VILNIUS TECH
Vilniaus Gedimino technikos universitetas

Kęstutis PAKRIJAUSKAS

INCREASING AVAILABILITY OF STATEFUL MICROSERVICES IN ORCHESTRATED CONTAINER SYSTEMS

DOCTORAL DISSERTATION

TECHNOLOGICAL SCIENCES
INFORMATICS ENGINEERING (T 007)

Vilnius, 2026

2026-020-M

VILNIUS GEDIMINAS TECHNICAL UNIVERSITY

Kęstutis PAKRIJAUSKAS

INCREASING AVAILABILITY OF STATEFUL MICROSERVICES IN ORCHESTRATED CONTAINER SYSTEMS

DOCTORAL DISSERTATION

TECHNOLOGICAL SCIENCES,
INFORMATICS ENGINEERING (T 007)

Vilnius, 2026

The doctoral dissertation was prepared at Vilnius Gediminas Technical University in 2020–2026.

Supervisor

Prof. Dr Dalius MAŽEIKA (Vilnius Gediminas Technical University, Informatics Engineering – T 007).

The Dissertation Defense Council of the Scientific Field of Informatics Engineering of Vilnius Gediminas Technical University:

Chairman

Prof. Dr Arnas KAČENIAUSKAS (Vilnius Gediminas Technical University, Informatics Engineering – T 007).

Members:

Dr Habil. Piotr Lech ARTIEMJEW (University of Warmia and Mazury in Olsztyn, Poland, Informatics Engineering – T 007),

Prof. Dr Rimantas BUTLERIS (Kaunas University of Technology, Informatics Engineering – T 007),

Assoc. Prof. Dr Vadimas STARIKOVIČIUS (Vilnius Gediminas Technical University, Mathematics – N 001),

Dr Povilas TREIGYS (Vilnius University, Informatics Engineering – T 007).

The dissertation will be defended at the public meeting of the Dissertation Defense Council of the Scientific Field of Informatics Engineering in the *Aula Doctoralis* Meeting Hall of Vilnius Gediminas Technical University at **10 a.m. on 21 May 2026.**

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 20 April 2026. A copy of the doctoral dissertation is available for review at the 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 g. 20, LT-44239 Kaunas, Lithuania).

Vilnius Gediminas Technical University book No. 2026-020-M

https://doi.org/10.20334/2026-020-M

kestutis.pakrijauskas@vilniustech.lt

VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS

Kęstutis PAKRIJAUSKAS

BŪSENAS IŠLAIKANČIŲ MIKROPASLAUGŲ PATIKIMUMO DIDINIMAS KOORDINUOTAI VALDOMOSE KONTEINERIŲ SISTEMOSE

DAKTARO DISERTACIJA

TECHNOLOGIJOS MOKSLAI,
INFORMATIKOS INŽINERIJA (T 007)

Vilnius, 2026

Disertacija rengta 2020–2026 metais Vilniaus Gedimino technikos universitete.

Vadovas

prof. dr. Dalius MAŽEIKA (Vilniaus Gedimino technikos universitetas, Informatikos inžinerija – T 007).

Vilniaus Gedimino technikos universiteto Informatikos inžinerijos mokslo krypties disertacijos gynimo taryba:

Pirmininkas

prof. dr. Arnas KAČENIAUSKAS (Vilniaus Gedimino technikos universitetas, Informatikos inžinerija – T 007).

Nariai:

habil. dr. Piotr Lech ARTIEMJEW (Varmijos ir Mazūrijos universitetas Olštyne, Lenkija, Informatikos inžinerija – T 007),
prof. dr. Rimantas BUTLERIS (Kauno technologijos universitetas, Informatikos inžinerija – T 007),
doc. dr. Vadimas STARIKOVIČIUS (Vilniaus Gedimino technikos universitetas, Matematika – N 001),
dr. Povilas TREIGYS (Vilniaus universitetas, Informatikos inžinerija – T 007).

Disertacija bus ginama viešame Informatikos inžinerijos mokslo krypties disertacijos gynimo tarybos posėdyje **2026 m. gegužės 21 d. 10 val.** Vilniaus Gedimino technikos universiteto *Aula Doctoralis* posėdžių salėje.

Adresas: Saulėtekio al. 11, LT-10223 Vilnius, Lietuva.
Tel.: (0 5) 274 4956; faksas (0 5) 270 0112; el. paštas doktor@vilniustech.lt

Pranešimai apie numatomą ginti disertaciją išsiųsti 2026 m. balandžio 20 d. Disertaciją galima peržiūrėti Vilniaus Gedimino technikos universiteto talpykloje http://etalpykla.vilniustech.lt/, Vilniaus Gedimino technikos universiteto bibliotekoje (Saulėtekio al. 14, LT-10223 Vilnius, Lietuva) ir Kauno technologijos universiteto bibliotekoje (K. Donelaičio g. 20, LT-44239 Kaunas, Lietuva).

Abstract

Stateful microservices are essential components in modern cloud computing environments. This research delves into the complexities and challenges of ensuring the reliability and availability of stateful microservices in container orchestration systems such as Kubernetes and Docker Swarm. Unlike stateless microservices, stateful microservices handle critical data, making their management and recovery more intricate and resource-intensive. The research highlights various existing methodologies to enhance the reliability of these services, including resource optimization, machine learning-based predictions, and custom schedulers.

A significant focus of the research is the impact that maintenance activities have on the availability of stateful microservices. The dissertation proposes a method for loosely coupled, transparent failover, which minimizes downtime by observing database connection activity and terminating idle client connections, thereby redirecting requests with minimal client impact. This method was validated through experiments, achieving near-zero downtime during failover operations.

Another critical aspect explored is burst tolerance in stateful microservices. The proposed rule-based method enhances burst tolerance through write-scaling and load balancing, distributing burst workloads across multiple nodes. This approach reduces failure rates and extends operational time under burst conditions, significantly improving service availability. Experimental results demonstrate that this method allows stateful microservices to sustain burst loads for nearly twice as long as traditional methods.

Additionally, the research investigates the influence of various factors such as load intensity, load balancer type, request length, and connection type on the availability of database clusters during failover. The findings indicate that load balancers operating at the 1st OSI level provide higher availability than those operating at the 4th OSI level.

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

Reziumė

Būsenas išlaikantys mikroservisai yra esminiai komponentai šiuolaikinėse debesų kompiuterijos aplinkose. Šiame tyrime gilinamasi į kompleksiškumą ir iššūkius, kylančius užtikrinant būsenas išlaikančių mikroservisų pasiekiamumą ir prieinamumą konteinerių koordinuoto valdymo sistemose, tokiose kaip „Kubernetes“ ir „Docker Swarm“. Skirtingai nuo būsenos nelaikančių mikroservisų, būsenas išlaikantieji tvarko duomenis, todėl jų valdymas ir atkūrimas yra sudėtingesnis ir reikalaujantis daugiau pastangų. Tyrime analizuojamos įvairios egzistuojančios metodikos, skirtos padidinti šių paslaugų patikimumą, įskaitant išteklių optimizavimą, mašininio mokymosi prognozes ir tinkintus planuoklius.

Vienas iš svarbių tyrimo dėmesio objektų yra priežiūros darbų poveikis būsenos turinčių mikroservisų prieinamumui. Disertacijoje siūlomas žemos sąsajos, skaidraus mazgų perjungimo metodas, sumažinantis prastovas, grįstas duomenų bazės klientų aktyvumo stebėjimu ir nenaudojamų klientų sesijų nutraukimu. Taip užklausos yra nukreipiamos į kitus mazgus su minimaliu poveikiu klientui. Šis metodas buvo patvirtintas eksperimentu, pasiekiant nereikšmingo dydžio įtaką mazgų perjungimo metu.

Kitas svarbus nagrinėjamas aspektas yra būsenas išlaikančių mikroservisų atsparumas staigiai padidėjusiai apkrovai. Siūlomas taisyklėmis paremtas metodas grįstas rašymo užklausų masteliavimu ir apkrovos balansavimu, paskirstant padidėjusį darbo krūvį tarp kelių mazgų. Šis metodas sumažina nepavykusių užklausų skaičių ir pailgina veikimo laiką staigiai ir reikšmingai pakilusios apkrovos sąlygomis, žymiai pagerindamas paslaugų prieinamumą. Eksperimentiniai rezultatai rodo, kad šis metodas leidžia būsenas išlaikantiems mikroservisams atlaikyti padidėjusią apkrovą beveik dvigubai ilgiau nei taikant tradicinius metodus.

Taip pat tyrime nagrinėjama įvairių veiksnių, tokių kaip apkrovos intensyvumas, apkrovos balansavimo įrenginio tipas, užklausos ilgis ir ryšio tipas, įtaka duomenų bazių klasterių prieinamumui perjungimo tarp mazgų metu. Išvados rodo, kad apkrovos balansavimo įrenginiai, veikiantys 1-ajame OSI lygyje, užtikrina didesnį prieinamumą, palyginti su apkrovos balansavimo įrenginiais, veikiančiais 4-ajame OSI lygyje.

Disertacijos rezultatai buvo paskelbti 4 moksliniuose straipsniuose, iš kurių 2 – recenzuojamuose mokslo žurnaluose, indeksuotuose *Web of Science* duomenų bazėje, ir pristatyti 2 tarptautinėse konferencijose.

Notations

Abbreviations

API – application programming interface (liet. *taikomųjų programų programavimo sąsaja*);

BAC – backup availability consistency (liet. *atsarginė kopija prienamumas nuoseklumas*);

CAP – consistency availability partitions (liet. *nuoseklumas prienamumas paskirstymas*);

CPU – central processing unit (liet. *centrinis procesorius*);

DevOps – development operations (liet. *plėtros operacijos*);

GCP – Google cloud platform (liet. „Google“ *debesijos platforma*);

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

HPC – high performance computing (liet. *aukšto našumo skaičiavimai*);

YCSB – Yahoo! cloud serving benchmark (liet. „Yahoo!“ *debesijos aptarnavimo lyginamasis rodiklis*);

IT – information technology (liet. *informacinės technologijos*);

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

OSI – open systems interconnection referential model (liet. *lygmeninis tinklų modelis*);

PD – persistent disk (liet. *pastovusis diskas*);

RAM – random access memory (liet. *laisvosios kreipties atmintis*);

RDBMS – relational database management system (liet. *reliacinės duomenų bazės valdymo sistema*);

REST – representational state transfer (liet. *reprezentacinis būsenos perdavimas*);
SD – standard deviation (liet. *standartinis nuokrypis*);
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*);
SQL – structured query language (liet. *struktūrinės užklausos kalba*);
TCP – transmission control protocol (liet. *perdavimo kontrolės protokolas*);
URI – uniform resource identifier (liet. *vienodas išteklių identifikatorius*);
VM – virtual machine (liet. *virtuali mašina*).

Definitions

BURST – in the context of microservices, a sudden and significant surge in service requests (liet. *mikroservisų kontekste staigiai ir reikšmingai padidėjęs užklausų skaičius*);

POD – can be defined as a collection of containers and its storage inside a node of the Kubernetes cluster (liet. *gali būti apibrėžtas kaip konteinerių rinkinys ir jo saugykla Kubernetes klasterio mazge*);

POOLED CONNECTION – in the context of client connections to RDBMS, a cache of reusable database connections managed by the client (liet. *klientų prisijungimo prie RDBVS kontekste, kliento valdomas pernaudojamų prisijungimų prie duomenų bazės*);

STATEFULSET – can be defined as a collection of unique pods managed by a Kubernetes cluster. User to run stateful applications. (liet. *gali būti apibrėžtas kaip unikalių rinkinys prižiūrimas Kubernetes klasterio. Naudojamas būsenas išlaikančioms aplikacijoms*).

Contents

Introduction

Problem Formulation

Nowadays, software and technology are the key differentiators in transforming organizations and delivering value to their customers and stakeholders. Investment in technologies can result in a competitive edge (McAfee & Brynjolfsson, 2008). The scale of IT systems and their importance to businesses is constantly growing.

In large, distributed, and complex systems, reliability is a concern in its own right. The success of a business causally relates to whether its systems are up and running. Tech giants such as Google cannot afford systems to go down for a prolonged period. The velocity of its systems' growth, the number of changes, and the demand for availability push businesses to adopt software engineering practices to solve their infrastructure and operational challenges.

Microservices enable scaling up to meet the demand and keeping pace with the emergence of new technologies. Microservice architecture, an implementation of Service-Oriented Architecture, consists of many independent and autonomous components. Like any other technology, microservices have their own challenges. Compared to monolithic software, monitoring, recovery, and load balancing are different in microservice architecture.

Maintenance and life cycle management activities affect the availability of a stateful microservice. Vertical scaling and patching may require downtime and, thus, the use of the limited availability budget. While the elasticity of stateful microservices in orchestrated container systems is high, improper resource allocation can negatively affect service availability if the workload suddenly and significantly increases. Overprovisioning of resources is an option that comes at a cost of underutilized resources.

Relevance of the Dissertation

Modern systems are becoming increasingly complex, reaching previously unimagined scales. In addition, the demand to achieve high availability of such systems increases as businesses rely on their IT systems more than ever (Mann et al., 2018, 2019).

The demand to improve build reliability and code delivery increases as DevOps practices gain wider adoption (Google LLC, 2023; Mann et al., 2018, 2019). Software teams can support organizational goals better by spending more time on developing and eliminating constraints (B. G. Kim et al., 2016). A constraint on code testing or delivery is eliminated if it reduces the need for manual input or waiting for certain processes to complete. In addition, reliability practices impact overall software delivery performance.

Downtime in microservice-based systems is reduced when stateful microservices return online in a timely and consistent manner. While stateless microservices recovery consists of recreation, stateful microservices require data recovery.

Containers and Kubernetes are widely used technologies in microservices. Stateful services hold and work with important and valuable commodity, i.e., data. Various factors may impact stateful microservice resilience: data size, storage and network speed, data type, encryption and compression algorithms, sharding and clustering, and geo-distribution. Given the distributed and often polyglot persistence of data, timely recovery is important to maintain data consistency across multiple microservices.

Shorter downtime allows an organization to remain competitive in the market or, for a public organization, provide better services. Reduced manpower requirement will result in smaller teams that work faster than larger teams (Newman, 2015). The saved-up unavailability budget can be used for other purposes, such as upgrades and other improvements (Campbell & Majors, 2017).

Research Object

The object of the research is the recoverability and resilience of stateful microservices against failures.

Aim of the Dissertation

The dissertation aims to improve methods and techniques for enhancing the availability of stateful microservices within orchestrated container environments.

Tasks of the Dissertation

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

1. To perform a scientific literature review on stateful microservice reliability and availability in orchestrated container systems.
2. To analyze and evaluate the factors impacting the availability of stateful microservices during failover.
3. To propose a method that allows for performing maintenance activities on stateful microservices with negligible impact on client applications.
4. To propose a method that allows for improving operational time and reducing the failed request rate during workload bursts on stateful microservices.

Research Methodology

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

- An *analytical literature review* was performed on stateful microservices, challenges of state management in distributed or containerized systems, and ML- and rule-based methods to improve stateful microservices availability. Existing gaps in ensuring stateful microservice availability were identified.
- The *statistical exploratory analysis* was applied to identify and evaluate the factors impacting stateful microservice availability during failover.
- The *experimental research method* was applied (1) to evaluate the proposed method and increase stateful microservice availability during failover and (2) to validate and evaluate the proposed rule-based method to increase stateful microservice resilience to burst workloads. Custom and

YCSB stateful load generators were used to generate workload against a stateful microservice. The experimental environment was set up in Google Cloud Platform. The data was collected and analyzed during experiments to assess the effectiveness of the proposed method against predefined evaluation criteria.

Scientific Novelty of the Dissertation

The scientific novelty of this research in informatics engineering is specified as follows:

1. The proposed novel method allows for ensuring high availability of a stateful microservice during maintenance operations and achieving near-zero availability loss when switching between nodes of a stateful microservice. The method stands out as low-coupling is preserved, while other approaches for maintenance either increase it or result in availability loss.
2. The proposed novel rule-based method couples write-scaling with load balancing, thus allowing for the improvement of resilience of a stateful microservice during a sudden and significant workload increase. This approach highlights the importance of algorithmic techniques and ensures high availability of stateful microservices when handling bursty workloads.

Practical Value of the Research Findings

The practical significance of the research lies in its potential to significantly enhance the reliability, availability, and scalability of stateful microservices in real-world applications. By introducing mechanisms for transparent failover and improved burst tolerance, the research ensures that services remain operational during failures and handle sudden workload spikes effectively. This enhancement reduces downtime and optimizes resource usage, leading to cost savings and improved operational efficiency. The proposed methods seamlessly integrate with modern container orchestration platforms like Kubernetes and Docker Swarm, supporting scalable architectures and future-proofing IT infrastructures. In industries such as healthcare, finance, and e-commerce, where continuous availability and performance are critical, these advancements translate into consistent service performance and enhanced user experiences. Additionally, the research lays the groundwork for further innovation and the establishment of best practices in managing distributed systems, contributing to broader advancements in cloud computing and microservices architecture.

Defended Statements

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

1. The OSI level at which a load balancer operates, together with the connection type, most significantly affects the availability of a stateful microservice during failover. A load balancer operating at the 7th OSI level affects the availability less than a load balancer operating at the 4th OSI level.
2. An SQL-aware load balancer and forceful connection termination mechanism can minimize the impact of a graceful failover in a stateful microservice to negligible values while maintaining low coupling with client microservices.
3. Burst tolerance of stateful microservices can be improved by combining load balancing and write scaling. It allows for reducing the impact of a sudden and significant workload increase (a burst) on the availability of a stateful microservice.

Approval of the Research Findings

The results of the dissertation were published in two scientific publications in reviewed scientific journals indexed in the *Clarivate Analytics Web of Science* database with a citation index, and two were published in conference proceedings. The author gave two presentations at international scientific conferences.

- 2021 IEEE Open Conference of Electrical, Electronic, and Information Sciences (eStream), 22 April 2021, Vilnius, Lithuania.
- 8th International Conference on Control, Decision, and Information Technologies (CoDIT), 17–20 May 2022, Istanbul, Turkey.

Structure of the Dissertation

The dissertation consists of an introduction, four main chapters, general conclusions, references, a list of the author's publications on the dissertation topic, and a summary in Lithuanian. In total, the dissertation has 115 pages, 3 equations, 32 figures, 8 tables and 102 references.

1

Literature Review on Stateful Microservice Availability in Orchestrated Container Systems

This chapter aims to summarize the research on the availability of stateful microservices in orchestrated container systems, the challenges of maintaining them, and recent advances in improving their availability. It starts with an introduction to stateful microservices, followed by complexity and challenges arising from having to maintain and manage state or, in other words, data. In addition, the capabilities of container orchestrators on stateful container management are broadly reviewed. This chapter reviews the SLIs and SLOs that could be used to drive decision-making. Finally, a data-driven and rule-based method for improving the availability of stateful microservices in orchestrated container management systems is reviewed.

Four publications discussing the topic of this chapter were published by the author (Pakrijauskas & Mažeika, 2021, 2022a, 2022b, 2025).

1.1. Stateful Microservices

Software and technology play a pivotal role in transforming organizations and delivering value to stakeholders and customers in modern times (Bhatnagar & Mahant, 2025; McAfee & Brynjolfsson, 2008). The success of a business is heavily dependent on its systems operating at the desired state. Microservices, as an implementation of Service-Oriented Architecture, enable companies to meet the demands of scaling and deploying new services (Newman, 2015; J. Yang, 2025). The challenges involved in running microservice-based applications differ from those of monolithic applications, including monitoring, recovery, and load balancing. Data or state are crucial assets for any business.

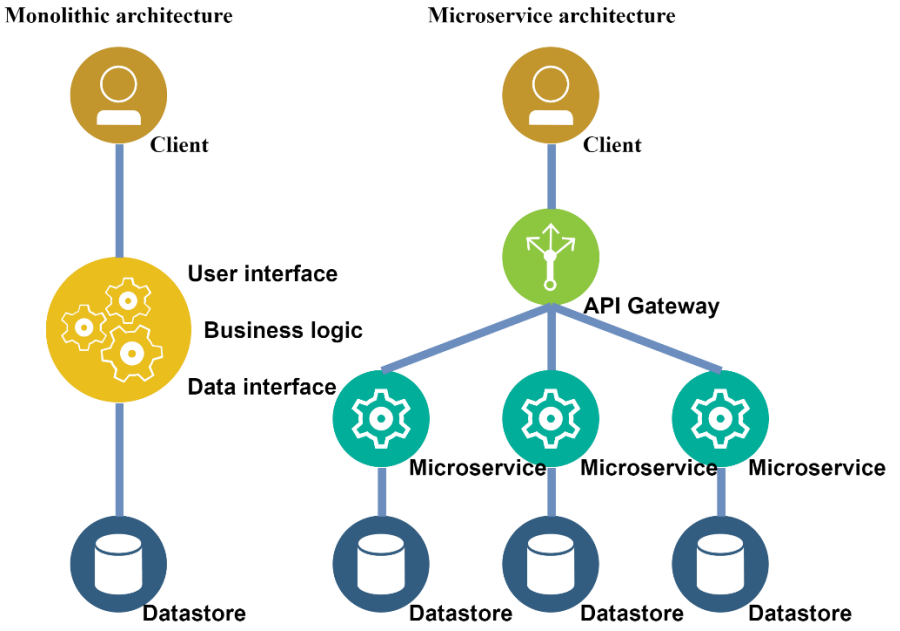


Fig. 1.1. Comparison of monolithic and microservice architectures

According to the DevOps reports from 2018 (Mann et al., 2018), 2019 (Mann et al., 2019), and 2023 (Google LLC, 2023), modern systems are becoming increasingly complex and grow in scale, making high system availability a significant concern (Gorton, n.d.). Consequently, enterprises like Google (Adkins et al., 2020; Google LLC, 2023) prioritize resilience and availability in their system designs. Downtime in microservice-based systems can be reduced if the micro-

services are returned online promptly. A stateless service is constrained to its function, with outputs dependent on inputs. Recovering stateless microservice components is relatively simple, as these components are ephemeral, posing no risk of data loss, and recreating a failed component is straightforward. However, stateful microservices, which manage data, present a more complex scenario.

State is defined as "*a sequence of values over time that contain the intermediate results of a desired computation*" (To et al., 2018). This state complicates deployment, management, scaling, and replication processes. Synchronizing a state across multiple replicas within a microservice is essential. Recovering a stateful microservice is complex and non-trivial. Even robust backup recovery strategies, replication, and sharding may not suffice to ensure high availability. Inter-service state consistency can be compromised if a single microservice is restored to a previous state. Data within a seemingly healthy stateful microservice can become corrupted, necessitating restore or rollback operations. Migrating to a healthier node within the platform poses additional challenges due to the resource-intensive and potentially disruptive nature of stateful microservices.

Container orchestration frameworks like Kubernetes, Docker Swarm, and Mesos were specifically designed with microservices in mind (Jamshidi et al., 2018). These tools facilitate the automation and abstraction of various microservices management tasks, such as service discovery, storage orchestration, rollout and rollback, and resilience (Kubernetes, 2025c). Evaluating the availability of applications within these container orchestration systems is crucial, given their role in managing microservices.

Reliability is typically defined as "*the probability that an item will perform a required function without failure under specified conditions for a given period of time*" (O'Connor & Kleyner, 2012). Mathematical and statistical methods are employed to measure reliability. However, practical uncertainties often make it challenging to calculate reliability accurately. As system costs and complexities rise, reliability has become a crucial parameter for effectiveness. Availability is related to reliability as it measures the proportion of time a system is operational and accessible when required, which is a key aspect of reliability. Reliability, more broadly, refers to the ability of a system to perform its required functions without failure over a specified period. In essence, high availability is an indicator of high reliability, as a system that is consistently available is likely to be reliable.

Modern microservice architecture emphasizes resiliency. Components should be engineered to handle failures rather than aiming for zero failures. Microservices and entire systems should be designed to tolerate failures and recover quickly (Newman, 2015; Pereira & Bruce, 2019). Techniques, such as graceful degradation, decoupling, retries, caching, and redundancy, are employed to ensure the reliability of stateful microservices.

A prerequisite for reliability and availability evaluation is establishing performance standards. These standards are defined by Service Level Agreements (SLAs), Service Level Objectives (SLOs), and Service Level Indicators (SLIs) (Beyer et al., 2016; Prabhu, 2024). An SLI is a defined quantitative measure of a service's performance aspect. An SLO is the target range or value for SLIs. Establishing SLOs is crucial for evaluating reliability and availability, as it provides transparency and clarifies whether the service level meets expectations. An SLA is an agreement with users on actions to be taken if the service does not perform within the SLO.

Having an established service health baseline and accurate predictions about microservice behavior is insufficient. Orchestrated container systems offer a variety of settings and techniques to enhance microservice reliability through data-driven decisions and fault detection.

1.2. Challenges of State Management

There are several differences between stateless and stateful microservices; one of them is the requirement of state management, which is the core difference between the two. State management brings several unique challenges related to data persistence, backups, failover handling, and maintaining consistency in distributed environments.

1.2.1. Federated Multidatabase

Fine-grained microservices are developed and operated by independent teams. Each microservice can be deployed and scaled independently. Stateful services rely on their own persistent storage mechanism. To reduce coupling, integration at the storage level, i.e., using one data storage mechanism, like a database, tends to be avoided. Interaction between microservices should be limited to APIs. However, because there is no guarantee that a link to another microservice to retrieve records is valid, maintaining consistency becomes challenging (Pardon et al., 2018).

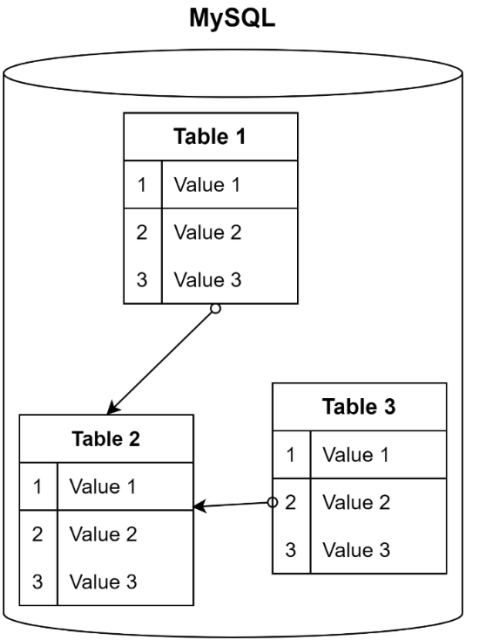


Fig. 1.2. Database in monolithic architecture

Microservices, being autonomous and independently deployed, may store data on a variety of platforms. Each microservice stores persistent data on its own private database. This data is accessible to other services only via API. Relationships to other entities in the REST architecture are expressed as URI links. URI is a Uniform Resource Identifier that globally addresses the referenced entities. The lifecycle of microservices is independent, and databases are backed up periodically and independently. In case of recovery, links between microservices may be broken due to inconsistent states after data are restored from a backup on one of them (Manouvrier et al., 2020).

Microservice architecture is designed to survive the failure of its individual components. Stateless and stateful services can be recovered independently. Since data in stateful services can be recovered from backups, there is a question of whether the restored data is consistent with the data on other microservices. The challenge is ensuring data consistency across multiple microservices, and how and when to perform backup operations (Pardon et al., 2018).

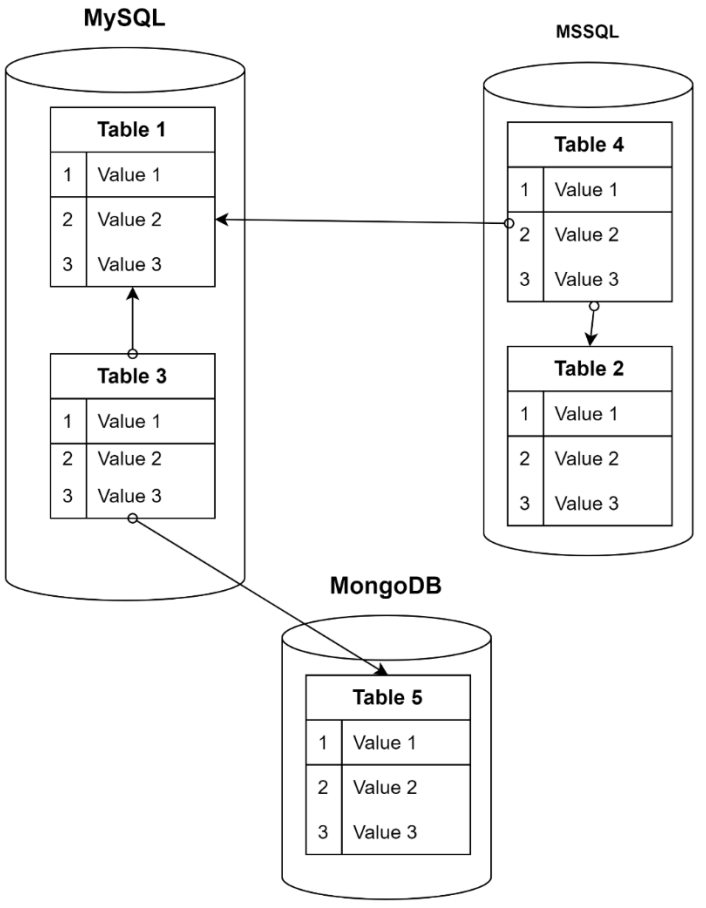


Fig. 1.3. Database in microservice architecture

Databases of microservices can be seen as a federated multidatabase – a hybrid between a centralized and a distributed database system, a database that is distributed for global users and centralized for local users. Each microservice treats its database as a centralized one, ensuring its durability and consistency (Manouvrier et al., 2020). However, managing its consistency is challenging due to distributed persistence. Foreign key relationships between databases of different microservices are represented as loosely coupled references like a URI. There is no guarantee that a retrieved URI points to a valid record in another microservice.

Although backup of individual microservices can be successfully used for independent recovery, it is likely that its state will not be consistent with the state

of the application. For example, if order information is stored across multiple stateful microservices, some of it may be lost in the event of recovery of an individual microservice. Thus, the state will not be consistent.

1.2.2. Backup Availability Consistency

Fine-grained, independent microservices may consist of many components, each possibly with its own mechanism for persistent data storage. Given the varying number of components and technologies in a modern microservice system, ensuring consistent, available backups can be challenging.

The BAC (Backup Availability Consistency) theorem states that: "*When backing up an entire microservices architecture, it is not possible to have both availability and consistency*" (Pardon et al., 2018). The trade-off is between independent microservices, which may lead to eventual inconsistency, and consistent backups of microservices, which lock the application's state for a period, thus limiting application availability. There are systems designed to back up microservices in containerized frameworks to help mitigate the challenge of inconsistent data (Deshpande et al., 2021).

It is like the CAP theorem, which states that "*any networked shared-data system can have at most two of three desirable properties: consistency (C) equivalent to having a single up-to-date copy of the data; high availability (A) of that data (for updates); and tolerance to network partitions (P)"* (Borrill, 2026; Brewer, 2012)

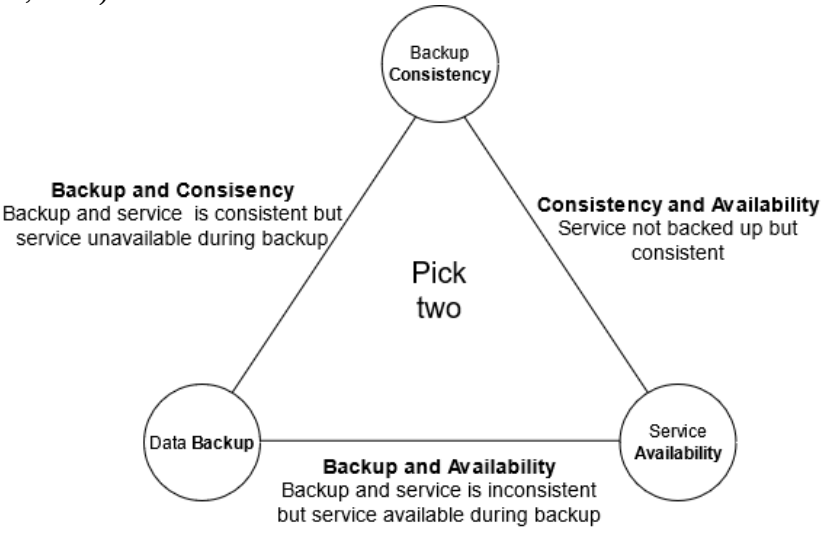


Fig. 1.4. Illustration of the BAC theorem

Inconsistency of services can manifest in the following ways:

- **Broken link:** a reference cannot be followed. For example, when a microservice is referencing data in an obsolete microservice that was restored from a backup.
- **Orphan state:** there is no reference to follow. For example, when data in a microservice is not referenced at all because the referencing microservice has no references to the orphan data.
- **Missing state:** state is obsolete. For example, two microservices have different states because one of them was restored from a backup.

There are three ways to deal with broken links:

- Reconstruct the missing references manually.
- Accept the inconsistency.
- Use cached data.

Dealing with an orphan state starts with its identification and flagging. Once the records are flagged, they can be deleted or overwritten.

A missing state can be reproduced from the source or replicated from other sources.

The challenges posed by BAC can either be acknowledged or avoided. Acknowledging implies that eventual inconsistency is accepted, and there are measures to handle and live with it. Avoiding the challenge of BAC can result in tighter coupling or other design solutions that make microservices more dependent on one another.

1.2.3. Performance Overhead of Stateful Workloads

Container orchestration systems introduced persistent, non-ephemeral storage to support stateful services, as many applications require databases. It is suggested that transient data be stored on containers running databases (Good, 2019) due to the flexibility they offer for scaling and high availability (Vitess, 2022; Zalando, 2022).

Performance overhead of Docker and container orchestration frameworks – Docker Swarm and Kubernetes – was evaluated in a study by E. Truyen, D. Van Landuyt, B. Lagaisse, and W. Joosen (Truyen et al., 2019). The study focuses on evaluating CPU-intensive Cassandra workload.

The study found that deployments to Docker containers result in negligible performance overhead compared to deployments to hosts. However, Docker Swarm and Kubernetes deployments resulted in additional performance overhead related to network and volume plugins.

Another study by Chae et al. confirmed that the performance of container-based software is better compared to Virtual Machine (VM)-based software (Chae et al., 2019).

Network bridges are used in the two evaluated container orchestration frameworks to ensure network isolation between containers. These network plugins increase the CPU utilization. Similar results were found in a study by E. Kim, K. Lee, and C. Yoo (E. Kim et al., 2021).

In addition, persistent volume plugins significantly affect the general database workload resource model. Even though the experimental Cassandra workload was CPU-intensive, volume plugins introduced a performance bottleneck in I/O operations.

On the one hand, the authors argue that container orchestration frameworks bring benefits to SLO-aware container scheduling. On the other hand, container orchestrators introduce additional performance overhead, which is to be optimized.

1.2.4. Stateful Pod Rescheduling

Pod is the smallest deployable unit in Kubernetes. It is a single or a group of containers sharing underlying resources such as storage, network, and namespaces. Deployments and Jobs are used to start Pods running stateless applications. StatefulSet is used to launch stateful Pods (Kubernetes, 2021b).

Pod availability is guaranteed by ReplicaSet. Even though it is a ReplicaSet that ensures the number of running Pods, Pods, and their number updates are made to the Deployment, which is a higher-level concept (Kubernetes, 2021c).

StatefulSet "*manages the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these Pods*" (Kubernetes, 2021d). Similarly, to Deployment, StatefulSet manages identical contained specification Pods. However, Pods in a StatefulSet are not interchangeable: the identifier of each Pod persists through rescheduling. StatefulSet uses persistent volumes, which, combined with stable Pod identifiers, make new Pod mapping to persistent storage easier. A Pod retains its identifier in a StatefulSet if it is restarted.

Each StatefulSet has a defined volumeClaimTemplate that provides persistent storage using PersistentVolumes. While Kubernetes Volumes are ephemeral and dependent on Pod lifecycle, a PersistentVolume is independent from any Pod that is using it. It is the PersistentVolumeClaim that is a storage request by a Pod.

In StatefulSet, PersistentVolumeClaim, used by a failed Pod, is not removed, as in the case of Volume in Deployment. The failed Pod is relaunched using the bind PersistentVolumeClaim. Thus, every Pod in a StatefulSet used the same PersistentVolumeClaim (Kubernetes, 2021a).

Kubernetes uses Horizontal Pod Autoscaler (HPA) that scales the number of Pods up and down based on resource utilization, for example, CPU or a custom-defined metric. However, additional application-specific actions, such as setting up replication, must be completed if a stateful Service is scaled out (Kubernetes, 2025b).

When a stateful service is scaled out, data must be replicated to the new Pod. Depending on the storage mechanism, for example, MySQL RDBMS, the overall performance of the stateful microservice is degraded during the process of copying the data. In addition, while a stateful Pod is being rescheduled, it may not be available for use because of Readiness and Liveness probes that wait while the Pod is fully operational.

1.3. Service Level Indicators and Objectives to Drive Decision-Making

Historic Service Level Indicators (SLIs) and Service Level Objectives (SLOs) can serve as the foundation for data-driven decisions to proactively assess, enhance, and maintain microservices. However, a critical question arises: How should this historical data be processed before it can be effectively used in various algorithms such as machine learning, deep learning, neural networks, rule-based systems, and selection methods? The wide array of methods and algorithms available for processing SLI and SLO data presents its own challenges.

A compilation of any microservice monitoring, logging, or tracing records can constitute an SLI. Microservice monitoring encompasses three key components (Newman, 2015):

- **Metrics**: This component includes service telemetry data, such as CPU, memory, and storage I/O usage.
- **Traces:** This component aids in understanding the interactions between microservices.
- **Logs:** Each service generates events that are logged. These event logs are crucial for comprehending system activities.

Each monitoring component can possess indicators or features, which are further categorized into three groups (Podolskiy et al., 2019):

- **Manageable features**: These include resource allocation and other configurable settings.
- **Partially manageable features**: These are SLIs of other microservices situated on shared infrastructure such as Kubernetes nodes.

- **Unmanageable features**: These features are entirely dependent on the users of the microservice and represent user demand and usage patterns. Examples include requests per second or the type of data.

Regarding the use of SLIs as features for making data-driven decisions, Table 1.1. summarizes various studies and the SLIs they selected to support their proposed methods.

Table 1.1. SLIs used as features to drive the proposed model in different papers

SLI	**Studies**
CPU	(Delnat et al., 2018; Podolskiy et al., 2019; Rossi et al., 2019, 2020; Sampaio et al., 2019; Soualhia et al., 2019; Toka et al., 2021)
Disk usage	(Delnat et al., 2018; Soualhia et al., 2019)
Network utilization	(Rossi et al., 2020)
Requests per second	(Delnat et al., 2018; Rossi et al., 2020)

CPU utilization serves as the primary indicator of the workload's significance. This holds true for stateful services as well. The workload of some stateful applications, such as the Cassandra NoSQL database management system, can be CPU-bound (Truyen et al., 2019). Measuring disk usage is a straightforward approach for stateful services, as these services handle data that is persistently stored on disk. The impact of network utilization, particularly network latency, as analyzed in the research (Rossi et al., 2020), is significant in distributed architectures. The measurement of requests per second serves as a universal indicator of how effectively the service is configured. Additionally, this indicator may depend on unmanageable features of the service, such as the type of data being managed. Service Level Objectives (SLOs), or the thresholds for Service Level Indicators (SLIs), originate from user requirements for a certain level of performance.

Pre-processing and preparation of data is a crucial step for data-driven methods (Bodik et al., 2010; Iqbal et al., 2018; Podolskiy et al., 2019; Soualhia et al., 2019). Anomalies and poorly trained algorithms may lead to suboptimal results from prediction algorithms.

1.4. Data-Driven Methods Used to Increase or Predict Stateful Microservice Availability

Data-driven methods form the backbone of Artificial Intelligence and Machine Learning-based methods in Site Reliability Engineering by enabling the systematic collection, analysis, and operationalization of system performance data. Site Reliability Engineering teams establish a robust data foundation by gathering Service Level Indicators (SLIs), Service Level Objectives (SLOs), logs, and performance metrics from distributed systems, which serve as the input for AI/ML systems. Data is leveraged to enhance system reliability through automated anomaly detection, predictive failure analysis, intelligent alert correlation, and dynamic incident response while ensuring optimal resource allocation (Becker et al., 2021; Dang et al., 2019; Lyu et al., 2021; Zhang et al., 2025).

1.4.1. Resource Optimization

Autonomic vertical scaling proves useful in densely packed containerized cloud environments where adding more nodes or instances is not feasible, making horizontal scaling impractical. Podolskiy, Mayo, Koey et al. proposed a method for deriving SLO-compliant resource allocation for containerized applications, using performance models and single-objective and multi-objective optimization techniques (Podolskiy et al., 2019).

Among the three regression algorithms, linear regression, lasso regression, and random forest, the Lasso model was deemed the most suitable due to its achieved R2 coefficient of determination and its simplicity compared to linear regression, despite the latter having similar R^2 performance. Further, multiple lasso regression models were evaluated, including independent models to predict individual SLIs, application-wise models to predict SLO compliance for entire applications, SLI-wise models to predict an SLI for all applications, and all-targets models to predict SLOs for all applications. The results indicated that a degree of 2 is sufficient for SLI prediction.

Analysis of the 99%-tile throughput and response time distributions for all three applications revealed that anomalies constitute up to 8.2% of all observations. The impact of anomalies was evaluated by removing fractions of anomalies and re-evaluating the R^2 score. For instance, removing 11% of anomalies optimized the R^2 score.

Two validation tests were conducted: a preliminary test to obtain SLIs for use as features in prediction modules that continuously allocate resources, and an evaluation test to acquire SLIs used in the Lasso regression model to predict SLIs for continuous constrained optimization and limited brute force search to model the desired SLOs. Validation results showed that SLOs were violated only twice

in 16 trials. Thus, the proposed performance modeling technique was deemed effective.

Placing Pods across distributed nodes presents a challenge that must be addressed during Pod scheduling. To address this challenge, F. Rossi, V. Cardellini, F. Lo Presto, and M. Nardelli proposed identifying the relationship between application and system metrics (Rossi et al., 2020).

In their research, the authors applied a reinforcement learning solution that, based on experience, learns the most suitable scaling policy. They introduce gekube, an orchestration tool for Kubernetes.

One challenge in reinforcement learning is finding the optimal balance between exploitation (using effective actions) and exploration (searching for effective actions). During the analysis stage, the reinforcement learning agent assesses the application's state and updates the expected long-term cost (Q-function). At the planning stage, the Replication Manager utilizes the reinforcement learning agent to determine the appropriate scaling action.

In ε-greedy selection, the reinforcement learning agent, with a probability of ε, chooses an exploration action to enhance knowledge. With a probability of 1-ε, the reinforcement learning agent selects the best-known action.

1.4.2. Machine Learning for Fault Prediction

Aiming to predict faults in distributed Kubernetes-based edge cloud environments, M. Soualhia, C. Fu, and F. Khomh suggested employing machine learning techniques for fault detection and neural networks for fault prediction (Soualhia et al., 2019). Support Vector Machine, Random Forest, and Neural Network models can detect non-fatal disk and CPU faults with an F1-Score exceeding 95%. Long Short-Term Memory and Convolutional Neural Networks can accurately predict faults in more than 85% of cases.

1.4.3. Workload Prediction

Machine learning algorithms are used to predict workload with the goal of guaranteeing the quality of the provided services and reducing the number of SLO violations. In their work, Abdullah et al. proposed a regression-based dynamic autoscaler for containerized workloads. It allowed for increasing the number of processed requests by 1.09 and reducing the number of SLO violations by 5.17 (Abdullah et al., 2022). Dang-Quang and Myungsik Yoo proposed a system with a Monitor–Analyze–Plan–Execute loop utilizing Bidirectional Long Short-term Memory to predict future HTTP workloads. It has been shown to outperform the native Kubernetes horizontal pod autoscaler and, compared to Long Short-term

Memory, reduced the number of prediction errors by up to 20% (Dang-Quang & Yoo, 2021).

The approach to workload prediction and resource usage optimization can be on the architectural level, as shown by Dakić et al. The proposed architecture is dedicated to improving HPC workload scheduling on Kubernetes based on workload prediction (Dakić et al., 2024). The proposed approach is based on the evaluation and prediction of the workload and current resources. Once the aforementioned factors are considered, the workload is scheduled on the most appropriate Kubernetes node.

In their work, Rubak & Taheri utilize Support Vector Machine and Linear Regression algorithms to improve the native Kubernetes HPA (Rubak & Taheri, 2023).

1.5. Rule-Based Methods and Techniques to Improve Availability of Stateful Microservices in Orchestrated Container Systems

1.5.1. Evaluating Efficiency of Availability Mechanisms

Container orchestrators incorporate mechanisms to maintain the high availability of the services they manage. However, due to their flexibility, these orchestrators can be enhanced or extended to perform custom actions. Service Level Indicators (SLIs) form the foundation for assessing the effectiveness of the improved mechanism. At this stage, various indicators are used for evaluation, including served requests per second (Rossi et al., 2020), CPU and memory utilization (Y. Yang & Chen, 2019), and the duration of outages (Abdollahi Vayghan et al., 2019).

1.5.2. Custom Scheduler for Kubernetes

The ge-kube, introduced by F. Rossi, V. Cardellini, F. Lo Presto, and M. Nardelli (Rossi et al., 2020), employs a custom Kubernetes scheduler. In their network-aware Pod placement solution, this custom scheduler captures a snapshot of the Kubernetes cluster and sends it to the Deployment Service. The Deployment Service then decides on Pod placement based on available resources and network delays. This decision is returned to the custom scheduler, which subsequently determines where to place the Pod.

In an experiment designed to validate results, the authors of ge-kube utilized network-aware, first-fit, and round-robin heuristic algorithms. They compared

these algorithms to the default Kubernetes scheduler and solved the Pod placement problem using integer linear programming. A scaled-out Redis cluster was employed to test these algorithms. The number of requests per second was measured to assess the optimal Pod placement in geographically distributed nodes.

The first-fit, round-robin, and default Kubernetes schedulers achieved similar performance, handling between 15,000 and 18,000 requests per second. In contrast, Pods placed by the network-aware algorithm achieved an average of 44,000 requests per second.

Y. Yang and L. Chen (Y. Yang & Chen, 2019) developed a custom scheduler. In their proposed three-module architecture, the third module handles dynamic resource scheduling and uses prediction data to enhance resource utilization. Since their research aimed to optimize the Pod scheduling mechanism, the evaluation of the proposed model focused on measuring how evenly CPU and memory utilization are distributed across Kubernetes nodes.

Custom schedulers facilitate fault-tolerant, redundant placement, ensuring that a stateful microservice within an orchestrated container system operates as intended within the defined SLO boundaries.

1.5.3. State Controller for Stateful Kubernetes Services

Aiming to enhance the availability of stateful services, Abdollahi Vayghan, Saied, Toeroe, and Khendek proposed a State Controller for Kubernetes (Abdollahi Vayghan et al., 2019). The proposed State Controller integrates the concept of active and standby states into Kubernetes to enhance the availability of stateful microservice applications. The idea is to assign labels to pods, indicating whether they are in an active or standby state. A service that exposes an application uses only the pods that are labeled as active.

The State Controller monitors the state of each pod; if a pod with an active status fails, another pod is assigned the active status. The pod with the newly assigned active status then becomes the entry point.

The State Controller can be integrated with both the StatefulSet and Deployment Controller. In the case of StatefulSet, the State Controller creates two pods with separate PersistentVolumes. It creates two services: one that exposes the active pod to clients, and another that replicates data to the standby pod(s). In the event of a failure, the State Controller switches the pods, and the service resumes the process from the last stored state.

With the Deployment Controller, the State Controller deploys pod replicas that share a PersistentVolume but have separate storage areas for each pod. In the event of a failure, the State Controller switches the pods, and the service resumes the process from the last stored state.

The State Controller was evaluated using three service outage scenarios: due to container process failure, pod process failure, and node failure. Additionally, OpenSAF, a middleware that implements the Availability Management Framework, was evaluated. Kubernetes, the State Controller, and OpenSAF handled failures differently. The proposed State Controller can increase service reliability by 55% to 92% compared to the built-in Kubernetes capabilities.

Table 1.2. Comparison of different controllers in terms of outage time

Outage type	**Outage duration in seconds**		
	Kubernetes	***State Controller***	***OpenSAF***
App Container Failure	2.2	1.2	0.2
Pod Process Failure	2.1	0.7	3.3
Node Reboot	164.5	2.9	3.3

1.5.4. Shared Components

High availability of stateful microservices can be improved by sharing storage or memory. While traditional approaches involve sharing data between two or more nodes in a cluster, novel techniques suggest sharing storage or memory rather than data.

The SHADOW system presented by Kim et al. simplifies RDBMS by offloading data replication to the storage layer rather than the database layer (J. Kim et al., 2015). The prototype system, based on a persistent disk shared by two PostgreSQL nodes, outperformed the native synchronous replication of PostgreSQL. A similar approach is proposed by Hong et al.; the same persistent disk in a Ceph cluster is shared between two or more Docker containers (Hong et al., 2019). The results show that shared storage has improved the availability and recoverability of stateful Docker containers.

The Active-Memory Replication method, proposed by Zamanian et al., is based on replicating the data directly between the memory of two or more database nodes, bypassing the storage layer altogether (Zamanian et al., 2018).

1.6. Maintenance of Low Latency Stateful Microservices

1.6.1. Maintenance of Stateful Microservices

Modern microservices are designed with resiliency in mind. Various components of microservice applications are engineered to manage failures rather than prevent them. Different techniques are employed to ensure fault tolerance of microservices and entire systems, including graceful degradation, decoupling, retries, caching, and redundancy, all of which help ensure the reliability of stateful microservices (Pereira & Bruce, 2019).

According to ISO 25010, availability is defined as *"the degree to which a system, product, or component is operational and accessible when required for use*" (ISO/IEC, 2011). Another approach is to measure availability based on what users can actually experience. The availability metric should capture both the duration of an outage and the number of failed requests (Hauer et al., 2020).

Availability is impacted by maintenance time if the microservice is considered a repairable item. Therefore, maintainability is a crucial aspect of ensuring high availability (O'Connor & Kleyner, 2012). Systems can undergo maintenance to either correct an error and return to an operational state or to prevent an error and maintain their operational state. Additionally, systems may be put into maintenance when rolling out new features that require downtime.

Before assessing availability, an expected level of performance must be established. This is defined by Service Level Agreements (SLAs), Service Level Objectives (SLOs), and Service Level Indicators (SLIs). Once an SLO is set, predictions can be made regarding whether the availability SLO would be breached during maintenance (Beyer et al., 2016).

Maintenance is relatively straightforward in orchestrated container systems compared to other deployment types. Orchestrated container systems, such as Kubernetes, offer a framework for running, managing, and maintaining containerized applications. Despite this, maintaining a stateful workload can be costly. Although lightweight, containers still have similarities to VMs, and certain limitations still apply. Thus, state management and maintenance remain challenging. This caveat applies to database management systems as well.

Although container orchestrator systems can abstract some maintenance tasks performed by human operators, particular aspects of state management remain challenging. Consider Kubernetes as an example: a relaunched Pod, depending on the state of its PersistentVolume, must remain in an "*initializing*" state until replication catches up. If the PersistentVolume is empty, all data must be replicated to that Pod (Kubernetes, 2025b).

One method to establish a highly available relational database management system is multi-Primary replication (Seybold et al., 2017). The failure of a single node in a multi-Primary replicated database does not impact overall availability. However, routing requests to such a replica set can be challenging; in the event of a node failure, some requests may fail while new requests are redirected to one of the healthy nodes.

The usual practice is to gracefully close all database connections from the application side when maintenance must be performed. However, in this chapter, given the high degree of maintainability of container-based deployment, the focus is on availability during maintenance under a load.

Orchestrated container systems offer options for service discovery and load balancing. Yet, routing requests to stateful multi-node services is challenging due to specific internal configurations, such as determining which node an application should connect to for consistent data reading (Kleppmann, 2017).

Load balancing and request routing are crucial components for ensuring high availability in database systems. This is the objective of the LABAREDA service, introduced by Marinho et al., which aims to distribute the load across replicated database nodes in the cloud to maintain performance within the Service Level Objectives (SLO). The service forecasts the performance of each database node in a replicated cluster and determines whether additional nodes are required to maintain performance within the SLO (Marinho et al., 2018).

Moreover, advanced request routing algorithms like Hihooi enable higher availability and elasticity (Georgiou et al., 2019). The routing algorithm of Hihooi directs incoming requests to consistent replicas, thereby enhancing load balancing and scalability. Load prediction can also assist with the proper sizing and scheduling of database deployments (Cao et al., 2018; Jindal et al., 2019).

A benefit of proper sizing is not only an optimal resource consumption but also a reduced need for maintenance to allocate varying amounts of computing resources. King Louie, a reproducible availability framework, enables the validation of database management system deployments by considering factors such as replication type, failure severity, architecture, and injecting failures (Seybold et al., 2020).

1.6.2. Low Latency Stateful Microservices

Applications use database systems to store and manage their data. This system comprises numerous objects, including storage mechanisms, query and programming languages, drivers, logic, and mappings between an application and its databases. One crucial object is the connection.

Establishing a database connection necessitates numerous resources and actions, such as a back-and-forth authentication process between client and server,

authorization, memory allocation, etc. The object-relational persistence framework proposes that a connection should serve a single transaction and then be terminated. However, because creating a connection is time-consuming, connection pooling supports short-lived connections. A connection pool allows for the reuse of established connections by returning them to the pool after use (Hohenstein et al., 2009). Reusing connections from a pool reduces request latency as it eliminates the need to perform the time-consuming process of connection establishment.

When a client connects to a database server process, the server either creates a new thread or assigns an idle one. The server process, specifically the threads it allocates, executes requests from connected clients. A thread executes a query sent by its allocated connection. Subsequently, the thread either closes the socket or, for reused connections, keeps the socket open and waits for additional incoming queries (Sippu & Soisalon-Soininen, 2017).

As the socket waits for more incoming queries, the connection stays open. Therefore, without explicitly checking its status on either the database or client side, it remains unclear whether a query is being executed. Closing a socket without knowing its active status may result in a failed query.

1.6.3. Replicated Stateful Microservices

Even with the rise of non-relational databases (Brewer, 2012) and distributed file systems (Thanh et al., 2008), relational database management systems continue to serve as the foundation for many software applications, whether in monolithic or microservices-based architectures. Database availability is ensured by distributing data across multiple nodes. Consistency among distributed nodes is maintained through data replication. There are two replication strategies (Seybold et al., 2017; Tamer & Patrick Valduriez, n.d.):

1. **Single-Primary** replication. This strategy involves directing all writes to a single node in a distributed database, which is then replicated to the remaining nodes. This setup reduces the risk of consistency conflicts. However, if the primary node fails, time is required to elect a new primary to handle writes. Single-Primary failover is not trivial and involves additional actions such as reconfiguration of distributed database nodes to change the replication source, pausing application traffic, reconfiguring application clients, etc. (Campbell & Majors, 2017; Deshpande, 2019).

2. **Multi-Primary** replication. This strategy allows multiple nodes in a distributed database to accept writes and distribute them across the remaining nodes. In case of node failure, any other node in the multi-Primary distributed database can take over. Additional load balancing solutions

or proxies can further minimize downtime and mask incompleteness in a multi-Primary distributed database during disasters. A significant drawback is the need for conflict resolution, as writes from multiple sources increase the risk of conflicts that need resolution. Conversely, writes can be directed to one node of the multi-Primary distributed database, creating a pseudo-single-primary setup (Campbell & Majors, 2017).

Since multi-Primary replication allows any cluster node to handle writes, transitioning a connection from one node to another is feasible. Connections can be gradually shifted to other database nodes while maintaining constant throughput. However, conflict resolution may impact response time due to multiple cluster nodes processing write requests.

Distributed database node outages can be unexpected (e.g., due to failure) or expected (e.g., for maintenance). The mechanism for taking down a node in a distributed database depends on the replication strategy. In single-Primary setups, any node except the primary can be removed from the cluster at any time. Upon rejoining the cluster, it must catch up on changes made since its last known transaction. Taking down the primary node in a single-Primary setup is non-trivial. The Primary node must step down, and a new Primary must be elected before it can be taken offline. Stepping down necessitates draining connections from the outgoing Primary node, and clients must then be reconfigured to use the newly elected Primary node (Kleppmann, 2017).

An additional challenge involves routing requests to various nodes of a distributed database cluster. There are two high-level approaches for routing requests (Kleppmann, 2017):

- Clients can send requests to any cluster node, which either processes or forwards the request to another appropriate node.
- Clients send requests via a routing level, where a decision is made about which node the client should connect to.

Regardless of the routing approach, the challenge of determining where to route the request persists. This challenge is commonly referred to as service discovery (Kleppmann, 2017).

Request routing and load balancing are crucial components of ensuring high availability for database systems. Marinho et al. have introduced a LABAREDA service for predictive and elastic load balancing. Its purpose is to predict Service Level Agreement (SLA) violations using prediction models (Marinho et al., 2018). The service employs Autoregressive Integrated Moving Average and Exponential Moving Average prediction models to predict and balance workloads against replicated databases. Although the proposed service can predict SLA violations and take necessary actions to maintain performance metrics within SLO, its impact on database availability during failover is limited.

Georgiou et al. introduced Hihooi, a replication-based middleware system featuring a novel routing algorithm (Georgiou et al., 2019). The system is designed to provide high scalability, consistency, and elasticity for transactional databases. Hihooi acts as a middleware system positioned between database engines and applications. It intercepts database-bound requests, and its routing algorithms direct them to the most consistent replicas. Based on transaction inspection, the algorithm avoids read request delays by directing them to replicas with consistent data. The method proposed in this research is also based on transaction content inspection. However, this inspection occurs at the database level rather than at the Transaction manager level, as in Hihooi.

The Taurus database system, proposed by Depoutovitch et al., leverages a different replication algorithm that allows for increasing the speed of writes while ensuring 100% availability (Depoutovitch et al., 2020). The proposed system consists of multiple low-dependent components. Data and Log stores are separated on the cluster nodes to distribute the load. Writes are considered durable when the minimal replication factor for data is reached.

In multi-Primary setups, connections must be drained from the node to be removed from the cluster. Applications should be unable to connect to the node being removed. Re-adding a node to the cluster allows applications to reconnect to it. Various techniques allow or disallow applications to connect to a node, including changing connection settings in applications, modifying Domain Name System (DNS) records, and/or altering the node pool in a proxy or load balancer layer (Campbell & Majors, 2017).

In both cases, the cluster becomes out of sync, necessitating data replication to nodes rejoining or being added to the distributed database cluster. Out-of-sync nodes must catch up for the database cluster to regain consistency. Taking a distributed database node offline for maintenance impacts the database cluster. The latency between replicated data on distributed database nodes is termed Replication lag (Campbell & Majors, 2017). Besides indicating the degree of inconsistency in a database cluster, replication lag may lead to negative user experiences, such as inconsistent reads and/or writes (Kleppmann, 2017; Petrov, 2019). Thus, taking a node offline in a cluster built for reliability may impact availability. Reducing the time to return to the cluster is crucial for minimizing replication lag's impact.

In other cases, data must be fully copied onto a node rejoining the distributed database cluster, for instance, due to replication lag exceeding the set Service Level Objective (SLO) (Campbell & Majors, 2017). Copying data from one node to another can take considerable time in larger databases. There is a risk of inconsistent copied data if the database is actively written to (Kleppmann, 2017). Additionally, copying data can degrade a distributed database cluster's performance, posing a risk to the SLO.

1.6.4. Retrying of Requests During Maintenance of Low Latency Stateful Microservice

Deploying certain database changes, such as memory settings or minor software version updates, necessitates a restart. Replication enables the service to remain online and process requests. However, pooled connections pose a challenge when they must be drained from a database node. There are two approaches for handling established connections to a database node: forcefully terminating them or draining them by reconfiguring the application (Campbell & Majors, 2017).

In the first case, connections are terminated, and the operator must accept the impact on availability. In the second case, the operator must modify the client application, for example, change the data source or reload the configuration. Reconfiguring client applications is a sound approach to draining connections from a database node, as it avoids impacting availability. However, it involves more components and increases microservice coupling.

Failing request causes can be categorized into two types: transient and persistent. Transient failures stem from non-permanent conditions, such as interim connection loss, reconfiguration, temporary resource exhaustion, etc. Persistent failures result from conditions that require human intervention to resolve (Mauri et al., 2021).

Microsoft recommends configuring retries to manage different types of exceptions. Handling exceptions based on their cause improves service availability by mitigating transient failures. When properly configured, retries minimize the impact of self-correcting faults on application availability (Coriani, 2021; Mauri et al., 2021).

There are three exception handling strategies (Microsoft Inc., 2022):

1. **Cancel** the failing request. If a failed request is due to a persistent condition, it should be canceled.
2. **Retry immediately**. If a failed request stems from a transient condition, it can be immediately retried. The failure condition, like network packet corruption, is unlikely to recur, making it safe to retry the request immediately.
3. **Retry after delay**. If a failed request results from a transient condition that resolves after some time, retry after a delay. The failure condition, such as a network connectivity issue or request throttling, requires time to be resolved. During this time, it prevents successful request execution. Thus, retries are issued after a delay.

Dai's work (Dai, 2012) demonstrates the importance of retries in error handling. Dai's Retryer (Dai, 2012) verifies whether requests should be retried against the database. It introduces client-side retry logic to determine if a failed request

should be retried and shows that a retry mechanism, albeit complex, can enhance database recovery following transient errors.

1.7. Burst Workloads in Stateful Microservices

Traditionally, a sudden and significant surge in service requests is referred to as a burst. Bursts are typically caused by unexpected events that lead to a sudden rise in user activity. Figure 1.5. illustrates an example of increased demand, a burst, on Michael Jackson's Wikipedia entry, observed both days before and after his death (Ali-Eldin et al., 2014). It is also worth noting that bursts can be caused by Denial of Service attacks (Ali-Eldin et al., 2014). Such workloads can lead to decreased service quality and violations of Service Level Objectives (SLOs), including availability loss.

A burst can be unpredictable in terms of volume and timing (Ali-Eldin et al., 2014). The two primary properties of a burst are its intensity and duration (Ali-Eldin et al., 2014; Bodik et al., 2010). Intensity refers to the number of requests, while duration refers to the length of time a service experiences the additional workload.

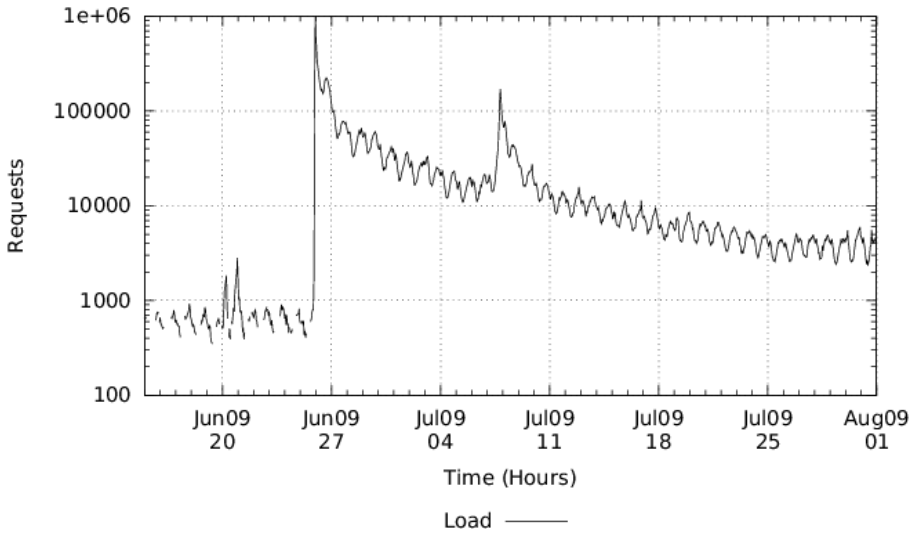


Fig. 1.5. Example of request patterns

When a stateful service is involved, a burst can exhibit additional properties, such as data spikes, including increased demand for specific data sets. For instance, in a partitioned database, a significant increase in demand for a particular data subset can be deemed a burst, even if the overall number of requests remains constant on the access layer (Bodik et al., 2010).

Various methods can measure burst workloads, such as calculating the standard deviation from the moving average, normalized entropy, or sample entropy (Ali-Eldin et al., 2014). As suggested by Woodruff et al., a burst is defined as a sequence of x requests arriving within a y timeframe of each other (Woodruff et al., 2019).

An application or service may scale up or employ other strategies to process the load if bursts are predictable or periodic. However, unpredictable workload bursts pose a challenge, as a system may struggle to operate effectively under stress.

Generally, two approaches can manage burst workloads: static or dynamic. The static approach is straightforward: allocate sufficient computing resources to handle potential burst workloads (Govardhana Miriyala Kannaiah, 2024; Rubak & Taheri, 2023; Shen & Chen, 2018). This approach is more costly and may lead to over-provisioning resources. Resources can be optimally allocated to microservices without affecting the latency of user requests (Baarzi & Kesidis, 2021). Baarzi and Kesidis improved resource allocation by 22% and request latency by 20%. However, the unpredictability of burst workloads poses risks regardless of how optimal the resource allocation is.

The dynamic approach is data-driven: bursts are predicted, and the system scales up accordingly to manage the load. This approach requires an elastic system and various mechanisms to predict upcoming bursts (Iqbal et al., 2018; Lassnig et al., 2010; Xie et al., 2024). Lassnig et al. present an averages-based model that estimates the probability of a burst occurring. However, according to the authors, the model requires optimization (Lassnig et al., 2010). Iqbal et al. utilized the expectation-maximization method and Scale-Weighted *K*-means to partition service request logs to capture workload characteristics. They used the distribution of these characteristics to compute a probability vector describing the distribution of incoming requests. This information was then used to proactively auto-scale applications (Iqbal et al., 2018).

PBScaler, proposed by Xie et al., collects real-time performance metrics of applications and constructs a correlation graph between microservices. The proposed scaling method achieves precision up to 0.92 using the Random Forest algorithm, and TopoRank, a random walk algorithm, identifies potential bottlenecks (Xie et al., 2024).

A burst can exhaust various system resources, including the Central Processing Unit (CPU), memory, network bandwidth, disk throughput, etc. Insufficient computing power, slow networks, and low disk read speeds impact SLOs, but memory exhaustion has the most significant effect on stateful microservice availability. A container is terminated when it reaches its allocated memory limit (Kubernetes, 2025a). Relational database management systems, such as MySQL, use memory to cache data, query execution plans, monitor data, and store client connection buffers and thread stacks (Oracle, 2025; Percona LLC, 2025). Thus, during a burst, memory is exhausted by additional client connections and the increased demand for cached data.

In multi-Primary database management systems, all nodes are equal and can simultaneously process read and write requests. However, write scaling, which increases a system's capacity to handle more write operations, is not recommended, as improvements in write throughput are usually limited, even with only two writer nodes (Percona LLC, 2025). The write operation on a database node is processed, and only the state update is transferred to other nodes, eliminating the need to reprocess the write (Codership Ltd., 2025). If one node is designated for processing write requests, only its memory is used to cache processing-related data, such as execution plans.

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

The first chapter of the dissertation provides an overview of stateful microservices in an orchestrated container environment.
The following conclusions have been drawn:

1. Stateful and stateless microservices differ primarily in state management, which introduces challenges like data persistence, backups, failover handling, and maintaining consistency in distributed environments. In microservice architecture, each service has its own database, accessed via APIs, leading to potential consistency issues due to broken links, orphan states, or missing states when recovering from backups. The BAC theorem highlights the trade-off between availability and consistency in microservice backups. Container orchestration frameworks, such as Docker Swarm and Kubernetes, introduce performance overhead due to network and volume plugins. Stateful workloads in Kubernetes, managed by StatefulSets, require careful handling of persistent volumes and can experience performance degradation during scaling or rescheduling. SLIs and SLOs are essential for data-driven decision-making, with metrics like

CPU usage, disk usage, network utilization, and requests per second being key indicators. Effective data processing is crucial for leveraging SLIs and SLOs in machine learning and other algorithms to enhance microservice performance reliability and availability.

2. There are two kinds of methods to improve the availability of stateful microservices in orchestrated container environments: data-driven methods and ML-based methods. In relation to stateful microservice reliability and availability, such methods are used for optimal resource allocation, fault, and workload prediction. Despite the advancements, the accuracy seldom reaches 100%. The other kind of method is rule-based. Methods and techniques based on a set of rules are transparent in their outcomes. Although they are predictable, rules rarely handle every conceivable situation.
3. Connection to a stateful microservice is an important component of an application. Establishing a connection for each request toward a database requires several time- and resource-consuming steps. Thus, connections are saved in a pool to be reused. This approach positively impacts request latency. However, without explicitly checking the socket's status on either the database or client side, it is unclear whether a query is being executed, and prematurely closing the socket may result in a failed query.
4. Database availability is maintained by distributing data across multiple nodes, with consistency ensured through data replication using either single-Primary or multi-Primary strategies. Single-Primary replication directs all writes to a single node, reducing consistency conflicts but requiring time to elect a new primary node in case of failure. Multi-Primary replication allows multiple nodes to handle writes, minimizing downtime but requiring conflict resolution. Routing requests to various nodes can be direct by clients or through a routing level, with service discovery being a key challenge. Taking nodes offline for maintenance or failure impacts the cluster's consistency and availability, with replication lag causing potential performance degradation and user experience issues.
5. Modern microservices are designed with resiliency in mind, employing techniques like graceful degradation, decoupling, retries, caching, and redundancy to ensure fault tolerance and reliability, particularly for stateful microservices. Availability, defined by ISO 25010 as the degree to which a system is operational when required, is crucial and can be measured by the duration of outages and the number of failed requests. Maintainability is essential for high availability, with systems undergoing maintenance to correct errors or prevent them. SLAs, SLOs, and SLIs establish performance expectations and help predict potential SLO breaches during

maintenance. Proper sizing and scheduling of database deployments reduce maintenance needs and optimize resource consumption. Handling database connections during maintenance can impact availability, with options to forcefully terminate or drain connections. Retry mechanisms are crucial for managing transient failures and improving service availability, with strategies including canceling failing requests, immediate retries, or retries after a delay.

6. A burst refers to a sudden and significant surge in service requests, often caused by unexpected events or Denial of Service attacks, leading to decreased service quality and potential SLO violations. Bursts are characterized by their intensity (number of requests) and duration (length of time under increased workload), and can also involve data spikes in stateful services. Handling bursts can be approached statically by allocating sufficient resources, which can be costly, or dynamically by predicting bursts and scaling resources accordingly. Dynamic methods include using models to estimate burst probabilities and algorithms to identify bottlenecks. Bursts can exhaust system resources, with memory exhaustion having the most significant impact on stateful microservice availability. In multi-Primary database systems, while all nodes can process read and write requests, write scaling is not recommended due to limited improvements in write throughput.

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

1. Identify and evaluate factors impacting stateful microservice availability during failover.
2. Introduce a low-coupled method to improve low-latency stateful microservice availability during maintenance activities.
3. Propose and evaluate a rule-based method to improve stateful microservice resilience to sudden and significant workload increases.

2

Identification of Factors Impacting Stateful Microservice Availability During Failover

This chapter explores the reliability and availability of stateful microservices during failover events in orchestrated container environments. The research evaluates various factors affecting service availability, including load intensity, load balancer type, and connection type. It demonstrates that SQL-aware load balancers, such as ProxySQL, provide higher availability than traditional ones, such as HAProxy. The research highlights methodologies to improve failover mechanisms and reduce downtime, contributing to more resilient and efficient cloud-based applications.

The details of the analysis and its results presented in this chapter were published in the author's publication (Pakrijauskas & Mažeika, 2022b).

2.1. Evaluated Factors

While the maintainability of stateful microservices in orchestrated container environments is high, their maintenance has a more significant impact on availability compared to stateless microservices. Having a clear understanding of this impact

allows for setting more accurate Service Level Objectives (SLOs) and availability budgets. Apart from the managed features of a stateful microservice, such as resource allocation, intricate data storage settings, or replication strategies, there are unmanaged features as well. When considering a stateful microservice deployed to an orchestrated container system, the primary unmanaged features that may have an impact on availability are the following:

- **Connection type.** Pooled connections allow for a decrease in latency, though at a higher complexity compared to single-user reopened connections.
- **Workload intensity.** The number of requests depends on the client of a stateful microservice.
- **OSI level of a load balancer.** Replication is the usual approach to ensure high availability of a stateful microservice. However, it adds the necessity of service discovery, request routing, and load balancing. It may or may not be managed by the same team that is responsible for a stateful microservice.
- **Request duration.** Query complexity depends on the client of a stateful microservice.

2.2. Investigated Architecture

This research investigated the availability of a multi-Primary database during maintenance operations within an orchestrated container environment. Due to its significant market share, Kubernetes was chosen as the container orchestrator system for this research. According to the 2021 annual survey by the Cloud Native Computing Foundation, 96% of the surveyed organizations are using or evaluating Kubernetes (Cloud Native Computing Foundation, 2021).

A load balancer appliance is used to route requests to a multi-Primary database cluster. This setup helps minimize disruption, with the takeover time being limited to rerouting requests to another node in the database cluster (Seybold et al., 2017). Two types of load balancers are utilized in this research: SQL-aware and TCP load balancers.

A multi-Primary database cluster operates on three Pods within a StatefulSet. Access to it is provided through a load balancer appliance. The investigated architecture is illustrated in Figure 2.1.

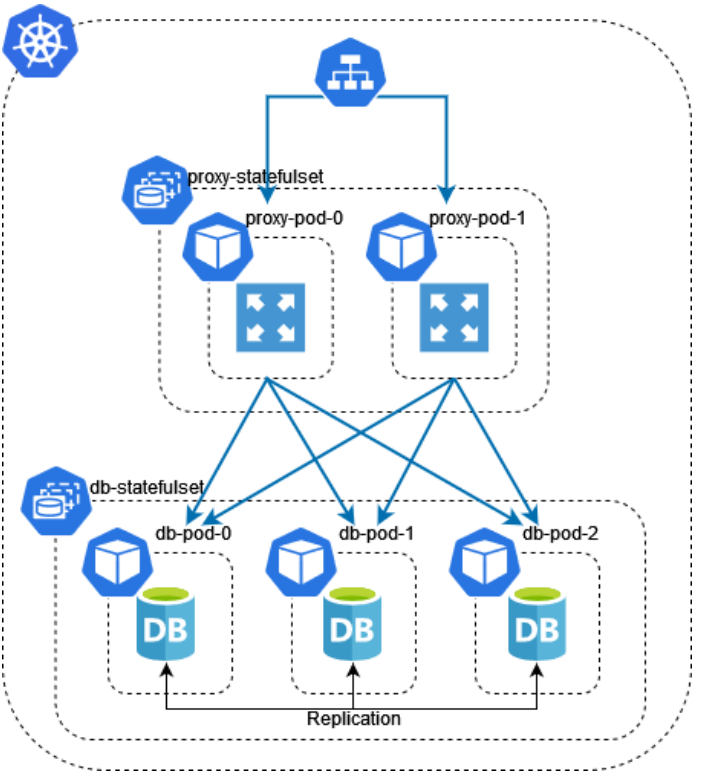


Fig. 2.1. Investigated architecture of a multi-Primary database cluster

StatefulSet update is performed in a rolling fashion (Kubernetes, 2021d): Pods are reinitialized one-by-one, starting with the one with the highest ordinal number to match the updated manifest. A pod is marked as inaccessible in the load balancer before performing its reinitialization.

2.3. Experiment Setup

The experiment was conducted on a three-node Kubernetes cluster hosted on Google Cloud Platform (GCP). All three nodes were identical, consisting of E2 cost-optimized instances equipped with 4 CPUs and 6GB of memory, each with a 50GB standard persistent disk (PD) attached.

The MySQL Galera cluster was selected because this database management system supports multi-Primary replication. Additionally, it is a SQL-aware load balancer (ProxySQL, 2026), whereas HAProxy functions as a TCP load balancer

(HAProxy Technologies, 2026). Both load balancers integrate well with the MySQL Galera Cluster.

The versions and resource limits of MySQL, ProxySQL, and HAProxy are displayed in Table 2.1. The experiment environment was deployed using a Kubernetes Operator for MySQL Galera Cluster. Thus, the software versions available are limited to those provided by the Operator (Percona LLC, 2021).

Table 2.1. Software Versions and Resource Allocation

Software and Version	**Number of Pods**	**CPU Limit**	**Memory Limit**
MySQL Galera Cluster 8.0.25	3	600m	1024MB
ProxySQL 2.0.18	2	300m	256MB
HAProxy 2.3.10	2	300m	256MB

Pod reinitialization will be triggered by changing the CPU limit of the MySQL Galera Cluster StatefulSet by 1, from 600 to 601 mCPUs.

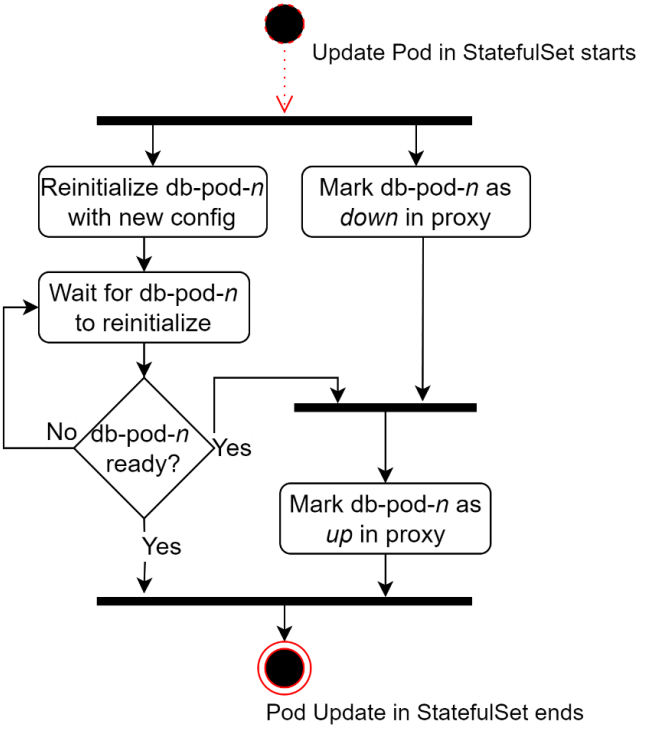


Fig. 2.2. Pod update in relation to the load balancer

Availability is assessed using probe requests. A probe request simulates a SQL query executing over a specified duration and executes a stored procedure against the database. The behavior of the probe is illustrated in Figure 2.3. Compared to existing workload generation and benchmarking tools, such as SysBench or YCSB, the custom probe requests allow for different logging and error handling techniques, which would be a challenge to customize in the existing tools.

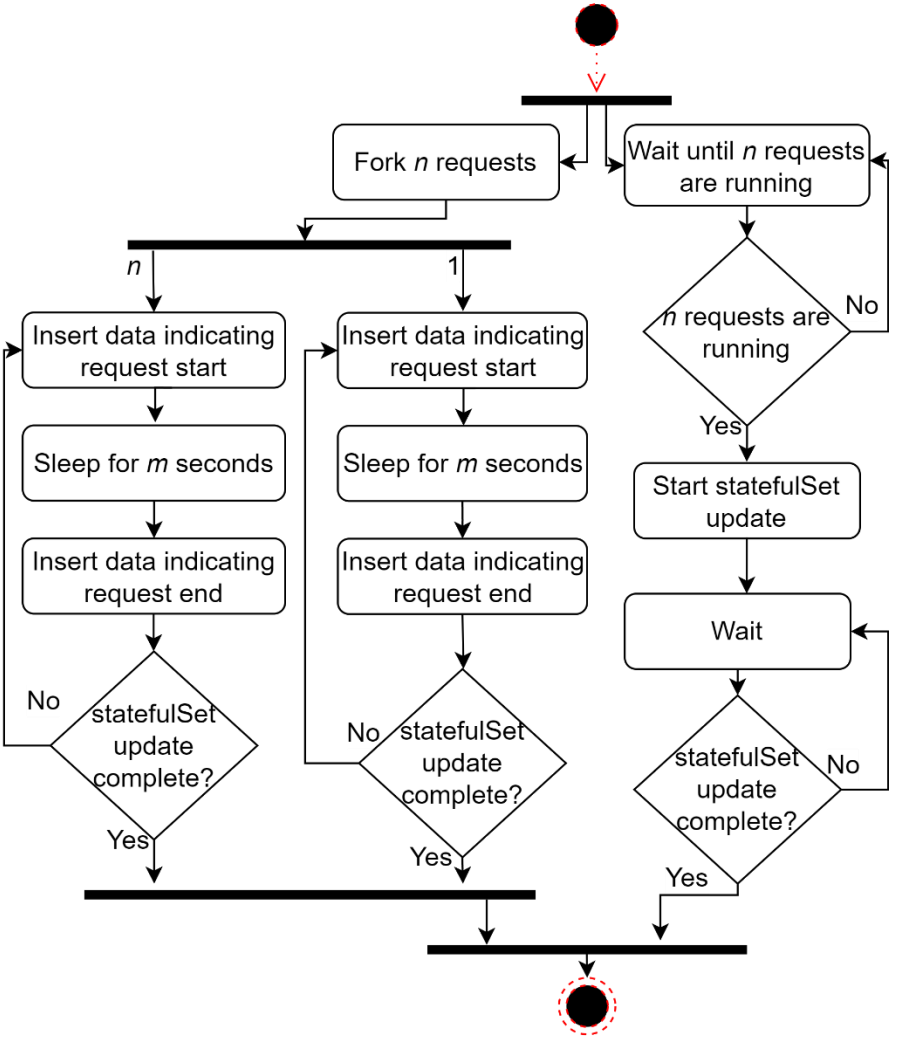


Fig. 2.3. Pod update in relation to the load balancer

The probe requests vary in duration, i.e., 0.5, 1, and 2 seconds, and in degrees of parallelism. Parallel probes were initiated every 0.33 seconds. The number of parallel probes is 10, 25, 50, 100, and 200. Probes establish connections to the database in two ways: either by creating a new connection for each request, reopening a connection, or by reusing a pooled connection. Creating and closing connections for each request incurs additional overhead (Liu, 2012).

Connections to the database cluster are established using a Python script that employs the PyMySQL module for both reopened and pooled connections. Pooled connections are configured to restart upon failure.

During the setup of the experiment, it was observed that the time required to update the MySQL Galera Cluster StatefulSet varies and can range from 240 to 1200 seconds to reinitialize all three Pods. Evaluating availability based on the numbers of the total, successful and failed requests may be imprecise because the number of total and successful requests depends on the time needed to update the StatefulSet. Therefore, the availability of the database is measured as the ratio of the number of parallel requests to the number of failed requests during the takeover process.

$$\text{PCT of failed requests} = \frac{\text{number of failed requests} * 100}{\text{number of parellel requests}} \qquad (2.1.)$$

The configuration of the experiment is as follows:

- **Load Balancer Type:** SQL-Aware (ProxySQL), TCP Load Balancer (HAProxy).
- **Connection Type:** Pooled, Reopened.
- **Request Duration (s):** 0.5, 1, 2.
- **Number of** Parallel **Requests:** 10, 25, 50, 100, 200.

2.4. Results of the Experiment

As anticipated, several requests failed during the failover process. The diagrams in Figures 2.4 and 2.5 illustrate the percentage of parallel requests that failed during a single failover operation. ProxySQL, which is SQL-aware, demonstrated a higher level of availability during the failover of MySQL Galera Cluster nodes.

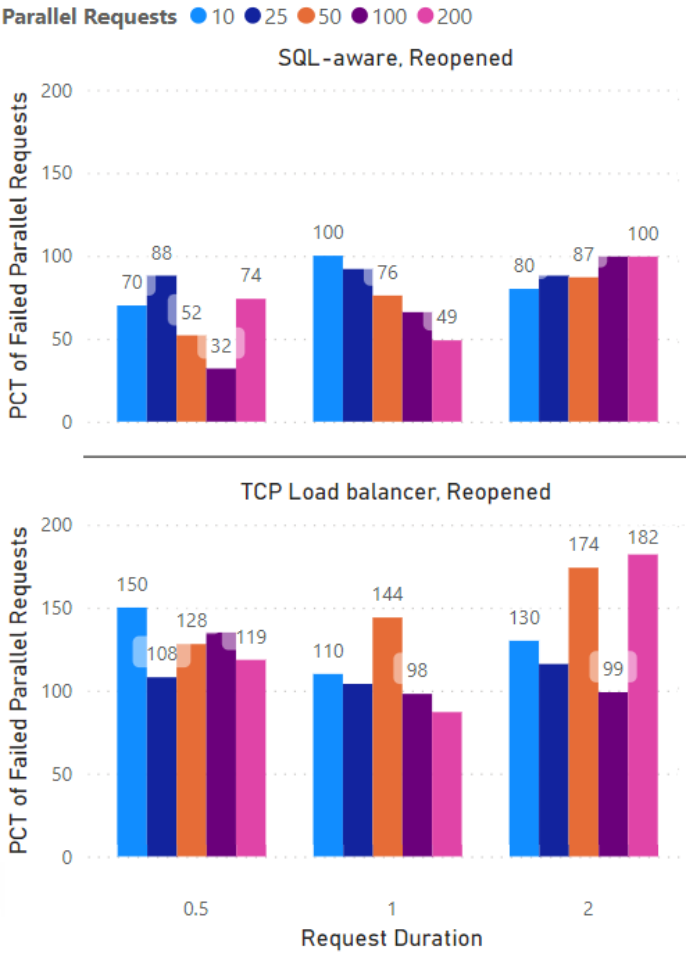


Fig. 2.4. Dependency of failed parallel requests on request duration and request number for reopened connections for both types of load balancers

It is important to note that HAProxy has a higher takeover time and consequently a greater number of failed requests, which is attributed to its method of detecting the status of a database node. The graceful failover mechanism of HAProxy differs from that of ProxySQL. Therefore, ProxySQL recognizes when a switch is occurring and can transfer requests to another database node with a lower failure rate.

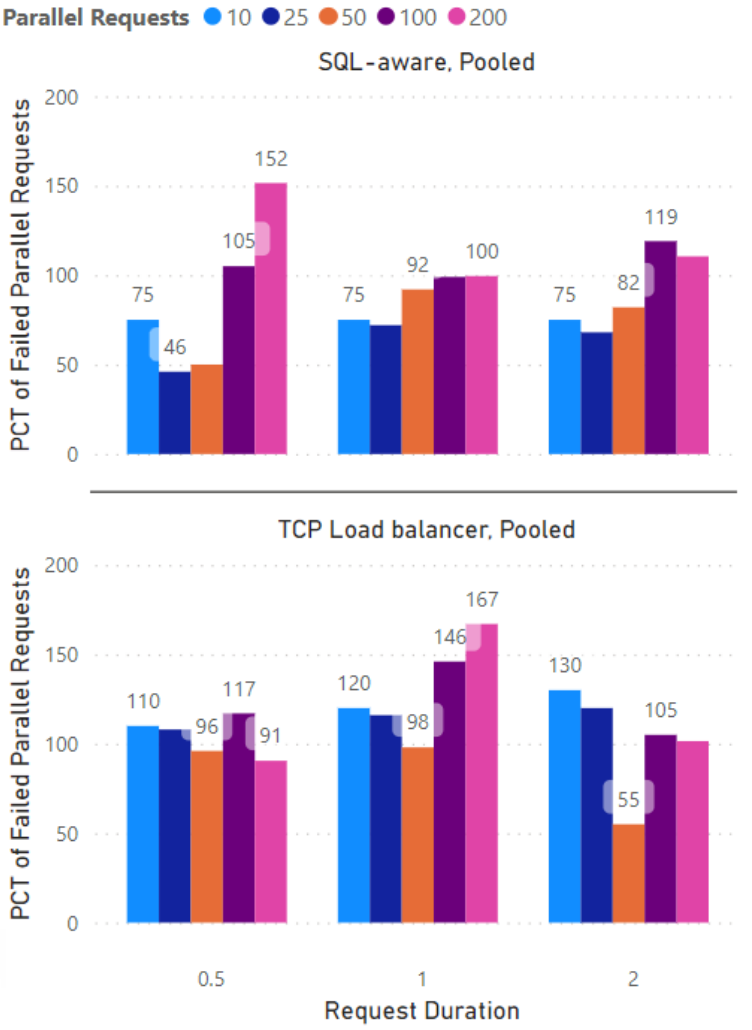


Fig. 2.5. Dependency of failed parallel requests on request duration and request number for pooled connections for both types of load balancers

Another factor influencing availability is the type of connection to the database. A reopened connection must undergo the authentication and authorization cycle once more. Therefore, ProxySQL exhibited a lower failure rate, particularly for short requests. Regarding the combination of pooled connections and ProxySQL, established and active connections were terminated, leading to a higher failure rate compared to reopened connections. As seen in Figure 2.6, the average percentage of failed requests decreased from 126% to 112% for HAProxy

and rose from 77% to 94% for ProxySQL across all request durations and numbers of parallel requests.

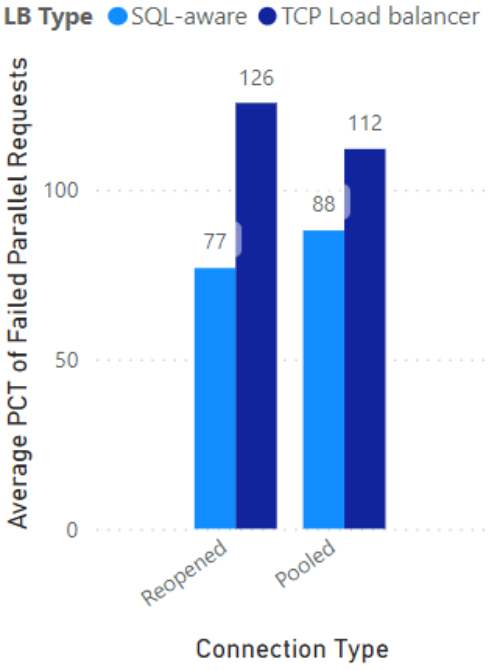


Fig. 2.6. Dependency of the average failed request percentage on connection type for both types of load balancers

The duration of a request also affects availability. As shown in Figure 2.7, the failure rate rises with increasing request duration. The most significant increase is observed from 74% to 91% when using ProxySQL, a SQL-aware load balancer. Shorter requests have a lower likelihood of being terminated mid-transaction.

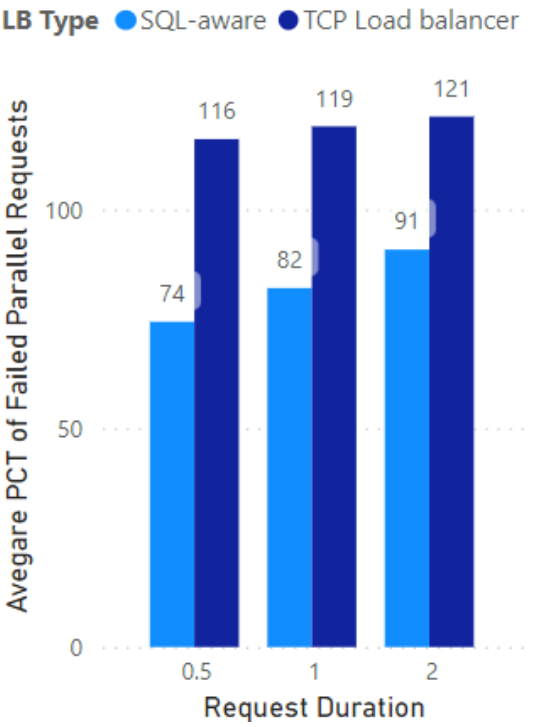


Fig. 2.7. Dependency of the average failed request percentage on request duration for both types of load balancers

The load, represented by parallel requests, has a more significant impact when ProxySQL is employed as the load balancer. When using ProxySQL as the load balancer, the average failure rate ranges from 73% to 97%. For HAProxy, the average failure rate ranges from 112% to 125%. As the comparative benchmark between ProxySQL and HAProxy indicates, HAProxy maintains more consistent performance across a wide range of loads (Tusa, 2021). Therefore, the impact of the number of parallel requests on the failure rate is less significant with HAProxy.

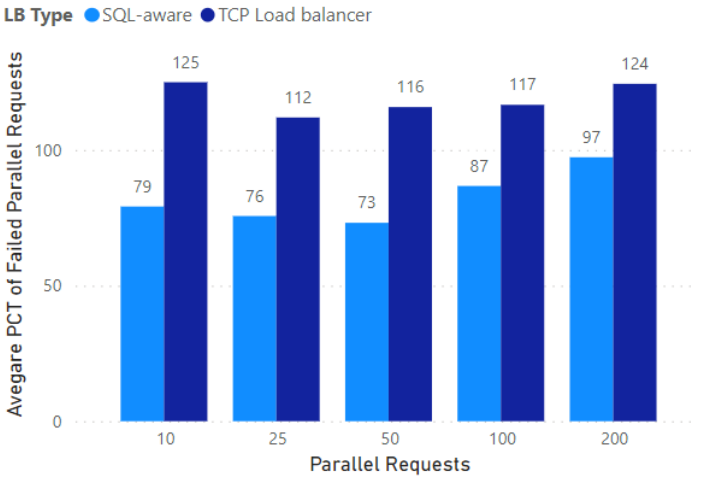


Fig. 2.8. Dependency of the average failed request percentage on the number of parallel requests for both load balancers

The relationship between connection type and load is illustrated in Figure 2.9. The failure rate remains below 100% (ranging from 79% to 98%) with a load of up to 50 parallel requests when pooled connections are utilized. However, the average failure rate increases with the load: it reaches 115% for 100 parallel requests and 120% for 200 parallel requests. The average failure rate for reopened connections varies between 88% and 110%.

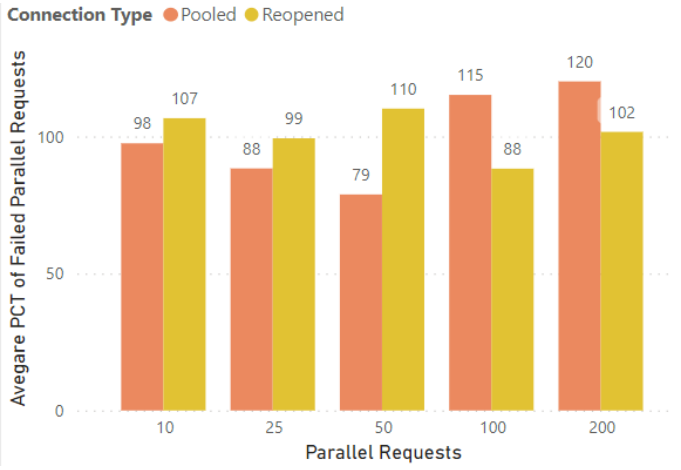


Fig. 2.9. Dependency of the average failed request percentage on the number of parallel requests for reopened and reused connections

Although the failure rate across various experiment dimensions spans a wide range, from 32% to 182%, a trend is evident: ProxySQL, a SQL-aware load balancer, ensures higher availability during failover, with an average failure rate of up to 82%. However, HAProxy, a TCP load balancer, has its own advantages: it is lighter-weight due to its stateless nature and demonstrates better performance under high read-heavy loads (Tusa, 2021). Additionally, changing the status of database nodes in ProxySQL to *"OFFLINE_SOFT"* should enhance availability during failover, but the duration of maintenance would depend on the load.

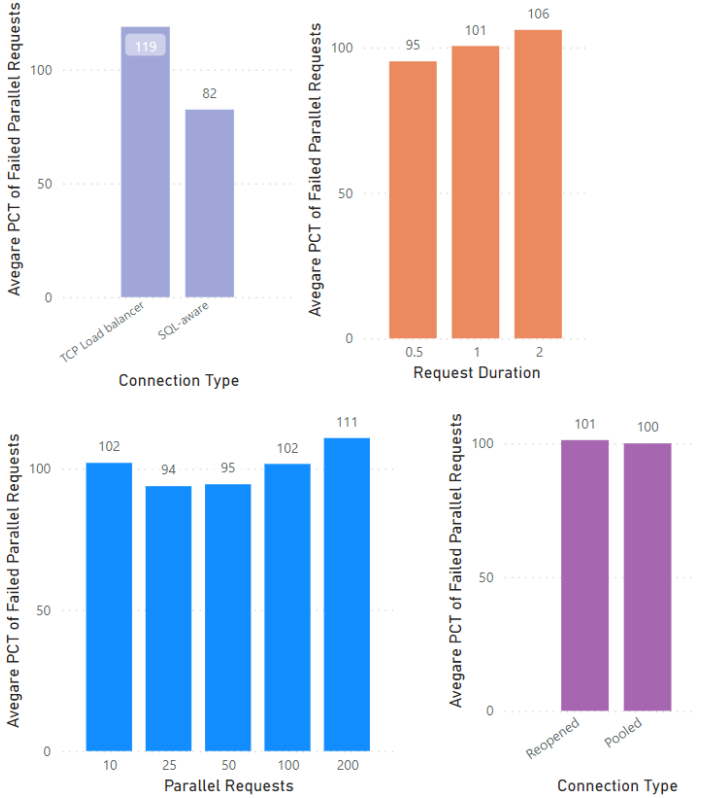


Fig. 2.10. Average failure rates for each experiment dimension

It is essential to note that the combination of factors makes the most significant impact on availability. For example, the difference between the average failure rates of reopened and pooled connections is a mere 1%. However, when coupled with other factors, say the type of Load Balancer, the failure rate varies by 49%. This finding highlights the importance of considering more than one factor.

2.5. Conclusions of the Second Chapter

The Second Chapter of the dissertation evaluates the factors impacting stateful microservice availability during failover. The following conclusions have been drawn:

1. Availability is a crucial aspect of any modern system. Stateful microservices deployed in orchestrated container systems benefit from a high level of maintainability. However, ensuring high availability of stateful microservices is a challenge.
2. The experiment conducted revealed that even clustered databases designed for reliability and availability experience a certain degree of availability loss during maintenance operations. The failure rate varies from 32% to 182%, depending on the setup and database load.
3. The type of connection, load balancer, and, to some extent, the database load affect the availability rate. Considering the average failure rate, the most significant impact comes from the load balancer type, with averages ranging from 82% to 119%. The number of parallel requests, connection type, and request duration have a less significant impact, up to 17%. Yet it is important to note that the availability rate is highly dependent on a combination of factors.
4. The type of load balancer affects not only performance and throughput but also availability. Having a multi-Primary database cluster behind a TCP load balancer may ensure higher performance for certain workloads. However, it is the SQL-aware load balancer that helps achieve a higher degree of availability during maintenance operations.

3

Loosely Coupled Failover for Low Latency Stateful Microservices

This chapter addresses the critical challenge of ensuring reliability and high availability in stateful microservices during failover operations. The method presented in this chapter aims to achieve loosely coupled and graceful failover in stateful microservices, a failover that is transparent to clients connected to the stateful microservice. By leveraging container orchestration systems like Kubernetes and utilizing advanced load balancers such as ProxySQL and HAProxy, the research explores a method to minimize disruptions and enhance availability.

The methodology involved a series of experiments on a three-node Kubernetes cluster with MySQL Galera Cluster, employing SQL-aware and TCP load balancers to observe their impact on failure rates and performance. Key findings reveal that the SQL-aware load balancer, ProxySQL, significantly enhances availability during maintenance operations, allowing for archiving a near-zero loss of availability during failover. By observing database connection activity and forcefully terminating idle client connections, the method allows for redirecting database requests from one node to another with negligible impact on the client.

The chapter highlights the importance of selecting appropriate load balancers and configurations to optimize the availability and performance of stateful microservices. These insights are crucial for developing resilient and efficient microservice architectures in modern cloud environments.

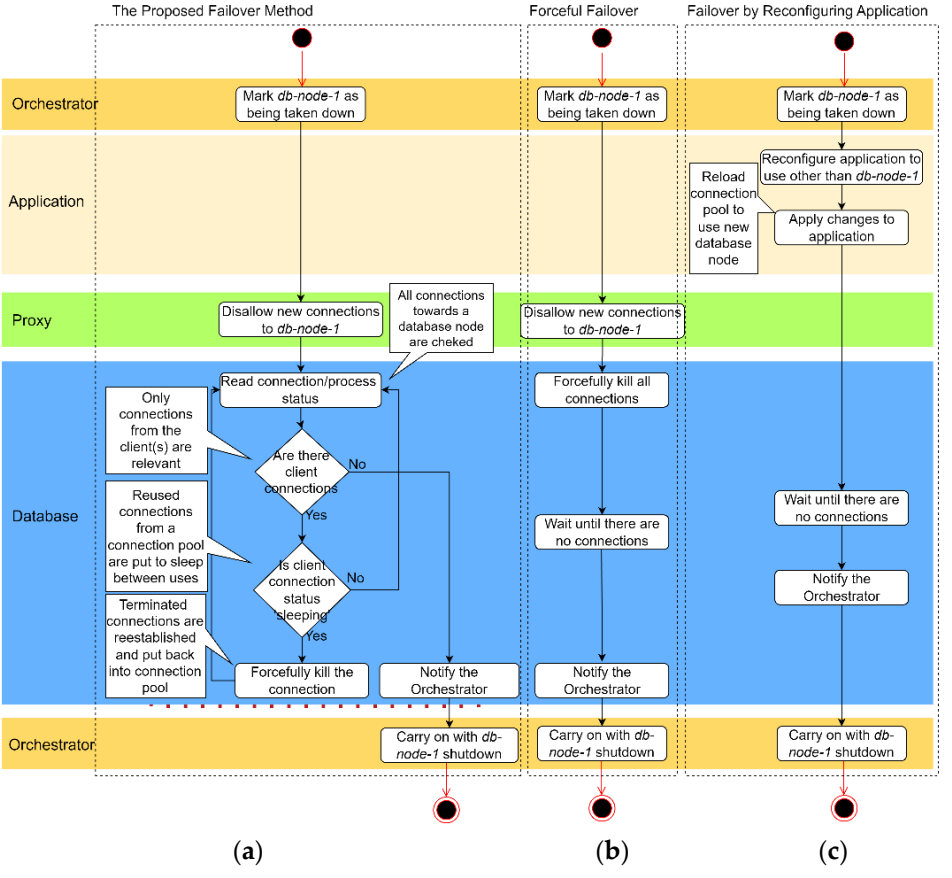
The proposed approach and results presented in this chapter were published in the author's two publications (Pakrijauskas & Mažeika, 2022a, 2024).

3.1. Method of Loosely Coupled Failover in Loosely Coupled Microservices

This section introduces the proposed method and explains its role in enhancing the availability of stateful microservices during failover in low-latency applications using pooled connections. When a node in a database cluster needs to be shut down, and a connection pool is in use, the orchestrator – whether a person or an automated script – must drain the connections either from the application side or on the database side. In the first scenario, the orchestrator requires access to the application. In the latter scenario, there is a risk of terminating actively used connections. The essence of the proposed method is to forcefully close only idle (sleeping) client connections to a database node.

The flow diagram (Fig. 3.1.) compares the proposed graceful failover method (Fig. 3.1.a) with the forceful failover method (Fig. 3.1.b) and failover by application reconfiguration (Fig. 3.1.c). The first step in the proposed method (Fig. 3.1.a) is performed by the orchestrator, which marks the node designated for shutdown as being taken down. Next, the database proxy (load balancer) removes the node from the database cluster, disallowing new connections to it. However, existing connections to the node are kept alive, and all incoming connections are rerouted to other nodes in the cluster.

Since the database proxy may not be aware of connection content, the forceful closure of connections is managed by the database node. The status of all client connections is read by the database node, which iterates through the statuses of the established client connections and terminates any idle ones until all are closed. Since terminated connections were not actively in use by a client application – they were in a free connection pool – their termination does not impact the client application. Only an attempt to reconnect is made.



(**a**) (**b**) (**c**)

Fig. 3.1. Flow diagrams of the proposed graceful failover method (**a**), forceful failover (**b**), and failover requiring changes on the application side (**c**)

Once all connections are drained from the database node, the orchestrator proceeds with shutting it down. A pooled connection is reestablished after being closed by the database. The application can continue querying the database, as new connections are routed by the database proxy to other nodes in the cluster, excluding the one being shut down.

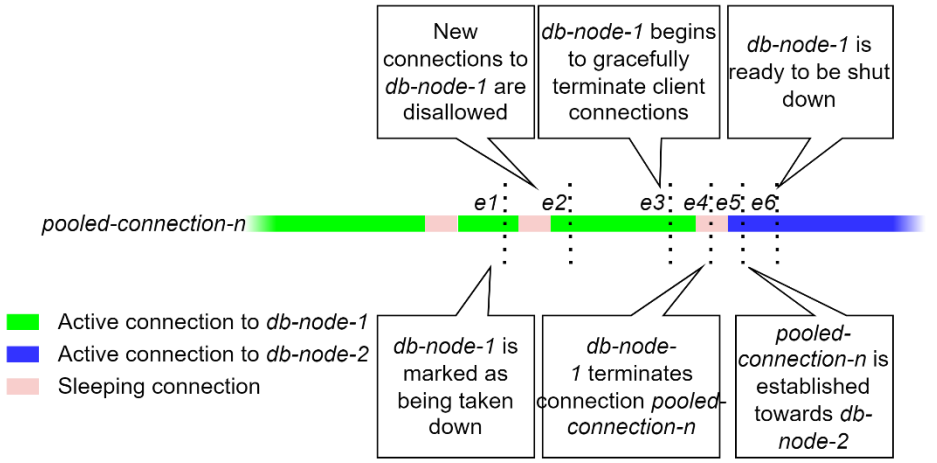


Fig. 3.2. Client connection transition

Figure 3.2. illustrates the transition of application client connections from one database node to another. In this example, a pooled connection transitions from *db-node-1* to *db-node-2*. Transition events are labeled as *e1* through *e6* in Figure 3.2. At the first event (*e1*), *db-node-1* is marked as being shut down, but no further action is taken against it. At *e2*, the database proxy isolates *db-node-1*, preventing new connections to it. At *e3*, *db-node-1* begins terminating incoming client connections. The pooled-connection-n enters a "*sleeping"* state, which is detected by *db-node-1* and terminated at *e4*. Once *pooled-connection-1* is reestablished, the database proxy redirects it to *db-node-2* at *e5*. After all connections have transitioned to *db-node-2*, *db-node-1* is marked ready for shutdown at *e6*.

3.2. Implementation of the Method

The proposed method enables the seamless transition of low-latency database connections from one node to another. However, for this method to function effectively, the distributed database cluster and its clients must be configured in a specific manner. As illustrated in Figure 3.3, the setup includes the following components:

- A database cluster configured for multi-Primary replication.
- A proxy layer capable of directing client requests to a specific node.
- A mechanism for terminating client connections.
- A connection pool to facilitate low-latency database connections.

- A retry mechanism on the client side to manage terminated connections.
- An orchestrator to manage and oversee the failover process.

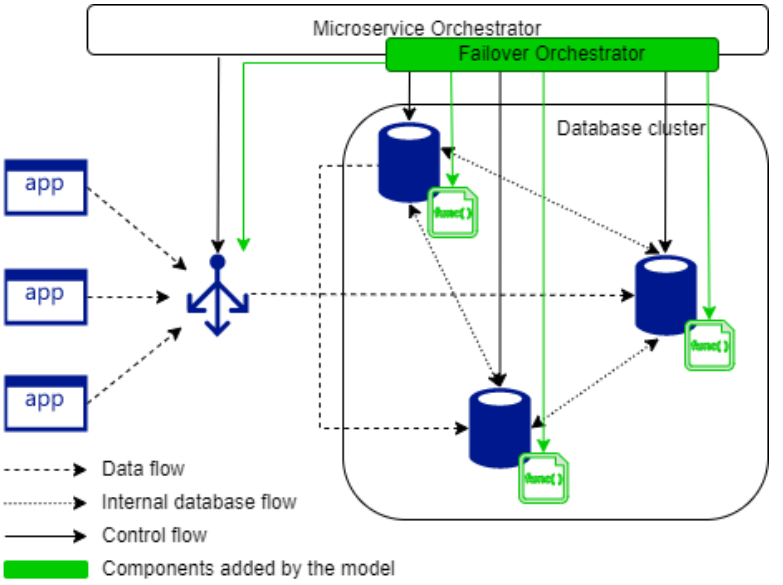


Fig. 3.3. Components of the proposed method

Multi-Primary replication enables the distributed database cluster to keep serving client requests during a graceful failover. A proxy layer, whether in the form of an appliance or service, balances requests between nodes during the switchover process (Algorithm 3.1). Once the switchover begins, new and reestablished client connections are routed to another node in the cluster, excluding the one being decommissioned.

Algorithm 3.1. Direct Database Connections

Input: *dbNode*, *connectionAction*
Output: *returnMsg*

```
1: procedure directDBConnections
2:   /* higher weight increases the node priority */
3:   defaultNodePriority ← 10
4:   if connectionAction = 'disallow' then
5:       // weight valued 0 disallows new connections toward the node
6:       nodePriority ← 0
7:       setNodePriority(dbNode, nodePriority)
8:       returnMsg ← 'connectionsDisallowed'
```

```
9:   end if
10:  if connectionAction = 'allow' then
11:      setNodePriority (dbNode, defaultNodePriority)
12:      returnMsg ← 'connectionsAllowed'
13:  end if
14:  return returnMsg
```

The client connection termination mechanism must read the content of a database process to decide on its termination. The algorithm is quite straightforward (Algorithm 3.2):

1. Collect a list of active client connections.
2. If there are client connections, iterate through the list and check their status:
   - Terminate the connection if the status is 'sleeping'.
   - Skip to the next connection if the status is other than 'sleeping'.
3. Refresh the list of client connections and check the status again.

Algorithm 3.2. Pseudocode of the connection termination algorithm.

```
     Input: clientConnectionIdentifier
     Output: returnMsg
1: procedure terminateClientConnections
2:   /* create list of client connections */
3:   clientConnectionList ← getClientConnections(clientConnectionIdentifier)
4:   while LEN(clientConnectionList) > 0
5:       /* loop thru client connections */
6:       for each: clientConnection ∈ clientConnectionList
7:           clientConnectionState ← getClientConnectionState(clientConnection)
8:           if clientConnectionState = 'sleeping'
9:               terminateConnection(clientConnection)
10:          else
11:              skipToNext()
12:          end if
13:      end for
14:      // refresh list of client connections
15:      clientConnectionList ← getClientConnections(clientConnectionIdentifier)
16:  end while
17.  returnMsg ← 'connectionsDrained'
18.  return returnMsg
```

The orchestrator does not affect connections that manage internal cluster traffic, such as replication or heartbeat, as interfering with these types of connections may lead to unwanted results. The cluster itself manages the internal cluster connections. Client connections can be identified by various parameters such as hostname, target database, or username.

The client is configured to retry failed requests in the event of a lost connection to the database. As the name suggests, the failover orchestrator supervises the failover of client connections between database nodes. During the entire managed failover process (Algorithm 3.1), the orchestrator issues commands to database nodes and the database proxy. By being aware of the status of client connections and the cluster configuration, the orchestrator can issue appropriate and timely commands to minimize the impact of managed failover on the availability of a database cluster.

The failover orchestrator can be integrated as part of a microservice orchestrator that manages microservices. For instance, it can be implemented as part of the Kubernetes Operator.

Algorithm 3.3. Failover orchestrator.

```
     Input: dbNodeList, proxyNode
     Output: exitStatus
1: procedure failoverOrchestrator
2:   for each: dbNode ∈ dbNodeList
3:       clientConnectionState ← proxyNode.directDBConnections(node,
'disallow')
4:           if clientConnectionState = 'connectionsDissallowed'
5:               terminateClientConnections.state ← node.terminateCli-
entConnections()
6:           end if
7:           if terminateClientConnections = 'connectionsDrained'
8:               /* call a generic maintenance procedure */
9:               call performDBNodeMaintenance(node)
10:          end if
11:      clientConnectionState ← proxyNode.directDBConnections(node, 'al-
low')
12:          if clientConnectionState = 'connectionsAllowed'
13:              /* start maintenance of other database nodes */
14:              skipToNext()
15:          end if
16: end for
17: return SUCCESS
```

The proposed method offers several advantages compared to other techniques for error-free managed failover:

- Support for connection pooling: Connection pooling achieves higher throughput and lower latency compared to reopened connections (Hohenstein et al., 2009).
- Low risk of double-write or other unwanted results when retrying a request (Dai, 2012). Since the connection is terminated between request executions, there is no need to retry the request; reestablishing a connection is sufficient. Even though Dai's Retryer (Dai, 2012) is a sound approach, it is more complex because, unlike the proposed method, it requires complex client-side logic.

One traditional approach to managed failover involves accepting an impact on availability. The proposed method enables performing a managed failover with near-zero loss of availability.

Another approach to managed failover involves rerouting requests by modifying the connection string on the client side. The proposed method eliminates the need for client-side changes to ensure a seamless switchover without failing requests. This reduces coupling, especially if the application consists of many microservices.

3.3. Setting up the Experiment

An experiment was conducted to evaluate how the proposed method functions under various conditions and compare it with the forceful failover method. A prototype environment was set up encompassing all essential components: a database cluster behind a proxy, a failover orchestrator, and a client. The two mentioned failover methods were assessed under different database loads and retry delays.

The prototype environment was deployed on a three-node Kubernetes cluster in Google Cloud Platform (GCP). The three nodes were identical, featuring E2 cost-optimized instances with 6 GB of memory, 4 CPUs, and 50 GB standard persistent disks (PD). The MySQL Galera was picked as a cluster as this database management system supports multi-Primary replication. ProxySQL served as the database proxy. The investigated architecture is shown in Figure 3.4.

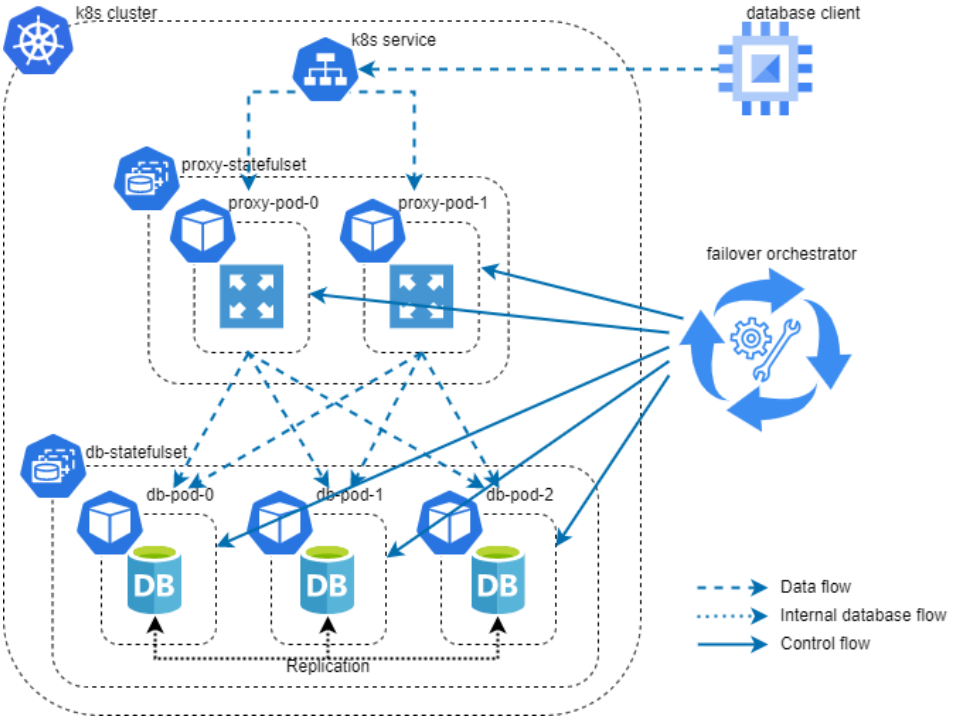


Fig. 3.4. Investigated architecture of the multi-Primary database cluster

The database client was configured on a VM in GCP. The VM was an E2 cost-optimized instance with 4 CPUs, 8 GB of RAM, and running Ubuntu 18.04. The failover orchestrator was deployed on an E2 cost-optimized instance with 1 CPU, 1 GB of RAM, running Ubuntu 18.04. The software versions and resource allocation for the MySQL Galera cluster and ProxySQL nodes are detailed in Table 3.1.

Table 3.1. Software Versions and Resource Allocation

Software and Version	Number of Pods	CPU Limit	Memory Limit
MySQL Galera Cluster 8.0.25	3	600m	1024MB
ProxySQL 2.0.18	2	300m	256MB

Various numbers of simultaneous requests were directed toward the database. Understanding how load (saturation) impacts the performance and effectiveness of the method is crucial.

```
CREATE TABLE IF NOT EXISTS logtable(
    id INT NOT NULL AUTO_INCREMENT,
    log_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    session_id VARCHAR(64),
    ordinal_number INT,
    PRIMARY KEY (id)
);

CREATE PROCEDURE probeRequest
    (sessionID VARCHAR(64), ordinalNumber INT)
BEGIN
    INSERT INTO logtable (session_id, ordinal_number)
    VALUES (sessionID, ordinalNumber);
END
```

Fig. 3.5. Database objects

The proxy layer can assist in the graceful failover of underlying database nodes. Therefore, this feature of the proxy layer was utilized during the experiment. Incoming requests were rerouted to the appropriate database nodes. The database node was set to the "*OFFLINE_SOFT"* state in ProxySQL to prevent connections to it (ProxySQL, 2021). This state prevents new connections to the database node while keeping existing connections intact.

The probe request executes the "*probeRequest*" stored procedure, which inserts data indicating a successful request. Once the failover is complete, the count of records per session is compared to the number of requests issued per session. The number of issued requests corresponds to the maximum *ordinal_number* of a session. The database objects are detailed in Figure 3.5. It was assumed that Select statements would remain unaffected by managed failover. Retrying read-only requests has a limited risk of encountering an exception compared to write requests.

Additionally, retrying a Select statement has an insignificant impact on the availability of read-only requests. Two types of retries were also evaluated: immediate retry and a backed-off retry after 1 second. Immediate retry is the simplest retry pattern. It was anticipated that the error-causing condition would permit an immediate retry of the request when using the proposed failover method. A backed-off retry is recommended if the error condition is expected to resolve after a certain delay (Microsoft Inc., 2022). It was assumed that 1 second is sufficient before retrying the request. The other two nodes in the cluster will be prepared to accept incoming requests within the 1-second delayed retry period.

The parameters of the experiment were as follows:

- Number of parallel sessions (saturation): 100, 200, 300, 400.

- Connection termination: graceful termination (the proposed method), forceful termination.
- Exception handling pattern: immediate retry, delayed retry (1 second).

All possible parameter sets were evaluated. Availability is impacted by maintenance time and failure rate (occurrence of failures) (O'Connor & Kleyner, 2012). The time required to transfer all connections from one node to another is considered the failover duration. As suggested by Hauer et al., the ratio of failed to successful requests was used to measure availability (Hauer et al., 2020). The failure ratio is the percentage of failed requests relative to the number of parallel requests. As the goal is to evaluate availability loss during a failover, it will be measured by how many requests fail relative to the number of parallel requests (saturation). Failover is the only factor impacting availability during the experiment. Given how connections are directed from one database node to another, it is assumed that the maximum number of failed requests per experiment execution is unlikely to exceed the saturation point.

$$\text{ratio} = \frac{\text{number of failed requests} * 100}{\text{number of parellel requests}} \tag{3.1.}$$

The experiment was run five times with each parameter set to compare the mean failover duration and failure ratio.

3.4. Experimental Results

As anticipated, the proposed method had a minimal impact on availability when compared to the forceful failover mechanism. During forceful failover, the mean failure rate across different parameter sets ranges from 53 to 94 (Fig. 3.5), increasing with the number of parallel requests. The proposed graceful failover method reduces the mean failure rate to nearly zero, with the highest mean failure rate being 3 and a deviation of up to 9. While the failure rate is still impacted by saturation, this effect is negligible compared to forceful failover.

Backed-off retries have a minimal impact on availability during graceful failover. A 1-second delay in retries positively impacts forceful failover at higher saturation levels. At 300 parallel requests, the mean failure rate drops to 93, and at 400, it decreases to 87.

Upon inspecting the failed requests, it was discovered that a deadlock was the cause of failure. High concurrency saturates the database node, causing queries to block each other, leading to deadlocks. Deadlocks were absent with graceful failover as clients transitioned gradually between nodes, preventing overload on the target database node. In forceful failover scenarios, database host overload

could be mitigated by gradual reconnection, reducing competition for database resources. However, this might necessitate additional modifications to client-side exception handling.

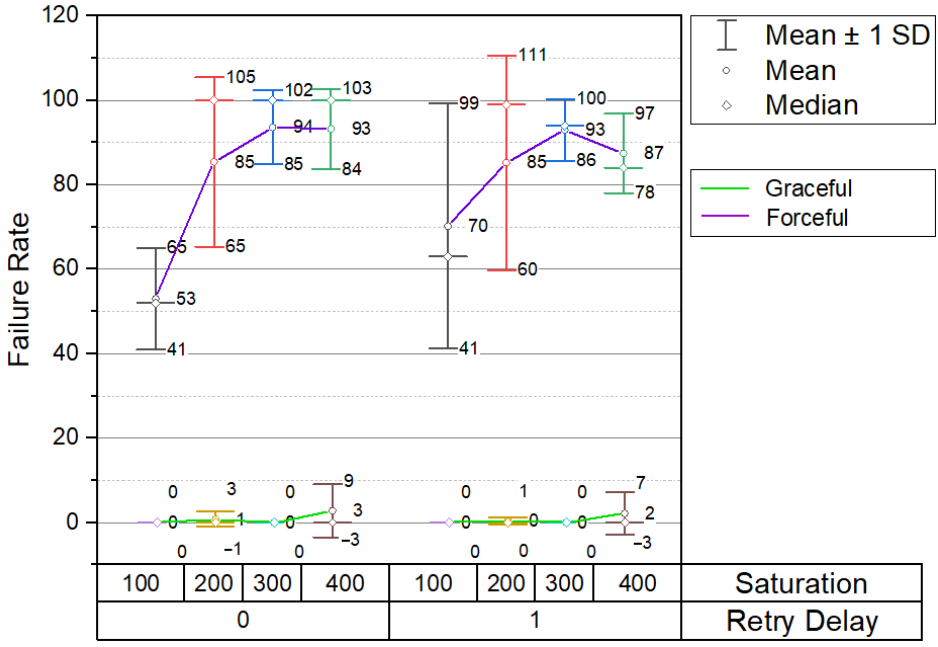


Fig. 3.6. Dependency of failure rate on failover method, saturation, and retry delay

Graceful failover between database nodes takes longer compared to other methods (Fig. 3.6). While the mean forceful failover time ranges from 54 to 63 seconds, graceful failover can take up to three times longer, with a mean duration ranging from 58 to 197 seconds. Failover duration increases with saturation, and the process is slowed by the need to inspect numerous connections. No action may be taken against a connection if it is actively in use by the client during the termination process. Thus, it undergoes multiple inspections during the failover preparation. Conducted in a round-robin manner, the inspection duration depends on the number of established connections, which is high in heavily saturated database clusters. Although graceful failover takes longer, it offers certain advantages. Client connections are gradually transferred to another node, preventing overload of the target node.

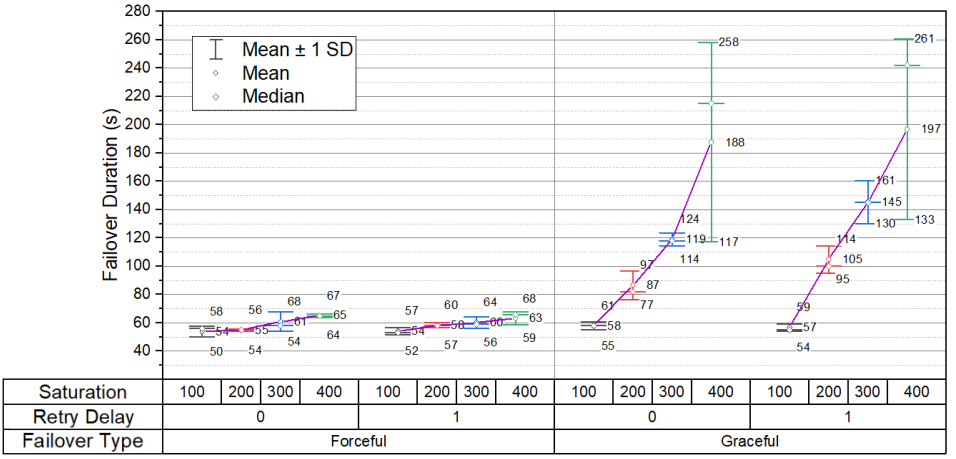


Fig. 3.7. Dependency of failover duration on failover method, saturation, and retry delay

While other methods and techniques for improving availability have their advantages, the proposed method has its unique benefits. Primarily, the method works with pooled connections, reducing database request latency and providing flexibility to failover between database cluster nodes with minimal impact on availability. Moreover, managed failover is transparent to applications. Not only does it require no client-side action, but importantly, it also raises no additional exceptions. This reduces coupling between the database cluster and its client in microservice applications. Since no client-side action is required, the database cluster operator can perform maintenance independently. Additionally, the method has been successfully implemented using open-source software like MySQL Galera Cluster and ProxySQL. However, the implementation options are not limited to these tools if the software supports the proposed method.

The proposed method has several drawbacks. One identified drawback is the requirement for a specific architecture for the method to function. A multi-Primary database setup is essential to attain high availability during managed failover. Compared to single-Primary replication setups, multi-Primary configurations are more complex. This complexity leads to slower performance due to communication latency and conflict resolution algorithms. However, if application requirements allow, the impact of this drawback can be mitigated by configuring the database cluster as pseudo-single-Primary. This setup would reduce the need for conflict resolution by restricting write workloads to a single database node. Additionally, failover time increases when employing the proposed method. Experiments have shown that in certain setups, shifting connections from one node to another may take three times longer. Although graceful failover takes longer than forceful failover, the extent of this disadvantage depends on the real-world

scenario. Since maintenance is typically planned, the additional minutes required for failover may be insignificant in the broader context. Another crucial aspect is that the proposed method has a limited impact on fault-tolerance. Despite achieving near-zero availability during managed failover, the method is not intended for disaster recovery.

3.5. Conclusions of the Third Chapter

Maintenance is an important aspect in the life cycle of a stateful microservice. The proposed method allows for limiting its impact on availability in a loosely coupled manner. The key findings discovered in this chapter are as follows:

1. The proposed method, which relies on observation and timely termination of connections, maintains a failure rate near zero even at high concurrency. Inactive pooled connections are terminated and gradually reestablished with another node, avoiding overload.
2. The results demonstrate that the proposed method significantly reduces the impact of managed failover on availability. While forceful failover causes at least half of the parallel requests to fail, the proposed graceful failover method achieves a near-zero failure rate during the failover of a low-latency stateful microservice, a clustered database with established pooled connections. Failure rates for forceful failover increase with saturation, whereas the failure rate for graceful failover also increases, but it is less significant compared to the other failover type. The achieved results indicate that retry delay impacts only forceful failover, leaving graceful failover unaffected. Even with a delayed retry, forcefully terminated connections overload the database node upon reconnection, resulting in failed requests. With the proposed method, the failure rate remains near zero despite a high number of simultaneous connections.
3. However, the time required to gracefully failover from one database node to another increases with the number of simultaneous client connections to a database node. Forceful failover time is insignificantly affected by the number of parallel requests. Implementing the method does not require substantial client-side modifications; it only needs to retry requests upon a transient connection fault. Additionally, the managed failover operation can be scripted or, in the case of Kubernetes, function as part of the Kubernetes Operator. A near-zero failure rate during managed failover operations enables database developers and administrators to perform various maintenance tasks at will, such as changing resource allocation or database options.

4. It was demonstrated that database maintenance operations can have a negligible impact on availability in systems requiring low latency while maintaining a low degree of coupling.

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

1. The experiment was performed using one RDMS. The results can be different from those of other RDBMSs on a NoSQL database management system.

4

Improved Burst Tolerance for Stateful Microservices

This chapter introduces a novel approach to improving the resilience of stateful microservices under sudden load spikes. Stateful microservices, which maintain data across interactions, face challenges in managing bursts of activity that can lead to performance degradation or failure.

The proposed rule-based method leverages predefined rules to dynamically adjust system behavior in response to increased load, thereby enhancing burst tolerance. This approach ensures that stateful microservices can handle traffic spikes efficiently, maintaining high availability and performance without compromising data integrity. It is demonstrated that the rule-based method effectively mitigates the impact of bursts, providing a robust solution for maintaining service availability in dynamic environments. This research significantly contributes to the field by offering a practical strategy for enhancing the resilience of stateful microservices in real-world applications.

The proposed approach and results presented in this chapter were published in the author's publication (Pakrijauskas & Mažeika, 2025).

4.1. Method for Burst Tolerance of Stateful Microservices

This section introduces the proposed method to enhance the burst tolerance of stateful microservices. In orchestrated environments, containers are allocated limited computing resources, which makes them prone to exhaustion under high loads. This risk is heightened by burst-type workloads. Stateful microservices require memory to process incoming requests, used for purposes such as caching, connection buffering, and query execution. Consequently, there is a limit to the number of concurrent requests and clients a single node can manage. In clustered stateful systems like databases, multiple nodes are available to distribute the workload.

It is assumed that write scaling may increase the overall capacity for both read and write requests. Requests are processed on a single node, and only the updated state of data is transferred to the secondary nodes. Memory utilization on secondary nodes is lower, providing capacity to process queries as the cache has space to store query plans and temporary data. However, there is a risk of reduced throughput and replication conflicts as database nodes are likely to operate at capacity limits. If a stateful microservice cannot handle a burst workload, a choice must be made regarding which Service Level Objectives (SLOs) to breach: accept reduced throughput while minimizing or avoiding loss of availability entirely, or solely accept loss of availability but on a larger scale.

The proposed method aims to distribute the burst workload while stateful microservice nodes undergo vertical scaling to meet the increased demand for resources (Fig. 4.1). The proposed method requires a stateful microservice cluster with at least three nodes to function. A cluster with three nodes is considered, denoted as db-nodes-*[0 − n]*, where $n \leq 2$. The flow of the proposed rule-based method is depicted in Figure 4.1a.

The orchestrator, such as the Kubernetes operator, monitors resource usage on the stateful microservice nodes. Once memory usage on the Pseudo-Primary node, *db-node-0*, exceeds the threshold, new connections are established with *db-node-m* (where $m \geq 2$), and *db-node-0* is blocked by the database proxy. The *db-node-m* (where $m \geq 2$) is prepared for increased demand, while additional load is directed to the remaining node, *db-node-1*. The orchestrator initiates vertical scaling of *db-node-m* (where $m \geq 2$). Then, load balancing begins between the remaining nodes, including the Pseudo-Primary *db-node-0* and *db-node-1*. The orchestrator monitors memory usage and instructs the database proxy to block new connections to the node with higher memory utilization, effectively balancing the load between two or more nodes.

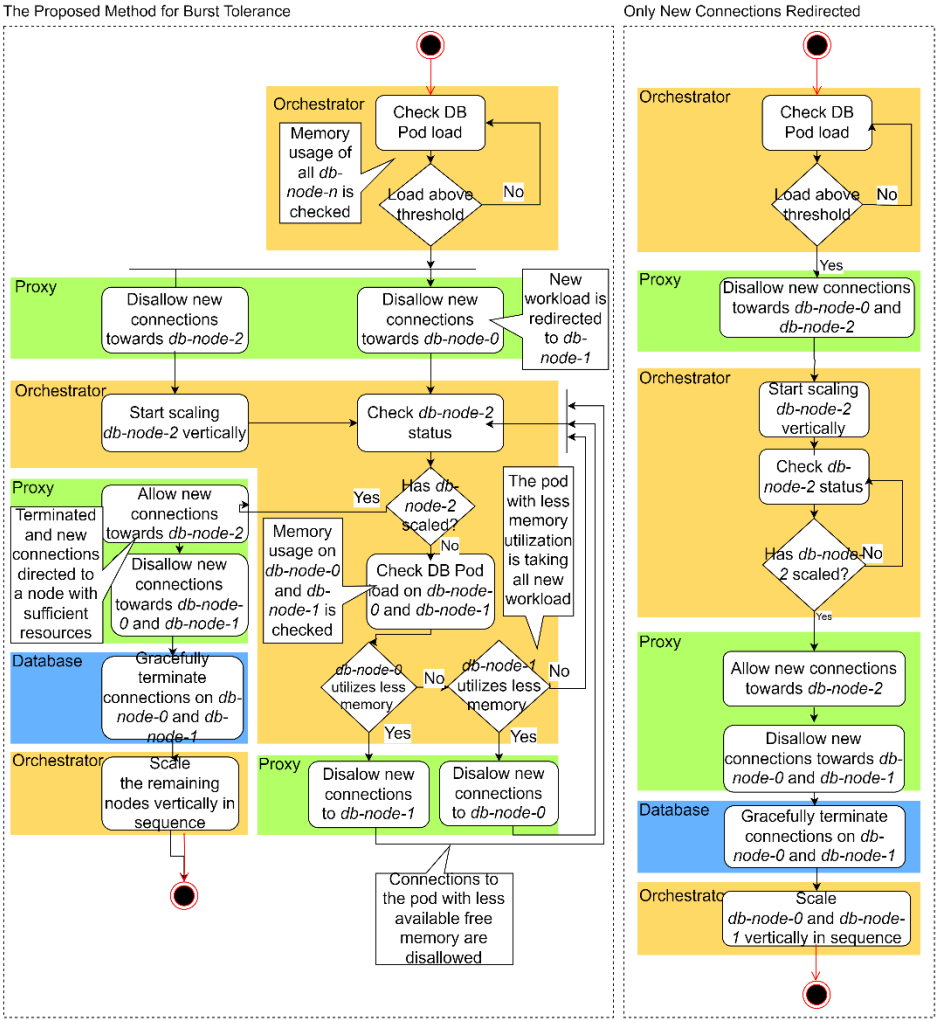


Fig. 4.1. Flow diagram of the proposed burst tolerance method (**a**) and an alternative approach (**b**)

New connections are established with nodes that have more available memory. As memory utilization changes on *db-node-0* and *db-node-1*, the node to which new connections are directed also changes. Existing connections remain intact. Once *db-node-m* (where $m \geq 2$) has been scaled, the orchestrator instructs the database proxy to permit new connections to that node. Existing connections are gracefully transferred from *db-node-0* and *db-node-1* by terminating client

connections through the database engine and reestablishing them toward db-node-m (where $m \geq 2$).

The graceful transfer of connections to the node that has just completed vertical scaling is performed with minimal impact on availability. The orchestrator then scales the remaining nodes vertically in sequence.

Threshold values and the new memory limit for the cluster can be determined in two ways. First, these values can be based on expert knowledge and may vary depending on the individual or team. For example, Microsoft recommends the threshold of 95% (Microsoft, 2025). However, the threshold could range from 85 to 95 percent, and the memory limit could be increased by an additional 10 to 25 percent. ML-based prediction values could further optimize resource usage by dynamically setting thresholds and new memory limits based on historical usage patterns.

Although there are additional metrics for measuring stateful microservice utilization, the decision to balance the load and initiate vertical scaling is based on memory usage. Other metrics, including CPU utilization percentage, the number of requests, response latency, etc., could also be considered. However, considering the impact of container memory exhaustion, the degradation of these metrics is less significant for the availability of SLOs.

Alternatively, only new connections can be directed to a secondary node. If only one node is designated as the Pseudo-Primary, the other nodes handle only the workload related to state transfer. Thus, their memory consumption is lower than that of a Pseudo-Primary node. There is no need to cache query execution plans, temporary data, etc. Therefore, a simpler approach would be to transfer only new connections to the secondary node (Fig. 4.1b). However, it is not expected that this method will be as effective as workload balancing. Redirecting only new connections leaves the remaining free memory on the original Pseudo-Primary node unused.

4.2. Implementation of the Method

The proposed method enabled a temporary distribution of the load across database cluster nodes during a workload burst. As demonstrated in the previous chapter, workloads can be transferred between database nodes with a minimal impact on availability. As illustrated in Figure 4.2, the essential components required for a stateful burst protector to function are as follows:

- A database cluster configured for multi-Primary replication.
- A proxy layer capable of directing client requests to a specific node;
- A connection pool to facilitate low-latency database connections.

The components essential for transparent failover of client sessions between database nodes are as follows:

- A mechanism for client connection termination (a stored procedure).
- A retry mechanism on the client side to manage termination.

The components essential for a stateful burst protector are as follows:

- A burst-protector mechanism to trigger failover and node scale-up during a burst.
- A modified failover orchestrator to manage the scale-up and workload transfer to an appropriately sized node with minimal impact on availability.

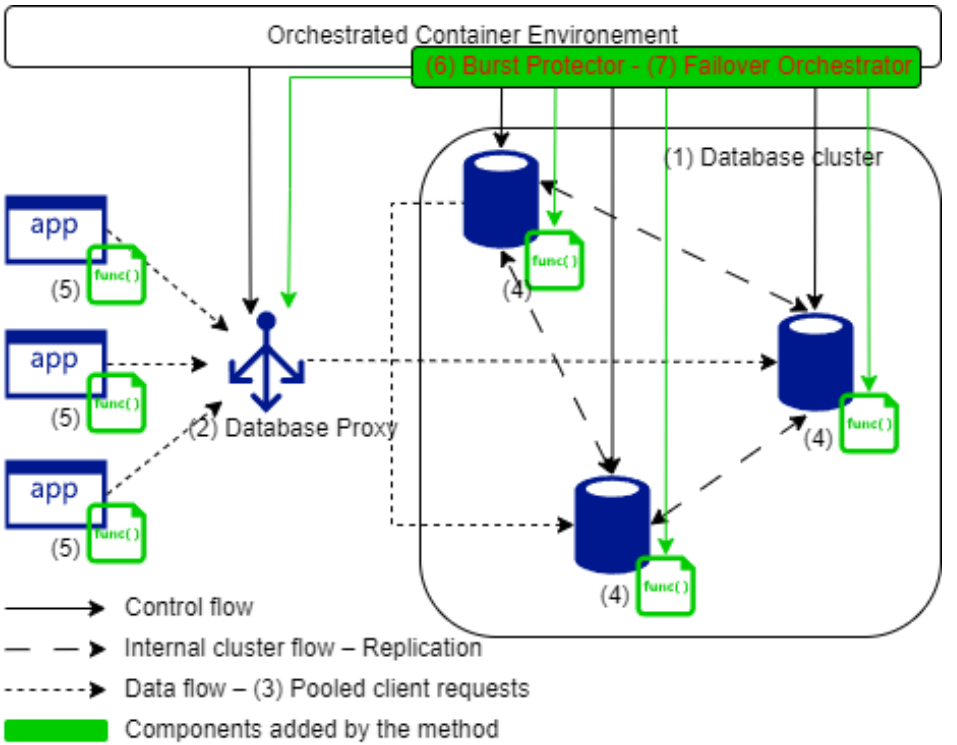


Fig. 4.2. Components required to implement the proposed method

Using multi-Primary replication, a distributed database cluster ensures uninterrupted service for clients during an unplanned failover of connections between database nodes. The proxy layer, capable of inspecting incoming queries, facili-

tates request balancing between nodes during burst workload handling. Additionally, it transfers requests to a scaled-up node with minimal loss of availability. This approach, derived from the previous chapter, is illustrated in Algorithm 4.1.

Algorithm 4.1. Direct Database Connections

```
    Input: dbNode, connectionAction
    Output: returnMsg
1: procedure directDBConnections
2:   /* higher weight increases the node priority */
3:   defaultNodePriority ← 10
4:   if connectionAction = 'disallow' then
5:       // weight valued 0 disallows new connections toward the node
6:       nodePriority ← 0
7:       setNodePriority(dbNode, nodePriority)
8:       returnMsg ← 'connectionsDisallowed'
9:   end if
10:  if connectionAction = 'allow' then
11:      setNodePriority (dbNode, defaultNodePriority)
12:      returnMsg ← 'connectionsAllowed'
13:  end if
14:  return returnMsg
```

The failover orchestrator, introduced in the previous chapter, must be modified to ensure nodes are scaled up in a specific order. It is updated to execute a failover and drain connections on a single node. Its algorithm is illustrated in Algorithm 4.2.

Algorithm 4.2. Prepare Node for Scaling

```
    Input: dbNode, proxyNode
    Output: exitStatus
1: procedure prepareNodeForScaling
2:   clientConnectionState ← proxyNode.directDBConnections(dbNode, 'disallow')
3:   if clientConnectionState = 'connectionsDisallowed' then
4:       terminateClientConnections.state ← dbNode.terminateClientConnections()
5:   end if
6:   if terminateClientConnections = 'connectionsDrained'
7:       return SUCCESS
8:   end if
```

Assuming there are at least three nodes in the cluster, one node is scaled up while the workload is distributed between the remaining two nodes. As depicted in Algorithm 4.3, the node to handle the burst workload is chosen based on the memory utilization of the nodes. The workload is directed to the node with lower memory usage. The workload is balanced between the two nodes while the third node is undergoing scaling. Once the third node is scaled and ready to accept connections, the balancing process returns a success message, and the burst protector procedure continues managing the database cluster and workload.

Algorithm 4.3. Balance Workload

```
    Input: dbNodeList, scaledDBNode, proxyNode
    Output: exitStatus
1: procedure workloadBalancer
2:   /* call a generic procedure to get initial scaled node status */
3:   scaledNodeStatus ← getNodeStatus(scaledDBNode)
4:   /* balance the connections while the node scaling is not complete */
5:   while scaledNodeStatus ≠ 'scalingComplete'
6:       for each: dbNode ∈ dbNodeList
7:           /* call a generic procedure to get memory usage and add it to a list
*/
8:           dbNodesMemUsage[] ← getNodeMemUsage(dbNode)
9:       end for
10:      /* find the node with the highest memory utilization by iterating the list
*/
11:      dbNodeMemHigh ← dbNodesMemUsage [0]
12:      for each: dbNodeMemUsage ∈ dbNodesMemUsage[]
13:          if dbNodeMemHigh.memUtilization < dbNodeMemUsage.memUti-
lization then
14:              dbNodeMemHigh ← dbNodesMemUsage
15:          end if
16:      end for
17:      nodeConnectivityState ← proxyNode.directDBConnec-
tions(dbNodeMemHigh,nodeName 'disallow')
18:      if nodeConnectivityState = 'connectionsDisallowed' then
19:          /* call a generic procedure to check scaled node status */
20:          scaledNodeStatus ← getNodeStatus(scaledDBNode)
21:      end if
22:  end while
23:  return SUCCESS
```

As depicted in Algorithm 4.4, the burst protection mechanism monitors memory usage and triggers both the balancing of client sessions and scale-up operation when a specific memory utilization threshold is reached on the Pseudo-Primary database node. Upon reaching the threshold, the burst protector initiates the vertical scaling operation of the Secondary node with the highest memory utilization and balances client connections between the Pseudo-Primary database node and the remaining node. The database cluster, consisting of the two balanced nodes, has additional memory to handle the burst when the Secondary node with the highest memory utilization is scaled first. Once the scaling up is complete, all client connections are transferred to the node with more computing resources from the remaining two nodes. These two nodes are then scaled up to meet the increased demand.

Algorithm 4.4. Burst Protection Mechanism

Input: *dbNodeList, pseudoPrimaryNode, memoryThreshold*
Output: *exitStatus*

```
1: procedure burstProtector
2:   /* call a generic procedure to get memory usage on pseudo-primary node */
3:   dbNodeMemUsage← getNodeMemUsage(dbNode)
4:   /* loop is exited once memory utilization on pseudo-primary node is above
the threshold */
5:   while dbNodeMemUsage ≤ memoryThreshold
6:       /* only check memory utilization on pseudo-primary node */
7:       /* while memory utilization is below the threshold */
8:       dbNodeMemUsage← getNodeMemUsage(pseudoPrimaryNode)
9:   end while
10:  /* collect memory usage on the nodes */
11:  for each: dbNode ∈ dbNodeList
12:      if dbNode.name ≠ pseudoPrimaryNode.name then
13:          /* call a generic procedure to get memory usage */
14:          dbNodesMemUsage[] ← getNodeMemUsage(dbNode)
15:      end if
16:  end for
17:  /* find the node with the highest memory utilization by iterating the list */
18:  dbNodeMemHigh ← dbNodesMemUsage[0]
19:  for each: dbNodeMemUsage ∈ dbNodesMemUsage[]
20:      if dbNodeMemHigh.memUtilization < dbNodeMemUsage.memUtiliza-
tion then
21:          dbNodeMemHigh ← dbNodesMemUsage
22:      end if
23:  end for
24:  /* call a generic procedure to scale the node up */
```

25: **call** *performDBNodeMaintenance*(*dbNodeMemHigh*)
26: /* start balancing sessions between remaining nodes */
27: *balancedNodesList* ← *dbNodeList.removeItem*(*dbNodeMemHigh*)
28: **call** *workloadBalancer*(*balancedNodesList*, *dbNodeMemHigh*,*proxyNode*, *memoryThreshold*)
29: **return** SUCCESS

The proposed method offers several advantages over other burst-handling techniques:

- No reliance on historical data. Undoubtedly, ML-based workload prediction is a powerful method for preparing for a burst. However, predictions may be imprecise if there is insufficient data to predict a burst. The proposed method provides an additional layer of protection against incidents related to stateful microservices that do not handle bursts properly. Expert knowledge or advice is sufficient for setting the memory usage threshold.
- The method is predictable. A rule-based approach is transparent, as the outcome is predefined by an understandable set of rules. Additionally, such an approach does not require the additional computing resources that ML-based approaches would need.
- Appropriate resource allocation. Stateful microservices have sufficient resources to manage baseline loads. Additional resources are allocated only when there is a need to handle increased loads.

If no historical data is available, the traditional approach is used to accept the loss of availability while a stateful microservice is scaled up. The proposed method minimizes loss of availability to some extent, allowing stateful microservices to operate under increased loads for longer durations, even if they run out of memory.

Another approach is to throttle clients (Wang et al., 2024). However, the proposed method does not require any action on the client side of a stateful microservice under burst conditions. This approach aligns with the microservice paradigm of reduced coupling.

Of course, if a burst workload cannot be balanced between the database nodes due to resource constraints, there are limited options other than accepting the loss of availability.

4.3. Setting up the Experiment

An experiment was conducted to assess the effectiveness of the proposed method under burst conditions and compare it with the out-of-the-box functionality of a

MySQL Galera cluster for handling burst workload balancing. A prototype environment was constructed, including the following essential components: a database proxy, a database cluster, a burst protection orchestrator, and a client. A synthetically generated dataset and synthetic workload were used to simulate burst scenarios and evaluate system behavior.

In the experiment, the focus was on the following aspects:

- Operational time under burst: This refers to the duration during which nodes, excluding the one being scaled up, can manage a burst workload. In other words, it is the time during which memory utilization rises from the set balancing threshold to its limit.
- The number of failed requests if scaling is successful. It is assumed that if one node fails to scale on time, the loss of availability becomes significant. Additionally, how requests are managed during a burst operation is crucial. As suggested by Hauer et al., the percentage ratio of failed to successful requests will serve as the measure of availability (Hauer et al., 2020).

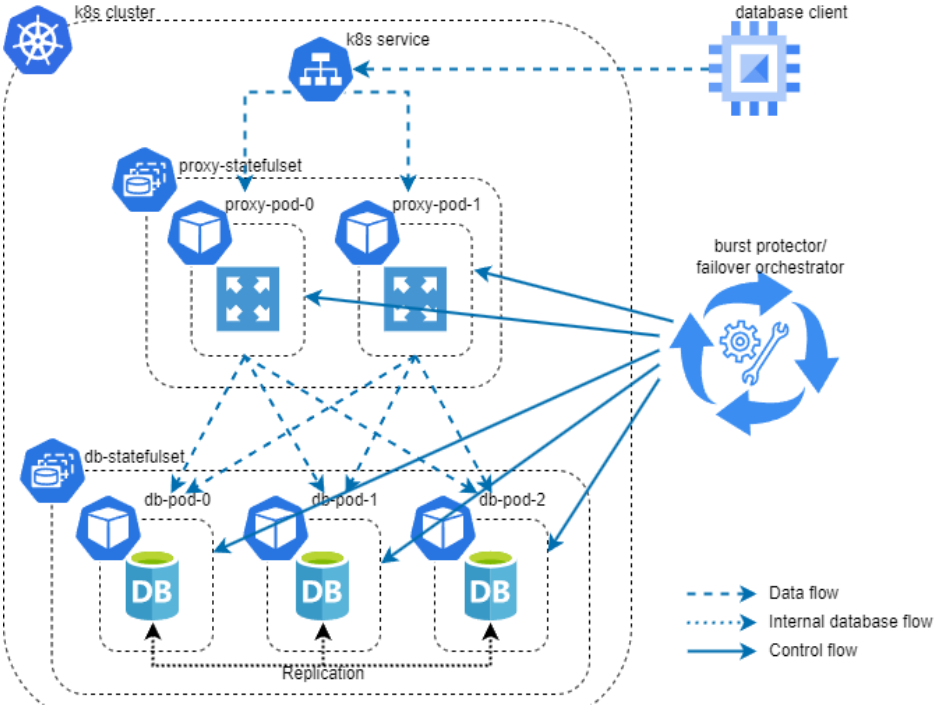


Fig. 4.3. Architecture of the multi-Primary database cluster investigated

The architecture of the cluster under investigation is depicted in Figure 4.3. The prototype environment, based on the investigated architecture, was established on a three-node Kubernetes cluster in the Google Cloud Platform (GCP). The three nodes were identical: E2 cost-optimized instances with 6 GB of memory, 4 CPUs, and 50 GB of standard persistent disk (PD). The MySQL Galera cluster was selected for its capability of multi-Primary replication. ProxySQL was utilized as the database proxy.

The database client was configured on a virtual machine (VM) in GCP. The VM was an E2 cost-optimized instance equipped with 4 CPUs and 8 GB of RAM, running Ubuntu 22.04. The failover orchestrator was configured on an E2 cost-optimized instance with 1 CPU and 1 GB of RAM, running Ubuntu 22.04. The software versions and resource allocation for the MySQL Galera cluster and ProxySQL nodes are detailed in Table 4.1.

Table 4.1. Software versions and resource allocation.

Software and Version	Number of Pods	CPU Limit	Memory Limit
MySQL Galera Cluster 8.0.36	3	500m	1024MB
ProxySQL 2.2.5	2	400m	256MB

The Yahoo! Cloud Serving Benchmark (YCSB) was utilized to generate both the synthetic data and the workload (Cooper et al., 2010). A table containing 100,000 records was loaded into the database cluster. To maintain consistency throughout the experiment, the same data set was used across all experiment runs. The generated workload was read-focused, comprising 83% selects, 15% updates, and 2% inserts. The request distribution was set to uniform. Although YCSB's functionality allows for generating requests in more realistic patterns and different workload ratios, the uniform request distribution and read-focused workload were sufficient for validating whether requests could be balanced across database nodes. In addition, using different types of workload distribution, such as Zipfian, would introduce more uncertainty to the experiment. The database driver on the client side was configured to make three reconnection attempts every 2 seconds.

Although data spikes are a characteristic of bursts in stateful microservices, they were not considered in the experiment. Given that the database cluster used in the experiment was relatively small and the data was fully replicated across nodes, the increase in requests for a subset of data had a negligible impact on the overall results.

The base workload generated by YCSB consisted of 10,000 operations and 160 threads, with a throughput target of five operations per second. The base workload included insert queries and was intended to run for a minimum of 120 seconds. The parameters for the base workload were chosen to achieve a memory

utilization of 600MB for the Pseudo-Primary database. However, in practice, due to limited control over MySQL's memory utilization, the base workload's memory usage may peak between 580MB and 620MB.

As previously mentioned, duration and intensity are the primary characteristics of a burst. Since the duration is expected to be mitigated by scaling and session balancing, the proposed method under varying intensities will be evaluated.

The burst workload commenced 10 seconds after the base workload. A new client with five threads was added every 0.25 or 0.5 seconds, targeting 15 operations per second and aiming to complete 2,500 operations. Each client was intended to run for approximately 33 seconds. A total of 30 clients with 150 burst threads were initiated. With a new client added every 0.25 seconds, the peak of 150 threads is reached, accounting for code execution overhead, in approximately 7.5 seconds. With a new client added every 0.5 seconds, the peak is reached at roughly 15 seconds. Regarding the total number of requests, 2,500 operations across over 30 client threads will amount to approximately 75,000 requests. The intensity of the burst is depicted in Figure 4.4. With the aforementioned parameters of operations per second, thread count, and target operation settings, it was sufficient to generate a burst that peaks memory utilization at around 800MB.

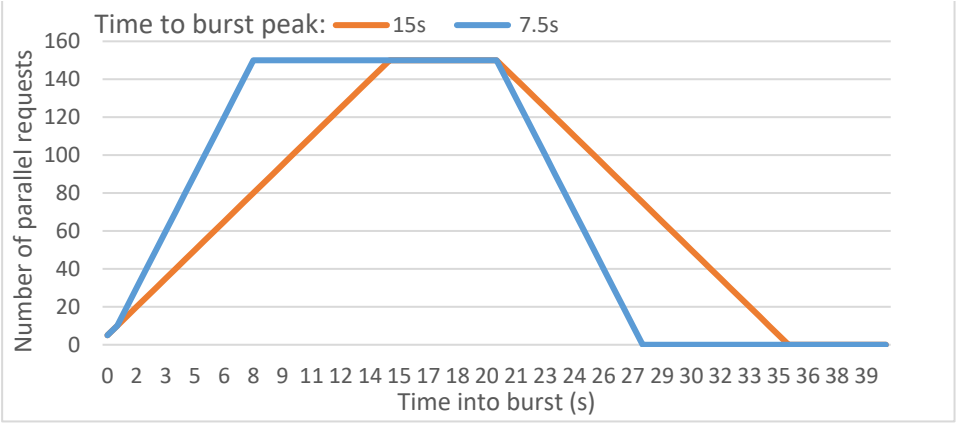


Fig. 4.4. Intensity of generated burst workloads

Once workloads commence, the orchestrator monitors the cluster as previously described, initiating the transfer of client connections between appropriate nodes at suitable memory utilization levels. Memory utilization on the pods is monitored. Vertical scaling of the third node will be simulated. As memory utilization reaches 750MB on either of the two operational nodes, it will be assumed

that the third node has scaled with sufficient computing resources. Then the existing connections to the two operational nodes are gracefully transferred to the third node.

Client sessions are managed in three distinct ways:

- No action taken due to the out-of-the-box functionality of the MySQL Galera Cluster and ProxySQL.
- Redirecting all new sessions to the second node.
- Balancing all sessions between two nodes.

In summary, the method will be assessed using the parameters detailed in Table 4.2.

Table 4.2. Parameters of the experiment

Workload Type Distribution, %			**Burst Handling Type**	**Time to Burst Peak (s)**
Reads	**Updates**	**Inserts**		
83	15	2	No action	7.5
				15
			Redirect	7.5
				15
			Balance	7.5
				15

The YCSB collects a comprehensive list of statistics for each experiment run. The necessary statistics to measure availability were selected and aggregated: the total number of operations and the number of failed operations. The total number of requests issued against the database follows a consistent pattern; for instance, the average for read requests is 85,976. However, there is a slight variation in each experiment; the total number of requests deviates from the average across all experiments by 2%. This deviation was not expected to impact the final results, as the failure rate percentage was measured.

$$\text{failure ratio} = \frac{\text{number of failed requests} * 100}{\text{number of parellel requests}} . \quad (4.1.)$$

The experiment was run five times with each parameter set to compare the error ratio and time under burst.

4.4. Verifying the Method

Upon analyzing the experiment results, the first observation was that the deviation is quite significant despite an overall pattern. Thus, median values are also important. The results show a notable deviation; however, on average, there is an improvement in handling burst conditions when the proposed method is applied. Despite the workload being synthetic and relatively simple, without complex calculations, the memory utilization pattern lacks consistency. It is not possible to control the inner workings of the XtraDB engine, a replacement for MySQL InnoDB. Consequently, in some experimental runs, the proposed method did not perform as expected. Conversely, in some experimental runs, the cluster crashed almost immediately if no action was taken.

As anticipated, the proposed write-scaling method for burst protection impacted availability concerning successful requests and time when compared to using a single node for write requests. Initially, the mean failure rate percentage ranged from 0.78 to 1.4 percent when no action was taken to balance the workload. If new sessions were redirected to another node, the range was from 0.35 to 1.09 percent, depending on burst intensity (Fig. 4.5). However, the proposed method allowed for certain burst intensities to reduce the failure rate percentage to zero, with the mean rate reaching only 0.04 percent.

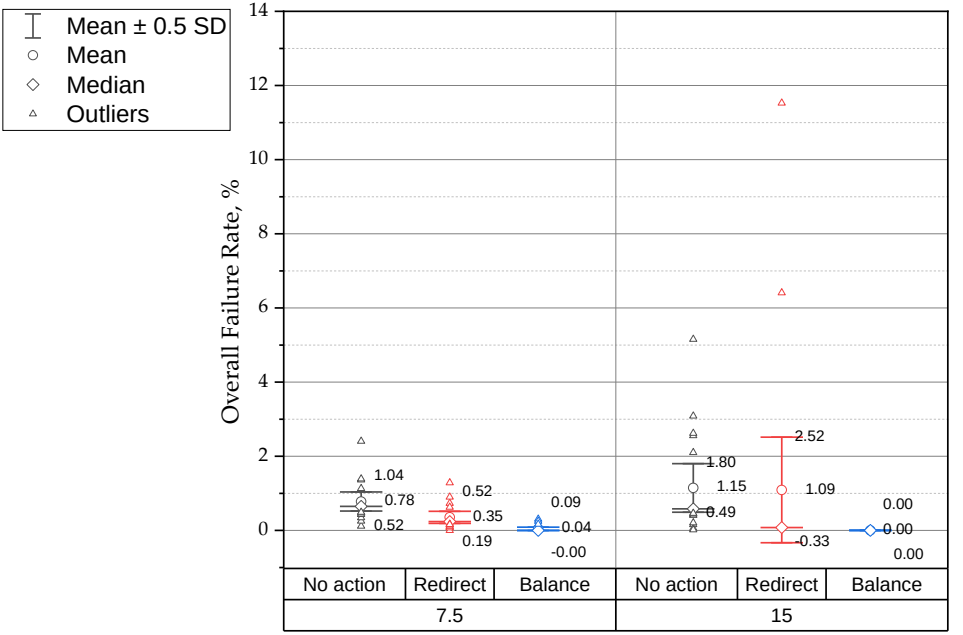


Fig. 4.5. Dependency of the overall failure rate on burst handling type and burst intensity

For read requests (Fig. 4.6), the proposed method significantly decreased the impact of bursts on the failure rate. Without any action or with only the burst workload redirected, the mean failure rate ranged from 0.33 to 0.61 percent. However, when sessions were balanced, the failure rate decreased to 0.04 percent.

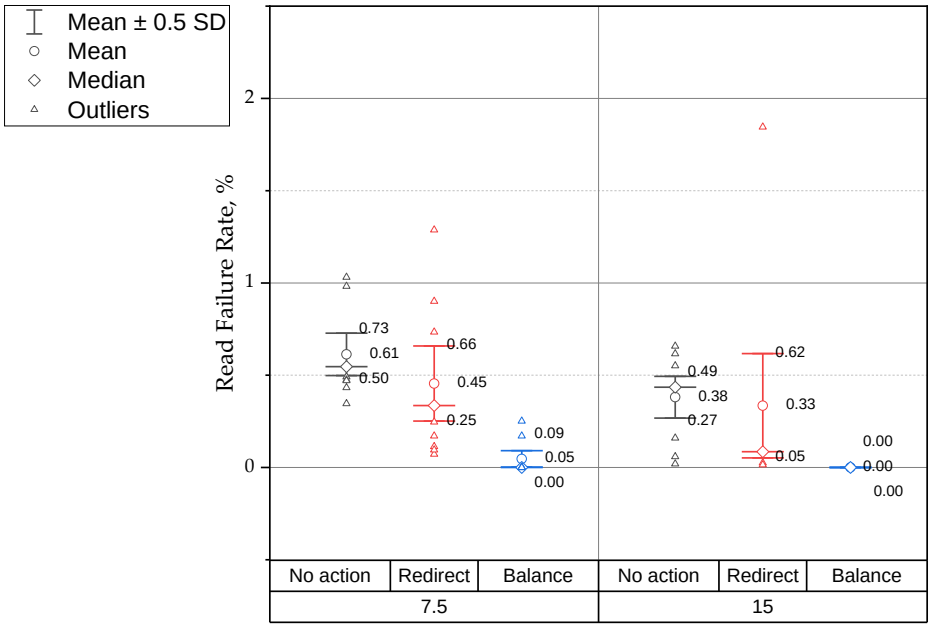


Fig. 4.6. Dependency of read failure rate on burst handling type and burst intensity

Similar to read requests, the proposed method also reduced the impact of bursts on the failure rate of updates (Fig. 4.7). Without intervention or with only the burst workload redirected, the mean failure rate ranged from 0.17 to 2.36 percent. The highest mean failure rate update occurred for redirected update requests under lower intensity. However, the standard deviation was quite significant due to a single outlier. If sessions were balanced, the failure rate would decrease to 0.01 percent.

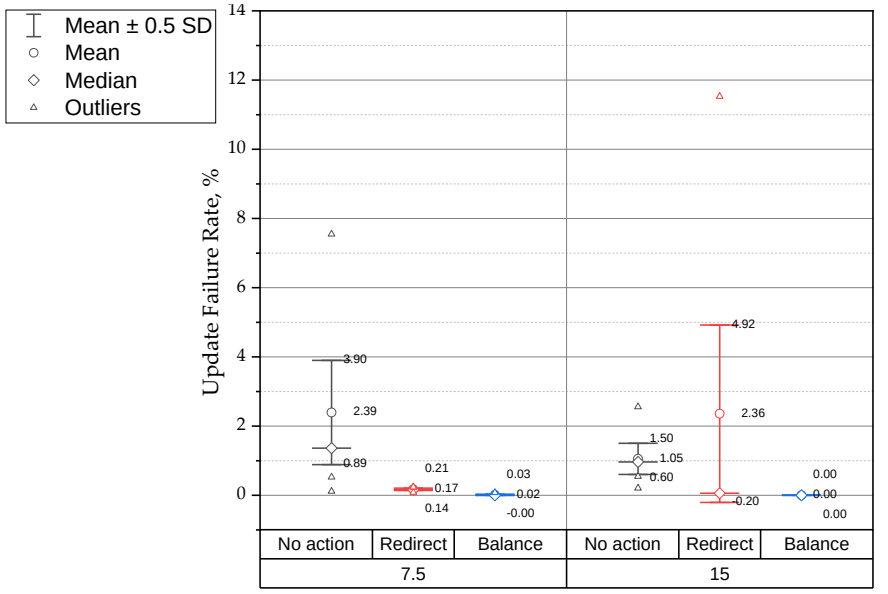


Fig. 4.7. Dependency of update failure rate on burst handling type and burst intensity

The proposed method reduced the failure rate of data insertion requests (Fig. 4.8). Compared to a failure range of 0.33 to 2.77 percent, balanced sessions reduced the failure rate to 0.61 percent. A few outliers skewed the results, but the overall trend remained unchanged.

In summary, session balancing significantly decreases the failure rate compared to taking no action or redirecting client sessions. The outliers, most notably manifesting when the workload is only redirected, are likely caused by the second node pushed to its limit, thus causing more requests to fail. If sessions are balanced, the workload is distributed more evenly, and thus fewer requests fail. As illustrated in Table 4.3, the failure rate decreased by at least 81.82%. Presumably, the reduced load on a database node minimized the number of failed requests during a graceful connection transfer to other nodes in the cluster. This finding supports the results of the previous chapter.

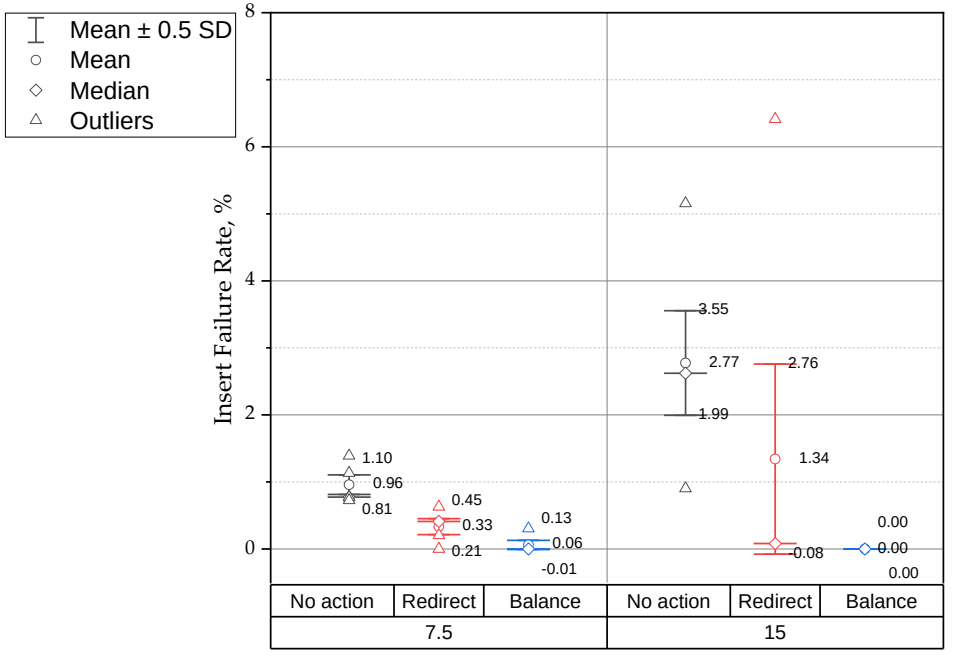


Fig. 4.8. Dependency of insert failure rate on burst handling type and burst intensity

Table 4.3. Summary of failure rate

Time to Burst Peak (s)	Workload	Average Failure Rate by Burst Handling Type			Comparative Decrease in Failure Rate Using Balancing (%)	
		No Action	Redirect	Balance	No Action	Redirect
7.5	Reads	0.61	0.45	0.05	91.80	88.89
	Updates	2.39	0.17	0.02	99.16	88.24
	Inserts	0.96	0.33	0.06	93.75	81.82
	All	0.78	0.35	0.04	94.87	88.57
15	Reads	0.38	0.33	0	100	100
	Updates	1.05	2.36	0	100	100
	Inserts	2.77	1.34	0	100	100
	All	1.15	1.09	0	100	100

The time required for any node to reach 750MB in memory utilization improved alongside the proposed method (Fig. 4.9). Under burst conditions, the time under burst increased to 42.4 and 40.8 seconds compared to 23 and 25 seconds when no action was taken. Redirecting burst sessions also led to an increased time under burst compared to taking no action. However, the increase in time under burst at higher load intensities when using balancing compared to redirecting was insignificant. While redirected sessions allowed the cluster to operate under burst for an average of 42.2 seconds, balanced sessions resulted in an average time under burst of 40.8 seconds. Based on the outliers, the time under burst may vary for each approach.

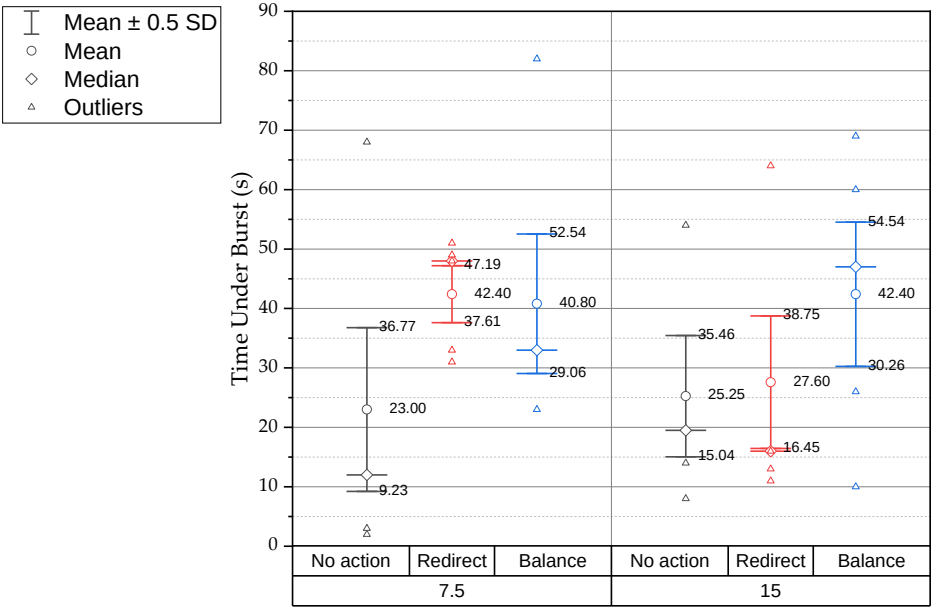


Fig. 4.9. Dependency of time under burst-on-burst handling types and burst intensity

Despite the outliers, there is still a positive trend in time under burst changes when balancing is used (Table 4.4). The combination of lower failure rate percentages and increased or slightly reduced times under burst, depending on the methods compared, indicates that session balancing is a viable approach to enhance stateful service availability under burst conditions.

One commonality among all types of workloads is the presence of outliers. In some instances, these outliers exhibit significant deviations from mean values and the overall trend, which remains consistent across experiments despite these outliers. Since MySQL memory usage is influenced by numerous factors, it can

occasionally deviate from the baseline. For instance, this can occur when new data or query plans are added to the cache.

Table 4.4. Summary of time under burst

Time to Burst Peak (s)	Average Time Under Burst			Comparative Increase in Time Under Burst when Using Balancing (%)	
	No Action	Redirect	Balance	No Action	Redirect
7.5	23	42.2	40.8	177.39	96.68
15	25.25	27.6	42.4	167.92	153.62

The drawbacks of the proposed method arise from the use of a multi-Primary relational database management system. This system exhibits reduced performance owing to added complexity and increased communication latency compared to single-Primary replication setups. Unlike single-Primary replication setups, burst workloads further increase latency. Conflict resolution introduces an additional level of latency in a multi-Primary replication setup during peak loads. This issue is partially alleviated by designating one node as pseudo-Primary. This approach reduces the need for conflict resolution, as only one database node would handle write operations.

As illustrated in Figure 4.10, average latency increases when workloads are balanced between two nodes. The additional latency stems from the increased volume of bidirectional replication as multiple nodes process writes, and when client connections are transferred to the third node. Although latency is higher compared to when no action is taken, it only increases when client sessions are balanced. Consequently, the SLOs related to stateful microservice latency are degraded for a limited duration.

However, in terms of burst handling, the proposed method utilizing a multi-Primary database offers multiple benefits. Firstly, a stateful microservice operator can manage bursts independently without requiring any client-side modifications. It employs a predictable and transparent rule-based approach, with clear and understandable reasoning behind the actions taken. However, to remain unbiased, there remains uncertainty regarding how MySQL or any other RDBMS would operate and utilize memory under peak workload conditions. Lastly, the proposed method has been proven effective using open-source components. Other software combinations are also likely to function similarly or even more effectively as implementations of this method.

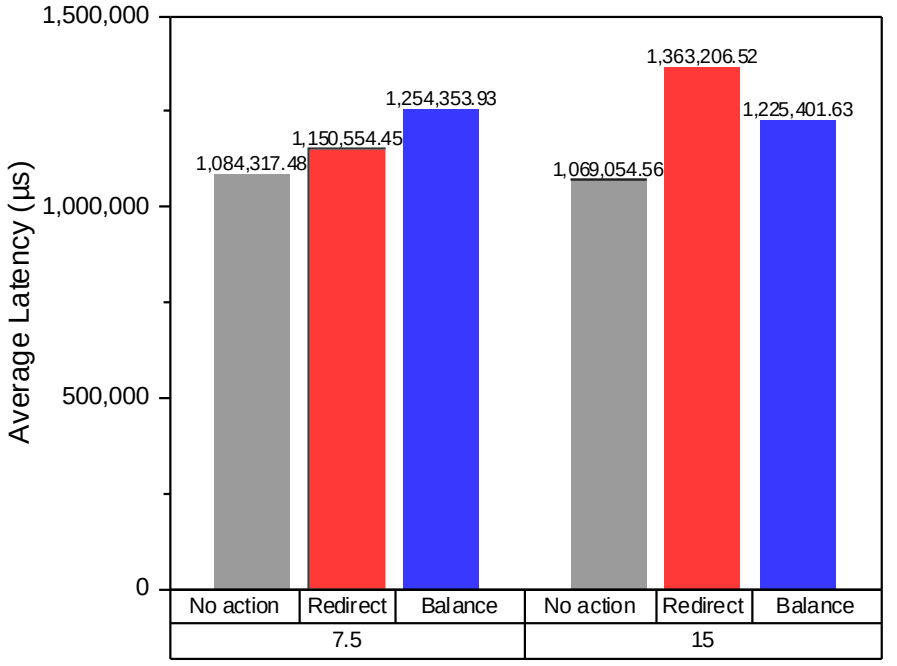


Fig. 4.10. Dependency of the average latency (μs) on burst handling type and burst intensity

There are several avenues for further research. Firstly, numerous other database management systems support multi-Primary replication, such as PostgreSQL with Bi-Directional Replication extension (EDB, 2025), CockroachDB (Cockroach Labs, 2026), CouchDB (CouchDB, 2026), and Cassandra (Apache Cassandra, 2025). PostgreSQL, being one of the classic RDBMSs, is likely to achieve results like the MySQL Galera Cluster with the proposed method. CockroachDB, a more modern RDBMS, is designed as a distributed database featuring a modern consensus protocol and automatic sharding. Despite being a more modern tool, CockroachDB still depends on third-party proxy solutions for workload balancing, enabling the proposed method to handle bursts. Cassandra and CouchDB, two NoSQL alternatives supporting multi-Primary replications, are designed with data distribution in mind. However, promoting a Cassandra node to a writer necessitates additional reconfiguration, and a CouchDB node must be pre-configured to accept writes. This requirement could pose challenges for Cassandra and CouchDB if the proposed method is used for burst handling. Investigating the efficiency of the proposed method using a different database management system other than MySQL Galera Cluster could be worthwhile.

Additionally, integrating the proposed rule-based method with ML-based techniques can improve burst workload management while optimizing resource usage, for example, scaling a microservice to meet the coming demand. This integration could serve as a failsafe measure, considering that workload predictions may be imprecise and a stateful microservice may not be properly adapted to actual demand. First, the proposed rule-based method can extend the operational time of a stateful microservice if vertical scaling is not initiated promptly. Hopefully, the additional time under burst would be sufficient for the stateful microservice to scale. Second, workload balancing may be initiated while the microservice undergoes vertical scaling.

4.5. Conclusions of the Fourth Chapter

The proposed method, involving write-scaling and load balancing to safeguard a stateful microservice during an overwhelming burst workload, enhanced its availability. The additional time gained under burst conditions, though modest, could be enough to complete vertical scaling in certain cases, thereby enabling the stateful microservice to manage the increased workload. The following conclusions can be drawn:

1. The experiments demonstrated that stateful microservices can function under burst conditions for extended periods. Burst intensity affects availability. When burst intensity is sufficiently low, load balancing and scaling writes can ensure that the stateful microservice scales promptly to meet heightened demand. Conversely, if the burst intensity is too high, a stateful microservice may crash irrespective of the proposed method's application. However, the failure rate percentage remains lower, and service unavailability periods are shorter.
2. The proposed method enabled a stateful microservice to operate under burst conditions nearly twice as long as with standard functionality. Although the proposed method causes an increase in response latency, its impact on the stateful microservice is limited to a brief period.
3. In resource-constrained environments, a stateful microservice can be configured to handle the expected standard workload and scale up to accommodate increased demand. Additionally, the proposed method can serve as a failsafe when workload prediction algorithms deliver imprecise forecasts.

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

1. The experiment was performed using one RDBMS. The results can be different when using other RDBMSs on a NoSQL database management system.

General Conclusions

The present research addresses the critical challenge of ensuring reliability and high availability in stateful microservices during failover operations and maintenance activities. The research aims to enhance the availability of stateful microservices in orchestrated container environments by identifying key factors impacting availability, proposing a method for low-latency stateful microservice availability during maintenance, and introducing a rule-based method to improve resilience to sudden workload increases. The performed research can be concluded as follows:

1. The literature review highlights the critical importance and inherent complexity of managing stateful microservices in orchestrated container systems. The challenges involved in state management include ensuring data consistency, maintaining backup availability, and addressing performance overhead. SLIs and SLOs, namely the CPU and memory utilization, are the primary factors for decision-making processes to ensure the reliability and availability. Data-driven methods are used for fault prediction, effective container scheduling, and workload prediction when improving the availability and reliability of stateful microservices. Rule-based methods, such as custom schedulers and state controllers, are used as techniques for improving reliability in orchestrated container systems.

2. Various factors affecting service availability, including load intensity, load balancer type, and connection type, were evaluated. It was demonstrated that SQL-aware load balancers, such as ProxySQL, provide higher availability compared to TCP load balancers like HAProxy, with averages ranging from 82% to 119%. This is attributed to the ability of SQL-aware load balancers to recognize failover events and transfer requests to another database node with a lower failure rate. In addition, the type of connection (pooled vs. reopened) also plays a significant role, with reopened connections showing a lower failure rate in the context of a SQL-aware load balancer. It is important to note that the availability rate is highly dependent on a combination of factors.

3. A method was proposed to address the challenge of ensuring high availability in stateful microservices during managed failover operations. The method aims to achieve transparent and graceful failover in stateful microservices by leveraging container orchestration systems like Kubernetes and utilizing load balancers to transfer low-latency client connections between database nodes with minimal impact on the client applications. Key findings reveal that the SQL-aware load balancer, ProxySQL, significantly enhances availability during maintenance operations, allowing for near-zero loss of availability during failover. While standard functionality allows for achieving a failure rate ranging from 53 to 94, the proposed method reduced it to negligible values, reaching 3. Although failover duration increases, the reduction of failure rate, maintenance operations may have an insignificant impact on stateful microservice while maintaining low coupling with client microservices.

4. A novel rule-based method to improve the tolerance of stateful microservices under sudden load spikes was introduced. The proposed method leverages predefined rules for write-scaling to dynamically adjust system behavior in response to increased load, thereby enhancing burst tolerance. Burst intensity impacts availability; with lower intensity, the load can be balanced and writes scaled to meet increased demand. However, with sufficiently high burst intensity, a stateful microservice may crash regardless of the method used. Nonetheless, the proposed method reduces the failure rate and allows the service to operate under burst conditions for nearly twice as long as standard functionality. The time under burst increases from 23 to 41 seconds. Furthermore, the proposed method can serve as a safeguard when workload prediction algorithms fail to accurately forecast the workload.

References

Abdollahi Vayghan, L., Saied, M. A., Toeroe, M., & Khendek, F. (2019). Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes. *Proceedings - 19th IEEE International Conference on Software Quality, Reliability and Security, QRS 2019*, 176–185. https://doi.org/10.1109/QRS.2019.00034

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*(3), 1448–1460. https://doi.org/10.1109/TSC.2020.2995937

Adkins, H., Beyer, B., Blankinship, P., Lewandowski, P., Stubblefield, O., & Stubblefield, A. (2020). *Building Secure and Reliable Systems*. O'Reilly Media, Inc.

Ali-Eldin, A., Seleznjev, O., Sjostedt-de Luna, S., Tordsson, J., & Elmroth, E. (2014). Measuring Cloud Workload Burstiness. *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, 566–572. https://doi.org/10.1109/UCC.2014.87

Apache Cassandra. (2025). *Cassandra Documentation - Overview*. https://cassandra.apache.org/doc/latest/cassandra/architecture/overview.html

Baarzi, A. F., & Kesidis, G. (2021). SHOWAR: Right-sizing and efficient scheduling of microservices. *SoCC 2021 - Proceedings of the 2021 ACM Symposium on Cloud Computing*, 427–441. https://doi.org/10.1145/3472883.3486999

Becker, S., Schmidt, F., Gulenko, A., Acker, A., & Kao, O. (2021). *Towards AIOps in Edge Computing Environments*. http://arxiv.org/abs/2102.09001

Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (2016). *Site Reliability Engineering: How Google Runs Production Systems*. https://landing.google.com/sre/sre-book/toc/

Bhatnagar, S., & Mahant, R. (2025). The art of decoding microservices: An in-depth exploration of modern software architecture. In *The Art of Decoding Microservices: An In-Depth Exploration of Modern Software Architecture*. Apress Media LLC. https://doi.org/10.1007/979-8-8688-1267-5

Bodik, P., Patterson, D. A., Jordan, M. I., Franklin, M. J., & Fox, A. (2010). Characterizing, modeling, and generating workload spikes for stateful services. *Proceedings of the 1st ACM Symposium on Cloud Computing*, 241–252.

Borrill, P. (2026). *Circumventing the CAP Theorem with Open Atomic Ethernet*. http://arxiv.org/abs/2602.21182

Brewer, E. (2012). CAP twelve years later: How the ”rules” have changed. *Computer*, *45*(February), 23–29. https://doi.org/10.1109/MC.2012.37

Campbell, L., & Majors, C. (2017). *Database Reliability Engineering* (1st ed.). O'Reilly Media.

Cao, R., Yu, Z., Marbach, T., Li, J., Wang, G., & Liu, X. (2018). Load Prediction for Data Centers Based on Database Service. *Proceedings - International Computer Software and Applications Conference*, *1*, 728–737. https://doi.org/10.1109/COMPSAC.2018.00109

Chae, M. S., Lee, H. M., & Lee, K. (2019). A performance comparison of linux containers and virtual machines using Docker and KVM. *Cluster Computing*, *22*(s1), 1765–1775. https://doi.org/10.1007/s10586-017-1511-2

Cloud Native Computing Foundation. (2021). *CNCF Annual Survey 2021*. https://www.cncf.io/reports/cncf-annual-survey-2021/

Cockroach Labs. (2026). *Data Resilience*. https://www.cockroachlabs.com/docs/stable/data-resilience

Codership Ltd. (2025). *Replication API*. Galeracluster.Com. https://galeracluster.com/library/documentation/architecture.html#wsrep-api

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010). Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10*, 143–154. https://doi.org/10.1145/1807128.1807152

Coriani, S. (2021). *Introducing Configurable Retry Logic in Microsoft.Data.SqlClient v3.0.0-Preview1*. Azure SQL Database Devs' Corner. https://devblogs.microsoft.com/azure-sql/configurable-retry-logic-for-microsoft-data-sqlclient/

CouchDB. (2026). *CouchDB - Technical Overview*. https://docs.couchdb.org/en/stable/intro/overview.html

Dai, H. (2012). Effective apply of design pattern in database-based application development. *Proceedings - 4th International Conference on Computational and Information Sciences, ICCIS 2012*, 558–561. https://doi.org/10.1109/ICCIS.2012.138

Dakić, V., Kovač, M., & Slovinac, J. (2024). Evolving High-Performance Computing Data Centers with Kubernetes, Performance Analysis, and Dynamic Workload Placement Based on Machine Learning Scheduling. *Electronics (Switzerland)*, *13*(13). https://doi.org/10.3390/electronics13132651

Dang, Y., Lin, Q., & Huang, P. (2019). AIOps: Real-world challenges and research innovations. *Proceedings - 2019 IEEE/ACM 41st International Conference on Software*

Engineering: Companion, ICSE-Companion 2019, 4–5. https://doi.org/10.1109/ICSE-Companion.2019.00023

Dang-Quang, N. M., & Yoo, M. (2021). Deep learning-based autoscaling using bidirectional long short-term memory for Kubernetes. *Applied Sciences (Switzerland)*, *11*(9). https://doi.org/10.3390/app11093835

Delnat, W., Truyen, E., Rafique, A., Van Landuyt, D., & Joosen, W. (2018). *K8-scalar*. 33–39. https://doi.org/10.1145/3194133.3194162

Depoutovitch, A., Chen, C., Chen, J., Larson, P., Lin, S., Ng, J., Cui, W., Liu, Q., Huang, W., Xiao, Y., & He, Y. (2020). Taurus Database: How to be Fast, Available, and Frugal in the Cloud. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1463–1478. https://doi.org/10.1145/3318464.3386129

Deshpande, U. (2019). Caravel: Burst tolerant scheduling for containerized stateful applications. *Proceedings - International Conference on Distributed Computing Systems*, *2019-July*, 1432–1442. https://doi.org/10.1109/ICDCS.2019.00143

Deshpande, U., Linck, N., & Seshadri, S. (2021). Self-service data protection for stateful containers. *HotStorage 2021 - Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, 71–76. https://doi.org/10.1145/3465332.3470876

EDB. (2025). *EDB Postgres Distributed (PGD)*. https://www.enterprisedb.com/docs/pgd/latest/

Georgiou, M. A., Paphitis, A., Sirivianos, M., & Herodotou, H. (2019). Towards autoscaling existing transactional databases with strong consistency. *Proceedings - 2019 IEEE 35th International Conference on Data Engineering Workshops, ICDEW 2019*, (i), 107–112. https://doi.org/10.1109/ICDEW.2019.00-26

Good, B. (2019). *To run or not to run a database on Kubernetes: What to consider*. Google Cloud. https://cloud.google.com/blog/products/databases/to-run-or-not-to-run-a-database-on-kubernetes-what-to-consider

Google LLC. (2023). *State of DevOps Report 2023*. Google. https://services.google.com/fh/files/misc/2023_final_report_sodr.pdf

Gorton, I. (2022). *Foundations of Scalable Systems*. http://oreilly.com

Govardhana Miriyala Kannaiah. (2024). *Kubernetes Anti-Patterns: Overcome common pitfalls to achieve optimal deployments and a flawless Kubernetes ecosystem* (1st ed.). Packt Publishing.

HAProxy Technologies. (2026). *What is HAProxy?* https://www.haproxy.com/glossary/what-is-haproxy

Hauer, T., Hoffman, P., Lunney, J., Ardelean, D., & Diwan, A. (2020). Meaningful Availability. *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 545–557. https://www.usenix.org/conference/nsdi20/presentation/hauer

Hohenstein, U., Jaeger, M. C., & Bluemel, M. (2009). Improving connection pooling persistence systems. *Proceedings of the 1st International Conference on Intensive Applications and Services, INTENSIVE 2009*, 71–77. https://doi.org/10.1109/INTENSIVE.2009.18

Hong, S., Li, D., & Huang, X. (2019). Database docker persistence framework based on swarm and ceph. *ACM International Conference Proceeding Series*, 249–253. https://doi.org/10.1145/3341069.3342985

Iqbal, W., Erradi, A., & Mahmood, A. (2018). Dynamic workload patterns prediction for proactive auto-scaling of web applications. *Journal of Network and Computer Applications*, *124*(September), 94–107. https://doi.org/10.1016/j.jnca.2018.09.023

ISO/IEC. (2011). *ISO/IEC 25010:2011 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35733

Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J., & Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, *35*(3), 24–35. https://doi.org/10.1109/MS.2018.2141039

Jindal, A., Podolskiy, V., & Gerndt, M. (2019). Performance modeling for cloud microservice applications. *ICPE 2019 - Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, 25–32. https://doi.org/10.1145/3297663.3310309

Kim, B. G., Humble, J., Debois, P., & Willis, J. (2016). *The DevOps handbook* (1st ed.). IT Revolution Press, LLC.

Kim, E., Lee, K., & Yoo, C. (2021). *On the Resource Management of Kubernetes*. 154–158. https://doi.org/10.1109/icoin50884.2021.9333977

Kim, J., Salem, K., Daudjee, K., Aboulnaga, A., & Pan, X. (2015). Database high availability using SHADOW systems. *ACM SoCC 2015 - Proceedings of the 6th ACM Symposium on Cloud Computing*, 209–221. https://doi.org/10.1145/2806777.2806841

Kleppmann, M. (2017). Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems. In A. Spencer & M. Beaugureau (Eds.), *O'Reilly Media, Inc.* O'Reilly Media, Inc. https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/%0Ahttp://shop.oreilly.com/product/0636920032175.do

Kubernetes. (2021a). *Persistent Volumes*. Kubernetes.Io. https://kubernetes.io/docs/concepts/storage/persistent-volumes/

Kubernetes. (2021b). *Pods*. Kubernetes.Io. https://kubernetes.io/docs/concepts/workloads/pods/

Kubernetes. (2021c). *ReplicaSet*. Kubernetes.Io. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

Kubernetes. (2021d). *StatefulSet*. Kubernetes.Io. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

Kubernetes. (2025a). *Assign Memory Resources to Containers and Pods*. https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/

Kubernetes. (2025b). *Run a Replicated Stateful Application*. Kubernetes.Io. https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/

Kubernetes. (2025c). *What is Kubernetes?* Kubernetes.Io. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

Lassnig, M., Fahringer, T., Garonne, V., Molfetas, A., & Branco, M. (2010). Identification, modelling and prediction of non-periodic bursts in workloads. *CCGrid 2010 - 10th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing*, 485–494. https://doi.org/10.1109/CCGRID.2010.118

Liu, F. (2012). A method of design and optimization of database connection pool. *Proceedings of the 2012 4th International Conference on Intelligent Human-Machine Systems and Cybernetics, IHMSC 2012*, *2*, 272–274. https://doi.org/10.1109/IHMSC.2012.161

Lyu, Y., Li, H., Sayagh, M., Jiang, Z. M., & Hassan, A. E. (2021). An Empirical Study of the Impact of Data Splitting Decisions on the Performance of AIOps Solutions. *ACM Transactions on Software Engineering and Methodology*, *30*(4). https://doi.org/10.1145/3447876

Mann, A., Stahnke, M., Brown, A., & Kersten, N. (2018). State of DevOps Report 2018. In *2018*. https://media.webteam.puppet.com/uploads/2019/11/Puppet-State-of-DevOps-Report-2018_update.pdf?_ga=2.109761029.1653166723.1593996394-1831254965.1593996394

Mann, A., Stahnke, M., Brown, A., & Kersten, N. (2019). *State of DevOps Report 2019. F5*. https://cdn.studio.f5.com/files/k6fem79d/production/9f7c0b36240d70de-aac22fe202f4d69c17e60b64.pdf

Manouvrier, M., Pautasso, C., & Rukoz, M. (2020). Microservice disaster crash recovery: A weak global referential integrity management. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): 12138 LNCS*. Springer International Publishing. https://doi.org/10.1007/978-3-030-50417-5_36

Marinho, C. S. S., Moreira, L. O., Coutinho, E. F., Filho, J. S. C., Sousa, F. R. C., & Machado, J. C. (2018). LABAREDA : A Predictive and Elastic Load Balancing Service for Cloud-Replicated Databases. *J. Inf. Data Manag.*, *9*(1), 94–106.

Mauri, D., Coriani, S., Hoffman, A., Mishra, S., & Popovic, J. (2021). Practical Azure SQL Database for Modern Developers. In *Practical Azure SQL Database for Modern Developers*. https://doi.org/10.1007/978-1-4842-6370-9

McAfee, A., & Brynjolfsson, E. (2008). Investing in the IT that Makes a Competitive Difference. *Harvard Business Review*, *86*, 98–107. https://hbr.org/2008/07/investing-in-the-it-that-makes-a-competitive-difference

Microsoft. (2025). *Recommended alert rules for Kubernetes clusters*. Azure Monitor Documentation. https://learn.microsoft.com/en-us/azure/azure-monitor/containers/kubernetes-metric-alerts?tabs=portal#pod-level-alerts

Microsoft Inc. (2022). *Retry pattern*. Microsoft Technical Documentation. https://docs.microsoft.com/en-us/azure/architecture/patterns/retry

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems* (1st ed.). O'Reilly Media.

O'Connor, P. D. T., & Kleyner, A. (2012). *Practical Reliability Engineering* (5th ed.). John Wiley & Sons, Ltd.

Oracle. (2025). *How MySQL Uses Memory*. MySQL.Com. https://dev.mysql.com/doc/refman/8.0/en/memory-use.html

Pardon, G., Pautasso, C., & Zimmermann, O. (2018). Consistent Disaster Recovery for Microservices: The BAC Theorem. *IEEE Cloud Computing*, *5*(1), 49–59. https://doi.org/10.1109/MCC.2018.011791714

Percona LLC. (2025). *Percona XtraDB Cluster Documentation*. https://learn.percona.com/hubfs/Documentation/MySQL/XtraDB-Cluster/Percona-XtradbCluster-8.4.pdf

Percona LLC. (2021). *Percona Distribution for MySQL Operator*. Percona.Com. https://www.percona.com/doc/kubernetes-operator-for-pxc/index.html

Percona LLC. (2025). *MySQL Memory Usage: A Guide to Optimization*. https://percona.community/blog/2025/11/11/mysql-memory-usage-a-guide-to-optimization/

Pereira, P. A., & Bruce, M. (2019). *Microservices In Action*. Manning.

Petrov, A. (2019). *Database Internals* (1st ed.). O'Reilly Media, Inc.

Podolskiy, V., Mayo, M., Koey, A., Gerndt, M., & Patros, P. (2019). Maintaining SLOs of Cloud-Native Applications Via Self-Adaptive Resource Sharing. *International Conference on Self-Adaptive and Self-Organizing Systems, SASO*, *2019-June*(June), 72–81. https://doi.org/10.1109/SASO.2019.00018

Prabhu, A. (2024). Integrating Site Reliability Engineering SRE for Effective Product Development: A Focus on SLAs, SLIs, and SLOs. *International Journal of Science and Research (IJSR)*, *13*(9), 228–230. https://doi.org/10.21275/sr24902093845

ProxySQL. (2026). *ProxySQL Architecture Overview*. Proxysql.Com. https://www.proxysql.com/documentation/architecture

ProxySQL. (2021). *Main (runtime tables definition)*. ProxySQL.Com. https://proxysql.com/documentation/main-runtime/

Rossi, F., Cardellini, V., Lo Presti, F., & Nardelli, M. (2020). Geo-distributed efficient deployment of containers with Kubernetes. *Computer Communications*, *159*(April), 161–174. https://doi.org/10.1016/j.comcom.2020.04.061

Rossi, F., Nardelli, M., & Cardellini, V. (2019). Horizontal and vertical scaling of container-based applications using reinforcement learning. *IEEE International Conference on Cloud Computing, CLOUD*, *2019-July*(Section V), 329–338. https://doi.org/10.1109/CLOUD.2019.00061

Rubak, A., & Taheri, J. (2023, December 4). Machine Learning for Predictive Resource Scaling of Microservices on Kubernetes Platforms. *16th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2023*. https://doi.org/10.1145/3603166.3632165

Sampaio, A. R., Rubin, J., Beschastnikh, I., & Rosa, N. S. (2019). Improving microservice-based applications with runtime placement adaptation. *Journal of Internet Services and Applications*, *10*(1). https://doi.org/10.1186/s13174-019-0104-0

Seybold, D., Hauser, C. B., Volpert, S., & Domaschka, J. (2017). Gibbon: An availability evaluation framework for distributed databases. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *10574 LNCS*, 31–49. https://doi.org/10.1007/978-3-319-69459-7_3

Seybold, D., Wesner, S., & Domaschka, J. (2020). King Louie: Reproducible availability benchmarking of cloud-hosted DBMS. *Proceedings of the ACM Symposium on Applied Computing*, 144–153. https://doi.org/10.1145/3341105.3373968

Shen, H., & Chen, L. (2018). Resource Demand Misalignment: An Important Factor to Consider for Reducing Resource Over-Provisioning in Cloud Datacenters.

IEEE/ACM Transactions on Networking, *26*(3), 1207–1221. https://doi.org/10.1109/TNET.2018.2823642

Sippu, S., & Soisalon-Soininen, E. (2017). Transaction Processing: Management of the Logical Database and its Underlying Physical Structure. In *Computer Science Handbook, Second Edition*. https://doi.org/10.1007/978-3-319-12292-2

Soualhia, M., Fu, C., & Khomh, F. (2019). Infrastructure fault detection and prediction in edge cloud environments. *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, SEC 2019*, 222–235. https://doi.org/10.1145/3318216.3363305

Tamer, M., & Patrick Valduriez, Ö. ·. (n.d.). *Principles of Distributed Database Systems*.

Thanh, T. D., Mohan, S., Choi, E., SangBum, K., & Kim, P. (2008). A taxonomy and survey on distributed file systems. *Proceedings - 4th International Conference on Networked Computing and Advanced Information Management, NCM 2008*, *1*, 144–149. https://doi.org/10.1109/NCM.2008.162

To, Q. C., Soto, J., & Markl, V. (2018). A survey of state management in big data processing systems. *The VLDB Journal -The International Journal on Very Large Data Bases*, *27*(6), 847–872.

Toka, L., Dobreff, G., Fodor, B., & Sonkoly, B. (2021). Machine Learning-based Scaling Management for Kubernetes Edge Clusters. *IEEE Transactions on Network and Service Management*, *4537*(c), 1–14. https://doi.org/10.1109/TNSM.2021.3052837

Truyen, E., Van Landuyt, D., Lagaisse, B., & Joosen, W. (2019). Performance overhead of container orchestration frameworks for management of multi-tenant database deployments. *Proceedings of the ACM Symposium on Applied Computing*, *Part F1477*, 156–159. https://doi.org/10.1145/3297280.3297536

Tusa, M. (2021). *Percona Kubernetes Operator for Percona XtraDB Cluster: HAProxy or ProxySQL?* Percona. https://www.percona.com/blog/2021/01/11/percona-kubernetes-operator-for-percona-xtradb-cluster-haproxy-or-proxysql/

Vitess. (2022). *What Is Vitess*. Vitess.Io. https://vitess.io/docs/overview/whatisvitess/

Wang, Z., Li, P., Zurich, E., Mike Liang, C.-J., Research, M., Wu, F., & Yan, F. Y. (2024). Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices. *21st USENIX Symposium on Networked Systems Design and Implementation*, 149–165. https://www.usenix.org/conference/nsdi24/presentation/wang-zibo

Woodruff, J., Moore, A. W., & Zilberman, N. (2019). Measuring Burstiness in Data Center Applications. *Proceedings of the 2019 Workshop on Buffer Sizing*, 1–6. https://doi.org/10.1145/3375235.3375240

Xie, S., Wang, J., Li, B., Zhang, Z., Li, D., & Hung, P. C. K. (2024). PBScaler: A Bottleneck-Aware Autoscaling Framework for Microservice-Based Applications. *IEEE Transactions on Services Computing*, *17*(2), 604–616. https://doi.org/10.1109/TSC.2024.3376202

Yang, J. (2025). Cloud Computing and MicroServices. In *Cloud Computing and Micro-Services*. Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-93478-0

Yang, Y., & Chen, L. (2019). Design of Kubernetes Scheduling Strategy Based on LSTM and Grey Model. *Proceedings of IEEE 14th International Conference on Intelligent Systems and Knowledge Engineering, ISKE 2019*, 701–707. https://doi.org/10.1109/ISKE47853.2019.9170419

Zalando. (2022). *Patroni*. Readthedocs.Com. https://patroni.readthedocs.io/en/latest/

Zamanian, E., Yu, X., Stonebraker, M., & Kraska, T. (2018). Rethinking database high availability with RDMA networks. *Proceedings of the VLDB Endowment*, *12*(11), 1637–1650. https://doi.org/10.14778/3342263.3342639

Zhang, X., Wang, T., Ma, L., & Mahadevan, S. (2025). Reliability engineering, risk management, and trustworthiness assurance for AI systems. *Journal of Reliability Science and Engineering*, *1*(2), 022001. https://doi.org/10.1088/3050-2454/adcef2

List of Scientific Publications by the Author on the Topic of the Dissertation

Papers in the Reviewed Scientific Journals

Pakrijauskas K, & Mažeika D. (2022a). A Method of Transparent Graceful Failover in Low Latency Stateful Microservices. *Electronics, 11*(23), 3936. https://doi.org/10.3390/electronics11233936

Pakrijauskas K, & Mažeika D. (2025). A Rule-Based Method for Enhancing Burst Tolerance in Stateful Microservices. *Electronics, 14*(14), 2752. https://doi.org/10.3390/electronics14142752

Papers in Other Editions

Pakrijauskas, K., & Mažeika, D. (2021). On Recent Advances on Stateful Orchestrated Container Reliability. 2021 IEEE Open Conference of Electrical, Electronic and Information Sciences (EStream), 1–6. https://doi.org/10.1109/eStream53087.2021.9431489

Pakrijauskas, K., & Mažeika, D. (2022). Investigation of Stateful Microservice Availability During Failover. 2022 8th International Conference on Control, Decision and Information Technologies (CoDIT), 286–290. https://doi.org/10.1109/CoDIT55151.2022.9804162

Summary in Lithuanian

Įvadas

Problemos formulavimas

Šiandien programinė įranga ir technologijos yra pagrindiniai veiksniai, lemiantys organizacijų transformaciją ir vertės teikimą jų klientams bei suinteresuotosioms šalims. Investicijos į technologijas gali suteikti konkurencinį pranašumą (McAfee & Brynjolfsson, 2008). IT sistemų mastas ir jų svarba verslui nuolat auga.

Didelėse, paskirstytose ir sudėtingose sistemose patikimumo užtikrinimas yra svarbus uždavinys – sėkmė yra tiesiogiai susijusi su tuo, ar jo sistemos veikia. Tokios technologijų milžinės kaip „Google“ negali sau leisti, kad sistemos ilgą laiką neveiktų. Sistemos augimo greitis, pokyčių skaičius ir prieinamumo poreikis paskatino „Google“ taikyti programinės įrangos inžinerijos praktikas, kad išspręstų savo infrastruktūros ir veiklos problemas.

Mikroservisai leidžia neatsilikti nuo poreikio didinti mastą ir naujų technologijų atsiradimo tempo. Mikroservisų architektūra, kaip paslaugomis grindžiamos architektūros (angl. *Service-Oriented Architecture*) įgyvendinimo forma, susideda iš daugelio nepriklausomų ir autonominių komponentų.

Priežiūros ir gyvavimo ciklo valdymo veikla turi įtakos būsenas išlaikančių mikroservisų prieinamumui. Vertikalus masteliavimas, programinės įrangos atnaujinimai gali pareikalauti prastovos ir todėl gali tekti naudoti ribotą prieinamumo biudžetą. Nors būsenas išlaikančių mikroservisų elastingumas koordinuotai valdomose konteinerių sistemose

yra didelis, netinkamas skaičiavimo išteklių paskirstymas gali neigiamai paveikti paslaugos prieinamumą, jei darbo krūvis staiga ir reikšmingai padidėja. Skiriant skaičiavimo išteklių su didesniu rezervu taip pat yra variantas, tačiau tas rezervas gali būti niekada nepanaudotas.

Darbo aktualumas

Šiuolaikinės sistemos tampa vis sudėtingesnės ir pasiekia anksčiau neįsivaizduojamą mastą. Be to, didėja poreikis užtikrinti tokių sistemų aukštą prieinamumą, nes vis daugiau įmonių sėkmingumas priklauso nuo jų IT sistemų (Mann et al., 2018, 2019).

Poreikis didinti programinės įrangos patikimumą ir kodo leidimo greitį didėja dėl vis plačiau taikomų „DevOps“ praktikų (Mann et al., 2018, 2019). Programuotojų komandos įgalina organizacijas pasiekti iškeltus tikslus lengviau, skirdamos daugiau laiko programinės įrangos kūrimui ir pašalindamos apribojimus (Kim et al., 2016). Kodo testavimo ar pristatymo apribojimas pašalinamas, jei sumažėja poreikis atlikti tam tikrus veiksmus rankiniu būdu ar laukti, kol bus užbaigti tam tikri procesai. Konteineriai ir „Kubernetes“ yra plačiai naudojamos technologijos mikroservisuose.

Mikroservisais pagrįstos sistemos neveikimo laikas sumažėja, jei būsenas išlaikantys mikroservisai grįžta į tinklą laiku ir nuosekliai. Nors būsenos neturinčių mikroservisų atkūrimas apsiriboja perkrovimu, būsenas išlaikantis mikroservisas gali reikalauti duomenų atkūrimo.

Būsenas išlaikantys mikroservisai saugo ir apdoroja svarbų ir vertingą turtą – duomenis. Būsenas išlaikančių mikroservisų atsparumui įtaką gali daryti įvairūs veiksniai: duomenų dydis, saugojimo ir tinklo greitis, duomenų tipas, šifravimo ir suspaudimo algoritmai, fragmentavimas ir klasterizavimas, geografinis pasiskirstymas. Atsižvelgiant į paskirstytus ir dažnai naudojamas įvairias duomenų saugojimo technologijas, laiku atkurti duomenis yra svarbu, kad būtų išlaikytas nuoseklumas tarp kelių mikroservisų.

Trumpesnis prastovos laikas leidžia organizacijai išlikti konkurencingai rinkoje arba, kaip viešajai organizacijai, teikti geresnes paslaugas. Sumažėjus darbo jėgos poreikiui, komandos taps mažesnės, bet dirbs greičiau nei didesnės (Newman, 2015). Sutaupytas prieinamumo biudžetas gali būti panaudotas kitiems tikslams, pavyzdžiui, atnaujinimams ir kitoms tobulinimo priemonėms įgyvendinti (Campbell & Majors, 2017).

Tyrimo objektas

Būsenas išlaikančių mikroservisų atkuriamumas ir atsparumas gedimams.

Darbo tikslas

Sukurti ir patvirtinti metodus bei technikas, skirtas padidinti būsenas išlaikančių mikroservisų patikimumą ir prieinamumą koordinuotai valdomose konteinerių sistemose.

Darbo uždaviniai

Darbo tikslui pasiekti sprendžiami šie uždaviniai:

1. Atlikti literatūros apžvalgą apie būsenas išlaikančių mikroservisų patikimumą ir prieinamumą koordinuotai valdomose konteinerių sistemose.
2. Išanalizuoti ir įvertinti veiksnius, darančius įtaką būsenas išlaikančių mikroservisų prieinamumui mazgų perjungimo metu.
3. Pasiūlyti metodą, kuris leistų atlikti būsenas išlaikančių mikroservisų priežiūros darbus, turinčius nežymų poveikį klientų programoms.
4. Pasiūlyti metodą, kuris leistų prailginti veikimo laiką ir sumažinti nesėkmingų užklausų skaičių esant staigiai ir reikšmingai padidėjusiai apkrovai būsenas išlaikančiuose mikroservisuose.

Tyrimų metodika

Nagrinėjant darbo objektą, taikyti šie metodai:

- Atlikta analitinė literatūros apžvalga apie būsenas išlaikančius mikroservisus, būsenos valdymo iššūkius paskirstytose arba konteinerių koordinuoto valdymo sistemose, mašininio mokymo ir taisyklėmis pagrįstus metodus, skirtus būsenas išlaikančių mikroservisų prieinamumui pagerinti. Nustatyti trūkumai, susiję su būseną išlaikančių mikroservisų prieinamumo užtikrinimu.
- Statistinė tiriamoji analizė buvo taikoma siekiant nustatyti ir įvertinti veiksnius, darančius įtaką būsenas išlaikančių mikroservisų prieinamumui perjungimo metu.
- Taikytas eksperimentinis tyrimo metodas. Pirma, siekiant įvertinti siūlomą metodą, skirtą padidinti būseną išlaikančių mikroservisų prieinamumą perjungimo metu. Antra, siekiant patvirtinti ir įvertinti siūlomą taisyklėmis pagrįstą metodą, skirtą padidinti tokių mikroservisų atsparumą staigiam ir reikšmingam apkrovos augimui. Apkrova būseną išlaikantiems mikroservisams generuota naudojant dedikuotą ir YCSB įrankį. Eksperimentinė aplinka sukurta „Google Cloud Platform“ debesijos platformoje. Duomenys buvo renkami ir analizuojami eksperimentų metu, siekiant įvertinti siūlomo metodo veiksmingumą pagal iš anksto nustatytus kriterijus.

Darbo mokslinis naujumas

Šio informatikos inžinerijos tyrimo mokslinis naujumas apibūdinamas taip:

1. Pasiūlytas naujas metodas, leidžiantis užtikrinti aukštą būseną išlaikančių mikroservisų prieinamumą techninės priežiūros operacijų metu. Šis metodas leidžia pasiekti artimą nuliui prieinamumo praradimą, perjungiant mikroserviso mazgus techninės priežiūros tikslais.
2. Pasiūlytas taisyklėmis pagrįstas metodas, kartu su apkrovos balansavimu, leidžia pagerinti būsenas išlaikančio mikroserviso atsparumą staigiai ir reikšmingai padidėjus apkrovai. Metodas įgalina būsenas išlaikantį mikroservisą apdoroti užklausas beveik dvigubai ilgiau staigiai ir reikšmingai padidėjusiai apkrovai dėl kurios, kitu atveju, mikroservisas taptų nepasiekiamas.

Darbo rezultatų praktinė reikšmė

Siūlomi naujieji metodai įgalina gerokai padidinti būsenas išlaikančių mikroservisų patikimumą, prieinamumą ir masteliavimo galimybę. Šie sprendimai sumažina prastovas ir optimizuoja išteklių naudojimą, todėl sutaupoma lėšų ir padidinamas išteklių naudojimo efektyvumas išlaikant aukštą mikroservisų sistemos pasiekiamumą.

Galimybė sklandžiai integruoti siūlomus metodus su šiuolaikinėmis konteinerių koordinuoto valdymo sistemomis, tokiomis kaip „Kubernetes“ ir „Docker Swarm“, įgalina ir užtikrina sklandų sistemų masteliavimą bei IT infrastruktūros atitikimą ateities poreikiams. Tokiose verslo šakose kaip sveikatos apsauga, finansai ir elektroninė prekyba, kur nuolatinis prieinamumas ir našumas yra labai svarbūs, šie pasiekimai užtikrina sklandų sistemų veikimą ir geresnę naudotojų patirtį. Be to, tyrimas sudaro pagrindą tolesnėms inovacijoms ir geriausios praktikos diegimui paskirstytų sistemų valdymo srityje, prisidedant prie platesnės debesų kompiuterijos ir mikroservisų architektūros panaudojimo.

Ginamieji teiginiai

Šie teiginiai, grindžiami šio tyrimo rezultatais, gali būti laikomi oficialiomis hipotezėmis, kurias reikės pagrįsti:

1. OSI lygis, kuriame veikia apkrovos balansavimo įrenginys, kartu su prisijungimo tipu daro didžiausią įtaką būsenas išlaikančių mikroservisų prieinamumui gedimo atveju. Apkrovos balansavimo įrenginys, veikiantis 7-ajame OSI lygyje, daro mažesnę įtaką prieinamumui nei įrenginys, veikiantis 4-ajame OSI lygyje.
2. Siūlomas skaidraus gedimo perjungimo mažai susietuose mikroservisuose metodas, naudojantis 7-ajame OSI lygyje veikiantį apkrovos balansavimo įrenginį ir priverstinio prisijungimo nutraukimo mechanizmą, gali sumažinti sklandaus mazgų perjungimo poveikį būsenas išlaikančiam mikroservisui iki nereikšmingų verčių.
3. Siūlomas būseną išlaikančių mikroservisų atsparumo staigiam ir reikšmingam apkrovos padidėjimui metodas, pagrįstas apkrovos balansavimu ir rašymo užklausų masteliavimu, leidžia sumažinti tokio apkrovos augimo poveikį mikroserviso prieinamumui.

Darbo rezultatų aprobavimas

Disertacijos rezultatai buvo paskelbti dviejuose recenzuojamuose mokslo žurnaluose, įtrauktuose į *Clarivate Analytics Web of Science* duomenų bazę ir turinčiuose citavimo rodiklį, taip pat dviejuose konferencijų leidiniuose.

Disertacijoje atliktų tyrimų rezultatai paskelbti dviejose mokslinėse konferencijose Lietuvoje ir užsienyje:

- eStream 2021: *IEEE Open Conference of Electrical, Electronic and Information,* 2021 m. balandžio 22 d., Vilniuje, Lietuvoje.
- CoDIT 2022: *8th International Conference on Control, Decision and Information Technologies,* 2022 m. gegužės 17–20 d., Stambule, Turkijoje.

Disertacijos struktūra

Disertaciją sudaro įvadas, keturi pagrindiniai skyriai, bendrosios išvados, literatūros šaltinių sąrašas, disertacijos autoriaus publikacijų sąrašas ir santrauka lietuvių kalba. Disertacijos apimtis: 115 puslapių, 3 formulės, 32 paveikslai, 8 lentelės ir 102 literatūros šaltiniai.

1. Būsenas išlaikančių mikroservisų patikimumo didinimo sprendimų literatūros apžvalga

Pirmajame disertacijos skyriuje pateikiama būsenas išlaikančių mikroservisų apžvalga koordinuotai valdomose konteinerių sistemose, pabrėžiant iššūkius ir sprendimus, susijusius su jų valdymu, patikimumu ir našumu.

Mikroservisų architektūra įgalima šiuolaikinės programų sistemas efektyviai masteliuotis, diegti ir atnaujinti esamus servisus. Skirtingai nuo būsenos neturinčių mikroservisų, būsenas išlaikantys mikroservisai saugo ir apdoroja duomenis. Iš to kyla diegimo, valdymo, mastelio keitimo ir duomenų replikacijos iššūkių. Konteinerių koordinuoto valdymo sistemos, tokios kaip „Kubernetes“, „Docker Swarm“ ir „Mesos“, yra sukurtos šiems iššūkiams suvaldyti automatizuojant tokias užduotis kaip servisų paieška (angl. *Service discovery*), duomenų saugyklos valdymas ir patikimumo užtikrinimas (Jamshidi et al., 2018; Kubernetes, 2025c). Patikimumas, apibrėžiamas kaip tikimybė atlikti reikiamą funkciją be gedimų, yra labai svarbus ir matuojamas naudojant paslaugos veikimo lygio indikatorius (SLI), paslaugos veikimo lygio tikslus (SLO) ir paslaugos teikimo lygio sutartis (SLA) (Beyer et al., 2016; O'Connor & Kleyner, 2012).

Istoriniai SLI ir SLO yra būtini priimant duomenimis grįstus sprendimus, skirtus prižiūrėti mikroservisus: įvertinti, tobulinti ir tvarkyti. SLI gali būti gauti iš monitorinimo, įvykių žurnalo ar sekimo įrašų. Kiekvienas prižiūrimas komponentas turi valdomas, iš dalies valdomas ir nevaldomas savybes (Podolskiy et al., 2019). Išankstinis duomenų paruošimas ir apdorojimas yra labai svarbūs norint efektyviai priimti sprendimus naudojant šiuos indikatorius (Bodik et al., 2010; Jindal et al., 2019; Newman, 2015; Podolskiy et al., 2019; Soualhia et al., 2019).

Duomenimis pagrįsti metodai, tokie kaip mašininis mokymasis ir sustiprinimo mokymasis, naudojami gedimams numatyti ir išteklių paskirstymui optimizuoti (Jindal et al., 2019; Rossi et al., 2020). SLI ir SLO numatyti naudojami tokie metodai kaip Lasso regresija ir atsitiktiniai miškai, o sustiprinimo mokymasis padeda dinamiškai planuoti išteklius (Soualhia et al., 2019).

Taisyklėmis grįsti metodai, įskaitant tinkintus planuoklius ir būsenos valdiklius, padidina būsenas išlaikančių mikroservisų patikimumą. Pasirinktiniai planuokliai, tokie kaip „ge-kube“, optimizuoja „Kubernetes“ podų išdėstymą atsižvelgiant į turimus išteklius ir tinklo delsą (Rossi et al., 2020). Būsenos valdikliai integruoja aktyvias ir budėjimo būsenas, kad pagerintų prieinamumą (Abdollahi Vayghan et al., 2019).

Duomenų bazės prieinamumas užtikrinamas naudojant duomenų replikacijos strategijas, tokias kaip vieno pagrindinio ir kelių pagrindinių mazgų replikacija. Vieno pagrindinio mazgo replikacija sumažina nuoseklumo konfliktus, tačiau gedimo atveju reikalingas laikas naujo pagrindinio mazgo rinkimui. Kelių pirminių mazgų replikacija leidžia

keliems mazgams tvarkyti įrašus, sumažinant prastovas, tačiau reikalaujant konfliktų sprendimo (Campbell & Majors, 2017; Percona LLC, 2025; Seybold et al., 2020).

Nelaukti užklausų šuoliai – staigiai ir reikšmingai padidėjusi apkrova, gali sumažinti tiekiamos paslaugos kokybę ir sudaryti sąlygas SLO pažeidimams (Ali-Eldin et al., 2014; Bodik et al., 2010; Xie et al., 2024). Su staigiai ir reikšmingai padidėjusia apkrova galima tvarkytis statiškai, skiriant pakankamai išteklių, arba dinamiškai, numatant apkrovą ir atitinkamai keičiant išteklius. Staigus ir reikšmingas apkrovos padidėjimas gali išeikvoti sistemos išteklius. Didžiausią įtaką prieinamumui turi operatyviosios atminties išeikvojimas (Kubernetes, 2025a; Oracle, 2025; Percona LLC, 2025).

2. Veiksnių, lemiančių būseną išlaikančių mikroservisų prieinamumą perjungimo metu, analizė

Šiame disertacijos skyriuje vertinami veiksniai, darantys įtaką būsenas išlaikančių mikroservisų prieinamumui perjungimo metu koordinuotai valdomose konteinerių sistemose.

Nors konteinerių koordinuoto valdymo sistemos leidžia supaprastinti mikroservisų priežiūros darbus, būseną išlaikančių mikroservisų atnaujinimai, diegimai ir kiti pakeitimai daro didesnę įtaką jų prieinamumui, palyginti su būsenos neturinčiais mikroservisais. Tyrime buvo vertinamos būsenas išlaikančio mikroserviso nevaldomos savybės, kurioms ne visada galima daryti įtaką: krūvio intensyvumas, apkrovos balansavimo įrankio OSI lygis, prisijungimo tipas ir užklausos trukmė. Sutelkti prisijungimai (angl. *Pooled connection*) sumažina delsą, bet padidina sistemos sudėtingumą; darbo krūvio intensyvumas ir užklausų trukmė priklauso nuo klientų, o apkrovos balansavimo įrankio OSI lygis turi įtakos užklausų nukreipimui.

Atliekamas pasiekiamumo tyrimas kelių pagrindinių mazgų reliacinės duomenų bazės „Kubernetes“ aplinkoje, kuri pasirinkta dėl plataus naudojimo rinkoje. Tiriamoji architektūra apima kelių pagrindinių duomenų bazių klasterį su trimis mazgais „Kubernetes“ *StatefulSet*, rinkinyje, pasiekiamą per apkrovos balansavimo įrenginį. Naudojami dviejų tipų apkrovos balansavimo įrenginiai: 7 OSI lygio „ProxySQL“ ir 4 OSI lygio „HAProxy“. „Kubernetes“ klasteris paleistas „Google Cloud Platform“ (GCP), kur kiekvienas mazgas turi identiškas specifikacijas. Kiekvienas mazgas veikė ant „E2 cost-optimized“ virtualios mašinos su 4 procesoriais ir 6 GB operatyviosios atminties. „MySQL Galera“ klasteris buvo pasirinktas dėl jo kelių pagrindinių mazgų replikacijos palaikymo. Eksperimento aplinka buvo įdiegta naudojant „Kubernetes“ operatorių, skirtą „MySQL Galera“ klasteriui.

Prieinamumas buvo įvertintas naudojant zondavimo užklausas, imituojančias SQL užklausas su skirtinga trukme: 0,5, 1 ir 2 sekundės; ir lygiagretumo rodikliu: 10, 25, 50, 100 ir 200 užklausų vienu metu. „Python“ skriptas, naudojantis „PyMySQL“ biblioteką, buvo naudojamas prisijungti prie duomenų bazės klasterio tiek pakartotinai naudojamoms, tiek naujai inicijuojamoms sesijoms. Prieinamumas vertintas pagal nepavykusių užklausų procentą kiekvienam parametrų rinkiniui.

Eksperimento rezultatai parodė, kad „ProxySQL“, veikiantis 7 OSI lygyje, pademonstravo didesnį prieinamumą „MySQL Galera“ klasterio mazgų perjungimo metu, palyginti su „HAProxy“. Ryšio su duomenų baze tipas taip pat turėjo įtakos prieinamumui, iš naujo atidarytiems ryšiams reikėjo pakartotinio autentifikavimo ir autorizacijos, todėl

trumpų užklausų gedimų dažnis buvo didesnis. Užklausos trukmė taip pat turėjo įtakos prieinamumui – ilgesnės užklausos turėjo didesnį gedimų dažnį. Apkrova, kurią reprezentavo lygiagrečios užklausos, turėjo didesnį poveikį, kai apkrovos balansavimas buvo vykdomas naudojant „ProxySQL“. Nepavykusių užklausų procentas skirtinguose eksperimento aspektuose labai skyrėsi – nuo 32 % iki 182 %.

Apkrovos balansavimo įrenginio tipas turi didžiausią įtaką – vidurkiai svyruoja nuo 82 % iki 119 %. Lygiagrečių užklausų skaičius, ryšio tipas ir užklausos trukmė turi mažesnę įtaką – iki 17 %. Prisijungimo tipas kombinacijoje su apkrovos balansavimo įrenginio turi didžiausią įtaką pasiekiamumo rodikliui – nuo 77 % iki 126 %.

Apkrovos balansavimo įrenginio tipas turi įtakos ne tik našumui ir pralaidumui, bet ir prieinamumui. Nors 4 OSI lygio apkrovos balansavimo įrenginys gali užtikrinti didesnį našumą tam tikroms apkrovoms, 7 OSI lygio apkrovos balansavimo įrenginys padeda pasiekti didesnį pasiekiamumo laipsnį priežiūros operacijų metu.

3. Nepriklausomas žemos delsos būsenas išlaikančių mikroservisų persijungimas

Atliekami priežiūros darbai būsenas išlaikančiuose mikroservisuose turi įtakos tiekiamos paslaugos prieinamumui ir pasiekiamumui. Siekiant išvengti įtakos serviso prieinamumui, galima laikinai sustabdyti užklausas į būsenas išlaikantį mikroservisą, tačiau tam reikalingas klientinių mikroservisų konfigūravimas. Tai padidina sąsają tarp mikroservisų.

Siūlomu metodu siekiama pagerinti būsenas išlaikančių mikroservisų prieinamumą planuotų darbų metu, pavyzdžiui, diegiant atjauninimus arba keičiant konfigūraciją. Metodas grindžiamas priverstiniu nenaudojamų sutelktų klientų sesijų uždarymu duomenų bazės mazge ir apkrovos balansavimu. Taip įmanoma nukreipti duomenų bazės užklausas, nedarant reikšmingos įtakos klientui.

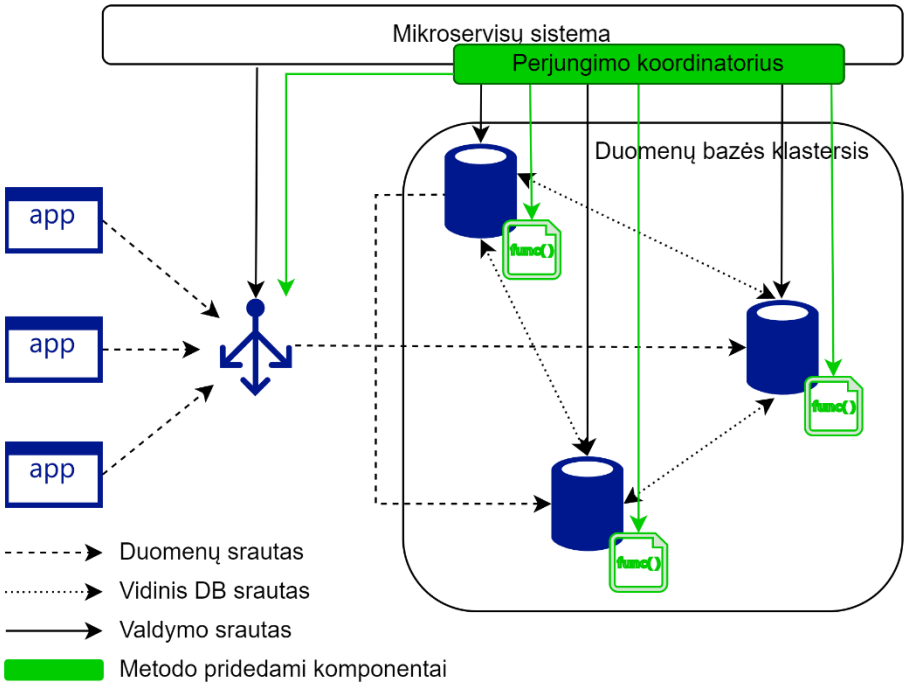
Metodui veikti reikalinga kelių pagrindinių mazgų duomenų bazės valdymo sistema, mazgų perjungimo koordinatorius, pavyzdžiui „Kubernetes“ operatorius, užklausų balansavimo įrenginys, mechanizmas klientų ryšiams nutraukti, ir pakartotinio bandymo mechanizmas kliento pusėje. Komponentai yra pavaizduoti S3.1 pav.

Inicijavus mazgų perjungimą, koordinatorius apkrovos balansavimo įrenginiui nurodo į kuriuos mazgus nukreipti klientų prisijungimus. Ant išjungiamo mazgo nenaudojami (angl. *Sleeping*) prisijungimai yra nutraukiami. Kliento pusėje nenaudojamas sutelktas prisijungimas prie duomenų bazės uždaromas ir iš naujo inicijuojamas kitame duomenų bazės mazge pagal apkrovos balansavimo įrenginio nukreipimą. Kliento prisijungimas buvo nenaudojamas, tad jokia veiklai reikšminga transakcija nėra nutraukiama.

Metodui validuoti atliktas eksperimentas trijų mazgų „Kubernetes“ klasteryje, naudojant „MySQL Galera“ reliacinę duomenų bazių valdymo sistemą ir „ProxySQL“ kaip apkrovos balansavimo sprendimą. Eksperimento metu vertinti įvairūs veiksniai: skirtinga duomenų bazės apkrova, pakartotinio bandymo vėlavimas. Siūlomas metodas buvo lyginamas su priverstiniu mazgų perjungimu. Generuota sintetinė apkrova. Nepavykusių užklausų ir lygiagrečiai paleistų santykis, išreikštas procentais, naudojamas pasiekiamumui vertinti. Eksperimento parametrai:

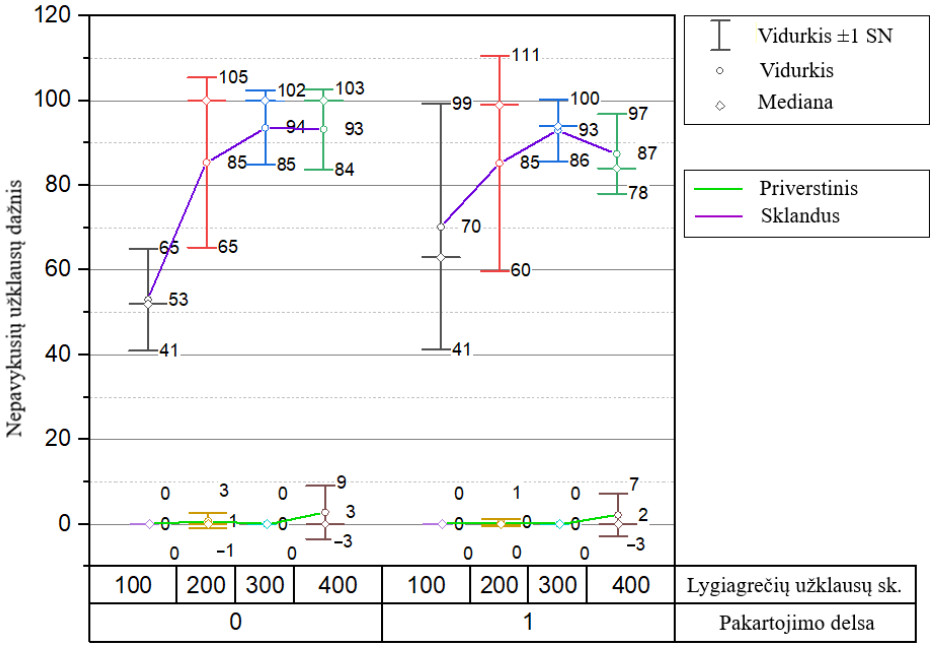
– lygiagrečių užklausų sesijų skaičius: 100, 200, 300, 400;

– perjungimo tipas: sklandus arba priverstinis;
– išimčių (klaidų) valdymo tipas: pakartojimas be delsimo, pakartojimas po 1 sekundės.

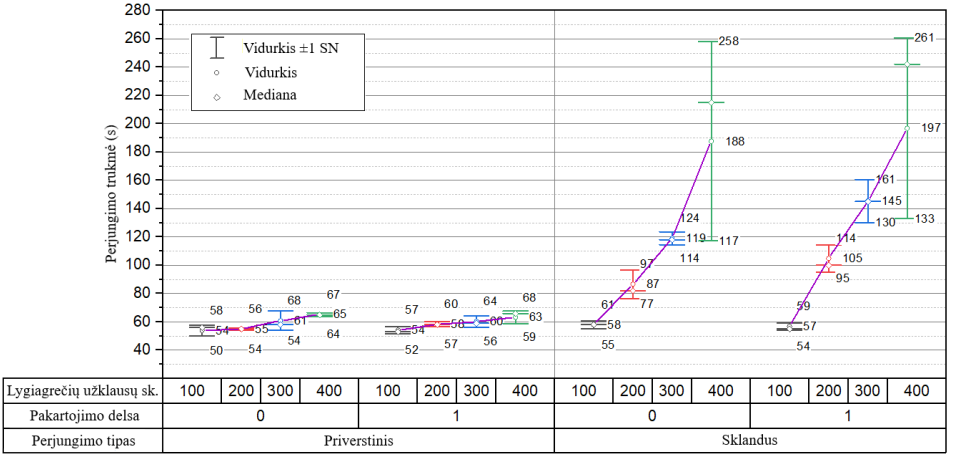


S3.1 pav. Siūlomo metodo komponentai

Eksperimento rezultatai parodė, kad siūlomas metodas reikšmingai sumažina nepavykusių užklausų procentą, palyginant su priverstiniu mazgų perjungimu. Vidutinis nepavykusių užklausų procentas sklandaus perjungimo metu buvo artimas nuliui – net ir prie didžiausios apkrovos vidurkis siekė 3 su standartiniu nuokrypiu iki 9. Naudojant priverstinį perjungimą tarp mazgų, dažnio vidurkis buvo tarp 53 ir 94 (S3.2 pav.).



S3.2 pav. Nepavykusių užklausų priklausomybė nuo perjungimo tipo, lygiagrečių užklausų skaičiaus ir pakartojimo delsos



S3.3 pav. Nepavykusių užklausų priklausomybė nuo perjungimo tipo, lygiagrečių užklausų skaičiaus ir pakartojimo delsos

Tačiau sklandus apkrovos perjungimas tarp mazgų užtrunka ilgiau nei priverstinis. Priverstinio perjungimo trukmės vidurkis varijuoja nuo 54 iki 63 sekundžių. Sklandus perjungimas gali trukti tris kartus ilgiau, o užfiksuotas vidurkis – nuo 58 iki 197 sekundžių (S3.2 pav.). Kuo daugiau klientų aktyviai naudoja būseną išlaikantį mikroservisą, tuo ilgiau trunka sklandus perjungimas.

Siūlomas metodas turi keletą privalumų, o vienas svarbiausių yra poreikio klientinių mikroservisų perkonfigūravimo nebuvimas, užtikrinant žemą sąsają tarp mikroservisų ir darant nereikšmingą įtaką prieinamumui. Tačiau, metodas turi ir trūkumų, tokių kaip specifinės architektūros poreikis ir pailgėjęs persijungimo šalinimo laikas.

4. Būsenas išlaikančių mikroservisų atsparumo didinimas staigiai ir reikšmingai padidėjusiai apkrovai

Staigus ir reikšmingas būseną išlaikančio mikroserviso apkrovos padidėjimas, pavyzdžiui, neprognozuotai išaugęs užklausų skaičius, kelia pasiekiamumo ir prieinamumo sumažėjimo riziką. Jei tokia apkrova nėra tinkamai valdoma, mikroservisas gali tapti visiškai nepasiekiamas dėl išnaudotų jam skirtų išteklių, ypač operatyviosios atminties.

Siūlomu metodu siekiama pagerinti būsenas išlaikančių mikroservisų atsparumą staigiam ir reikšmingam apkrovos augimui, ją paskirstant tarp mazgų. Taip siekiama prailginti sistemos veikimo laiką padidėjusios apkrovos sąlygomis. Metodas, pagrįstas apdorojamų užklausų ir išnaudojamų skaičiavimo resursų stebėjimu bei apkrovos balansavimu.

Metodui veikti yra reikalinga bent 3 pagrindinių mazgų duomenų bazės valdymo sistema, mazgų perjungimo ir resursų stebėjimo koordinatorius, pavyzdžiui „Kubernetes“ operatorius, užklausų balansavimo įrenginys. Išeikvojus operatyviąją atmintį, konteineris yra priverstinai perkraunamas ar sustabdomas, tad apkrovos dydžiui stebėti buvo pasirinkta ši metrika. Koordinatorius stebi operatyvios atminties sunaudojimą būsenas išlaikančio mikroserviso mazge. Sunaudojimui pasiekus tam tikrą ribą ant vieno iš trijų ar daugiau mazgų, kitas iš mazgų yra paruošiamas vertikaliam masteliavimui. Kol šis mazgas persikrauna, visa apkrova yra balansuojama tarp likusių dviejų ar daugiau mazgų. Kai masteliavimui pasirinktas mazgas pakyla, visa apkrova nukreipiama į jį ir likę mazgai masteliuojami. Naujai pakilęs mazgas turi turėti pakankamai resursų apdoroti padidėjusią apkrovą.

Metodui validuoti atliktas eksperimentas trijų mazgų „Kubernetes“ klasteryje, naudojant „MySQL Galera“ reliacinių duomenų bazių valdymo sistemą ir „ProxySQL“ kaip apkrovos balansavimo sprendimą. Eksperimento metu, naudojant „Yahoo! Cloud Serving Benchmark“ (YCSB), sugeneruota staigiai ir reikšmingai padidėjusi apkrova. Apkrovos generatorius turėjo sugeneruoti 75 000 užklausų, vienu atveju apkrovos pikas buvo pasiekiamas po 7,5 sekundės, antru – po 15 sekundžių. Siūlomas apkrovos balansavimo metodas buvo lyginamas su standartiniu „MySQL Galera“ klasterio funkcionalumu, kai apkrova nėra paskirstoma tarp mazgų, o užklausos nukreipiamos be balansavimo. Lentelėje S4.1 parodytas parametrų rinkinys. Vertinimui buvo matuojamas nepavykusių užklausų procentas ir laikas iki duomenų bazės klasteris tapdavo nepasiekiamas.

S4.1 lentelė. Eksperimento parametrų rinkinys

Apkrovos užklausų paskirstymas, %			**Apkrovos valdymo būdas**	**Laikas iki apkrovos piko (s)**
Skaitymas	**Naujinimas**	**Įrašymas**		
83	15	2	Jokio veiksmo	7,5
				15
			Nukreipimas	7,5
				15
			Balansavimas	7,5
				15

Eksperimento rezultatai parodė, kad sesijų balansavimas žymiai sumažina gedimų procentą, palyginti su standartiniu funkcionalumu, kai jokių veiksmų nesiimama, arba kliento seansų nukreipimu. Kaip parodyta S4.2 lentelėje, gedimų dažnis sumažėjo iki 81,82 %. Tikėtina, kad sumažėjusi duomenų bazės mazgo apkrova lėmė mažesnį nepavykusių užklausų skaičių, nes ryšys buvo sklandžiai perduodamas kitiems klasterio mazgams. Šie rezultatai taip pat patvirtina ankstesnio skyriaus rezultatus.

Laikas, reikalingas bet kuriam mazgui pasiekti eksperimente nustatytą operatyviosios atminties limitą, pagerėjo naudojant siūlomą metodą. Užklausų apdorojimo laikas pailgėjo iki 42,4 ir 40,8 sekundės, palyginti su 23 ir 25 sekundėmis, kai nebuvo imtasi jokių veiksmų.

Siūlomas metodas leido prailginti būsenas išlaikančio mikroserviso veikimo laiką sumažinant nepavykusių užklausų skaičių. Metodas gali būti naudojamas kaip papildomas apsaugos mechanizmas kartu su mašininių mokymu grįstais apkrovos prognozavimo algoritmais.

S4.2 lentelė. Nepavykusių užklausų vertinimo santrauka

Laikas iki apkrovos piko (s)	**Apkrova**	**Apkrovos valdymo būdas**			**Nepavykusių užklausų procento sumažėjimo balansuojant apkrovą palyginimas (%)**	
		Jokio veiksmo	**Nukreipimas**	**Balansavimas**	**Jokio veiksmo**	**Nukreipimas**
7,5	Skaitymas	0,61	0,45	0,05	91,80	88,89
	Naujinimas	2,39	0,17	0,02	99,16	88,24
	Įrašymas	0,96	0,33	0,06	93,75	81,82
	Bendrai	0,78	0,35	0,04	94,87	88,57
15	Skaitymas	0,38	0,33	0	100	100
	Naujinimas	1,05	2,36	0	100	100
	Įrašymas	2,77	1,34	0	100	100
	Bendrai	1,15	1,09	0	100	100

S4.3 lentelė. Laiko apdorojant apkrovą santrauka

Laikas iki apkrovos piko (s)	Apkrovos valdymo būdas			Laiko iki pasiekiamumo padidėjimas naudojant balansavimą palyginimas (%)	
	Balansavimas	Balansavimas	Balansavimas	Jokio veiksmo	Nukreipimas
7,5	23	42,2	40,8	177,39	96,68
15	25,25	27,6	42,4	167,92	153,62

Bendrosios išvados

Šis tyrimas sprendžia esminį iššūkį – užtikrinti aukštą būseną išlaikančių mikroservisų prieinamumą ir patikimumą atliekant gedimų šalinimo ir priežiūros operacijas. Tyrimo tikslas – pagerinti būsenas išlaikančių mikroservisų prieinamumą koordinuotai valdomose konteinerių sistemose, nustatant pagrindinius jam įtaką darančius veiksnius, pasiūlant mažos delsos metodą prieinamumui užtikrinti priežiūros metu ir pristatant taisyklėmis grindžiamą metodą, skirtą atsparumui didinti staigaus ir reikšmingo apkrovos padidėjimo metu. Atliktą tyrimą galima apibendrinti taip:

1. Literatūros apžvalga pabrėžia, kad būseną turinčių mikroservisų priežiūra koordinuotai valdomose konteinerių sistemose yra ypač svarbus ir iš esmės sudėtingas uždavinys. Su būsenos valdymu susiję iššūkiai apima duomenų nuoseklumo užtikrinimą, atsarginių kopijų prieinamumo palaikymą ir našumo užtikrinimą. SLI ir SLO rodikliai, tokie kaip procesoriaus ir atminties panaudojimas, yra pagrindiniai veiksniai, lemiantys sprendimų priėmimą, siekiant užtikrinti sistemos patikimumą ir prieinamumą. Duomenimis pagrįsti metodai naudojami gedimų prognozavimui, optimaliam konteinerių planavimui ir darbo krūvio prognozavimui, siekiant pagerinti būseną turinčių mikroservisų prieinamumą ir patikimumą. Taisyklėmis pagrįsti metodai, tokie kaip pritaikyti planavimo įrankiai ir būsenos valdikliai, naudojami sistemų patikimumo didinimui koordinuotai valdomose konteinerių sistemose.
2. Įvertinti įvairūs veiksniai, darantys įtaką paslaugos prieinamumui, įskaitant apkrovos intensyvumą, apkrovos balansavimo įrenginio tipą ir ryšio tipą. Parodyta, kad SQL palaikantys apkrovos balansavimo įrenginiai, tokie kaip „ProxySQL“, užtikrina didesnį prieinamumą, palyginti su TCP apkrovos balansavimo įrenginiais, pavyzdžiui, „HAProxy“ – vidutinis prieinamumas svyruoja nuo 82 % iki 119 %. Tai siejama su SQL palaikančių apkrovos balansavimo įrenginio gebėjimu atpažinti perjungimo į kitą serverį įvykius ir perduoti užklausas kitam duomenų bazės mazgui, taip sumažinant nepavykusių užklausų skaičių. Be to, ryšio tipas (sutelktas arba iš naujo atidaromos) taip pat turi įtakos nepavykusių užklausų skaičiui: iš naujo atidaromos užklausos rodo mažesnį gedimų skaičių, kai naudojamas SQL palaikantis apkrovos balansavimo įrenginys. Svarbu pažymėti, kad prieinamumo rodiklis labai priklauso nuo veiksnių kombinacijos.
3. Pasiūlytas metodas, skirtas užtikrinti aukštą būseną išlaikančių mikroservisų prieinamumą atliekant valdomas perjungimo operacijas. Šio metodo tikslas – pasiekti

skaidrų ir sklandų būseną išlaikančių mikroservisų perjungimą, pasitelkiant konteinerių koordinuoto valdymo sistemos, pavyzdžiui, „Kubernetes“, ir apkrovos balansavimo įrenginių funkcionalumą, kad žemos delsos klientų prisijungimai būtų perkelti iš vieno duomenų bazės mazgo į kitą su minimaliu poveikiu klientų programoms. Rezultatai rodo, kad SQL palaikantis apkrovos balansavimo įrenginys „ProxySQL“ reikšmingai padidina prieinamumą priežiūros operacijų metu, sumažindamas prieinamumo praradimą per perjungimą iki nežymių reikšmių. Standartinis funkcionalumas leidžia pasiekti nepavykusių užklausų dažnį nuo 53 iki 94, siūlomas metodas jį sumažino iki nereikšmingų verčių, siekiančių 3. Nors perjungimo trukmė padidėja, sumažinus gedimų dažnį priežiūros operacijos gali turėti nereikšmingą poveikį būseną išlaikantiems mikroservisams, išlaikant mažą sąsają su kliento mikroservisais.

4. Pristatytas naujas, taisyklėmis pagrįstas metodas, skirtas pagerinti būseną išlaikančių mikroservisų atsparumą staigiai ir reikšmingai padidėjusiai apkrovai. Siūlomas metodas naudoja iš anksto nustatytas rašymo užklausų mastelio keitimo taisykles, kad dinamiškai pritaikytų sistemos elgseną reaguojant į padidėjusią apkrovą, taip padidindamas atsparumą staigiai ir reikšmingai padidėjusiai apkrovai. Staigiai ir reikšmingai padidėjusios apkrovos intensyvumas daro įtaką prieinamumui; esant mažesniam intensyvumui, apkrovą galima subalansuoti, o rašymo užklausas masteliuoti taip, kad būtų apdorotas padidėjęs poreikis. Tačiau esant pakankamai didelio intensyvumo staigiam ir reikšmingam apkrovos padidėjimui, būseną išlaikantis mikroservisas gali nustoti veikti nepriklausomai nuo to, ar taikomas siūlomas metodas. Nepaisant to, siūlomas metodas sumažina gedimų dažnį ir leidžia paslaugai veikti esant apkrovos šuoliui beveik dvigubai ilgiau nei naudojant standartinį funkcionalumą. Laikas, per kurį paslauga veikia esant apkrovos šuoliui, padidėja nuo 23 iki 41 sekundės. Be to, siūlomas metodas gali tarnauti kaip apsauga, kai darbo krūvio prognozavimo algoritmai nesugeba tiksliai numatyti apkrovos.

Kęstutis PAKRIJAUSKAS

INCREASING AVAILABILITY OF STATEFUL MICROSERVICES IN ORCHESTRATED CONTAINER SYSTEMS

Doctoral Dissertation

Technological Sciences,
Informatics Engineering (T 007)

BŪSENAS IŠLAIKANČIŲ MIKROPASLAUGŲ PATIKIMUMO DIDINIMAS KOORDINUOTAI VALDOMOSE KONTEINERIŲ SISTEMOSE

Daktaro disertacija

Technologijos mokslai,
Informatikos inžinerija (T 007)

Lietuvių kalbos redaktorė Deimantė Grigaitė

Anglų kalbos redaktorė Jūratė Griškėnaitė

2026 04 20. 10,3 sp. l. Tiražas 20 egz.
Leidinio el. versija *https://doi.org/10.20334/2026-020-M*
Vilniaus Gedimino technikos universitetas
Saulėtekio al. 11, 10223 Vilnius
Spausdino UAB „Ciklonas“,
Žirmūnų g. 68, 09124 Vilnius