, 2015.

[56] A. Biswas, S. Majumdar, B. Nandy, and A. El-Haraki. Predictive auto-scaling techniques for clouds subjected to requests with service level agreements. In *2015 IEEE World Congress on Services*, pages 311–318, 2015.

[57] Anshuman Biswas, Shikharesh Majumdar, Biswajit Nandy, and Ali El-Haraki. A hybrid auto-scaling technique for clouds processing applications with service level agreements. *Journal of Cloud Computing*, 6(1):29, Dec 2017.

[58] M. Woodside, Tao Zheng, and M. Litoiu. Service system resource management based on a tracked layered performance model. In *2006 IEEE International Conference on Autonomic Computing*, pages 175–184, 2006.

[59] Ahmed Ali-Eldin, Maria Kihl, Johan Tordsson, and Erik Elmroth. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing*, ScienceCloud '12, page 31–40, New York, NY, USA, 2012. Association for Computing Machinery.

[60] A. Alieldin, Johan Tordsson, E. Elmroth, and M. Kihl. Workload classification for efficient auto-scaling of cloud resources. 2013.

[61] Zhicheng Cai, Duan Liu, Yifei Lu, and Rajkumar Buyya. Unequal-interval based loosely coupled control method for auto-scaling heterogeneous cloud resources for web applications. *Concurrency and Computation: Practice and Experience*, n/a(n/a):e5926.

[62] E. B. Lakew, C. Klein, F. Hernandez-Rodriguez, and E. Elmroth. Towards faster response time models for vertical elasticity. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 560–565, 2014.

[63] Soodeh Farokhi, Pooyan Jamshidi, Ewnetu Bayuh Lakew, Ivona Brandic, and Erik Elmroth. A hybrid cloud controller for vertical memory elasticity: A control-theoretic approach. *Future Generation Computer Systems*, 65:57 – 72, 2016. Special Issue on Big Data in the Cloud.

[64] S. Farokhi, E. B. Lakew, C. Klein, I. Brandic, and E. Elmroth. Coordinating cpu and memory elasticity controllers to meet service response time constraints. In *2015 International Conference on Cloud and Autonomic Computing*, pages 69–80, 2015.

[65] E. B. Lakew, A. V. Papadopoulos, M. Maggio, C. Klein, and E. Elmroth. Kpi-agnostic control for fine-grained vertical elasticity. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 589–598, May 2017.

[66] Mina Sedaghat, Francisco Hernandez-Rodriguez, and Erik Elmroth. A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, New York, NY, USA, 2013. Association for Computing Machinery.

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

[68] Adel Nadjaran Toosi, Jungmin Son, Qinghua Chi, and Rajkumar Buyya. Elasticsfc: Auto-scaling techniques for elastic service function chaining in network functions virtualization-based clouds. *Journal of Systems and Software*, 152:108 – 119, 2019.

[69] S. Heidari and R. Buyya. A cost-efficient auto-scaling algorithm for large-scale graph processing in cloud environments with heterogeneous resources. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

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

[71] A. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, and S. Kounev. Chamulteon: Coordinated auto-scaling of micro-services. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 2015–2025, 2019.

[72] Cornel Barna, Mark Shtern, Mike Smit, Hamoun Ghanbari, and Marin Litoiu. Model-driven elasticity and dos attack mitigation in cloud environments. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 13–24, Philadelphia, PA, June 2014. USENIX Association.

[73] A. U. Gias, G. Casale, and M. Woodside. Atom: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1994–2004, 2019.

[74] N. Beigi-Mohammadi, M. Shtern, and M. Litoiu. Adaptive load management of web applications on software defined infrastructure. *IEEE Transactions on Network and Service Management*, 17(1):488–502, 2020.

[75] Z. Qiu, J. F. Pérez, R. Birke, L. Chen, and P. G. Harrison. Cutting latency tail: Analyzing and validating replication without canceling. *IEEE Transactions on Parallel and Distributed Systems*, 28(11):3128–3141, 2017.

[76] T. Ye, X. Guangtao, Q. Shiyou, and L. Minglu. An auto-scaling framework for containerized elastic applications. In *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*, volume 00, pages 422–430, Aug. 2018.

[77] Leonardo Aniello, Silvia Bonomi, Federico Lombardi, Alessandro Zelli, and Roberto Baldoni. An architecture for automatic scaling of replicated services. In Guevara Noubir and Michel Raynal, editors, *Networked Systems*, pages 122–137, Cham, 2014. Springer International Publishing.

[78] I. Gergin, B. Simmons, and M. Litoiu. A decentralized autonomic architecture for performance control in the cloud. In *2014 IEEE International Conference on Cloud Engineering*, pages 574–579, 2014.

[79] Jim (Zhanwen) Li, John Chinneck, Murray Woodside, and Marin Litoiu. Fast scalable optimization to configure service systems having cost and quality of service constraints. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, page 159–168, New York, NY, USA, 2009. Association for Computing Machinery.

[80] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. Resource deflation: A new approach for transient resource reclamation. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[81] N. Beigi-Mohammadi, H. Khazaei, M. Shtern, C. Barna, and M. Litoiu. Adaptive service management for cloud applications using overlay networks. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 386–392, 2017.

[82] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar 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.

[83] R. Birke, J. F. Pérez, Z. Qiu, M. Björkqvist, and L. Y. Chen. Power of redundancy: Designing partial replication for multi-tier applications. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017.

[84] R. Birke, J. F. Perez, Z. Qiu, M. Borkqvist, and L. Y. Chen. spare: Partial replication for multi-tier applications in the cloud. *IEEE Transactions on Services Computing*, pages 1–1, 2017.

[85] Floriment Klinaku, Markus Frank, and Steffen Becker. Caus: An elasticity controller for a containerized microservice. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 93–98, New York, NY, USA, 2018. ACM.

[86] Y. Liu, N. Rameshan, E. Monte, V. Vlassov, and L. Navarro. Prorenata: Proactive and reactive tuning to scale a distributed storage system. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 453–464, 2015.

[87] M. Amir Moulavi, Ahmad Al-Shishtawy, and Vladimir Vlassov. State-space feedback control for elastic distributed storage in a cloud environment. In *ICAS 2012 : The Eighth International Conference on Autonomic and Autonomous Systems*, pages 589–596, 2012. QC 20130524QC 20151216.

[88] A. Arman, A. Al-Shishtawy, and V. Vlassov. Elasticity controller for cloud-based key-value stores. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 268–275, 2012.

[89] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[90] S. Dipietro, R. Buyya, and G. Casale. Pax: Partition-aware autoscaling for the cassandra nosql database. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9, 2018.

[91] M. Sedaghat, F. Hern´ndez, and E. Elmroth. Unifying cloud management: Towards overall governance of business level objectives. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 591–597, 2011.

[92] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[93] J. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai. Performance model driven qos guarantees and optimization in clouds. In *2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 15–22, 2009.

[94] Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai. Sla decomposition: Translating service level objectives to system level thresholds. In *Fourth International Conference on Autonomic Computing (ICAC'07)*, pages 3–3, 2007.

[95] K. Hwang, X. Bai, Y. Shi, M. Li, W. G. Chen, and Y. Wu. Cloud performance modeling with benchmark evaluation of elastic scaling strategies. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):130–143, Jan 2016.

[96] Anne Koziolek, Danilo Ardagna, and Raffaela Mirandola. Hybrid multi-attribute qos optimization in component based software systems. *Journal of Systems and Software*, 86(10):2542 – 2558, 2013.

[97] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 25–32, New York, NY, USA, 2019. Association for Computing Machinery.

[98] B. Beyer, C. Jones, J. Petoff, and N.R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. Oreilly, 2016.

[99] Mina Sedaghat, Francisco Hernández-Rodriguez, and Erik Elmroth. Decentralized cloud datacenter reconsolidation through emergent and topology-aware behavior. *Future Generation Computer Systems*, 56:51 – 63, 2016.

[100] M. Sedaghat, F. Hernández-Rodríguez, and E. Elmroth. Autonomic resource allocation for cloud data centers: A peer to peer approach. In *2014 International Conference on Cloud and Autonomic Computing*, pages 131–140, Sep. 2014.

[101] R. Han, C. H. Liu, Z. Zong, L. Y. Chen, W. Liu, S. Wang, and J. Zhan. Workload-adaptive configuration tuning for hierarchical cloud schedulers. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2879–2895, 2019.

[102] W. Fang, Z. Lu, J. Wu, and Z. Cao. Rpps: A novel resource prediction and provisioning scheme in cloud data center. In *2012 IEEE Ninth International Conference on Services Computing*, pages 609–616, June 2012.

[103] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, and Gabriel Iszlai. Feedback-based optimization of a private cloud. *Future Generation Computer Systems*, 28(1):104 – 111, 2012.

[104] Christina Delimitrou and Christos Kozyrakis. Hcloud: Resource-efficient provisioning in shared cloud systems. *SIGARCH Comput. Archit. News*, 44(2):473–488, March 2016.

[105] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 97–110, New York, NY, USA, 2015. Association for Computing Machinery.

[106] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 127–144, New York, NY, USA, 2014. Association for Computing Machinery.

[107] Christina Delimitrou and Christos Kozyrakis. Qos-aware scheduling in heterogeneous datacenters with paragon. *ACM Trans. Comput. Syst.*, 31(4), December 2013.

[108] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery.

[109] Pawel Janus and Krzysztof Rzadca. *SLO-Aware Colocation of Data Center Tasks Based on Instantaneous Processor Requirements*, page 256–268. Association for Computing Machinery, New York, NY, USA, 2017.

[110] Ye Hu, Johnny Wong, Gabriel Iszlai, and Marin Litoiu. Resource provisioning for cloud computing. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '09, page 101–111, USA, 2009. IBM Corp.

[111] L. Tomás, E. B. Lakew, and E. Elmroth. Service level and performance aware dynamic resource allocation in overbooked data centers. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 42–51, May 2016.

[112] L. A. Barroso, U. Hölzle, P. Ranganathan, and M. Martonosi. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. 2018.

[113] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for µsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[114] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008.

[115] F. Pascual and K. Rzadca. Optimizing egalitarian performance when colocating tasks with types for cloud data center resource management. *IEEE Transactions on Parallel and Distributed Systems*, 30(11):2523–2535, 2019.

[116] Nikolas Herbst, André Bauer, Samuel Kounev, Giorgos Oikonomou, Erwin Van Eyk, George Kousiouris, Athanasia Evangelinou, Rouven Krebs, Tim Brecht, Cristina L. Abad, and Alexandru Iosup. Quantifying cloud performance and dependability: Taxonomy, metric design, and emerging challenges. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(4), August 2018.

[117] Adalberto R. Sampaio, Julia Rubin, Ivan Beschastnikh, and Nelson S. Rosa. Improving microservice-based applications with runtime placement adaptation. *Journal of Internet Services and Applications*, 10(1):4, Feb 2019.

[118] Piotr Skowron and Krzysztof Rzadca. Flexible replica placement for optimized p2p backup on heterogeneous, unreliable machines. *Concurrency and Computation: Practice and Experience*, 28(7):2166–2186, 2016.

[119] S. Blagodurov, A. Fedorova, E. Vinnik, T. Dwyer, and F. Hermenier. Multi-objective job placement in clusters. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[120] Ali Fahs, Guillaume Pierre, and Erik Elmroth. Voilà: Tail-Latency-Aware Fog Application Replicas Autoscaler. In *MASCOTS 2020 - 27th IEEE Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, Nice, France, November 2020.

[121] Baptiste Lepers, Vivien Quema, and Alexandra Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 277–289, Santa Clara, CA, July 2015. USENIX Association.

[122] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4), December 2010.

[123] Tian Guo, Upendra Sharma, Timothy Wood, Sambit Sahu, and Prashant Shenoy. Seagull: Intelligent cloud bursting for enterprise applications. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 361–366, Boston, MA, June 2012. USENIX Association.

[124] Michael Tighe and Michael Bauer. Topology and application aware dynamic vm management in the cloud. *Journal of Grid Computing*, 15(2):273–294, Jun 2017.

[125] Y. M. Ramirez, V. Podolskiy, and M. Gerndt. Capacity-driven scaling schedules derivation for coordinated elasticity of containers and virtual machines. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 177–186, 2019.

[126] Mor Sides, Anat Bremler-Barr, and Elisha Rosensweig. Yo-yo attack: Vulnerability in auto-scaling mechanism. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 103–104, New York, NY, USA, 2015. ACM.

[127] A. Bremler-Barr, E. Brosh, and M. Sides. Ddos attack on cloud auto-scaling mechanisms. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[128] Sakil Barbhuiya, Zafeirios Papazachos, Peter Kilpatrick, and Dimitrios S. Nikolopoulos. Rads: Real-time anomaly detection system for cloud data centres, 2018.

[129] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. Microservices in industry: Insights into technologies, characteristics, and software quality. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 187–195, 2019.

[130] Panagiotis Patros, Kenneth B. Kent, and Michael Dawson. Mitigating garbage collection interference on containerized clouds. In *2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 168–173, 2018.

[131] Panagiotis Patros, Kenneth B. Kent, and Michael Dawson. Why is garbage collection causing my service level objectives to fail? *International Journal of Cloud Computing*, 7(3-4):282–322, 2018.

[132] R.J. Hyndman, A.B. Koehler, R.D. Snyder, and S. Grose. A state space framework for automatic forecasting using exponential smoothing methods. *International J. Forecasting*, 18(3):439–454, 2002.

[133] Charles C. Holt. Forecasting trends and seasonal by exponentially weighted averages. *International J. Forecasting*, 20(1):5–10, 2004.

[134] Peter R. Winters. Forecasting sales by exponentially weighted moving averages. *Management Science*, 6(3):324–342, 1960.

[135] George E. P. Box, Gwilym M. Jenkins, and Gregory C. Reinsel. *Time Series Analysis: Forecasting and Control*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2008.

[136] C. W. J. Granger and R. Joyeux. An introduction to long-memory time series models and fractional differencing. *Journal of Time Series Analysis*, 1:15–30, 1980.

[137] Robert Engle. Garch 101: The use of arch/garch models in applied econometrics. *Journal of Economic Perspectives*, 15(4):157–168, 2001.

[138] Nina Golyandina and Anatoly Zhigljavsky. *Singular Spectrum Analysis for Time Series*. Springer, Heidelberg, Germany, 2013.

[139] Nina Golyandina, Vladimir Nekrutkin, and Anatoly Zhigljavsky. *Analysis of Time Series Structure: SSA and related techniques*. Chapman & Hall, CRC, Washington, D.C., 2001.

[140] Harris Drucker, Christopher J. C. Burges, Linda Kaufman, Alexander J. Smola, and Vladimir N. Vapnik. Support vector regression machines. *Advances in Neural Information Processing Systems*, 9:155–161, 1996.

[141] Tilmann Gneiting and Adrian E Raftery. Strictly proper scoring rules, prediction, and estimation. *Journal of the American Statistical Association*, 102(477):359–378, 2007.

[142] Victor S Sheng, Foster Provost, and Panagiotis G Ipeirotis. Get another label? improving data quality and data mining using multiple, noisy labelers. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 614–622. ACM, 2008.

[143] F. T. Liu, K. M. Ting, and Z. Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422, Dec 2008.

[144] Raymond Hemmecke, Matthias Köppe, Jon Lee, and Robert Weismantel. *Nonlinear Integer Programming*, pages 561–618. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[145] R. Byrd, R. Schnabel, and G. Shultz. A trust region algorithm for nonlinearly constrained optimization. *SIAM Journal on Numerical Analysis*, 24(5):1152–1170, 1987.

[146] David J. Wales and Jonathan P. K. Doye. Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms. *The Journal of Physical Chemistry A*, 101(28):5111–5116, 1997.

[147] Panagiotis Patros, Kenneth B Kent, and Michael Dawson. Investigating the effect of garbage collection on service level objectives of clouds. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 633–634. IEEE, 2017.

[148] Panagiotis Patros, Kenneth B Kent, and Michael Dawson. Why is garbage collection causing my service level objectives to fail? *International Journal of Cloud Computing*, 7(3-4):282–322, 2018.

[149] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu. Detecting transient bottlenecks in n-tier applications through fine-grained analysis. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 31–40, July 2013.

[150] Albert-László Barabási and Márton Pósfai. *Network science*. Cambridge University Press, Cambridge, 2016.

[151] Phillip Bonacich. Power and centrality: A family of measures. *American Journal of Sociology*, 92(5):1170–1182, 1987.

[152] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[153] Docker compose files to analyze structural patterns of containerized microservice applications. `https://doi.org/10.5281/zenodo.3573846`, 2020. [Online; accessed 8-June-2020].

[154] Robert V. Krejcie and Daryle W. Morgan. Determining sample size for research activities. *Educational and Psychological Measurement*, 30(3):607–610, 1970.

[155] Adery C. A. Hope. A simplified monte carlo significance test procedure. *Journal of the Royal Statistical Society. Series B (Methodological)*, 30(3):582–598, 1968.

[156] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples)†. *Biometrika*, 52(3-4):591–611, 1965.

[157] M. A. Stephens. Edf statistics for goodness of fit and some comparisons. *Journal of the American Statistical Association*, 69(347):730–737, 1974.

[158] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74:47–97, Jan 2002.

[159] Vladimir Podolskiy, Maria Patrou, Panos Patros, Michael Gerndt, and Kenneth B. Kent. The weakest link: Revealing and modeling the architectural patterns of microservice applications. In *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering*, CASCON '20, page 113–122, USA, 2020. IBM Corp.

[160] X. Liu, J. Heo, and L. Sha. Modeling 3-tiered web applications. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 307–310, Sep. 2005.

[161] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. *IEEE Network*, 14(3):30–37, 2000.

[162] R. D. Yates, M. Tavan, Y. Hu, and D. Raychaudhuri. Timely cloud gaming. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[163] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[164] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25, pages 1223–1231. Curran Associates, Inc., 2012.

[165] David N. Reshef, Yakir A. Reshef, Hilary K. Finucane, Sharon R. Grossman, Gilean McVean, Peter J. Turnbaugh, Eric S. Lander, Michael Mitzenmacher, and Pardis C. Sabeti. Detecting novel associations in large data sets. *Science*, 334(6062):1518–1524, 2011.

[166] Gábor J. Székely, Maria L. Rizzo, and Nail K. Bakirov. Measuring and testing dependence by correlation of distances. *Ann. Statist.*, 35(6):2769–2794, 12 2007.

[167] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[168] Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. Online passive-aggressive algorithms. *J. Mach. Learn. Res.*, 7:551–585, December 2006.

[169] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[170] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association.

[171] V. Podolskiy, A. Jindal, and M. Gerndt. Iaas reactive autoscaling performance challenges. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 954–957, 2018.

[172] T. L. Nguyen and A. Lebre. Virtual machine boot time model. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 430–437, 2017.

[173] Podolskiy Vladimir. Docker compose files to analyze structural patterns of containerized microservice applications, Dec 2019.

[174] V. Podolskiy, M. Mayo, A. Koay, M. Gerndt, and P. Patros. Maintaining slos of cloud-native applications via self-adaptive resource sharing. In *2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 72–81, June 2019.

[175] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[176] V. Podolskiy, A. Jindal, M. Gerndt, and Y. Oleynik. Forecasting models for self-adaptive cloud applications: A comparative study. In *2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 40–49, 2018.

[177] Vladimir Podolskiy. Rubberband: Enabling elastic federated learning with the temporary state management. In *Proceedings of the 21st International Middleware Conference Doctoral Symposium*, Middleware '20 Doctoral Symposium, 2020.

[178] W. Hasselbring and G. Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 243–246, April 2017.

[179] Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, Alessandro V. Papadopoulos, Bogdan Ghit, Dick Epema, and Alexandru Iosup. An experimental performance evaluation of autoscaling policies for complex workflows. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 75–86, New York, NY, USA, 2017. ACM.

[180] A. Bauer, M. Züfle, N. Herbst, A. Zehe, A. Hotho, and S. Kounev. Time series forecasting for self-aware systems. *Proceedings of the IEEE*, 108(7):1068–1093, 2020.

[181] Hongzhi Yin, Bin Cui, Ling Chen, Zhiting Hu, and Zi Huang. A temporal context-aware model for user behavior modeling in social media systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 1543–1554, New York, NY, USA, 2014. Association for Computing Machinery.

[182] Geoffrey I. Webb, Michael J. Pazzani, and Daniel Billsus. Machine learning for user modeling. *User Modeling and User-Adapted Interaction*, 11(1):19–29, Mar 2001.

[183] Gerhard Fischer. User modeling in human–computer interaction. *User Modeling and User-Adapted Interaction*, 11(1):65–86, Mar 2001.

[184] Fabian Abel, Eelco Herder, Geert-Jan Houben, Nicola Henze, and Daniel Krause. Cross-system user modeling and personalization on the social web. *User Modeling and User-Adapted Interaction*, 23(2):169–209, Apr 2013.

[185] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service*, IWQoS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[186] C. Jiang, G. Han, J. Lin, G. Jia, W. Shi, and J. Wan. Characteristics of co-allocated online services and batch jobs in internet data centers: A case study from alibaba cloud. *IEEE Access*, 7:22495–22508, 2019.