VILNIUS GEDIMINAS TECHNICAL UNIVERSITY

Olesia POZDNIAKOVA

RESEARCH OF SERVICE LEVEL AGREEMENT AWARE AUTOSCALING ALGORITHMS FOR CONTAINERIZED CLOUD-NATIVE APPLICATIONS

DOCTORAL DISSERTATION

TECHNOLOGICAL SCIENCES,
INFORMATICS ENGINEERING (T 007)

Vilnius, 2025

The doctoral dissertation was prepared at Vilnius Gediminas Technical University in 2016–2025.

The dissertation will be defended at the public meeting of the Dissertation Defense Council of the Scientific Field of Informatics Engineering in the *Aula Dotoralis* Meeting Hall of Vilnius Gediminas Technical University at **1 p.m. on 16 May 2025**.

Address: Saulėtekio al. 11, LT-10223 Vilnius, Lithuania.
Tel.: +370 5 274 4956; fax +370 5 270 0112; e-mail: doktor@vilniustech.lt

A notification on the intended defense of the dissertation was sent on 15 April 2025.
A copy of the doctoral dissertation is available for review at Vilnius Gediminas Technical University repository https://etalpykla.vilniustech.lt, the Library of Vilnius Gediminas Technical University (Saulėtekio al. 14, LT-10223 Vilnius, Lithuania), and the Library of Kaunas University of Technology (K. Donelaičio st. 20, LT-44239 Kaunas, Lithuania).

Vilnius Gediminas Technical University book No. 2025-011-M

https://doi.org/10.20334/2025-011-M

VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS

Olesia POZDNIAKOVA

KONTEINERIZUOTŲ DEBESŲ KOMPIUTERIJOS PROGRAMŲ SISTEMŲ AUTOMATINIO MASTELIAVIMO ALGORITMŲ, PAGRĮSTŲ SUSITARIMU DĖL PASLAUGOS LYGIO, TYRIMAS

DAKTARO DISERTACIJA

TECHNOLOGIJOS MOKSLAI,
INFORMATIKOS INŽINERIJA (T 007)

Vilnius, 2025

Disertacija rengta 2016–2025 metais Vilniaus Gedimino technikos universitete.

Disertacija bus ginama viešame Informatikos inžinierijos mokslo krypties disertacijos gynimo tarybos posėdyje **2025 m. gegužės 16 d. 13 val.** Vilniaus Gedimino technikos universiteto *Aula Doctoralis* posėdžių salėje.

Adresas: Saulėtekio al. 11, LT-10223 Vilnius, Lietuva.
Tel.: +370 5 274 4956; fax +370 5 270 0112; el. paštas: doktor@vilniustech.lt

Pranešimai apie numatomą ginti disertaciją išsiųsti 2025 m. balandžio 15 d. Disertaciją galima peržiūrėti Vilniaus Gedimino technikos universiteto talpykloje https://etalpykla.vilniustech.lt/ ir Vilniaus Gedimino technikos universiteto bibliotekoje (Saulėtekio al. 14, LT-10223 Vilnius, Lietuva) ir Kauno technologijos universiteto bibliotekoje (K. Donelaičio g. 20, LT44239 Kaunas, Lietuva).

Abstract

The development of cloud-native applications focuses on scalability and loose coupling of containerized microservices to ensure smooth deployment on cloud or container orchestration platforms. An autoscaler is a crucial component responsible for dynamically provisioning compute resources. When dynamically provisioning resources, addressing issues such as timelines and the amount of resources to be provisioned is important. Therefore, most autoscaling algorithms aim to find a balance between avoiding Service Level Agreement (SLA) violations and effectively managing costs or energy. Various rules-based autoscaling approaches were created to address quality of service concerns and minimise the risk of SLA violations. When resources are allocated and adjusted as needed, an autoscaler typically evaluates current service performance by comparing it to a predefined service level indicator (SLI) value. However, this alone may be insufficient to address changes in SLA conformance. To respond appropriately, the autoscaler must also consider the system's overall SLA fulfillment status.

This research presents two innovative self-adaptive autoscaling solutions for SLA-sensitive applications. The first solution focuses on maintaining the defined Service Level Objective (SLO) to recover from service degradation and achieve the desired service level. The second solution features a novel SLA-aware dynamic CPU threshold adjustment algorithm. The algorithm aims to ensure that the application has sufficient resources to operate at a level that keeps the number of response time violations compliant with the SLO. Additionally, it aims to ensure that the system operates as closely as possible to the defined Service Level Objectives, thus minimising resource wastage. The solution employs exploratory data analysis techniques in conjunction with moving average smoothing to determine the target utilisation threshold.

The Kubernetes Horizontal Pod Autoscaler (HPA) remains the most widely used threshold-based autoscaling due to its simple setup, operation, and seamless integration with other Kubernetes functionalities. For that reason, this research compares the autoscaling solutions proposed here with the Kubernetes Horizontal Pod Autoscaler and evaluates their effectiveness and performance across various real-world workload scenarios. The evaluation methods for algorithms focus on their ability to operate near-defined SLOs and the effectiveness of resource provisioning. The analysis of the experimental results demonstrates that these solutions are successful in SLA fulfillment and SLO restoration goals while providing an adequate amount of resources to achieve these objectives.

The results of the dissertation were published in six scientific publications, two of which were in reviewed scientific journals indexed in *Web of Science* and presented at five international conferences.

Reziumė

Kuriant debesų kompiuterijos taikomąsias programas, daug dėmesio skiriama tam, kad konteinerizuoti mikroservisai būtų lengvai masteliuojami bei turėtų silpną sankibą, kas užtikrina sklandų taikomųjų sistemų diegimą debesų kompiuterijos ar konteinerių orkestravimo platformose. Automatinio masteliavimo komponentas (angl. *autoscaler*) yra esminis elementas, kai kalbama apie skaičiavimo resursų dinaminį paskirstymą, reaguojant į resursų poreikį. Kai automatinis masteliavimo komponentas paskirsto resursus, jis turi spręsti terminų bei tinkamo resursų kiekio nustatymo uždavinius, kurie daro įtaką paslaugos kokybei. Todėl dauguma automatinio masteliavimo algoritmų siekia rasti pusiausvyrą tarp susitarimo dėl paslaugų teikimo lygio (angl. *Service Level Agreement,* SLA) sąlygų pažeidimų išvengimo ir efektyvaus išlaidų ar energijos valdymo. Siekiant išspręsti paslaugų kokybės užtikrinimo problemas ir sumažinti SLA pažeidimų riziką, buvo sukurti įvairūs taisyklėmis pagrįsti automatinio mastelio keitimo metodai.

Šiame tyrime pristatomi du novatoriški, savaime prisitaikantis automatinio masteliavimo sprendimai. Pirmas sprendimas skirtas palaikyti paslaugų teikimo lygio tiksluose (angl. *Service Level Obectives,* SLOs) nurodytą lygį tais atvejais, kai jis pablogėja dėl netinkamo ar uždelsto resursų teikimo. Sprendimu siekiama palaikyti nustatytą paslaugų lygį bei atstatyti jį degradavus. Taip pat šiame tyrime pristatomas naujas, žiniomis apie SLA pagrįstas dinaminio slenksčio (angl. *threshold*) koregavimo algoritmas, skirtas CPU apkrovos slenksčiams nustatyti. Algoritmas siekia užtikrinti tokį resursų kiekį, kad taikomosios programos atsako į užklausas laikas neviršytų nustatyto paslaugos lygio tiksluose daugiau kartų nei leidžiama pagal paslaugos susitarimą. Be to, algoritmas siekia užtikrinti, kad teikiamos paslaugos kokybė kuo labiau atitiktų nustatytą paslaugos teikimo lygio tikslą, taip mažindamas resursų švaistymą. Slenksčiui nustatyti naudojami tiriamieji duomenų analizės metodai ir slankiojo vidurkio glodinimas.

HPA yra plačiausiai naudojamas automatinio masteliavimo komponentas. Jis išlieka populiarus dėl pakankamai paprasto valdymo ir integravimo su kitais *Kubernetes* komponentais. Dėl šios priežasties tyrime siūlomi automatinio masteliavimo sprendimai yra palyginti su HPA. Tyrimo tikslas yra įvertinti siūlomų sprendimų gebėjimą veikti pagal nustatytus SLO, kartu įvertinant jų efektyvumą paskirstyti resursus, esant įvairių tipų apkrovoms. Rezultatų analizė rodo, kad siūlomi sprendimai sėkmingai paskirsto resursus, užtikrindami SLO palaikymą ar SLO atkūrimą.

Disertacijos rezultatai buvo paskelbti 6 moksliniuose leidiniuose, iš kurių 2 – recenzuojamuose mokslo žurnaluose, indeksuotuose *Web of Science*, ir pristatyti 5 tarptautinėse konferencijose.

Notations

Symbols

R_{max}, R_{min} – the highest (max) and lowest (min) number of replicas that can be provisioned by autoscaler (liet. *Didžiausias (maks) ir mažiausias (min) replikų skaičius, kurį gali aprūpinti automatinio masteliavimo valdiklis*).

R_d^D – the number of pod replicas calculated based on the resource utilization metrics. Throughout the document, $D \in [up, down]$ represents the direction of the autoscaling action (liet. R_d^D *yra „pod" replikų skaičius, apskaičiuotas remiantis resursų apkrovos matavimo metrika. Visame dokumente* $D \in [Up, Down]$ *nurodo automatinio masteliavimo veiksmo kryptį*).

R_{tgt}^D – the target number of replicas refers to the number of pods determined based on various factors, including the velocity of load, SLO compliance state, or traffic volatility (liet. *Tikslinis replikų skaičius nurodo „pod" skaičių, nustatytą pagal įvairius veiksnius, įskaitant apkrovos greitį, SLO suderinamumo būseną arba srauto kintamumą*).

R_{BP} – the number of replicas provisioned when the current SLO value SLO_n is way below target (liet. *Pateiktų replikų skaičius, kai dabartinė SLO reikšmė* SLO_n *yra žemiau tikslinės reikšmės*).

ΔR_V – the downscale step represents the number of replicas that will be removed from operations when traffic is volatile (liet. *Mažinimo žingsnis nurodo replikų, kurios bus pašalintos iš aplinkos, esant kintamai apkrovai, skaičių*).

SLO_{tgt} – the current value of SLO measurement (liet. *Dabartinė SLO matavimo vertė*).

SLO_{BP} – the SLO restoration breaking point threshold (liet. *SLO atstatymo lūžio taško slenkstis*).

$SLO_{noDownScale}$ – the no downscale action SLO threshold is used to control the risk of SLA violation after SLO recovery action. Autoscaling actions are prohibited until the specified SLO threshold is reached (liet. *Slenkstis, naudojamas SLO neįvykdymo rizikos mažinimui po degraduotos paslaugos kokybės atstatymo. Automatinio masteliavimo veiksmai yra uždrausti tol, kol pasiekiama nurodyta SLO riba*).

L_{max} – the highest load (request per second, connections per second, packets per second) that a single replica can handle without violating the target service level (liet. *Didžiausia apkrova (užklausa per sekundę, sujungimai per sekundę, paketai per sekundę), kurią gali apdoroti viena replika, nepažeidžiant nustatyto paslaugos lygio*).

$T^{D}_{cooldown}$ – the cooldown period is when no autoscaling action happens (liet. *Laikotarpis, per kurį nevyksta jokie veiksmai*).

$\Delta T^{D}_{cooldown}$ – the cooldown period adjustment steps. Used to shorten upscale action cooldown period if load increase is high and prolong it in cases when the load is low (liet. *Laikotarpio, per kurį nevyksta jokie veiksmai, koregavimo žingsniai, naudojami norint sumažinti šį laikotarpį, kai apkrova stipriai padidėja, ir pratęsti laikotarpį, kai apkrova maža*).

T_n – a length of synchronization period between the SAA solution and the Kubernetes cluster on the number of running replicas (liet. *Nurodo periodo, per kurį SAA ir* Kubernetes *sprendimai sinchronizuoja informaciją apie replikų kiekį, ilgį*).

T_m – the length of the monitoring samples collection period (liet. *Metrikų rinkimo laikotarpio ilgis*).

t^{D}_{delay} – replica provisioning or deprovisioning time (liet. *Replikos teikimo arba teikimo atšaukimo laikas*).

$t^{D}_{totalDelay}$ – resource provisioning delay is the time it takes to adjust resource provision or deprovision based on demand shifts (liet. *Resursų suteikimo delsa – tai laikas, reikalingas resursų suteikimui arba panaikinimui, priklausantis nuo resursų poreikio pokyčių*).

$v_{baseline}$ – baseline velocity defines what the maximum load (L_{max}) increase per second can be handled by a single replica during a period equal to $t^{D}_{totalDelay}$ (liet. *Bazinis greitis apibrėžia, kokia maksimali apkrova (L_{max}) gali būti apdorojama vienos replikos per laikotarpį, lygų $t^{D}_{totalDelay}$*).

α_D – velocity factor gives a raw estimate of how many times the current load change velocity v_n is different from the baseline velocity (liet. *Greičio faktorius nurodo, kiek kartų dabartinis apkrovos kitimo greitis v_n skiriasi nuo bazinio greičio*).

A_k – raw velocity is a ratio between $v_{baseline}$ and velocity v_k, where k is the array element index (liet. *Nemodifikuotas greitis nurodo santykį tarp $v_{baseline}$ ir greičio v_k, kur k yra masyvo elemento indeksas*).

$V_k(A_k)$ – velocities vector is used by the volatility detector module to analyze the last K monitoring samples of velocity and determine if the load is volatile (liet. *Kintamo srauto detektoriaus modulis naudoja greičio vektorių paskutinių K greičio matavimo stebėjimų analizei ir kintamos apkrovos aptikimui*).

C_n – current average CPU utilization of all replicas that are running and available to handle the workload (liet. *Visų replikų, paruoštų ir naudojamų apkrovos apdorojimui, dabartinė vidutinė CPU apkrova*).

CT^D – the indicator of CPU threshold selected ($CT \in [upper, mid, lower]$) for upscaling or downscaling action ($D \in [Up, Down]$) (liet. *Rodiklis, rodantis, koks CPU slenkstis pasirinktas ($CT \in [upper, mid, lower]$) replikų kiekio padidinimo arba sumažinimo veiksmui ($D \in [Up, Down]$)*).

$C_c^{CT^D}$ – the dynamic CPU Thresholds are dynamically adjusted thresholds used to trigger autoscaling action and calculate the desired replica number (liet. *Dinaminiai CPU slenksčiai yra dinamiškai koreguojami slenksčiai, naudojami masteliavimo veiksmui iššaukti ir replikų kiekiui skaičiuoti*).

ΔC^D – CPU threshold adjustment steps used to adjust CPU or another threshold (e.g. RAM) used for autoscaling action (liet. *CPU slenksčio reguliavimo žingsnio dydis naudojamas koreguoti CPU arba kito tipo apkrovos slenkstį (pvz., operatyvios atminties panaudojimo), taikomą automatiniam masteliavimui*).

T_c – CPU adjustment period (s) is the period during which the CPU Adjuster module validates the state of SLO (above, on target, or below) and adjusts CPU utilization threshold values (liet. *CPU koregavimo laikotarpis (sek.) tai laikotarpis, per kurį CPU koregavimo modulis patikrina SLO būseną (viršija, pagal arba žemiau tikslo) ir koreguoja CPU apkrovos slenksčio reikšmes*).

Abbreviations

AKS – Azure Kubernetes Service (liet. Azure Kubernetes *paslauga*);

API – application programming interface (liet. *programinė sąsaja*);

AWS – Amazon Web Services (liet. Amazon *Web Services*);

CA – Cluster Autoscaler (liet. *klasterio automatinio masteliavimo valdiklis*);

CAP – consistency, availability, and partitioning (liet. *nepriėštaringumas, pasiekiamumas ir skaidymas*);

CMA – centered moving average (liet. *centruotas slankusis vidurkis*);

CNCF – Cloud Native Computing Foundation (liet. *debesų kompiuterijos kilmės pagrindas*);

CNI – container network interface (liet. *konteinerio tinklo sąsaja*);

CPA – Custom Pod Autoscaler (liet. *tinkintas „pod“ automatinio masteliavimo valdiklis*);

DCTA – dynamic CPU threshold adjuster (liet. *dinaminis CPU slenksčio reguliatorius*);

EDGAR – electronic data gathering, analysis, and retrieval (liet. *elektroninių duomenų rinkimas, analizė ir gavimas*);

DMAR – Dynamic Multi-level Autoscaling Rules (liet. *Dinaminės kelių lygių automatinio masteliavimo taisyklės*);

EC2 – Elastic Compute (liet. *elastinis skaičiavimo resursas*);

HPA – horizontal pod autoscaler (liet. *horizontalaus „pod“ masteliavimo valdiklis*);

GCP – Google Cloud Platform (liet. Google *debesų kompiuterijos platforma*);

IAAS – Institute of Architecture of Application Systems (liet. *Taikomųjų sistemų architektūros institutas*);

IQR – interquartile range (liet. *tarpkvartilinis diapazonas*);

ML – machine learning (liet. *mašininis mokymas*);

MSA – microservice architecture (liet. *mikroservisų architektūra*);

OCI – Open Container Initiative (liet. *Atvirosios kompiuterijos iniciatyva*);

QoS – Quality of Service (liet. *paslaugos kokybė*);

RAM – Random Access Memory (liet. *operatyvioji atmintis*);

RT – response time (liet. *atsako laikas*);

RU – resource utilization (liet. *resursų panaudojimas*);

SAA – SLA-aware autoscaler (liet. *susitarimu dėl paslaugos teikimo lygio pagrįstas automatinis masteliavimo valdiklis*);

SaaS – Software-as-a-Service (liet. *programinė įranga kaip paslauga*);

SLA – service level agreement (liet. *susitarimas dėl paslaugos teikimo lygio*);

SLI – service level indicator (liet. *paslaugos veikimo lygio indikatorius*);

SLO – service level objective (liet. *paslaugos teikimo lygio tikslas*);

SATA – SLA-Aware Threshold Adjustment (liet. *susitarimu dėl paslaugos teikimo lygio slenksčio pagrįstas reguliatorius*);

SMA – simple moving average (liet. *paprastasis slankusis vidurkis*);

VPA – Vertical Pod Autoscaler (liet. *vertikalaus „pod“ masteliavimo valdiklis*).

Contents

INTRODUCTION 1
Problem Formulation 1
Relevance of the Dissertation 2
Research Object 2
Aim of the Dissertation 2
Tasks of the Dissertation 3
Research Methodology 3
The Scientific Novelty of the Dissertation 4
The Practical Value of the Research Findings 4
Defended Statements 4
Approval of the Research Findings 5
The Structure of the Dissertation 5

1. LITERATURE REVIEW OF AUTOSCALING METHODS FOR THE ASSURANCE OF SERVICE LEVEL AGREEMENT FULFILLMENT IN CLOUD-NATIVE APPLICATIONS 7
1.1. Cloud-Native Applications 7
1.1.1. Application Containerization 12
1.1.2. Microservice Architecture 14
1.2. Service Level Measurement and Monitoring for the Cloud-Native Applications 16

1.2.1. Service Level Agreements, Objectives, Indicators, and Scaling Indicators 16
1.3. Autoscaling of Containerized Cloud-Native Applications 19
1.4. Kubernetes 27
1.4.1. Kubernetes Architecture 28
1.4.2. Autoscaling in Kubernetes 29
1.5. Autoscaler Performance Evaluation Methods 35
1.6. Conclusions of the First Chapter and Formulation of the Tasks of the Dissertation 38

2. DESIGN OF SERVICE LEVEL AGREEMENT-AWARE AUTOSCALING ALGORITHMS 41
2.1. Development of a Service Level Agreement-Adaptive Autoscaling Algorithm 41
2.1.1. Motivation for Developing a Rules-Based Autoscaler Solution . . . 42
2.1.2. Service Level Agreement-Aware Algorithm Design 42
2.1.3. Evaluation Criteria for the Efficacy of Autoscals in Service Level Agreement Compliance Assurance 58
2.2. Service Level Agreement-Adaptive Dynamic Threshold-Adjustment Algorithm Design for Rules-based Autoscalers 60
2.2.1. Motivation For Developing a Solution to Adjust the Thresholds in the Context of Horizontal Pod Autoscaler 60
2.2.2. Target Threshold Determination Algorithm 62
2.2.3. Dynamic Threshold-Adjustment Algorithm 68
2.2.4. Threshold-Adjustment Algorithm Evaluation Criteria 73
2.3. Conclusions of the Second Chapter 75

3. EXPERIMENTAL INVESTIGATION AND EVALUATION OF SERVICE LEVEL AGREEMENT-ADAPTIVE RULES-BASED AUTOSCALING ALGORITHMS 77
3.1. Common Experimental Setup 77
3.2. Experimental Evaluation of the Service Level Agreement Adaptive Autoscaling Algorithm 81
3.2.1. Experimental Setup for Autoscaler Performance Evaluation with Synthetic Workloads 83
3.2.2. Experimental Setup for the Autoscalers Performance Evaluation under Real-World Workload Conditions 86
3.2.3. Autoscaler Solutions Used for the Comparative Analysis 88
3.2.4. Results of Experimental Evaluation 92
3.2.5. Discussion 111

3.3. Experimental Evaluation of the Service Level Agreement-Adaptive Threshold-Adjustment Algorithm 113
3.3.1. Evaluation of the Impact of Different Settings on the Algorithm Performance 114
3.3.2. Evaluation of Performance under Different Workload Scenarios . 118
3.3.3. Discussion 128
3.4. Conclusions of the Third Chapter 131

GENERAL CONCLUSIONS 133

REFERENCES 137

LIST OF SCIENTIFIC PUBLICATIONS BY THE AUTHOR ON THE TOPIC OF THE DISSERTATION 149

SUMMARY IN LITHUANIAN 151

Introduction

Problem Formulation

The rise in popularity of application containerization and microservice architecture in recent years has led to the emergence of a new paradigm known as cloud-native applications (CNA). CNA often consists of multiple scalable and loosely coupled services that run as containerized application instances. However, it is crucial to provision these application instances on time and in the amount required to meet the performance objectives defined in the Service Level Agreement (SLA). Container orchestration platforms are specifically designed to streamline the operation of larger-scale containerized applications, where the autoscaler component plays a critical role in addressing the application demand for compute resources.

Autoscalers must address such issues as timelines and the amount of resources to be provisioned. Premature or excessive resource provisioning can lead to increased costs, while delays could result in service degradation and SLA violations. Therefore, most autoscaling algorithms aim to find a balance between avoiding SLA violations and effectively managing resources. The importance of considering infrastructure costs in business cannot be underestimated. However, the quality of service delivered to end users can directly impact the overall success of a business (Arapakis, Park, & Pielot, 2021; Sekhi, 2023). Failure to meet SLAs can result in business loss or penalties. Therefore, autoscalers aimed at fulfilling SLAs must ensure the desired quality of service while effectively managing resource demand (Al-Dhuraibi, Paraiso, Djarallah, & Merle, 2018; Amiri & Mohammad-Khanli, 2017). This requires the implementation of SLA awareness mechanisms to react appropriately to deviations from the defined service level indicators and objectives.

Relevance of the Dissertation

Cloud adoption and application containerization are increasing, with more applications moving to the cloud and microservice architecture (Kazanavičius, Mažeika, & Kalibatienė, 2022), leading to a new cloud-native paradigm. The popularity of cloud-native solutions has led to the establishment of the Cloud Native Computing Foundation (CNCF) (CNCF, 2024b). The foundation aims to promote the adoption of technologies that enable organizations to develop and operate scalable applications in modern, dynamic environments, including public, private, and hybrid clouds. The cloud-native landscape of CNCF encompasses over 40 open-source projects and proprietary products categorized under Orchestration and Scheduling (CNCF, 2024a), most of which are dedicated to containerized application orchestration. The number of new members and solutions continues to increase annually. Furthermore, Kubernetes is an emerging container orchestration platform that is adopted in growing numbers each year (CNCF, 2024c; Datadog, 2024). Companies like NetApp, Google, and Datadog are developing autoscaler products, and academia is actively researching autoscaling solutions for cloud-native applications. Thousands of articles and research papers dedicated to cloud-native and autoscaling can be found in databases, such as *Semantic Scholar*, *Web of Science*, and *Scopus*. Despite more than a decade of research in this area, trends continue to grow, and the problem of efficient autoscaling remains relevant.

The initial autoscaling solutions mostly focused on cost efficiency. However, there is a growing emphasis on the SLA aspect of autoscaling, as consumers and customers prioritize performance and quality of service (Arapakis et al., 2021; Pusztai et al., 2021; Sekhi, 2023). In the last five years, vendors like VMWare (VMware, 2024), Google (Rzadca et al., 2020), and Splunk (Splunk, 2024) and academia (Poojitha & Ravindranath, 2025; Pusztai et al., 2021; Qian et al., 2022; Ruiz, Pueyo, Mateo-Fornes, Mayoral, & Tehas, 2022; Tonini et al., 2023; Wen et al., 2023) are paying more attention to SLA awareness in cloud-native solutions.

Research Object

The object of the present dissertation is the SLA-aware autoscaling algorithms for cloud-native applications.

Aim of the Dissertation

The dissertation aims to enhance the fulfillment of performance-based Service Level Objectives for containerized cloud-native applications by utilizing rules-based autoscaling algorithms.

Tasks of the Dissertation

The following tasks must be solved to achieve the aim of the dissertation:

1. To perform a scientific literature review on the current state of autoscaling algorithms and methods that aim to ensure cloud-native applications' performance compliance with service-level objectives.
2. To develop the SLA-aware autoscaling approaches for autoscalers designed for containerized cloud-native applications, addressing the SLA fulfillment aspect.
3. To propose the SLA fulfillment efficacy evaluation methods for autoscaling approaches.
4. To assess the efficacy and efficiency of the proposed autoscaling methods under various workload conditions.
5. To perform a comparative analysis of the proposed autoscaling approaches and to compare them with widely adopted autoscaling solutions.

Research Methodology

The following *research methods* were chosen to investigate the *object*:

1. An *analytical literature review* has been conducted on cloud-native trends and existing autoscaling algorithms to ensure the SLA compliance of cloud-native applications. Strengths and weaknesses were evaluated, and the main SLA performance-impacting factors were assessed. Existing gaps in autoscaling solutions were identified.
2. The *comparative analysis* was used to evaluate the advantages and disadvantages of the analyzed and evaluated approaches.
3. The *statistical exploratory analysis* method was applied to understand how CPU threshold selection impacts performance-based SLAs' achievement.
4. The *experimental research* method was applied. The experiments were conducted to evaluate the effectiveness of the proposed rules-based autoscaling approaches in fulfilling the performance SLA requirements un-

der various load conditions. The Gatling and Jmeter load generation tools were utilized to create multiple types of workloads. An experimental environment was established on the Azure public cloud platform using Azure Kubernetes Service (AKS). All prototypes for autoscaling algorithms were implemented in the Java programming language. The data collected during the experiments was analyzed to assess the solutions against predefined evaluation criteria.

The Scientific Novelty of the Dissertation

The scientific novelty of this dissertation is specified as follows:

- The proposed algorithms enhance the SLA fulfillment for rules-based autoscalers. They are designed to mitigate the complexity introduced by the use of machine learning algorithms while improving compliance with performance-based service level agreements.
- In scenarios where allocating additional resources can significantly improve service performance and minimize the likelihood of violating SLO, the addition of resources enables SLO compliance recovery after it is degraded. This approach can be used in addition to the traditionally used SLA violation avoidance mechanisms in rules-based cloud-native application autoscalers.
- The introduced SLA restoration approach improves compliance with the defined performance-based service level objectives.

The Practical Value of the Research Findings

The proposed approaches are significant from theoretical and practical perspectives in ensuring that cloud-native applications meet SLA performance requirements. The approaches also aim to optimize the performance of existing autoscaler solutions with respect to performance-based SLA fulfillment. As cloud-native practices become more popular, there is a growing need for more robust, reliable, and SLA-efficient autoscaling solutions.

Determining the threshold to meet performance-based Service Level Objectives (SLOs) in many widely used autoscaling solutions is currently a manual and error-prone process. This is due to its reliance on static threshold principles, requiring manual configuration. The proposed dynamic threshold determination approach saves time when tuning HPA performance and lays the groundwork for further research into efficiency optimization.

Defended Statements

The following statements, based on the results of the present dissertation, may serve as the official hypotheses to be defended:

1. In scenarios where the allocation of additional resources can significantly improve service performance and minimize the likelihood of violating SLO, the addition of resources allows for recovery of SLO compliance after it is degraded, and this approach can be used as a mechanism to improve performance-based SLO compliance, along with the traditionally used SLA violation avoidance mechanisms in rules-based cloud-native application autoscalers.
2. The dynamic manipulation of the utilization threshold can be used to improve SLA fulfillment in rules-based autoscaling solutions when the utilization threshold is the most influential SLA factor.
3. Extending the tracking timeframe for SLA compliance enhances the ability of cloud-native autoscalers to meet established performance-based SLOs, in contrast to autoscalers that rely solely on current service level indicator values for their scaling decisions.

Approval of the Research Findings

The results of the dissertation were published in two scientific publications in reviewed scientific journals indexed in *Web of Science* Citation Index, three were published in conference proceedings, and one in a conference presentation abstract in an international DB. The author gave five presentations at international scientific conferences:

- The 8th Data Analysis Methods for Software Systems (DAMSS), 1–3 December 2016, Druskininkai, Lithuania.
- 2017 Open Conference of Electrical, Electronic and Information Sciences (eStream), 27 April 2023, Vilnius, Lithuania.
- The 24th International Conference on Information and Software Technologies (ICIST2018), 4–6 October 2018, Vilnius, Lithuania.
- The 14th Data Analysis Methods for Software Systems (DAMSS), 30 November – 2 December 2023, Druskininkai, Lithuania.
- The 1st International Symposium on Parallel Computing and Distributed Systems (PCDS2024), 21–22 September 2024, Singapore.

The Structure of the Dissertation

The dissertation consists of an introduction, three main chapters, general conclusions, references, a list of the author's publications on the topic of the dissertation, and a summary in Lithuanian. Its total scope is 167 pages, 44 equations, 29 figures, and 24 tables.

1

Literature Review of Autoscaling Methods for the Assurance of Service Level Agreement Fulfillment in Cloud-Native Applications

This chapter provides an overview of cloud-native trends and explores the primary definition of cloud-native applications. It also discusses autoscaling solutions and methods used for cloud-native applications, including a comprehensive analysis of autoscaling approaches and types, focusing on autoscaling solutions oriented toward Service Level Agreement (SLA) fulfillment. It also provides an overview of autoscaler efficiency evaluation methods from viewpoints of resource waste and quality of service. The chapter concludes by summarizing the literature review findings and clarifying the main objectives and tasks of the dissertation.

Regarding the topic discussed in this chapter, four publications were published by the author: Pozdniakova and Mažeika (2017b), Pozdniakova, Mažeika, and Cholomskis (2018a), Pozdniakova, Cholomskis, and Mažeika (2023), Pozdniakova, Mažeika, and Cholomskis (2024).

1.1. Cloud-Native Applications

Before going deeper into the autoscaling of containerized cloud-native applications, it is crucial to understand what cloud-native applications are and their main characteristics. This sub-chapter will provide an overview of cloud-native applications based on articles presented in Table 1.1 starting from early history.

Amazon Web Services (AWS) was the first public cloud provider in the world and is considered the pioneer of public cloud technology. It introduced its first service, Elastic Compute, in August 2006, preceding other popular providers such as Microsoft Azure, which appeared in 2008, Alibaba in 2009, and Google in 2011. Cloud-native is still a growing trend with a high level of adoption across industry (CNCF, 2024c) and high interest in academia. The earliest mention of a cloud-native (or cloud-ready or cloud-aware) solution in bibliometric databases such as *IEEE Xplore*, *Scopus*, and *Web of Science* dates back to 2008, two years after AWS was released. This might be considered a representation of the early interest in the cloud-native paradigm.

Table 1.1. Studies by the cloud-native research area question mapping made by the author

Research area	**Paper**
What is the cloud-native application?	Andrikopoulos, Strauch, Fehling, and Leymann (2012); Brown and Capern (2014); Deng et al. (2024); Fehling, Leymann, Retter, Schupeck, and Arbitter (2014); Gannon, Barga, and Sundaresan (2017); Inzinger et al. (2014); Kavis (2014); Kratzke and Peinl (2016); Kratzke and Quint (2017); Leymann, Fehling, Wagner, and Wettinger (2016); Pivotal (2017); Retter and Fehling (2013); Toffetti, Brunner, Blöchlinger, Spillner, and Bohnert (2016); VMware (2016)
What main characteristics do cloud-native applications have?	Brunner, Blochlinger, Toffetti, Spillner, and Bohnert (2016); Casper, Bette, and Louie (2014); CNCF (2024b); Deng et al. (2024); Fehling, Leymann, Mietzner, and Schupeck (2011); Fehling et al. (2014); Hole (2016); Kavis (2014); Kosińska, Brotoń, and Tobiasz (2024); Kourtesis, Bratanis, Bibikas, and Paraskakis (2012); Kratzke and Peinl (2016); Kratzke and Quint (2017); Leymann et al. (2016); Lichtenthäler and Wirtz (2024); Peinl, Holzschuher, and Pfitzer (2016); Retter and Fehling (2013); Roussev, Ahmed, Barreto, McCulley, and Shanmughan (2016); Sodhi and Prabhakar (2011); Stine (2015); Toffetti et al. (2016); Weinman (2016); Wilder (2012); Zimmermann (2017)

The adoption of public cloud solutions was initially slow due to security concerns. The same trend was in the interest of researchers in this area. As can be seen from Figure 1.1, the interest in cloud solutions began to grow between 2012 and 2017. During this period, Docker and Kubernetes were released in 2011 and 2014, respectively. Several articles were published aimed at defining cloud-native applications and their non-functional characteristics as per the cloud-native applications vision of that period. Multiple terms are used in the analyzed literature to define applications that are designed and developed specifically to run on the cloud, with the earliest mention dating back to 2012. Here are just a few examples of the findings. Andrikopoulos et al. (2012), Toffetti et al. (2016), Wilder (2012), Kratzke and Peinl (2016), and Kratzke and Quint (2017) use the "cloud-native" term in their work, each providing a definition for it. "Cloud-ready" is mentioned by Brown and Capern (2014) from IBM, Kavis (2014), and Weinman (2016). The "Cloud-aware" term is met in the Open Datacenter Alliance report (Casper et al., 2014) and Sodhi and Prabhakar (2011).

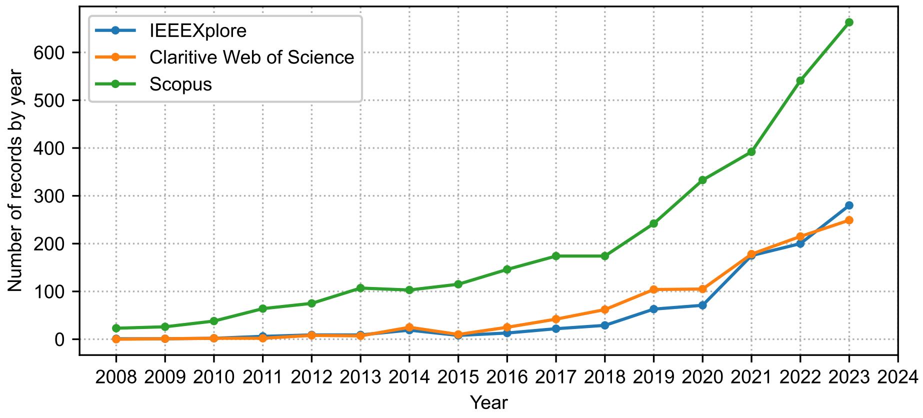


Fig. 1.1. Number of research articles returned for "cloud-native OR cloud-ready OR cloud-aware" query by year in Scopus (Scopus, 2024), Web Of Science (Clarivate, 2024) and IEEE Xplore (IEEEXplore, 2024) bibliometric databases

The early attempts to define the "cloud-native application" term were made by Andrikopoulos et al. (2012) and Wilder (2012). Andrikopoulos et al. (2012) define cloud-native applications as "…applications that are specifically designed and developed on top of a constellation of Cloud services, and which can fully exploit the characteristics of Cloud computing." He also defines cloud-enabled software as software that was specifically adopted to be suitable for the cloud (Andrikopoulos, Binz, Leymann, & Strauch, 2013). This is similar to the definition provided by Wilder (2012): "A cloud-native application is architected to take full advantage of cloud platforms." Later, Toffetti et al. (2016) made this definition even more universal: "Cloud-native application …is an application that has been specifically designed to run in a cloud environment." Once containerization technology, such as OCI and Docker (defined in sub-chapter 1.1.1), became popular, the cloud-native definition became less technology-agnostic. The definitions incorporated guidelines on how cloud-native applications should be developed. In 2017, the term "cloud-native application" was popularized by VMware and Pivotal to describe containerized applications or applications developed using microservice architecture. At approximately the same time, in 2015, the Cloud Native Computing Foundation (CNCF) was established by the Linux Foundation. CNCF aims to serve as a vendor-neutral center for cloud-native computing. CNCF defines cloud-native as the collection of technologies that "…breakdown applications into microservices and packages them in lightweight containers to be deployed and orchestrated across a variety of servers" (Deng et al., 2024). Notably, the "cloud-native application" term was used by Andrikopoulos et al. (2013) long before the emergence of microservices, Docker, or similar solutions. The ambiguity surrounding this term has also been acknowledged by Kratzke and Peinl (2016), Leymann et al. (2016) and Gannon et al. (2017).

Due to lacking common definition in academic literature, Kratzke and Peinl (2016) aimed to define cloud-native application more explicitly: "A cloud-native application is a distributed, elastic and horizontal scalable system composed of (micro)services which isolate state in a minimum of stateful components. The application and each self-contained deployment unit of that application are designed according to cloud-focused design patterns and operated on a self-service elastic platform." In later published work, Kratzke and Quint (2017) provided a more detailed definition and explanation of the terms, clarifying that the term "self-contained deployment unit" used in this definition refers to containers. The latest definition of cloud-native application found is provided by Mitchell (2023): "Cloud native applications are well-architected systems that are "container" packaged, and dynamically managed."

The above definitions emphasize the importance of using specific architecture patterns, containerization, and an elastic platform or orchestration. The systematic mapping study of cloud-native application design and engineering provided by Odun-Ayo, Goddy-Worlu, Ajayi, Edosomwan, and Okezie (2019) showed that the tools used in cloud-native applications and cloud-native applications architecture evaluation are the most popular topics in the cloud-native area. The cloud-native applications must comply with specific functional and non-functional requirements. What are those? The team from the Institute of Architecture of Application Systems (IAAS) of the University of Stuttgart did a lot of work on the rise of cloud-native applications, aiming to define approaches and patterns for cloud-native application development. Andrikopoulos et al. (2012) evaluated the effect of design decisions on the consistency, availability, and partitioning (CAP) properties of cloud-native applications. Andrikopoulos et al. (2013) discussed challenges in the adoption of applications for the cloud, which include complexity related to applications migration to the cloud, its operations, including cost and SLA, and portability across clouds. Fehling et al. (2011, 2014) discussed various cloud computing services and their application patterns, the challenges of deploying these patterns in the cloud, and proposed possible solutions to overcome these challenges. In their work, they define IDEAL properties for a cloud-native application, which should address the challenges of running applications on an elastic platform such as the cloud. IDEAL is an acronym where “I” stands for an isolated state, “D” is for distributed, “E” is elastic, “A” is automated, and “L” is loosely coupled. Wilder (2012) defines eleven properties of cloud-native applications, such as loose coupling, horizontal and automatic scale, fault tolerance, and resiliency (upgrades, migration, and faults should happen without downtime). The work presents 13 patterns that enable these properties in cloud-ready applications.

Other academic efforts also aimed to define the requirements for cloud-native applications. Kratzke and Peinl (2016), in their cloud-native application reference model for enterprise architects, called ClouNS, raised the importance of vendor lock-in avoidance in cloud-ready application design. A systematic mapping study prepared by Kratzke and Quint (2017) provides cloud-native application principles, such as the need for automation platforms, software-defined infrastructure, migration, and interoperability between the clouds. Aiming to evaluate the quality of the cloud-native application architecture, Lichtenthäler and Wirtz (2024) used 69 cloud-native application characteristics.

Deng et al. (2024) stated that in addition to the microservices architecture, a cloud-native application is also characterized by containerization and orchestration technologies.

O’Reilly’s report, written by Stine (2015), provides a set of characteristics that cloud-native applications should have, such as fault tolerance, state and fault isolations, horizontal scalability, automatic recovery, and statelessness. As per CNCF, the main goal of being cloud-native is to enhance the speed of application delivery, scalability, and resilience while reducing the technical risks associated with deployment. To achieve this, the cloud-native approach involves loosely connected systems that operate together securely, resiliently, and in a manageable, sustainable, and observable manner (CNCF, 2024b; Kosińska et al., 2024). Cloud-native also involves ensuring that applications can be easily moved across different cloud platforms, regardless of whether they are public, private, or hybrid cloud or they are using AWS, Azure, or GCP. This approach decreases vendor lock-in, as developers treat all resources as cloud-hosted or designed for the cloud (CNCF, 2024b).

Even though the literature contains many definitions of cloud-native applications and authors do not agree on some implementation details (e.g., statefulness of components and mandatory use of (micro)services), there are several common traits. First, a cloud-native application runs on the cloud, which is a distributed system. As a result, architects designing applications that run on cloud computing will face similar problems that are common for all distributed systems (even those that are not cloud-native, conventional ones). Second, cloud-native applications must be elastic. Herbst, Kounev, and Reussner (2013) defined what elasticity in cloud computing means: “Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an automatic manner, such that at each point in time the available resources match the current demand as closely as possible.” Most conventional applications were not designed to run in an environment that automatically scales as the number of transactions increases; not all conventional applications were designed to run on elastic platforms and be elastic.

The cloud-native application characteristics mentioned above aim to address the challenges of running applications on a dynamically changing (elastic) platform. The cloud resource payment model operates on a pay-as-you-go system, allowing users to pay only for the resources they use to complete specific tasks and to turn them off when they are no longer needed. Consequently, it is essential to maintain the system state if cloud instances are turned on or off, or to design instances to be stateless. Therefore, the application instance should be as small as possible, and the system should be portable across different cloud platforms to avoid vendor lock-in.

Based on the overview above, it could be concluded that a cloud-native application is designed as a distributed system with loosely coupled components intended for scalability and capable of running on an automated and elastic platform, such as the cloud. Ideally, these applications should be easily transferable between different cloud platforms without causing service interruptions (Alonso et al., 2023; CNCF, 2024b; Fehling et al., 2011; Kratzke & Quint, 2017). When all the aforementioned characteristics are met, a cloud-native application achieves the highest level of maturity (Kratzke, 2018). As a result, to be elastic, the cloud-native applications must be architected and designed using specific architecture patterns and tooling that enable elasticity.

Elasticity and automation are crucial features that distinguish cloud-ready applications from traditional ones. To fully leverage the cloud’s potential, the thoughtfully chosen technologies and software architecture should help mitigate potential challenges related to the operations of applications on the cloud and enable the realization of the benefits offered by the cloud. A microservices-based architecture (MSA) supports the elasticity feature. Adopting containerization technology by cloud-native applications helps address the challenges of portability and elasticity (Deng et al., 2024; Kratzke & Quint, 2017). Cloud and container orchestration platforms and autoscalers enable elasticity (Kosinska, Balis, Konieczny, Malawski, & Zielinski, 2023). The following sub-chapters will provide more details about these technologies.

1.1.1. Application Containerization

The widespread adoption of containerization technology has brought numerous benefits, including enhanced agility in application development, optimized resource utilization, and faster resource provisioning. Containers can start as fast as a new process and more quickly than it takes to boot a new virtual machine. Furthermore, the implementation of containers creates an extra layer of abstraction, allowing multiple application instances to share a single virtual machine. Linux Operating System (OS) level virtualization creates isolated user-space environments, commonly referred to as containers (Pozdniakova & Mažeika, 2017a). These containers share the host OS resources, including CPU, network, storage, and memory. They share a single operating system kernel, which reduces performance overhead caused by context switching between processes. This allows multiple instances of the same application to run on the same compute host instead of spinning up multiple hosts per application instance. Containers also separate applications from the underlying host infrastructure, making deployment easier in different cloud or OS environments Figure 1.2.

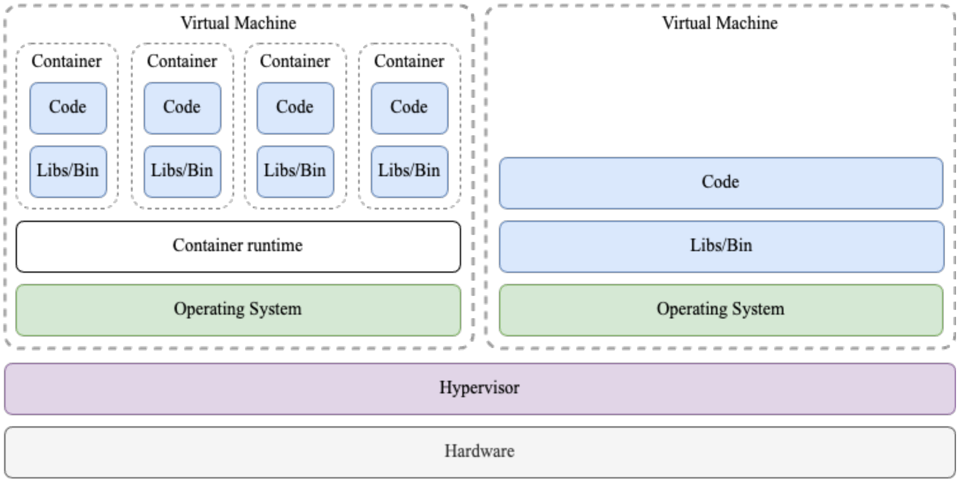


Fig. 1.2. Architectural differences between virtual machines and containers

The most adopted containerization technology is Docker. It was specifically designed to streamline application development and deployment, driving its widespread adoption in cloud environments. In 2015, Docker established the Open Container Initiative, which is an open governance structure that expresses the purpose of creating open industry standards around container formats and runtimes. The OCI currently contains three specifications: the Runtime Specification (runtime-spec), the Image Specification (image-spec), and the Distribution Specification (distribution-spec). Image specification defines the structure of the image and how to read the image file and content. To create a container image, developers define what should be included in the image (code, dependencies) and how this image should be built using a Dockerfile.

The Runtime Specification outlines how to run container images on machines. Distribution Specification defines an API protocol that facilitates and standardizes the distribution of content. The typical architecture of the container technology designed using Open Container Initiative (OCI) is presented in Figure 1.3.

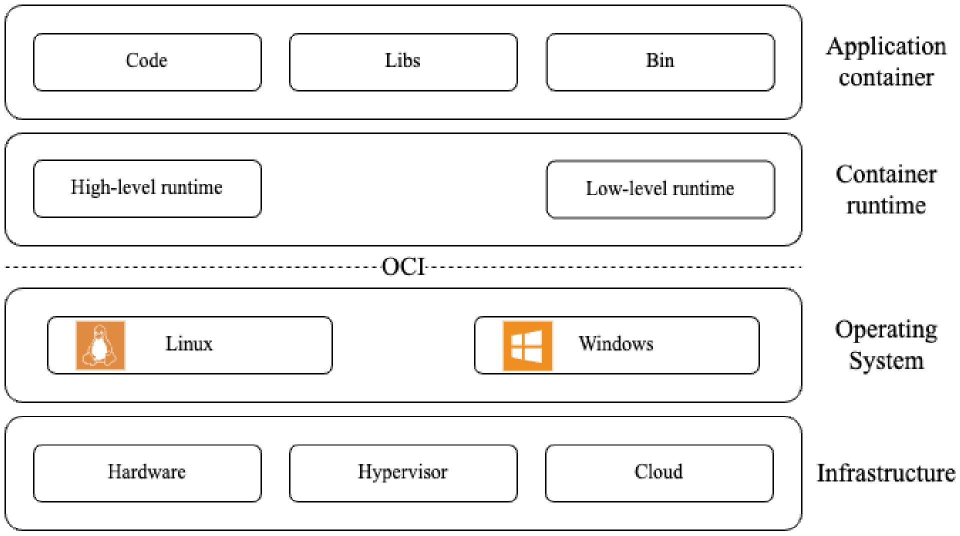


Fig. 1.3. Conceptual architecture diagram of a container

The solution standardization made containerization popular and brought platform and cross-cloud portability, making it ideal for cloud applications. This technology is crucial for maximizing the efficiency of computing resource usage and portability across different cloud platforms and services. However, it may not be the best fit for large monolithic applications that were not initially developed for the cloud.

The microservice architecture aims to build solutions using small services; such an approach enables more granular application elasticity. The operating system virtualization, together with application containerization technologies enforced by automated container management and orchestration solutions, serve as facilitators of the microservices architectural style (Peinl et al., 2016). The next subchapter briefly introduces this architectural style.

1.1.2. Microservice Architecture

The microservice architectural style was introduced in 2014. The concept of microservice architecture was popularized by Lewis and Fowler. They provided the following definition of this style: "The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and are independently deployable by fully automated deployment machinery. There is minimal centralized management of these services, which may be written in different programming languages and use different data storage technologies.”

The conceptual diagram of the example application built as a monolith and later decomposed into microservices is presented in Figure 1.4.

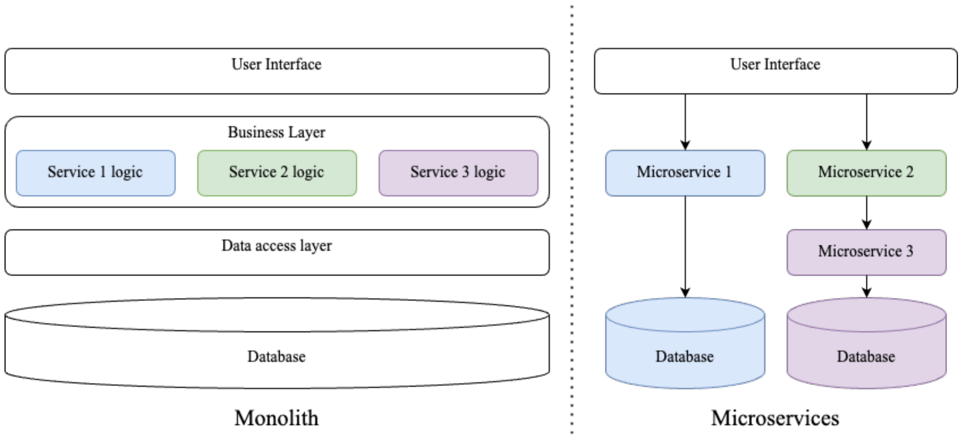


Fig. 1.4. Monolithic and microservice architectures

Commonly, microservice-based architecture is proposed as an alternative to monolithic application architecture (Pozdniakova & Mažeika, 2017b). MSA solves a set of problems like fault tolerance, speed of development, and horizontal scalability but also brings operational and organizational complexity compared to monolithic applications. Still, numerous articles highlight microservice architecture as a suitable approach for the development of a cloud-native application (Deng et al., 2024; Kazanavicius & Mazeika, 2019; Kosińska et al., 2024; Kratzke, 2018). The development and operational complexity of the microservice architecture makes it more suitable for large-scale application deployments or for delivery of Software as a Service (Leymann et al., 2016). From the viewpoint of service availability, when properly designed, MSA brings infrastructure cost savings via improved elasticity of the SaaS solution (Villamizar et al., 2017), minimizes the impact of a single service failure for the whole application availability, and provides faster service instance startup times compared to monolithic applications. Microservices can be scaled if and only if their associated load requires it (Stine, 2015).

When combined with application containerization, this architectural style becomes ideal for larger-scale applications, such as Software-as-a-Service (SaaS). As applications become larger and more intricate, it becomes crucial to dynamically spin up additional instances on time, matching the corresponding demand to provide the desired quality of service. This is where autoscalers come into play to address these issues. For the autoscaler to comply with quality of service requirements, it should be aware of the system and SLA states. The monitoring system provides information that is utilized by the autoscaler during the decision-making process.

The next sub-chapter provides an overview of SLA terminology used in this work. It also discusses monitoring used to measure compliance with SLA and to support autoscalers in autoscaling decision-making.

1.2. Service Level Measurement and Monitoring for the Cloud-Native Applications

The application and infrastructure monitoring tools allow measurement of service performance through the collection of application and infrastructure metrics (Pozdniakova, Mažeika, & Cholomskis, 2018b). These metrics can serve two purposes. First, metrics can be used as parameters for the qualitative measurement of the quality of service. These negotiated parameters are used as input to the Service Level Agreement (SLA) (Kosińska et al., 2024). Second, metrics collected by the monitoring system are used as input for autoscaling decision-making.

The following sub-chapters provide an overview of service level agreements, the use of metrics for service level measurement, and making autoscaling decisions in autoscaling solutions for cloud-native applications.

1.2.1. Service Level Agreements, Objectives, Indicators, and Scaling Indicators

The *Service Level Agreement* or SLA is an agreement between a customer and a provider to receive a service at a specific level. SLAs might include penalty clauses for service providers who fail to meet the pre-agreed Service Level Objectives.

Service Level Objective (SLO) represents a commitment to maintaining a service at a particular state in a given period (Keller & Ludwig, 2003). In other words, it defines what service performance target or goal the service must meet during the period defined in the contract, e.g., in a month. SLO defines the minimally acceptable level of service for each user (Beyer, Jones, Petoff, & Murphy, 2016; Sahal, Khafagy, & Omara, 2016). SLOs are divided into types by the SLA criteria they address, such as availability, performance, security, disaster recovery, resolution or response time, and others (Sahal et al., 2016).

SLOs are measured using *Service Level Indicators (SLIs)*, which is a service performance metric that indicates what measure of performance a customer is receiving at a given time (Beyer et al., 2016). In simpler terms, SLIs determine what aspects are being monitored to measure the service level or its quality, such as response time and error rate, as well as how these values evolve over time. The type of SLI heavily depends on the type of application, what task it performs (Amiri & Mohammad-Khanli, 2017) and what type of SLO it aims to achieve. For example, response time and requests per second are used to measure performance-oriented SLO, while Recovery Time Objectives are used to measure disaster recovery SLO. Response time and tail latency are the most commonly met SLI for microservices-based containerized cloud-native applications in analyzed literature, which is presented in Table 1.2.

As mentioned before, monitoring metrics are not only used to measure compliance with SLA but also used by autoscalers to make scaling decisions. These metrics are called *scaling indicators* (SI), that is, metrics used to make scaling decisions (Qu, Calheiros, & Buyya, 2018). To make scaling decisions, autoscalers utilize metrics collected from applications, infrastructure components, or both to understand the state of the system. These metrics can be split into two categories (Koperek & Funika, 2012; Taherizadeh, Jones, Taylor, Zhao, & Stankovski, 2018):

- *Low-level* metrics (also known as host or system) are collected from machines operating systems or containers. CPU, RAM, storage I/O utilization, and network throughput are examples of host metrics. These metrics are commonly used for making decisions about auto-scaling, as they are easy to collect and are independent of the applications running on a host.
- *High-level metrics or application* are application-specific metrics, such as response time, number of requests, and number of errors. These are typically used to monitor SLAs but can also be used to make auto-scaling decisions.

Selecting appropriate metrics is a complex task, as they must support the measurement of SLOs and provide sufficient information to detect and address any service level violations. Various monitoring approaches are discussed in the literature in this regard. As presented in Table 1.2, the solutions suggested by Casalicchio and Perciballi (2017); Hu and Wang (2021); Ruiz et al. (2022); Toka, Dobreff, Fodor, and Sonkoly (2021); Ye, Guangtao, Shiyou, and Minglu (2017) rely solely on low-level metrics. Contraversely, Amiri and Mohammad-Khanli (2017); Koperek and Funika (2012); Lorido-Botran, Miguel-Alonso, and Lozano (2014); Nikravesh, Ajila, and Lung (2017); Taherizadeh et al. (2018) suggested using only application metrics for SI (Table 1.2). Relining purely on low-level metrics could lead the SLA to be unfulfilled due to a risk of lack of performance degradation detection (e.g., increased response time) due to container migration to a host with a lower-performance CPU. On the other hand, relying purely on a high-level metrics approach provides awareness about service performance and compliance with performance-based SLO. However, it brings the potential risk of resource waste as there is no information about resource utilization, and as a result, it is hard to detect overprovisioning. To eliminate such blind spots, using high and low-level metrics is prevalent when autoscalers aim to fulfill SLA requirements and avoid resource waste (Casalicchio, 2019; Casalicchio & Perciballi, 2017).

When selecting appropriate SLA awareness metrics for autoscalers, it is important to consider the limitations of tracking average response time as a means to improve application performance. While this is a common approach discussed in literature (Khaleq & Ra, 2021; Pozdniakova et al., 2023; Taherizadeh et al., 2018; Wu, Yu, Lu, Qian, & Xue, 2019; Xu, Qiao, Wang, & Zhu, 2022), it may not provide sufficient information about non-compliant events with SLOs due to the averaging of values. The n^{th} percentile of response time, or tail latency, is a more informative metric, as it provides information about the percentage of events in which the response time value violated the SLO targets during the monitoring period. This information can also be used as an indicator of potentially upcoming SLO violations. In the case of the average response metric, the information relevant to SLO is lost due to the value-averaging process.

It is important to note that, except for the approaches suggested by Kumar and Gondhi (2018) and Pozdniakova et al. (2023), none of the previously mentioned solutions utilize SLO state tracking for autoscaling decisions. Monitoring the SLO state (whether it is within, below, or above the target) can help detect periods where the quality of service declines. This information is essential for identifying situations that require actions to recover the SLO or make improvements to meet the SLO. For example, to recover SLO, the system can add additional resources or adopt a less risky autoscaling strategy. Response time is a popular SLI. Typically, the autoscaling algorithms verify whether the current measurement of response time complies with the target defined in the SLA. If it does not, the scaling algorithms reactively adjust the number or placement of resources to restore the performance to the desired state and avoid further service degradation; that is, it enables *"SLA fulfillment failure avoidance"* mechanism.

Validating compliance with the target response time value ensures that the system allocates sufficient resources to meet the resource demand at this moment. However, relying solely on tracking current response time is insufficient to determine if the long-term SLO meets its target during the defined contract timeframe. In these situations, autoscaling algorithms must take into account the current status of the SLO (long-term) and respond effectively to restore it (SLO recovery). This concept can be illustrated with the following example. Imagine a system was functioning within an SLO target of 98%, but then its SLO dropped to 97% for an hour. To fix the situation, the system must operate at a minimum SLO of 99% for at least one hour, provided that the load and other factors affecting the SLO remain consistent with the previous hour.

Carefully selected SLI and SI are crucial in helping autoscalers achieve specific SLOs, such as performance, cost, and energy consumption. However, while metrics are important, an effective autoscaling solution also requires a well-designed autoscaling decision-making component to make the most of the metrics received from the monitoring system.

The following sub-chapter provides an overview of autoscalers, covering their objectives, techniques employed, resource management aspects addressed, and the types widely adopted in the cloud. It is important to note that this overview focuses on a subset of autoscaling techniques found to be most relevant to this work. More comprehensive descriptions are provided by Al-Dhuraibi et al. (2018); Amiri and Mohammad-Khanli (2017); Lorido-Botran et al. (2014).

1.3. Autoscaling of Containerized Cloud-Native Applications

Large-scale cloud-native applications consist of multiple small containerized instances, making manual provisioning of additional resources a complex task. The containers need to be optimally distributed across virtual machines or other container execution environments, and the required number of containers must be provisioned and deprovisioned based on resource demand. Container orchestrators are designed to simplify resource management tasks for containerized applications. They help users build, scale, and manage complex applications and oversee the lifecycle of a cloud-native application at runtime (Kosińska et al., 2024).

The container scheduler and the autoscaler are two key components that play crucial roles in container orchestration. The scheduler is responsible for placing containers across available nodes in the cluster and managing the life cycle of containers. Autoscaler is responsible for provisioning and deprovisioning compute resources based on defined metrics, policies, or rules (Ahmad, AlFailakawi, Al-Mutawa, & Alsalman, 2022). This dissertation primarily focuses on application autoscalers, which practically are container autoscalers (Deng et al., 2024). However, to provide a comprehensive overview of autoscaling techniques, some of the scaling techniques applied to virtual machines are included as they seem relevant to containerized applications' autoscaling.

Autoscaling solutions are designed to ease the task of adjusting resources to meet SLA requirements at any given time. To fulfill the SLA requirements, autoscalers aim to meet the following goals (Al-Dhuraibi et al., 2018; Amiri & Mohammad-Khanli, 2017; Lorido-Botran et al., 2014):

- solve *resource planning* problem – automated resource provisioning and application scaling must be done on time to avoid SLA violations;
- solve *resources utilization optimization* problem – the difference between provisioned and consumed resources should be as low as possible.

To achieve these goals, autoscalers use different approaches or modes (Al-Dhuraibi et al., 2018). Based on the autoscaling approach used by autoscalers, autoscalers can be divided into the following categories (Al-Dhuraibi et al., 2018; Amiri & Mohammad-Khanli, 2017; Lorido-Botran et al., 2014; Vazquez, Krishnan, & John, 2015):

- **The horizontal, vertical, and hybrid.** *Horizontal autoscalers* increase or decrease the number of compute resource replicas (containers, jobs, virtual machines) that run concurrently based on specific autoscaling rules or policies. *Vertical autoscalers* adjust the number of resources assigned to a single compute instance by increasing or reducing the allocated CPU power or memory. *Hybrid (also known as bi-directional) autoscalers* are a combination of both vertical and horizontal autoscalers.
- **Proactive or reactive.** *Proactive or predictive autoscalers* attempt to foresee future changes in the system by performing the necessary scaling actions before such changes occur. *Reactive autoscalers* scale resources when predefined rules are met or thresholds are exceeded. The scaling action is a reaction to a change in the system, and therefore, such autoscalers do not anticipate such a change Lorido-Botran et al. (2014). A combination of both is possible. However, no widely adopted specific term is available for such autoscalers.

Horizontal autoscaling is widely adopted by both infrastructure and application autoscalers. Vertical autoscaling is less adopted as it is more complex to implement, even though it interests academia (Baresi, Hu, Quattrocchi, & Terracciano, 2021; Ding & Huang, 2021; Nguyen, Yeom, Kim, Park, & Kim, 2020; Verreydt, Beni, Truyen, Lagaisse, & Joosen, 2019).

Reactive and proactive autoscalers are implemented using various techniques. Reactive autoscalers often use rules or policies (set of rules) to make an autoscaling decision (Lorido-Botran et al., 2014). Proactive autoscalers commonly use more advanced techniques, such as machine learning, queuing and network theory, or statistical analysis. As a result, autoscalers are also classified based on techniques used for autoscaling (Al-Dhuraibi et al., 2018; Lorido-Botran et al., 2014; Qu et al., 2018):

- *Rules or policy-based* – the autoscaling actions are performed when a specific trigger happens. Based on the trigger, the autoscalers are classified as:

  - *Schedule-based autoscalers* consider the cyclical pattern of the daily workload. The scaling actions are configured manually, based on the time of the day, so the system cannot adapt to the unexpected changes in the load.
  - *Threshold-based* autoscalers perform autoscaling action when a certain threshold or set of thresholds (policies) are met. The system monitors one or a set of metrics, such as CPU and memory utilization, requests per second, response time, and so forth, to perform an autoscaling action.
    - *Static threshold-based* rules or policies use fixed value thresholds or sets of thresholds (policies) to trigger autoscaling action.
    - *Dynamic thresholds*, or adaptive thresholds, are adjusted dynamically according to the state of the monitored system.
- *Time-series analysis* is utilized by autoscaling to estimate future workload or resource usage. A time series is a sequence of measurements taken at fixed or uniform intervals. Autoscalers use past measurements, a list of the last w (history window) observations of the time series, to predict future values and make decisions accordingly. Then, a suitable scaling action is taken based on the predicted value. The techniques can be further classified based on the methods applied to time series:
  - *Averaging methods* are used to smooth a time series to remove noise or to make predictions. The forecast of the desired resource or SLA metric value $y_t + 1$ is calculated as the weighted or unweighted average of the last w consecutive values.
  - *Machine learning (ML)* is an umbrella term that refers to a broad range of algorithms that perform intelligent predictions based on a data set (Nichols, Chan, & Baker, 2019). ML-based autoscalers commonly model the application behavior as a time series (create a mathematical model). According to the constructed model and the previous behavior of the application, the desired metric is predicted. There are multiple model-based algorithms used to support autoscaling decisions; their comprehensive overview is provided in Amiri and Mohammad-Khanli (2017); Lorido-Botran et al. (2014).
- *Reinforcement learning-based (RL)* autoscaler is a model-less machine learning-based autoscaler version. An autoscaler acts as a decision-making agent that gains knowledge (learns) through interactions between itself and the system or environment to get a maximal reward for a decision taken. For example, it will decide whether to add or remove resources to the appli-

cation (actions) depending on the current input workload, performance, or another set of variables (state) and always try to minimize the application response time or cost or maximize throughput (or another scalar reward).

- *Control theory-based* autoscalers use controllers as their decision-making mechanisms. Open-loop controllers, feedback controllers, feedback-forward, or a combination of feedback and feedback-forward can be used to make autoscaling decisions. Open-loop, also known as non-feedback, controllers make their decisions based on the input from the target system using the current state and its model. Feedback controllers monitor the action results to decide whether the system is working well or not and correct any deviation from the desired goal, while feedback-forward controllers predict system errors and react before they occur. The prediction may fail, and for this reason, feedback and feedback-forward controllers are usually combined.
- *Queuing theory based.* Autoscalers that adopt the classic queuing theory model each application as a queue of requests, using established methods to estimate required performance metrics or resources, considering the waiting time, arrival rate, service time, etc.

In addition to the previously mentioned classification, autoscalers are classified by the purpose or concern they aim to address. Autoscalers have different purposes, such as improving performance and ensuring availability, increasing resource capacity, saving energy, and reducing cost (Lorido-Botran et al., 2014). Amiri and Mohammad-Khanli (2017) categorized cloud resource management into two main aspects: resource-wasting avoidance and SLA fulfillment. As per Amiri and Mohammad-Khanli (2017), efficient resource management aims to address two main aspects:

- *Resource-wasting avoidance.* Its primary objective is to reduce costs or energy consumption. Decreasing energy usage results in lower carbon emissions, which can promote environmentally friendly cloud computing. A decrease in the amount of resources used leads to increased profit. Under these circumstances, a certain degree of SLA violations may be deemed acceptable if the cost or energy savings justify potential SLA penalties or align with other business objectives.
- *SLA-fulfillment* or Quality of Experience-oriented (Al-Dhuraibi et al., 2018) autoscaling solutions are designed to minimize SLA violations and ensure adequate resources to maintain system performance according to the defined SLO targets. The resources allocated to each application should be close to the application demand in a way that SLA is satisfied and resource wasting is minimized. However, in this case, justifying higher resource provisioning over the acceptable resource amount is reasonable.

Prioritizing customer satisfaction and availability takes precedence over cost or energy savings(Amiri & Mohammad-Khanli, 2017; Arapakis et al., 2021; Ilyushkin et al., 2018).

Autoscaling solutions cannot fulfill both aspects at the same time; each solution normally handles one aspect better than the other. However, solutions try to find an optimal way to balance some of the contradicted objectives (Lorido-Botran et al., 2014). Moreover, autoscalers can be classified based on their scope, which refers to the target at which autoscaler actions can be applied: either at the infrastructure or application/platform level (Al-Dhuraibi et al., 2018). This dissertation primarily focuses on application autoscalers, which practically are container autoscalers (Deng et al., 2024); however, it also includes infrastructure autoscalers where applicable to provide a comprehensive overview.

Table 1.2 provides an overview of autoscaling approaches that prioritize the efficient management of cloud resources while ensuring SLA fulfillment — Quality of Service (QoS) oriented autoscaling solutions. Additionally, these approaches are categorized based on their ability to prevent (avoid) or recover from SLO failures. Furthermore, this table includes solutions that focus on resource waste avoidance or simultaneously address both SLA fulfillment and resource waste avoidance aspects, offering a more comprehensive overview of the current state-of-the-art.

Such a wide variety of autoscaling solutions and approaches implies the difficulty in developing general-purpose autoscalers for cloud applications. Several factors, including diverse application resource and performance requirements, the non-homogeneity of cloud resources, dynamic workload characteristics, the timeliness of scaling decisions, and oscillation mitigation, must be considered when designing an atorscaler for cloud-native applications (Amiri & Mohammad-Khanli, 2017; Qu et al., 2018).

Different applications have different resource demands to perform as per SLA. To address this challenge, Sun et al. (2019), Mirhosseini et al. (2021), Ding and Huang (2021), Toka et al. (2021) employ application profiling in their solutions, which establishes a correlation between application performance and resource demand (Qu et al., 2018). Since cloud resources are dynamic, fluctuations in their performance may occur. For instance, when virtual machines with lower performance are provisioned to a pool of higher-performance machines or when “noisy neighbors" are placed on the same physical machine(Makroo & Dahiya, 2016; Sun et al., 2019). As a result, a new profile should be dynamically created once performance degradation is detected to avoid degradation of service quality; however, none of the above-mentioned works propose such an approach. To tackle the inconsistency of cloud resources performance and application profile change issues, dynamic thresholds-based solutions (Keller & Ludwig, 2003; Mirhosseini et al., 2021; Pozdniakova et al., 2023, 2024; Taherizadeh & Stankovski, 2019; Ye et al., 2017) or fuzzy Kalman filters (Sun et al., 2019) can be employed.

Table 1.2. Overview of QoS oriented autoscaling solutions (made by the author)

		Scaling indicators								Service level indicators						
Work	Autoscaling policy or mode	Resource utilization (RU)	Response time (RT)	Requests per second	Number of pods	SLO violations or state	Connections per second	Absolute RU	Pod startup time	Average RT	Tail latency	Number of pods	Throughput	Number of violations	SLO status	Cost
Aspect: SLA fulfillment (avoidance) (Amiri & Mohammad-Khanli, 2017)																
Hu & Wang, 2021	Time-series analysis (TSA)		x			x				x		x				
Abdullah, Iqbal, Berral, Polo, & Carrera, 2022	TSA, machine learning		x	x						x						
Ye et al., 2017	TSA, dynamic and static thresholds	x								x						
Khaleq & Ra, 2021	Dynamic thresholds, reinforcement learning	x	x							x						
Horovitz & Arian, 2018	Dynamic threshold, reinforcement learning				x						x					
Kang & Lama, 2020	Dynamic threshold, machine learning	x									x					
Taherizadeh & Stankovski, 2019	Dynamic thresholds, threshold-based policies	x	x						x	x		x				
Sun, Meng, & Song, 2019	Threshold-based policies, machine learning	x	x							x						

Continued Table 1.2

		Scaling indicators								Service level indicators						
Work	Autoscaling policy or mode	Resource utilization (RU)	Response time (RT)	Requests per second	Number of pods	SLO violations or state	Connections per second	Absolute RU	Pod startup time	Average RT	Tail latency	Number of pods	Throughput	Number of violations	SLO status	Cost
Aspect: SLA fulfillment (avoidance) (Amiri & Mohammad-Khanli, 2017)																
Casalicchio, 2019	Threshold-based policies							x		x						
Kumar & Gondhi, 2018	Threshold-based policies					x								x[a]		
Ruiz et al., 2022	Threshold-based policies	x								x			x			
Aspect: SLA fulfillment (recovery and avoidance)																
Pozdniakova et al., 2023	Dynamic thresholds, threshold-based policies	x		x		x	x			x					x	
Aspect: Resource waste avoidance																
Mirhosseini, Elnikety, & Wenisch, 2021	Queuing theory, dynamic thresholds	x	x	x			x				x					
Ding & Huang, 2021	Queuing theory, threshold-based policies	x	x	x	x					x						x
Beloglazov & Buyya, 2010	Dynamic threshold	x												x[b]		

[a]Number of RT violations

[b]Difference between requested and allocated millions of instructions per second for all virtual machines in scope

End of Table 1.2

		Scaling indicators								Service level indicators						
Work	Autoscaling policy or mode	Resource utilization (RU)	Response time (RT)	Requests per second	Number of pods	SLO violations or state	Connections per second	Absolute RU	Pod startup time	Average RT	Tail latency	Number of pods	Throughput	Number of violations	SLO status	Cost
Aspect: SLA fulfillment failure and resource waste avoidance																
Toka et al., 2021	Machine learning			x	x							x		x[a]		

[a]Number of lost requests

Cooldown is another widely employed mechanism used by autoscalers to coupe with load and performance oscilations (kubernetes.io, 2022; Lorido-Botran et al., 2014; Qu et al., 2018; Taherizadeh et al., 2018). The length of the cooldown period directly affects the timeliness of autoscaling decision-making (Nguyen et al., 2020). Zhang, Tang, Li, Khan, and Li (2019), in their work, has emphasized that the cooldown period should be long enough. Users must pay attention to the cooldown period; if it is shorter than the scaling period, it will lead to unpredictable scaling behavior, causing users to lose control of the autoscaler and rendering the set strategies ineffective.

Bursty and unpredictable load patterns may adversely affect the effectiveness of threshold-based solutions, as threshold-based solutions are commonly reactive in nature. Machine learning-based algorithms are effective in predicting loads that follow repeating patterns; however, they are less performant in case of unpredictable load patterns (Lorido-Botran et al., 2014). Abdullah et al. (2022) propose a method for predicting bursts using Decision Tree Regression.

The solution proposed in the second chapter and published in Pozdniakova et al. (2023) aims to tackle all the above-mentioned problems to fulfill SLA requirements. It addresses the issues by implementing different modules. It incorporates application profiling. However, it also utilizes several dynamically adjusted CPU thresholds to minimize the impact of load and infrastructure performance fluctuations and provide the required system performance. The dynamic thresholds are adjusted based on the speed of load changes. It introduces the velocity impact factor, a linear model used to determine how fast the load changes compared to the predefined performance baseline of the system. The solution also utilizes a traffic volatility detector to detect fluctuations and adjust autoscaling logic accordingly. By incorporating velocity impact factor and traffic volatility detector mechanisms, the solution aims to enhance the efficiency of handling bursty loads and maintaining SLA compliance. Furthermore, the solution implements SLA awareness mechanisms. It tracks the SLO status over the entire SLA monitoring timeframe and executes various actions based on whether the SLO needs to be restored or if the system is already compliant with the SLO. Notably, all this is done using policy- and rules-based autoscaling methods.

To summarise, the overview indicates that current rules-based autoscaling solutions and relevant scientific research have not yet provided a targeted approach to reduce SLA violations through adherence to SLO by employing the SLO recovery concept. The evaluated concept is applicable to embarrassingly parallel computing scenarios. It is important to emphasize that the notion of recovery, as discussed here, is distinct from the cost compensation strategies proposed by Khan, Chan, and Chua (2016); Qian et al. (2022), where autoscaling algorithms may opt to accept SLA violations if incurring penalties is deemed more financially viable than the costs associated with deploying additional resources.

This section provided an overview of types of autoscalers and the issues they aim to address using various methods. The next sub-chapter will give an overview of Kubernetes, which is the most widely used autoscaler in the industry (CNCF, 2024c; Datadog, 2024). It is also commonly encountered in work related to application autoscalers, and interest in autoscaling policies in Kubernetes has grown in recent years, specifically in Microservices, serverless, and edge computing application areas (Joyce & Sebastian, 2023).

1.4. Kubernetes

Since 2015, CNCF has offered support, oversight, and guidance for rapidly growing cloud-native projects. To support cloud-native application development, the CNCF landscape provides a list of cloud-native projects categorized by the problems the project aims to address. Container orchestration platforms are specifically created to handle containerized application deployment in large-scale clusters. Cloud providers have developed a variety of containerized application orchestrators, both cloud-dependent solutions, such as Azure Service Fabric and Amazon Web Services Elastic Container Service, and cloud-independent ones. The most widely adopted cloud-independent orchestrator is Kubernetes. Other less widely adopted cloud-independent orchestrators include Docker Swarm, Nomad, Red Hat OpenShift and Mesos, as well as the emerging projects Crossplane and Volcano orchestrator. However, altogether, these orchestrators did not become as popular as Kubernetes (CNCF, 2024b).

Originally developed at Google and released as open-source in 2014, Kubernetes is the most widely adopted container orchestration platform (85% of market penetration), and its adoption continues to grow (CNCF, 2024c; Raj, Vanga, & Chaudhary, 2022). The platform has become the *de facto* standard for container orchestration (Carrión, 2022). It is available for self-deployment on private and public clouds, and it is also managed by public cloud providers as a Platform-as-a-Service (PaaS). Examples of such managed services include Amazon Web Services (AWS), Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), and Google Cloud Platform Google Kubernetes Engine (GKE).

1.4.1. Kubernetes Architecture

A Kubernetes cluster consists of a control plane and worker nodes, which are responsible for running containerized applications (Fig. 1.5).

Each cluster must have at least one worker node. Every node in a Kubernetes cluster runs containers that make up the pods assigned to that node. Pods are the smallest computing units that can be created and managed in Kubernetes. A pod is a group of one or more containers with shared storage and network resources, along with a specification for running the containers. Containers in a pod are co-located and co-scheduled to run on the same node (virtual or physical machine).

Each worker node runs multiple components that maintain running pods and provide the Kubernetes runtime environment. The key components hosted by worker nodes are:

- *Kubelet* – an agent that runs on each node in the cluster and is responsible for node registration within the cluster. It ensures that containers are running in a pod.
- *Kube-Proxy* maintains network rules on nodes, allowing network communication to pods from network sessions inside or outside of your cluster.
- *Container Runtime* – the software responsible for running containers, such as Docker and containerd.
- *CNI* – the container network interface (CNI) is a software component responsible for allocating IP addresses to pods and enabling them to communicate with each other within the cluster.

The control plane is responsible for managing the worker nodes and pods in the cluster. Its components make global decisions about the cluster, such as scheduling, and also respond to cluster events, like starting up a new pod. Control plane components may run on worker nodes or have dedicated nodes that host only control plane components. The key components of the control plane:

- *API Server* serves as the front end for the Kubernetes control plane. It exposes the Kubernetes API, allowing users, management tools, and other components to communicate with the cluster.

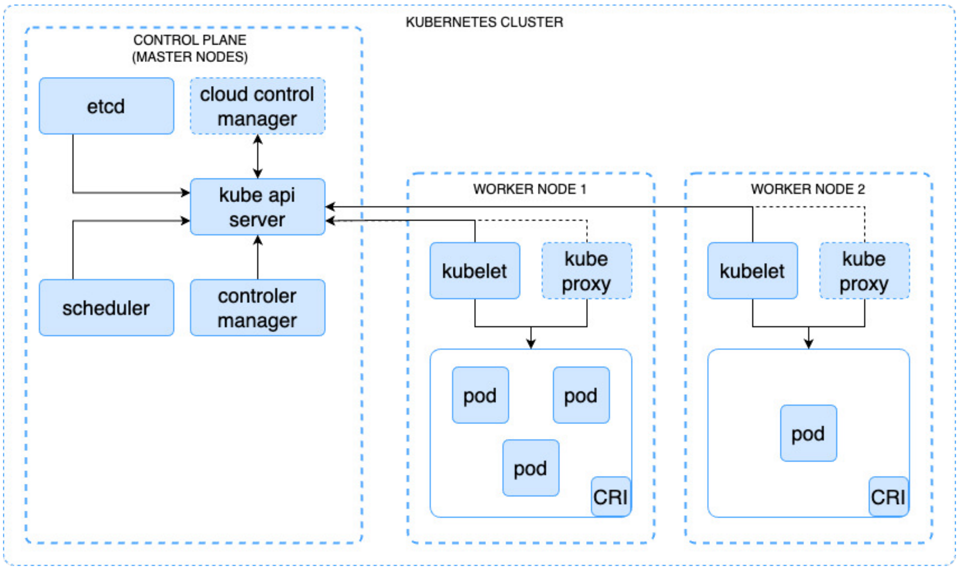


Fig. 1.5. Kubernetes architecture diagram (made by the author)

- *etcd* – a lightweight, distributed key-value store used to store all cluster data, providing a single source of truth for the state of the cluster.
- *Scheduler* watches for newly created pods with no assigned node and selects a node for them to run on based on resource availability, constraints, and affinity specifications.
- *Controller Manager* is responsible for ensuring that the cluster operates in a desired state. This is done through a set of different controllers, each of them performing specific functions.
- *Cloud Controller Manager* links the cluster to your cloud provider's API, allowing the cluster to interact directly with the cloud provider's infrastructure.

Different autoscalers can be implemented into Kubernetes. These solutions are presented in the next sub-chapter.

1.4.2. Autoscaling in Kubernetes

Kuberntes build-in (*Kubernetes-native*) container autoscaling solutions are provided within the Controller Manager. Nevertheless, Kubernetes' architecture allows for the implementation of *custom-built* autoscalers. Both Kubernetes-native and custom-built autoscaler approaches are subjects of dedicated academic research areas (Joyce & Sebastian, 2023; Mondal et al., 2023; Nguyen et al., 2020). This subchapter starts with Kubernetes-native solutions.

Kubernetes solution provides three different types of built-in autoscalers:

- *Cluster Autoscaler (CA)* is responsible for automated node provisioning and de-provisioning, reacting to pod demand for resources;
- *Vertical Pod Autoscaler (VPA)* is used for resource autoscaling within a pod, that is, setting the CPU and RAM resources used by individual pods;
- *Horizontal Pod Autoscaler* is the responsible adjustment of a number of pod replicas reacting to load or resource utilization changes.

This research is dedicated to the application of autoscalers. As a result, the CA autoscaler will not be discussed in more detail as it is an infrastructure autoscaler. VPA is not widely adopted as its current, implementation requires the restart of container instances to increase resources, which negatively impacts service availability and SLA (Dixit, Gupta, Dubey, & Misra, 2022; Nguyen et al., 2020). On the other hand, HPA is more popular due to its easy-to-understand configuration (Likosar, 2023), and it is the most adopted autoscaling solution provided by Kuberntes (Datadog, 2024). For this reason, the remainder of this work will primarily focus on HPA.

HPA utilizes threshold-based policies as an autoscaling method to optimize resource utilization and minimize infrastructure costs. The HPA method is based on determining the ratio between the desired metric value and the current metric value, as represented by Equation (1.1):

$$R_d = \left\lceil R_n * \frac{M_n}{M_d} \right\rceil, \tag{1.1}$$

here, R_d is the desired or target number of pod replicas, R_n is the number of current pod replicas ready to provide services, M_n – currently collected metric value, M_d – target utilization value or a threshold that operates as a trigger for autoscaling action.

HPA adjusts the number of resources allocated based on metrics such as CPU usage, RAM, or network throughput. When the current resource usage surpasses a defined threshold (referred to as the target utilization threshold), the HPA will increase the number of pod replicas. In contrast, if the resource usage falls below this target, the HPA will reduce the number of pod replicas. The configuration of the target threshold utilization is the key element that influences the application's performance, as it determines the resource allocation during each autoscaling event.

Even though the HPA is considered simple, it is not trivial to configure it (Huo, Li, Xie, & Li, 2022). For instance, in addition to the policy presented in Equation (1.1), HPA performance can be optimized through multiple parameters, such as stabilization window, autoscaling policies, and utilization threshold tolerance. The stabilization window is used to reduce frequent variations in the number of replicas caused by the dynamic nature of workloads. Additionally, Kubernetes autoscaling policies play a crucial role by governing the rate at which replicas can be added or removed and establishing the maximum and minimum thresholds for the number of replicas during the specified provisioning window. The threshold tolerance setting ensures that minor fluctuations in resource usage, whether above or below the defined limits, do not lead to unnecessary autoscaling actions, which can create oscillations. In essence, the stabilization window and tolerance settings are designed to maintain the stability of the HPA, while the scaling policies aim to prevent high overprovisioning or underprovisioning in the face of significant changes in resource demand between autoscaling events.

The desired utilization threshold setting is the most impactful factor in controlling the application's performance per the defined SLO. As presented in Equation (1.1), this setting controls the amount of resources to be provisioned or deprovisioned during each autoscaling action, thus ensuring that the application meets its performance-based SLO. However, determining the threshold that would allow satisfying performance-based SLOs is a long, error-prone manual process.

In addition to multiple parameters to be adjusted, the solution has its limitations. By default, the Kubernetes-provided resource metrics are limited to the CPU and memory usage of pods and host machines (low-level metrics). Therefore, Kubernetes is not able to detect degradation in quality of service related to application performance, such as response time, without integration with a non-Kubernetes-native monitoring system, such as Prometheus.

As can be seen, many configuration parameters exist in HPA, so academia has made several attempts to support users in the HPA adoption. Nguyen et al. (2020) aims to support researchers and practitioners by providing recommendations on operating HPA. Casalicchio (2019) analyzed the impact of using absolute and relative metrics in Kubernetes autoscaling. Huo, Li, Li, Xie, and Li (2023) proposed a strategy that sets a higher tolerance value for utilization thresholds than the Kubernetes default one. The approach minimizes the number of timeout requests in high concurrency load scenarios compared to the default HPA settings. All of these recommendations and adjustments must be implemented manually, and there are no guidelines on how to select the optimal values for each of the parameters.

The solution suggested by Huo et al. (2022) is more specific in providing recommendations. The suggestion is to minimize the stabilization window to 0s when performing upscale actions and extend stabilization windows for downscale actions to up to 9 minutes to minimize resource waste. This work shows the impact of stabilization window length on the ability to react faster to load spikes compared to default stabilization window settings. However, it is unclear how efficient the strategy is for the risk minimization of SLO violations and how it would behave under various load conditions.

One of the latest works that aim to dynamically optimize the performance of HPA and address resource waste avoidance is Augustyn, Wyci´slik, and Sojka

(2024). Augustyn et al. (2024) suggested the approach for identifying the maximum number of pods to be provisioned by HPA. The approach allows customers to continue using HPA while improving resource utilization. The work does not aim to ensure that system performance conforms with the SLO.

Khaleq and Ra (2021) recognized the challenge of identifying the right values for the different autoscaling parameters that will guarantee QoS in a changing dynamic environment. As a result, they propose the multi-component system for intelligent autoscaling, which aims to adjust thresholds for the HPA reactively to maintain the application performance as per the defined SLI, such as average response time. The system uses application profiling and reinforcement learning components. The experiment results showed up to 20% improvement in response time compared to default HPA. It also showed that training and validating RL agents to identify threshold values for autoscaling can potentially satisfy the QoS of response time. However, the authors use application performance traces to simulate the performance of the proposed algorithm using MATLAB. So, the efficiency of the solution has not been evaluated in a real infrastructure environment and is left as an open area for future development and research.

Table 1.3 summarizes the differences between the works mentioned above. It lists the solutions that aim to improve HPA performance by addressing different resource management aspects and highlights whether the improvements should be applied to HPA manually or automatically. The table displays whether the suggested approaches were tested in real infrastructure or simulated environments. As indicated in the table, the dynamic threshold adjustment for HPA is a vaguely investigated area, even though it is the most influential parameter when talking about resource provisioning to satisfy the performance-based SLA (Qu et al., 2018).

Before suggesting any improvements in dynamic threshold adjustment, it is worth analyzing the current state of the art of custom-built autoscalers for Kubernetes that use dynamic threshold policies.

Dynamic Thresholds Adjustment in Custom Autoscalers

The static threshold determination process requires an understanding of the application characteristics and expert knowledge to determine the thresholds for proper actions (Qu et al., 2018). If thresholds are set too low, overprovisioning of resources can occur; however, this would allow for a quicker response to load changes and, as a result, improve application performance. Conversely, choosing a threshold that is too high may lead to fewer replicas being provisioned and leaving no buffer for detection and reaction to load increase (Developers, n.d.). These two factors may cause a decline in performance and an increased risk of failing to meet the application performance SLOs (Sahal et al., 2016).

Huo et al. (2022) and Lorido-Botran et al. (2014) find that static threshold-based autoscalers are slow to react. The utilization threshold must be set lower

Table 1.3. Overview created by the author, summarizing various efforts that propose enhancements for configuring Kubernetes Horizontal Pod Autoscaler

Authors	Enhancement/ Strategy	Adjustment	Resource management aspect	Test environment
Huo et al., 2022	Stabilization windows length setup	Manual	Resource waste avoidance	Real infrastructure
Huo et al., 2023	Setting up tolerance threshold	Manual	SLA-fulfillment	Real infrastructure
Augustyn et al., 2024	Maximum number of pod replicas determination	Automated	Resource waste avoidance	Real infrastructure
Khaleq & Ra, 2021	Dynamic utilization thresholds adjustment	Automated	SLA-fulfillment	Simulation in MAtlab
Pozdniakova et al., 2024	Dynamic utilization thresholds adjustment	Automated	SLA-fulfillment	Real infrastructure

to allow time for a reaction to an increase in load and wait for new replicas to be provisioned. This buffer is essential for ensuring that the system can cope with sudden changes in demand (Developers, n.d.); however, it is unclear how to estimate the size of such a buffer.

The threshold determination becomes even more challenging as the cloud environment is not homogeneous. The non-homogeneity causes inconsistency in resource provisioning (Al-Haidari, Sqalli, & Salah, 2013; Khaleq & Ra, 2021; Rzadca et al., 2020). "Noisy neighbors" are another problem that causes inconsistent performance of provisioned resources (Balla, Simon, & Maliosz, 2020; Makroo & Dahiya, 2016)). Additionally, cloud-native applications are constantly updated and redeployed, requiring dynamic updates to autoscaling thresholds. As a result, finding a suitable threshold is a challenging process.

Table 1.4 summarizes the research results on custom-built rules-based autoscaling solutions for Kubernetes. The solutions are categorized based on the timing of the decisions made (reactive and proactive autoscaling), the indicators (utilization, response time, etc.), and the methods used for autoscaling. Most studies use machine learning to predict the required thresholds proactively.

The dynamic threshold adjustment problem has been investigated by academia for a long time. As presented in Table 1.4, the oldest work analyzed in this regard is Beloglazov and Buyya (2010). The authors have introduced a set of heuristics designed to dynamically adjust thresholds through the statistical analysis of historical data collected throughout the lifespan of virtual machines (VMs). Rather than service quality, this algorithm's primary focus is on reducing power consumption during the live migration of VMs.

Table 1.4. Overview of the characteristics of autoscalers employing dynamic threshold adjustment algorithms (made by the author)

Authors	strategy	Scale indicators	Methods	Service level indicator	Resource management aspect
Beloglazov & Buyya, 2010	Proactive	Node CPU utilization	Rules-based	Requested versus allocated resources	Resource waste avoidance
Kang & Lama, 2020	Proactive	Resource utilization	Gaussian process regression	Tail latency	SLA fulfillment
Horovitz & Arian, 2018	Proactive	Number of resources	Rules-based, reinforcement learning	Tail latency	SLA fulfillment
Pozdniakova et al., 2023	Reactive	Resource utilization	Rule-based	Number of violations	SLA fulfillment
Pozdniakova et al., 2024	Reactive	Resource utilization	Rules-based	Number of violations	SLA fulfillment
Khaleq & Ra, 2021	Reactive	Resource utilization	Rules-based, reinforcement learning	QoS: response time	SLA fulfillment
Mondal et al., 2023	Proactive	CPU utilization	Machine learning	Number of pods	Resource waste avoidance
Taherizadeh & Stankovski, 2019	Reactive	Resource utilization	Rules-based	Response time	Resource waste avoidance

Horovitz and Arian (2018) adapt the Q-learning algorithm of reinforcement learning to identify the utilization threshold. The authors acknowledge that the adoption of Q-Learning is limited due to various challenges. They propose an approach to streamline the application of Q-Learning for threshold adjustments by choosing a state space that represents the current resource allocation and an action space that encompasses an action for each utilization threshold value. The solution is a custom build threshold-based autoscaler, which triggers autoscaling actions within the Kubernetes environment. The HPA serves only as a benchmark for performance assessment in the study, with the authors manually adjusting the thresholds for the HPA in each experiment. While the results indicate that their solution outperforms the HPA in resource utilization by 52%, it remains unclear whether the utilization thresholds set for the HPA ensured the SLO achievement.

Kang and Lama (2020) proposes an autoscaler that predicts the end-to-end tail latency of microservice workflows using the Gaussian Process Regression (GP) model. The model determines the threshold by leveraging predicted tail latency alongside historical resource utilization data corresponding to a specific tail latency value. The evaluation results of RScale have shown that the proposed system can meet the defined SLOs (e.g., tail latency) even in case of varying interference and evolving system dynamics. However, there is no clarity regarding the solution's efficiency in terms of resource consumption. RScale is a standalone custom autoscaler that adjusts its thresholds, and it is not part of the HPA.

The developers of the adaptive scaling solution for Kubernetes pods, referred to as Libra (Balla et al., 2020), have created a custom autoscaler that links the number of requests a pod can handle in compliance with SLO to actual CPU utilization. This approach enables the system to dynamically detect the CPU resource limits of the pod and horizontally scale the application when the request count reaches 90% of the maximum requests in relation to the actual CPU utilization value. The authors claim that this method provides better performance than the HPA. However, it is important to note that Libra does not monitor the number of SLO violations, leaving it unclear whether it effectively meets SLO requirements throughout the SLA monitoring timeframe.

Taherizadeh and Stankovski (2019) presented a solution called Dynamic Multi-level Autoscaling Rules (DMAR). However, the solution utilizes low-level and high-level monitoring data for threshold adjustment. The solution employs average response time and CPU or memory utilization as key metrics for modifying CPU thresholds. The authors of DMAR evaluated their approach against seven widely used rules-based autoscaling methods. The findings revealed that DMAR was the most efficient solution regarding resource usage while ensuring an acceptable level of QoS across all of the evaluated solutions.

The Self-adaptive Autoscaling Algorithm (SAA) for SLA-sensitive applications described in the second chapter and published in Pozdniakova et al. (2023) focuses on ensuring SLA fulfillment through the use of dynamically adjusted thresholds. Although it does not explicitly address the threshold adjustment problem as the primary goal, it employs dynamic thresholds to achieve the SLA-fulfillment goal. The solution uses multiple thresholds, which are adjusted by SAA based on the state of SLO fulfillment, making it unique compared to the solutions assessed in this sub-chapter.

1.5. Autoscaler Performance Evaluation Methods

Before proceeding with the analysis of performance evaluation methods for autoscalers, it is important to note that the studies examined in this research were assessed from two perspectives: efficacy and efficiency. Efficacy refers to the ability of the proposed solution to produce the desired outcome, such as meeting SLAs. On the other hand, efficiency measures how effectively the solution can achieve its goals while using the least amount of resources possible.

The authors of the solutions examined in this study have employed diverse criteria to evaluate the effectiveness of their proposals. Notably, some of the solutions that do not rely on high-level metrics as scaling indicators utilize response time as a measure of efficacy (Horovitz & Arian, 2018; Kang & Lama, 2020; Ye et al., 2017). Solutions that use predictive techniques commonly use the ML models evaluation metrics to evaluate the prediction effectiveness of the algorithm (Amiri & Mohammad-Khanli, 2017). Other criteria met in these works for evaluating efficacy include cost optimization (Ding & Huang, 2021), throughput improvement (Taherizadeh & Stankovski, 2019), total pod uptime (Taherizadeh & Stankovski, 2019; Toka et al., 2021), requests loss reduction (Toka et al., 2021), or the number of running pods (Hu & Wang, 2021). Most of these metrics are used to measure compliance with SLA or SLI. While the level of service delivered by the application is measured using SLI, which represents the performance characteristics of the application, such SLI is not representative in identifying how efficient the autoscaling solution is (Herbst et al., 2013).

Herbst et al. (2016) provided the definition of elasticity and outlined a method for elasticity measurement. They suggest specific metrics for benchmarking and recommend using demand, accuracy, timeshare, and jitter metrics for this purpose. Demand refers to the minimum resources needed to meet a specific performance-related service level objective, while accuracy is determined by the average difference between the demanded and provisioned resources. The timeshare metric indicates the amount of time that provisioned resources spend in overprovisioned or underprovisioned states, and the jitter metric measures the variance between the actual and required resource adaptations performed by the autoscaler. The graphical illustration of demand is presented in Figure 1.6.

The authors also propose an approach for evaluating autoscaling methods using a composite metric called elastic speedup. This method allows for comparing platforms without assessing each elasticity metric separately. Users can assign different weights to each elasticity metric when calculating the elastic speedup metric. Finally, this metric can be used to compare the elasticity levels of different autoscalers against each other.

Zhang et al. (2019) evaluates the Mesos autoscaler elasticity (E) by measuring the deployment speed of containers. The deployment speed of containerization is measured as a ratio between the average number of containers (S) and the average time spent to provision and deprovision containers during scaling action (T). The elasticity is calculated as the total number of containers (S) divided by the time spent (T) during the whole experiment ($E = S \div T$).

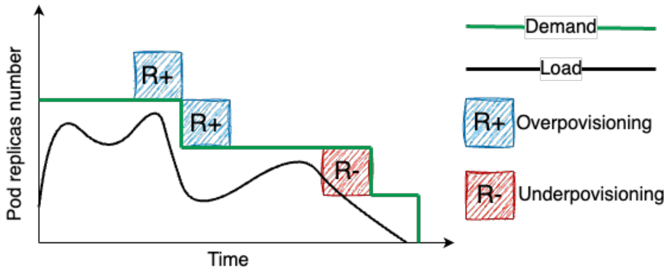


Fig. 1.6. Illustration for the definition of demand and supply in elasticity evaluation (made by the author)

Both approaches enable users to evaluate the efficacy of autoscaling solutions based on resource utilization and timelines for resource provisioning; however, both solutions consider how the difference in the number of provisioned resources and delay impact service levels.

To evaluate autoscaler efficacy for SLA compliance, Podolskiy, Jindal, and Gerndt (2019) proposed a multi-layer autoscaling evaluation involving containers and VMs. They also proposed a performance evaluation for single-layered autoscaling using the following metrics:

- *Autoscaling latency* – the time between the decision to autoscale and when the desired number of replicas is ready to handle the load. This metric characterizes how effectively the autoscaler reacts.
- *Required response time (RRT)* – the user-side metric for measuring response time violations.
- *Required maximal failure rate (RMFR)* – a user-side metric for measuring maximal failure rate violations.

These metrics evaluate the relationship between the time taken to initiate and complete an autoscaling action and the increase in response time or failure rate higher than a predefined threshold caused by resource provisioning delay, which causes resource starvation. This relationship is expressed as $t_{starvation}/t_{autoscaling}$. In general, it demonstrates the proportion between the time taken to autoscale and the time when the violation of response time or other quality of service metrics was occurring. The authors state that this approach provides more comprehensive metrics for evaluating autoscaler performance. While this method offers valuable insights into how resource provisioning delay affects application performance, it doesn't provide as much information about autoscaler efficiency, such as over- and under-provisioning of resources. It also doesn't quantify the impact of resource starvation on the achievement of the overall SLO. For instance, losing 1% of requests in one minute has a much smaller impact than losing 90% of requests.

It is worth noting that the various methods for measuring autoscaler efficiency mentioned above do not consider that different applications may have distinct container startup times due to the specifics of the frameworks or languages used to write them. Therefore, the duration of upscaling actions is not always within the control of the autoscaler, as it is influenced by the design of the application (Herbst et al., 2013). Consequently, these approaches are more suitable for comparing different autoscalers when testing is conducted using the same application and infrastructure with similar parameters. It is also worth noting that none of the analyzed work provides a method for quantitatively measuring the efficacy of autoscaling solutions in meeting SLA compliance requirements.

An additional observation from the literature review on autoscaler efficiency evaluation is that the HPA is frequently employed as a baseline for comparative analysis. However, various studies utilize different methods for determining HPA target thresholds, and they often lack clear explanations for selecting specific thresholds. This absence of a consistent method for establishing HPA baseline thresholds complicates the assessment of evaluation results and their validity.

1.6. Conclusions of the First Chapter and Formulation of the Tasks of the Dissertation

The first chapter of the dissertation provides an overview of cloud-native applications, autoscaling solutions, and evaluation methods for the efficiency of autoscaling solutions. The following conclusions have been drawn:

1. Cloud-native applications are specifically designed to utilize the characteristics of the cloud, such as elasticity and automation. These applications prioritize scalability and loose coupling, and they make use of containerization technology. The microservice architecture is frequently associated with cloud-native application designs, as it enables greater scalability of the application. Microservice applications are commonly deployed in the form of multiple instances of containers. Container orchestration solutions simplify operations of large-scale deployments of containerized applications. Autoscaling is responsible for automatically adjusting resources to meet application performance needs. While there are several proposed approaches and solutions for autoscaling, starting from simple rule-based solutions and ending with multi-model machine-learning-based approaches, none fully address issues such as timely resource provisioning and accurately determining the required resources to meet performance needs.
2. Autoscaling solutions are usually developed to address specific business challenges, such as reducing costs, conserving energy resources (resource waste avoidance), or enhancing customer satisfaction by meeting SLAs (SLA fulfillment). While resource waste avoidance-oriented autoscalers

prioritize efficiency in resource management, SLA-fulfillment-driven autoscalers prioritize effective resource provisioning to ensure high-quality service, even if it requires additional resources.

3. The effectiveness of an autoscaling solution in efficiently managing resources and fulfilling SLAs strongly depends on how the SLA is measured. Most approaches consider high-level metrics, with response time being the most popular. Some strive to ensure SLA compliance by relying solely on low-level metrics. However, it is important to consider both types of metrics to avoid overlooking important factors in autoscaling decisions. Another common oversight is relying solely on SLI thresholds, like response time, to determine compliance with SLOs. Simply restoring the current SLI measurement to the defined level and initiating performance recovery does not guarantee that SLOs will achieve the agreed-upon level for the entire service level measurement period. Therefore, solutions should incorporate SLO awareness to facilitate mechanisms for recovering the SLO state, where possible, to attain the desired level of application performance during SLO measurement over a specific duration. Analyzed autoscaling solutions lack such capability.
4. Two main problems were identified for autoscaling solutions that directly impact the SLA: the timelines of resource provisioning and resource planning, which ensures that the resource supply meets the demand in a timely manner. To overcome these two problems, autoscalers must address the adaptivity to the platform and application resource changes, dynamic workload characteristics and timeliness of scaling decisions, and oscillations mitigation concerns. Academia has developed and proposed multiple cus–tom-built autoscaling solutions to address these challenges. The solutions range from simple rules-based policies to advanced machine-learning models. Proactive autoscalers are effective in addressing timelines of resource provisioning, as commonly, those use advanced techniques such as statistical analysis and autoscaling. However, due to complexity, most ML-based solutions are proprietary and commercialised; as a result, there is no widely adopted solution that would use these techniques for application autoscaling. Easier-to-understand rule-based solutions are more prevalent for non-com-mercial use, with HPA being the most adopted autoscaler and widely discussed in academia.
5. The most commonly used autoscaling solution is a rules-based reactive autoscaler called Kubernetes Horizontal Autoscaler. Its widespread adoption is due to its ease of understanding compared to ML-based solutions, straightforward implementation, and seamless integration with the Kubernetes orchestrator. However, it lacks SLA awareness and has limited performance SLA fulfillment mechanisms. The HPA involves multiple pa-

rameters for manual adjustment, which can be error-prone. Thus, a new research area focused on enhancing existing autoscaling solutions like HPA rather than creating entirely new autoscalers. Despite not being originally designed as an SLA-aware autoscaler, there is potential for improvement in this area.

6. While various assessment methods have been suggested to evaluate autoscaler behavior, none of them explains how to measure the solution's efficacy from the perspective of a performance-based SLA.

Based on the conclusions, the following tasks are formulated to achieve the goal:

1. Propose and evaluate an SLO-aware autoscaling solution for containerized cloud-native applications.
2. Introduce SLA awareness and adoption mechanisms to Kubernetes native autoscaler and evaluate its performance.
3. Propose or improve existing evaluation methods for SLA-aware autoscaling solutions for containerized cloud-native applications.

2 Design of Service Level Agreement-Aware Autoscaling Algorithms

The chapter discusses two rule-based autoscaling solutions that use autoscaling algorithms designed to be aware of Service Level Agreements (SLAs) and to adapt dynamically to the SLA requirements. The first is a custom autoscaler implementation to overcome multiple challenges of traditional rule-based systems in ensuring compliance with Service Level Agreements (SLAs). The second solution is an add-on that improves static rule-based autoscalers using SLA-aware threshold policies for scaling decisions. The chapter provides a comprehensive overview of the proposed SLA-aware autoscaling solutions, detailing the algorithms and functions used and the evaluation criteria used to assess the solutions. Additionally, it explains the rationale for favoring threshold-based autoscalers over those that rely on machine learning.

On the topic of this chapter, 2 publications were published by the author in international journals (Pozdniakova et al., 2023, 2024).

2.1. Development of a Service Level Agreement-Adaptive Autoscaling Algorithm

This sub-chapter provides an overview of the proposed SLA-Adaptive Autoscaling Algorithm (SAA), which aims to ensure compliance with the service level objectives. The solution employs policy- and threshold-based autoscaling methods and aims to address multiple challenges that must be resolved to ensure SLA fulfillment (Pozdniakova et al., 2023). The next sub-chapter details the rationale for selecting a rules-based autoscaler over machine learning alternatives, despite the reactive nature of rules-based decision-making.

2.1.1. Motivation for Developing a Rules-Based Autoscaler Solution

Hu and Wang (2021), Hu and Wang (2021), Abdullah et al. (2022), Ye et al. (2017), Khaleq and Ra (2021), Sun et al. (2019), Mirhosseini et al. (2021), Ding and Huang (2021), Toka et al. (2021), Ju, Singh, and Toor (2021) proposed advanced prediction techniques, such as machine learning (ML) or statistical analysis, to ensure SLA fulfillment. However, the implementation and maintenance of machine learning models can be complex, and there is no guarantee that the solution will successfully meet the defined SLOs. As a result, the use of simpler, threshold-based policies is prevalent despite their drawbacks, such as the difficulty in determining suitable threshold values and their reactive nature (Al-Dhuraibi et al., 2018), which carries the risk of increased SLO violations. As an illustrative example, the most widely adopted autoscaling solutions are rules-based reactive autoscalers, such as a Kubernetes Horizontal Pod Autoscaler, or autoscalers employed by cloud providers for their compute solutions, such as Amazon Web Services (AWS) Elastic Compute (EC2) or Azure Virtual Machines scale sets. Its wide adoption can be explained by the fact it is easier to understand and adopt than ML-based solutions. Custom autoscalers are low in adoption, so bringing improvements related to SLA compliance to the existing widely adopted solution increases the possibility of the solution being adopted. The next sub-chapter presents the proposed SLA-aware autoscaling algorithm design.

2.1.2. Service Level Agreement-Aware Algorithm Design

The SAA algorithm aims to address issues related to diverse application resource and performance requirements, non-uniformity of cloud resources, dynamic workload characteristics, timeliness of scaling decisions, and oscillation mitigation highlighted in the first chapter to fulfill autoscaling SLA requirements. To address these issues, SAA comprises several modules, and their design follows the single responsibility principle (Martin, 2017). Figure 2.1 provides an overview of the modules, the functions each module performs, and their relationships. Each component is responsible for a specific function:

- *The Autoscaler* module is the main decision-making component of the system, as it executes the autoscaling algorithm.
- *The Dynamic CPU Thresholds Adjuster* (DCTA) dynamically adjusts CPU thresholds in response to the current SLO state to reduce the effects of un-

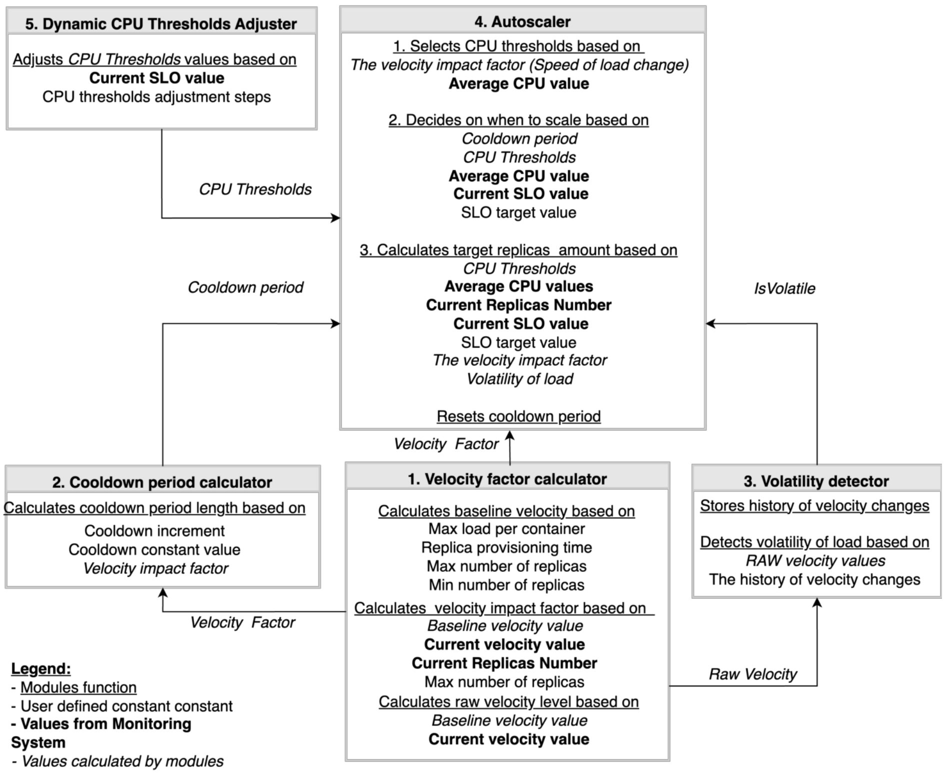

5. Dynamic CPU Thresholds Adjuster
Adjusts CPU Thresholds values based on
Current SLO value
CPU thresholds adjustment steps
4. Autoscaler
1. Selects CPU thresholds based on
The velocity impact factor (Speed of load change)
Average CPU value
2. Decides on when to scale based on
Cooldown period
CPU Thresholds
Average CPU value
Current SLO value
SLO target value
3. Calculates target replicas amount based on
CPU Thresholds
Average CPU values
Current Replicas Number
Current SLO value
SLO target value
The velocity impact factor
Volatility of load
Resets cooldown period
CPU Thresholds
Cooldown period
IsVolatile
Velocity Factor
2. Cooldown period calculator
Calculates cooldown period length based on
Cooldown increment
Cooldown constant value
Velocity impact factor
1. Velocity factor calculator
Calculates baseline velocity based on
Max load per container
Replica provisioning time
Max number of replicas
Min number of replicas
Calculates velocity impact factor based on
Baseline velocity value
Current velocity value
Current Replicas Number
Max number of replicas
Calculates raw velocity level based on
Baseline velocity value
Current velocity value
3. Volatility detector
Stores history of velocity changes
Detects volatility of load based on
RAW velocity values
The history of velocity changes
Velocity Factor
Raw Velocity
Legend:
- Modules function
- User defined constant constant
- Values from Monitoring System
- Values calculated by modules

Fig. 2.1. Modules of the SLA Adaptive Autoscaler solution and their relations diagram

derlying infrastructure performance variations and ensure the required system performance for meeting SLA fulfillment and compliance restoration objectives.

- *The Velocity Factor Calculator* utilizes velocity as a means to characterize the load processed by the workload.
- *The Volatile Traffic Detector* prevents autoscaling decisions from repeating load fluctuation patterns that could harm the quality of service. It monitors sudden shifts in velocity to detect volatility.
- *The Cooldown Period Calculator* adjusts the duration of the cooldown period depending on the load velocity. A higher velocity results in a more extended cooldown period for downscale and a shorter one for upscale actions.. This ensures autoscaling actions are triggered sooner in case a sudden load increases and avoids sudden removal of resources in case of an instant load drop.
- *The value processing and storage* gathers, pre-processes, and stores the application and infrastructure metrics needed for the SAA solution. It is

used for the collection and initial processing of metrics to ensure they are prepared for further analysis and decision-making by the SAA solution. Additionally, the module enables the exchange and storage of metric values between different modules within the system.

To improve the readability of this chapter, Table 2.1 introduces parameters used later in this work – their notation and description. The SLO presented in Table 2.1 is calculated in alignment with the SLO description provided by (Quach, 2020; Sahal et al., 2016).

$$SLO = \frac{good\_events}{valid\_events} * 100\%, \tag{2.1}$$

here,

- $good\_events$ refers to the count of monitored events where the SLI values, such as response time, meet a specified target, e.g., response time value must be less than 3 seconds;
- $valid\_events$ represents the total count of all monitored events of the respective SLI.

Subsequent sub-chapters explain the modules responsible for preparing the data required by the Autoscaler module to make autoscaling decisions starting from the Velocity Factor Calculator module.

Velocity Factor Calculator

To make timely scaling decisions and ensure SLA fulfillment, an autoscaling solution must be capable of responding to changes in load patterns by providing adequate resources required to comply with SLO. The SAA solution utilizes velocity $v_n(L)$[1] (Equation (2.2)) to characterize the load sent to the workload (L).

$$v_n(L) = \frac{|L_{n-1} - L_n|}{T_m}, \tag{2.2}$$

here, L_n represents the most recent value of the total throughput or another load metric, such as the number of concurrent users or requests per second.

The autoscaling algorithm must detect moments when the workload's velocity is increasing faster than expected. The Velocity Factor Calculator module classifies velocity into high, moderate, or stable levels using the below-describe approach. The velocity impact factor[2] (α_D) is calculated to measure the velocity, that is, how rapidly the velocity changes. The velocity factor is determined by comparing the current velocity value with the baseline velocity ($v_{baseline}$). By evaluating the ratio

[1]Here and throughout the document, the index n represents the sequence number of the monitoring sample.

[2]Also referred to as the velocity factor.

Table 2.1. Description of SLA Adaptive Autoscaler algorithm parameters

Parameter	Description
R_{max}, R_{min}	*The highest (max) and lowest (min) number of replicas* that can be provisioned by autoscaler.
R_d^{Da}	*The desired number of replicas* is the number of pod replicas calculated based on the resource utilization metrics.
R_{tgt}^D	The *target number of replicas* refers to the number of pods determined based on various factors, including the velocity of load, SLO compliance state, or traffic volatility.
R_{BP} ΔR_V	The number of replicas provisioned when the current SLO value SLO_n is way below target. *Downscale step when the load is volatile* is the number of replicas that will be removed from operations when traffic is volatile.
SLO_{tgt}	Current value of SLO measurement.
SLO_{BP}	The *SLO restoration breaking point threshold*.
$SLO_{noDownScale}$	*No downscale action SLO threshold* is used to control the risk of SLA violation after SLO recovery action. Autoscaling actions are prohibited until the specified SLO threshold is reached.
L_{max}	*The highest load* (request per second, connections per second, packets per second) that a single replica can handle without violation of the target service level.
$T_{cooldown}^D$	The *cooldown period* is when no autoscaling action is happening.
$\Delta T_{cooldown}^D$	*Cooldown period adjustment steps*. Used to shorten upscale action cooldown period if load increase is high and prolong in cases when the load is low.
T_n	A length of *synchronization period between the SAA solution and the Kubernetes* cluster on the number of running replicas.
T_m	*Monitoring samples collection period.*
t_{delay}^D	*Replica provisioning or deprovisioning time*.
$t_{totalDelay}^D$	*Resource provisioning delay* is the time it takes to adjust resource provision or deprovision based on demand shifts.
$v_{baseline}$	*Baseline velocity* defines what the maximum load (L_{max}) increase per second can be handled by a single replica during a period equal to $t_{totalDelay}^D$.
$V_k(A_k)$	*Velocities vector* is used by the volatility detector module to analyze the last K monitoring samples of velocity and determine if the load is volatile.

[a]Here and throughout the document, $D \in [up, down]$ represents the direction of the autoscaling action

End of the Table 2.1

Parameter	Description
α_D	*Velocity factor* gives a raw estimate of how many times the current velocity v_n is different from the baseline velocity.
A_k	*Raw velocity* is a ratio between $v_{baseline}$ and velocity v_k, where k is the array element index.
C_n	*Current average CPU utilization* of all replicas that are running and available to handle the workload.
CT^D	The indicator of what CPU threshold is selected ($CT \in [upper, mid, lower]$) for upscaling or downscaling action ($D \in [Up, Down]$).
$C_c^{CT^D}$	The *dynamic CPU Thresholds* are dynamically adjusted thresholds that trigger autoscaling action and calculate desired replicas number.
ΔC^D	CPU *threshold adjustment steps* are used to adjust the CPU or another threshold (e.g., RAM) used for autoscaling action.
T_c (s)	*CPU adjustment period* is the period during which the CPU Adjuster module validates the state of SLO (above, on target, or below) and adjusts CPU utilization threshold values.

between v_n and $v_{baseline}$, the algorithm can categorize the load change as high, moderate, or stable, enabling appropriate decision-making in response to varying workload dynamics.

To measure the change, first, the baseline must be calculated as per Equation (2.3) representing the linear model. This model illustrates the average load that can be managed after adding or removing a single replica from the existing running replicas. It takes into account the time required to provision or deprovision that replica $t^D_{totalDelay}$. The value of $v^D_{baseline}$ is limited by a maximum number of replicas (R_{max}) or a minimum number of replicas (R_{min}). This method estimates the baseline necessary for the system to function within the limits of SLO:

$$v^D_{baseline} = \frac{L_{max} * R_{max}}{t^D_{totalDelay} * (R_{max} - R_{min})}, \tag{2.3}$$

here L_{max} represents the maximum throughput that a single replica can handle without resource starvation.

The $t^D_{totalDelay}$ present in the equation is a sum of times required to:

- provision ($D = up$) or deprovision ($D = down$) a single replica, that is, get replica to ready state (t^D_{delay});
- collect metrics (metric collection interval T_m);
- system to cooldown (cooldown period $T^D_{cooldown}$ (Equation (2.4))) .

$$t^D_{totalDelay} = t^D_{delay} + T^D_{cooldown} + T_m. \tag{2.4}$$

The t^D_{delay} value is empirically estimated through monitoring of pods startup behavior.

The resulting baseline velocity is used for the calculation of two types of velocity factors: the *increase factor* (α_{up} (Equation (2.5))) is calculated when load increases, and the *decrease factor* (α_{down} (Equation (2.6))) when the velocity is decreasing.

$$\alpha_{up}(v_n, R_n) = \left\lceil \frac{v_n}{(R_{max} - R_n - 1) * v_{baseline}} \right\rceil; \tag{2.5}$$

$$\alpha_{down}(v_n, R_n) = \left\lceil \frac{v_n}{(R_n + 1) * v_{baseline}} \right\rceil. \tag{2.6}$$

As presented in the equations above, the number of running replicas (R_n) influences the velocity factors. The increase factor has a lower impact as more replicas are active, while it increases as the number of replicas decreases. For example, with just a single replica active, introducing an additional one can boost the increase by as much as 0In contrast, the decrease factor works differently. It allows for faster downscaling when more replicas are active and slower downscaling when fewer replicas are in use.

The computed velocity factors are crucial in classifying the velocity into stable, moderate, and high levels. When the average load is equal to or lower than the defined baseline ($|\alpha_D| \in \{0, 1\}$), the velocity is categorized as stable. On the other hand, the moderate velocity corresponds to a state where the load exceeds the baseline up to (α_{high}) times compared to the baseline.

The high-velocity level ($|\alpha_D| \geq \alpha_{high}$) denotes a state where the load change, in relation to the baseline, exceeds the α_{high} threshold. α_{high} represents the upper bound value of the velocity factor, beyond which an increase in the velocity factor introduces a high risk of excessive resource over or underprovisioning. It is important to note that $|\alpha_D|$ represents the absolute value of the velocity factors, disregarding the velocity direction ($v < 0$ when the load is decreasing and $v > 0$ when the load is increasing). By establishing a threshold at α_{high}, the system can prevent excessive resource allocation and support more optimal scaling decisions, even when velocity is high.

These velocity impact factors offer several benefits:

- They contribute to timely decision-making for autoscaling by acting as input for adjusting the cooldown period and CPU threshold selection.
- They help detect frequent traffic fluctuations that may cause the autoscaler to follow the pattern, increasing the risk of SLO violations.

- They act as multipliers for determining the desired number of replicas, ensuring adequate resource allocation when the load increase per replica is too high for a single replica to handle.

The following paragraph provides a detailed description of the next module, which is named the Cooldown Period Calculator.

Cooldown Period Calculator

To prevent SLO violations, it is crucial for the algorithm to respond to unexpected loads promptly (Nguyen et al., 2020; Taherizadeh & Grobelnik, 2020). A faster reaction can be achieved by minimizing the cooldown period, even if it causes the potential risk of overprovisioning resources. However, in the case of resource deprovisioning, it is preferable to have an extended cooldown period to avoid the risk of the autoscaler removing replicas too soon.

The length of the cooldown period is calculated by the Cooldown Period Calculator, which takes into account the load velocity, as shown in Equation (2.7). An increased velocity factor value results in an extended cooldown period ($T^{D}_{cooldown}$) for downscale actions, while it shortens the cooldown period for upscale actions.

$$T^{D}_{cooldown}(\alpha_D) = T^{D}_{cooldown_0} - \mathit{\Delta} T^{D}_{cooldown} * \alpha_D, \tag{2.7}$$

here, $T^{D}_{cooldown_0}$ represents the default period during which the autoscaling algorithm should refrain from executing any autoscaling action. $\mathit{\Delta} T^{D}_{cooldown}$ is the increment step for adjusting the cooldown period, allowing for the extension or reduction of the time needed for system stabilization following the addition or removal of replicas.

A detailed description of another autoscaler module, Volatile Traffic Detector, is presented next.

Volatile Traffic Detector

The Volatile Traffic Detector module is critical in ensuring that autoscaling decisions do not repeat load fluctuation patterns that could negatively impact the QoS. This module detects volatility by identifying sudden changes in velocity, whether shifting from an increase to a decrease or vice versa.

To identify sudden changes in velocity, the Volatile Traffic Detector module collects the latest K velocity values (v_k) and calculates their ratio to the baseline velocity. The results are stored in an array $A[A_1, \ldots, A_k]$. The value of A_k, as defined in Equation (2.8), represents a raw velocity level. Differently from the velocity factor α_D it does not consider the current number of replicas.

$$A_k(v_k, v_{baseline}) = \left\lceil \frac{v_k}{v_{baseline}} \right\rceil \tag{2.8}$$

As a next step, the Volatile Traffic Detector module detects shifts in velocity levels from a high or moderate increase to a high or moderate decrease. It accomplishes this by identifying instances where velocity values exceeds a specific threshold, α_{high}, and determining when the traffic transitions from a high increase ($A_k \geq \alpha_{high}$) to a high decrease ($A_k \leq -\alpha_{high}$), or vice versa. This detection process is outlined in Equation (2.9).

As the final step, the Volatile Traffic Detector module classifies the load as volatile as presented in Equation (2.10). In this equation, the load is considered volatile if the number of shifts in velocity directions exceeds a predefined threshold, denoted here as $L_{volatile}$.

$$V_k(A_k) = \begin{cases} 1, & \text{if } (A_k \geq \alpha_{high}) \wedge (A_{k-1} \leq -\alpha_{high}) \\ 1, & \text{if } (A_k \leq -\alpha_{high}) \wedge (A_{k-1} \geq \alpha_{high}) \ ; \\ 0, & \text{otherwise} \end{cases} \tag{2.9}$$

$$isVolatile(V_k) = \begin{cases} true, & \sum_{i=0}^{k} V_k \geq L_{volatile} \\ false, & \text{otherwise.} \end{cases} \tag{2.10}$$

The following text in this sub-chapter further explains the Autoscaler module, the core of the SAA solution.

Autoscaler

The Autoscaler module serves as the primary decision-making component as it executes the core autoscaling algorithms. This module carries out several key functions, including:

- Selection of Dynamic CPU Threshold. TThe Autoscaler module determines the CPU threshold for scaling and computes the replica count.
- Scaling Decision-Making. The Autoscaler module analyzes the gathered monitoring data and determines when to trigger scaling actions.
- Calculation of Target Replicas. Based on the current load and scaling decisions, the Autoscaler module calculates the appropriate number of replicas that should be provisioned to handle the workload efficiently.
- Cooldown Period Reset. The Autoscaler module manages the cooldown period, which is the duration during which autoscaling actions are temporarily suspended after a scaling event. It resets the cooldown period when necessary, ensuring timely and responsive scaling actions.

The remaining text of this sub-chapter describes the components responsible for these functions in more detail, starting from the CPU threshold selection function, which serves as an important input for scaling action decision-making.

Compute Processing Unit Threshold Selection

To enhance the efficiency and stability of autoscaling decision-making, the SAA algorithm uses two sets of CPU thresholds: upscale and downscale thresholds. These thresholds are further divided into ranges, namely upper, mid, and lower.

The upper threshold represents the highest CPU utilization level and is the default threshold. It determines when scaling-up actions should be triggered. On the other hand, the lower threshold denotes the lowest CPU utilization value, indicating the threshold at which scaling-down actions should be considered. The mid threshold represents the intermediate range of CPU utilization, providing additional flexibility in determining scaling actions based on workload characteristics.

To facilitate a better understanding, the notation used in the equations is introduced below:

- C_n denotes the average CPU value of all currently running and ready-to-serve replicas. The CPU thresholds are defined using the following notation:
- CT^D represents the identifier of the selected CPU threshold, where $CT \in [upper, mid, lower]$. Additionally, $D \in [up, down]$ denotes the autoscaling action directions, specifically upscale or downscale;
- SLO_n stands for the current SLO value.

Furthermore, the index c represents the CPU threshold adjustment iteration, which occurs every T_c seconds. This iteration allows for periodic updates and adjustments to the CPU thresholds based on the changing workload conditions.

Equation (2.11) outlines the threshold selection logic for upscaling actions, while (2.12) demonstrates the selection of downscale thresholds. These threshold selection functions consider the velocity of the data. When the velocity is high, a lower threshold ($C_c^{lower^D}$) is chosen. For moderate velocity, a mid threshold ($C_c^{mid^D}$) is selected, and for a stable load, an upper threshold ($C_c^{upper^D}$) is utilized by solution. These approaches share similarities, albeit with a few exceptions. Both functions consider the current CPU load but for different reasons. The upscale action function checks if the current CPU utilization has already reached its "peak" value. If this condition is true, the function selects the upper threshold to minimize the risk of resource overprovisioning. For example, if two replicas are running with an average CPU utilization of 94%, the desired number of replicas will be three if the CPU threshold is set to 90% and four if it is set to 60% (as per logic presented in Equation (1.1)). On the other hand, the downscale function assesses the current CPU value to prevent fluctuations in autoscaling decisions.

By considering the current CPU utilization, the downscale function aims to avoid premature scaling-down actions triggered by temporary drops in CPU load. This helps to stabilize the autoscaling process and prevent unnecessary scaling actions based on transient changes in CPU utilization.

$$CT^{up}(C_n, SLO_n, SLO_{n-1}, \alpha_{up}, C_c^{upper^{up}}) = \begin{cases} upper, & \text{if } C_n > C_c^{upper^{up}} \\ lower, & \text{if } SLO_n - SLO_{n-1} < 0 \\ upper, & \text{if } \alpha_{up} \leq 1 \\ mid, & \text{if } 1 < \alpha_{up} \leq \alpha_{high} \\ lower, & \text{if } \alpha_{up} \geq \alpha_{high} \\ upper, & \text{otherwise} \end{cases} ; \quad (2.11)$$

$$CT^{down}(C_n, \alpha_{down}, C_c^{lower^{up}}) = \begin{cases} upper, & \text{if } C_n > C_c^{lower^{up}} \\ upper, & \text{if } |\alpha_{down}| \leq 1 \\ mid, & \text{if } 1 < |\alpha_{down}| \leq \alpha_{high} \\ lower, & \text{if } |\alpha_{down}| \geq \alpha_{high} \\ upper, & \text{otherwise} \end{cases} . \quad (2.12)$$

Additionally, the upscale threshold selection function considers changes in the SLO value since the last scaling request. If the SLO value decreases, indicating a need for faster response and increased resource provisioning, the lower threshold is selected. Also, when the velocity is high, the lower threshold is chosen. This enables a quicker reaction to sudden load changes, as the lower threshold is reached faster, triggering the provisioning of additional replicas. By selecting a lower threshold during high-velocity periods, the autoscaling algorithm ensures a more responsive and efficient scaling process, effectively adapting to dynamic workload fluctuations. The autoscaling decision-making is described next.

Autoscaling Decision Making

The SAA solution utilizes two algorithms, namely Algorithm 1 for upscale decisions and Algorithm 2 for downscale decisions. These algorithms operate in three distinct, sequentially ordered steps, employing dynamic CPU thresholds calculated in advance by the Dynamic CPU Thresholds Adjuster module.

In the first step, both algorithms use the dynamic CPU thresholds pre-calculated by the Dynamic CPU Thresholds Adjuster module.

Algorithm 1 Upscale action decision-making algorithm

```
Require: $SLO_n, SLO_{n-1}, SLO_{tgt}, C_n, C_c^{upper^{up}}$,
    $C_n, C_c^{upper^{up}}$
Ensure: Trigger the target replicas calculation
    function when $doUpscaling = true$
1: if $(SLO_n < SLO_{tgt}) \wedge (SLO_n - SLO_{n-1} < 0) \wedge \left(C_n > \frac{C_c^{upper^{up}}}{2}\right)$ then
2:     $doUpscaling \leftarrow true$
3: else if $((C_n \neq 0) \wedge (C_n > C_c^{CT^{up}}))$ then
4:     $doUpscaling \leftarrow true$
5: else
6:     $doUpscaling \leftarrow false$
7: end if
```

In the second step, the upscale algorithm checks if the current SLO decreases below the target SLO, denoted as SLO_{tgt}. If this condition is met, indicating a potential SLO violation, the upscale algorithm accelerates its decision-making process. It achieves this by utilizing a CPU threshold that is half of the highest CPU threshold, denoted as $C_c^{upper^{up}}$.

The utilization of a lower threshold in the upscale algorithm offers two benefits. First and foremost, it enables autoscaling to be triggered at an earlier stage when the lower threshold is reached, allowing for a timelier response to increasing workload demands. At the same time, it also ensures that a greater number of replicas are provisioned by autoscaler, effectively allocating more resources to handle the workload and, once again, minimizing the risk of failing the SLO compliance.

In the third step, the downscale action decision-making algorithm aims to mitigate the degradation of QoS. It achieves this by avoiding downscaling actions in two scenarios: when the SLO value is decreasing, indicating a decline in performance, and when the ratio of the current SLO value to the target SLO falls below a predefined threshold denoted as “no downscaling” ($SLO_{noDownScale}$). $SLO_{noDownScale}$ plays a crucial role in controlling the risk of the SLO falling

Algorithm 2 Downscale action decision-making algorithm

```
Require: $SLO_{noDownScale}, SLO_n, SLO_{n-1},$
    $SLO_{tgt}, v_n, C_n, C_c^{CT^{down}}, C_{noLoad}$
Ensure: Trigger the target replicas calculation
    function when $doDownscaling = true$
 1: if $(SLO_n - SLO_{n-1} < 0)$ then
 2:     $doDownscaling \leftarrow false$
 3: else if $\frac{SLO_{tgt}}{SLO_n} < SLO_{noDownScale}$ then
 4:     if $((v_n = 0) \wedge (C_n < C_{noLoad}))$ then
 5:         $doDownscaling \leftarrow true$
 6:     else
 7:         $doDownscaling \leftarrow false$
 8:     end if
 9: else if $((C_n \neq 0) \wedge (C_n < C_c^{CT^{down}}))$ then
10:     $doDownscaling \leftarrow true$
11: else
12:     $doDownscaling \leftarrow false$
13: end if
```

below the target, as it ensures that the system works above the target SLO. This threshold is determined empirically, and if $SLO_{noDownScale} < \frac{SLO_{tgt}}{SLO_n}$, downscale actions are re-enabled. Users have the flexibility to choose a lower $SLO_{noDownScale}$ value to minimize resource overprovisioning, but this decision comes with an increased risk of SLO violations. Conversely, a higher $SLO_{noDownScale}$ value is suitable for scenarios with fluctuating or unpredictable traffic patterns, while a lower value is more appropriate for constant and intensive loads. During experimentation with different values of $SLO_{noDownScale}$, it was observed that settings in the range of 1.00125–1.01 were sufficient to achieve the SLA compliance goal under all experimented workload conditions. When there is no load on the system, the expression represented by C_{noLoad} in line 4 of Algorithm 2 is used for downscaling.

The following text will delve into the logic behind replica calculation implemented in the Autoscaler module.

Replica Calculation

The Autoscaler module utilizes the Replica calculation algorithm (see Algorithm 3) to determine the target number of replicas. The primary objective of this algorithm is to prevent any violations of the SLO. The algorithm calculates the target number of replicas, R_{tgt}^D, based on several factors, including the current number of running replicas, load characteristics (such as velocity and volatility), current average CPU utilization, and service compliance with the SLO. The algorithm employs a state control loop to ensure synchronization between the Autoscaler module and the Kubernetes cluster. Every T_n seconds, it checks if the number of currently running replicas equals the previously calculated number of target replicas. Once these two values are equal, indicating that the system is in a stable state, the algorithm initiates the replica calculation process.

As the first step of re-estimating the target number of replicas, the algorithm confirms if the cooldown period is over, and when confirmed, the Autoscaler module proceeds with the re-estimation of the target number of replicas. In the case of upscaling actions, the algorithm performs additional checks. It verifies whether the SLO is increasing, even if it is currently violated (line 5), allowing for SLO recovery without the need for provisioning additional resources. Furthermore, the algorithm validates that the velocity has not changed its direction and remains at a moderate or high level, that is, $\alpha_D > 1$. If these conditions are unmet, the system re-evaluates the target number of replicas.

The estimation of R^D_{tgt} can vary depending on the SLO compliance state of the system undergoing re-scaling. When there are no SLO violations, the system is not overloaded, and the load remains stable and non-volatile, the desired number of replicas, denoted as R^D_d in Equation (2.13), is returned as R^D_{tgt}.

$$R^D_d(C_n, R_n, C^{CT^D}_c) = \left\lceil \frac{C_n}{C^{CT^D}_c} * R_n \right\rceil \tag{2.13}$$

R^D_d is used when the velocity of the load is close to the baseline. When the velocity of the load increases beyond the baseline, additional resources must be allocated to mitigate the risk of SLO violations. To address this, the velocity impact is factored into the estimation of the number of target replicas. This adjusted number of replicas is denoted as R_{α_D}, and its calculated using the following equation:

$$R_{\alpha_D}(R^D_d, \alpha_D) = \lceil R^D_d * \alpha_D \rceil. \tag{2.14}$$

When the system is not compliant with the SLO and service quality decreases, it might be necessary to overprovision resources to improve performance. The target replica count calculation considers the ratio of current to target SLO, specifically SLO_n/SLO_{tgt}. As the drop in SLO increases, more replicas are provisioned to mitigate performance degradation. However, in cases where $SLO_n/SLO_{tgt} \rightarrow 1$, the proportional difference has a minimal impact, resulting in slower SLO recovery. To accelerate recovery, the equation considers the difference between the target and current service levels ($SLO_{tgt} - SLO_n$). This adjustment accelerates the recovery of SLO when the SLO value is close to the target. Further, a velocity factor is applied to the provisioned replica count (R_{BP}) when $SLO_n/SLO_{tgt} \rightarrow 1$.

Algorithm 3 Target replica R^{D}_{tgt} calculation

```
Require: $R_n, R_{max}, R_{min}, v_n, SLO_n, SLO_{n-1}, SLO_{tgt}, \alpha_D, T^{D}_{cooldown}$,
    $C_n, L_n, L_{n-1}, doUpscaling, doDownscaling$
    $T_n$ – a synchronization period between the autoscaling solution and the Kubernetes cluster on the number of running replicas.
Ensure: Provide $R^{D}_{tgt}$ for the next autoscaling iteration and update time of the next
    cooldown period $t^{D}_{cooldown}$ end
 1: Every $T_n$ seconds check
 2: if $doUpscaling = true$ then
 3:     if $R_n = R_{max}$ then
 4:         return $R_n$
 5:     else if $t_{now} - t^{up}_{cooldown} > 0 \wedge SLO_n < SLO_{tgt} \wedge SLO_n - SLO_{n-1} \geq 0$ then
 6:         return $R_n$
 7:     else if $v_n < 0 \wedge \alpha_{down} \geq 1$ then
 8:         return $R_n$
 9:     else
10:         return $R^{up}_{tgt}, t^{Up}_{cooldown} \leftarrow t_{now} + T^{Up}_{cooldown}$
11:     end if
12: else if $doDownscaling = true$ then
13:     if $R_n = R_{min}$ then
14:         return $R_n$
15:     else if $(t_{now} - t^{down}_{cooldown} \leq 0)$ then
16:         return $R_n$
17:     else if $v_n > 0 \wedge \alpha_{up} \geq 1$ then
18:         return $R_n$
19:     else
20:         return $R^{down}_{tgt}, t^{down}_{cooldown} \leftarrow t_{now} + T^{down}_{cooldown}$
21:     end if
22: else
23:     return $R_n$
24: end if.
```

If the SLO value reaches the SLO breaking point threshold (SLO_{BP}), the QoS recovery slows down significantly. This typically happens when the SLO value is 2–5% below the target. At this point, $R_{BP} \approx R^{up}_{d}$, this may not be enough to recover the SLO. To speed up the process, the target replica count is adjusted by the velocity impact factor ($R_{BP} * \alpha_{up}$).

$$
\begin{aligned}
R_{BP}(R_d^{up}, SLO_n, SLO_{tgt}) = R_d^{up} + \left\lceil \frac{SLO_n}{SLO_{tgt}} * R_d^{up} \right\rceil \\
+ \lceil (SLO_{tgt} - SLO_n) * R_d^{up} \rceil)
\end{aligned}
\tag{2.15}
$$

When determining the number of target replicas, the Autoscaler module selects the appropriate equation based on the scaling action and direction. For upscaling, Equation (2.16) is used, while for downscaling, Equation (2.17) is employed.

$$
\begin{aligned}
&R_{tgt}^{Up}(\alpha_{up}, C_n, SLO_n, SLO_{n-1}, C_c^{upper^{up}}) = \\
&\quad \begin{cases}
R_d^{up}, & \text{if } C_n > C_c^{upper^{up}} \wedge SLO_n \geq SLO_{tgt} \\
& \wedge\, SLO_n - SLO_{n-1} \geq 0 \\
R_{BP}, & \text{if } SLO_n < SLO_{tgt} \wedge \frac{SLO_n}{SLO_{tgt}} \leq SLO_{BP} \\
R_{BP} * \alpha_{up}, & \text{if } SLO_n < SLO_{tgt} \\
R_{\alpha_{up}}, & \text{otherwise}
\end{cases}
\end{aligned}
\tag{2.16}
$$

The traffic volatility is considered during the downscaling process as presented in Equation (2.17). When there is significant fluctuation in the workload, the downscaling occurs gradually in small increments. At each step, the number of replicas is reduced by 10%, considering the velocity decrease factor. This incremental approach ensures that the number of replicas is not drastically decreased, particularly when fewer than ten replicas remain.

$$
\begin{aligned}
&R_{tgt}^{down}(R_n, \alpha_{down}) = \\
&\quad \begin{cases}
\lfloor R_n * (1 - 0.1 * \alpha_{down}) \rfloor, & \text{if } (isVolatile = true \wedge R_n > 3) \\
R_d^{down}, & \text{otherwise}
\end{cases}
\end{aligned}
\tag{2.17}
$$

To ensure adequate resource capacity and reduce the risk of SLO violations, downscale actions are not taken when there are three or more replicas present. Removing a single replica in these situations would lead to a significant capacity loss of at least 33%. By adopting this approach, the Autoscaler module enhances reliability in managing volatile traffic patterns. This improvement reduces the risk of SLO violations and ensures smoother overall performance. However, it slightly increases resource consumption. There are predefined maximum and minimum limits for the number of replicas to maintain control over resource allocation. If $R_{tgt}^D \geq R_{max}$, the value of R_{tgt}^D is set to R_{max} to ensure it does not exceed the maximum capacity. Likewise, if $R_{tgt}^{D} \leq R_{min}$, it is adjusted to R_{min} to guarantee a minimum level of resources.

The following paragraph will delve into the details of the Dynamic CPU Thresholds Adjuster (DCTA) module and the process involved in calculating the desired CPU thresholds. It will provide a comprehensive understanding of how these thresholds are dynamically adjusted based on the system's performance and SLO requirements.

Dynamic Compute Processing Unit Thresholds Adjuster

The process of selecting CPU thresholds is known to be complex and prone to errors (Lorido-Botrán, 2012). Inadequately chosen CPU thresholds can lead to various issues, including resource delays in autoscaling decisions, under-provisioning, or over-provisioning. There is also a possibility of thresholds becoming obsolete if there are changes in the characteristics of applications or underlying infrastructure, which is especially relevant for containerized microservice architecture-based applications running in shared environments like cloud computing.

To address the aforementioned issues and mitigate the risk of SLO violations, the Dynamic CPU Threshold Adjuster (DCTA) module is introduced. The DCTA dynamically adjusts the CPU thresholds based on the current state of SLO compliance (met or not met). The adjustment algorithm, represented by Equation (2.18), determines how the CPU thresholds must be adjusted.

The DCTA module performs periodic adjustments to the CPU thresholds at regular intervals, denoted as T_c. These adjustments are based on the current state of the SLO. When the SLO is being met, the DCTA increases the CPU thresholds by a predefined adjustment step, denoted as $\varDelta C^{up}$. This means that the scaling-up process is delayed until the CPU utilization reaches a higher threshold, thereby minimizing resource waste while ensuring SLA compliance.

Contrariwise, when the SLO is not met, the DCTA module decreases the upscale CPU thresholds by $\varDelta C^{down}$ step. This granular adjustment enables faster scaling up, allowing the system to respond promptly to increasing loads. Additionally, it delays the scaling-down process as the load decreases, ensuring stability and preventing premature resource deallocation.

$$C_c^{CT^D}(C_{c-1}^{CT^D}, SLO_c) = \begin{cases} C_c^{CT^D}, & \text{if } t_{now} - (t_{c-1} + T_c) = \\ & = 0 \\ C_{c-1}^{CT^D} + \varDelta C^D, & \text{if } SLO_c > SLO_{tgt} \\ C_{c-1}^{CT^D} - \varDelta C^D, & \text{if } SLO_c < SLO_{tgt} \\ C_c^{CT^D}, & otherwise \end{cases} \tag{2.18}$$

Here, t_c represents the time when the latest adjustment was made, while c represents the index of the data point associated with that adjustment. t_{now} denotes the current time at which the CPU thresholds are being validated.

The initial values of the CPU thresholds $C_0^{CT^D}$, used at the initialization of the algorithm, are calculated using Equation (2.19).

$$C_0^{CT^D} = \frac{C_{max}^{CT^D} + C_{min}^{CT^D}}{2} \tag{2.19}$$

Subsequent values $C_{c+1}^{CT^D}$ are calculated as presented in the following equation:

$$C_{c+1}^{CT^D} = C_c^{CT^D} + \varDelta C^D. \tag{2.20}$$

To ensure that the CPU thresholds remain within appropriate ranges, upper and lower limits are set for both the upscale and downscale thresholds. These limits are denoted as $C^{CT^D} \in [C_{min}^{CT^D}, C_{max}^{CT^D}]$. The limits for each application and infrastructure environment are individually determined based on their unique characteristics and may vary accordingly.

Notably, the number of CPU thresholds and velocity levels may vary. Manipulating the number of levels provides more fine-tuned adjustments.

The following sub-chapter presents the evaluation criteria used to assess the effectiveness and efficiency of the autoscaling solutions in the scope for this sub-chapter.

2.1.3. Evaluation Criteria for the Efficacy of Autoscals in Service Level Agreement Compliance Assurance

The autoscaling solutions to be analyzed in this sub-chapter are evaluated using two criteria.

The primary evaluation criterion was the solutions' efficacy in supporting or exceeding the SLO value compared to the target SLO value (Equation (2.21)). If a solution can consistently maintain $SLO_n \geq SLO_{tgt}$ or successfully recover and reach the target SLO value, it is considered to have passed this criterion. The target SLO (SLO_{tgt}) was calculated using Equation (2.21).

$$SLO_{tgt} = \frac{\sum_0^n l_n^{SLI_{tgt}}}{\sum_0^n l_n^{all}} \tag{2.21}$$

Here, index n represents the data point index value of the latest event included in SLO measurement period. The term $l_n^{SLI_{tgt}}$ refers to the number of requests that were delivered with a response time equal to or less than the target response time value (SLI_{tgt}), while l_n^{all} represents the total number of requests served by the system during the specified service window. The measurement of the SLO value began from the moment the system served the first request till the end of the experiment.

To measure the performance of the system, the response time of the requests was used as the service level indicator. It is worth noting that the choice of service level indicators is flexible if the indicator is based on performance measurements such as response time, error rate, percentage of lost packets, availability, N-th percentile latency, etc.

The efficiency in resource provisioning was a secondary criterion. Resource utilization efficiency was evaluated using the method described in (Pozdniakova et al., 2023) and referred to as the "touchstone" autoscaler. The essence of the method is that the theoretical number of replicas required to process the load within the defined SLO (R_{η_n}) is calculated using Equation (2.22). R_{η_n} is determined by dividing the number of requests processed at a given moment (L_n) by the maximum number of requests that a single replica can handle (L_{max}) without SLO violation. This calculation represents an ideal "touchstone" autoscaler that provisions the required number of replicas without any delay, ensuring optimal resource utilization. By comparing the actual number of replicas provisioned by the evaluated solutions with R_{η_n}, their resource utilization efficiency can be assessed.

$$R_{\eta_n} = \left\lceil \frac{L_n}{L_{max}} \right\rceil \tag{2.22}$$

In cases where the SLO target is not being met, the value of R_{η_n} is adjusted by the value of SLO_n/SLO_{tgt}. This adjustment allows for determining the required number of replicas, denoted as $R_{\eta_n}^{tgt}$ (as described in Equation (2.23)), needed to attain the SLO target.

$$R_{\eta_n}^{tgt} \approx \left\lceil R_{\eta_n} * \frac{SLO_{tgt}}{SLO_n} \right\rceil \tag{2.23}$$

The difference between the replica count provided by the solution and that of the "touchstone" autoscaler is calculated to determine if the solution provisions the minimum required replicas. If the difference is positive (R_n^+ in Eqaution (2.24)), it indicates over-provisioning, meaning that there are more replicas than necessary. On the other hand, if the difference is negative (R_n^-) in Equation (2.25)), it indicates under-provisioning, meaning that there are fewer replicas than required to handle the current demand in resources. This comparison provides insights into the degree of over-provisioning or under-provisioning of replicas in relation to the number of replicas provisioned by the "touchstone" autoscaler.

Before concluding this sub-chapter, it's important to note that addressing the challenges of meeting performance-based SLA requirements led to a multi-module solution. While this enhances algorithm robustness and consistency in achieving SLA fulfillment, it also introduces configuration complexity. Furthermore, this solution involves a custom autoscaler, which can require significant effort for practical implementation. Consequently, this research aims to simplify the implementation process and increase the likelihood of adopting the proposed autoscaling solution for SLA fulfillment.

$$\sum_{0}^{n} R_n^{+} = \begin{cases} R_n^{+} = R_n - R_{\eta_n}^{tgt}, & \text{if } (R_n - R_{\eta_n}^{tgt}) > 0 \\ \\ R_n^{+} = 0, & \text{if } (R_n - R_{\eta_n}^{tgt}) < 0 \end{cases}; \tag{2.24}$$

$$\sum_{0}^{n} R_n^{-} = \begin{cases} R_n^{-} = R_n - R_{\eta_n}^{tgt}, & \text{if } (R_n - R_{\eta_n}^{tgt}) < 0 \\ \\ R_n^{-} = 0, & \text{if } (R_n - R_{\eta_n}^{tgt}) > 0 \end{cases}. \tag{2.25}$$

Experiment results presented in sub-chapter 3 demonstrate that selecting utilization thresholds affects performance-based SLA fulfillment outcomes. The following sub-chapter introduces a solution designed to assist in determining resource utilization thresholds for rule-based autoscaling solutions, mainly focusing on the widely adopted Kubernetes Horizontal Pod Autoscaler.

2.2. Service Level Agreement-Adaptive Dynamic Threshold-Adjustment Algorithm Design for Rules-based Autoscalers

The threshold-based autoscalers, despite their simplicity, suffer from challenging threshold management, impacting SLO compliance. This sub-chapter provides an overview of the proposed dynamic threshold adjustment approach, the SLA-Aware Threshold Adjustment (SATA) algorithm, which aims to ensure compliance with the service level objectives (Pozdniakova et al., 2024).

2.2.1. Motivation For Developing a Solution to Adjust the Thresholds in the Context of Horizontal Pod Autoscaler

The rise of containerized applications has led to the development of container orchestration platforms such as Kubernetes (kubernetes.io, 2024). Kubernetes employs the Horizontal Pod Autoscaler (HPA) (kubernetes.io, 2022), which calculates the required resources based on metrics like CPU, RAM, and network throughput. HPA is one of the most adopted solutions that use static thresholds to estimate the right amount of resources to be provisioned to ensure desired performance for applications where selected utilization thresholds are the most influential factor ensuring the desired QoS. The HPA scales up the number of pod replicas if the current resource usage exceeds a specific threshold (target utilization). Conversely, if the current resource usage falls below the target, the HPA scales down the number of pod replicas.

The target utilization threshold is crucial in managing the performance of the application as it dictates the resource allocation during each autoscaling action. However, the Kubernetes creators do not provide specific guidance on setting the threshold, particularly when aiming to align the performance of the application with the Service Level Objectives (SLOs) outlined in the SLA. As a result, determining the threshold becomes a long and challenging process (Shafi, Abdullah, Iqbal, Erradi, & Bukhari, 2024). If the thresholds are set too low, resources can be overprovisioned. However, this will allow for a quicker response to load changes and, as a result, lead to improved application performance. Conversely, choosing a threshold that is too high may lead to fewer provisioned replicas and leave no buffer for the detection of and reaction to the load increase (Developers, n.d.). These two factors may cause a decline in performance and an increased risk of failing to meet the application performance regarding the SLOs (Sahal et al., 2016).

Another problem is that HPA has a slow reaction (Huo et al., 2022), so as it will be presented in the third chapter, it is not enough to benchmark the application performance and establish a relationship between the response time and resource utilization to find thresholds that ensure compliance with the SLOs. The utilization threshold must be set lower to allow time for reaction and replica provisioning. This buffer is essential for ensuring that the system can cope with sudden changes in demand (Developers, n.d.); however, it is unclear how to estimate the size of such a buffer. The threshold determination becomes even more challenging, as the cloud environment is not homogeneous, which causes inconsistency in resource provisioning (Al-Haidari et al., 2013; Khaleq & Ra, 2021). Noisy neighbors are another problem that causes inconsistency in the performance of provisioned resources (Makroo & Dahiya, 2016). Cloud-native applications are frequently updated and redeployed, requiring dynamic adjustments to autoscaling thresholds. Manually finding and updating a suitable threshold to ensure the desired level of system performance becomes a challenging process (Al-Haidari et al., 2013; Kang & Lama, 2020; Khaleq & Ra, 2021).

To solve the abovementioned issues, practitioners and academia propose many alternatives to the HPA in the form of custom autoscalers. Nevertheless, HPA continues to be one of the most widely used horizontal autoscaling solutions (CNCF, 2024b; Datadog, 2024). Therefore, this study proposes an approach for determining HPA thresholds that can be incorporated into automation processes or serve as a starting point for adjusting thresholds with minimal implementation complexity and ease its practical adoption.

The following sub-chapter presents the method for SLA-aware dynamic threshold determination.

2.2.2. Target Threshold Determination Algorithm

This sub-chapter introduces an approach to identifying the target utilization threshold that ensures the system performs at the level defined in the SLA. A series of steps constitutes the implementation of this approach. As the first step, a sufficient number of metrics is collected to provide suggestions. In the second step, the collected metrics are cleaned up, and outliers are removed to improve the accuracy of the algorithm. As a third step, the collected metrics are aggregated into CPU ranges, and the ratio between the number of compliant events and violations is calculated per each range. In the fourth step, the smoothing technique is applied to remove noise in values. As the last step, the suitable threshold is determined by finding the highest CPU value where the desired SLO is met. The steps described above are elaborated in more detail in the text below.

Step 1. Collection of a sufficient number of monitoring data points.

As the first step, the system should collect enough metrics M_{suff} to identify the number of violations per threshold. The following metrics are collected at each moment n to achieve the goal:

- CPU_n – average CPU utilization;
- SLI_n– performance-based service level indicator value, such as average response time, tail latency, throughput (e.g., requests per second (rps));
- RPS_n– the average number of requests per second;
- Pod_n–number of pods in “Ready” state.

Let the set of the metrics provided above be denoted as tuple m. Then, $M_{suff} = \langle m_0, m_1, \ldots m_n \rangle$, where $m_n = \langle CPU_n, SLI_n, RPS_n, Pod_n \rangle$.

The size of M_{suff} depends on two factors: the length of the period during which the data for threshold evaluation are collected (threshold evaluation period T_{eval}) and how frequently the metrics are collected (length of metric-collection period T_m). In other words, $|M_{suff}| = T_{eval}/T_m$. The experiment results indicated that detecting the threshold is feasible with 150 samples per evaluation period; however, a sample size of 300 enhances accuracy.

The use of RPS_n and Pod_n metrics is optional, as those are used to remove outliers in the CPU performance values, which appear when the number of pods is very small. These outliers can significantly skew threshold calculations, leading to inaccurate autoscaling decisions. The process of eliminating outliers and other invalid data is explained in the following step.

Step 2. Data cleaning.

This step is applied to improve the accuracy of the algorithm. It involves identifying and removing invalid values and outliers that might have been introduced due to system-specific monitoring issues. For instance, when the monitoring system is overloaded, it may generate empty values of requests per second or response time. To clean up the data, as the first step, empty metrics are removed from M_{suff}. For instance, data points (M_{nosli}) where the $m(SLI)$ metric is not available are removed as presented in Equation (2.26).

$$M_{sli} = M_{suff} \setminus M_{nosli} \tag{2.26}$$

Here, $M_{nosli} = \{m : m \in M_{suff} \wedge m(SLI)\ \textit{does not exist}\}$.

Next, to ensure that the algorithm does not propose very low CPU utilization, the metrics collected when there was no load are removed, denoted as M_{noload} in Equation (2.27).

$$M_{nozeros} = M_{sli} \setminus M_{noload} \tag{2.27}$$

Here, $M_{noload} = \{m : m \in M_{sli} \wedge ((m(RPS) = 0) \wedge (m(SLI) = 0) \wedge (m(RPS) = 0) \wedge (m(Pod) = 0))\}$.

Outliers can be introduced during the upscale and downscale actions where the number of pod replicas is low. For instance, during an upscale action, the system may report many violations while CPU utilization drops. This occurs because the load has not yet been distributed among all replicas. Older replicas, which may deliver a significant amount of system capacity, still report high response times despite low CPU utilization. Removing such anomalous data is recommended to improve the accuracy of the algorithm. This work follows the recommendation and uses the interquartile range (IQR) method (Dash, Behera, Dehuri, & Ghosh, 2023; Han, Kamber, & Pei, 2012) for outlier detection, which is presented below in more detail.

The interquartile range is a statistical technique used to identify outliers within a dataset. The dataset is first sorted in ascending order and then divided into four equal parts. The points dividing the dataset into four equal parts are known as quartiles. The first quartile ($Q1$) represents the initial 25% of the data or the 25th percentile, the second quartile ($Q2$) represents the middle point or median, while the third quartile ($Q3$) represents the final 25% or the 75th percentile. The interquartile range represents the middle half of the data, which includes all the data between the third quartile ($Q3$) and the first quartile ($Q1$). All values falling at least $1.5 \times IQR$ above the third quartile or below the first quartile are considered anomalous. The values of $Q1$ and $Q3$ used for determining the IQR, and the IQR itself, are computed using the following equation:

$$IQR = Q3 - Q1, \text{ where } Q1 = X_{\lceil (z+1)/4 \rceil},\ Q3 = X_{\lceil 3\times(z+1)/4 \rceil}. \tag{2.28}$$

Here, X denotes an element of an ordered dataset, z represents the number of elements in the dataset (size of the dataset), and the subscript of X represents the equation used to identify the index of the element belonging to the respective quartile $(Q1, Q3)$.

As described at the beginning of the step description, there might be occasions when the system may report anomalous metrics. Analyzing the data number of requests per second (rps) that a single CPU can handle (RPC_z) and the number of rps that a single pod can handle (RPP_z) helps identify instances of anomalous container performance in pods and removing the data from evaluation. This removal improves the quality of threshold detection. For each metric $m_z \in M_{nozeros}$, the RPC and RPP values are calculated as presented in Equation (2.29) and Equation (2.30), respectively.

$$RPC_z = \frac{RPS_z}{CPU_z}; \tag{2.29}$$

$$RPP_z = \frac{RPS_z}{Pod_z}. \tag{2.30}$$

Once the RPC and RPP are calculated, the metrics $M_{nozeroes}$ are sorted by the RPC value in ascending order, and the IQR method is applied to remove anomalies. The value of the first quartile $(Q1_{RPC})$ and third quartile $(Q3_{RPC})$ of the RPC is identified using Equation (2.28). Finally, all the metrics, where $RPC_z \notin (Q1_{RPC} - 1.5 \times IQR; Q3_{RPC} + 1.5 \times IQR)$, are considered as outliers and are removed from $M_{nozeroes}$. The same procedure is repeated using the RPP, to obtain a set of a better quality metrics to proceed with CPU threshold estimation, denoted as M_{eval}.

Step 3. Data grouping by CPU range and the calculation of the number of violations per CPU range.

In this step, the collected and cleaned metrics are grouped into ranges by the CPU using the following actions, denoted as A:

- **A1**. The cleaned metrics M_{eval} are first ordered by the CPU from the low to high CPU value. Let the new sequence be denoted as $MC = \{m_c : m_c \in M_{eval}, m_c(CPU) \leq m_{c+1}(CPU))\}\}$.
- **A2**. The elements of MC are grouped into smaller subsequences, or ranges, based on their CPU values. Metrics with CPU values that fall into the same 1% CPU range are placed into the same group (MCR_i). This procedure is applied to all available metrics while maintaining their original sorting by the CPU value. In such a way, the sequence of sequences is created $MCR = \{MCR_i : \textit{i is an integer, } i \in [0; 100]\}$, where $MCR_i = \{mcr_{(i)} : mcr_{(i)} \in MCR_i, i-1 < mcr_{(i)}(CPU) < i+1\}$ is the sequence of metrics that belongs to the same 1% CPU. Here, i is an index of the CPU range.

- **A3**. It is assumed that the 1% CPU range should contain at least 1% of all collected metrics during the threshold evaluation period (T_{eval}). However, MCR_i might contain a smaller number of elements. As a result, up to three following MCR_i subsequences might be united into a bigger or the same size range MCR_r to ensure they contain at least 1% of all metrics collected during T_{eval}, but not less than five elements ($minSize = \max(5, \frac{|M_{eval}|}{100})$). If $|MCR_r| < minSize$, then $MCR_r = \varnothing$. Here, r denotes an index equal to the index of the last CPU range included in the combined range. For instance, if MCR_r unites the MCR_2 and MCR_3 ranges, then $r = 3$; if MCR_r unites only one range, then $r = i$. Combining metrics into larger ranges may reduce the algorithm's precision in detecting thresholds, so going beyond a 3% range is not recommended.

After grouping the metrics, the SLO (SLO_r) for each CPU range MCR_r is calculated as presented in Equation (2.31).

$$SLO_r = \begin{cases} 100 - \frac{1}{b}\sum\limits_{p=0}^{p=b} V_p, & \text{if } |MCR_r| \geq minSize \\ 100, & \text{if } r = 0 \\ 0, & \text{if } r >= 99 \\ SLO_{r-1}, & \text{otherwise} \end{cases} \tag{2.31}$$

Here, $[\sum_{p=0}^{p=b} V_p]$ represents the total number of events V_p where the SLI value ($mc_p(SLI) \in MCR_r$) exceeded the SLI target value (SLI_{tgt}), indicating a violation of the SLO, as shown in Equation (2.32). The index p corresponds to the elements in the set MCR_r, and b indicates the index of the last element in the range.

$$V_p = \begin{cases} 1, & \text{if } mc_p(SLI) > SLI_{tgt} \\ 0, & \text{if } mc_p(SLI) \leq SLI_{tgt} \end{cases} \tag{2.32}$$

As can be seen in the first line of Equation (2.31), SLO_r is the percentage of events that conform with the target SLI value within a CPU range. An SLO of 100% is assigned to the CPU range of 0%, as there are no violations when there is no or minimal load. Conversely, SLO compliance is equal to 0% when the CPU range index is 99 or higher because the SLO cannot be met when CPU utilization is near 100%. This allows the imputation (Sidekerskiene & Damasevicius, 2016) of missing values by replacing missing initial and last values for the SLO.

To finalize this step, the mapping is created between the SLO of CPU range (SLO_r) and the corresponding ID of $MCR_r(CPU)$ range, denoted as CTR in Equation (2.33).

$$f : CTR \implies SLO_r \tag{2.33}$$

Here, $CTR = \lceil \max MCR_r(CPU) \rceil$ is the ID of a range that is equal to the value of the CPU metric with the highest value in the range.

Before proceeding to the next step, it is important to note that the metrics are grouped into ranges larger than 1% to ensure that each range has enough metrics to calculate the SLO metric accurately. This minimizes fluctuations between neighboring values. Let us say we have two metrics collected at a very low CPU utilization, where the CPU value is around 3%. In this scenario, each metric will be given a weight of 50% when calculating the SLO of the range. For instance, if the next range has a CPU value of 4% and contains 10 compliant values, then the SLO for the range will be 100%. However, if there is only one non-compliant event in the range of 3%, the SLO for the range may fluctuate up to 50%. While it is possible to unite more than three 1% CPU ranges (MCR_i), it is not recommended to estimate the SLO for ranges larger than 3% as this would negatively impact the algorithm's accuracy.

The next step aims to improve the accuracy of threshold prediction and remove noise caused by fluctuations between neighboring values.

Step 4. Smoothing of the SLO values per the CPU range.

The algorithm may be effective even with a relatively low number of events; however, the limited number of events per MCR_r can lead to fluctuations in the relationship between CTR and SLO_r. This phenomenon is illustrated in Figure 2.2a and discussed in the previous step. To mitigate this issue, a smoothing technique known as the Simple Moving Average (SMA) is applied. The SMA helps eliminate fluctuations and highlights underlying trends (Raudys, Lenčiauskas, & Malčius, 2013). This technique calculates the average value of a set of numbers over a specified number of previous periods, known as to as a window or lag. The formula for calculating the SMA is provided in the following equation:

$$SLO_{r_w} = \frac{1}{w} \sum_{i=1}^{w} SLO_{r-i}. \tag{2.34}$$

Here, SLO_{r_w} is the SLO value of a range r smoothed over a window of size w; SLO_{r-i} are the SLO values of the CPU ranges with indexes varying within the window size w (from $r - i$ to r).

The recommendation provided by Hyndman and Athanasopoulos (2019) and discussed in Hyndman (2014) is used to determine the appropriate window size (w), or lag, to be applied for SMA smoothing. to be applied for SMA smoothing. The approach is represented in Equation 2.35. According to this equation, the maximum allowable size for the window is 10.

$$w = \min(|MCR|/5, 10) \tag{2.35}$$

Step 5. Suggestion for the desired CPU threshold value.

After smoothing out the CPU values, the next step is to choose the highest CPU range with a number of violations (SLO_{r_w}) that is lower than or equal to the SLO-defined number of violations (SLO_{tgt}). This chosen threshold is then considered the target utilization threshold CTR_{slo} and is determined using the following equation:

$$CTR_{slo} = \max_{SLO_r \geq SLO_{tgt}} \{f^{-1}(SLO_{r_w}) : SLO_{r_w} \geq SLO_{tgt} \text{ exists}\}. \tag{2.36}$$

Here, $f^{-1}(SLO_{r_w}) = \{CTR_{slo} \in R : f(CTR_{slo} = SLO_{r_w})\}$.

Before concluding this sub-chapter, it is worth mentioning that, in this work, the use of the Centered Moving Average (CMA) (Hyndman & Athanasopoulos, 2019) was also evaluated to smooth out the fluctuations and identify the target utilization threshold. It was assumed that this method would provide more conservative suggestions for thresholds than the SMA. Three experiments using the HPA were executed, and the utilization thresholds were set to values of 44%, 48%, and 50% to evaluate these approaches.

The results are presented in Figure 2.2b and c, where the lines present the relationship between the CPU range ID and SLO, and the SLOs achieved by the end of each experiment are presented as dots. The figures illustrate that the SMA identifies the relationship between the threshold and SLO values more accurately than the CMA. Therefore, the SMA is employed as the primary smoothing technique in this study. It is important to ensure that the algorithm can access data points that closely align with the desired SLO performance. The prototype outlined in the upcoming sub-chapter includes mechanisms designed to facilitate the collection of appropriate data points, enabling more accurate estimations.

The algorithm presented in this sub-chapter is designed to identify a threshold value that static threshold-based autoscaling algorithms can use to maintain the desired quality of service (QoS). The static threshold-based method is most effective in scenarios with no significant changes in performance or load during the timeframe used for estimating the threshold. It also works best when the load patterns in the future period do not significantly differ from those used for the estimate. However, in real-world production settings, load fluctuations are common. The performance of cloud resources can vary, and the provisioned resources may not be consistent. Consequently, the target utilization threshold that ensures the system operates at the required performance level to meet the SLO during nighttime could fall short during daytime operations. Furthermore, the algorithm may recommend a lower CPU threshold when analyzing performance over extended periods, which can lead to excessive resource allocation. Therefore, an autoscaler must dynamically adjust the proposed threshold value to adhere to the SLA requirements.

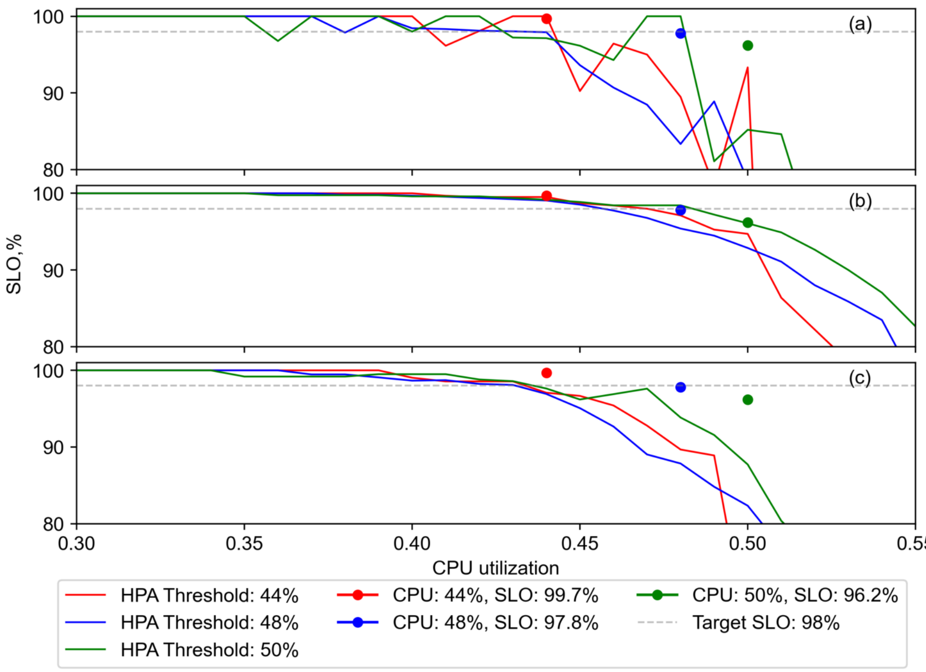


Fig. 2.2. Target threshold graphical determination. (a) Line graph of the range of achieved SLO values at specific CPU ranges without smoothing. (b) Line graph of the range of achieved SLO values at specific CPU ranges calculated using Simple Moving Average smoothing. (c) Line graph of the range of achieved SLO values at specific CPU ranges calculated using Centered Moving Average smoothing. Dots represent the SLO achieved in the experiments when the Horizontal Pod Autoscaler (HPA) was configured with a static CPU utilization threshold

In the following sub-chapter, the dynamic threshold-adjustment algorithm will be outlined to address the above-mentioned issues. The algorithm is then implemented as a SATA prototype solution to assess the effectiveness of CTR_{slo} threshold detection and dynamic adjustment.

2.2.3. Dynamic Threshold-Adjustment Algorithm

This sub-chapter presents a prototype solution designed to assess the proposed threshold determination method's effectiveness. Specifically, it evaluates how well the proposed algorithm determines the threshold value that enables the system to operate as closely as possible to the defined SLO. Additionally, it examines the applicability of the proposed threshold determination method across various workload conditions.

The proposed SATA prototype is designed to support HPA, as HPA employs threshold-based policies to govern autoscaling. The SATA solution uses a set of rules and algorithms to dynamically adjust the target utilization threshold for HPA (see Equation (1.1)). The CPU utilization threshold will be used for the prototype evaluation. For better readability, the CPU threshold will be referred to as the desired CPU threshold and denoted as (CT_d) instead of the original metric notation (M_d) used by HPA. The selection of the algorithm and rules to be used by the SATA prototype depends on operational conditions, which are as follows:

- The system is in **the initialization phase**, meaning that the minimum of required metrics has not been collected yet to estimate the target utilization threshold ($|M| < |M_{suff}|$);
- The system is in a **resource underprovisioning state**, meaning it is impossible to identify CTR_{slo} since all the SLO_{r_w} values are below SLO_{tgt}, indicating that not enough resources are provisioned;
- The **normal operational conditions** cover all other cases not mentioned above.

A different CT_d adjustment logic is then used based on the operational state of the identified system.

The following text explains how resource underprovisioning is detected and how detected underprovisioning impacts the duration of the threshold-adjustment period (T_{adjust}). Then, sub-chapter 2.2.3 explains how the CT_d value is adjusted based on the system's operational condition.

Resource Underprovisioning Detection

Detecting resource starvation in a system is critical, as it can negatively affect the state of SLOs. As a result, such conditions should be detected as soon as possible. Algorithm 4 describes a logic that determines if the system is experiencing resource underprovisioning. The algorithm counts the number of underprovisioning events that occur between autoscaling actions.

The calculated algorithm values are used as the input parameters by the expedite function, which is defined in Equation (2.37). This function checks if the number of consequent T_{scale} periods that contain underprovisioning events ($underpov$) exceeds the threshold of maximum allowed underprovisioning events $bndry_{under}$ or if the number of periods between which the SLO value decreases (SLO_{drop}) exceeds a boundary ($bndry_{slo}$) of maximum allowed number of the periods.

If none of the $bndry_{slo}, bndry_{under}$ thresholds are exceeded, the time taken to initiate an update of the target threshold ($timeToUpdate()$) is equal to T_{adjust}. However, if any of the thresholds are exceeded, $timeToUpdate()$ is reduced to three scale periods. This action is described in Equation (2.38). The reduction of time accelerates the process of adjusting thresholds as outlined in Algorithm 5, allowing for a quicker response to resource underprovisioning and reducing the

Algorithm 4 Resource underprovisioning detection algorithm

```
Require:
    $SLO_{tgt}$;
    $SLO_{now}$– current SLO value;
    $SLO_{lastScale}$– the SLO value before the last autoscaling action taken;
    $T_{scale}$– time between autoscaling actions (autoscaling period);
    $M_{T_{scale}}$ is a collection of metrics collected during the autoscaling period ($T_{scale}$);
    $|M_{under}|$ is the number of metrics collected when the system was in a high under-provisioning state during the autoscaling period, where $M_{under} = \{m_u : m_u \in M_{T_{scale}}, m_u(CPU) \neq 0 \wedge (m_u(RPS) = 0 \vee m_u(SLI) = 0)\}$.
Ensure: Compute and return:
    $underpov$– the number of consequent periods during which underprovisioning events occur, that is $|M_{under}| > 0$, or when fewer metrics were collected than expected to collect during $T_{scale}$;
    $SLO_{drop}$– the number of consequent periods during which the SLO decreased between the two latest autoscaling actions.
1: if $(SLO_{now} - SLO_{lastScale} < 0) \wedge (SLO_{now} < SLO_{tgt})$ then
2:     $SLO_{drop} \leftarrow SLO_{drop} + 1$
3: else
4:     $SLO_{drop} \leftarrow 0$
5: end if
6: if $(|M_{T_{scale}}| < |M_{T_{scale}}| \times \frac{T_{scale}}{T_m}) \vee (|M_{under}| > 0)$ then
7:     $underpov \leftarrow underpov + 1$
8: else
9:     $underpov \leftarrow 0$
10: end if
11: return $SLO_{drop}, underpov$
```

risk of SLA violations. The length of three upscale events is recommended to ensure that the previously collected metrics have an impact on future scaling action and minimize fluctuations caused by occasional sudden spikes in load.

$$expedite(SLO_{drop}, underpov, bndry_{slo}, bndry_{under}) = \begin{cases} true, & \text{if } SLO_{drop} > bndry_{slo} \\ true, & \text{if } underpov > bndry_{under} \\ false, & \text{otherwise.} \end{cases} \tag{2.37}$$

Here, $bndry_{slo}$ and $bndry_{under}$ identify the maximum number of SLO_{drop} and $underpov$ events that trigger the CT_d's recalculation earlier than by default.

$$timeToUpdate(SLO_{drop}, underpov, t_{updated}, t_{now}, T_{adjust}, T_{scale}) = \begin{cases} true, & \text{if } |t_{updated} - t_{now}| > T_{adjust} \\ true, & \text{if } |t_{updated} - t_{now}| > 3 \times T_{scale} \land expedite(SLO_{drop} \\ & underpov) = true \\ false, & \text{otherwise} \end{cases} \tag{2.38}$$

Here, $t_{updated}$ is the time when the CPU threshold was last updated, and t_{now} is the current time.

As seen from Algorithm 5, the boundary counters are dropped after the scaling action if SLO_{drop} and $underpov$ exceed their boundaries.

Algorithm 5 The dynamic threshold-adjustment algorithm.

```
Require: $T_{adjust}, T_{scale}, t_{updated}, t_{now}, SLO_{drop}, underpov, bndry_{slo}$,
    $bndry_{under}$
Ensure: Reset the $SLO_{drop}$ and $underpov$ counters to zero and return $CT_d$ if
    autoscaling action was triggered due to resource underprovisioning.
1: if $timeToUpdate(T_{adjust}, T_{scale}, t_{updated}, t_{now}, SLO_{drop}, underpov) =$
   $true$ then
2:     if $SLO_{drop} > bndry_{slo}$ then
3:         $SLO_{drop} \leftarrow 0$
4:     end if
5:     if $underpov > bndry_{under}$ then
6:         $underpov \leftarrow 0$
7:     end if
8:     return $CT_d$
9: end if
```

After determining if the system is underprovisioning or not, the target utilization threshold can be calculated and adjusted as described below.

Target Utilization Threshold Selection

As mentioned at the beginning of this sub-chapter, the prototype selects different target utilization thresholds depending on the operational state of the system.

This behavior is presented in Equation (2.39). When the algorithm initializes, M_{suff} is not collected yet. Therefore, the current CPU threshold (CT_{now}) is selected. However, if underprovisioning or an increase in SLA violations is detected ($SLO_{diff} < SLO_{drop_{Tupdate}}$), the current threshold is adjusted to a lower value using the approach outlined in Equation (2.40).

$$CT_d(SLO_{now}, SLO_{lastUpdate}, CT_{now}) =
\begin{cases}
CTR_{slo}, & \text{if } SLO_{now} \geq SLO_{tgt} \wedge \exists CTR_{slo} \\
CTR_{SLO_{expedite}}, & \text{if } expedite = true \wedge \exists CTR_{slo} \\
CT_{expedite}, & \text{if } \nexists CTR_{slo} \wedge expedite = true \\
CT_{expedite}, & \text{if } |M_{eval}| < |M_{suff}| \\
 & \wedge SLO_{diff} < SLO_{drop_{Tupdate}} \\
 & \wedge SLO_{now} < 80\% \\
CT_{now} \times \frac{SLO_{tgt}}{SLO_{now}}, & \text{if } |M_{eval}| < |M_{suff}| \\
 & \wedge SLO_{diff} < SLO_{drop_{Tupdate}} \\
 & \wedge SLO_{now} > 80\% \\
CT_{now}, & \text{otherwise}
\end{cases} \tag{2.39}$$

Here, $SLO_{diff} = SLO_{now} - SLO_{lastUpdate}$ is the difference between the SLO values collected during the last threshold-adjustment action ($SLO_{lastUpdate}$) and the currently collected SLO value (SLO_{now}); $SLO_{drop_{Tupdate}}$ is acceptable SLO decrease per evaluation period.

The $CT_{expedite}$ results in the provisioning of more replicas if the current threshold setting leads to a decrease in SLA performance. For instance, if a target threshold of 50% is set and utilization is 100%, then the number of replicas will increase at a maximum of twice during each scale iteration as per Equation (1.1) ($\lceil (M_m/M_d) \times R_n \rceil = \lceil 100/50 \times R_n \rceil = 2 \times R_n$).

$$CT_{expedite} = 100 \div \left\lceil \frac{100}{CT_{now}} + 1 \right\rceil \tag{2.40}$$

To expedite the recovery of the service level as per the SLA, the M_d will be replaced with threshold $CT_{expedite}$ set at 33%, resulting in the provision of three times the number of replicas ($CT_{expedite} = 100 \div \lceil 1 + 100/50 \rceil = 100 \div 3 \approx 33$). In case the SLA value does not start increasing, the following threshold will be set at 25% ($100 \div \lceil 1 + 100/33 \rceil = 25$), leading to an increase of four times the number of pod replicas provisioned. This process will be repeated until the measured SLO value stops declining.

$CT_{expedite}$ is selected when the service's SLO has dropped below 80%. This helps to prevent an infinite increase in replicas. Setting a higher threshold would not have any impact due to the Kubernetes tolerance threshold setting. The tolerance threshold ensures that, the HPA algorithm will perform the scaling only if the ratio between M_d and M_n is less than 0.9 or larger than 1.1 (kubernetes.io, 2022).

It is worth noting that the calculation of the $CT_{expedite}$ threshold was introduced because the HPA could not break free from the failure loop when the threshold was set above 50%. This was due to an inadequate increase in the number of replicas. This limitation occurred because each autoscaling interaction could only double the number of replicas. Setting the threshold below 33% allowed the HPA to successfully escape the failure loop, with the number of replicas increasing by up to three times in each scaling action.

When enough metrics (M_{suff}) are gathered, the algorithm estimates the threshold CTR_{slo} using the method described in sub-chapter 2.2.2. If the algorithm detects resource underprovisioning, it will select a threshold lower than the current threshold ($CTR_{SLO_{expedite}}$) from the SLO_r values (Equation (2.41)).If there are no suitable CTR_{slo} or $CTR_{SLO_{expedite}}$, then CT_{now} or $CT_{expedite}$ is selected, respectively.

$$CTR_{SLO_{expedite}} = \max_{CTR_{slo} < CT_{now}} \{f(SLO_r) : SLO_r \geq SLO_{tgt} \text{ exists}\} \quad (2.41)$$

It is important to mention that period, T_{adjust}, should span a minimum of 3 to 5 upscale periods. This ensures that the system can more accurately evaluate the impact of changes in the previous threshold. If the update period is too short, the algorithm will become overly sensitive to load fluctuations. If the update period is too long, the system will operate at lower thresholds for an extended period, leading to increased overprovisioning of the resources. However, it will be less sensitive to accidental load spikes. A similar logic should be applied to threshold evaluation intervals. The threshold evaluation intervals should also be set to last a minimum of 3 to 5 threshold evaluation periods (T_{eval}).

2.2.4. Threshold-Adjustment Algorithm Evaluation Criteria

The main goal of the adaptive threshold algorithm is to ensure that the autoscaler provides sufficient resources to ensure compliance with the defined service level objectives. Three criteria were used to assess the algorithm's ability to achieve this goal.

First, the total number of containers used in each monitoring period is calculated to measure the efficiency of resource provisioning (Equation (2.42). Fewer containers are considered to be a better result, but only when the defined SLO is met. The over- and under-provisioning are calculated by comparing the number of resources used by the HPA. The threshold for HPA is set as closely as possible to the value that allows the system to provision a sufficient number of replicas to operate at a performance level where the number of violations does not exceed the maximum allowed.

$$P_{total} = \sum_{t=1}^{n} M(P_t) \tag{2.42}$$

Here, P_{total} is the total number of pods reported at each monitoring data point, n is the number of data points, and $M(P_t)$ is the number of ready pods reported at time t.

Additionally, the tables presenting the experiment results in sub-chapter 3.3 include the percentage difference from the best result for total pods. This data helps to reduce the cognitive load on the reader by eliminating the need to calculate and estimate the number of pods that were overprovisioned compared to the experiment that achieved the lowest number of pods necessary to fully meet SLO compliance across the entire experiment. Let's denote the percentage difference from the best result for total pods as $P_{bestdiff}$. Then, the value $P_{bestdiff}$ is calculated as presented in the following equation :

$$P_{bestdiff} = \left\lceil \frac{P_{total}^{N} - P_{total}^{min}}{P_{total}^{min}} \times 100\% \right\rceil . \tag{2.43}$$

Here, P_{total}^{min} is a minimal number of total pods across all experiments where SLO compliance was fully met, while P_{total}^{N} represents any other experiment in the relevant experiment set.

The second evaluation criterion is the accuracy of the algorithm, that is, the ability to operate as close as possible to the defined SLO (SLO_{tgt}), which is measured using the symmetric Mean Absolute Percentage Error (sMAPE) (Chen, Twycross, & Garibaldi, 2017) (Equation (2.44)). This helps understand if overprovisioning of the resources is justified (Dang-Quang, Yoo, De, & Santana, 2021).

$$sMAPE = \frac{100\%}{n} \sum_{t=1}^{n} \frac{2 \times (|M(SLA)_t - SLO_{tgt}|)}{|M(SLA)_t| + |SLO_{tgt}|} \tag{2.44}$$

Here, n is the number of data points, and $M(SLA)_t$ denotes the SLA value at the time t.

The algorithm's ability to meet the defined SLO is the third and most important criterion of the evaluation. It is important to mention that each experiment evaluating the algorithm includes a period (M_{suff}) required to collect sufficient events for threshold estimation and the time needed for the system to change its behavior (4–5 upscale actions). Metrics collected during this period are excluded from the evaluation criteria to assess the ability of the algorithm to meet the defined SLOs.

The upcoming chapter outlines the experimental environments and experiments conducted to assess the proposed threshold-detection approach against evaluation criteria. The following sub-chapter concludes the results presented in this chapter.

2.3. Conclusions of the Second Chapter

The chapter presented two SLA-aware autoscaling approaches for containerized cloud-native applications that aim to fulfill performance-based SLA requirements. The following conclusions have been drawn:

1. The SAA solution is designed to tackle various challenges that autoscaling systems must address to ensure compliance with SLAs. To achieve this, multiple modules have been developed to handle issues, such as adapting to changes in load, accommodating diverse application resource and performance requirements, adjusting to the non-uniformity of cloud resources, ensuring timely scaling decisions, and mitigating oscillations. Additionally, SAA includes logic that aims to restore SLA values to their target levels after they have been breached. In comparison to other solutions, as per the conducted literature review, only SAA is designed to ensure desirable service levels by actively working to restore service quality and achieve the defined SLO. This capability represents a key strength of the SAA solution and highlights its potential to recover SLOs in scenarios where adding resources can significantly enhance performance.
2. This chapter presented a threshold-detection approach and SATA prototype for dynamic threshold adjustment to enhance SLA fulfillment in autoscalers for cloud native applications. The approach is based on data explanatory analysis and moving average smoothing, which helps to understand and implement the solution without extensive knowledge of machine learning.
3. The evaluation criteria are proposed for each approach. These criteria enable the assessment of solutions from two perspectives: the efficacy of SLA fulfillment and resource management efficiency.

3

Experimental Investigation and Evaluation of Service Level Agreement-Adaptive Rules-Based Autoscaling Algorithms

The chapter presents the experimental investigation of two proposed SLA-adaptive rules-based autoscaling solutions discussed in the previous chapter: SLA-Adaptive Autoscaler (SAA) and SLA-adaptive threshold adjuster (SATA). It also covers the experimental evaluation of the proposed methods and compares those with state-of-the-art autoscaling solutions. The chapter includes details on the practical implementation of each solution and experimental environment. Finally, it concludes by evaluating the tested autoscaling approaches and the results achieved.

The primary research findings of this chapter were published in two publications (Pozdniakova et al., 2023, 2024) prepared by the author of this dissertation.

3.1. Common Experimental Setup

This sub-chapter outlines the common components utilized in all of the experiments. Although different configuration parameters were established based on each scenario, the corresponding prototypes and supporting components remained consistent throughout all the experiments. Detailed information about the settings can be found in the relevant sub-chapters.

Azure Kubernetes Service (AKS) (Microsoft, 2024a) served as the test bed for the experimental evaluation. The master node utilized a “Standard DS2 v2” instance, which includes two vCPUs and seven GiB of RAM for all experiments. The number of worker nodes varied in each experiment, with more detailed information on exact configuration and setting provided in the corresponding sub-chapters of this chapter.

A virtual machine instance of “Standard B4ms” size, which provided four vCPUs and 16 GiB of RAM, was used to host load-generating tools. The virtual machine ran the Ubuntu 20.04 operating system.

The open-source Prometheus solution (Prometheus, 2025) was used to collect application performance data in the implemented solution.

The following metrics were scraped from the target application: the transactions count and the duration of requests, measured using a histogram. Metrics were collected every 5 seconds, and their evaluation was done every 30 seconds ($T_m = 30$). The average CPU utilization of a pod and the number of running pods were collected from Kubernetes exposed endpoints at a frequency of every 10 seconds. Such a short scraping interval enables a prompt response to load changes.

The Azure load balancer (Microsoft, 2024c) was utilized to provide access from the machine hosting the load-generating tools to the applications hosted on the Kubernetes cluster.

It is important to note that in all scenarios presented in this dissertation, the load was measured by the number of requests per second (rps). However, another type of workload, such as the number of connections or throughput, can be used instead of rps as long as it strongly correlates with the utilization metric (Toka et al., 2021) and SLO.

To evaluate the performance of the autoscaling solution according to the evaluation criteria described in Chapter 2, the assessment was conducted considering the methodological principles for reproducible performance evaluation in cloud computing proposed by (Papadopoulos et al., 2021).

The graphical analysis method was utilized to evaluate the scattering between results together with confidence interval calculation for experiments that ran more than once. For example, the SATA algorithm was assessed using the WorldCup’98 and Electronic Data Gathering, Analysis, and Retrieval (EDGAR) (SEC.gov, 2025) workload described in sub-chater 3.3.2. In the evaluation of the SATA algorithm, the achieved SLA results for the WorldCup’98 load using SMA $4{\times}10^1$ varied in range from $\tilde{1}\%$ to $\tilde{2}\%$ with a confidence interval ranging in the same diapason, while for EDGAR workload, the results were even more precise and ranged from 0% to less than 0.5% with a confidence interval ranging between 0.1% and 0.7% (Figs. 3.1a and d, 3.3, 3.4).

The achieved pod replicas utilization results for the WorldCup’98 load using the SMA $4{\times}10$ varied in the range from 0 to 10 pods ($\tilde{3}0\%$) with a confidence

[1]The SMA $4{\times}10$ notation is used later in this document. It is described in more detail in sub-chapter 3.3. Here, SMA refers to the Simple Moving Average smoothing technique; $4{\times}10$ in this context represents the threshold adjustment period set to 4 autoscaling periods, while the threshold evaluation period is defined as 10 autoscaling periods in length.

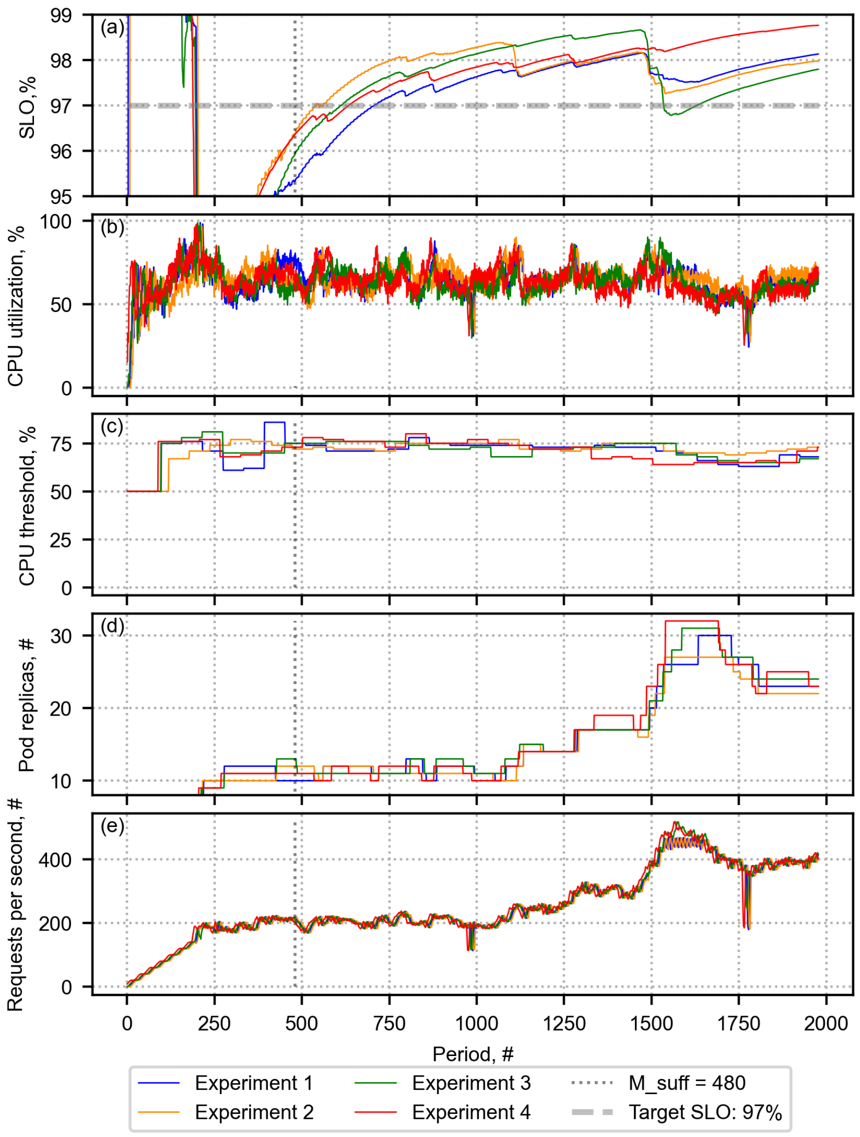


Fig. 3.1. Line plots presenting the scattering of results across four experiments assessing the performance of the SLA-Adaptive Threshold Adjustment (SATA) solution with Simple Moving Average (SMA) in the WorldCup'98 workload scenario. (a) SLO value after collecting a sufficient number of events (after the dotted line). (b) Average CPU utilization. (c) Applied target utilization threshold in each period. (d) Number of pods provisioned in each period. (e) Generated workload requests per second

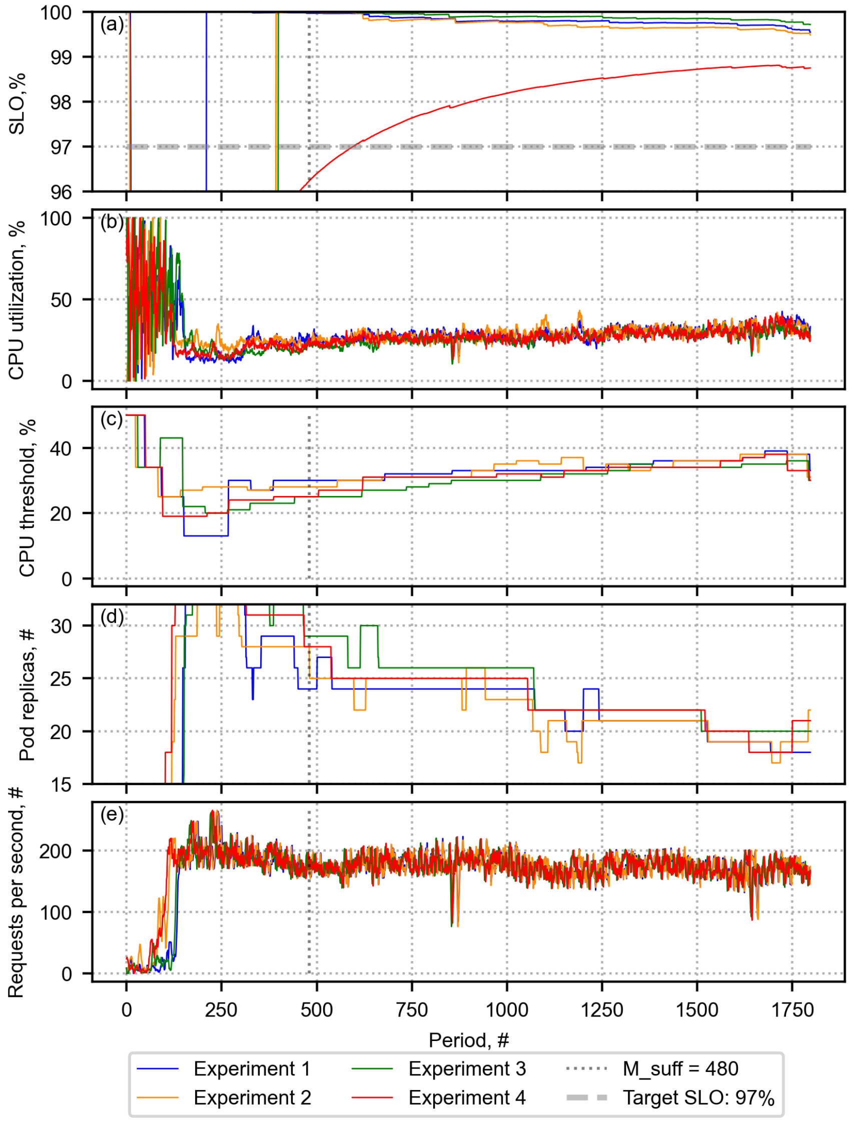


Fig. 3.2. Line plots presenting the scattering of results across four experiments assessing the performance of the SLA-Adaptive Threshold Adjustment (SATA) solution with Simple Moving Average (SMA) in the EDGAR workload scenario. (a) SLO value after collecting a sufficient number of events (after the dotted line). (b) Average CPU utilization. (c) Applied target utilization threshold in each period. (d) Number of pods provisioned in each period. (e) Generated workload requests per second

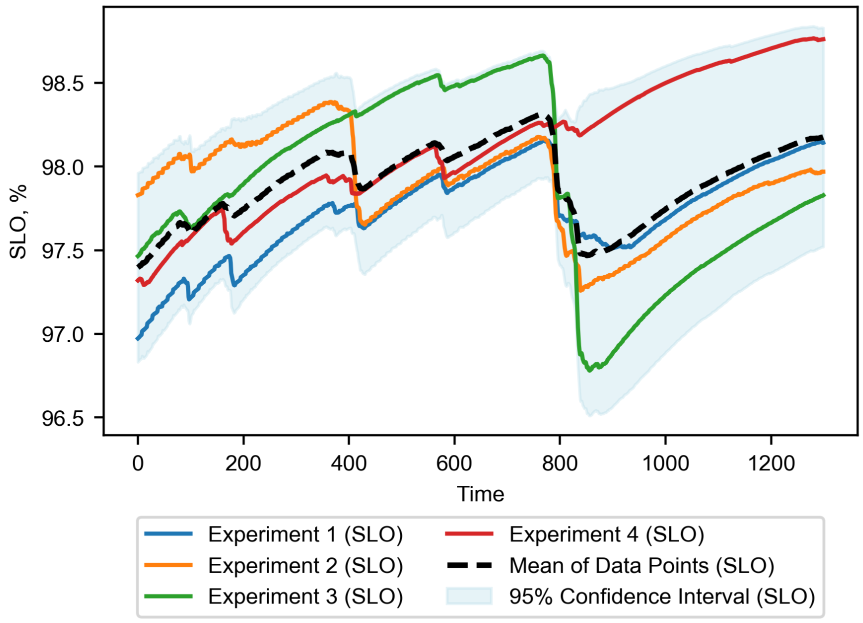


Fig. 3.3. Line plots presenting the mean and confidence intervals of Service Level Objective values across four experiments assessing the performance of the SLA-Adaptive Threshold Adjustment (SATA) solution with Simple Moving Average (SMA) in the WorldCup'98 workload scenario

interval ranging from zero to 6 pods ($\tilde{2}3\%$) and at an average of 2 pods. For EDGAR, the results ranged from 0 to 5 (20%) pods with a confidence interval ranging between 0 and 6 ($\tilde{2}3\%$) at an average of around 1.8 pods (Figs. 3.2a and d, 3.5, 3.6).

3.2. Experimental Evaluation of the Service Level Agreement-Adaptive Autoscaling Algorithm

This sub-chapter describes the experimental evaluation of the SLA-Aware Autoscaling Algorithm (SAA) (Pozdniakova et al., 2023). The sub-chapter includes the prototype implementation and the experimental environment details, including workloads and the experiments' results. The experimental investigation also compares the proposed method with state-of-the-art rules-based autoscaling solutions.

The SAA prototype was developed in Java programming language to evaluate the proposed SAA solution and its algorithms, which were described in detail in the previous sub-chapter. Figure 3.7 shows the component diagram of the SAA so-

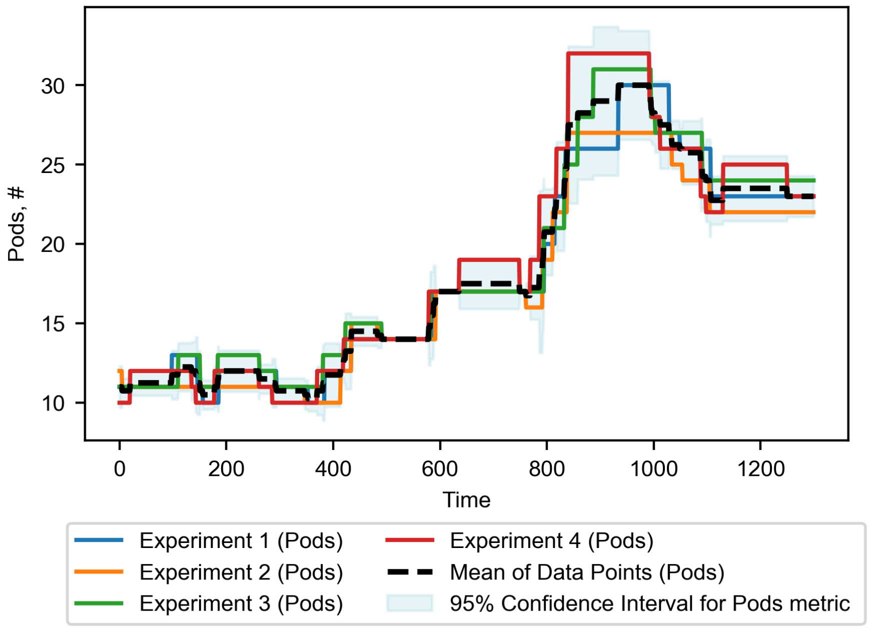


Fig. 3.4. Line plots presenting the mean and confidence intervals of a number of pods used across four experiments assessing the performance of the SLA-Adaptive Threshold Adjustment (SATA) solution with Simple Moving Average (SMA) in the WorldCup'98 workload scenario

lution implementation, illustrating the different components and their interactions. The modules were implemented in the respective Java classes. The source code of the solution can be accessed in the GitHub repository (Pozdniakova, 2023).

In addition to the modules discussed in sub-chapter 2.1.2, the experimental environment incorporated two extra modules: the metrics processor and the values storage module. These modules collect, pre-process, and store the application and infrastructure metrics the SAA solution needs. The metrics processor gathers and prepares the metrics for analysis and decision-making. The values storage module enables the exchange and storage of these metric values among system modules.

The SAA used Custom Pod Autoscaler (CPA) middleware to integrate the Kubernetes API server. The CPA periodically retrieves information about the replicas running in the Kubernetes cluster at intervals (T_m) and shares it with the SAA solution. If the number of replicas reported by the CPA and Prometheus components is equal, the SAA autoscaling algorithm initiates the autoscaling process according to the established rules and thresholds. The synchronization activities among these components added a 20-second delay to the autoscaling logic execution.

Two types of workloads were utilized to assess algorithm performance: man-

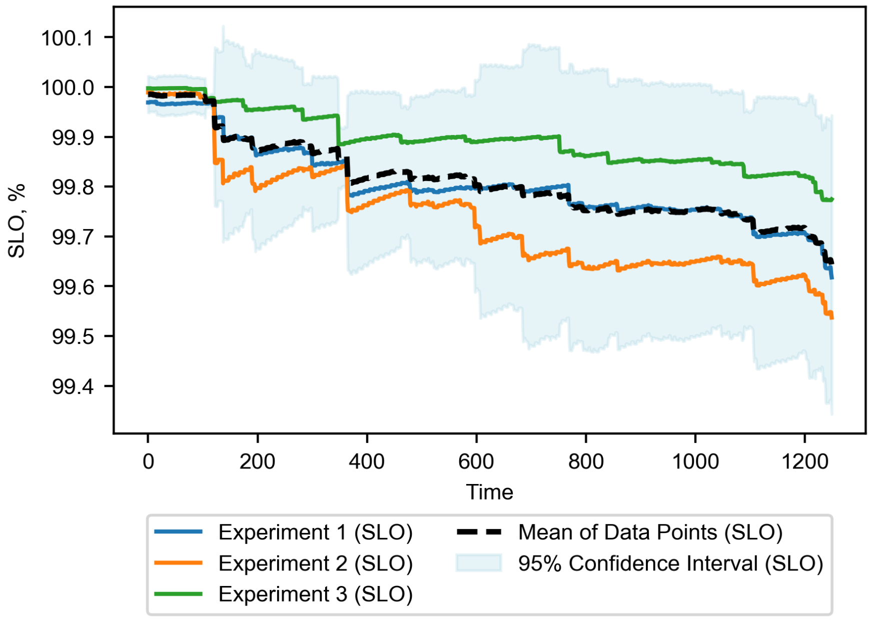


Fig. 3.5. Line plots presenting the mean and confidence intervals of Service Level Objective values across four experiments assessing the performance of the SLA-Adaptive Threshold Adjustment (SATA) solution with Simple Moving Average (SMA) in the EDGAR workload scenario

ually created synthetic patterns and real-world web traffic traces. The synthetic workload tested the capability of each solution under specific conditions, while the real-world web traffic traces provided an evaluation of algorithm performance under conditions closely resembling real-world scenarios.

The next sub-chapter presents experimental environments in more detail, starting with experiments utilizing synthetic workloads.

3.2.1. Experimental Setup for Autoscaler Performance Evaluation with Synthetic Workloads

This sub-chapter outlines the synthetic workloads and infrastructure components employed in the experimental environment used for the empirical evaluation of the proposed autoscaler solution.

The solution ran on the version 1.19 AKS cluster. The deployment consisted of four virtual machines: one master node and three worker nodes. The operating system used across all machines was Ubuntu 18.04. The worker nodes were configured as “Standard D8 v5” instances, providing eight vCPUs and 32 GiB of

RAM to serve the application load. Two CPUs in this cluster were allocated to run the monitoring and autoscaling solutions, such as Prometheus, CPA, and other related components. This allocation ensured that the monitoring and autoscaling components had dedicated resources to perform their tasks without impacting the resources available for the application.

The Apache JMeter tool (JMeter, 2025) was used as a load generator to simulate different workload patterns. Additionally, the JMeter Constant Throughput Timer add-on was employed to maintain a consistent and predictable workload. The Jmeter was configured to generate a maximum load of 9000 requests per minute or 150 requests per second on average ($\max L_n = 150$). This request rate was selected to ensure that the system can scale to at least three times the number of replicas needed to maintain the service level defined in the SLA.

Five synthetic workload patterns were generated using different levels of velocity and volatility. These patterns included slowly changing load ($|\alpha| \leq 1$)[2], moderately changing load ($1 < |\alpha| \leq 3$), fast-changing load ($|\alpha| > 3$), peaks with

[2]Here, $|\alpha|$ represents the load velocity factor discussed in the second chapter.

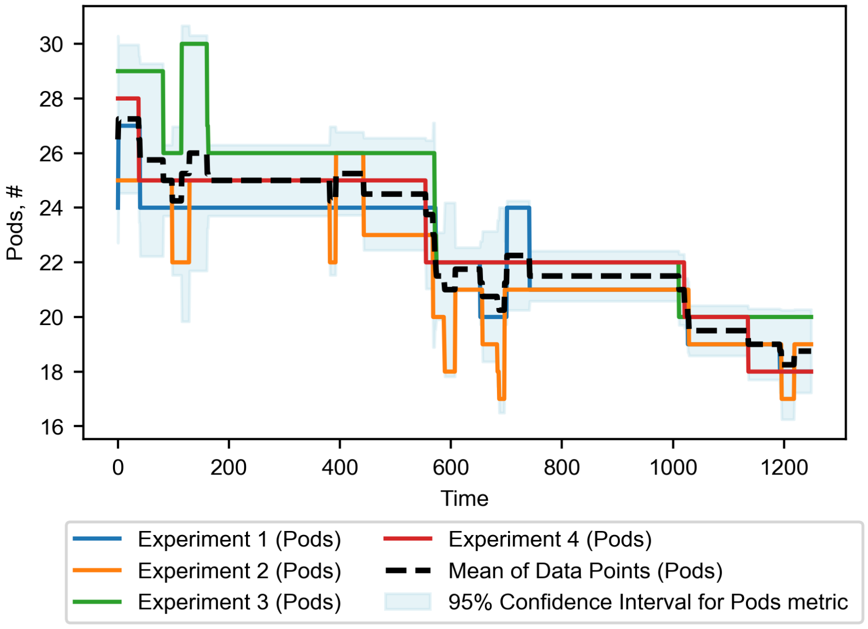


Fig. 3.6. Line plots presenting the mean and confidence intervals of a number of pods used across four experiments assessing the performance of the SLA-Adaptive Threshold Adjustment (SATA) solution with Simple Moving Average (SMA) in the EDGAR workload scenario.

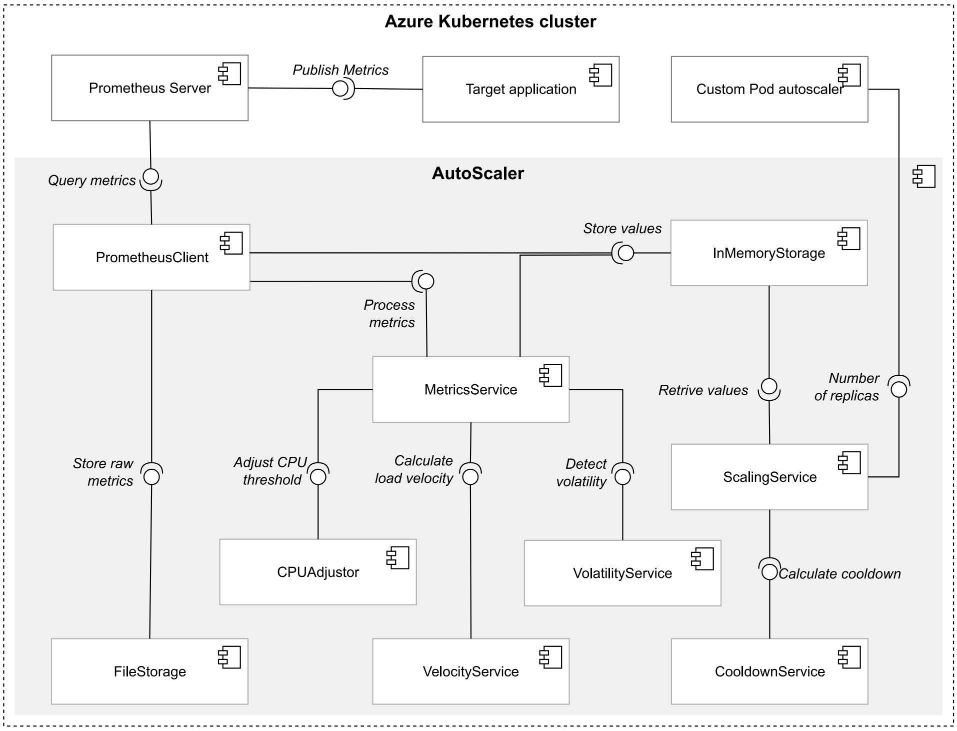


Fig. 3.7. Component diagram of the Service Level Agreement-Adaptive Autoscaling Solution (made by the author)

pauses, and a mix of various traffic patterns. These workload patterns, representing diverse application loads with different volatility levels, allowed for evaluating autoscaling solutions' performance and provided insights into these solutions' behavior under varying workload conditions.

To evaluate the behavior of the solutions discussed in this study under varying load conditions, a CPU-intensive application was developed using the Java programming language. For each incoming request, it calculates the factorial of 8000, generating significant CPU usage. When receiving an HTTP request, a random delay is introduced to the response time to mimic interactions with external systems, such as databases. This approach creates operational conditions close to a real-world environment where communication processes involve integrating with external systems.

The application was deployed as a Docker container with 1 GiB of memory allocated to it, along with specified limits of 1 CPU. Additionally, readiness probes were configured to ensure that data was not sent to replicas that were not in a "Not ready" state. The average time required for the application to prepare to serve a load after startup was 20 seconds.

A load test was conducted to create the application's performance profile. JMeter generated a load that increased linearly. A one-and-a-half-second response time set was the SLO target (SLI_{tgt}). Response time remained below the target value when CPU utilization reached 65% at a rate of 23 rps. The benchmarking results indicated that the application's performance declined exponentially once the load was higher than 23 rps and requests either exceeded 1.5 seconds in processing time or failed to be processed. The duration of requests was reported using a histogram with the following four buckets: 1 s, 1.5 s, 2 s, and > 2 s.

3.2.2. Experimental Setup for the Autoscalers Performance Evaluation under Real-World Workload Conditions

This sub-chapter presents the setup used to evaluate systems performance as closely as possible to the real-world workload conditions, referred to in this work as a real-world workload scenario. The sub-chapter presents only differences from the environment described in sub-chapter 3.2.1.

A larger AKS cluster, version 1.25, was deployed to evaluate autoscaling solution performance in real-world workload scenarios. The deployment comprised six virtual machines, including one master node and five worker nodes. All machines used Ubuntu 22.04 operating system. The worker nodes were configured as "Standard D8s v5" instances, providing eight vCPUs and 32 GiB of RAM.

The system was set up to handle a maximum of 55 pod replicas on a four-node, 8-CPU cluster. One CPU per worker node was reserved for running the control plane components. Additionally, a separate worker node was designated for monitoring tools and autoscaling solutions like Prometheus, CPA, and other related components. This allocation guaranteed that the monitoring and autoscaling mechanisms had dedicated resources to carry out their tasks effectively without affecting the resources available for the application replicas.

JMeter, which is a closed system load generator, was replaced with Gatling. Gatling was set up to function as an open model workload generation tool to ensure the load follows real-world trace patterns. In an open system model, new requests can arrive independently of completions, and the system has no negative feedback. In contrast, in a closed model system, such as JMeter, a new request is only initiated once the previous request has been completed (Schroeder, Wierman, & Harchol-Balter, 2006).

The two real-world workload scenarios were evaluated, presenting two types of workload patterns:

- mixed load pattern is based on 1998 World Cup website access logs (referred to as WorldCup'98 in this work) (Arlitt & Jin, 2000);
- shaky load pattern is based on the access logs of the Electronic Data Gathering, Analysis, and Retrieval (EDGAR) system's website.

The WorldCup'98 dataset used in this work contains all the requests made to the 1998 World Cup Website between 30 April 1998 and 26 July 1998 and is commonly referenced in research articles (Bogachev, Kuzmenko, Markelov, Pyko, & Pyko, 2023; Dang-Quang et al., 2021; Imdoukh, Ahmad, & Alfailakawi, 2020; Taherizadeh & Stankovski, 2019; Ye et al., 2017). The load generated by the WorldCup 98 watchers on the 78th day from 19:00 to 22:00 was selected for the experiment. This pattern contains sudden high-load spikes followed by a stable load with minimal variation.

The EDGAR dataset consists of logs from the Electronic Data Gathering, Analysis, and Retrieval system. It is a public database that allows users to research, for example, a public company's financial information and operations by reviewing the filings made by the company with the U.S. Securities and Exchange Commission. The dataset used in this study includes all search requests for filings stored in EDGAR and made through the SEC.gov site on June 30, 2023, between 2:00 AM and 5:00 AM. The load can be described as a lightly shaking load with consistent small fluctuations of 10 to 15% in magnitude and occasional large fluctuations of 100 to 200% in magnitude.

Another CPU-intensive application was developed in the Rust programming language to assess the solutions evaluated in this work (Pozdniakova, 2024). When the application receives a request, it calculates the hash function of an arbitrary number, resulting in a high CPU workload. This imitates document encryption services. The arbitrary number is selected to ensure that the system simulates a real-world environment where requests to the API might require different processing times and resources. An arbitrary delay is added to the response time for the same reasons as in the Java Application case.

The application was deployed as a Docker container. The Rust programming language allowed for more efficient CPU thread usage, which resulted in more containers being deployed on clusters of similar size. The container was assigned the limits of 250 ms and 512 MiB of memory. The average time required for the application to prepare to serve a load after startup was one second, which is twenty times faster than the Java application.

An application performance profile was generated during a load test using Gatling, following the same approach as in the Java Application case. The application resource congestion point was found to be around 85% of CPU utilization. The target of 400 millisecond response time was set as SLI. It was observed that the target SLO was on track value until CPU utilization reached approximately 92% at a rate of 24 rps on average. The duration of requests was measured and reported using a histogram with the eight buckets: 100 ms, 150 ms, 200 ms, 400 ms, 500 ms, 1 s, 2 s, and > 2 s.

Table 3.1 presents a summary of the differences between Java and Rust applications and their settings used throughout experiments.

Table 3.1. Experimental applications characteristics

Application	Java	Rust
Target SLI (SLI_{tgt}),ms	1500	400
Target SLO (SLO_{tgt}),%	98	97
Startup Time (t_{delay}), s	55	1
CPU saturation point, %	65	85
Maximum rps ($\max\limits_{SLO_n \geq SLO_{tgt}} RPS$), rps	23	24
Reported histogram buckets values, ms	1000, 1500, 2000, and > 2000	100, 150, 200, 400, 500, 1000, 2000, and > 2000
Container resource limits	CPU: 1000 ms, Memory: 1000 MiB	CPU: 256 ms, Memory: 512 MiB

Before concluding this sub-chapter, it is important to note that a lower SLO target of 97% was set to create a buffer, allowing for better assessment of overprovisioning's impact on service quality. A 100% target would not reveal whether the system operated at the SLO threshold or provided significantly higher quality than was expected.

The following sub-chapter presents the rules-based state-of-the-art autoscaling solution used for performance comparison of the proposed algorithm.

3.2.3. Autoscaler Solutions Used for the Comparative Analysis

A comparative assessment against two existing rule-based autoscaling solutions was conducted to evaluate the effectiveness of the proposed solution in achieving SLO recovery and maintenance: Dynamic Multi-level Autoscaling Rules for Containerized Applications (DMAR) and the Horizontal Pod Autoscaler (HPA), which were discussed in the first chapter.

Even though several differences exist between DMAR and SAA, the evaluation included DMAR because it utilizes scaling methods and indicators similar to those in the SAA approach. DMAR was also compared to seven state-of-the-art rules-based autoscaling approaches and demonstrated the best results in efficiency and its ability to maintain quality of service (QoS). As a result, it is considered relevant for comparison in this research. There are key differences between DMAR and SAA. DMAR uses average response time to adjust CPU thresholds, whereas SAA relies on the SLO value for this adjustment. Additionally, DMAR does not consider load patterns and fluctuations, while SAA does.

Other researchers, including DMAR authors, commonly use HPA as a solution in their comparative analyses (Ding & Huang, 2021; Hu & Wang, 2021; Ju et al., 2021; Mirhosseini et al., 2021; Toka et al., 2021; Verreydt et al., 2019). This work includes HPA as a baseline for assessing SAA's performance in relation to DMAR and other solutions. The comparison intends to emphasize the advantages and enhancements offered by the SAA method concerning SLO management, with primary consideration given to SLA fulfillment.

The motivation for choosing two solutions was driven by the intention to test as many prospective solutions as possible. The next sub-chapter provides configuration details of each of the evaluated solutions.

The following text presents the settings used for each of the solutions in different workload evaluation scenarios, along with the rationale for specific settings selections.

Configuration Parameters of Evaluated Autoscaler Solutions Used in Synthetic Workload Scenario

This part outlines the configuration parameters employed for the evaluated solutions during the synthetic workload experiment.

Table 3.2 presents the parameters configured for each of the evaluated autoscaling solutions.

All solutions were configured with the same minimum and maximum number of replicas and the same default CPU value. The maximum number of replicas was limited by cluster size. The target application's benchmark results were used to identify the default CPU threshold. The benchmark showed that the system met the target SLO as long as the average CPU usage of each pod remained below 65%. As a result, a 60% CPU threshold was set for all the autoscaling solutions tested. This configuration provided a 5% buffer, allowing room for the autoscaling solution to react and provision additional resources before breaching the SLO target.

The DMAR solution used a response time threshold of 500 ms as the autoscaling threshold. The threshold is used to trigger autoscaling actions before resource congestion occurs. This threshold is two times lower than the response time defined as the target SLO. Since the target application is CPU-bound, no memory utilization-based autoscaling threshold was implemented, as it is irrelevant for autoscaling decisions. The remaining settings were kept at the default values recommended by the authors of the DMAR approach in their work (Taherizadeh & Stankovski, 2019).

The HPA stabilization windows were set to ten for upscaling and 20 for downscaling, which enables faster provisioning of replicas and slower deprovisioning. These settings were chosen based on insights provided by (Nguyen et al., 2020; Taherizadeh & Grobelnik, 2020) to minimize the risk of SLO violations.

The SAA employed a 10-second adjustment step for the calculation of the dynamic cool-down period, which resulted in a length ranging from 10 to 30 seconds. These values were minimized, as the synchronization of replica numbers between

Table 3.2. Settings of the evaluated autoscaling solutions

Autoscaler	SLA Adaptive Autoscaler		DMAR	HPA
Parameter	Values			
Service Level Objective (SLO_{tgt}), ms	98%			
Service Level Indicator Target (SLI_{tgt}), ms	1000			
Max replica number (R_{max})	22			
Min replica number (R_{min})	1			
Default CPU threshold, %	60			
Autoscaling decision adaptation period (T_n), s	10		10	60[a]
Response time (Scaling Indicator), ms	–		500	–
Conservative Constant	–		5	–
Cool-down/stabilization period, s	–		30	Upscale:10, Downscale: 20
Tolerable throughput (L_{max}), rps	23		–	–
CPU adjustment period (T_c), s	600		–	–
Velocities vector ($V_k(A_k)$) size (K)	60		–	–
Velocity factor (α) to velocity level mapping	Stable: $\lvert\alpha\rvert \leq 1$, Moderate: $1 < \lvert\alpha\rvert \leq 2$, High: $\lvert\alpha\rvert > 2$		–	–
Scaling direction[b]	Upscale	Downscale	–	
$T^D_{cooldown_0}$, $\Delta T^D_{cooldown}$, s	40, 10	0, 10	–	–
$t^D_{totalDelay}$ (t^D_{delay} $+\max(T^D_{cooldown})$ $+T_m$), s	75 (20 + 50 + 5)	55 (0 + 50 + 5)	–	–
CPU adjustment step below SLO ($-\Delta C^D$), %	5	3	–	–
CPU adjustment step above SLO (+ΔC^D), %	3	2	–	–

[a]By default, the highest autoscaling action frequency of the HPA on Azure Kubernetes Service is one autoscaling action once per 60 seconds. The minimal stabilization window length is set to allow a faster reaction to changes.

[b]$D \in [up, down]$ denotes the autoscaling action directions, specifically upscale or downscale.

End of the Table 3.2

Autoscaler	SLA Adaptive Autoscaler		DMAR	HPA
Parameter	Values			
CPU Thresholds ranges [min,max] ($[C_{min}^{CT^D}, C_{max}^{CT^D}]^a$), %	Upper: [70,85], Mid: [65,80], Lower: [60,75]	Upper: [30,40], Mid: [25,35], Lower: [20,30]	–	–
Downscale step when load is volatile ΔR_V	–	0.1	–	–
SLO threshold without impact factor (SLO_{BP})	95%	–	–	–
No down scale action SLO threshold ($SLO_{noDownScale}$)	–	1.0025	–	–

[a]CT^D represents the identifier of the selected CPU threshold, where $CT \in [upper, mid, lower]$.

Prometheus and Kubernetes created a delay of at least 20 seconds for any autoscaling action. As a result, the total stabilization period was set to 30–50 seconds.

The value of $t_{totalDelay}^{D}$ was calculated based on monitoring and benchmark parameters presented in sub-chapter 3.2.1. The CPU adjustment interval T_c used by the DCTA module was set to ten minutes. This relatively short interval was chosen in the experimental environment. The selected CPU adjustment interval of ten minutes in the experimental environment was intentionally chosen to facilitate frequent threshold changes and to observe their effects on the stability of the autoscaler.

In real-world applications, the CPU adjustment interval can often be much longer, typically ranging from six to 12 hours or more, especially when the SLA compliance is measured over a longer time frame, such as a month. A more extended observation period is more effective in identifying trends in SLO decreases rather than just temporary drops in performance. The lower CPU threshold of 60% was specifically chosen to be used only during periods of significant load increase. In most other scenarios, higher CPU thresholds triggered upscaling actions. This approach ensured that the system performed upscaling actions when CPU utilization reached or exceeded 70%, thereby maintaining SLA compliance.

The decrease step was established as 10% of the replicas to enable effective downscaling in a volatile load environment. When the system is scaled to a relatively small number of replicas (fewer than ten), the solution will reduce the number of replicas by at most one. This strategy minimizes the risk of violating SLOs while ensuring sufficient resource allocation.

Configuration Parameters of Evaluated Autoscaler Solutions in the Real-World Workloads Scenario

This part highlights the parameter changes made to the evaluated autoscaling solutions during real-world workload scenario assessment. Evaluating solutions in these scenarios facilitates a more comprehensive evaluation of their performance across diverse environments and with various target applications.

The default CPU threshold of 75% was intentionally selected for use across all solutions because it is below the congestion point of 85%. This allows for a 10% buffer for making autoscaling decisions. However, below the 92% CPU threshold at which SLO was below the target.

The DMAR solution used a response time threshold of 130 ms for autoscaling. The threshold triggers autoscaling when resource congestion occurs, but the SLO target is not violated yet (giving a 270 ms buffer for autoscaling decision-making).

The SAA utilized a lower CPU threshold of 75% only when the load increase was high. In other situations, it used higher CPU values to trigger upscaling actions. As a result, most of the time, SAA operated at an upper threshold, approaching levels where resource congestion could occur. Upscaling actions happened when the CPU utilization was at or above 85%. Despite these challenging conditions, SAA successfully maintained compliance with the SLA. The value of ($t^{D}_{totalDelay}$) was adjusted based on the benchmarking results of the Rust application presented in the sub-chapter 3.2.2. The remaining settings were unchanged as presented in Table 3.3 (normal text).

The next sub-chapter presents and discusses the results of evaluating the autoscaling solutions discussed here under various synthetic and near real-world workload conditions.

3.2.4. Results of Experimental Evaluation

The quantitative evaluation results of the HPA, DMAR, and SAA solutions are provided in this sub-chapter. The evaluation approach, outlined in sub-chapter 2.1.3, was used to assess the performance of the solutions. Further details, including the implementation of DMAR, the HPA configuration manifest, JMeter and Gatling test plans, and the collected experimental data, are available on the GitHub repository (Pozdniakova, 2023).

The experimental results were divided into three datasets representing the QoS recovery and the QoS support phases (Table 3.4) and whole experiment results (Table 3.5).

Table 3.3. Evaluated autoscaling solutions settings used in the real-world workload pattern evaluation scenario

Autoscaler	SLA Adaptive Autoscaler		DMAR	HPA
Parameter	Values			
Service Level Objective (SLO_{tgt}), ms	**97%**			
Service Level Indicator Target (SLI_{tgt}), ms	**400**			
Max replica number (R_{max})	**55**			
Min replica number (R_{min})	1			
Default CPU threshold (CT^{Up}, CT^{Down}), %	**75**			
Autoscaling decision adaptation period (T_n), s	10		10	60[a]
Response time (Scaling Indicator), ms	–		**130**	–
Conservative Constant	–		5	–
Cool-down/stabilization period, s	–		30	Upscale:10, Downscale: 20
Tolerable throughput (L_{max}), rps	**24**		–	–
CPU adjustment period (T_c), s	600		–	–
Velocities vector's ($V_k(A_k)$) size (K)	60		–	–
Velocity factor (α) to velocity level mapping	Stable: $\lvert\alpha\rvert \leq 1$, Moderate: $1 < \lvert\alpha\rvert \leq 2$, High: $\lvert\alpha\rvert > 2$		–	–
Scaling direction[b]	Upscale	Downscale	–	
$T^{D}_{cooldown_0}$, $\Delta T^{D}_{cooldown}$, s	40, 10	0, 10	–	–
$t^{D}_{totalDelay}$ (t^{D}_{delay} $+ \max(T^{D}_{cooldown})$ $+T_m$), s	**56** (1 + 50 + 5)	**55** (0 + 50 + 5)	–	–
CPU adjustment step below SLO ($-\Delta C^{D}$), %	5	3	–	–

[a]By default, the highest autoscaling action frequency of the HPA on Azure Kubernetes Service is one autoscaling action once per 60 seconds. The minimal stabilization window length is set to allow a faster reaction to changes.

[b]$D \in [up, down]$ denotes the autoscaling action directions, specifically upscale or downscale.

End of the Table 3.3

Autoscaler	SLA Adaptive Autoscaler		DMAR	HPA
Parameter	Values			
Scaling direction	Upscale	Downscale	–	
CPU adjustment step above SLO (+ΔC^{D}), %	3	2	–	–
CPU Thresholds ranges [min,max] ($[C_{min}^{CT^{D}}, C_{max}^{CT^{D}}]^{a}$), %	**Upper: [55,95], Mid: [45,75], Lower: [35,55]**	**Upper: [30,49], Mid: [20,40], Lower: [15,30]**	–	–
Downscale step when load is volatile (ΔR_{V})	–	0.1	–	–
SLO threshold without impact factor (SLO_{BP})	95%	–	–	–
No down scale action SLO threshold ($SLO_{noDownScale}$)	–	1.0025	–	–

[a] CT^{D} represents the identifier of the selected CPU threshold, where $CT \in [upper, mid, lower]$.

During the QoS recovery phase ($n \in [0; 199]$), the effectiveness of the solutions was assessed at times when there was a high decrease in the SLO value. This phase represents a system failure scenario and evaluates how quickly the solutions can recover from such high-impact disruptions. The findings from this phase offer insights into how quickly the solutions can restore the desired level of service.

During the QoS support phase ($n \in [200; N]$) the performance of the solutions was evaluated during periods, where there was no significant decline in QoS (less than 5% per minute). This phase represents typical operational scenarios without any unusual incidents. The duration of each experiment in this phase was limited to the length of the shortest experiment within the dataset, consisting of N intervals of six seconds each. Assessing the solutions during this phase provides insights into their ability to maintain the desired service level under normal operating conditions.

The autoscaling solutions were primarily evaluated in each experiment based on their ability to meet or exceed the desired SLO throughout the evaluation period. This capability was assessed by determining the percentage of time the system adhered to SLO target (t_{SLO}^{tgt}). This parameter evaluates the performance of the solutions during the QoS support phase. It is expected that the SLO value should

Table 3.4: HPA, DMAR, and SAA performance evaluation during QoS support and recovery time under synthetic workload scenarios

Load type	Slow			Moderate			Fast			Peaks and pauses			Mixed		
Parameter	HPA	DMAR	SAA	HPA	DMAR	SAA	HPA	DMAR	SAA	HPA	DMAR	SAA	HPA	DMAR	SAA
Evaluation during QoS support time (i=200,j=1220)															
Time t_{SLO}^{tgt}, %	29.7	97.7	100.0	0.0	0.0	82.3	0.0	100.0	97.0	0.0	0.0	49.7	0.0	28.4	56.4
$\frac{\sum_{n=i}^{j} R_n}{\sum_{n=i}^{j} R_{\eta_n}^{tgt}}$	1.04	1.09	1.1	0.93	1.33	1.59	0.77	2.31	1.6	0.68	3.15	3.44	0.92	2.83	2.47
$\sum_{n=i}^{j} R_n$	4664	4906	4900	3568	6061	7559	3119	10321	7135	1778	10126	11594	2476	9099	8469
$\sum_{n=i}^{j} R_{\eta_n}^{tgt}$	4476	4495	4468	3822	4563	4762	4037	4476	4472	2620	3213	3374	2702	3216	3435
$\sum_{n=i}^{j} R_n^+ + \lvert R_n^- \rvert$	410	533	724	1076	1622	2915	1288	5887	2835	1554	7023	8254	926	5977	5130
$\sum_{n=i}^{j} R_n^-$	–111	–61	–146	–665	–62	–59	–1103	–21	-86	–1198	–55	–17	–576	–47	–48
$\sum_{n=i}^{j} R_n^+$	299	472	578	411	1560	2856	185	5866	2749	356	6968	8237	350	5930	5082
Evaluation during QoS recovery time (i=0,j=199)															
$\frac{\sum_{n=i}^{j} R_n}{\sum_{n=i}^{j} R_{\eta_n}^{tgt}}$	0.94	0.99	1.01	0.76	1.25	1.45	0.64	3.05	2.65	0.46	3.18	2.3	0.71	2.04	2.43
$\sum_{n=i}^{j} R_n$	777	841	860	557	1050	1286	541	2748	2447	305	2065	1567	523	1718	2091
$\sum_{n=i}^{j} R_{\eta_n}^{tgt}$	826	846	855	729	841	887	850	902	923	657	650	682	734	841	862
$\sum_{n=i}^{j} R_n^+ + \lvert R_n^- \rvert$	73	69	83	290	305	427	337	1884	1572	460	1461	927	251	989	1263
$\sum_{n=i}^{j} R_n^-$	–61	–37	–39	–231	–48	–14	-323	–19	–24	–406	–23	–21	–231	–56	–17
$\sum_{n=i}^{j} R_n^+$	12	32	44	59	257	413	14	1865	1548	54	1438	906	20	933	1246

Table 3.5: HPA, DMAR, and SAA performance evaluation results during the whole experiment under synthetic workload scenarios

Load type	Slow			Moderate			Fast			Peaks and pauses			Mixed		
Parameter	HPA	DMAR	SAA	HPA	DMAR	SAA	HPA	DMAR	SAA	HPA	DMAR	SAA	HPA	DMAR	SAA
Evaluation during whole time of the experiment (i=0,j=1220)															
$\frac{\sum_{n=i}^{j} R_n}{\sum_{n=i}^{j} R_{\eta_n}^{tgt}}$	1.03	1.08	1.08	0.91	1.32	1.57	0.75	2.43	1.78	0.64	3.16	3.25	0.87	2.67	2.46
$\sum_{n=i}^{j} R_n$	5448	5755	5767	4128	7120	8852	3663	13089	9590	2085	12194	13170	3004	10822	10570
$\sum_{n=i}^{j} R_{\eta_n}^{tgt}$	5309	5348	5330	4555	5410	5656	4890	5383	5401	3282	3863	4057	3440	4060	4301
$\sum_{n=i}^{j} R_n^{+} + \lvert R_n^{-} \rvert$	483	603	807	1367	1930	3342	1625	7786	4409	2017	8487	9189	1178	6968	6399
$\sum_{n=i}^{j} R_n^{-}$	–172	-98	–185	–897	–110	–73	–1426	–40	–110	–1607	–78	–38	–807	–103	–65
$\sum_{n=i}^{j} R_n^{+}$	311	505	622	470	1820	3269	199	7746	4299	410	8409	9151	371	6865	6334

Table 3.6. HPA, DMAR, and SAA performance evaluation in real-world workload scenarios

Load type	WorldCup'98			EDGAR		
Parameter	HPA	DMAR	SAA	HPA	DMAR	SAA
Evaluation during QoS support time (i=200,j=1500)						
Time t_{SLO}^{tgt} , %	68.2	83.4	100.0	0.0	0.0	93.5
$\frac{\sum_{n=i}^{j} R_n}{\sum_{n=i}^{j} R_{\eta_n}^{tgt}}$	1.25	1.17	1.58	0.84	0.94	2.15
$\sum_{n=i}^{j} R_n$	15859	14691	19538	1512	12378	22023
$\sum_{n=i}^{j} R_{\eta_n}^{tgt}$	12698	12560	12339	1799	13222	10227
$\sum_{n=i}^{j} R_n^+ + \lvert R_n^- \rvert$	3201	2163	7199	957	1350	11796
$\sum_{n=i}^{j} R_n^-$	–20	–16	0	–622	–1097	0
$\sum_{n=i}^{j} R_n^+$	3181	2147	7199	335	253	11796
Evaluation during QoS recovery time (i=0,j=199)						
$\frac{\sum_{n=i}^{j} R_n}{\sum_{n=i}^{j} R_{\eta_n}^{tgt}}$	0.94	1.04	1.46	1.04	1.01	1.76
$\sum_{n=i}^{j} R_n$	694	715	1026	334	1587	2418
$\sum_{n=i}^{j} R_{\eta_n}^{tgt}$	741	689	703	320	1568	1370
$\sum_{n=i}^{j} R_n^+ + \lvert R_n^- \rvert$	59	30	333	116	215	1064
$\sum_{n=i}^{j} R_n^-$	–53	–2	–5	–51	–98	–8
$\sum_{n=i}^{j} R_n^+$	6	28	328	65	117	1056
Evaluation during whole time of the experiment (i=0,j=1500)						
$\frac{\sum_{n=i}^{j} R_n}{\sum_{n=i}^{j} R_{\eta_n}^{tgt}}$	1.23	1.16	1.58	0.87	0.94	2.11
$\sum_{n=i}^{j} R_n$	16560	15413	20574	1847	13977	24460
$\sum_{n=i}^{j} R_{\eta_n}^{tgt}$	13446	13256	13049	2120	14802	11606
$\sum_{n=i}^{j} R_n^+ + \lvert R_n^- \rvert$	3260	2193	7535	1073	1565	12870
$\sum_{n=i}^{j} R_n^-$	–73	–18	–5	–673	–1195	–8
$\sum_{n=i}^{j} R_n^+$	3187	2175	7530	400	370	12862

generally remain below the SLO target during the recovery phase. The evaluation results are presented in Table 3.5.

Secondary assessment criteria included the total number of provisioned containers ($\sum_{n=i}^{j} R_n^+$), the total number of overprovisioned ($\sum_{n=i}^{j} R_n^+$) and underprovisioned ($\sum_{n=i}^{j} R_n^+$) containers compared to the "touchstone" autoscaler's provisioned containers ($\sum_{n=i}^{j} R_{\eta_n}^{tgt}$), and the ratio of provisioned containers to the "touchstone" autoscaler's provisioned containers ($\sum_{n=i}^{j} R_n / \sum_{n=i}^{j} R_{\eta_n}^{tgt}$).

The outcomes of experiments performed with a synthetic workload are described and illustrated in the remainder of this sub-chapter. The figures are presented in pairs to demonstrate the correlation between CPU utilization and QoS, as CPU utilization has the most significant impact on SLO (Agarwal, 2020). In the figures representing QoS, a dashed line indicates the target SLO value. Figures displaying average CPU utilization present the threshold value (dashed line) at and above which resource congestion occurs, as well as changes in the upper CPU threshold for upscale action value defined in SAA throughout the experiment. Following the CPU and SLO figures, there are figures showing the number of replicas and the number of processed transactions. These figures are presented together to illustrate the relationship between replica provisioning and the number of requests. The "slowly changing load" pattern experiment is presented first.

Slowly Changing Load

The load generated during "the slowly changing load" pattern experiment is depicted in Figure 3.8d. This pattern follows a specific sequence: it begins with zero sessions and increases by 20 sessions every 210 seconds, reaching a peak load of 150 rps. This peak load is maintained for 210 seconds. After the peak phase, the number of sessions decreases at the same rate as it increases. Following the decrease, there is a period of no load for 180 seconds. The entire cycle lasts 2260 seconds and repeats until the end of the experiment.

This load pattern is particularly effective for assessing the capabilities of HPA. The load either increases or decreases linearly, with a rate of approximately L_{max} every 230 seconds. It is important to note that the default configuration of HPA on AKS allows for one autoscaling action to be performed every 60 seconds. This interval provides sufficient time for HPA to respond to the increasing load and make the necessary scaling decisions.

According to the data presented in Table 3.4 ("Slow" column) and Figure 3.8a, all the evaluated solutions performed at or above the defined SLO level. In the case of the SAA-operated system, it consistently delivered services at a level higher than the SLO. This resulted in the upscaling CPU threshold gradually increasing from its initial value of 75% of average CPU utilization to 78% after approximately ten minutes of operation, as shown in Figure 3.8b.

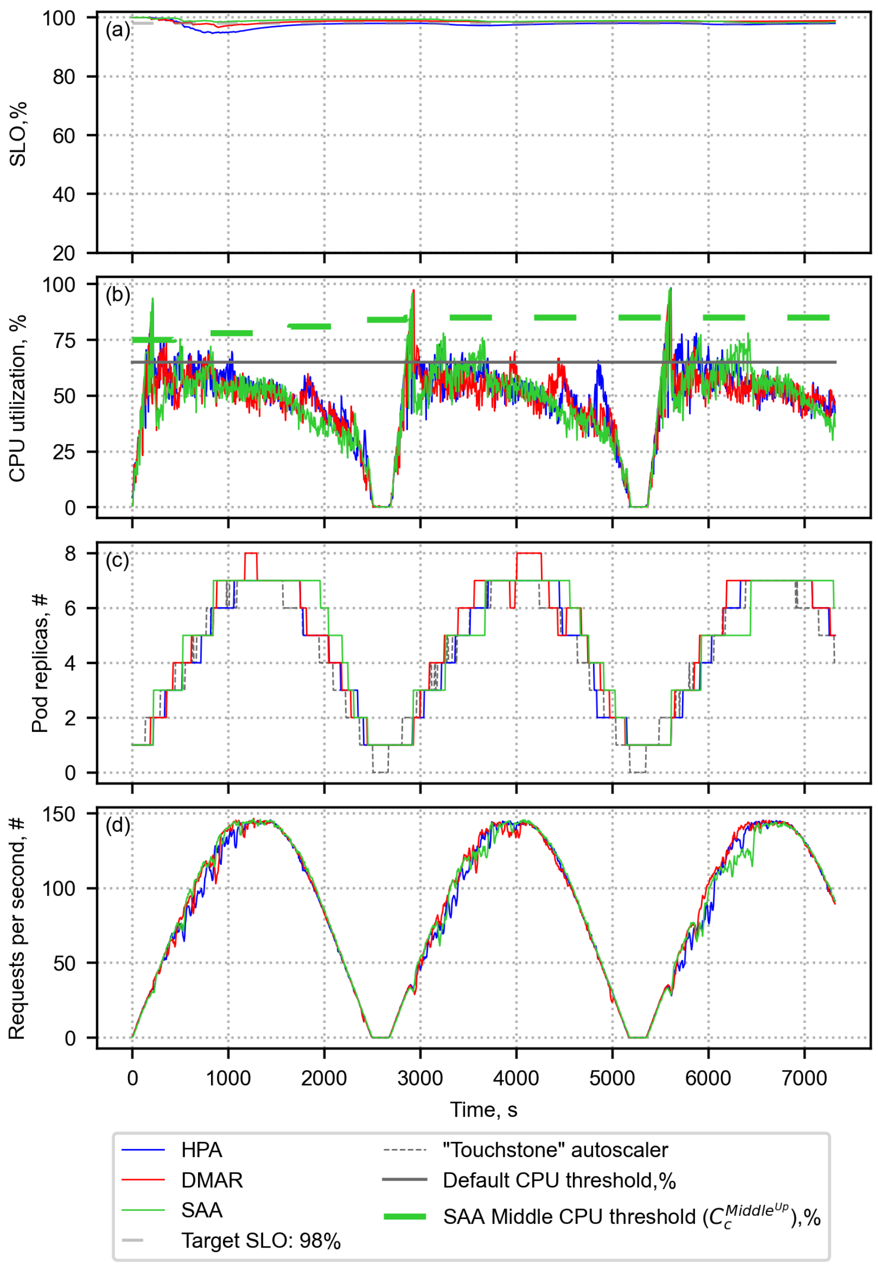


Fig. 3.8. Evaluation of SAA, DMAR, and HPA solutions in the slowly changing load scenario. (a) Achieved SLO value. (b) Average CPU utilization. (c) Number of pods provisioned at each period. (d) Processed workload requests per second

Over the next 40 minutes, the threshold value gradually rose to a maximum of 85%. This resulted in the highest and longest period of increased CPU utilization, which occurred approximately between 6200 and 6500 seconds (as shown in Fig. 3.8b). Consequently, the SLO value dropped to its lowest point during the entire evaluation period. This increase in the CPU threshold also resulted in a longer delay in provisioning pods, as indicated in Figure 3.8c. Consequently, some of the requests were not processed within the desired timeframe, as depicted in Figure 3.8d. Evidently, SAA is capable of adjusting its CPU thresholds to align with the existing resource demand.

Figure 3.8c illustrates that no significant difference exists in the number of provisioned replicas among the evaluated solutions. The number of replicas closely aligns with the theoretically provisioned number of replicas by the “touchstone autoscaler”. Both the SAA and HPA solutions show a similar number of under-provisioned replicas. However, during the QoS recovery phase, both the DMAR and HPA solutions faced under-provisioning, which resulted in a decline in the SLO value, dropping it below the desired level. Despite these variations, all the evaluated solutions processed a comparable number of requests.

In summary, the results indicate that all evaluated solutions are capable of delivering services in compliance with the SLO with only minor deviations. Furthermore, these solutions demonstrate efficient resource utilization with minimal overprovisioning, typically below 10%. It is worth noting that the solutions could potentially achieve even better results if the more extended autoscaling periods were set. In summary, the results show that the assessed solutions successfully meet SLO requirements and efficiently utilize resources when the load changes slowly.

Moderately Changing Load

The load pattern in the “moderately changing load” experiment, as depicted in Figure 3.9d, follows a specific pattern. The load gradually increases the number of sessions by 20 every 50 seconds, starting from zero and reaching a peak of 150 rps for a duration of ten seconds. After reaching this peak, the number of sessions will decrease at the same rate as it increased. This cycle repeats throughout the experiment, with each cycle lasting a total of 510 seconds.

Figure 3.9a demonstrates that the SAA solution performs close to requirements, achieving the desired SLO with minor fluctuations slightly below the target. The SAA solution maintained compliance with the SLO for 82.4% of the experiment duration. It may be beneficial to adjust the $SLO_{noDownScale}$ threshold to a slightly higher value, such as 1.005. This adjustment would allow further mitigation of the risk of SLO violations without changing the initial dynamic CPU threshold values. Figure 3.9b illustrates that both the HPA and DMAR solutions could not meet the target SLO using the 60% CPU utilization threshold.

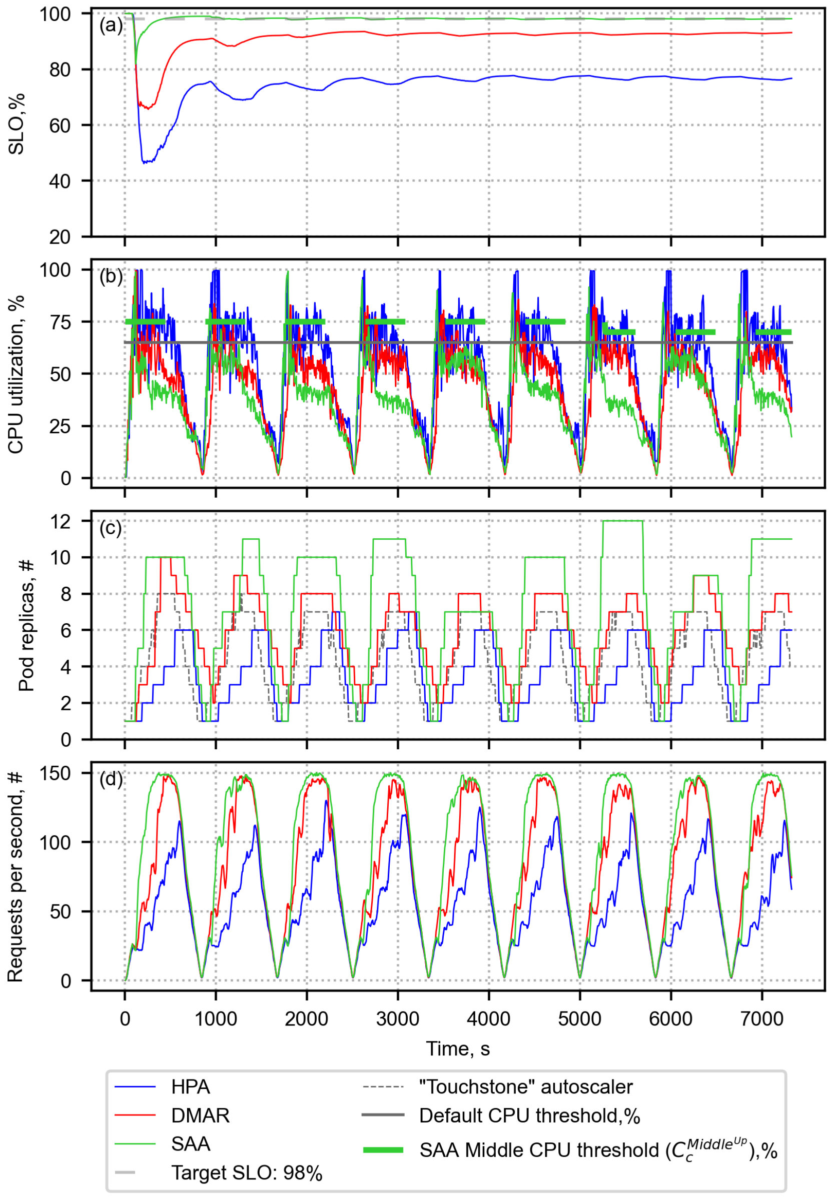


Fig. 3.9. Evaluation of SAA, DMAR, and HPA solutions in the moderately changing load scenario. (a) Achieved SLO value. (b) Average CPU utilization. (c) Number of pods provisioned at each period. (d) Processed workload requests per second

The benchmark results and the comparison with the “touchstone” autoscaler experiment indicate that the 60% CPU threshold could be considered adequate for delivering services within the defined SLO.

In the case of SAA, it operated with the upper upscale CPU threshold set to 75% for approximately 75% of the experiment time. After that, it dynamically adjusted the CPU threshold value to the minimum allowed value of 70% to achieve the desired SLO. These results suggest that SAA is able to adapt its replica provisioning logic and accomplish the SLO maintenance goal, even when the dynamic CPU threshold limits are initially set too high.

Figure 3.9c and the “Moderate” column of Table 3.5 demonstrate that SAA tends to overprovision resources in advance due to the velocity impact factor. This proactive provisioning strategy proves beneficial as it minimizes the risk of provisioning delays and, in turn, yields better results in terms of QoS support and restoration, as demonstrated by Figure 3.9a. Additionally, as depicted in Figure 3.9d, this strategy helps mitigate the risk of request processing delays caused by resource shortages. It also reduces the likelihood of unprocessed requests (Toka et al., 2021).

Even though DMAR consistently provisioned a maximum number of containers close to R_η, CPU utilization was kept below the resource congestion threshold most of the time. However, the delayed autoscaling reaction resulted in a drop in the QoS level and the number of processed transactions.

On the other hand, HPA demonstrated the slowest adaptation to fluctuations in load, frequently resulting in under-provisioning and a failure to manage the peak load of 150 rps. This delay was linked to the technical constraints of Kubernetes operating on Azure, where the fixed autoscaling action interval of 60 seconds is not adjustable (Microsoft, 2024b).

Fast-Changing Load

The load pattern shown in Figure 3.10d represents a “fast-changing load” scenario. It involves a rapid increase in the number of sessions from 0 to 100 at a rate of 20 sessions every ten seconds, followed by a peak load of 150 rps for ten seconds. Subsequently, the number of sessions decreases at the same rate as it increases. This cycle repeats until the end of the experiment, with each cycle lasting a total of 110 seconds.

Figure 3.10a illustrates that both DMAR and SAA successfully provide service at the target SLO level. In contrast, the system managed by HPA exhibited average CPU utilization consistently above the 60% threshold (Fig. 3.10b), resulting in the lowest QoS performance among the evaluated solutions. The results presented in Figure 3.10b suggest that the initially set upper threshold for CPU upscaling caused issues for SAA in achieving the target SLO. As a result, SAA adjusted the threshold to the minimum defined level of 70% during the first round of CPU adjustments.

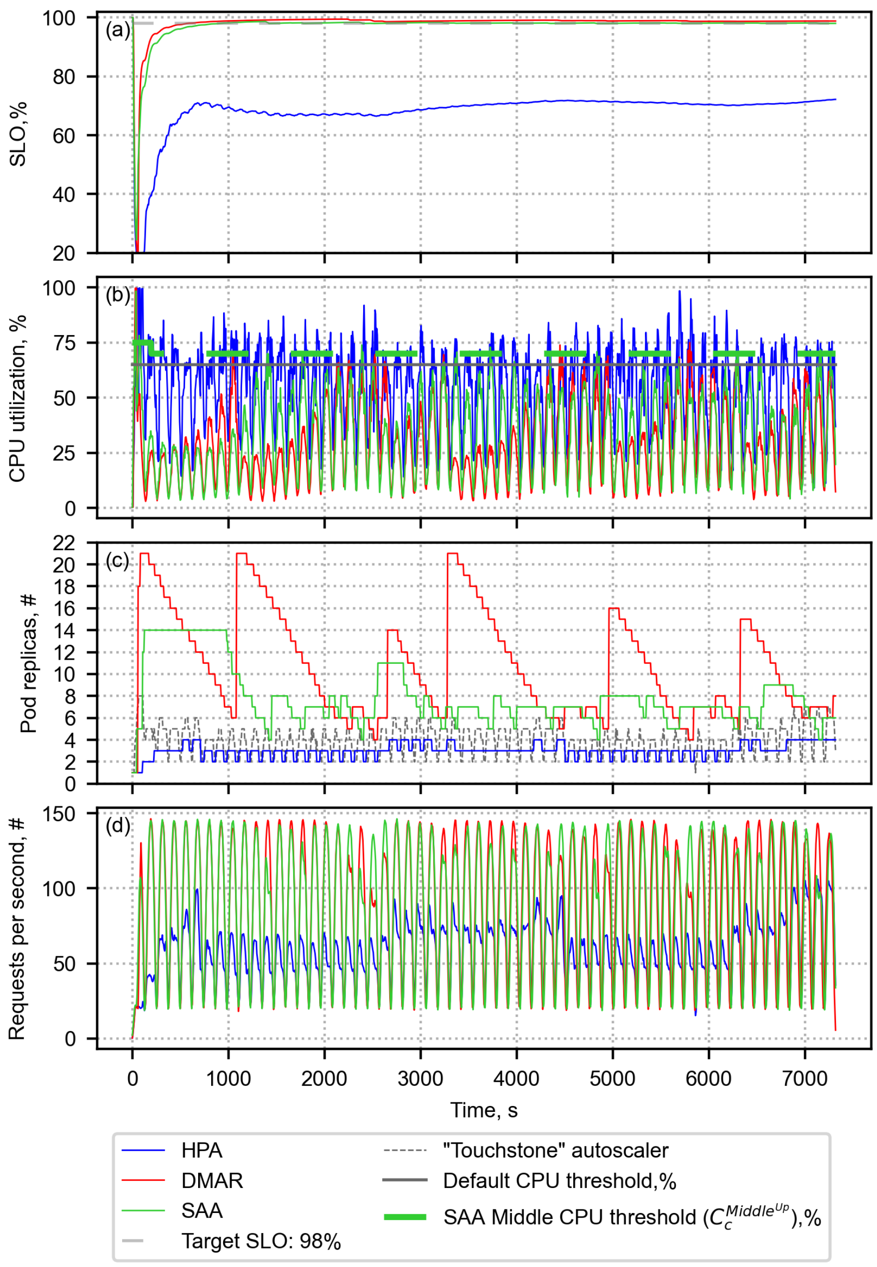


Fig. 3.10. Evaluation of SAA, DMAR, and HPA solutions in the moderately changing load scenario. (a) Achieved SLO value. (b) Average CPU utilization. (c) Number of pods provisioned at each period. (d) Processed workload requests per second

This flexible approach allowed SAA to improve its performance in fulfilling the SLO criteria.

DMAR shows improved performance regarding SLO support but requires more resources (Fig. 3.10c). Table 3.5 suggests that DMAR overprovisions by more than twice the number of replicas compared to the "touchstone" autoscaler. Additionally, it requires three times the number of replicas to recover from a decrease in QoS. This overprovisioning can be linked to the calculation performed by DMAR, which uses the ratio between application throughput per container in the current interval and the application throughput in two previous monitoring periods to calculate the number of the required replicas. In scenarios with rapidly changing loads, the significant variations in throughput between monitoring samples can cause spikes in replica provisioning.

Figure 3.10d shows that DMAR and SAA processed a similar number of transactions, whereas HPA managed to handle only about one-third of that load. Although both SAA and DMAR met their SLO, they incurred higher resource overprovisioning compared to the baseline. However, SAA demonstrated better resource provisioning during the QoS support periods, utilizing 1.5 times less resources. This can be attributed to the mechanism limiting overprovisioning of replicas in SAA. SAA limits a maximal number of replicas by α_{high} and does not exceed $\alpha_{high} * R_n^D$. On the other hand, DMAR is only limited by the maximum replica number setting, as indicated in Table 3.2. The data in Table 3.4 clearly demonstrates the trend of HPA towards under-provisioning resources, leading to less reliable support for QoS compared to SAA and DMAR.

Load with Peaks and Pauses

In this load pattern, the number of sessions gradually increases from 0 to peak values ranging from 20 to 210 rps. Most peaks reach approximately 150 rps. Variable-length pauses, lasting between tens and 70 seconds, are introduced between these peaks. The total duration of one cycle in this load pattern is 679 seconds, and this pattern is repeated throughout the entire experiment, as shown in Figure 3.11d.

The results suggest that the load pattern characterized by variable-length pauses and peak values of 150 rps presented a great challenge in achieving the defined SLO. Both SAA and DMAR utilized a similar number of replicas, and only the SAA solution supported the SLO for about half of the experiment duration. However, it should be noted that DMAR exhibited a tendency to overprovision replicas when faced with fast load changes and frequent metrics evaluation (periods of less than 30 seconds), as indicated by Figure 3.11c, and Tables 3.4 and 3.5.

The results reveal an interesting observation: despite DMAR operating below the SLO level, it processed nearly the same number of requests as SAA (Fig. 3.11a and (d)). Figure 3.11b shows that the average CPU load for both DMAR and SAA was either at or slightly above the resource congestion threshold. As a result, the

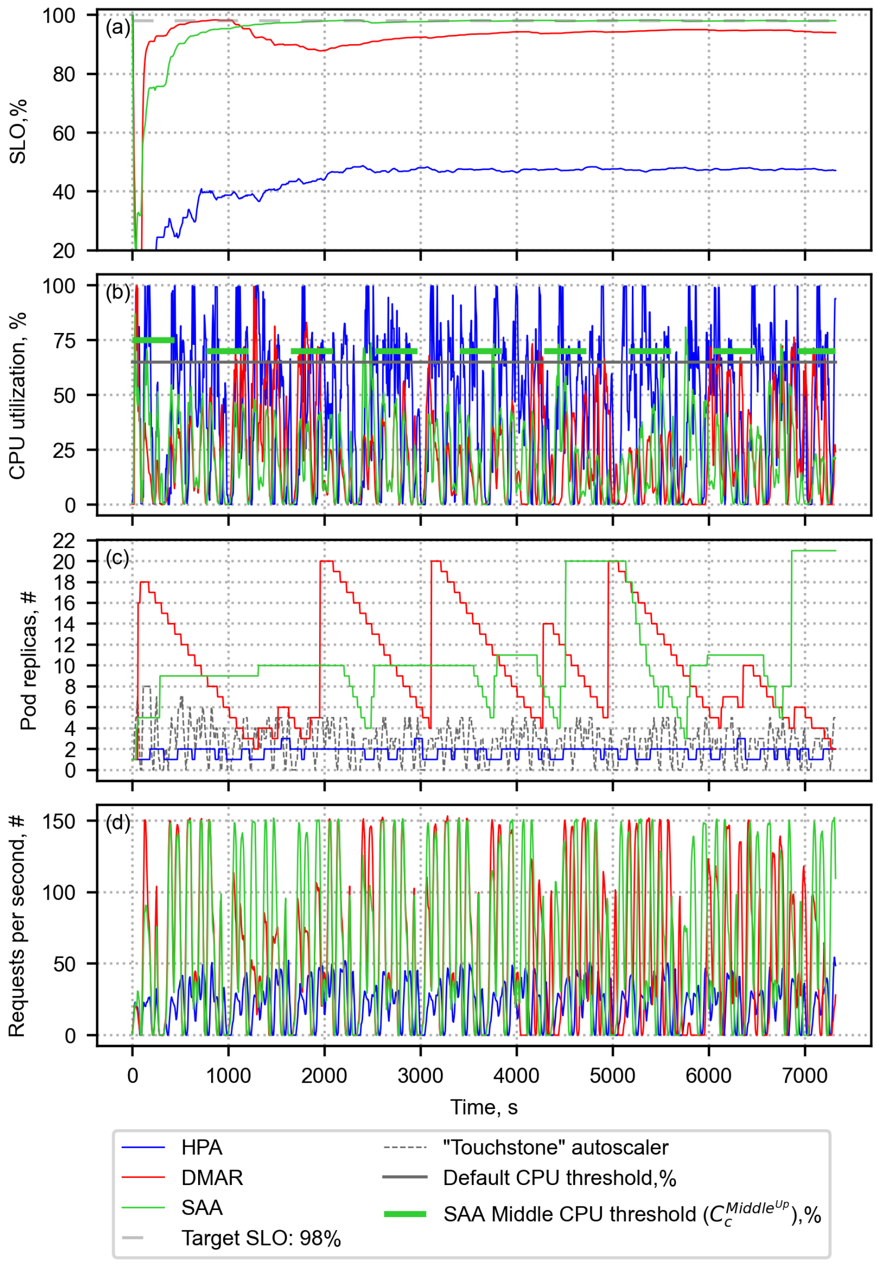


Fig. 3.11. Evaluation of SAA, DMAR, and HPA solutions in the load with peaks and pauses scenario. (a) Achieved SLO value. (b) Average CPU utilization. (c) Number of pods provisioned at each period. (d) Processed workload requests per second

number of processed requests for both systems was similar, with the exception of a brief period of roughly 1200 seconds. During this time, DMAR experienced the most significant drop in QoS and the number of requests processed. In contrast, SAA lowered its upper CPU threshold to address the situation.

As illustrated in Figure 3.11b, the HPA-operated system reached 100% CPU utilization multiple times and, as a result, exhibited the worst QoS results among the evaluated solutions. HPA continued to demonstrate a tendency for a low level of request processing and resource under-provisioning. As preseneted in Figure 3.11c, HPA served only one-third of the requests compared to SAA and DMAR. This highlights the limitations of HPA in delivering the desired service level and efficiently utilizing resources.

Mixed Pattern Load

The mixed load was created by randomly combining all the load patterns described in the previous sub-chapters, including the very slowly changing load. This combination aimed to produce a less predictable load pattern presented in Figure 3.12d.

As seen in Figure 3.12a and Tables 3.4 and 3.5 (column "Mixed"), both DMAR and SAA operated at levels close to the SLO, with SAA demonstrating slightly better results. However, SAA experienced longer periods of no downscaling due to SLO fluctuations during volatile load conditions (Fig. 3.12c). On the other hand, HPA-provisioned replicas reached 100% CPU utilization due to their slow reaction time (Fig. 3.12b). Interestingly, both DMAR and SAA-provisioned replicas consistently operated below the CPU resource congestion point of 65% (Fig. 3.12b).

It is worth noting the "spiky" pattern in pods provisioning for DMAR, demonstrated in Figure 3.12c, indicates its sensitivity to sudden load changes, even at a lower request rate of approximately 45 requests per second. In contrast, HPA on AKS consistently under-provision pods and provided a lower level of QoS.

1998 World Cup Website Access Logs-Based Load

In this experiment, the number of requests per second varied between approximately 80 and 335. Synthetic workload experiments demonstrated that HPA struggled to manage sudden increases in load, often becoming stuck in a failure loop. To address this, the experiment began with a gradual increase in the number of requests per second, rising from 0 to 140 requests per second over 700 seconds. After that, a workload based on website access logs from the WorldCup'98 event was generated. This approach allowed HPA to respond effectively and provision one replica per minute during each autoscaling iteration. The total duration of the experiment was 9000 seconds. Figure 3.13d displays the resulting load pattern.

The results presented in Table 3.6 and Figure 3.13 show that although all solutions achieved close to the desired system performance, only SAA could consistently maintain the desired service level throughout the entire experiment, albeit

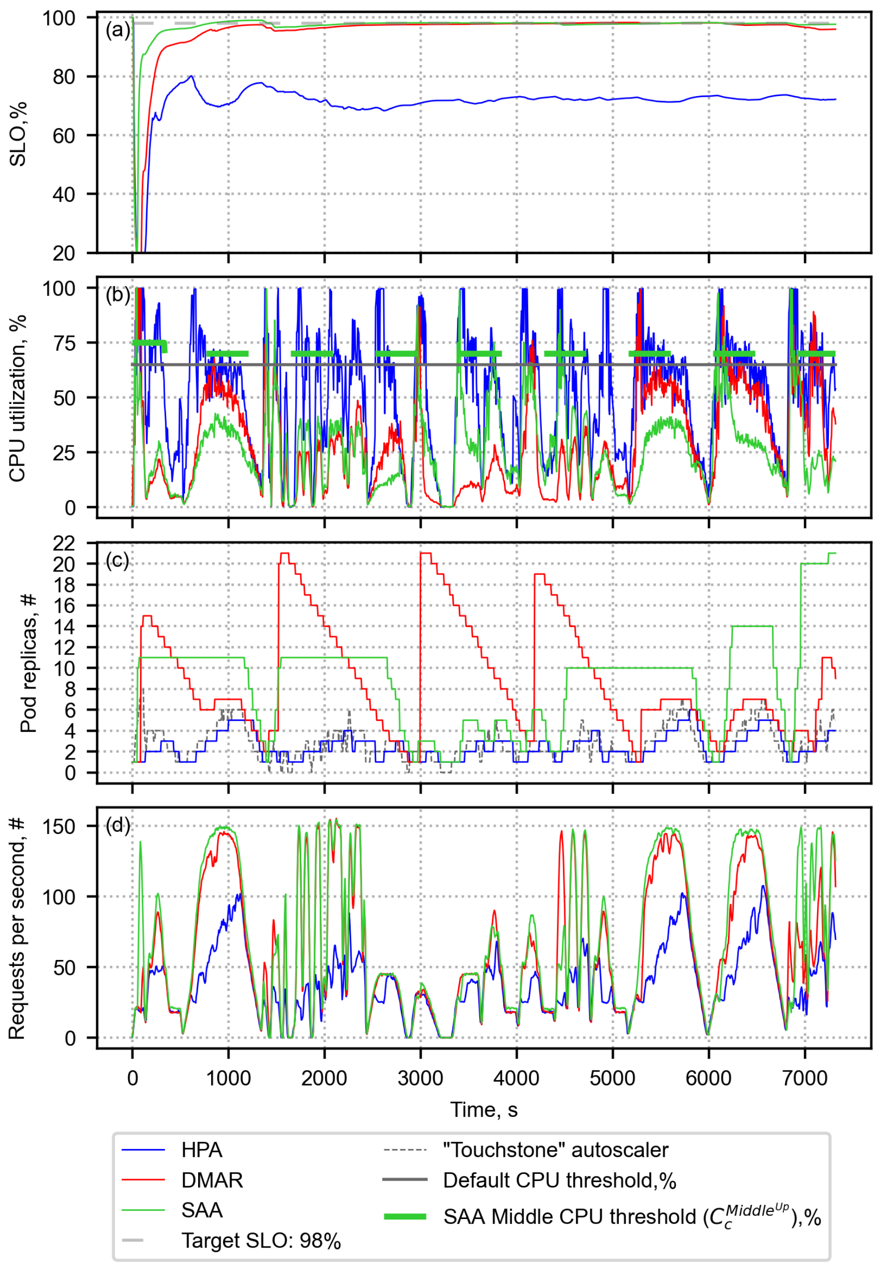


Fig. 3.12. Evaluation of SAA, DMAR, and HPA solutions in the load with peaks and pauses scenario. (a) Achieved SLO value. (b) Average CPU utilization. (c) Number of pods provisioned at each period. (d) Processed workload requests per second

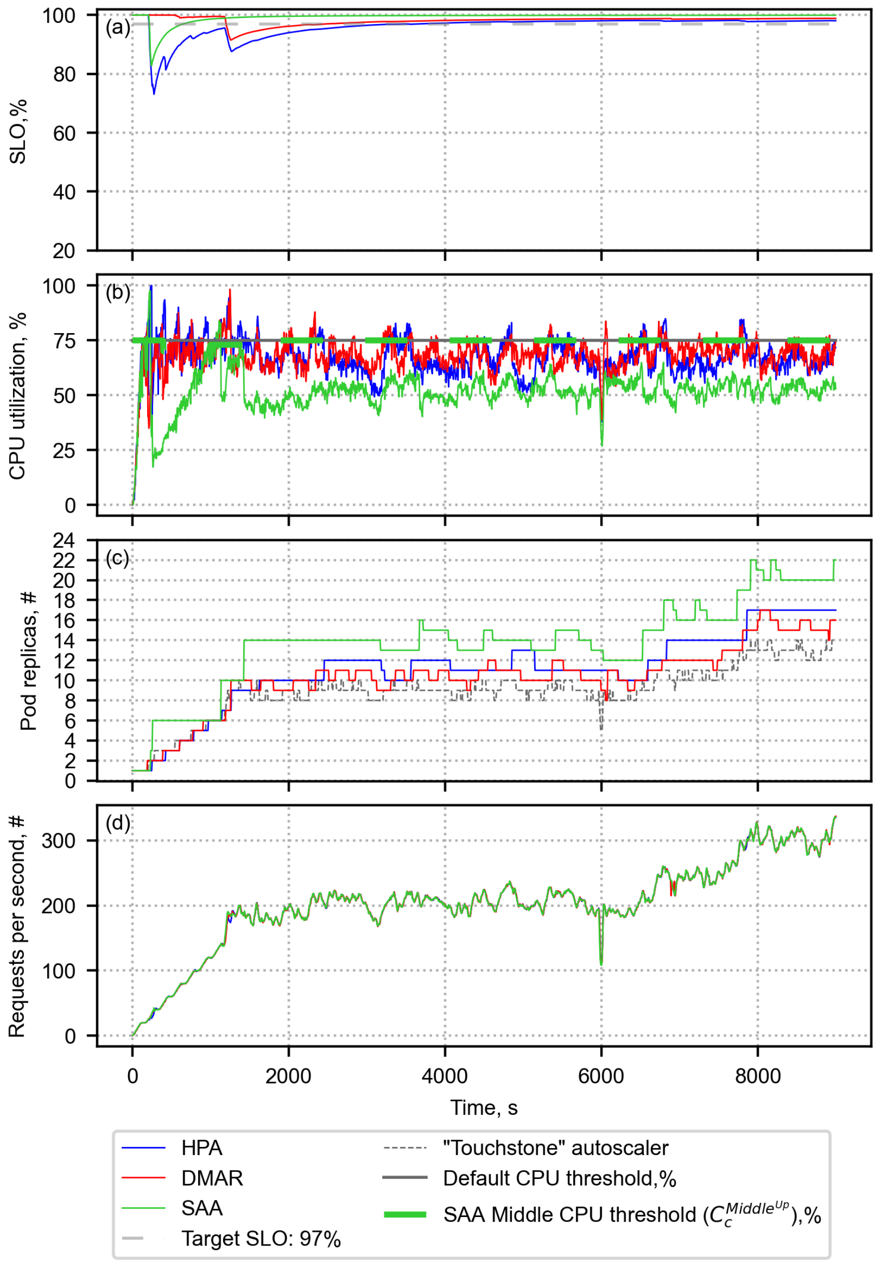


Fig. 3.13. Evaluation of SAA, DMAR, and HPA solutions in the WorldCup'98 web traces-based load scenario. (a) Achieved SLO value. (b) Average CPU utilization. (c) Number of pods provisioned at each period. (d) Processed workload requests per second.

with the highest overprovisioning cost. While DMAR demonstrated superior precision in resource provisioning during the QoS recovery period, it could not restore the SLO value to the desired level and maintain the desired quality of service for the entire experiment, as it could not recover SLO from the delayed autoscaling actions.

As shown in Figure 3.13b, the SAA consistently operated with a 75% CPU threshold throughout the entire experimental period, except for a brief period when the SLO value dropped. During that time, the SAA adjusted its threshold to a lower value to recover more quickly and meet the SLO. Both DMAR and HPA operated as close as possible to the desired threshold value and, as a result, showed higher resource efficiency compared to SAA. SAA tended to operate at a level lower than the threshold at which SLO started to degrade.

As can be seen from the results, all solutions showed similar efficiency results compared to moderate load scenarios; however, they had a lower tendency to underprovision. One reason for this could be that the Rust application used in this experiment was provisioned faster than the Java application. Based on these results, it can be assumed that the consistent behavior of algorithms is independent of the application or environment.

EDGAR Website Access Logs-Based Load

In this pattern, the number of requests per second ranged from approximately 140 to 265. At the start of the experiment, the number of requests per second increases gradually from 0 to 200 rps in 200 seconds, followed by the EDGAR website access logs-based workload. This required HPA to provision at least two pods per minute. The total length of this experiment was 9000 seconds. Figure 3.14d displays the resulting load pattern.

The results shown in Table 3.6 and Figure 3.14 indicate that only SAA could provide the desired performance. However, this was achieved for only 93% of the QoS support time and at the cost of twice the resources compared to the actual demand, as measured in the form of "touchstone" provisioned replicas. Both DMAR and HPA DMAR constantly underprovisioned the resources, which resulted in the desired QoS not being delivered. Nevertheless, DMAR demonstrated superior precision in resource provisioning during the QoS recovery period, which was insufficient to restore SLO.

As presented in Figure 3.13b, The SAA initially operated with a 75% CPU threshold, but after the SLO decreased, it adjusted its threshold to 60%. This new threshold allowed it to achieve the desired quality of service for most of the experiment. It is worth noting that the SAA tended to operate at a level nearly half of the threshold at which the SLO began to degrade, resulting in almost double the amount of resource overprovisioning. DMAR and HPA consistently exceeded the 75% CPU utilization threshold despite DMAR's use of a threshold adjustment

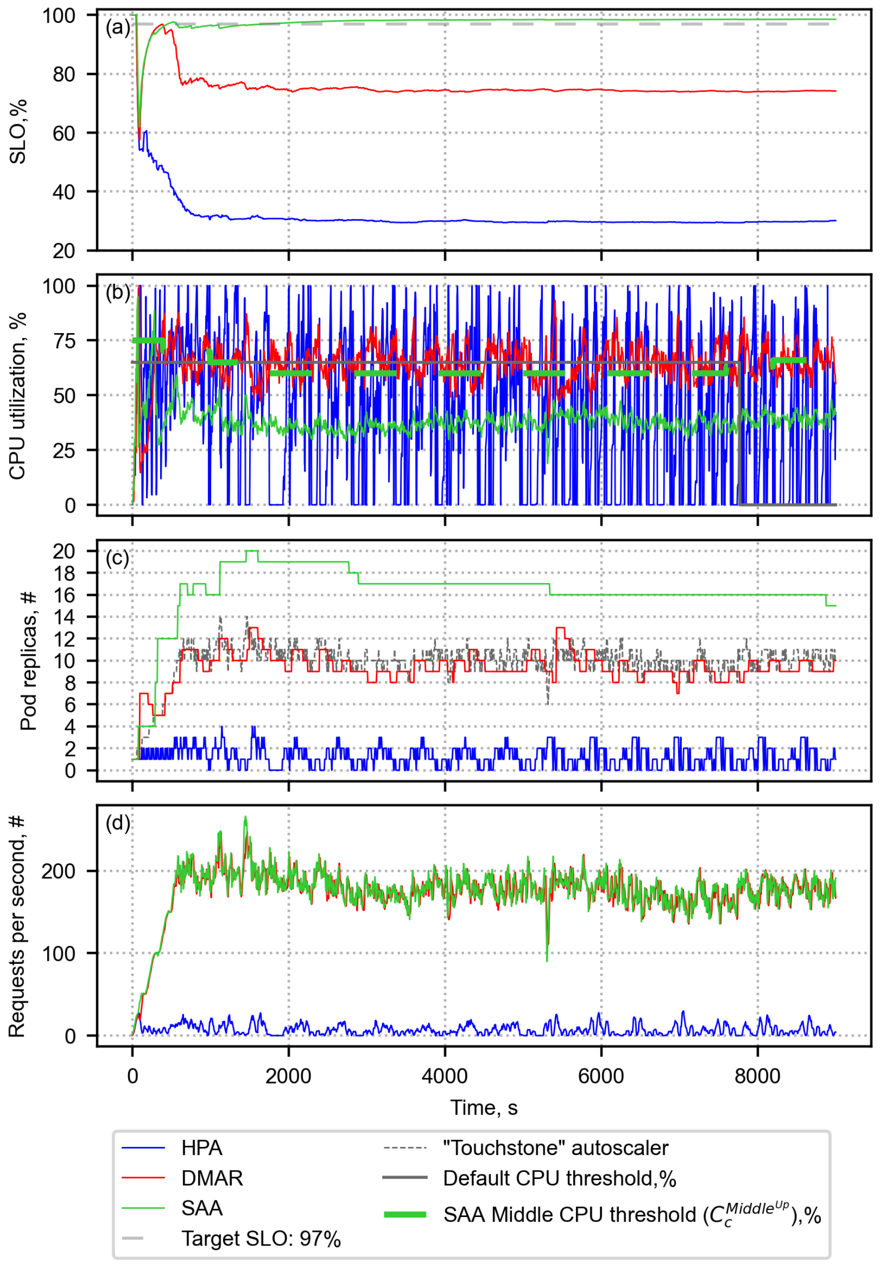


Fig. 3.14. Evaluation of SAA, DMAR, and HPA solutions in the EDGAR web traces-based load scenario. (a) Achieved SLO value. (b) Average CPU utilization. (c) Number of pods provisioned at each period. (d) Processed workload requests per second

mechanism. DMAR's reliance on average response time left it vulnerable to frequent load fluctuations. HPA, which made autoscaling decisions based solely on low-level metrics, became stuck in a provisioning loop of 1–4 replicas (Fig. 3.13c) due to the specific implementation of the HPA algorithm. This meant that the number of replicas provided in the next iteration depended entirely on the set CPU threshold value. The 75% threshold only allowed for a maximum increase of 25% in replicas (100 ÷ 75), which was often insufficient when there were a small number of pods and the load experienced high fluctuations.

Interestingly, DMAR showed high under-provisioning in the resources, even though the scaling indicator was set to a lower level than the service level indicator. This behavior was not demonstrated in any of the synthetic workloads. On the other hand, the SAA algorithm did not demonstrate under-provisioning, except during the recovery phase. This behavior was not demonstrated with any of the synthetic workloads either.

The results show that the SAA can adapt to different real-world workloads, helping to restore the system to fulfill the SLA requirements and maintain the QoS at the desired level. A more detailed conclusion and discussion of the experiments will be presented in the next sub-chapter.

The experiments indicate that the HPA struggles to consistently meet SLAs due to varying workload patterns, even with the same threshold value. Dynamic adjustments of thresholds, in response to changes in workload or the environment, are essential for improving SLA fulfillment in rules-based autoscaling solutions, such as HPA. Sub-chapter 2.2 introduced an algorithm designed to facilitate these adjustments, helping similar algorithms achieve their SLA targets. The next sub-chapter will present the results from the experimental evaluation of this algorithm.

3.2.5. Discussion

The sub-chapter aims to discuss and summarize the findings and results of experiments presented in this sub-chapter.

1. The comparative analysis was used to evaluate the Service Level Agreement-Adaptive (SAA) autoscaling solution, comparing it to two rules-based solutions: the Kubernetes Horizontal Pod Autoscaler (HPA) and Dynamic Multi-level Autoscaling Rules for Containerized Applications (DMAR). The SAA solution was additionally evaluated in terms of recovering the performance-based SLO. The key findings discovered during the SAA investigation are provided below:

1.1. The SAA demonstrated its ability to maintain system operations at QoS levels that are close to the desired targets, even when CPU threshold values are set at the point of resource saturation – where CPU utilization is high enough that response times start to exceed the target.

1.2. The inclusion of service-level objective (SLO) tracking supported the SAA algorithm in achieving its goals for restoring and maintaining SLOs. The application scaled by the SAA offers QoS support for longer periods than the HPA and DMAR solutions while utilizing a similar or twice greater amount of resources.

1.3. The solution has demonstrated its effectiveness in SLO restoration, particularly with load patterns characterized by a high density of requests per second, such as the moderately changing and fast-changing load patterns presented in this document. This can be attributed to the fact that the SAA can recover from a decrease in QoS more quickly when there is a greater capacity to process a higher volume of requests within shorter time intervals.

1.4. The results from both SAA and DMAR highlighted the effectiveness of utilizing both low-level and high-level metrics to ensure more predictable QoS, in contrast to HPA, which relies solely on low-level metrics for autoscaling decisions.

1.5. Implementing the $SLO_{noDownScale}$ threshold helped effectively postpone downscale actions, resulting in the successful restoration of QoS to its original level.

1.6. Dynamic adjustment of the cool-down period supported SAA in timely decision-making, leading to lower underprovisioning levels.

1.7. The inclusion of velocity impact factors in SAA contributes to effective resource provisioning during sudden load increases, providing resources in advance once a sudden load increase is detected. This dynamic nature of SAA enables it to operate closely to the defined SLO without requiring additional manual or ML-based adjustments in response to changes in traffic patterns. However, at the same time, this leads to increased overprovisioning of resources.

1.8. The results presented in this work are based on the SLA using the response time metric as SLI. However, the experiments could be extended to include experimentation with other performance-based SLA metrics. Moving forward, future work should focus on automating the adjustment of parameters such as cool-down, CPU threshold ranges, and baseline velocity. By automating these aspects, the solution's initialization process can be streamlined, reducing manual effort. Furthermore, there is a considerable opportunity for improvement in optimizing resource provisioning. Additionally, SAA requires a large number of parameters to be estimated empirically or theoretically, making it harder to adopt in real life. Automating the parameters, such as the length of the cool-down period and the threshold range values detection, could be a further area for research and improvement.

Cross-platform applicability could be another area of research to validate whether the proposed algorithms can be applied to platforms beyond Kubernetes or Azure Cloud. These future directions can enhance the scalability and applicability of the proposed SAA solution in real-world scenarios.

1.9. Experiments with HPA revealed the importance of proper threshold determination for HPA to be able to deliver a sufficient amount of resources as per demand and, as a result, ensure the desired quality of service.

3.3. Experimental Evaluation of the Proposed Service Level Agreement-Adaptive Dynamic Threshold-Adjustment Algorithm

This sub-chapter introduces the experiments conducted to assess the efficacy of the proposed dynamic threshold-adjustment algorithm discussed in sub-chapter 2.2. It also describes the experimental environment and metrics used to assess the efficacy of the proposed threshold-determination approach.

The efficacy and efficiency of the proposed algorithm were evaluated through two sets of experiments using the SLA-adaptive threshold adjuster (SATA) prototype developed explicitly for this purpose (Pozdniakova et al., 2024).

The first set of experiments was conducted to assess the influence of various settings on algorithm performance. The assessments include evaluations of the impact of threshold evaluation period length, the frequency of threshold updates, the implementation of different types of moving-average-smoothing techniques, and the usage of average response time instead of the N_{th} percentile for threshold determination on the efficacy and efficiency of the solution.

The second set of experiments assesses how the algorithm performs under changing real-world workload conditions. The aim was to evaluate the effectiveness of the suggested approach in adjusting the HPA target utilization threshold value dynamically. Through the dynamic adjustment of the threshold value, the system would operate as close as possible to the defined SLO. The experiments used the WorldCup'98 and EDGAR datasets and Java and Rust applications described in the previous sub-chapter.

All experiments were executed using AKS cluster, load-generating, and monitoring tools settings described in sub-chapter 3.2.2. The size of the AKS cluster varied across experiments depending on the workload and applications used; as a result, details about the AKS cluster size are provided in relevant sub-chapters. $SLO_{drop_{Tupdate}}$ parameter was set to 0.5%.

The autoscaling solution's performance evaluation was conducted while considering the methodological principles for reproducible performance evaluation in cloud computing proposed by Papadopoulos et al. (2021). The experiments and evaluation results are described in detail in the following sub-chapters.

3.3.1. Evaluation of the Impact of Different Settings on the Algorithm Performance

The selected settings can have a significant impact on the performance and effectiveness of the solution. This sub-chapter will focus on evaluating the influence of settings before assessing the solution's performance under different workload conditions.

All experiments described in this sub-chapter utilized the following settings:

- **Application:** calculation of the factorial of a number between 8000 and 12,000.
- **Pod replicas:** Min: 1, Max: 35.
- **Amount of Worker Nodes**: 5
- **Smoothing technique:** SMA.

The proposed algorithm can employ various smoothing techniques and SLIs, while also allowing for the adjustment of threshold evaluation period lengths to estimate thresholds. The evaluation of the impact of these settings is presented in the following text of this sub-chapter.

Impact of Threshold Adjustment and Evaluation Periods Length

Four experiments were conducted to evaluate the influence of threshold adjustment frequency (length of threshold adjustment period T_{adjust}) and the impact of the duration of the period used to collect metrics for the threshold value estimation T_{eval} on autoscaling efficiency and the ability to dynamically determine the threshold that allows operating as close as possible to the SLO-defined performance target (CTR_{slo}). The workload was generated using traces collected from the WorldCup'98 site on the 78th day from 19:37 to 21:37.

For clarity, in the tables and figures presented in remaining sub-chapters, each experiment will be referred to as $n \times m$, where n and m are multipliers of the scale periods used in $T_{adjust} = n \times T_{scale}$ and $T_{eval} = m \times T_{scale}$. For example, 4×10 means that the threshold is adjusted every 4^th scale period, and data collected from the last 10 scale periods provide information for the target threshold estimation. The evaluation periods, T_{eval}, of lengths 10 and 20, were used in the evaluation. This covers cases when the T_{adjust} length is close to T_{eval} (8×10 case), when

T_{adjust} is 2.5- or 5-times longer than T_{eval} (4×10, 8×20 and 4×20 cases, accordingly).

Table 3.7 shows the values of T_{adjust} and T_{eval} used in each experiment. These values are presented as multiples of the upscale period (T_{scale}). The value of T_{adjust} was set to 4, as it is the minimum number of periods required for the system to detect the impact of the latest threshold adjustment action. The setting of T_{adjust} to 8 should be sufficient to observe the effect of a longer update period without requiring a drastic extension of the experiment's length.

The results of these experiments are presented in Figure 3.15 and Table 3.7.

The data presented in Figure 3.15 and Table 3.7 suggests that the SATA solution is the most accurate in meeting the defined SLOs (the system performs closest to the desired SLO value) when T_{adjust} is equal to 4 (red and dark red lines). Longer T_{eval} periods tended to suggest a lower threshold. However, the system was able to make more granular suggestions as more events were collected (dark red and dark blue lines). Extending T_{eval} also increased the overprovisioning period, as the algorithm adapted to the lowest threshold that satisfied the target SLO over the period, leading to persistent overprovisioning. On the other hand, longer update periods proved advantageous when the load was fluctuating, as a lower threshold ensured that the autoscaling operations were more stable and did not repeat the fluctuation pattern. As a result, this minimized the risk of violations. Experiment 8×10 showed the worst performance and was the least adaptive to the detected workload changes (light blue line). However, SATA solution configured with this setting was still able to support the SLO.

Table 3.7. Impact of threshold adjustment periods on algorithm effectiveness evaluation results[1]

Settings	4×10	4×20	8×10	8×20
SLO supported	Fully	Fully	Fully	Fully
sMAPE [2], %	**1.6**	1.8	1.9	1.8
Total pods used (P_{total})	12992	**12783**	14502	13107
Difference from the best result for total pods in %	1.6	**0**	13.4	2.5

[1] Description of the evaluation criteria (sMAPE, Total Pods, and SLO supported) is provided in sub-chapter 2.2.4.

[2] Symmetric Mean Absolute Percentage Error (sMAPE).

Results from the experiments suggest that the threshold evaluation period (T_{eval}) should be a minimum of two to three times the threshold adjustment time (T_{adjust}). The algorithm performed well with M_{suff} set to 150, but when M_{suff} was increased to 300, the approach demonstrated improved efficiency and precision in threshold determination.

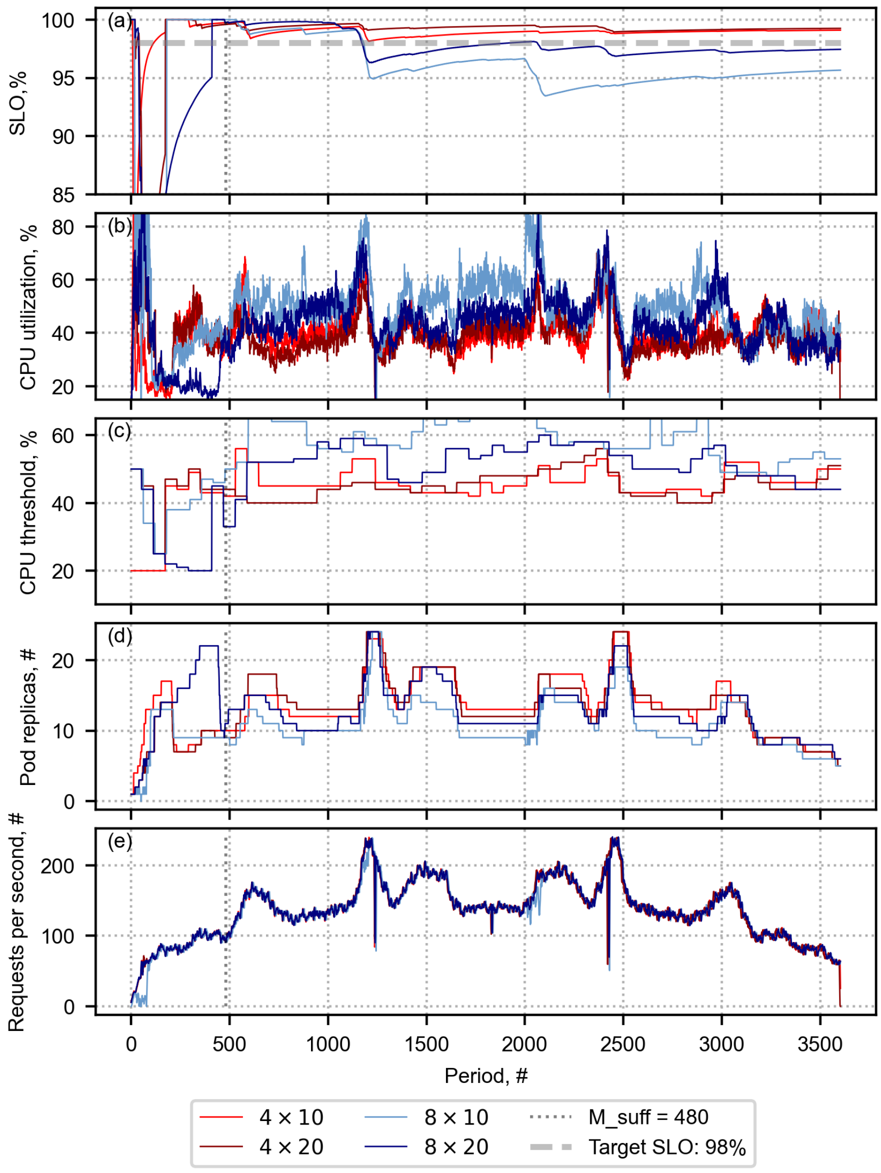


Fig. 3.15. Evaluation of the impact of threshold adjustment and evaluation period length on the effectiveness of the SLA-adaptive threshold adjustment (SATA) solution using Simple Moving Average (SMA). (a) SLO value after collecting a sufficient number of events (after the dotted line). (b) Average CPU utilization. (c) Applied target utilization threshold in each period. (d) Number of pods provisioned in each period. (e) Generated workload requests per second

The experiment described in the following sub-chapter assesses the impact of service level indicator selection on the algorithm efficacy and efficiency.

Impact of the Service Level Indicator

The prototype solution proposed in this work tracks the number of events where the n^{th} percentile of the response time values resulted in SLO violations during the specified monitoring period. This approach helps identify potentially upcoming violations and considers the violations that happened while the system operated at a particular CPU utilization level. However, it is not clear what impact the adoption of tail latency has on the efficiency and efficacy of the solution in comparison to the response time metric, which is more commonly observed in the literature.

The goal of the experiment described in this sub-chapter is to evaluate the impact of the SLI indicator used for SLO measurement on the algorithm's ability to adjust CPU thresholds to maintain the defined SLO as close as possible to the SLO target. The average response time and tail latency were used as SLI in this experiment. The acceptable threshold of 1500 ms was defined as SLO. The assessment was conducted using the SATA with a threshold adjustment period equal to four autoscaling periods, which showed better results compared to the previous experiment that used a threshold adjustment period equal to eight autoscaling periods. The workload was generated using traces collected from the WorldCup'98 site on the 78th day from 19:00 to 21:00 and its inverse version. The results of this assessment are presented in Table 3.8 and Figure 3.16.

Table 3.8. Evaluation of the impact of service level indicator (SLI) selection on the effectiveness of an algorithm that uses the Simple Moving Average (SMA) smoothing technique[1]

Settings	SMA 4×10		SMA 4×20	
SLI	98 percentile	Average response time	98 percentile	Average response time
SLO supported	**Fully**	Partialy	Fully	Partialy
sMAPE [2] , %	**1**	1.3	2.5	0.9
Total pods used (P_{total})	**43460**	34524	43809	39726
Difference from the best result for total pods in %	**0**	–20	1	–9

[1] Description of the evaluation criteria (sMAPE, Total Pods, and SLO supported) is provided in sub-chapter 2.2.4.

[2] Symmetric Mean Absolute Percentage Error (sMAPE).

Based on Figure 3.16 and Table 3.8, the algorithm equipped with average response time as SLI had the best efficiency and precision. However, it was not able to support the required level of service. Furthermore, regardless of the length of the threshold evaluation period, the provisioned service level tended to decline. In this case, the algorithm's efficiency was challenging to evaluate, as it failed to meet its main goal of SLO assurance throughout the experiment period in all evaluation cases. On the other hand, the experiment confirmed the use of tail latency as SLI as a more effective option for ensuring that SLO is maintained at the desired level.

In this sub-chapter, the experiments were conducted to gain a deeper insight into the behavior of the threshold adjustment algorithm in various scenarios. The findings suggest that the algorithm performs more effectively when tail latency is used as an SLI and that a shorter adjustment period yields more accurate results in achieving the desired performance. The next sub-chapter presents experiments designed to evaluate the performance of SATA under varying load conditions and compare its performance with the state-of-the-art technology, the HPA. The experiments also utilize different smoothing techniques, providing an overview of selected smoothing techniques in algorithm performance.

3.3.2. Evaluation of Performance under Different Workload Scenarios

In this sub-chapter, the experiments were executed to evaluate the algorithms' performance under varying load conditions and their ability to support the SLO for the whole period while the experiment was conducted. Two real-world workload scenarios were evaluated: WorldCup'98 and EDGAR.

The efficacy of the SATA approach was evaluated by comparing it to the HPA to understand the improvements brought to the HPA by SATA. Before assessing the impact of SATA on the HPA, it is important to configure the HPA with a threshold that enables the system to meet the SLO, as this is the primary objective of SATA. The target thresholds for the HPA were determined using the methodology outlined in sub-chapter 2.2.2 and refined through experiments to ensure that the system can closely achieve the desired SLO level.

For Java and Rust applications, the appropriate thresholds were identified using the EDGAR and WorldCup'98 workloads. For the Java application, the system met the performance requirements when the HPA CPU utilization threshold was set to 47% for the WorldCup'98 workload (Fig. 2.2) and 35% for EDGAR. As for the Rust application, the system met the performance requirements when the HPA threshold was set to 69% for the WorldCup'98 workload and 42% for the EDGAR workload. The data required to identify these thresholds was collected through the experiments, which employed the thresholds of 75% for the WorldCup'98 workload and 48% for EDGAR. Additional value adjustment and validation were performed through several value fine-tuning experiments.

The settings for each experiment described in this sub-chapter varied because different applications required different resources and infrastructure setups. De-

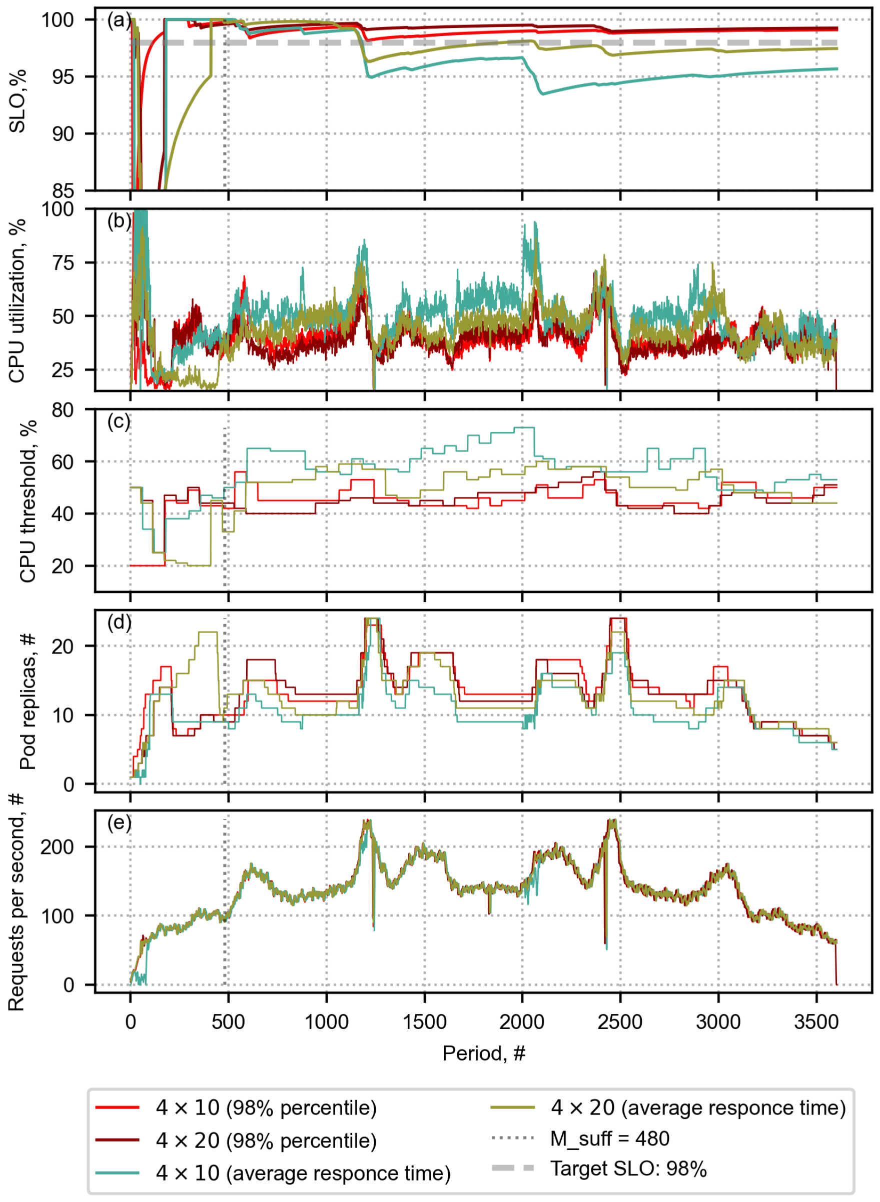


Fig. 3.16. Evaluation of the impact of the service level indicator on the effectiveness of the SLA-adaptive threshold adjustment solution using Simple Moving Average. (a) SLO value after collecting a sufficient number of events (after the dotted line). (b) Average CPU utilization. (c) Applied target utilization threshold in each period. (d) Number of pods provisioned in each period. (e) Generated workload requests per second

Table 3.9. Details of the experimental environment used to evaluate the performance of algorithms using various workloads and applications

Experiment reference	Workload	HPA threshold, %	Maximum number of pod replicas	Amount of Worker Nodes
Java application				
Java &World-Cup'98	WorldCup'98 (78th day from 19:00 to 22:00 and its inverse version)	47	34	5
Java &EDGAR	EDGAR (30 July 2023, from 02:00 to 05:00)	35	27	4
Rust Application				
Rust &World-Cup'98	WorldCup'98 (78th day from 19:37 to 21:37)	69	55	2
Rust &EDGAR	EDGAR (30 July 2023, from 02:00 to 05:00)	42	55	2

tails of the settings used for each combination of application and workload are provided in Table 3.9. For improved readability, each combination will be referred to as *"application name & workload name"* in this work.

It is important to note that due to the limitations of the load generation tool, the traffic log data sets were divided into three parts during the load generation activities. As a result, load generation sessions were executed sequentially with small time gaps of 15 to 20 seconds, during which no load was generated. This can be observed in the load patterns presented in the remaining figures of this work, where sudden dips or spikes appear in the CPU and requests per second performance metrics.

The next sub-chapters will provide a detailed evaluation of the algorithm's performance using various workloads and applications and the results achieved.

Java&WorldCup'98 Experiment Description and Results

In this experiment, the algorithm was evaluated for its ability to adjust the threshold value in the World Cup 98 scenario, which represents a mixed pattern load. As the WorldCup'98 load in the selected time window tended to increase constantly, its inverse version was appended to see how algorithms perform when the load tends to decrease continually. The newly generated workload pattern is presented in Figure 3.17e.

The experiment results presented in Figure 3.17 and Table 3.10 show that all of the evaluated approaches were compliant with the SLO throughout the experiment.

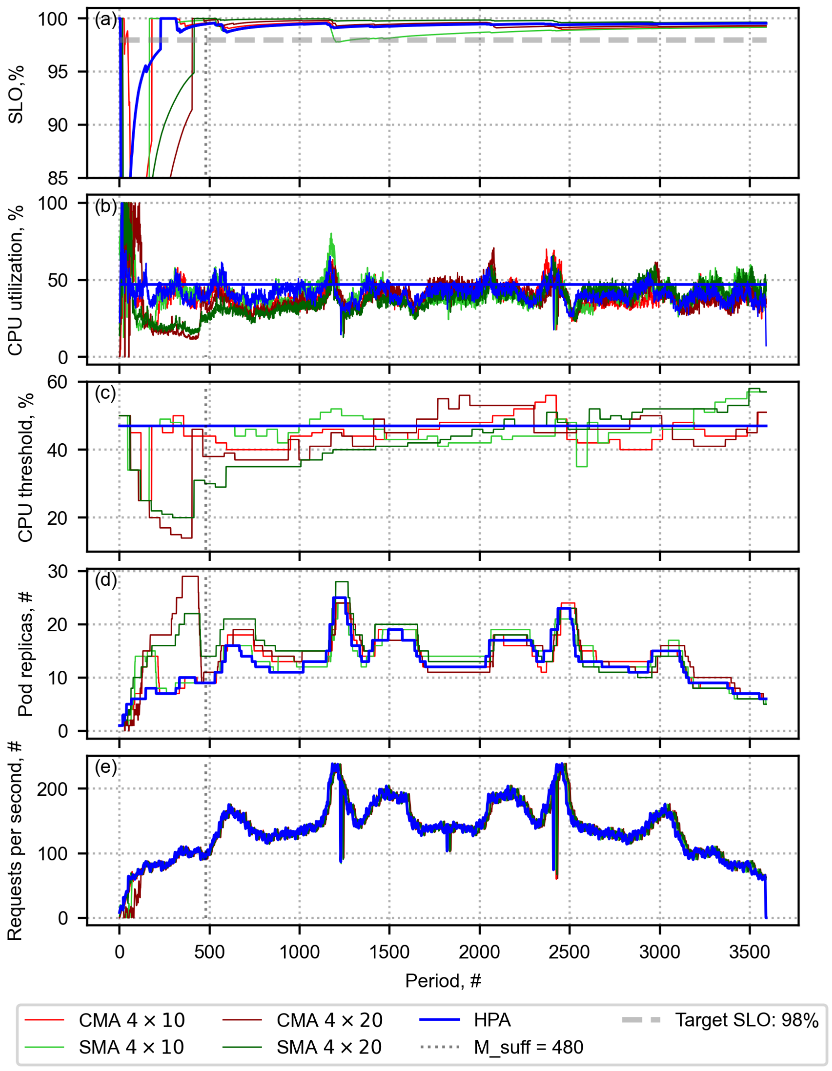


Fig. 3.17. Evaluation of the SLA-adaptive threshold adjustment (SATA) solution using two different smoothing techniques – Simple Moving Average (SMA) and Centered Moving Average (CMA)—with two different settings: $T_{adjust} = 4$ and $T_{eval} = 10$ autoscaling periods (4 × 10) and $T_{adjust} = 4$ and $T_{eval} = 20$ autoscaling periods (4 × 20) for the WorldCup'98 workload scenario when Java application is deployed. (a) Achieved SLO value after collecting a sufficient number of events. (b) Average CPU utilization. (c) The number of pods provisioned in each period. (d) The applied target utilization threshold in each period. (e) Generated workload requests per second

Table 3.10. Results of evaluation of the SLA-adaptive threshold adjustment (SATA) solution using two different smoothing techniques – Simple Moving Average (SMA) and Centered Moving Average (CMA) – and Horizontal Pod Autoscaler (HPA) in the Java application and WorldCup'98 workload scenario[1]

Settings	CMA 4×10	CMA 4×20	SMA 4×10	SMA 4×20	HPA 47%
SLO supported	Fully	Fully	**Fully**	Fully	Fully
sMAPE [2], %	1.3	1.4	**0.9**	1.8	1.4
Total pods used (P_{total})	43769	44652	43745	46161	**42178**
Difference from the best result for total pods in %	4	6	4	9	**0**

[1] Description of the evaluation criteria (sMAPE, Total Pods, and SLO supported) is provided in sub-chapter 2.2.4.
[2] Symmetric Mean Absolute Percentage Error (sMAPE).

Based on Figures 3.17a, b, a noticeable decrease occurs in the SLO value when the CPU threshold surpasses the target utilization threshold of 47% (dark blue dashed line in Figure 3.17b).

As seen in Figure 3.17c, in the event of sudden load increases, the algorithms decrease the threshold, and vice versa. At the same time, the algorithms tend to approach the target utilization value, aiming to increase efficiency by adjusting the provisioned resource number to actual resource demand. When the load decreases, SMA sets higher thresholds than the CMA, improving efficiency in downscale scenarios. However, higher threshold selection caused by SMA might become less effective in upscale scenarios, as it increases SLO violation risk, especially when load is volatile. This can be observed in the last half of the experiment (Fig. 3.17d) for downscale scenarios and in the first half of the experiment for upscale scenarios.

Based on Table 3.10, SATA required slightly more resources in all scenarios than the HPA. The sMAPE values suggest that SATA was more precise in the 4×10 scenarios, with sMAPE of 0.9% for SMA and 1.3% for CMA. However, in the 4×20 scenarios, SATA was less precise, with an sMAPE of 1.4% for CMA and 1.8% for SMA. Furthermore, the precision results were lower in comparison to HPA, which exhibited an sMAPE similar to CMA's (1.4%) When using the CMA as the smoothing method, the duration of the threshold evaluation period did not significantly impact the efficiency and accuracy of SATA. However, the adoption of a more extended period with the SMA positively impacted the precision of SATA under this workload scenario, consequently leading to stable service quality delivery.

The experiment with the SMA in a 4 $\times$ 10 scenario demonstrated the highest accuracy across SATA algorithms while supporting the defined SLO. The CMA slightly overprovisioned resources compared to the HPA in a 4 4 $\times$ 10 scenario scenario. However, the SMA in a 4 $\times$ 10 scenario was more accurate in meeting the desired SLO. This is because SATA with SMA smoothing has greater accuracy in detecting threshold values, but it becomes more susceptible to unexpected load growth when a shorter threshold evaluation period is applied. This can be seen in Figure 3.17a at around the 600th and 1200th metrics' collection periods (light green line).

The HPA using a static threshold was the most efficient, demonstrating precision and accuracy comparable to SATA with the CMAin a 4 $\times$ 10 scenario. It is important to note that achieving such precision in practice is challenging without prior knowledge of the workload. The resource overprovisioning of 4% produced by SATA could be considered as negligible as it is within the HPA tolerance range of 10% (kubernetes.io, 2022) and is expected from SLA-fulfillment-oriented solutions (Kim, Kim, Lee, & Yu, 2025; Taherizadeh & Stankovski, 2019; Tonini et al., 2023). For example, Pramesti and Kistijantoro (2022) conducted experiments demonstrating that autoscalers that use a performance-based SLI, such as response time, to ensure compliance with the SLA, require more resources than the HPA, which uses CPU-utilization-based thresholds.

Interestingly, the HPA enabled by the SATA solution accomplished the SLO support objective solely by adjusting the threshold values. As demonstrated by the experiment, the SATA solution is responsive to load fluctuations and automatically detects the threshold values to achieve the expected SLO. When downscaling occurs, the SATA increases the threshold value, facilitating quicker downscaling and resulting in cost savings. Conversely, when there is an increase in load, it lowers the threshold, enabling faster resource provisioning and reducing the likelihood of SLO violations.

The following text describes the experiments with a slightly shaky workload scenario.

Java&EDGAR Experiment Description and Results

In this experiment, the EDGAR workload was used to assess algorithms' ability to adjust thresholds under the real-world shaky load pattern. The load is presented in Figure 3.18e. The load contains a number of periods where the load changes unexpectedly with high magnitude, that is, the number of requests that change more than twice in a short period of time, e.g., see the number of events at around 600th the 900th monitoring periods. The load is highly volatile but has much lower fluctuations in magnitude for the rest of the experiment.

The experiment results are presented in Figure 3.18 and Table 3.11. As seen in Figure 3.18a, all of the evaluated approaches were compliant with the SLO throughout the experiment, except for the SATA CMA 4 $\times$ 10 use case. At the begining of the experiment, the SATA CMA 4 $\times$ 10 set higher than required threshold. However, the threshold was adjusted in the next two upscale periods, and hence, the operation was restored to the desired level.

As seen in Figure 3.18a and b, any increase in CPU utilization above 35% caused a decrease in the SLO value. As presented in Figure 3.18c, SATA sets lower threshold values in all cases. This suggests that the solution is sensitive to frequent load fluctuations. However, longer threshold evaluation periods resulted in better stability and adjustment of the threshold to a value close to 35%. All scenarios, except for CMA 4 $\times$ 10, showed similar accuracy, with SMA 4 $\times$ 20 demonstrating slightly better accuracy in identifying the target utilization threshold. Thus, HPA with SMA 4 $\times$ 20 operated closer to the desired utilization threshold of 35%, and was more efficient than other SATA setups, as seen in Figure 3.18d and Table 3.11.

The experiment demonstrated that in the HPA without the SMA scenario, the SLO values began to decrease towards the end of the experiment, as depicted in Figure 3.18a. This suggests that the current baseline threshold may not be adequate for future workload changes and could fail to restore the SLO. In contrast, the SATA approaches have demonstrated effectiveness in maintaining compliance with the SLO, as shown by the results.

As presented in Table 3.11, the HPA showed the best resource-management efficiency and accuracy across algorithms that met the SLO during all evaluation periods. The CMA 4 $\times$ 10 was the most accurate but was not able to support the SLO in all periods, even though it was the second most overprovisioning solution in this experiment. As presented in sub-chapter 2.2.2, while CMA's tendency to underestimate the target utilization threshold improves efficiency in volatile load scenarios, it causes a decrease in efficiency when the load is stable, as evidenced by the data presented in Table 3.11. The results indicate that SMA 4 $\times$ 20 achieved the highest levels of accuracy and efficiency across all SATA settings, with only 10% overprovisioning compared to the HPA using a static threshold setup. As discussed in sub-chapter 3.3.2, some overprovisioning is expected, and a 10% overprovisioning rate can be regarded as a decent result when the algorithm aims to ensure SLA fulfillment.

The assessment of the algorithm's performance reproducibility in a different environment is presented below.

Rust&WorldCup'98 Experiment Description and Results

To validate that the above-mentioned results are not biased to specific applications and environments, another set of experiments was executed, but using different application and infrastructure setups. The infrastructure and application were presented in sub-chapters 3.2.2 and 3.3.1.

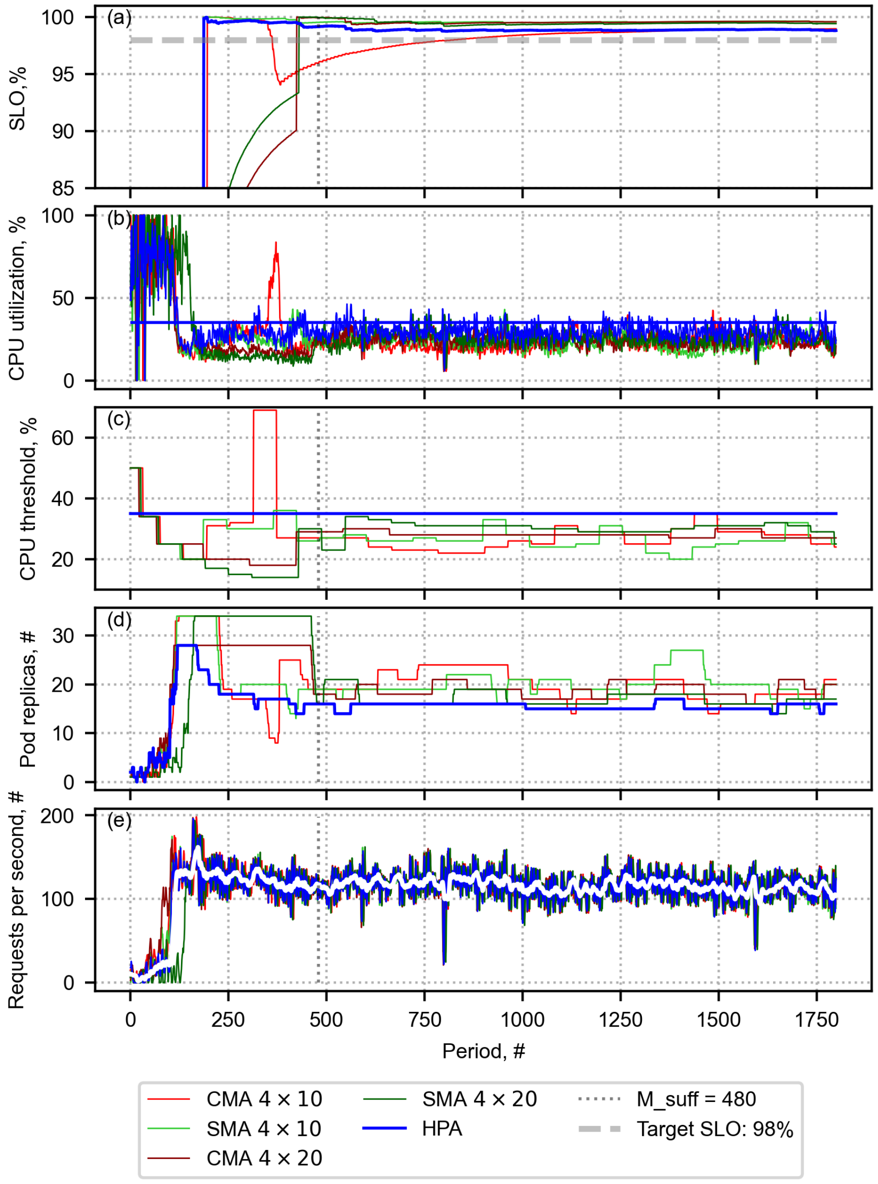


Fig. 3.18. Evaluation of the SATA solution using SMA and CMA with two different settings: 4 × 10 and 4 × 20 for the EDGAR workload scenario when Java application is deployed. (a) Achieved SLO value after collecting a sufficient number of events. (b) Average CPU utilization. (c) Number of pods provisioned in each period. (d) Applied target utilization threshold in each period. (e) Generated workload requests per second

Table 3.12 presents the combined results that are used as the basis for concluding the performance evaluation of the algorithm under WorldCup’98 and EDGAR loads conditions while SMA 4×10 settings are applied. According to the results, the algorithm showed similar levels of overprovisioning and slightly lower precision compared to experiments with Java applications.

The evaluation of the algorithm in the WorldCup’98 workload scenario using Rust application experiment results is presented in Figure 3.19 and Table 3.12. The results show that all of the evaluated approaches were compliant with SLO throughout the experiment.

The load pattern and amount of the requests processed per second are presented in Figure 3.19e.

The SLO value presented in Figure 3.19 decreases at the times when the HPA target utilization threshold of 69% (represented by the bold dark blue line in Figure 3.19b) is violated.

Graph c of Figure 3.19 shows that the algorithm exhibited the self-adjustment behavior required to meet the SLO targets. The algorithm lowered the threshold settings when the SLO dropped and then consistently increased them while the SLO was above the target. As depicted in Figure 3.19d, this adjustment led to more efficient resource usage than HPA when the SLO was above the target or during periods without high-load spikes. However, during periods of high load increase, the algorithm lowered the thresholds to enable faster provisioning of resources, which resulted in overprovisioning compared to HPA. Notably, this adjustment was based solely on the suggested thresholds from the method described in Sub-chapter 2.2.3,

Table 3.11. Results of evaluation of the SLA-adaptive threshold adjustment (SATA) solution using two different smoothing techniques – Simple Moving Average (SMA) and Centered Moving Average (CMA) – and Horizontal Pod Autoscaler (HPA) in the Java application and EDGAR workload scenario[1]

Settings	CMA 4×10	CMA 4×20	SMA 4×10	SMA 4×20	HPA 35%
SLO supported	Partially	Fully	Fully	**Fully**	Fully
sMAPE [2], %	0.7	1.6	1.6	1.5	**0.9**
Total pods used (P_{total})	25899	24606	26523	22517	**20501**
Difference from the best result for total pods in %	27	20	29	10	**0**

[1] Description of the evaluation criteria (sMAPE, Total Pods, and SLO supported) is provided in sub-chapter 2.2.4.

[2] Symmetric Mean Absolute Percentage Error (sMAPE).

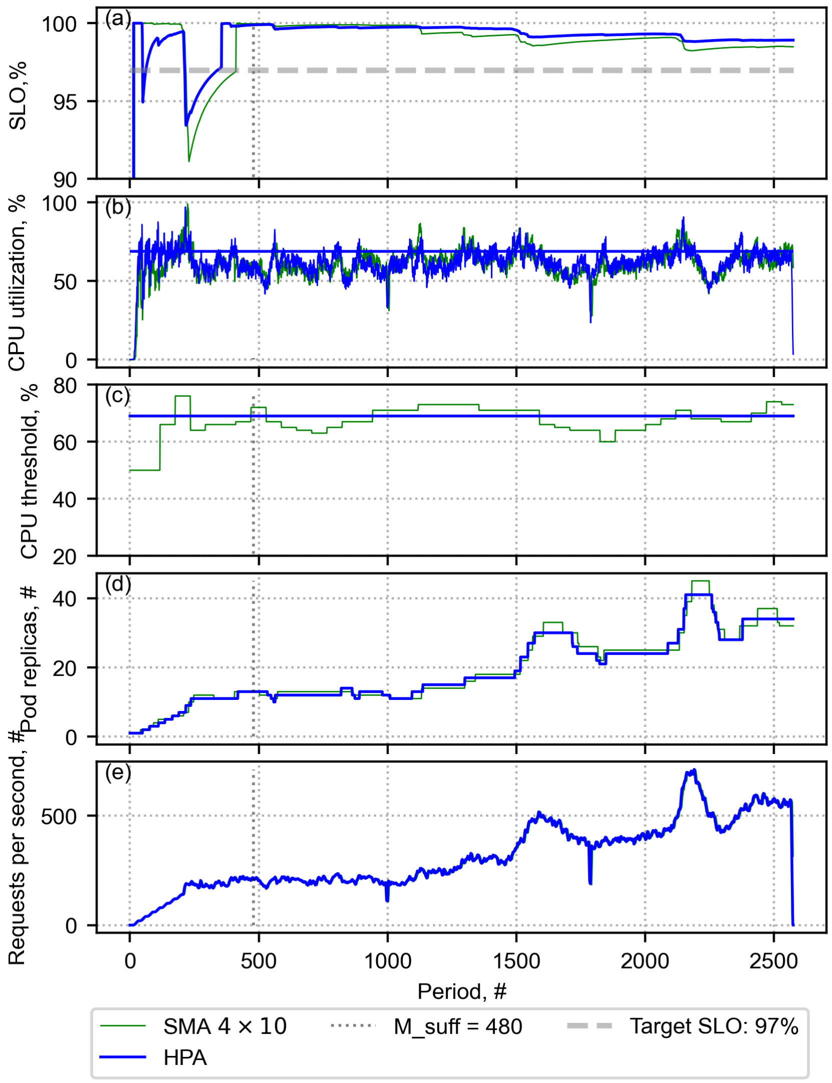


Fig. 3.19. Evaluation of SATA using Simple Moving Average (SMA) smoothing technique with $T_{adjust} = 4$ and $T_{eval} = 10$ autoscaling periods (4×10) setting for the WorldCup'98 workload scenario when Rust application is deployed. (a) Achieved SLO value after collecting a sufficient number of events. (b) Average CPU utilization. (c) Number of pods provisioned in each period. (d) Applied target utilization threshold in each period. (e) Generated workload requests per second

Table 3.12. Results of evaluation of the SLA-adaptive threshold adjustment (SATA) solution using Simple Moving Average smoothing technique and Horizontal Pod Autoscaler (HPA) in the Rust application and WorldCup'98 and EDGAR workload scenarios[1]

Workload	WorldCup'98		EDGAR	
Solution	HPA 69%	SATA 4×10	HPA 42%	SATA 4×10
SLO supported	Fully	Fully	Fully	Fully
sMAPE [2] , %	2.7	**2.5**	**1.3**	2.8
Total pods used (P_{total})	**45199**	46046	**23379**	29015
Difference from the best result for Total Pods in %	**0**	1.8	0	25

[1] Description of the evaluation criteria (sMAPE, Total Pods, and SLO supported) is provided in Sub-chapter 2.2.4.

[2] Symmetric Mean Absolute Percentage Error (sMAPE).

without the use of explicit rules for SLO recovery. This behavior was consistently observed across different experimental environments. This experiment provided a better understanding of the suggested algorithm's behavior, showing how threshold adaptation to high load spikes contributed to overprovisioning while improving application performance.

The subsequent experiment, involving the Rust application within a slightly fluctuating workload scenario, is presented below.

Rust&EDGAR Experiment Description and Results

The goal of this experiment was to evaluate the algorithms' capacity to adjust thresholds in response to a real-world shaky load pattern utilizing the Rust application. The results are presented in Figure 3.20 and Table 3.12.

As presented in Figure 3.18a, both solutions were able to meet their SLO. However, SMA continued to exhibit sensitivity to volatile loads and, as expected, showed higher overprovisioning, similar to the experiment with the Java application. Recovery from the initial threshold value drop (Fig. 3.20c) took considerable time. This drop prevented HPA from entering a low pod number provisioning loop, as described in Sub-chapter 2.2.2 Step 3.

The following sub-chapter aims to discuss the achieved results.

3.3.3. Discussion

SATA was compared with HPA to assess SLA awareness and performance improvements brought to HPA once SATA was incorporated into HPA. Additionally, the impact of various settings applied to the SATA on both efficiency and effec-

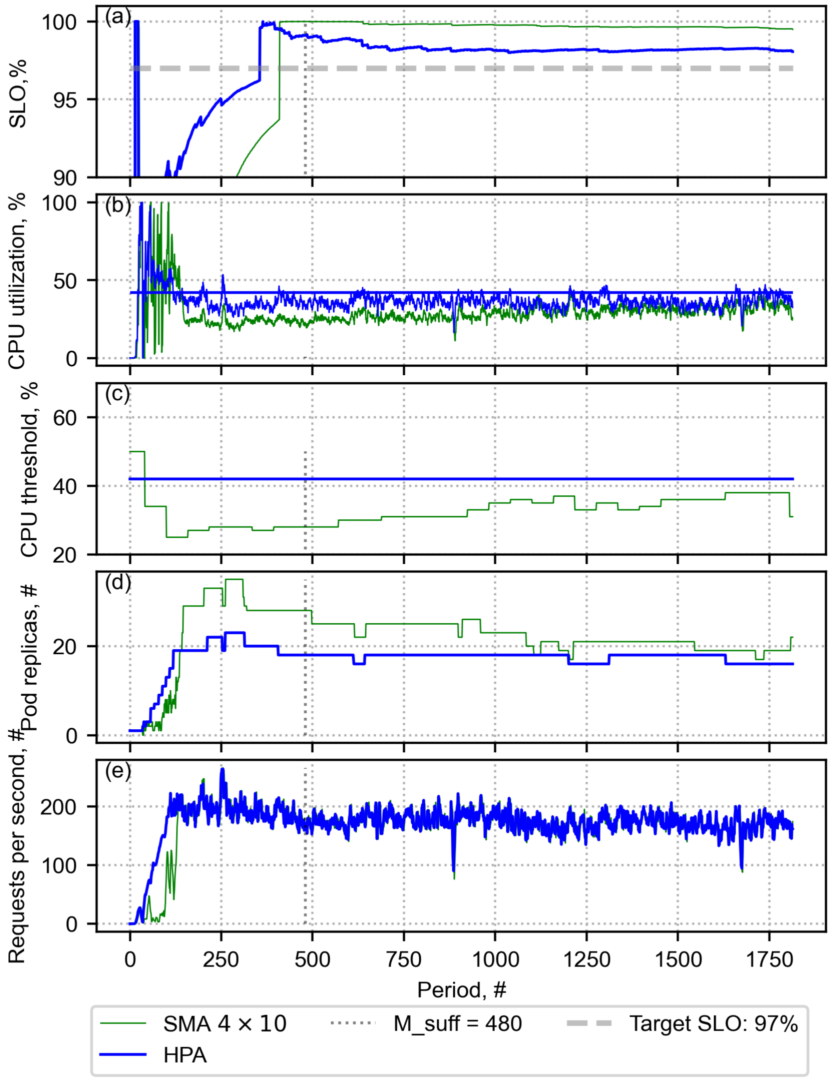


Fig. 3.20. Evaluation of SATA using Simple Moving Average (SMA) smoothing technique with $T_{adjust} = 4$ and $T_{eval} = 10$ autoscaling periods (4×10) setting for the EDGAR workload scenario when Rust application is deployed. (a) Achieved SLO value after collecting a sufficient number of events. (b) Average CPU utilization. (c) Number of pods provisioned in each period. (d) The applied target utilization threshold in each period. (e) Generated workload requests per second

tiveness was analyzed. The key findings discovered during the investigation are provided below:

1. The experiments conducted with the different workloads demonstrated that the solution could identify thresholds that allow the system to operate close to the defined SLO, however, with slight overprovisioning.
2. The experiments demonstrated that the solution efficiency and effectiveness depend on multiple factors: type of moving average utilized, length of threshold evaluation and frequency of threshold adjustment, load pattern and service level indicator used to measure violation numbers. Specifically, the algorithm is more sensitive to volatile load when shorter threshold evaluation periods are used. It also demonstrates better efficiency in cases of non-volatile load. The type of moving average can also be selected to control efficiency based on the load pattern, whether it is highly volatile or not. The use of the average response time metric as SLI, showed lower efficiency and effectiveness results compared to the use of the 98th percentile. Centred moving average with a threshold adjustment period of length of 3 to 2.5 evaluation periods showed better or similar effectiveness and efficiency results compared to other settings evaluated.
3. The amount of resource overprovisioning compared with HPA configured with the most optimal static threshold can vary from negligible to deviating from 10% to 30% in volatile load scenarios. It is important to note that the desired HPA threshold value is never known upfront due to constantly varying conditions, so, in practice, it is challenging to achieve such precision threshold settings. As a result, such a level of overprovisioning is expected and can be considered as an acceptable efficiency result.
4. The tracking of the SLO state supported the prototype in more suitable threshold selection, which enabled it to achieve a system operation state as close as possible to the desired SLO target in case of the SLO decrease, showing similar to the previously presented SAA algorithm, but with less setting and parameters to adjust.
5. The algorithm consistently displayed similar behavior across different applications, even under similar workload conditions. This suggests that the algorithm's behavior is reproducible in diverse environments.
6. The experiments revealed that, although the same application and pods were used with identical resource settings, different target utilization values must be applied based on the load pattern to ensure compliance with the SLO. This observation leads to the conclusion that methods relying on load testing a single pod instance to determine the maximum CPU utilization that allows the application's performance to meet SLO requirements may not be adequate for establishing the target utilization threshold for the HPA.

7. While the intention was to develop a solution, which is as simple as possible for CPU threshold determination, it still requires further development and improvements as it is yet in the early stages. For instance, the threshold determination algorithm relies on data points that align closely with the desired SLO. The SATA prototype facilitates the collection of these data points. Future efforts should focus on further maturing the threshold determination and adjustment algorithms by enhancing existing rules and experimenting with other statistical techniques. Also, additional development is needed to automatically determine the threshold length and improve algorithm efficiency based on recommendations provided in this work. These enhancements should improve the algorithm's stability and efficiency, especially in volatile workload cases. Future research and improvements could also include further prototype stability and efficiency improvement, an adaptation of the approach to other threshold-based autoscaling solutions, and the use of different service level indicators and utilization metrics. It would be beneficial to experiment with various statistical methods in a broader range of cloud-native scenarios. The improvements and recommendations mentioned above will support the real-world implementation of the solution, especially in tasks suitable for parallelization, such as image processing, format conversion, transcoding, edge computing, and serverless services like functions as a service.

3.4. Conclusions of the Third Chapter

The chapter details an experimental investigation of two proposed SLA-aware, rules-based autoscaling solutions: the SLA-Adaptive Autoscaler (SAA) and the SLA-Adaptive Threshold Adjuster (SATA). The experiments presented in this chapter were executed on the Azure Kubernetes Service platform using different CPU-intensive applications and workloads. These experiments assess the efficacy of achieving SLA fulfillment objectives and resource provisioning efficiency. In conclusion, the conducted experiments indicated that:

1. Both of the proposed solutions were demonstrated to be effective in ensuring performance-based SLA compliance.
2. The inclusion of service-level objective tracking supported both of the solutions in achieving their goals of SLA compliance assurance.
3. Dynamic threshold manipulation has proven effective for ensuring performance-based SLA fulfillment in applications where the threshold metric is the most influential SLA parameter.

Known limitations and threats to the validity of the conducted research are provided below:

1. The average response time metric was utilized as a service level indicator. Using different service level indicators to measure SLO might reveal variances in the efficacy of the assessed algorithms.
2. The experiment was conducted using Azure Kubernetes Service, which is a cloud environment. Measured benchmark and efficiency results can be different when using another cloud provider or service, as can the size of resources or event time for experiment execution due to the non-homogeneity of the cloud environment.
3. It is important to note that the efficiency results may vary depending on the execution environment, load patterns, or type of applications.

General Conclusions

The present study contributes to a better understanding of the SLA-aware autoscaling algorithms and approaches used to address the performance-based SLA fulfillment aspect for cloud-native containerized applications. The performed research can be concluded as follows:

1. The literature review revealed that the cloud-native paradigm is a topic of significant investigation within academia and industry. Interest in SLA fulfillment in cloud-native applications autoscaling has been rising in the last few years. The research showed that many autoscaling strategies for cloud-native applications have been developed, ranging from straightforward rule-based policies to sophisticated machine-learning models. Machine learning-based algorithms commonly make decisions in a more timely manner compared to rules-based solutions; however, rule-based autoscaling solutions are prevalent in the industry due to their simplicity, with the Horizontal Pod Autoscaler emerging as the most widely adopted approach. Given the difficulties in achieving timely decision-making, mechanisms for recovering from SLA violations may be employed as a remediation strategy in rules-based autoscalers. The literature review identifies that no such recovery mechanisms have been proposed in the academic literature. Additionally, the literature review revealed that most of the analyzed solutions made decisions based solely on immediate changes in service-level indicator values. These solutions did not implement longer SLA status tracking timeframes in SLA fulfillment-oriented autoscalers,

resulting in a lack of overall SLA compliance awareness. Furthermore, the review revealed the lack of consistent evaluation methodologies that effectively measure the efficiency and efficacy of autoscaling solutions in the context of SLA fulfillment.

2. The proposed SAA solution implements the mechanism to recover the SLO in situations where adding resources can significantly enhance service performance. In cases of service-level degradation, the solution either adds additional resources or pauses downscaling actions until the defined service levels are achieved. During experiments, the SLO recovery capability and performance efficiency were compared with DMAR and HPA solutions. The ability to recover SLO as fast as possible and maintain SLO as long as possible during the experiment was proposed as a method for evaluating the efficacy of the solution in fulfilling SLA requirements. The "touchstone" autoscaler was also suggested as a benchmark for assessing overall resource provisioning efficiency. SAA demonstrated better results than HPA and DMAR in its ability to recover and maintain defined SLA targets in six out of seven conducted experiments. The resource overprovisioning was observed to be 1.5 to 3.5 times higher than that of the "touchstone autoscaler," depending on the type of workload. This level of overprovisioning was comparable to that of the DMAR, which solely relied on tracking current SLI value violations for SLA awareness. The achieved results support the first defended statement that recovering SLO compliance by adding additional resources can be used to improve the fulfillment of the defined SLO. This approach is particularly relevant for embarrassingly parallel workloads, where adding resources can positively impact SLO fulfillment. The need for manual parameter estimation complicates the real-world adoption of SAA. To streamline solution adoption and improve efficiency, future work should focus on automating the adjustment or determination of parameter values (such as cooldown periods, CPU threshold ranges, and baseline velocity) using statistical techniques or machine learning.
3. The proposed SATA solution is a dynamic threshold adjustment add-on for threshold-based autoscalers, enabling them to meet performance-based SLO requirements when the utilization threshold is the most influential SLA factor. By using the number of SLA violations collected during a defined timeframe and resource utilization as inputs for threshold determination, SATA calculates the number of violations that appeared at particular utilization levels to identify target thresholds. SATA was adopted by HPA to assess the effectiveness of the proposed approach in improving SLA fulfillment. During experiments executed as part of this research, the HPA, enabled by the SATA solution, demonstrated self-adaptation to

environmental performance changes, achieving performance levels that maintained the target SLO with a precision of 1–2.7%. It successfully maintained the SLO in 15 out of 16 evaluated cases, even without prior knowledge of the required target utilization threshold. Resource overprovisioning ranged from 10% to 30% compared to the resources utilized by the HPA with a static target threshold set to the highest value that allowed meeting the SLO. The overprovisioning level directly depended on workload volatility. The experiments suggest that the dynamic manipulation of the utilization threshold can be used to improve the fulfillment of SLAs in rules-based autoscaling solutions when the utilization threshold is the most influential SLA factor. Future research should focus on improving threshold algorithms for better stability and efficiency in volatile workloads through the improvement of existing rules and exploration of other statistical techniques. Additionally, experimenting with the solution in broader cloud-native scenarios and adopting other service-level indicators should aid in the real-world adoption of the solution.

4. SAA and SATA solutions used the number of SLA violations collected over a longer timeframe in autoscaling decision-making together with momentary resource utilization values used as scaling indicators. Both solutions have demonstrated better SLA-fulfillment results compared to the other evaluated solutions as presented earlier. The findings suggest that monitoring SLO status over an extended period, alongside traditional SLA violation avoidance methods in rules-based cloud-native application autoscalers, enhances compliance with performance-based SLOs.

References

Abdullah, M., Iqbal, W., Berral, J. L., Polo, J., & Carrera, D. (2022). Burst-aware predictive autoscaling for containerized microservices. *IEEE Transactions on Services Computing*, *15*, 1448–1460. www.ieee.org/publications/rights/index.html

Agarwal, S. (2020). *An approach of sla violation prediction and qos optimization using regression machine learning techniques.* https://scholar.uwindsor.ca/etd/8342

Ahmad, I., AlFailakawi, M. G., AlMutawa, A., & Alsalman, L. (2022, 7). Container scheduling techniques: A survey and assessment. *Journal of King Saud University - Computer and Information Sciences*, *34*, 3934–3947.

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., & Merle, P. (2018). Elasticity in cloud computing: State of the art and research challenges. *IEEE Transactions on Services Computing*, *11*, 430–447. https://hal.inria.fr/hal-01529654

Al-Haidari, F., Sqalli, M., & Salah, K. (2013, 12). Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources. In (pp. 256–261). IEEE. http://ieeexplore.ieee.org/document/6735431/

Alonso, J., Orue-Echevarria, L., Casola, V., Torre, A. I., Huarte, M., Osaba, E., & Lobo, J. L. (2023, 12). Understanding the challenges and novel architectural models of multi-cloud native applications – a systematic literature review. *Journal of Cloud Computing*, *12*, 1–34. https://journalofcloudcomputing.springeropen.com/articles/10.1186/s13677-022-00367-6

Amiri, M., & Mohammad-Khanli, L. (2017). Survey on prediction models of applications for resources provisioning in cloud. *Journal of Network and Computer Applications*, *82*, 93–113.

Andrikopoulos, V., Binz, T., Leymann, F., & Strauch, S. (2013). How to adapt applications for the cloud environment: Challenges and solutions in migrating applications to the cloud. *Computing*, *95*, 493–535. http://link.springer.com/10.1007/s00607-012-0248-2

Andrikopoulos, V., Strauch, S., Fehling, C., & Leymann, F. (2012). Cap-oriented design for cloud-native applications. In *Proceedings of the 2nd international conference on cloud computing and service science, closer 2012, 18-21 april 2012, porto, portugal* (pp. 365–374). SciTePress. http://www.iaas.uni-stuttgart.de/RUS-data/INPROC-2012-09-DesigningforCAP.pdfhttp://link.springer.com/10.1007/978-3-319-04519-1_14

Arapakis, I., Park, S., & Pielot, M. (2021, 3). Impact of response latency on user behaviour in mobile web search. *CHIIR 2021 - Proceedings of the 2021 Conference on Human Information Interaction and Retrieval*, 279–283. https://dl.acm.org/doi/10.1145/3406522.3446038

Arlitt, M., & Jin, T. (2000, 8). Workload characterization of the 1998 world cup web site. *IEEE Network J*, *14*.

Augustyn, D. R., Wyci´slik, L. W., & Sojka, M. (2024, 1). Tuning a kubernetes horizontal pod autoscaler for meeting performance and load demands in cloud deployments. *Applied Sciences 2024, Vol. 14, Page 646*, *14*, 646. https://www.mdpi.com/2076-3417/14/2/646/htmhttps://www.mdpi.com/2076-3417/14/2/646

Balla, D., Simon, C., & Maliosz, M. (2020, 4). Adaptive scaling of kubernetes pods. *Proceedings of IEEE/IFIP Network Operations and Management Symposium 2020: Management in the Age of Softwarization and Artificial Intelligence, NOMS 2020*.

Baresi, L., Hu, D. Y. X., Quattrocchi, G., & Terracciano, L. (2021). Kosmos: Vertical and horizontal resource autoscaling for kubernetes. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *13121 LNCS*, 821–829.

Beloglazov, A., & Buyya, R. (2010, 11). Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In (pp. 1–6). ACM. https://dl.acm.org/doi/10.1145/1890799.1890803

Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (2016). *Site reliability engineering: How google runs production systems* (1st ed.; B. Anderson, Ed.). O'Reilly Media, Inc. https://research.google/pubs/pub45305/

Bogachev, M. I., Kuzmenko, A. V., Markelov, O. A., Pyko, N. S., & Pyko, S. A. (2023, 3). Approximate waiting times for queuing systems with variable long-term correlated arrival rates. *Physica A: Statistical Mechanics and its Applications*, *614*, 128513.

Brown, K., & Capern, M. (2014). Top 9 rules for cloud applications. *IBM Middleware Technical Journal for Developers*. http://www.ibm.com/developerworks/websphere/techjournal/1404_brown/1404_brown.html

Brunner, S., Blochlinger, M., Toffetti, G., Spillner, J., & Bohnert, T. M. (2016). Experimental evaluation of the cloud-native application design. In *Proceedings - 2015 ieee/acm 8th international conference on utility and cloud computing, ucc 2015* (pp. 488–493). (Not good enought - no new info)

Carrión, C. (2022, 12). Kubernetes as a standard container orchestrator - a bibliometric analysis. *Journal of Grid Computing*, *20*, 1–23. https://link.springer.com/article/10.1007/s10723-022-09629-8

Casalicchio, E. (2019, 9). A study on performance measures for auto-scaling cpu-intensive containerized applications. *Cluster Computing*, *22*, 995–1006. https://link.springer.com/article/10.1007/s10586-018-02890-1

Casalicchio, E., & Perciballi, V. (2017). Auto-scaling of containers: The impact of relative and absolute metrics. *Proceedings - 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems, FAS*W 2017*, 207–214.

Casper, D., Bette, C., & Louie, K. (2014). *Best practices : Architecting cloud-aware applications.* Open Data Center Alliance. https://www.opendatacenteralliance.org//docs/architecting_cloud_aware_applications.pdf

Chen, C., Twycross, J., & Garibaldi, J. M. (2017, 3). A new accuracy measure based on bounded relative error for time series forecasting. *PLoS ONE*, *12*. /pmc/articles/PMC5365136//pmc/articles/PMC5365136/?report=abstracthttps://www.ncbi.nlm.nih.gov/pmc/articles/PMC5365136/

Clarivate. (2024). *Web of science master journal list - search.* https://mjl.clarivate.com/search-results

CNCF. (2024a). *Cncf landscape.* https://landscape.cncf.io/stats

CNCF. (2024b). *foundation/charter.md at main · cncf/foundation · github.* https://github.com/cncf/foundation/blob/main/charter.md

CNCF. (2024c). *Reports | cncf.* https://www.cncf.io/reports/?_sft_lf-report-type=survey

Dang-Quang, N.-M., Yoo, M., De, J. F., & Santana, P. (2021, 4). Deep learning-based autoscaling using bidirectional long short-term memory for kubernetes. *Applied Sciences 2021, Vol. 11, Page 3835*, *11*, 3835. https://www.mdpi.com/2076-3417/11/9/3835/htmhttps://www.mdpi.com/2076-3417/11/9/3835

Dash, C. S. K., Behera, A. K., Dehuri, S., & Ghosh, A. (2023, 3). An outliers detection and elimination framework in classification task of data mining. *Decision Analytics Journal*, *6*, 100164.

Datadog. (2024). *10 insights on real-world container use | datadog.* https://www.datadoghq.com/container-report/

Deng, S., Zhao, H., Huang, B., Zhang, C., Chen, F., Deng, Y., … Zomaya, A. Y. (2024, 1). Cloud-native computing: A survey from the perspective of services. *Proceedings of the IEEE*, *112*, 12–46. https://ieeexplore.ieee.org/document/10433234/

Developers, G. (n.d.). *Best practices for running cost-optimized kubernetes applications on gke | cloud architecture center | google cloud.* https://cloud.google.com/architecture/best-practices-for-running-cost-effective-kubernetes-applications-on-gke#horizontal_pod_autoscaler

Ding, Z., & Huang, Q. (2021). Copa: A combined autoscaling method for kubernetes. *2021 IEEE International Conference on Web Services (ICWS).*

Dixit, A., Gupta, R. K., Dubey, A., & Misra, R. (2022). Machine learning based adaptive auto-scaling policy for resource orchestration in kubernetes clusters. *Lecture Notes in Networks and Systems*, *340 LNNS*, 1–16. https://link.springer.com/chapter/10.1007/978-3-030-94507-7_1

Fehling, C., Leymann, F., Mietzner, R., & Schupeck, W. (2011). *Universität stuttgart a collection of patterns for cloud types , cloud service models , and cloud-based application architectures institute of architecture of application systems daimler ag.*

Fehling, C., Leymann, F., Retter, R., Schupeck, W., & Arbitter, P. (2014). *Cloud computing patterns*. Springer Vienna. http://link.springer.com/10.1007/978-3-7091-1568-8

Gannon, D., Barga, R., & Sundaresan, N. (2017, 9). Cloud-native applications. *IEEE Cloud Computing*, *4*, 16–21.

Han, J., Kamber, M., & Pei, J. (2012, 1). Getting to know your data. *Data Mining*, 39–82.

Herbst, N., Kounev, S., & Reussner, R. (2013). Elasticity in cloud computing : What it is , and what it is not. In *Presented as part of the 10th international conference on autonomic computing* (pp. 23–27). USENIX. http://sdqweb.ipd.kit.edu/publications/pdfs/HeKoRe2013-ICAC-Elasticity.pdf

Herbst, N., Krebs, R., Oikonomou, G., Kousiouris, G., Evangelinou, A., Iosup, A., & Kounev, S. (2016, 4). *Ready for rain? a view from spec research on the future of cloud metrics* (Tech. Rep.). http://arxiv.org/abs/1604.03470

Hole, K. J. (2016). Toward an anti-fragile e-government system. In (Vol. 1, pp. 57–65). Springer International Publishing. http://link.springer.com/10.1007/978-3-319-30070-2_6

Horovitz, S., & Arian, Y. (2018, 9). Efficient cloud auto-scaling with sla objective using q-learning. *Proceedings - 2018 IEEE 6th International Conference on Future Internet of Things and Cloud, FiCloud 2018*, 85–92.

Hu, T., & Wang, Y. (2021, 1). A kubernetes autoscaler based on pod replicas prediction. In Liu (Ed.), (pp. 238–241). IEEE. https://ieeexplore.ieee.org/document/9407757/

Huo, Q., Li, C., Li, S., Xie, Y., & Li, Z. (2023, 3). High concurrency response strategy based on kubernetes horizontal pod autoscaler. *Journal of Physics: Conference Series*, *2451*, 012001. https://iopscience.iop.org/article/10.1088/1742-6596/2451/1/012001https://iopscience.iop.org/article/10.1088/1742-6596/2451/1/012001/meta

Huo, Q., Li, S., Xie, Y., & Li, Z. (2022). Horizontal pod autoscaling based on kubernetes with fast response and slow shrinkage. *Proceedings - 2022 International Conference on Artificial Intelligence, Information Processing and Cloud Computing, AIIPCC 2022*, 203–206.

Hyndman, R. J. (2014, 1). *Rob j hyndman - thoughts on the ljung-box test.* https://robjhyndman.com/hyndsight/ljung-box-test/

Hyndman, R. J., & Athanasopoulos, G. (2019). *Forecasting: Principles and practice, 2nd edition.*

IEEEXplore. (2024). *Ieee xplore search results.* https://ieeexplore.ieee.org/search

Ilyushkin, A., Ali-Eldin, A., Herbst, N., Bauer, A., Papadopoulos, A. V., Epema, D., & Iosup, A. (2018, 6). An experimental performance evaluation of autoscalers for complex workflows. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, *3*, 1–32.

Imdoukh, M., Ahmad, I., & Alfailakawi, M. G. (2020, 7). Machine learning-based auto-scaling for containerized applications. *Neural Computing and Applications*, *32*, 9745–9760. https://link.springer.com/article/10.1007/s00521-019-04507-z

Inzinger, C., Nastic, S., Sehic, S., Vogler, M., Li, F., & Dustdar, S. (2014). Madcat: A methodology for architecture and deployment of cloud application topologies. In *Proceedings - ieee 8th international symposium on service oriented system engineering, sose 2014* (pp. 13–22).

JMeter, A. (2025). *Apache jmeter - apache jmeter™.* https://jmeter.apache.org/

Joyce, J. E., & Sebastian, S. (2023). Enhancing kubernetes auto-scaling: Leveraging metrics for improved workload performance. *2023 Global Conference on Information Technologies and Communications, GCITC 2023.*

Ju, L., Singh, P., & Toor, S. (2021). Proactive autoscaling for edge computing systems with kubernetes. *ACM International Conference Proceeding Series*, *22*, 1–8.

Kang, P., & Lama, P. (2020, 12). Robust resource scaling of containerized microservices with probabilistic machine learning. *Proceedings - 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing, UCC 2020*, 122–131.

Kavis, M. J. (Ed.). (2014). John Wiley & Sons, Inc. http://doi.wiley.com/10.1002/9781118691779

Kazanavicius, J., & Mazeika, D. (2019, 4). Migrating legacy software to microservices architecture. *2019 Open Conference of Electrical, Electronic and Information Sciences, eStream 2019 - Proceedings.*

Kazanavičius, J., Mažeika, D., & Kalibatienė, D. (2022, 6). An approach to migrate a monolith database into multi-model polyglot persistence based on microservice architecture: A case study for mainframe database. *Applied Sciences (Switzerland)*, *12*. https://www.researchgate.net/publication/361384899_An_Approach_to_Migrate_a_Monolith_Database_into_Multi-Model_Polyglot_Persistence_Based_on_Microservice_Architecture_A_Case_Study_for_Mainframe_Database

Keller, A., & Ludwig, H. (2003, 3). The wsla framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, *11*, 57–81. https://link.springer.com/article/10.1023/A:1022445108617

Khaleq, A. A., & Ra, I. (2021). Intelligent autoscaling of microservices in the cloud for real-time applications. *IEEE Access*, *9*, 35464–35476. https://ieeexplore.ieee.org/document/9361549/

Khan, H. M., Chan, G. Y., & Chua, F. F. (2016, 3). An adaptive monitoring framework for ensuring accountability and quality of services in cloud computing. *International Conference on Information Networking*, *2016-March*, 249–253.

Kim, D., Kim, H., Lee, E., & Yu, H. (2025). Lare-hpa: Co-optimizing latency and resource efficiency for horizontal pod autoscaling in kubernetes. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *15405 LNCS*, 19–34. https://link.springer.com/chapter/10.1007/978-981-96-0808-9_2

Koperek, P. L., & Funika, W. L. (2012). Dynamic business metrics-driven resource provisioning in cloud environments.

Kosinska, J., Balis, B., Konieczny, M., Malawski, M., & Zielinski, S. (2023). Toward the observability of cloud-native applications: The overview of the state-of-the-art. *IEEE Access*, *11*, 73036–73052.

Kosińska, J., Brotoń, G., & Tobiasz, M. (2024, 2). Knowledge representation of the state of a cloud-native application. *International Journal on Software Tools for Technology Transfer*, *26*, 21–32. https://link.springer.com/article/10.1007/s10009-023-00705-2

Kourtesis, D., Bratanis, K., Bibikas, D., & Paraskakis, I. (2012). Software co-development in the era of cloud application platforms and ecosystems: The case of cast. In *Ifip advances in information and communication technology* (Vol. 380 AICT, pp. 196–204).

Kratzke, N. (2018, 8). A brief history of cloud application architectures. *Applied Sciences 2018, Vol. 8, Page 1368*, *8*, 1368. https://www.mdpi.com/2076-3417/8/8/1368/htmhttps://www.mdpi.com/2076-3417/8/8/1368

Kratzke, N., & Peinl, R. (2016, 9). Clouns-a cloud-native application reference model for enterprise architects. In *Proceedings - ieee international enterprise distributed object computing workshop, edocw* (Vol. 2016-Septe, pp. 198–207). IEEE. http://ieeexplore.ieee.org/document/7584353/ (Good refrence story about migrationfrom one cloud provider to another. Provides Cloud NAtiver applications stack. Bring a pararel between OSI layer abd Cloud Native application stack)

Kratzke, N., & Quint, P.-C. (2017, 4). Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. *Journal of Systems and Software*, *126*, 1–16. http://dx.doi.org/10.1016/j.jss.2017.01.001http://linkinghub.elsevier.com/retrieve/pii/S0164121217300018

kubernetes.io. (2022). *Horizontal pod autoscaling.* https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

kubernetes.io. (2024). *Cluster architecture | kubernetes.* https://kubernetes.io/docs/concepts/architecture/

Kumar, D., & Gondhi, N. K. (2018). A qos-based reactive auto scaler for cloud environment. In A. Gupta & K. Vijay (Eds.), (pp. 1–6).

Leymann, F., Fehling, C., Wagner, S., & Wettinger, J. (2016). Native cloud applications : Why virtual machines , images and containers miss the point ! In (pp. 7–15). SciTePress. http://closer.scitevents.org

Lichtenthäler, R., & Wirtz, G. (2024, 3). Formulating a quality model for cloud-native software architectures: conceptual and methodological considerations. *Cluster Computing*, 1–17. https://link.springer.com/article/10.1007/s10586-024-04343-4

Likosar, B. (2023, 3). *Getting the most from kubernetes autoscaling - the new stack.* https://thenewstack.io/getting-the-most-from-kubernetes-autoscaling/

Lorido-Botran, T., Miguel-Alonso, J., & Lozano, J. A. (2014). A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, *12*, 559–592.

Lorido-Botrán, T. (2012). Auto-scaling techniques for elastic applications in cloud environments. *Department of Computer ...*, 1–44. http://www.chinacloud.cn/upload/2012-11/12112312255214.pdf

Makroo, A., & Dahiya, D. (2016, 5). A systematic approach to deal with noisy neighbour in cloud infrastructure. *Indian Journal of Science and Technology*, *9*.

Martin, R. C. (2017). *Clean architecture: A craftsman's guide to software structure and design*. Pearson. https://www.oreilly.com/library/view/clean-architecture-a/9780134494272/ch7.xhtml

Microsoft. (2024a). *Azure kubernetes service (aks) documentation.* https://learn.microsoft.com/en-us/azure/aks/

Microsoft. (2024b). *Concepts - scale applications in azure kubernetes services (aks) - azure kubernetes service.* https://learn.microsoft.com/en-us/azure/aks/concepts-scale

Microsoft. (2024c). *Load balancer.* https://learn.microsoft.com/en-us/azure/load-balancer/

Mirhosseini, A., Elnikety, S., & Wenisch, T. F. (2021). Parslo: A gradient descent-based approach for near-optimal partial slo allotment in microservices. In C. Curino, G. Koutrika, & N. Ravi (Eds.), (pp. 442–457). ACM. https://doi.org/10.1145/3472883.3486985

Mitchell, B. S. (2023). Cloud native software engineering. *Preprint*.

Mondal, S. K., Wu, X., Kabir, H. M. D., Dai, H. N., Ni, K., Yuan, H., & Wang, T. (2023, 6). Toward optimal load prediction and customizable autoscaling scheme for kubernetes. *Mathematics 2023, Vol. 11, Page 2675*, *11*, 2675. https://www.mdpi.com/2227-7390/11/12/2675/htmhttps://www.mdpi.com/2227-7390/11/12/2675

Nguyen, T. T., Yeom, Y. J., Kim, T., Park, D. H., & Kim, S. (2020, 8). Horizontal pod autoscaling in kubernetes for elastic container orchestration. *Sensors (Basel, Switzerland)*, *20*, 1–18. /pmc/articles/PMC7471989/

Nichols, J. A., Chan, H. W. H., & Baker, M. A. (2019, 2). Machine learning: applications of artificial intelligence to imaging and diagnosis. *Biophysical Reviews*, *11*, 111. /pmc/articles/PMC6381354//pmc/articles/PMC6381354/?report=abstracthttps://www.ncbi.nlm.nih.gov/pmc/articles/PMC6381354/

Nikravesh, A. Y., Ajila, S. A., & Lung, C.-H. (2017, 12). An autonomic prediction suite for cloud resource provisioning. *Journal of Cloud Computing*, *6*, 3. http://journalofcloudcomputing.springeropen.com/articles/10.1186/s13677-017-0073-4

Odun-Ayo, I., Goddy-Worlu, R., Ajayi, L., Edosomwan, B., & Okezie, F. (2019, 12). A systematic mapping study of cloud-native application design and engineering. *Journal of Physics: Conference Series*, *1378*, 032092. https://iopscience.iop.org/article/10.1088/1742-6596/1378/3/032092

Papadopoulos, A. V., Versluis, L., Bauer, A., Herbst, N., Kistowski, J. V., Ali-Eldin, A., … Iosup, A. (2021, 8). Methodological principles for reproducible performance evaluation in cloud computing. *IEEE Transactions on Software Engineering*, *47*, 1528–1543. (Measurement)

Peinl, R., Holzschuher, F., & Pfitzer, F. (2016, 6). Docker cluster management for the cloud - survey results and own solution. *Journal of Grid Computing*, *14*, 265–282. http://link.springer.com/10.1007/s10723-016-9366-y (Good for future reserch)

Pivotal. (2017). *Cloud-native.* Pivotal Software, Inc. https://pivotal.io/cloud-native

Podolskiy, V., Jindal, A., & Gerndt, M. (2019, 6). Multilayered autoscaling performance evaluation: Can virtual machines and containers co-scale? *International Journal of Applied Mathematics and Computer Science*, *29*, 227–244.

Poojitha, S. A., & Ravindranath, K. (2025). Quality aware batch scheduling of containers in cloud computing environment. *International Journal of Information Technology (Singapore).*

Pozdniakova, O. (2023). *Github - aurimasch/autoscaling.* https://github.com/aurimasch/autoscaling

Pozdniakova, O. (2024). *olesiapoz/sata: The sla-adaptive threshold adjustment algorithm for kubernetes horizontal autoscaler.* https://github.com/olesiapoz/sata

Pozdniakova, O., Cholomskis, A., & Mažeika, D. (2023, 7). Self-adaptive autoscaling algorithm for sla-sensitive applications running on the kubernetes clusters. *Cluster Computing*, 1–28. https://link.springer.com/article/10.1007/s10586-023-04082-y

Pozdniakova, O., & Mažeika, D. (2017a, 4). A cloud software isolation and cross-platform portability methods. In (pp. 1–6). IEEE. http://ieeexplore.ieee.org/document/7950315/

Pozdniakova, O., & Mažeika, D. (2017b). Systematic literature review of the cloud-ready software architecture. *Baltic J. Modern Computing*, *5*, 124–135. http://dx.

Pozdniakova, O., Mažeika, D., & Cholomskis, A. (2018a). Adaptive resource provisioning and auto-scaling for cloud native software. In (Vol. 920, pp. 113–129). Springer, Cham. https://link.springer.com/chapter/10.1007/978-3-319-99972-2_9

Pozdniakova, O., Mažeika, D., & Cholomskis, A. (2018b). Adaptive resource provisioning and auto-scaling for cloud native software. *Communications in Computer and Information Science*, *920*, 113–129. https://link.springer.com/chapter/10.1007/978-3-319-99972-2_9

Pozdniakova, O., Mažeika, D., & Cholomskis, A. (2024, 4). Sla-adaptive threshold adjustment for a kubernetes horizontal pod autoscaler. *Electronics (Switzerland)*, *13*.

Pramesti, A. A., & Kistijantoro, A. I. (2022). Autoscaling based on response time prediction for microservice application in kubernetes. *2022 9th International Conference on Advanced Informatics: Concepts, Theory and Applications, ICAICTA 2022*.

Prometheus. (2025). *Prometheus - monitoring system & time series database.* https://prometheus.io/

Pusztai, T., Morichetta, A., Pujol, V. C., Dustdar, S., Nastic, S., Ding, X., … Xiong, Y. (2021). Slo script: A novel language for implementing complex cloud-native elasticity-driven slos. *Proceedings - 2021 IEEE International Conference on Web Services, ICWS 2021*, 21–31.

Qian, H., Wen, Q., Sun, L., Gu, J., Niu, Q., & Tang, Z. (2022). Robustscaler: Qos-aware autoscaling for complex workloads. In F. C. et al. Harris (Ed.), *Proceedings - international conference on data engineering* (Vol. 2022-May, pp. 2762–2775).

Qu, C., Calheiros, R. N., & Buyya, R. (2018). Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys*, *51*, 33. https://doi.org/10.1145/3148149

Quach, C. (2020). *Setting slos: a step-by-step guide.* https://cloud.google.com/blog/products/management-tools/practical-guide-to-setting-slos

Raj, P., Vanga, S., & Chaudhary, A. (2022, 10). Kubernetes architecture, best practices, and patterns. *Cloud-Native Computing*, 49–70.

Raudys, A., Lenčiauskas, V., & Malčius, E. (2013). Moving averages for financial data smoothing. *Communications in Computer and Information Science*, *403*, 34–45.

Retter, R., & Fehling, C. (2013). *Applying architectural patterns for the cloud: Lessons learned during pattern mining and application.* http://www.sei.cmu.edu/library/assets/presentations/retter-saturn2013.pdf

Roussev, V., Ahmed, I., Barreto, A., McCulley, S., & Shanmughan, V. (2016, 9). Cloud forensics–tool development studies
future outlook. *Digital Investigation*, *18*, 79–95. http://linkinghub.elsevier.com/retrieve/pii/S1742287616300536

Ruiz, L. M., Pueyo, P. P., Mateo-Fornes, J., Mayoral, J. V., & Tehas, F. S. (2022). Autoscaling pods on an on-premise kubernetes infrastructure qos-aware. *IEEE Access*, *10*, 33083–33094. https://github.com/gcd-cloud-

Rzadca, K., Findeisen, P., Swiderski, J., Zych, P., Broniek, P., Kusmierek, J., … Wilkes, J. (2020, 4). Autopilot: Workload autoscaling at google. *Proceedings of the 15th European Conference on Computer Systems, EuroSys 2020.* https://dl.acm.org/doi/10.1145/3342195.3387524

Sahal, R., Khafagy, M. H., & Omara, F. A. (2016). A survey on sla management for cloud computing and cloud-hosted big data analytic applications. *International Journal of Database Theory and Application*, *9*, 107–118. http://dx.doi.org/10.14257/ijdta.2016.9.4.10

Schroeder, B., Wierman, A., & Harchol-Balter, M. (2006, 5). Open versus closed: A cautionary tale. USENIX Association. /articles/journal_contribution/Open_Versus_Closed_A_Cautionary_Tale/6608078/1

Scopus. (2024). *Scopus - document search results.* https://www.scopus.com/

SEC.gov. (2025). *Sec.gov | edgar log file data sets.* https://www.sec.gov/about/data/edgar-log-file-data-sets

Sekhi, I. (2023, 12). Selecting the sla guarantee by evaluating the qos availability. *Multidiszciplináris Tudományok*, *13*, 80–102.

Shafi, N., Abdullah, M., Iqbal, W., Erradi, A., & Bukhari, F. (2024, 1). Cdascaler: a cost-effective dynamic autoscaling approach for containerized microservices. *Cluster Computing*, 1–21. https://link.springer.com/article/10.1007/s10586-023-04228-y

Sidekerskiene, T., & Damasevicius, R. (2016). Reconstruction of missing data in synthetic time series using emd. In *Proceedings of the international conference for young researchers in informatics, mathematics and engineering* (Vol. Vol-1712, pp. 7–17). https://ceur-ws.org/Vol-1712/p02.pdf

Sodhi, B., & Prabhakar, T. V. (2011, 1). Application architecture considerations for cloud platforms. In *2011 third international conference on communication systems and networks (comsnets 2011)* (pp. 1–4). IEEE. http://ieeexplore.ieee.org/document/5716417/

Splunk. (2024). *Observability | new in splunk observability cloud ... - splunk community.* https://community.splunk.com/t5/Product-News-Announcements/Observability-New-In-Splunk-Observability-Cloud-Native-SLO/ba-p/676339

Stine, M. (2015). *Migrating to cloud-native application architectures.* O'Reilly Media, Inc. http://content.wkhealth.com/linkback/openurl?sid=WKPTLP:landingpage&an=00004045-201405001-00009

Sun, Y., Meng, L., & Song, Y. (2019, 6). Autoscale: Adaptive qos-aware container-based cloud applications scheduling framework. *KSII Transactions on Internet and Information Systems*, *13*, 2824–2837.

Taherizadeh, S., & Grobelnik, M. (2020, 2). Key influencing factors of the kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications. *Advances in Engineering Software*, *140*.

Taherizadeh, S., Jones, A. C., Taylor, I., Zhao, Z., & Stankovski, V. (2018, 2). Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review. *Journal of Systems and Software*, *136*, 19–38. https://www.sciencedirect.com/science/article/pii/S016412121730256X

Taherizadeh, S., & Stankovski, V. (2019, 2). Dynamic multi-level auto-scaling rules for containerized applications. *The Computer Journal*, *62*, 174–197. https://academic.oup.com/comjnl/article/62/2/174/4993728

Toffetti, G., Brunner, S., Blöchlinger, M., Spillner, J., & Bohnert, T. M. (2016, 9). Self-managing cloud-native applications: Design, implementation, and experience. *Future Generation Computer Systems*. http://www.sciencedirect.com/science/article/pii/S0167739X16302977http://linkinghub.elsevier.com/retrieve/pii/S0167739X16302977

Toka, L., Dobreff, G., Fodor, B., & Sonkoly, B. (2021, 3). Machine learning-based scaling management for kubernetes edge clusters. *IEEE Transactions on Network and Service Management*, *18*, 958–972.

Tonini, F., Natalino, C., Temesgene, D. A., Ghebretensaé, Z., Wosinska, L., & Monti, P. (2023). A service-aware autoscaling strategy for container orchestration platforms with soft resource isolation. *2023 Joint European Conference on Networks and Communications and 6G Summit, EuCNC/6G Summit 2023*, 454–459.

Vazquez, C., Krishnan, R., & John, E. (2015). Time series forecasting of cloud data center workloads for dynamic resource provisioning. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, *6*, 36–53. http://isyou.info/jowua/papers/jowua-v6n3-5.pdf

Verreydt, S., Beni, E. H., Truyen, E., Lagaisse, B., & Joosen, W. (2019, 12). Leveraging kubernetes for adaptive and cost-efficient resource management. In P. Heidari (Ed.), (pp. 37–42). ACM Press. http://dl.acm.org/citation.cfm?doid=3366615.3368357

Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., … Lang, M. (2017, 6). Cost comparison of running web applications in the cloud using monolithic, microservice, and aws lambda architectures. *Service Oriented Computing and Applications*, *11*, 233–247. http://link.springer.com/10.1007/s11761-017-0208-y

VMware. (2016). *Cloud-native applications: Vmware.* https://www.vmware.com/solutions/cloudnative.html

VMware. (2024). *Use case 3: Use actionable slos to influence autoscaling.* https://docs.vmware.com/en/VMware-Tanzu-Service-Mesh/services/slos-with-tsm/GUID-1B9A2D61-D264-44FB-8A06-40277AD42A8E.html

Weinman, J. (2016). Migrating to - or away from - the public cloud. *IEEE Cloud Computing*, *3*, 6–10.

Wen, L., Xu, M., Gill, S. S., Hilman, M. H., Srirama, S. N., Ye, K., & Xu, C. (2023, 5). Statuscale: Status-aware and elastic scaling strategy for microservice applications. *ACM Transactions on Autonomous and Adaptive Systems*. https://dl.acm.org/doi/10.1145/3686253

Wilder, B. (2012). *Cloud architeture patterns* (First Edit ed.; R. Roumeliotis, Ed.). O'Reilly Media, Inc.

Wu, Q., Yu, J., Lu, L., Qian, S., & Xue, G. (2019, 12). Dynamically adjusting scale of a kubernetes cluster under qos guarantee. *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, *2019-December*, 193–200.

Xu, Y., Qiao, K., Wang, C., & Zhu, L. (2022, 10). Lp-hpa:load predict-horizontal pod autoscaler for container elastic scaling. *ACM International Conference Proceeding Series*, 591–595. https://dl.acm.org/doi/10.1145/3569966.3570115

Ye, T., Guangtao, X., Shiyou, Q., & Minglu, L. (2017, 11). An auto-scaling framework for containerized elastic applications. *Proceedings - 2017 3rd International Conference on Big Data Computing and Communications, BigCom 2017*, 422–430.

Zhang, F., Tang, X., Li, X., Khan, S. U., & Li, Z. (2019, 9). Quantifying cloud elasticity with container-based autoscaling. *Future Generation Computer Systems*, *98*, 672–681. (Measurement)

Zimmermann, O. (2017, 2). Architectural refactoring for the cloud: a decision-centric view on cloud migration. *Computing*, *99*, 129–145. http://link.springer.com/10.1007/s00607-016-0520-y (IDAL meantions)

List of Scientific Publications by the Author on the Topic of the Dissertation

Papers in the Reviewed Scientific Journals

Pozdniakova, O., Mažeika, D., & Cholomskis, A. (2024). SLA-Adaptive Threshold Adjustment for a Kubernetes Horizontal Pod Autoscaler. *Electronics*, 13(7), 1–28. https://doi.org/10.3390/electronics13071242

Pozdniakova, O., Cholomskis, A., & Mažeika, D. (2023). Self-Adaptive Autoscaling Algorithm for SLA-Sensitive Applications Running on the Kubernetes Clusters. *Cluster computing*, 27(3), 2399–2426. https://doi.org/10.1007/s10586-023-04082-y

Pozdniakova, O., & Mažeika, D. (2017). Systematic Literature Review of the Cloud-Ready Software Architecture, *Baltic Journal of Modern Computing*, 5(1), 124–135. https://doi.org/10.22364/bjmc.2017.5.1.08

Papers in Other Editions

Pozdniakova, O., & Mažeika, D. (2017). A Cloud Software Isolation and Cross - Platform Portability Methods . In *Proceedings of the 2017 Open Conference of Electrical, Electronic and Information Sciences – eStream*(pp. 1–6) Vilnius, Lithuania. https://doi.org/10.1109/eStream.2017.7950315

Pozdniakova, O., Mažeika, D., & Cholomskis, A. (2018). Adaptive Resource Provisioning and Auto-Scaling for Cloud Native Software . In *Proceedings of Communications in computer and information science. Information and Software Technologies (ICIST 2018) 24th International Conference* (pp. 113–129), Vilnius, Lithuania. https://doi.org/10.1007/978-3-319-99972-2_9

Cholomskis, A., Pozdniakova, O., & Mažeika, D. (2018). Cloud Software Performance Metrics Collection and Aggregation for Auto-scaling Module. In *Proceedings of Communications in Computer and Information Science. Information and Software Technologies (ICIST 2018) 24th International Conference* (pp. 130–138), Vilnius, Lithuania. https://doi.org/10.1007/978-3-319-99972-2_10

Pozdniakova, O., & Mažeika, D. (2023). Performance-Based SLO Recovery for Containerized Applications. In *Presentation abstract of DAMSS 2023: 14th Conference on data analysis methods for software systems* (pp. 72–73), Druskininkai, Lithuania. https://doi.org/10.15388/DAMSS.14.2023

Summary in Lithuanian

Įvadas

Problemos formulavimas

Pastaraisiais metais išaugęs programų konteinerizavimo ir mikroservisų architektūros populiarumas paskatino naujos paradigmos, žinomos kaip debesų kompiuterijos kilmės programos (angl. *Cloud Native Applications*, toliau CNA), atsiradimą. CNA dažnai susideda iš kelių keičiamo dydžio ir laisvai susietų paslaugų, kurios veikia kaip konteinerizuotų programų egzemplioriai. Labai svarbu šiuos egzempliorius paleisti tinkamu laiku ir tinkamu kiekiu, kad būtų įvykdyti susitarime dėl paslaugų teikimo lygio (angl. *Service Level Agreement*, toliau SLA) nustatyti našumo reikalavimai. Konteinerių orkestravimo platformos yra specialiai sukurtos tam, kad supaprastintų didesnio masto konteinerizuotų programų veikimą, kuriose automatinio masteliavimo komponentas (angl. *autoscaler*) atlieka svarbų vaidmenį užtikrindamas taikomosioms programoms reikiamų išteklių gavimą.

Automatinio masteliavimo priemonės turi spręsti terminų ir tinkamo resursų kiekio užtikrinimo uždavinius. Per anksti arba per daug suteiktų resursų didina išlaidas, o delsimas gali pabloginti paslaugos kokybę ir sudaro sąlygas pažeisti susitarimą dėl paslaugos teikimo lygio. Todėl dauguma automatinio masteliavimo algoritmų siekia rasti balansą tarp paslaugos kokybės užtikrinimo ir efektyvaus išteklių valdymo. Negalima nuvertinti infrastruktūros sąnaudų mažinimo svarbos versle. Tačiau teikiamų paslaugų kokybė galutiniams vartotojams gali daryti tiesioginę įtaką visai verslo sėkmei (Arapakis et al., 2021; Sekhi, 2023). Nesilaikant susitarimo dėl paslaugų teikimo lygio, gali būti prarastas verslas arba skirtos baudos. Todėl automatinio masteliavimo sprendimai, kurie yra orientuoti į teikiamos paslaugos kokybės užtikrinimą, turi efektyviai valdyti išteklius, kartu išlaikydami pageidaujamą paslaugų kokybę (Al-Dhuraibi et al., 2018; Amiri & Mohammad-Khanli, 2017). Tam reikalingi nuolatinio paslaugų kokybės, sistemos našumo, žinių apie SLA vykdymo stebėjimo mechanizmai, kurių įgyvendinimas leistų tinkamai reaguoti į nukrypimus nuo apibrėžtų paslaugų teikimo lygio rodiklių ir tikslų.

Darbo aktualumas

Debesų kilmės kompiuterijos bei konteinerizuotų taikomųjų programų naudojimas auga, vis daugiau esamų programų yra perkeliama į debesų kompiuterijos platformas bei mikroservisų architektūrą (Kazanavičius et al., 2022), taip sukuriant naują debesųkompiuterijos kilmės paradigmą. Debesų kompiuterijos kilmės sprendimų populiarumas nulėmė *Cloud Native Computing Foundation* (toliau CNCF) projekto atsiradimą. Šio projekto tikslas – skatinti diegimą technologijų, suteikiančių organizacijoms galimybę kurti ir naudoti masteliuojamas taikomąsias programas šiuolaikinėje dinamiškoje aplinkoje, įskaitant viešus, privačius ir hibridinius debesis. CNCF sukurtas debesų kompiuterijos kilmės technologijų landšaftas apima daugiau nei 40 atvirojo kodo projektų ir patentuotų produktų, priskirtų planavimo ir orkestravimo (angl. *scheduling and orchestration*) kategorijai (CNCF, 2024a). Kasmet joje vis daugėja naujų narių ir sprendimų.

Kubernetes taip pat tampa kasmet vis labiau įsitvirtinančia konteinerių orkestravimo platforma (CNCF, 2024c; Datadog, 2024). Tokios kompanijos kaip *NetApp*, *Google* ir *Datadog* kuria automatinio masteliavimo produktus, o akademinė bendruomenė aktyviai ieško automatinio masteliavimo sprendimų, skirtų debesų kompiuterijos kilmės programoms. *Semantic Scholar*, *Web Of Science* ir *Scopus* duomenų bazėse galima rasti tūkstančius straipsnių, skirtų debesų kompiuterijai ir automatinio masteliavimo sprendimams. Nepaisant daugiau nei dešimtmetį trukusių šios srities mokslinių tyrimų, susidomėjimas ir toliau auga, o veiksmingo automatinio masteliavimo problema vis dar aktuali.

Pirmuosiuose automatinio masteliavimo sprendimuose daugiausia dėmesio buvo skirta efektyviam kaštų panaudojimui. Tačiau vis dažniau susitelkiama į susitarimų dėl paslaugos kokybės įvykdymo aspektą, nes vartotojai ir klientai pirmenybę teikia paslaugų našumui ir kokybei (Arapakis et al., 2021; Pusztai et al., 2021; Sekhi, 2023). Per pastaruosius penkerius metus tokie gamintojai kaip *VMware* (VMware, 2024), *Google* (Rzadca et al., 2020) ir *Splunk* (Splunk, 2024), taip pat akademinė bendruomenė (Poojitha & Ravindranath, 2025; Pusztai et al., 2021; Qian et al., 2022; Ruiz et al., 2022; Tonini et al., 2023; Wen et al., 2023) daugiau dėmesio skiria debesų kompiuterijos stebėjimo spendimams, orientuotiems į žinių apie SLA surinkimą (angl. *SLA-awarness*).

Tyrimų objektas

Darbo tyrimų objektas – žiniomis apie SLA pagrįsti automatinio masteliavimo algoritmai, skirti debesų kompiuterijos kilmės taikomosioms programoms.

Darbo tikslas

Patobulinti paslaugos lygio tikslų įvykdymą, naudojant taisyklėmis pagrįstus automatinio masteliavimo algoritmus, kai tikslai turi užtikrinti debesų kompiuterijos kilmės taikomųjų programų našumo reikalavimų įvykdymą.

Darbo uždaviniai

Darbo tikslui pasiekti sprendžiami šie uždaviniai:

1. Atlikti mokslinės literatūros apžvalgą apie dabartinę automatinio masteliavimo algoritmų ir metodų, kuriais siekiama užtikrinti debesų kompiuterijos kilmės taikomosios programos veikimo atitikimą paslaugos teikimo lygio tikslams, būklę.
2. Sukurti SLA žiniomis pagrįstus automatinio masteliavimo metodus, skirtus debesų kilmės taikomųjų programų automatinio masteliavimo sprendimams, orientuotiems į SLA įvykdymo aspektą.
3. Pasiūlyti metodus, kurie vertina automatinio masteliavimo metodų veiksmingumą užtikrinant SLA įvykdymą.
4. Įvertinti siūlomų automatinio masteliavimo metodų veiksmingumą ir efektyvumą įvairiomis darbo krūvio sąlygomis.
5. Atlikti palyginamąją siūlomų automatinio masteliavimo metodų analizę ir palyginti juos su plačiai naudojamais automatinio masteliavimo sprendimais.

Tyrimų metodika

1. Atlikta *analitinė literatūros apžvalga* apie debesų kompiuterijos kilmės tendencijas ir esamus automatinio masteliavimo algoritmus, kurie užtikrina SLA debesų kompiuterijos kilmės atitiktį taikomosioms programoms. Buvo įvertintos stipriosios ir silpnosios pusės, pagrindiniai SLA įvykdymui įtaką darantys veiksniai, nustatytos esamos automatinio masteliavimo sprendimų spragos.
2. Atlikta *lyginamoji analizė,* norint įvertinti analizuotų metodų pranašumus ir trūkumus.
3. Taikytas *statistinės tiriamosios analizės metodas,* siekiant suprasti, kaip centrinio procesoriaus (angl. *Central processing unit*, toliau CPU) apkrovos slenksčio reikšmės pasirinkimas daro įtaką našumu pagrįsto SLA vykdymui.
4. Taikytas *kiekybinis tyrimo metodas* – atlikti *eksperimentai* su skirtingomis apkrovomis. Eksperimentai atlikti siekiant įvertinti išanalizuotų taisyklėmis pagrįstų automatinio masteliavimo sprendimų efektyvumą, tenkinant SLA našumo reikalavimus. Kuriant kelių tipų darbo krūvius buvo naudojami *Gatling* ir *JMeter* krūvio generavimo įrankiai. Eksperimentai atlikti *Azure* viešoje debesų kompiuterijos platformoje, naudojant *Azure Kubernetes Service* (AKS). Automatinio masteliavimo algoritmų prototipai parašyti *Java* programavimo kalba. Surinkti eksperimentų duomenys buvo išanalizuoti, siekiant įvertinti sprendimus pagal iš anksto apibrėžtus vertinimo kriterijus.

Darbo mokslinis naujumas

- Pasiūlytas metodas, skirtas dinamiškai koreguoti slenksčius automatinio masteliavimo sprendimuose, pritaikytuose konteinerizuotoms debesų kompiuterijos programoms, pagerina našumo atitiktį nustatytiems SLA reikalavimams. Naudojant šį metodą, slenksčiais pagrįsti automatinio masteliavimo sprendimai savarankiškai prisitaiko prie platformos ir programų našumo pokyčių, palaiko ir atkuria paslaugos kokybę iki nustatyto SLA lygio, aptikus degradacijos tendencijas. Sprendimui įgyvendinti taikomi tiriamieji duomenų analizės metodai ir slankusis vidurkio glodinimas, siekiant nustatyti CPU panaudojimo slenkstį.

- Pasiūlyti algoritmai pagerina SLA įvykdymą taisyklėmis pagrįstuose automatinio masteliavimo sprendimuose. Jie sukurti siekiant sumažinti mašininio mokymosi algoritmo įvestą sudėtingumą, kartu pagerinant nustatytų paslaugų lygio našumo susitarimų įvykdymą.
- Pasiūlyta SLA atkūrimo metodika pagerina nustatytų našumo paslaugų lygio tikslų laikymąsi.

Darbo rezultatų praktinė reikšmė

Siūlomi naujieji metodai svarbūs tiek teoriniu, tiek praktiniu požiūriu, siekiant užtikrinti, kad taikomųjų programų našumas atitiktų SLA reikalavimus. Šiais sprendimais taip pat siekiama optimizuoti esamų automatinio mastelio keitimo sprendimų našumą. Kadangi debesų kompiuterijos kilmės sprendimai tampa vis populiaresni, didėja poreikis naudoti sudėtingesnius, patikimesnius ir SLA efektyvius automatinio masteliavimo sprendimus.

Daugelyje plačiai naudojamų automatinio masteliavimo sprendimų resursų kiekio ir masteliavymo veiksmų savalaikiškumo nustatymui yra naudojami resursų apkrovos slenksčiai. Tačiau slenksčių, leidžiančių pasiekti našumu pagrįstus paslaugos teikimo lygio tikslus (angl. *Service Level Objectives*, toliau SLO), nustatymas šiuo metu yra sudėtingas ir nuo klaidų neapsaugotas procesas. Taip yra dėl to, kad statinės slenksčio reikšmės paprastai yra nustatomos rankiniu būdu. Siūlomas dinaminis slenksčio nustatymo metodas ne tik sutaupo laiko derinant automatinio masteliavimo sprendimo našumą, bet ir padeda pamatus tolesniems efektyvumo optimizavimo tyrimams.

Ginamieji teiginiai

1. Scenarijuose, kai papildomų išteklių paskirstymas gali žymiai pagerinti paslaugų veikimą ir sumažinti SLO pažeidimo tikimybę, papildomų išteklių pridėjimas leidžia atkurti neįvykdyto paslaugos teikimo lygio tikslo būseną į įvykdyto tikslo būseną. Toks paslaugos lygio sutarties įvykdymo būsenos atkūrimo būdas gali būti taikomas, siekiant pagerinti į našumą orientuotų paslaugos reikalavimų vykdymą, kartu su tradiciškai naudojamais SLA pažeidimų vengimo mechanizmais taisyklėmis pagrįstuose debesų kompiuterijos programų automatinio masteliavimo valdikliuose.
2. Dinamiškai keičiant taisyklėmis pagrįstų automatinio masteliavimo sprendimų panaudojimo slenksčio vertes galima pagerinti SLA įvykdymą, kai panaudojimo slenkstis yra labiausiai SLA įvykdymui įtaką darantis veiksnys.
3. Pratęsus SLA įvykdymo būsenos stebėjimo laikotarpį, pagerėja debesų kompiuterijos automatinio masteliavimo sprendimų gebėjimas užtikrinti nustatytus paslaugos lygio našumo reikalavimus, palyginti su automatinio masteliavimo sprendimais, kurie remiasi tik momentiniais paslaugų lygio matavimais priimant masteliavimo sprendimus.

Darbo rezultatų aprobavimas

Disertacijos tema paskelbta 2 žurnaluose, įtrauktuose į *Web of Science* duomenų bazę ir turinčiuose citavimo rodiklį, 3 mokslinių konferencijų pranešimų rinkiniuose, o 1 – konferencijos pristatymo santraukoje. Disertacijoje atliktų tyrimų rezultatai buvo pristatyti 5 mokslinėse konferencijose Lietuvoje ir užsienyje:

- DAMSS 2016: 8th Data Analysis Methods for Software Systems, 2016 gruodžio 1–3 d., Druskininkuose, Lietuvoje.
- eStream 2017: Open International Conference of Electrical, Electronic and Information Sciences, 2017 Balandžio 27 d., Vilniuje, Lietuvoje.
- ICIST 2018: 24th International Conference on Information and Software Technologies, 2018 Spalio 4–6 d., Vilniuje, Lietuvoje.
- DAMSS 2023: 14th Data Analysis Methods for Software Systems, 2023 lapkričio 30 d. –Gruodžio 2 d., Druskininkuose, Lietuvoje.
- PCDS 2024: 1st International Symposium on Parallel Computing and Distributed Systems, 2024 Rugsėjo 21–22 d., Singapūre, Azijoje.

Disertacijos struktūra

Disertaciją sudaro įvadas, trys pagrindiniai skyriai, bendros išvados, literatūros sąrašas, autoriaus publikacijų disertacijos tema sąrašas ir santrauka lietuvių kalba. Disertacijos apimtis (be priedų) – 167 puslapiai, 44 lygtys, 29 iliustracijos ir 24 lentelės.

1. Automatinio masteliavimo sprendimų, skirtų susitarimo dėl paslaugos teikimo lygio užtikrinimui debesų kompiuterijos kilmės programų sistemose, literatūros apžvalga

Šiame skyriuje pateikiama debesų kompiuterijos kilmės tendencijų apžvalga bei debesų kompiuterijos kilmės taikomųjų programų apibrėžtis. Remiantis atlikta literatūros analize, galima daryti išvadą, kad debesų kompiuterijos kilmės taikomoji programa turi tokias savybes: ji kuriama kaip paskirstytoji sistema su mažą sankibą turinčiais komponentais, skirtais išplečiamumui ir galinčiais veikti automatizuotoje bei elastingoje platformoje. Šiomis savybėmis siekiama spręsti problemas, kylančias naudojant programas dinamiškai besikeičiančioje platformoje.

Konteineriai dažnai naudojami kompiuterijos kilmės taikomojų programų paleidimui (Deng et al., 2024; Kosińska et al., 2024; Kratzke, 2018; Kratzke & Quint, 2017; Villamizar et al., 2017). Automatizuotas konteinerių gyvavimo ciklo priežiūros įrankis – konteinerių orkestratorius – palengvina konteinerizuotų programų sistemų priežiūrą. Orchestratoriaus automatinio masteliavimo komponentas sprendžia resursų teikimo terminų ir kiekio nustatymo uždavinius, kad paslaugos kokybė atitiktų reikalavimus. Tam, kad automatinio masteliavimo sprendimas teiktų resursus pagal SLA nustatytus reikalavimus, svarbu tinkamai parinkti sistemos bei paslaugos stebėjimo parametrus ir automatinio masteliavimo būdą. Paslaugos kokybei stebėti naudojami paslaugos veikimo lygio indikatoriai (angl. *Service Level Indicators*, toliau SLI), kurie naudojami paslaugos teikimo lygio tikslų (SLO) matavimui. SLO aprašomi susitarime dėl paslaugos teikimo lygio (SLA). Informacijos apie paslaugos sutarties įvykdymo statusą įtraukymas į masteliavimo sprendimo priėmimo procesą turėtų pagerinti automatinio masteliavimo sprendimų veiksmingumą, užtikrinant sistemos našumą pagal SLA, tačiau šis metodas retai sutinkamas literatūroje. Informacija apie SLA leidžia automatinio masteliavo sprendimui imtis atitinkamų veiksmų. Literatūros apžvalgos ir analizės metu tokių sprendimų nebuvo aptikta.

Kubernetes Horizontal Pod Autoscaler (HPA) yra dažniausiai literatūroje bei praktikoje sutinkamas automatinio masteliavimo sprendimas, kuris taikomas debesų kompiuterijos kilmės taikomųjų programų resursų valdymui. Reikiamo resursų kiekio skaičiavimui šis sprendimas naudoja statinius resursų panaudojimo slenksčius. Statiniais slenksčiais pagrįsti masteliavimo sprendimai yra inertiški ir jiems sunku parinkti tinkamą slenkstį (Qu et al., 2018) ir yra didelė tikimybė, kad SLA bus pažeistas arba bus panaudotas didesnis nei pakankamas resursų kiekis. Norint palengvinti HPA taikymą praktikoje, akademikai siūlo skirtingus HPA parametrų automatinio nustatymo sprendimus (Augustyn et al., 2024; Huo et al., 2023, 2022; Khaleq & Ra, 2021). Literatūros analizės metu nebuvo identifikuota automatinio slenksčių vertės nustatymų sprendimų, kurie būtų ištestuoti debesų kompiuterijos platformoje bei netaikytų mašininio apmokymo algoritmų.

Literatūros apžvalga atskleidė, kad šiame darbe išanalizuotų tyrimų autoriai naudojo įvairius kriterijus automatinio masteliavo sprendimų efektyvumui bei veiksmingumui įvertinti. Tačiau nė vienas iš sprendimų nevertina efektyvumo užtikrinant SLA įvykdymą.

Apibendrinant literatūros apžvalgą galima teigti, kad nėra vieno automatinio malsteliavimo metodo, kuris veiksmingai išspręstų visas SLA įvykdymo užtikrinimo problemas vienu metu. Taisyklėmis pagrįsti automatinio masteliavimo sprendimai yra dažniausiai pasitaikantys praktikoje, nors, palyginti su mašinino apmokymo sprendimais, jie turi silpnų vietų, ypač paslaugos teikimo lygio užtikrinimo srityje. HPA yra dažniausiai praktikoje bei akademiniame pasaulyje taikomas automatinio masteliavimo sprendimas. Jo populiarumas lėmė atskiros HPA tyrimo šakos, skirtos konfigūracijos parametrų automatizavimui, atsiradimą.

2. Prie susitarimo dėl paslaugų kokybės prisitaikančio automatinio masteliavimo algoritmo kūrimas

Skyriuje aptariami du taisyklėmis pagrįsti automatinio masteliavimo sprendimai, kuriuose naudojami automatinio masteliavimo algoritmai. Sukurti algoritmai siekia užtikrinti paslaugos teikimo lygio sutarties (SLA) vykdymą. Pirmas siūlomas sprendimas yra automatinis masteliavimas, prisitaikantis prie SLA reikalavimų (angl. *SLA Adaptive Autoscaling Algorithm*, toliau SAA), skirtas tradicinių taisyklėmis pagrįstų sistemų iššūkiams įveikti. Antras sprendimas – su SLA suderinamas dinaminis slenksčio koregavimo algoritmas (angl. *SLA Adaptive Threshold Ajustment*, toliau SATA). Šis sprendimas patobulina jau egzistuojančius, taisyklėmis pagrįstus automatinio masteliavimo sprendimus, kurie naudoja resursų apkrovos slenksčius (angl. *utilization thresholds*, toliau slenksčiai) masteliavimo sprendimams priimti. Skyriuje pateikiama išsami siūlomų sprendimų apžvalga, detalizuojami naudojami algoritmai ir funkcijos bei algoritmų vertinimo kriterijai.

Siūlomi metodai publikuoti “Cluster Computing” (Pozdniakova et al., 2023) ir “Electronics” (Pozdniakova et al., 2024) tarptautiniuose žurnaluose.

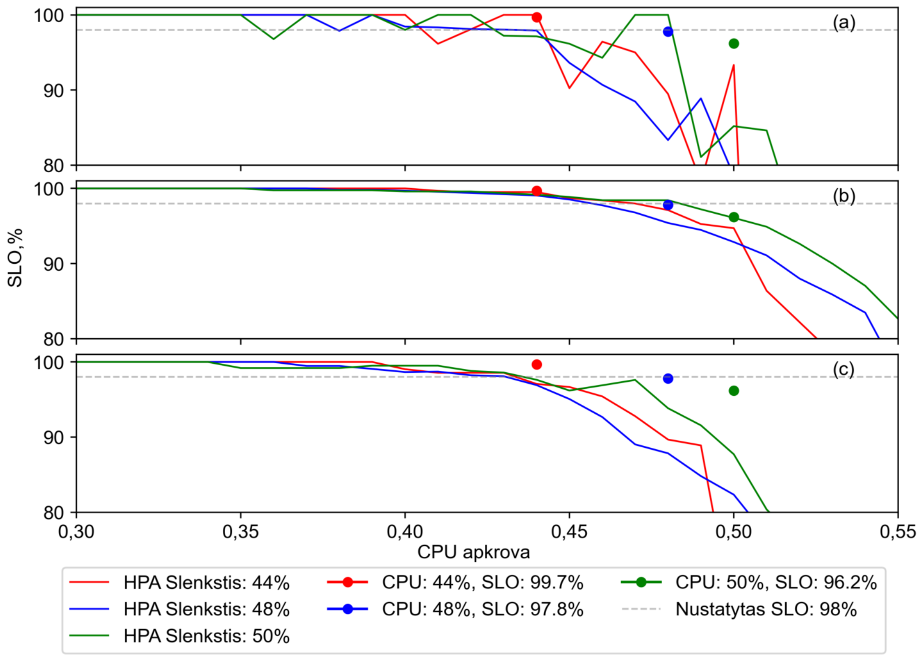
Pirmasis skyriuje pristatytas SAA algoritmas sprendžia problemas, susijusias su skirtingais taikomosios programos išteklių ir našumo reikalavimais, debesų kompiuterijos teikiamų resursų našumo svyravimais, dinamiškomis darbo krūvio charakteristikomis, laiku priimamais masteliavimo sprendimais ir resursų valdymo svyravimo (angl. *oscilation*) mažinimu. Šios problemos turi būti sprendžiamos tam, kad būtų užtikrinti programų našumo SLA reikalavimai. SAA sudaro keli moduliai, kurių kiekvienas sprendžia konkrečią problemą. Automatinio masteliavimo modulis yra pagrindinis sistemos sprendimų priėmimo komponentas, atsakingas už automatinio masteliavimo algoritmo vykdymą. SAA naudoja dinaminį slenksčių masteliavimo metodą. Dinaminis CPU slenksčių koregavimo modulis reguliuoja CPU slenksčius, atsižvelgdamas į dabartinę paslaugos lygio tikslo (SLO) būseną. Jis padeda sumažinti infrastruktūros veikimo svyravimų poveikį ir užtikrina SLA vykdymui būtiną sistemos našumą. Apkrovos kitimo greičiui nustatyti naudojamas pagreičio poveikio veiksnio modulis. Kintančio srauto detektoriaus modulis identifikuoja dažnai kintančio srauto požymius, o tai leidžia automatinio masteliavimo moduliui priimti konservatyvesnius sprendimus, siekiant išvengti SLA nevykdymo. Priklausomai nuo srauto pagreičio, sprendimas naudoja skirtingos trukmės laikotarpius, per kuriuos nevykdoma jokia masteliavimo veikla, taip, esant reikalui, pagreitinant arba sustabdant resursų pridėjimą arba pašalinimą.

SAA efektyvumui bei našumui įvertinti buvo naudojami du kriterijai. Tyrimuose automatinio masteliavimo sprendimai pirmiausia buvo įvertinti pagal jų gebėjimą teikti paslaugas pageidaujamu arba aukštesniu SLO lygiu per visą vertinimo laikotarpį. Antrinio vertinimo kriterijus apima bendrą sprendimo suteiktų, nesuteiktų, perteklinį ir ne laiku suteiktų konteinerių skaičių, teorinį konteinerių poreikį (skyriuje žymimi kaip *touchstone* automatinio masteliavimo sprendimo konteineriai) bei santykį tarp suteiktų ir teorinių konteinerių kiekių.

Toliau skyriuje pristatomas SATA algoritmas, kuris susideda iš dviejų dalių. Pirma dalis – tai CPU apkrovos slenksčio nustatymo pagal SLA reikalavimus metodas, o antra – algoritmo, atsakingo už slenksčio dinaminį parinkimą, prototipas.

Skyriuje detaliai aprašomas CPU apkrovos slenksčio nustatymo metodas, kuriuo užtikrinama, kad sistemos našumas atitiktų apibrėžtą našumo SLA. Siūlomas metodas sudarytas iš kelių žingsnių. Pirmuoju žingsniu surenkamas toks metrikų kiekis, kad būtų galima teikti patikimus slenksčio vertės siūlymus. Šiame žingsnyje renkama CPU apkrovos metrika (toliau CPU) bei paslaugų veikimo lygio indikatoriaus (SLI) metrika. Antrame žingsnyje tam, kad algoritmo teikiami rezultatai būtų kuo tikslesni, iš surinktos metrikos verčių pašalinamos išskirtys (angl. *outliers*) bei blogos reikšmės. Trečias žingsnis – metrikų duomenys surūšiuojami pagal CPU ir sugrupuojami pagal CPU rėžius. Toliau, kiekvienam CPU rėžiui apskaičiuojamas pasiektas rėžio SLO – procentinis santykis tarp įvykių, kuriuose SLI reikšmė atitinka SLA nustatytą reikšmę, ir visų į šį rėžį patenkančių matavimo įvykių skaičių. Šis santykis naudojamas sukurti santykio kreivę tarp CPU rėžio ir rėžio SLA (pav S2.1a). Ketvirtas žingsnis – taikant glodinimo metodą, iš sukurtos kreivės siekiama pašalinti triukšmą (pav S2.1a ir c). Paskutinis žingsnis – tinkamas slenkstis nustatomas pasirenkant didžiausią CPU reikšmę, kur rėžio SLO atitinka pageidaujamą SLO. Siūlomas metodas gali būti naudojamas kaip pagalbinė priemonė pirminiams apkrovos slenksčiams nustatyti.

Toliau pateikiamas prototipo, sukurto įvertinti siūlomo metodo veiksmingumą aprašymas. Prototipas sukurtas veikimui su HPA. Prototipas naudoja skirtingas taisykles ir algoritmus tam, kad koreguotų CPU apkrovos slenkstį pagal veikimo būsenas. Sprendimo veiksmingumui bei efektyvumui įvertinti buvo taikomi trys kriterijai – vienas pagrindinis ir du antriniai. Pagrindinis vertinimo kriterijus buvo gebėjimas vykdyti SLO per visą vertinimo laikotarpį (kiek algoritmas veiksmingas). Gebėjimas operuoti lygyje, artimame nustatytam SLO, ir per stebėjimo laikotarpį panaudotų konteinerių suma buvo antriniai kriterijai . Tikslumas buvo vertinamas apskaičiuojant simetrinę procentinę absoliutinę paklaidą.



S2.1 pav. Grafinis slenksčio nustatymo algoritmo atvaizdavimas. (a) SLO rėžio ryšio su konkrečiais CPU rėžiais linijinė diagrama (be glodinimo); (b) Atitikmens tarp SLO ir CPU rėžių linijinė diagrama, naudojant paprastojo slankiojo vidurkio glodinimą; (c) Atitikmens tarp SLO ir CPU rėžių linijinė diagrama, naudojant centruoto slankiojo vidurkio glodinimą. Taškai nurodo SLO, gautą atliekant eksperimentus, kai *Horizontal Pod Autoscaler* (HPA) buvo sukonfigūruotas naudojant konkretų statinį CPU apkrovos slenkstį

Skyrius baigiamas šiomis išvadomis:

1. SAA sprendimas skirtas paslaugų teikimo lygio susitarimo užtikrinimo uždaviniams spręsti. Šiam tikslui pasiekti buvo sukurti keli moduliai, kurie atsakingi už tokias funkcijas kaip prisitaikymas prie apkrovos pokyčių, įvairių taikomųjų išteklių ir našumo reikalavimų tenkinimas, prisitaikymas prie debesų kompiuterijos išteklių skirtumų, sprendimų dėl mastelio keitimo terminų laikymasis ir apkrovos svyravimų pasekmių mažinimas. Be to, SAA siekia atstatyti SLA įvykdymo būse-

ną iš neįvykdytos į įvykdytą. Remiantis atlikta literatūros apžvalga, galima teigti, kad tik SAA turi SLA įvykdymo atkūrimo mechanizmus. Šie mechanizmai yra išskirtinė sprendimo savybė, galinti užtikrinti paslaugų lygio tikslų (SLO) įvykdymą scenarijuose, kuriuose papildomų išteklių suteikimas gerokai pagerina našumo bei paslaugos kokybės rezultatus.

2. Šiame skyriuje pristatytas automatinis apkrovos slenksčio nustatymo metodas, pagrįstas duomenų aiškinamąja analize ir slankiojo vidurkio glodinimu, kurie leidžia įgyvendinti sprendimą be išsamių žinių apie mašininio mokymo metodus.
3. Kiekvienam metodui siūlomi naujoviški vertinimo kriterijai. Šie kriterijai leidžia įvertinti sprendimus dvejomis perspektyvomis – SLA vykdymo veiksmingumo ir išteklių valdymo efektyvumo.

3. Su paslaugų teikimo lygio sutarties reikalavimais suderinamo dinaminio slenksčio koregavimo algoritmo, skirto taisyklėmis pagrįstiems automatinio masteliavimo sprendimams, kūrimas ir vertinimas

Apžvelgiami trečiame skyriuje pristatytų taisyklėmis pagrįstų automatinio masteliavimo sprendimų eksperimentiniai tyrimai ir jų rezultatai. Be to, aprašoma eksperimentinė aplinka ir sprendimų konfigūraciniai nustatymai. Galiausiai, skyrius užbaigiamas išbandytų automatinio masteliavimo sprendimų rezultatų apibendrinimu.

Pateikiami tyrimai, o dalis rezultatų paskelbti “Cluster Computing” (Pozdniakova et al., 2023) ir “Electronics” (Pozdniakova et al., 2024) tarptautiniuose žurnaluose.

Pirmiausia, skyriuje pateikiami SAA sprendimo eksperimentinės aplinkos ir eksperimentų aprašymai. SAA sprendimas buvo palygintas su *Kubernetes Horizontal Pod Autoscaler* (toliau HPA) ir kitu, taisyklėmis grindžiamu metodu, naudojanču dinamiškai koreguojamus slenksčius – *Dynamic Multi-level Auto-scaling Rules* (toliau DMAR). Tyrimams buvo naudojami įvairaus dydžio *Azure Kubernetes Sevice* klasteriai, taikomosios programos, parašytos *Rust* ir *Java* programavimo kalbomis, 5 tipų sintetinės apkrovos bei dvi apkrovos, sugeneruotos naudojant internetinių portalų vartotojų prieigos žurnalus (WorldCup’98, EDGAR). Apkrova buvo generuojama naudojant *Gatling* bei *JMeter* apkrovos generavimo įrankius.

SAA tyrimo rezultatai buvo padalinti į tris duomenų rinkinius. Pirmas rinkinys – tai duomenys surinkti, per etapą, kai paslaugos kokybės (angl. *Quality of Service*, toliau QoS) lygis buvo atstatomas iki lygio, nustatyto SLA. Antras – QoS išlaikymo etapas, o trečias – viso eksperimento rezultatų apibendrinimas (lentelė S3.1). QoS atkūrimo etape, $n \in [0; 199]$, sprendimų SLO vertės atstatymo efektyvumas buvo įvertintas stipraus SLO vertės sumažėjimo metu. QoS palaikymo etape, $n \in [200; N]$, sprendimų efektyvumas buvo įvertintas įprasto veikimo laikotarpiu, t. y., kai QoS teikiama pagal SLO reikalavimus su galimai nedideliais paslaugos kokybės svyravimais (mažesniais nei 5% per vertinimo laikotarpį). Eksperimentų rezultatai parodė, kad SAA sprendimas yra veiksmingas atkuriant ir palaikant nustatytus paslaugos lygio tikslus esant skirtingoms apkrovoms. Toliau pateikiami SATA eksperimentinio tyrimo rezultatai bei detalus jų aprašymas.

SATA prototipas buvo įvertintas *Azure Kuberntes Services* platformoje, naudojant konteinerizuotas taikomąsias programas, reikalaujančias daug CPU resursų. Programos

S3.1 lentelė: HPA, DMAR ir SAA efektyvumo įvertinimas QoS palaikymo ir QoS atkūrimo metu esant skirtingiems sintetinės apkrovos scenarijams

Apkrovos tipas	Lėtai kintanti			Vidutiniškai kintanti			Greitai kintanti			Pikai ir pauzės			Mišri		
Parametras[a]	HPA	DMAR	SAA	HPA	DMAR	SAA	HPA	DMAR	SAA	HPA	DMAR	SAA	HPA	DMAR	SAA
Įvertinimas QoS palaikymo metu ($i = 200$, $j = 1220$)															
SLA vykdymo laikas, %	29,7	97,7	100,0	0,0	0,0	82,3	0,0	100,0	97,0	0,0	0,0	49,7	0,0	28,4	56,4
Santykis tarp suteiktų konteinerių ir teorinio poreikio	1,04	1,09	1,1	0,93	1,33	1,59	0,77	2,31	1,6	0,68	3,15	3,44	0,92	2,83	2,47
Suteikti konteineriai	4664	4906	4900	3568	6061	7559	3119	10321	7135	1778	10126	11594	2476	9099	8469
Teorinis konteinerių poreikis	4476	4495	4468	3822	4563	4762	4037	4476	4472	2620	3213	3374	2702	3216	3435
Ne laiku suteikti konteineriai	410	533	724	1076	1622	2915	1288	5887	2835	1554	7023	8254	926	5977	5130
Laiku nesuteikti konteineriai	–111	–61	–146	–665	–62	–59	–1103	–21	–86	–1198	–55	–17	–576	–47	–48
Perteklinis konteinerių kiekis	299	472	578	411	1560	2856	185	5866	2749	356	6968	8237	350	5930	5082
Vertinimas QoS atkūrimo laikotarpiu (i = 0, j = 199)															
Santykis tarp suteiktų konteinerių ir teorinio poreikio	0,94	0,99	1,01	0,76	1,25	1,45	0,64	3,05	2,65	0,46	3,18	2,3	0,71	2,04	2,43
Suteikti konteineriai	777	841	860	557	1050	1286	541	2748	2447	305	2065	1567	523	1718	2091
Teorinis konteinerių poreikis	826	846	855	729	841	887	850	902	923	657	650	682	734	841	862

[a] Jei nenurodyta kitaip, matavimo vienetas — vnt per periodą

Lentelės S3.1 pabaiga

Apkrovos tipas	Lėtai kintanti			Vidutiniškai kintanti			Greitai kintanti			Pikai ir pauzės			Mišri		
Parametras[a]	HPA	DMAR	SAA	HPA	DMAR	SAA	HPA	DMAR	SAA	HPA	DMAR	SAA	HPA	DMAR	SAA
Vertinimas QoS atkūrimo laikotarpiu (i = 0, j = 199)															
Ne laiku suteikti konteineriai	73	69	83	290	305	427	337	1884	1572	460	1461	927	251	989	1263
Laiku nesuteikti konteineriai	–61	–37	–39	–231	–48	–14	–323	–19	–24	–406	–23	–21	-231	–56	–17
Perteklinis konteinerių kiekis	12	32	44	59	257	413	14	1865	1548	54	1438	906	20	933	1246
Įvertinimas QoS palaikymo metu ($i = 200$, $j = 1220$)															
Santykis tarp suteiktų konteinerių ir teorinio poreikio	1,03	1,08	1,08	0,91	1,32	1,57	0,75	2,43	1,78	0,64	3,16	3,25	0,87	2,67	2,46
Suteikti konteineriai	5448	5755	5767	4128	7120	8852	3663	13089	9590	2085	12194	13170	3004	10822	10570
Teorinis konteinerių poreikis	5309	5348	5330	4555	5410	5656	4890	5383	5401	3282	3863	4057	3440	4060	4301
Ne laiku suteikti konteineriai	483	603	807	1367	1930	3342	1625	7786	4409	2017	8487	9189	1178	6968	6399
Laiku nesuteikti konteineriai	–172	–98	–185	–897	–110	–73	–1426	–40	–110	–1607	–78	–38	–807	–103	–65
Perteklinis konteinerių kiekis	311	505	622	470	1820	3269	199	7746	4299	410	8409	9151	371	6865	6334

[a] Jei nenurodyta kitaip, matavimo vienetas — vnt per periodą

buvo parašytos *Java* ir *Rust* programavimo kalbomis bei pasižymėjo skirtingomis našumo charakteristikomis. Apkrovai generuoti buvo naudojami internetinių portalų *WorldCup'98* ir EDGAR prieigos žurnalai bei *Gatling* apkrovos generavimo įrankis.

Skyriuje aprašyti eksperimentai, skirti sprendimo konfigūracijos parametrų įtakai jo veiksmingumui bei našumui įvertinti. Buvo atlikti eksperimentai su skirtingomis slenksčio

S3.2 lentelė. HPA, DMAR ir SAA veiksmingumo įvertinimas realiojo pasaulio prieigos žurnalais pagrįstuose srauto apkrovos scenarijuose

Apkrovos tipas	WorldCup'98			EDGAR		
Parametras[a]	HPA	DMAR	SAA	HPA	DMAR	SAA
Įvertinimas QoS palaikymo metu (i = 200, j = 1500)						
SLA vykdymo laikas, %	68,2	83,4	100,0	0,0	0,0	93,5
Santykis tarp suteiktų ir teorinio konteinerių poreikio	1,25	1,17	1,58	0,84	0,94	2,15
Suteikti konteineriai	15859	14691	19538	1512	12378	22023
Teorinis konteinerių poreikis	12698	12560	12339	1799	13222	10227
Ne laiku suteikti konteineriai	3201	2163	7199	957	1350	11796
Laiku nesuteikti konteineriai	–20	–16	0	–622	–1097	0
Perteklinis konteinerių kiekis	3181	2147	7199	335	253	11796
Įvertinimas QoS atkūrimo laikotarpiu (i = 0, j = 199)						
Santykis tarp suteiktų ir teorinio konteinerių poreikio	0,94	1,04	1,46	1,04	1,01	1,76
Suteikti konteineriai	694	715	1026	334	1587	2418
Teorinis konteinerių poreikis	741	689	703	320	1568	1370
Ne laiku suteikti konteineriai	59	30	333	116	215	1064
Laiku nesuteikti konteineriai	–53	–2	–5	–51	–98	–8
Perteklinis konteinerių kiekis	6	28	328	65	117	1056
Įvertinimas viso eksperimento metu (i = 0, j = 1500)						
Santykis tarp suteiktų konteinerių ir teorinio poreikio	1,23	1,16	1,58	0,87	0,94	2,11
Suteikti konteineriai	16560	15413	20574	1847	13977	24460
Teorinis konteinerių poreikis	13446	13256	13049	2120	14802	11606
Ne laiku suteikti konteineriai	3260	2193	7535	1073	1565	12870
Laiku nesuteikti konteineriai	–73	–18	–5	–673	–1195	–8
Perteklinis konteinerių kiekis	3187	2175	7530	400	370	12862

[a]Jei nenurodyta kitaip, matavimo vienetas — vnt per periodą

koregavimo ir įvertinimo trukmėmis, paslaugų lygio indikatoriais (vidutiniu atsako laiku ir laiko procentiliu) bei glodinimo metodais (paprastoju (angl. *Simple Moving Average*, toliau SMA) ir centruotu (angl. *Centered Moving Average*, toliau CMA) slankiaisiais vidurkiais).

Sprendimo konfigūracijos parametrų įtakos vertinimo eksperimentams buvo naudojama taikomoji programa, parašyta *Java* programavimo kalba, ir *WorldCup'98* skirtingo ilgio fragmentai. Atitinkamų eksperimentų rezultatai pateikti lentelėse S3.3, S3.4, S3.5, S3.6. Šiose bei sekančiose lentelėse parinktų laikotarpių ilgiai pateikti naudojant masteliavimo laikotarpio trukmę kaip matavimo vienetą, pavyzdžiui, SMA 4×10, reiškia SMA su 4 masteliavimo periodų ilgio slenksčio koregavimo periodu ir 10 masteliavimo periodų ilgio slenksčio įvertinimo periodu. Aprašytų eksperimentų atveju masteliavimo periodo ilgis buvo lygus 90 s.

S3.3 lentelė. Slenksčio vertės koregavimo laikotarpių poveikio algoritmo efektyvumui įvertinimo rezultatai

Parametrai	4×10	4×20	8×10	8×20
SLO įvykdymas	Pilnas	Pilnas	Pilnas	Pilnas
Simetrinė procentinė absoliutinė paklaida, %	**1,6**	1,8	1,9	1,8
Bendras konteinerių kiekis, vnt per periodą	12992	**12783**	14502	13107
Skirtumas nuo geriausio bendro konteinerių kiekio rezultato, %	1,6	**0**	13,4	2,5

S3.4 lentelė. Paslaugos lygio indikatoriaus pasirinkimo poveikio algoritmo efektyvumui įvertinimo rezultatai

Parametrai	SMA 4×10		SMA 4×10	
SLI	98 procentilis	Vidutinis atsako laikas	98 procentilis	Vidutinis atsako laikas
SLO įvykdymas	**Pilnas**	Iš dalies	Pilnas	Iš dalies
Simetrinė procentinė absoliutinė paklaida, %	**1**	1,3	2,5	0,9
Bendras konteinerių kiekis, vnt per periodą	**43460**	34524	43809	39726
Skirtumas nuo geriausio bendro konteinerių kiekio rezultato, %	**0**	–20	1	–9

Toliau buvo atlikti eksperimentai, skirti SATA veikimui vertinti esant skirtingiems apkrovos tipams bei aplinkoms. HPA sprendimas buvo naudojimas kaip etalonas sprendimo efektyvumui palyginti. HPA CPU apkrovos slenksčio reikšmė buvo nustatyta taip, kad butų užtikrintas SLO įvykdymas esant didžiausiai slenksčio vertei, siekiant maksimaliai padidinti resursų teikimo efektyvumą.

Atliktų eksperimentų rezultatai atskleidė (žr. lentelės S3.5, S3.6, S3.7), kad algoritmas parodė didesnį efektyvumą, bet mažesnį tikslumą, kai slenksčio įvertinimo periodas buvo ilgesnis. Algoritmas buvo efektyvesnis dažniau koreguojant slenksčio vertę ir nau-

S3.5 lentelė. SATA įvertinimo rezultatai, naudojant Java taikomąją programą ir WorldCup'98 apkrovą

Parametrai	CMA 4 × 10	CMA 4 × 20	SMA 4 × 10	SMA 4 × 20	HPA 47%
SLO įvykdymas	Visiškas	Visiškas	**Visiškas**	Visiškas	Visiškas
Simetrinė procentinė absoliutinė paklaida, %	1,3	1,4	**0,9**	1,8	1,4
Bendras konteinerių kiekis, vnt	43769	44652	43745	46161	**42178**
Skirtumas nuo geriausio bendro konteinerių kiekio rezultato, %	4	6	4	9	**0**

S3.6 lentelė. SATA įvertinimo rezultatai, naudojant Java taikomąją programą ir EDGAR apkrovą

Parametrai	CMA 4 × 10	CMA 4 × 20	SMA 4 × 10	SMA 4 × 20	HPA 35%
SLO įvykdymas	Iš dalies	Visiškas	Visiškas	**Visiškas**	Visiškas
Simetrinė procentinė absoliutinė paklaida, %	0,7	1,6	1,6	1,5	**0,9**
Bendras konteinerių kiekis, vnt	25899	24606	26523	22517	**20501**
Skirtumas nuo geriausio bendro konteinerių kiekio rezultato, %	27	20	29	10	**0**

S3.7 lentelė. SATA įvertinimo rezultatai, naudojant Rust taikomąją programą ir WorldCup'98 bei EDGAR apkrovas

Apkrova	WorldCup'98		EDGAR	
Sprendimas	HPA 69%	SATA 4 × 10	HPA 42%	SATA 4 × 10
SLO įvykdymas	Visiškas	Visiškas	Visiškas	Visiškas
Simetrinė procentinė absoliutinė paklaida, %	2,7	**2,5**	**1,3**	2,8
Bendras konteinerių kiekis, vnt	**45199**	46046	**23379**	29015
Skirtumas nuo geriausio bendro konteinerių kiekio rezultato, %	**0**	1,8	0	25

dojant 98-ąjį SLI procentilį. Variantas, kuriame CMA buvo naudojamas kaip glodinimo būdas, parodė didesnį efektyvumą esant apkrovos šuoliams. Variantas su SMA buvo jautresnis staigiems apkrovos padidėjimams. Abiejų algoritmų tikslumas buvo panašus, tačiau verta paminėti, kad CMA 4×10 nepavyko užtikrinti norimo našumo lygio naudojant

EDGAR scenarijų. Todėl tolesniuose vertimuose buvo naudojamas SMA 4×10 konfigūracijos rinkinys.

Remiantis eksperimentais, buvo suformuluotos tokios išvados:

1. Abu siūlomi sprendimai veiksmingi užtikrinant SLA našumo reikalavimų įvykdymą.
2. Buvo pristatytas pirminis SLO atkūrimo metodo modelis ir eksperimentiniais rezultatais įrodytas jo veiksmingumas. Paslaugų lygio teikimo tikslų (SLO) sekimo įtraukimas padėjo abiems sprendimams pasiekti SLA įvykdymo tikslą.
3. Eksperimentai parodė, kad dinaminis slenksčio vertės atnaujinimas yra efektyvus užtikrinant SLA įvykdymą programose, kuriose našumo slenksčio vertės nustatymas turi didžiausią įtaką SLA vykdymui.

Bendrosios išvados

Atliktų tyrimų tikslas – pagerinti žinias apie prie SLA prisitaikančius automatinio masteliavimo algoritmus ir metodus, naudojamus debesų kompiuterijos kilmės konteinerizuotoms taikomosioms programoms. Suformuluotos tokios atlikto tyrimo bendrosios išvados:

1. Literatūros apžvalga rodo, kad debesų kompiuterijos kilmės paradigmos tyrimas yra aktualus ir akademinėje aplinkoje, ir pramonėje. Apžvalga atskleidė, kad sukurti įvairūs automatinio masteliavimo sprendimai, taikantys tiek paprastas taisyklėmis pagrįstas politikas, tiek sudėtingus mašininio mokymo modelius. Mašininiu mokymusi pagrįsti algoritmai yra efektyvesni, užtikrinant resursų teikimo terminų laikymasi, palyginti su taisyklėmis pagrįstais sprendimais. Tačiau dėl savo paprastumo taisyklėmis pagrįsti automatinio masteliavimo sprendimai dažniau sutinkami praktikoje. Vienas iš tokių pavyzdžių yra labiausiai paplitęs automatinio masteliavimo sprendimas, vadinamas *Kubernetes Horizontal Pod Autoscaler* (HPA). Dėl savo populiarumo, HPA akademinėje aplinkoje yra tiriamas kaip atskiras reiškinys. Taisyklėmis pagrįsti sprendimai yra reaktyvūs, o tai dažnai lemia uždelstą sprendimų priėmimą ir, kaip pasekmę, SLA pažeidimus. Vienas iš būdų atstatyti SLA į pageidaujamą lygį gali būti mechanizmas, padedantis teikti paslaugas aukštesniame nei numatyta SLA lygyje. Atlikus literatūros apžvalgą, akademinėje literatūroje panašūs mechanizmai nebuvo aptikti. Be to, literatūros apžvalga atskleidė, kad dauguma išanalizuotų sprendimų masteliavimo sprendimus priimdavo remdamiesi tik informacija apie momentinius paslaugų teikimo lygio rodiklių pokyčius. Šie sprendimai neatsižvelgdavo į SLA būsenos pasikeitimus per ilgesnį laikotarpį, o tai lėmė neišsamų tikrosios SLA įvykdymo būsenos vaizdą. Taip pat apžvalga atskleidė vienodos vertinimo metodikos, kurį leistų efektyviai įvertinant automatinio masteliavimo sprendimų efektyvumą užtikrinant SLA, trūkumą.

2. Pasiūlytas SAA sprendimas įgyvendina SLO įvykdymo būsenos atkūrimo mechanizmą, kai resursų pridėjimas gali žymiai pagerinti paslaugos kokybę. Paslaugos lygio degradacijos atvejais sprendimas arba prideda papildomų resursų, arba sustabdo resursų mažinimo veiksmus, kol vėl pasiekiami nustatyti paslaugos teikimo lygio tikslai, taip pagerinant nustatytų SLA tikslų įvykdymą. SLO įvykdymo būsenos atkūrimo veiksmingumas ir resursų valdymo našumas buvo palyginti su

DMAR ir HPA sprendimais. Gebėjimas kuo greičiau atkurti bei kuo ilgiau išlaikyti numatytą paslaugos lygį buvo pasiūlytas kaip metodas sprendimo SLA reikalavimų įvykdymo veiksmingumui vertinti. *Touchstone* automatinio masteliavimo sprendimas taip pat buvo pasiūlytas kaip etalonas bendram resursų kiekio nustatymo efektyvumui vertinti. Šešiuose iš septynių atliktų eksperimentų SAA parodė geresnius SLA tikslų palaikymo ir atkūrimo rezultatus, palyginti su HPA ir DMAR. Resursų panaudojimo perviršis buvo 1,5–3,5 karto didesnis, palyginti su *touchstone* automatinio masteliavimo sprendimu, ir priklausė nuo darbo krūvio tipo. Šis perviršio lygis buvo panašus į DMAR. Pasiekti rezultatai palaiko pirmą ginamąjį teiginį, kad SLO įvykdymo būsenos atkūrimas, pridedant papildomus resursus, gali būti naudojamas siekiant pagerinti nustatyto SLO įvykdymą. Šis metodas ypač aktualus lygiagrečių skaičiavimų atvejais.Tolesni darbai turėtų būti orientuoti į parametrų reikšmių nustatymo automatizavimą, naudojant statistinius metodus ar mašininį mokymąsi.

3. Pasiūlytas SATA sprendimas yra dinaminis slenksčio reguliavimas, skirtas automatinio masteliavimo valdikliams, pagrįstas slenksčiais. Tai leidžia valdikliams pasiekti nustatytus SLO našumo reikalavimus, kai resursų panaudojimo slenkstis yra labiausiai SLA įvykdymui įtaką darantis veiksnys. Naudojant per nustatytą laikotarpį surinktų SLA pažeidimų skaičių kartu su resursų panaudojimu kaip įvestis slenksčio nustatymui, SATA suskaičiuoja pažeidimų skaičių, atsiradusį tam tikruose panaudojimo rėžiuose, ir taip nustato slenksčio vertę. SATA buvo pritaikytas veikimui su HPA, kad įvertintų siūlomo metodo veiksmingumą, siekiant pagerinti SLA įvykdymą. Šio tyrimo eksperimentų metu HPA, naudojant SATA sprendimą, sugebėjo savarankiškai prisitaikyti prie aplinkos našumo pokyčių, pasiekdamas našumo lygius, kurie užtikrino sistemos veikimą arti nustatyto SLO su 1–2,7 % tikslumu. Sprendimas sėkmingai išlaikė SLO 15 iš 16 vertintų atvejų, net ir neturint išankstinės informacijos apie SLA įvykdymą užtikrinantį resursų panaudojimo slenkstį. Resursų perviršis svyravo nuo 10% iki 30%, palyginus su resursais, naudojamais HPA, kai HPA slenksčiui buvo nustatyta aukščiausia leidžiama vertė, leidžianti pasiekti SLO. Perviršio lygis tiesiogiai priklausė nuo generuojamos apkrovos kintamumo. Eksperimentų rezultatai leidžia teigti, kad dinamiškai keičiant slenksčio vertę galima pagerinti SLA įvykdymą taisyklėmis pagrįstuose automatinio masteliavimo sprendimuose, kai panaudojimo slenkstis yra svarbiausias SLA įvykdymui įtaką darantis veiksnys. Tolesni tyrimai turėtų orientuotis į algoritmų stabilumo ir efektyvumo tobulinimą dinamiškose apkrovose, tobulinant esamas taisykles ir taikomus statistinius metodus.
4. SAA ir SATA sprendimai naudojo per ilgesnį laikotarpį surinktų SLA pažeidimų skaičių masteliavimo sprendimų priėmimui. Abu sprendimai parodė geresnius SLA įvykdymo rezultatus, palyginti su HPA ir DMAR. Šie rezultatai rodo, kad SLO būsenos stebėjimas per ilgesnį laikotarpį, kartu su tradiciniais SLA pažeidimų vengimo metodais, pagrįstais taisyklėmis, gali pagerinti nustatytų našumo SLO laikymąsi debesų kompiuterijos automatinio masteliavimo sprendimuose.

Olesia POZDNIAKOVA

RESEARCH OF SERVICE LEVEL AGREEMENT AWARE AUTOSCALING ALGORITHMS FOR CONTAINERIZED CLOUD-NATIVE APPLICATIONS

Doctoral Dissertation

Technological Sciences,
Informatics Engineering (T 007)

KONTEINERIZUOTŲ DEBESŲ KOMPIUTERIJOS PROGRAMŲ SISTEMŲ AUTOMATINIO MASTELIAVIMO ALGORITMŲ, PAGRĮSTŲ SUSUSITARIMU DĖL PASLAUGOS LYGIO, TYRIMAS

Daktaro disertacija

Technologijos mokslai,
Informatikos inžinerija (T 007)

Lietuvų kalbos redaktorė Deimantė Grigaitė
Anglų kalbos redaktorė Jūratė Griškėnaitė

2025 04 15. 15,3 sp. l. Tiražas 20 egz.
Leidinio el. versija *https://doi.org/10.20334/2025-011-M*
Vilniaus Gedimino technikos universitetas
Saulėtekio al. 11, 10223 Vilnius,
Spausdino UAB „Ciklonas“
Žirmūnų g. 68, 09124 Vilnius