Scalability Benchmarking of Cloud-Native Applications Applied to Event-Driven Microservices

Sören Henning

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2023

Kiel Computer Science Series (KCSS) 2023/2 dated 2023-03-21

ISSN 2193-6781 (print version)
ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via `https://www.informatik.uni-kiel.de/kcss`

The author can be contacted via `post@soeren-henning.de`

Published by the Department of Computer Science, Kiel University

Software Engineering Group

Please cite as:

Sören Henning. *Scalability Benchmarking of Cloud-Native Applications Applied to Event-Driven Microservices*. Number 2023/2 in Kiel Computer Science Series. Department of Computer Science, 2023. Dissertation, Faculty of Engineering, Kiel University.

```
@book{Henning23,
  author    = {S{\"o}ren Henning},
  title     = {Scalability Benchmarking of Cloud-Native Applications
               Applied to Event-Driven Microservices},
  publisher = {Department of Computer Science,
               Faculty of Engineering, Kiel University},
  year      = {2023},
  number    = {2023/2},
  doi       = {10.21941/kcss/2023/2}
  series    = {Kiel Computer Science Series},
  note      = {Dissertation, Faculty of Engineering,
               Kiel University.}
}
```

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at Kiel University. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit `http://www.informatik.uni-kiel.de/kcss` for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Gutachter:	Prof. Dr. Wilhelm Hasselbring Christian-Albrechts-Universität zu Kiel
2. Gutachter:	Univ.-Prof. Mag. Dr. Rick Rabiser Johannes Kepler Universität Linz
3. Gutachter:	Prof. Dr.-Ing. David Bermbach Technische Universität Berlin

Datum der mündlichen Prüfung: 14. März 2023

Zusammenfassung

Cloud-native Anwendungen stellen einen aktuellen Trend für den Entwurf großer Software-Systeme dar. Trotz einer Vielzahl entsprechender Tools und Muster, gibt es allerdings keine allgemein akzeptierte Methode, ihre Skalierbarkeit empirisch zu benchmarken.

Diese Dissertation stellt unsere Benchmarkingmethode Theodolite vor. Sie erlaubt es, empirische Skalierbarkeitsevaluationen von Cloud-nativen Anwendungen, Frameworks und Betriebsvarianten durchzuführen. Unsere Benchmarkingmethode besteht aus Metriken, Messmethoden und einer Architektur für ein entsprechendes Benchmarkingtool. Sie quantifiziert Skalierbarkeit mittels isolierter Experimente, die für verschiedene Last- und Ressourcenkombinationen bewerten, ob spezifizierte Service Level Objectives (SLOs) erfüllt werden. Um Benutzbarkeit und Reproduzierbarkeit in Einklang zu bringen, kann der Kompromiss zwischen Ausführungszeit und statistisch belastbaren Ergebnissen individuell konfiguriert werden.

Wir wenden unsere Benchmarkingmethode auf ereignisgesteuerte Microservices an, einem speziellen Typ von Cloud-nativen Anwendungen, die verteilte Frameworks zur Datenstromverarbeitung einsetzen. Angelehnt an reelle Anwendungsfällen zur Analyse industrieller Stromverbrauchsdaten, entwerfen wir Skalierbarkeits-Benchmarks. Diese umfassen Datenfluss-Architekturen zur Datenaggregation, konfigurierbare Last- und Ressourcendimensionen sowie SLOs.

In umfangreichen experimentellen Evaluationen benchmarken und vergleichen wir die Skalierbarkeit verschiedener Frameworks zur Datenstromverarbeitung in unterschiedlichen Konfigurationen. Unsere Evaluationen zeigen, dass statistisch belastbare Ergebnisse in angemessener Zeit erzielt werden können. Wir führen unsere Experimente in verschiedenen Public- und Private-Cloud-Umgebungen durch und vergleichen unsere Ergebnisse mit denen von Function-as-a-Service. Des Weiteren setzen wir Theodolite in drei Fallstudien ein und stellen fest, dass es auch die Skalierbarkeit verschiedenster anderer Anwendungen benchmarken kann.

Abstract

Cloud-native applications constitute a recent trend for designing large-scale software systems. However, even though several cloud-native tools and patterns have emerged to support scalability, there is no commonly accepted method to empirically benchmark their scalability.

This thesis introduces our Theodolite benchmarking method, allowing researchers and practitioners to conduct empirical scalability evaluations of cloud-native applications, frameworks, and deployment options. Our benchmarking method consists of scalability metrics, measurement methods, and an architecture for a scalability benchmarking tool. Following fundamental scalability definitions and established benchmarking best practices, we propose to quantify scalability by performing isolated experiments for different load and resource combinations, which assess whether specified service level objectives (SLOs) are achieved. To balance usability and reproducibility, our method provides configuration options, controlling the trade-off between execution time and statistical grounding.

We apply our benchmarking method to event-driven microservices, a specific type of cloud-native applications that employ distributed stream processing frameworks to scale with massive data volumes. Based on studying use cases for analyzing industrial power consumption data, we propose a set of scalability benchmarks for such microservices. They comprise dataflow architectures for different types of sensor data aggregation, configurable load and resource dimensions, as well as SLOs.

In extensive experimental evaluations, we benchmark and compare the scalability of various stream processing frameworks under different configurations and deployments. Our evaluations show that statistically grounded results can be obtained in reasonable time. We run these experiments in different public and private cloud environments and compare their costs with those of Function-as-a-Service deployments. Furthermore, we employ Theodolite in three case studies and find that it can be applied to a wide range of applications beyond stream processing.

Preface by
by Prof. Dr. Wilhelm Hasselbring

Scalability is a primary motivation for building cloud-native applications. In empirical software engineering, benchmarks can be used for comparing different methods, techniques and tools. A challenge, addressed by Sören Henning with this thesis, is to empirically benchmark the scalability of cloud-native applications.

In this thesis, Sören Henning, implements and evaluates the new, innovative Theodolite benchmarking method, allowing researchers and practitioners to conduct empirical scalability evaluations of cloud-native applications, frameworks, and deployment options. The Theodolite benchmarking method consists of scalability metrics, measurement methods, and an architecture for the scalability benchmarking tool.

Besides the conceptual work, this thesis contains a significant experimental part with an implementation and a multifaceted evaluation. Thus, this engineering dissertation has been extensively evaluated with cloud benchmarking experiments, including data from industrial systems.

Besides the publication of research papers with associated replication packages, the Theodolite open-source software implementation has been reviewed and accepted for the SPEC Research Group's repository of peer-reviewed tools for quantitative system evaluation and analysis. Similar to the peer-reviewing process for scientific publications, submitted software tools are thoroughly evaluated by at least three reviewers of the SPEC Research Group.

If you are interested in benchmarking cloud-native applications, this is a recommended reading for you.

Wilhelm Hasselbring
Kiel, March 2023

Acknowledgments

This work would not have been possible without the support and advice of many people. First of all, I would like to thank Willi Hasselbring for supervising my doctoral studies and guiding my way into the world of software engineering research. He gave me the freedom to explore my research interests but also encouraged me to look beyond them, for example, on topics such as benchmarking. Always a source of inspiration and motivation, he especially advised me on how to balance practical and scientific relevance and on how to effectively present research results. Moreover, I would like to thank the second and third reviewers of this thesis, Rick Rabiser and David Bermbach, as well as the remaining members of my examining committee, Peer Kröger and Olaf Landsiedel, for their efforts.

Throughout the entire time working on this research, I was employed at the Software Engineering Group at Kiel University. My sincere thanks go to my colleagues Alexander Barbie, Sven Gundlach, Malte Hansen, Reiner Jung, Alexander Krause-Glau, Christine Krüger, Arnd Plumhoff, Henning Schnoor, Nelson Tavares de Sousa, Matthias Westphal, Christian Wulf, and Christian Zirkelbach for the productive working atmosphere, fruitful discussions, memorable conference travels, and great team experiences.

Moreover, I would like to thank the many students who assisted in designing and implementing this research. Special mention should be made of Lorenz Boguhn, Simon Ehrenstein, Björn Vonheiden, and Benedikt Wetzel, whose large contributions become evident by only looking at the Git history of our Theodolite tool and at this thesis' evaluation chapters.

Large parts of this research have been conducted in collaboration with industry partners. I would especially like to thank the people at b+m Informatik AG, IBAK Helmut Hunger GmbH & Co. KG, Kieler Zeitung GmbH & Co. Offsetdruck KG, and wobe-systems GmbH for giving me the opportunity to work on "real-world problems". I also wish to acknowledge Tobias Pfandzelter and Trever Schirmer for the academic collaboration that led to Chapter 15 of this thesis.

I would like to thank my friends and family who supported me in various ways throughout the years of my doctoral studies. I am deeply grateful to my parents Gabi and Claus Henning for allowing me to receive the education resulting in this work and for always having a sympathetic ear for me. There is also a special thank you to Jette Henning for proof-reading large parts of this work and her patience in pointing out the same mistakes over and over again. Finally, I would like to express my deepest gratitude to Kea Ille for her unwavering love and support, for encouraging me to not always take the path of least resistance, but also for reminding me of the truly important things in life.

Contents

Chapter 1

Introduction

Scalability is a driving requirement for many software systems, especially for those that are designed as cloud-native applications and event-driven microservices. Empirically evaluating and comparing the scalability of such applications is therefore of great interest for software engineers, architects, and researchers. This thesis presents and evaluates a scalability benchmarking method for cloud-native applications along with specific scalability benchmarks for event-driven microservices.

We start this thesis with this introductory chapter, by first motivating and setting its context (Section 1.1). We continue with stating problems found in the state-of-the-art (Section 1.2) and define the guiding goals of the thesis along with research questions to be addressed (Section 1.3). Afterward, we summarize the contributions of this thesis (Section 1.4). Lastly, we give an overview of preliminary work included in this thesis (Section 1.5) and outline the document structure (Section 1.6).

1.1 Motivation and Context

Over the last two decades, software engineering was tremendously driven by scale [Gor22]. This includes drastically increased range of functions, user bases, and data volumes of software systems. While in 2011, John Ousterhout [Ous11] stated that "scale has been the single most important force driving changes in system software over the last decade", an end to this growth does not yet appear to be in sight. The capability of a software system to handle such growth is referred to as scalability [HKR13]. In the shade of the often reported hyper-scalable, Internet-facing systems [Gor17], scalability has also become a driving requirement for many software systems in other domains such as the Industrial Internet of Things (IIoT) [SSH+18].

Since more than a decade now, cloud computing is both a solution and a driver for building large-scale software systems [AFG+10]. Under the impression of seemingly unlimited computing resources, cloud users can expand the capacity of their software on demand and with pay-per-use billing to handle ever-increasing amounts of users and data. On the other hand, being able to store and analyze giant volumes of data allows for formerly unimaginable services, leading to even larger systems.

Following the rise of cloud computing as preferred deployment infrastructure for many applications, we are now witnessing how large-scale software systems are increasingly being designed as cloud-native applications [GBS17]. Under the umbrella term "cloud-native", a wide range of tools and patterns emerged for simplifying, accelerating, and securing the development and operation of software systems in the cloud. Key concepts are containers, service meshes, microservices, immutable infrastructure, and declarative APIs to build resilient, manageable, observable, and, in particular, scalable software systems [Clo18]. Cloud-native software systems and practices to guarantee their scalability are recognized as one of the trending topics in software engineering research in 2022 [FBS22].

Microservices are an architectural pattern, particularly suited for building scalable, cloud-native software systems [Has16]. Supported by functional decomposition, polyglot persistence, polyglot programming, and often eventual consistency, loosely coupled microservices can be deployed and scaled independently [Zim17; Has18]. Scalability is in fact often reported as the main driver for adopting microservice and cloud-native architectures in industry [KQ17; STV18; KH19].

More recently, a shift toward combining event-driven architectures with microservices can be observed [LZS+21]. In such systems, individual microservices communicate via asynchronous messages and events, which are managed by messaging or queuing systems. Especially data-intensive applications and big data analytics systems are increasingly designed as event-driven microservices, leading to several frameworks for processing continuous data streams in a scalable manner [DL20].

However, scalability does not come "out of the box" just by building software systems as cloud-native and with an event-driven microservice architecture. Scalability might be affected by several decisions regarding a software's architecture, employed technologies, and services as well as various configuration and deployment options [Gor22]. Especially choosing among the wide range of cloud deployment options [GKK+12; FFH13] and configuration options of rapidly evolving big data stream processing systems [HCL20] is challenging. Benchmarking is a well-established method in software engineering to assess and compare the quality of software systems and services [Has21]. As such, it is used both in research and engineering [KLvK20] to choose among competing software solutions, to evaluate the quality of new ones, or to assure quality levels over time. Whether by manual benchmark analysis or automated configuration tuning, benchmark results are often used to optimize various aspects of a software system. Hence, benchmarking can and should be used to evaluate and compare the scalability of cloud-native applications. As detailed in the following section, however, a corresponding scalability benchmarking method is yet to be established.

1.2 Problem Statement

Scalability is widely considered an essential quality attribute of cloud-native applications and, in particular, of event-driven microservices [KQ17; STV18; KH19; LZS+21]. Nevertheless, research is lacking a commonly accepted benchmarking method to empirically assess their scalability. In the following, we break this down to discuss the lack of a suitable scalability benchmarking method and the lack of suitable benchmarks for distributed stream processing frameworks used in event-driven microservices.

1.2.1 Lack of a Commonly Accepted Scalability Benchmarking Method

Although precise definitions of scalability exist and were refined over the last two decades [JW00; DRW07; WHG+14], we observe that no commonly accepted method exists for benchmarking scalability of cloud-native applications. In particular, we notice the following shortcomings:

- We found that several studies conducting scalability evaluations do not describe their employed methodology with sufficient detail. A possible reason is that such evaluations are often conducted as part of larger (often engineering) works and, thus, are not considered to be the main contributions of such works.
- Scalability benchmarking studies build upon different definitions of scalability, which are not generally transferable to scalability of cloud-native applications.
- Several employed methods to empirically evaluate scalability lack statistical soundness. Increasing statistical grounding would come at the expense of reduced usability and, hence, also reproducibility.
- To the best of our knowledge, there are neither tools nor corresponding architectural concepts available, aiming to increase usability of scalability benchmarks.

Even though most studies from academia and industry share similar understandings of scalability, the lack of well-defined metrics and measurement methods contradicts the fundamental principle of benchmarking.

1.2.2 Lack of Scalability Benchmarks for Event-Driven Microservices

Research on microservice architecture performance and benchmarking primarily targets synchronous, request–response communication between services. Although some studies also consider asynchronous communication via messaging queues, these studies focus on simple produce and consume tasks. They do not consider complex data processing inside services as enabled by modern stream processing frameworks.

On the other hand, benchmarks and performance evaluations for distributed stream processing frameworks are often designed as microbenchmarks instead of application benchmarks. Hence, they are designed according to capabilities of stream processing frameworks and not based on real use cases. Moreover, such benchmarks mainly focus on performance attributes such as processing latency and throughput and are not explicitly designed for evaluating scalability. We discuss existing stream processing benchmarks in detail in Section 11.6.

1.3 Guiding Goals and Research Questions

With this thesis, we address the two previously raised problems by defining two overarching goals. For each goal, we state research questions (RQ) to be addressed in this thesis.

1.3.1 A Method for Benchmarking the Scalability of Cloud-Native Applications

Our first goal is to engineer a scalability benchmarking method for cloud-native applications. It addresses the problem stated in Section 1.2.1. As we detail in Chapter 3, a benchmarking method usually defines metrics, corresponding measurement methods, and is often accompanied by a tool for running the benchmarks. We therefore state the following research questions:

RQ 1.1 How can scalability of cloud-native applications be quantified?

RQ 1.2 How can scalability be measured in a metric-conformant way, providing statistically grounded results in reasonable execution times?

RQ 1.3 How can a tool be designed that performs such a measurement method in a way that provides a high degree of usability and reproducibility?

1.3.2 A Benchmark to Assess and Compare the Scalability of Event-Driven Microservice Architectures

Our second goal is to engineer a benchmark that allows assessing and comparing the scalability of different stream processing frameworks, configuration options, and deployment options for event-driven microservices. It addresses the problem stated in Section 1.2.2. To approach this goal, we formulate the following research questions:

RQ 2.1 What are relevant use cases for event-driven microservices?

RQ 2.2 How do corresponding dataflow architectures look like that match state-of-the-art stream processing dataflow models?

RQ 2.3 Along which dimensions should load on and provisioned resources of event-driven microservices be considered?

RQ 2.4 How can the proper functioning of event-driven microservices be assessed?

1.4 Contributions and Evaluation Summary

With this thesis, we contribute to the research fields of empirical performance evaluation for cloud computing, microservice architectures, and distributed stream processing. In the following, we summarize our contributions, grouped into three categories.

1.4.1 The Theodolite Scalability Benchmarking Method

We present the Theodolite[1] benchmarking method, allowing researchers and practitioners to conduct empirical scalability evaluations of cloud-native applications, frameworks, and deployment options. As summarized in the following, our benchmarking method consists of scalability metrics, measurement methods, and an architecture for a scalability benchmarking tool, particularly suited for cloud-native applications. In particular, we make the following contributions:

– two scalability metrics, namely the resource demand and the load capacity metric, which quantify scalability based on the notions of load, resources, and service level objectives (SLOs),

[1]A theodolite is a precision optical instrument used in geodesy for measuring angles in the horizontal and vertical planes. Inspired by its namesake, our Theodolite method as well as our corresponding Theodolite benchmarking framework measures the horizontal and vertical scalability of cloud-native applications.

- a scalability measurement method, quantifying our metrics by running isolated experiments for automatically selected load and resource combinations, which assess whether specified SLOs are fulfilled, and
- an architecture for a scalability benchmarking framework as well as a corresponding implementation, which implement the proposed method as a Kubernetes operator, allowing for declarative descriptions of benchmarks and their executions.

1.4.2 The Theodolite Scalability Benchmarks for Event-Driven Microservices

Based on our benchmarking method, we present a set of benchmarks to empirically assess the scalability of frameworks, configurations, and deployment options for event-driven microservices. In particular, we make the following contributions:

- a set of four task samples, derived from real use cases for IIoT analytics, which serve as the basis for our scalability benchmarks,
- implementations of these task samples for several state-of-the-art stream processing frameworks, particularly suited to build event-driven microservices, as well as a scalable load generator,
- multiple load types and resource types, regarding which our benchmarks evaluate scalability, and
- two SLOs for stream processing frameworks, which can be efficiently measured in cloud-native environments.

1.4.3 Experimental Evaluations

We perform extensive experimental evaluations of and with our benchmarking method and provide experimental results for the scalability of event-driven microservices. In particular, we make the following contributions:

– evaluations of our method's configuration options in different cloud environments with multiple systems, suggesting that our method can provide statistically grounded results in reasonable execution times,

– scalability evaluations of the stream processing frameworks Apache Flink, Hazelcast Jet, Apache Kafka Streams, and Apache Beam (using the Apache Flink and the Apache Samza runner) with different configuration and deployment options,

– scalability evaluations of different techniques for aggregations on continuous data streams over sliding time windows,

– evaluations of cost scalability for different types of stream processing deployments and Function-as-a-Service (FaaS) deployments in the cloud, and

– conceptual evaluations of our benchmarking method for other types of cloud-native applications, namely a commercial software for the promotional loan business, an open-source research software for software visualization, and a microservice reference application, frequently used in research.

1.5 Preliminary Work

This thesis builds upon 17 peer-reviewed journal and conference publications with 6 associated replication packages, a non-reviewed project report, and 14 co-supervised student theses. We provide an overview of all these works below.

1.5.1 Peer-Reviewed Publications

1. S. Henning and W. Hasselbring. “Benchmarking scalability of cloud-native applications”. In: *Software Engineering 2023*. Bonn: Gesellschaft für Informatik e.V., 2023

2. S. Henning and W. Hasselbring. “A configurable method for benchmarking scalability of cloud-native applications”. In: *Empirical Software Engineering* 27.6 (2022). DOI: `10.1007/s10664-022-10162-1`

3. T. Pfandzelter, S. Henning, T. Schirmer, W. Hasselbring, and D. Bermbach. “Streaming vs. functions: a cost perspective on cloud event processing”. In: *2022 IEEE International Conference on Cloud Engineering (IC2E)*. 2022, pp. 67–78. DOI: 10.1109/IC2E55432.2022.00015

4. S. Henning and W. Hasselbring. “Demo paper: benchmarking scalability of cloud-native applications with Theodolite”. In: *2022 IEEE International Conference on Cloud Engineering (IC2E)*. 2022, pp. 275–276. DOI: 10.1109/IC2E55432.2022.00037

5. S. Henning, B. Wetzel, and W. Hasselbring. “Cloud-native scalability benchmarking with Theodolite: applied to the TeaStore benchmark”. In: *Softwaretechnik-Trends* 43.1 (Feb. 2023). (Proceedings of the 13th Symposium on Software Performance (SSP 2022)), pp. 23–25

6. S. Henning and W. Hasselbring. “How to measure scalability of distributed stream processing engines?” In: *Companion of the ACM/SPEC International Conference on Performance Engineering*. ICPE ’21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 85–88. DOI: 10.1145/3447545.3451190

7. S. Henning and W. Hasselbring. “Theodolite: scalability benchmarking of distributed stream processing engines in microservice architectures”. In: *Big Data Research* 25 (2021), p. 100209. DOI: 10.1016/j.bdr.2021.100209

8. S. Henning, W. Hasselbring, H. Burmester, A. Möbius, and M. Wojcieszak. “Goals and measures for analyzing power consumption data in manufacturing enterprises”. In: *Journal of Data, Information and Management* 3.1 (2021), pp. 65–82. DOI: 10.1007/s42488-021-00043-5

9. S. Henning and W. Hasselbring. “The Titan Control Center for Industrial DevOps analytics research”. In: *Software Impacts* 7 (2021). DOI: 10.1016/j.simpa.2020.100050

10. S. Henning, B. Wetzel, and W. Hasselbring. “Reproducible benchmarking of cloud-native applications with the Kubernetes Operator Pattern”. In: *Symposium on Software Performance 2021*. 2021. URL: http://ceur-ws.org/Vol-3043

11. S. Henning and W. Hasselbring. "Scalable and reliable multi-dimensional sensor data aggregation in data-streaming architectures". In: *Data-Enabled Discovery and Applications* 4.1 (2020). DOI: 10.1007/s41688-020-00041-3

12. S. Henning and W. Hasselbring. "Toward efficient scalability benchmarking of event-driven microservice architectures at large scale". In: *Softwaretechnik-Trends* 40.3 (Nov. 2020). (Proceedings of the 11th Symposium on Software Performance (SSP 2020)), pp. 28–30

13. S. Henning and W. Hasselbring. "Scalable and reliable multi-dimensional aggregation of sensor data streams". In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019, pp. 3512–3517. DOI: 10.1109/BigData47090.2019.9006452

14. S. Henning, W. Hasselbring, and A. Möbius. "A scalable architecture for power consumption monitoring in industrial production environments". In: *2019 IEEE International Conference on Fog Computing (ICFC)*. 2019, pp. 124–133. DOI: 10.1109/ICFC.2019.00024

15. W. Hasselbring, S. Henning, B. Latte, A. Möbius, T. Richter, S. Schalk, and M. Wojcieszak. "Industrial DevOps". In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 2019, pp. 123–126. DOI: 10.1109/ICSA-C.2019.00029

16. B. Latte, S. Henning, and M. Wojcieszak. "Clean code: on the use of practices and tools to produce maintainable code for long-living software". In: *Proceedings of the Workshops of the Software Engineering Conference 2019*. Vol. Vol-2308. Stuttgart, Germany: CEUR Workshop Proceedings, Feb. 2019, pp. 96–99. URL: http://ceur-ws.org/Vol-2308

17. S. Henning. "Monitoring electrical power consumption with Kieker". In: *Softwaretechnik-Trends* 39.3 (Nov. 2019). (Proceedings of the 9th Symposium on Software Performance (SSP 2018)), pp. 31–33

1.5.2 Replication Packages

1. S. Henning and W. Hasselbring. *Replication package for: benchmarking scalability of stream processing frameworks deployed as event-driven microservices in the cloud*. Zenodo, 2022. DOI: 10.5281/zenodo.7497281

2. T. Pfandzelter, S. Henning, T. Schirmer, W. Hasselbring, and D. Bermbach. *Replication package for: streaming vs. functions: a cost perspective on cloud event processing*. Zenodo, 2022. DOI: `10.5281/zenodo.7495024`

3. S. Henning and W. Hasselbring. *Replication package for: a configurable method for benchmarking scalability of cloud-native applications*. Zenodo, 2021. DOI: `10.5281/zenodo.5596982`

4. S. Henning and W. Hasselbring. *Replication package for: Theodolite: scalability benchmarking of distributed stream processing engines in microservice architectures*. Zenodo, 2021. DOI: `10.5281/zenodo.4476083`

5. S. Henning and W. Hasselbring. *Replication package for: scalable and reliable multi-dimensional sensor data aggregation in data-streaming architectures*. Zenodo, 2020. DOI: `10.5281/zenodo.3736689`

6. S. Henning and W. Hasselbring. *Replication package for: scalable and reliable multi-dimensional aggregation of sensor data streams*. Zenodo, 2019. DOI: `10.5281/zenodo.3540895`

1.5.3 Non-Reviewed Publications

1. W. Hasselbring, S. Henning, B. Latte, I. Stemmler, M. Wojcieszak, and U. Glockmann. *Abschlussbericht KMU-innovativ: Verbundprojekt Titan Industrial DevOps Plattform für iterative Prozessintegration und Automatisierung*. Tech. rep. Kiel: Selbstverlag des Instituts für Informatik, Kiel, 2021

1.5.4 Co-Supervised Bachelor's and Master's Theses

1. L. A. Mertens. "Reengineering Theodolite with the Java Operator SDK". Bachelor's Thesis. Kiel University, 2022

2. L. Boguhn. "Benchmarking the scalability of distributed stream processing engines in case of load peaks". Master's Thesis. Kiel University, 2022

3. D. B. Wetzel. "Scalability benchmarking of a promotional loan system". Master's Thesis. Kiel University, 2022

4. S. B. N. F. A. Ehrenstein. "Scalability evaluation of ExplorViz with the Universal Scalability Law". Master's Thesis. Kiel University, 2022
5. B. Vonheiden. "Empirical scalability evaluation of window aggregation methods in distributed stream processing". Master's Thesis. Kiel University, 2021
6. J. Grabitzky. "A showcase for the Titan Control Center". Bachelor's Thesis. Kiel University, 2021
7. J. R. Bensien. "Scalability benchmarking of stream processing engines with Apache Beam". Bachelor's Thesis. Kiel University, 2021
8. C. Tsatia Tsida. "Analyzing environmental data with the Titan platform". Master's Thesis. Kiel University, 2020
9. N. A. Biernat. "Scalability benchmarking of Apache Flink". Bachelor's Thesis. Kiel University, 2020
10. T. Koch. "Scalable and interactive real-time visualization of time series data". Bachelor's Thesis. Kiel University, 2020
11. L. Boguhn. "Forecasting power consumption of manufacturing industries using neural networks". Bachelor's Thesis. Kiel University, 2020
12. S. B. N. F. A. Ehrenstein. "Distributed sensor management for an Industrial DevOps monitoring platform". Bachelor's Thesis. Kiel University, 2019
13. D. B. Wetzel. "Entwicklung eines Dashboards für eine Industrial DevOps Monitoring Plattform". Bachelor's Thesis. Kiel University, 2019
14. A. Hansen. "Exploring an energy-status-data set from industrial production". Bachelor's Thesis. Kiel University, 2019

1.6 Document Structure

This thesis is structured into 5 parts, containing 18 chapters including this introductory chapter (Chapter 1).

Part I discusses the foundations for this thesis. Chapter 2 provides and discusses the definition of scalability our benchmarking method builds upon. Chapter 3 introduces the general concept of benchmarking as a research method. Chapter 4 discusses the concepts of cloud-native applications, microservice architectures, and event-driven microservices in particular.

Part II addresses the first goal of this thesis and presents our scalability benchmarking method. Chapter 5 describes our research design and employed methods. Chapter 6 addresses RQ 1.1 and derives our scalability metrics. Chapter 7 addresses RQ 1.2 and describes our method to measure scalability according to our metrics. Chapter 8 addresses RQ 1.3 and presents our software architecture for a scalability benchmarking tool.

Part III addresses the second goal of this thesis and presents our scalability benchmark for event-driven microservices. Chapter 9 describes our research design and employed methods. Chapter 10 addresses RQ 2.1 and identifies relevant use cases for event-driven microservices from studying the domain of analyzing power consumption data within real industrial settings. Chapter 11 addresses RQ 2.2–2.4 and presents our Theodolite scalability benchmarks for event-driven microservices.

Part IV consists of 5 evaluation chapters. Chapter 12 evaluates the effects of our benchmarking method's configuration options on statistical grounding and time-efficient execution using our benchmarks for event-driven microservices. Chapter 13 evaluates the scalability of state-of-the-art stream processing frameworks and different configuration and deployment options using these benchmarks. Chapter 14 evaluates the scalability of different methods for time window aggregations in stream processing using. Chapter 15 evaluates and compares cost scalability of stream processing deployments in the cloud with FaaS offerings. Chapter 16 reports on three case studies, which evaluate our scalability benchmarking method for other cloud-native applications beyond our event-driven microservice benchmarks.

In Part V, Chapter 17 concludes this thesis and Chapter 18 points out future research directions based on the contributions of this thesis.

Part I

Foundations

Chapter 2

Scalability of Software Systems

Just because a software system works well for a particular load or task size, it does not necessarily work well if load or task size increase. Even with increased computing resources, desired quality of service [BHP+06] may no longer be achieved. For example, response times increase or services become unavailable. Intuitively, a software system can be considered scalable if it is functioning in an adequate way, independently of the load it is subject to.

This chapter introduces a precise definition of scalability, building the foundation of the scalability benchmarking method presented in this thesis. We start by looking at how scalability is defined in general parallel and distributed systems in Section 2.1. Afterward in Section 2.2, we discuss scalability for the special case of cloud computing including the scalability definition used in this thesis. Section 2.3 shows how this definition covers both vertical and horizontal scalability and Section 2.4 delimits the concept of scalability from elasticity.

2.1 Scalability in Parallel and Distributed Systems

In traditional parallel computing research, scalability is often discussed in the context of the speed-up [Hil90]. It describes how much faster n processors can compute a task compared to a single processor. More formally [EZL89], the speed-up $S(n)$ is defined as the ratio of the time required by a single processor T_1 to the time required by n processors T_n,

$$S(n) = \frac{T_1}{T_n}.$$

In an ideal case, this ratio increases linearly and, in particular, is $S(n) = n$ for each number of processors n. As stated by Amdahl's law [Amd67], however, this is never the case in reality as every parallel program also has a sequential component.

Scalability definitions based on the speed-up are still frequently used in high-performance computing (HPC), where typically a certain problem has to be solved as fast as possible. In this context, also the distinction between strong and weak scalability is common [HB15]: Strong scalability describes how the completion time evolves with increasing processors for a fixed-size problem. Weak scalability describes how the completion time evolves with increasing processors while also scaling the problem size proportionally.

While there are several works transferring speed-up-based definitions of scalability to distributed systems, Jogalekar and Woodside [JW00] argue that scalability in distributed systems is more complex and should be discussed differently. Besides several technical differences between parallel and distributed systems, they state that distributed systems do not run a single job, but instead process multiple jobs at a time arriving continuously. Although we would argue that this property is not specific to distributed systems, but rather a property of any continuously operating system (e.g., web services or stream processing systems), their criticism remains valid. Furthermore, Jogalekar and Woodside [JW00] argue that "scalability means not just the ability to operate, but to operate efficiently and with adequate quality of service, over the given range of configurations." In a closely related work, Bondi [Bon00] define *load scalability* as "the ability to function gracefully at light, moderate, or heavy loads while making good use of available resources." A more precise definition is given by Smith and Williams [SW02] stating that "scalability is the ability of a system to continue its response time or throughput objectives as the demand for the software functions increases." Duboc et al. [DRW07] generalize several scalability definitions to also incorporate other quality of service objectives. According to van Hoorn [vHoo14], scalability can thus be considered a "meta-qualty" of other quality characteristic.

2.2 Scalability in Cloud Computing

More recently, definitions such as the one from Duboc et al. [DRW07; DLR13] have been specified to target the peculiarities of scalability in cloud computing [HKR13; LEB15; BHI$^+$17]. A definition of scalability in cloud computing is, for example, given by Herbst et al. [HKR13], which states that "scalability is the ability of [a] system to sustain increasing workloads by making use of additional resources". In a subsequent work, Weber et al. [WHG$^+$14] refine this definition and highlight that scalability is characterized by the following three attributes:

Load intensity is the input variable to which a system is subjected. Scalability is evaluated within a range of load intensities.

Service levels objectives (SLOs) are measurable quality criteria that have to be fulfilled for every load intensity.

Provisioned resources can be increased to meet the SLOs if load intensities increase.

A software system can be considered scalable within a certain load intensity range if for all load intensities within that range it is able to meet its service level objectives, potentially by using additional resources. Both load intensity and provisioned resources can be evaluated with respect to different dimensions. Typical load dimensions are, for example, the number of concurrent users or number of requests, while resources are often varied in the number of processing instances or equipment of the individual instances. This understanding of scalability (albeit less formally) is shared by textbooks addressed to practitioners [Kle17; Gor22].

A similar definition, albeit formulated inversely, is used in multiple publications of the "CloudScale" project [LEB15; BHI$^+$17; BMH$^+$21]. They define scalability as "a system's ability to increase its capacity by consuming more resources" [BMH$^+$21], where capacity describes the maximum load the system can handle while fulfilling all "quality thresholds". Here, the notion of quality thresholds corresponds to what Weber et al. [WHG$^+$14] and others term SLOs.

2.3 Vertical and Horizontal Scalability

A distinction is often made between horizontal and vertical scalability [MMS$^+$07; LEB15]:

Horizontal scaling refers to adding computing nodes to cope with increasing load intensities.

Vertical scaling means increasing the computing resources of a single node. A special case of vertical scaling in cloud computing is migrating from one VM type to another [WHG$^+$14].

The scalability definition presented previously covers both horizontal and vertical scalability as both refer to different types of provisioned resources [WHG$^+$14].

In cloud-native deployments, the underlying physical or virtualized hardware is usually abstracted by containerization and orchestration techniques. Nevertheless, different types of scaling resources also exist in cloud-native applications. Orchestration tools such as Kubernetes allow for limiting the CPU or memory resources of containers. Increasing these resources can be considered a form of vertical scaling. On the hand, such orchestration tools also allow increasing the number of container replicas, representing a form of horizontal scaling.

2.4 Scalability vs. Elasticity

Another quality that is often used in cloud computing is elasticity [LEB15]. Scalability and elasticity are related, but elasticity takes temporal aspects into account. It describes how fast and how precisely a system adapts its provided resources to changing load intensities [HKR13; ILF$^+$12]. Scalability, on the other hand, is a time-free notion describing whether increasing load intensities can be handled eventually. Scalability is a subquality of adaptability, meaning that a system can be adapted to increasing loads, while elasticity always requires some auto-scaling components, which performs such adaptations. Scalability is a prerequisite for elasticity since only a scalable system can be scaled by an auto-scaler.

Chapter 3

Benchmarking Software Systems

Benchmarking is a method used in software engineering to assess and compare the quality of different methods, techniques, and tools. In contrast to mathematical proofs or reasoning based on expert knowledge, benchmarking is based on standardized experiments to evaluate the strengths and weaknesses of different test subjects. In this thesis, we use benchmarking to empirically evaluate cloud-native applications regarding their scalability.

We start this chapter by providing a definition for the terms benchmarks and benchmarking (Section 3.1). Afterward, we discuss why and how benchmarking is used as an empirical method in software engineering research (Section 3.2). In Section 3.3, we give an overview of the individual components of a benchmark. In Section 3.4, we discuss quality attributes of benchmarks often demanded in the literature, while Section 3.5 shows different classes of benchmarks. Section 3.6 concludes this chapter by discussing particularities for benchmarking in cloud computing.

3.1 Definitions for Benchmarks

A definition of a benchmark is formulated by Kounev et al. [KLvK20] and states that a benchmark is a "standard tool coupled with a methodology for the evaluation and comparison of systems or components according to specific characteristics, such as performance, reliability, or security". This definition is built upon the perspectives of the Standard Performance Evaluation Corporation (SPEC) and the Transaction Processing Performance Council (TCP), two industrial consortia, which also stress the competitive aspects of benchmarks [vKAH+15]. The system or component from the previous definition, which is to be evaluated, is commonly referred to as system under test (SUT) [KLvK20]. An SUT can be any software system, component, or service. In particular, SUTs can also be variations of the same system, component, or service, for example, configured or deployed in different ways.

While both Kounev et al. [KLvK20] and von Kistowski et al. [vKAH+15] leave it open whether the tool must be an executable software, other definitions are less strict so that the benchmark simply describes how SUTs are evaluated and compared [SEH03]. In practice, we can observe that benchmarks usually come with an executable tool, but applying a benchmark to other SUTs often requires modifying or rebuilding that tool.

Benchmarking describes the process of systematically applying benchmarks [BWT17], that is, stressing an SUT according to the benchmark's description or, more commonly, executing a benchmarking tool. This is often done as part of a research study to compare different SUTs or configurations of the same SUT in different environments or at different times [BWT17].

3.2 Benchmarking as an Empirical Method in Software Engineering Research

Over the last decades, the use of empirical methods in software engineering research has tremendously increased [Bas13; MP19]. Often a distinction is made between qualitative and quantitative research with controlled experiments being a method frequently used in quantitative studies [WHH03]. Controlled experiments are conducted to compare different software engineering techniques, methods, and tools. In controlled experiments, dependent and independent variables are defined with the objective to determine to what extent independent variables affect dependent variables while keeping a high level of control over confounding factors.

Benchmarking can be seen as a form of experimentation [Tic14], in which qualities (dependent variables) of methods, techniques, and tools (independent variables) are studied while aiming for a high level of control over all variables affecting the outcome [Has21]. However, benchmarking shares also characteristics from case study research [SEH03]. A major advantage of benchmarks is that, in contrast to several other research methods, they do not require human subjects [Tic14]. Also, analysis and interpretation of experiments can be automated to a great extent. This significantly speeds up the research process as a wide range of subjects can be tested at low costs and at early stages [Tic98; Tic14]. Automation and precise specification of benchmarks also support repeatability of research [SEH03; Tic14].

Sim et al. [SEH03] argue that benchmarks advance research as their creation and adaption often come with rapid technical progress and community building. The existence of benchmarks therefore also indicates the maturity of a research area [SEH03]. As noted by Tichy [Tic98], building a benchmark is intense work on its own and should be shared by a research community. While Sim et al. [SEH03] state that benchmarks even must be constructed in a community effort, Waller [Wal14] argue that it suffices to start building a benchmark with a small group of researchers as an offer to a larger community.

The recently published ACM SIGSOFT Empirical Standards for Software Engineering Research [RbAB^{+}21] define expectations for empirical software engineering research. There exist different standards for different research methods with each standard defining essential, desirable, and extraordinary attributes as well as examples, antipatterns, and invalid criticisms. There is one standard for benchmarking studies [Has21], covering both defining benchmarks and executing them.[1] The ACM SIGSOFT Empirical Standards are meant to serve as evaluation guidelines for manuscripts in the scientific peer review process and to help researchers conduct studies of higher quality.

3.3 Components of Benchmarks

The ACM SIGSOFT Empirical Standard for Benchmarking [RbAB^{+}21; Has21] names four essential components of a benchmark:

Quality The quality of a system that is benchmarked should be clearly named. Typical qualities are performance, availability, security, or—as in this thesis—scalability.

[1] `https://acmsigsoft.github.io/EmpiricalStandards/docs/?standard=Benchmarking`

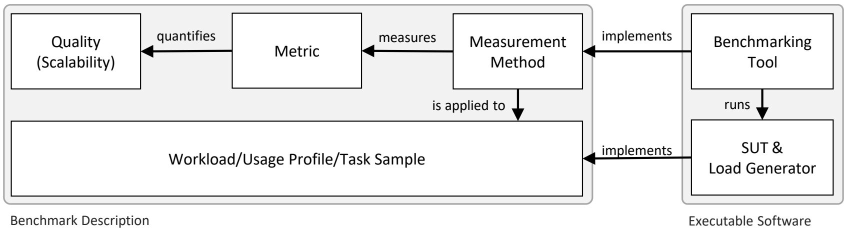


Figure 3.1. Components of a benchmark. A benchmark description defines the quality to be benchmarked (scalability in this thesis), a metric quantifying the quality, a corresponding measurement method, and workloads, usage profiles, or task samples to which the benchmarking method can be applied. Additionally, benchmarks usually come with an executable software, implementing the measurement method as well as SUTs and load generators implementing the workload, usage profiles, or task samples.

Metrics Metrics quantify the quality and, thus, allow comparing different SUTs regarding that quality.

Measurement methods Measurement methods define how operational data of SUTs is collected and analyzed. The results of applying the methods must correspond to the benchmark's metrics.

Workloads, usage profiles, and task samples They describe what the SUTs are doing when the measurements are taken. Depending on the type of SUT, this can be a description of the load an SUT is subject to, a description of how an SUT is used, a task sample that the SUT implements, or a combination.

In addition to these components, benchmarks usually come with a benchmarking tool to automate the benchmarking process. Fig. 3.1 shows the components of a benchmark and their relations.

A typical benchmarking tool architecture contains separate components for load generation and the SUT [BWT17]. The SUT is either a ready-to-use software (or service) or a software (or service) that implements a task sample defined by the benchmark. The load generation component stresses the SUT according to the workload or usage profile defined by the benchmark. Additional components in a benchmarking architecture are responsible for experiment controlling, data collection, and data analysis and, thus, implement the benchmark's measurement method [BWT17]. Sometimes also a visualization component for passive observation is included [BWT17].

3.4 Quality Attributes of Benchmarks

A set of five desired quality attributes for benchmarks is presented by von Kistowski et al. [vKAH$^+$15], which represents the perspectives of the SPEC and TPC committees:

Relevance The benchmark's behavior should closely correlate to behaviors that are of interest to consumers of the results.

Reproducibility Benchmarks should consistently produce similar results when they are run with the same test configuration.

Fairness The benchmark should allow different test configurations to compete on their merits without artificial limitations.

Verifiability The benchmark's results should provide confidence that they are accurate.

Usability Obstacles for users to run the benchmark in their test environments should be avoided.

These and similar quality attributes can also be found by Gray [Gra93], Sim et al. [SEH03], Huppler [Hup09], and Folkerts et al. [FAS$^+$13] as well as in textbooks on (cloud) benchmarking [BWT17; KLvK20].

3.5 Classification of Benchmarks

There are two common dimensions along which benchmarks can be classified. The first concerns the way how a benchmark is provided to its users and the second one concerns the size and scope of a benchmark's task sample [KLvK20].

3.5.1 Provision of Benchmarks

Regarding the way how they are provided, benchmarks can be classified into specification-based, kit-based, and hybrid benchmarks [vKAH+15].

Specification-based benchmarks describe a business problem to be solved by an SUT along with input parameters and, hence, also the expected output. The actual implementation is left to the individual running the benchmark and may vary among different SUTs. This allows different SUTs to implement a task in a way best suited for them and without being artificially constrained. Hence, specification-based benchmarks by design provide a high level of fairness (see Section 3.4). On the other hand, specification-based benchmarks require a higher implementation effort and detailed knowledge about an SUT. Verifiability might thus be impaired as verifying whether an implementation fulfills the benchmark's specification can be complex.

Kit-based benchmarks come with the actual implementation of a "specification". This significantly eases using the benchmark and supports verifiability (see Section 3.4), but may hamper innovation. SUTs implementing a business function in a substantially different way may not be honored with better benchmark results. Moreover, a kit-based benchmark might eventually deviate from the state-of-the-art of solving a certain problem, requiring permanent benchmark adaption.

Hybrid benchmarks are in between specification-based and kit-based benchmarks. They provide the majority of the benchmark's function as a ready-to-use implementation, but allow certain parts to be implemented in a way specifically tailored for the SUT.

3.5.2 Sizes and Scopes of Benchmarks

Regarding their size and scope, benchmarks are often classified into microbenchmarks and application benchmarks [BWT17; KLvK20; Wal14]. According to Waller [Wal14], other categories sometimes found in the literature [Lil00] can be considered as subcategories of microbenchmarks and application benchmarks.

Microbenchmarks test a specific part of a system independent of the rest of the system. This can be a specific operation or (small) component.

Due to their limited scope, microbenchmarks are usually designed as white box tests, meaning based on the available source code of a software [Wal14]. In general, microbenchmarks are simpler to implement and execute than application benchmarks, not least because of the supporting tools available. Hence, they often come as kit-based benchmarks. As they also have shorter execution times, microbenchmarking is mostly done with large numbers of repetition, providing a high internal validity of the results. Care should be taken when interpreting the results of microbenchmarks. Due to their simplicity, they cannot assess complex interactions between different qualities of a system. Hence, there is a risk of making false conclusions regarding the SUT's behavior in production and, thus, optimizing SUTs for unrealistic use cases [Wal14].

Application benchmarks (also referred to as application-driven benchmarks [BWT17] or macrobenchmarks [Wal14]) test entire systems, applications, or components at a larger scale (e.g., independently deployable microservices as discussed in Section 4.2). Due to their size, application benchmarks consider the SUT as black box and test it based on their specification. Hence, application benchmarks are often designed as specification-based benchmarks. As application benchmarks need to test the behavior of entire applications, they are more laborious to implement. Given their black-box-test nature, it is also more difficult to draw conclusions about the root cause of observations. On the other hand, application benchmarks are considered to capture the behavior of realistic applications [KLvK20], hence being of high relevance.

3.6 Benchmarking in Cloud Computing

Benchmarking as a method for measuring the quality of software systems becomes particularly important if those systems are running in the cloud or if they are relying on cloud services [BWT17]. Cloud providers allow only limited insight into the inner working of their services and offer only limited quality guarantees, leaving benchmarking as the only option to study their quality of services. In fact, there are several cases in which specific behavior of cloud services could only be discovered by conducting rigorous benchmarking [Ber17].

As cloud service benchmarkers usually have no chance to understand the function of cloud services in detail, benchmarking is often done from a client's perspective [BWT17]. That means benchmarks, which particularly evaluate a cloud service or consider interactions of an SUT with a cloud service, are often designed as application benchmarks. However, of cause also microbenchmarks can be executed in cloud environments, for example, to study the impact of the cloud environment on the benchmarks themselves [LSL19].

Performance experiments in computer science exhibit large variability in their results for various reasons [MDJ+18]. For experiments in public cloud environments, this variability is even larger due to effects of changing physical hardware or software of different customers running on the same hardware [AB17]. To increase reproducibility, measurements should therefore be repeated, potentially with different configurations, and the confidence in the final results should be quantified [PVB+21]. Moreover, careful description of the experiment design and availability of artifacts allows repeating experiments and validating results of benchmarking studies [PVB+21]. When benchmarking containerized software, one should be aware that results are not directly generalizable to non-containerized deployments [GHP+19].

Chapter 4

Cloud-Native Applications and Event-Driven Microservices

Cloud-native applications are a contemporary phenomenon of software systems, explicitly designed and built to run in the cloud. Microservices are an architectural pattern for building large-scale cloud-native applications. Event-driven microservices are a special type of microservice architectures, in which individual services primarily communicate asynchronously, usually via messaging systems. They enable a further level of scalability, as required, for example, for big data online analytics.

We start this chapter by giving a brief introduction to the characteristics of cloud-native applications in Section 4.1. Afterward, we describe the microservice architectural pattern in more detail in Section 4.2, followed by providing the foundations for a specific subtype, namely event-driven microservices (Section 4.3). Finally, we discuss the foundations of distributed stream processing, a pattern increasingly used to design and implement event-driven microservices (Section 4.4).

4.1 Cloud-Native Applications

In the following, we briefly outline the cloud computing paradigm, provide a definition for cloud-native applications, and give an overview of the cloud-native ecosystem.

4.1.1 Cloud Computing

With cloud computing, computing, storage, and network resources are provided "as a service" via the Internet [AFG+10; TBP+22]. Cloud customers can acquire such resources dynamically on demand and at self-service, while being billed on a per-use basis. Under the impression of infinite resources available, cloud customers can thus scale their software systems to react to ever-increasing, bursting, periodically fluctuating, or even unpredictable loads, without having to provide the necessary resources in advance [FLR+14]. Cloud computing is enabled by pooling the physical resources for many customers in large clusters. Since not all customers need resources to the same degree simultaneously, cloud providers can allocate resources to customers on demand.

A frequently cited definition of cloud computing was provided by the US National Institute of Standards and Technology (NIST) in 2011 [MG11]. In addition to the five essential cloud computing characteristics on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service, this definition also names the four deployment models private cloud, community cloud, public cloud, and hybrid cloud—a classification which is still common today. The NIST definition also differentiates between infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). However, it is argued that this differentiation is outdated [Ber17] since the emergence of container technologies and Function-as-a-Service (FaaS) offerings.

A central technology for enabling cloud computing is virtualization [TBP+22]. Virtualization allows placing software and data of different cloud users on the same computing or storage resources in isolation.[1] In contrast to physical resources, virtual ones can be created on demand, enabling the rapid elasticity required for cloud computing. Infrastructure as Code is a practice, assisting cloud users to acquire virtual resources. There are different levels of virtualization such as virtual machines and containerization, in practice often used together.

4.1.2 A Definition for Cloud-Native

A definition of the term "cloud-native" is provided by the Cloud Native Computing Foundation (CNCF),[2] a suborganization within the Linux

[1]Although co-located users should not be able to see each other's data or processes, they may still affect each other's performance as discussed in Section 3.6.

[2]`https://www.cncf.io`

Foundation. It states that "cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds" [Clo18]. Examples are containerization, service meshes, microservices, immutable infrastructure, and declarative APIs. They enable loosely coupled systems that are resilient, manageable, and observable as well as frequent and predictable changes when combined with robust automation.

An earlier yet similar definition can be found by Kratzke and Quint [KQ17], which is based on reviewing the state of literature in 2017. Pahl et al. [PJZ18] discuss architectural principles for cloud-native systems, describing some of the aforementioned properties in more detail. Like the CNCF's definition, also Kratzke and Quint [KQ17] and Pahl et al. [PJZ18] emphasize that cloud-native applications must be scalable.

4.1.3 The Cloud-Native Ecosystem

Supported by major cloud vendors, an entire ecosystem of tools has emerged for simplifying, accelerating, and securing the development and operation of software systems in the cloud. The natural home for many such projects is the Cloud Native Computing Foundation. In the following, we briefly introduce Kubernetes and Prometheus as examples of such tools, which are also of particular relevance to this work. Additionally, we introduce Kubernetes operators, a design pattern often used for cloud-native tools.

Kubernetes Kubernetes [BGO+16] is the de-facto standard orchestration tool for cloud-native applications [Clo22]. Users describe the desired state of an application in a declarative way as Kubernetes objects. In a continuous reconciliation loop, Kubernetes adopts the current system to reach the desired state. A typical example of an object is a Deployment, which describes an application component (e.g., a microservice) consisting of container images, a desired number of replicas, volumes to be mounted, and others. A common way for users to interact with Kubernetes is by describing objects in YAML files (manifests) and applying these files to Kubernetes by using the kubectl command-line tool.

Prometheus Prometheus [Pro22] is a tool for cloud-native observability [Kra22], which is widely used in production [Clo22]. It continuously requests metrics data from so-called exporters and stores them in a time series database. Other tools can request these metrics via the PromQL query language, for example, for visualization, alerting, or automatic SLO assessment. Numerous exporters are available for various systems, including web servers, databases, and messaging systems. Moreover, several cloud-native tools and frameworks (e.g., Kubernetes) support Prometheus' data format and API without requiring an additional exporter.

Operator Pattern The Kubernetes Operator pattern [HS19; IH19] is a recent approach to integrating domain knowledge into Kubernetes' orchestration process. Implementing this pattern involves two things: First, custom resource definitions (CRDs) define new types of resources that can be managed by the Kubernetes API. Second, a dedicated software component, the operator, runs inside the Kubernetes cluster and manages the entire life cycle of what is described by the CRDs. The operator interacts with the Kubernetes API and reacts to the creation, modification, or removal of custom resource objects.

Prominent examples of Kubernetes operators are the Prometheus Operator and Strimzi.[3] While the former allows registering monitoring endpoints for the Prometheus monitoring tool at the infrastructure level, the latter allows maintaining Apache Kafka messaging topics along with other software components.

4.2 The Microservice Architectural Pattern

Microservice architectures are a pattern, particularly suited for building cloud-native applications [BHJ16a; GBS17; PJZ18].[4] In the following, we describe the core characteristic of microservice-based applications, discuss

[3] `https://prometheus-operator.dev`, `https://strimzi.io`

[4] Although microservices are generally considered to be cloud-native, it should be noted that there are good reasons for certain types of cloud-native technologies not to be implemented as microservices [MBM$^+$21].

how they enable scalability, and give an overview of the state-of-the-art in microservice performance testing and benchmarking.

4.2.1 Microservice Characteristics

Microservices are an architectural style for the modularization of large-scale software systems [Has18]. In microservice architectures, a single application is composed of multiple small services that are built around business capabilities [LF14]. Individual microservices run in their own processes, may use different technology stacks, and communicate via lightweight, fault-tolerant mechanisms over the network. This allows services to be deployed and scaled individually. Microservices can be considered an implementation of service-oriented architectures [Zim17]. Newman [New15] lists seven principles for designing microservices:

- Model around business concepts
- Adopt a culture of automation
- Hide internal implementation details
- Decentralize all the things
- Independently deployable
- Isolate failure
- Highly observable

Similar to these principles, Lewis and Fowler [LF14] list a set of nine characteristics of microservices. While both principles and characteristics have a lot of overlap [Zim17], Lewis and Fowler [LF14] also cover aspects focusing more on organizing the development and operation of microservices.

4.2.2 Microservices for Enabling Scalability

Microservice architectures promise to enable scalability for various reasons [Has16]. In contrast to large monolithic software systems that can only be scaled as a whole, microservices are loosely coupled, allowing for scaling individual microservices independently. Since microservice architectures should be designed such that they can tolerate the failure of services at any time, also scaling them becomes more resilient. Supported by containerization technologies and cloud computing, individual services can easily be replicated under high loads. Moreover, as services are decomposed based on business functions, this replication can also be triggered based on the demand for these functions.

The Different Dimensions of Scaling Microservices

A model often used to describe the different options to scale software is the scale cube [AF09]. In essence, it describes that an application can be scaled horizontally along the three dimensions of horizontal duplication, data partitioning, and function decomposition. A great benefit of microservice architectures is that they support scaling software along all these dimensions:

Horizontal duplication simply means running multiple replicas of the same microservice or microservice container. Combined with some type of load balancer, each instance handles only part of the load.

Data partitioning is also implemented by running multiple instances of a microservice, yet with each instance being only responsible for a subset of the data. This is typically implemented by partitioning the overall data set of a service based on some attribute (e.g., the user ID) and redirecting incoming requests or messages to the corresponding instance.

Function decomposition is enabled by designing microservices around business capabilities. Each service handles only load for a subset of the application's functions.

As noted by Newman [New21], there is also the option to scale microservices vertically, meaning to provide more computing resources. At least to a certain degree, this is very easy with public cloud providers and orchestration tools such as Kubernetes.

Scalability as a Main Driver for Microservice Adoption

Empirical research shows that scalability is considered a core quality attribute of microservices and one of the main drivers for their adoption from both academia and industry. In an early systematic mapping study on microservices, Pahl and Jamshidi [PJ16] found that scalability is the most frequently mentioned quality of microservice architectures. Similarly, in a systematic literature review on quality attributes of microservice architectures, Li et al. [LZJ+21] reported that scalability is the most frequently addressed attribute. Kratzke and Quint [KQ17] conducted an early systematic mapping study on cloud-native applications, which also showed that scalability is perceived as a fundamental property of cloud-native applications, especially in combination with microservice architectures.

In a qualitative study, Taibi et al. [TLP17] interviewed 21 practitioners who adopted microservice architectures at least two years before their study. They report that scalability was one of the most frequently mentioned motivation drivers for microservice migration. Furthermore, scalability improvement was also named as a main benefit of the migration. Similarly, Fritzsch et al. [FBW+19] conducted an interview study with 16 software professionals from 10 companies, which also showed that scalability of the architecture was a main driver for migrating legacy applications to microservices. More recently, interviews with practitioners from 20 software companies in China also highlight scalability as a main motivation for adopting microservices [ZLC+23]. In a larger, quantitative study, Knoche and Hasselbring [KH19] surveyed 71 professionals from Germany regarding drivers and barriers for microservice adoption. They found that—almost independently of the industry domain—high scalability and elasticity were mentioned most frequently as drivers for adopting microservices. Improving scalability was also frequently mentioned as a goal of modernization using microservices.

Soldani et al. [STV18] systematically reviewed industrial gray literature (i.e., blog posts, whitepapers, and videos) on microservice adoption. They report that scalability can be considered the most important benefit of microservices from an operational point of view. In a large mixed-method study, combining a systematic literature review on microservice adoption in industry, analyzing open-source repositories, and an online survey with practitioners and researchers, scalability was identified as major driver for adopting microservices [LZS+21]. Experience reports from adopting microservice architectures in industrial software systems for scalability are reported by, for example, Balalaie et al. [BHJ16b], Hasselbring and Steinacker [HS17], and Bucchiarone et al. [BDD+18].

4.2.3 Performance Testing and Benchmarking of Microservice Architectures

Eismann et al. [EBS+20] report that microservice-based architectures have some benefits for performance testing compared to large monolithic software systems. This concerns in particular their self-containment and loose coupling, which allows testing individual microeservices in isolation. Containerization helps with setting up a test environment and the ecosystem of cloud-native monitoring tools provides an easier access to performance metrics. Additionally, microservices allow integrating performance testing in DevOps. However, regarding the last point there seems to be some disagreement. While Bermbach [Ber17] also state that microservices foster benchmarking as part of the build process, Heinrich et al. [HvHK+17] argue that continuous performance testing becomes more difficult with microservices being re-deployed frequently. Additional challenges for performance testing of microservices were investigated in an experimental study by Eismann et al. [EBS+20]. They found that the execution environments of microservices (i.e., cloud platforms) exhibit unstable behavior, affecting reproducibility of performance evaluations. It particular, detecting small performance differences is difficult (e.g., between two versions).

Aderaldo et al. [AMP+17] define a set of requirements for microservice benchmarks. Their first requirement is that a microservice benchmark should not only provide a description of all included microservices, but should also explicitly define how services interact. Moreover, it should employ common patterns of microservice architectures and provide alternate implementations for certain parts of the microservice. Other requirements include the adaption of DevOps principles as part of the benchmarks development process and relevance for the research community. Approaches to reduce the benchmarking efforts are, for example, presented by Grambow et al. [GMW+20] and Düllmann and van Hoorn [DvH17].

4.3 Event-Driven Microservices

Event-driven microservice architecture are an approach combining event-driven architectures (EDA) with microservices. In such architectures, individual microservice communicate with each other primarily asynchronously by issuing and consuming events [Bel20]. An important property is that log-based messaging systems are often used to not only transmit and queue events, but also to partition, replicate, and replay event streams [KBS19]. Combined with distributed stream processing architecture patterns and frameworks, this enables the scalability, fault tolerance, and determinism often required by microservice applications [KF19].

At the time of writing this thesis, the topic of event-driven microservices architectures is covered only superficially in scientific literature. While there are a couple of case studies reporting on event-driven microservices, research is still lacking a systematic evaluation of this new architectural style. On the other hand, some textbooks for practitioners [Bel20; Sto18] were recently published, which also serve as a reference for this section. Despite the lack of systematic studies, event-driven microservices have been named as an emerging trend [FCK^{+}20; KÇC^{+}21] and the need for further research on this topic has been recognized [KF19; LZS^{+}21].

4.3.1 Log-Based Messaging Systems as Backbone

Event-driven microservices employ log-based messaging systems for their communication. To eventually reach consistency among individual event-driven microservices, the log must be durable, append-only, fault-tolerant, partitioned, and must support sequential reads [KBS19]. Probably the most prominent messaging system fulfilling these properties is Apache Kafka [KNR11; WKS^{+}15], which is heavily used in industry.[5]

4.3.2 Event-Driven Microservice Architecture Patterns

In the following, we give a brief overview of architectural patterns for designing event-driven microservices. Although in this thesis, we mainly

[5] `https://kafka.apache.org/powered-by`

focus on microservices employing distributed stream processing techniques, we also briefly introduce basic producer and consumer services and Function-as-a-Service deployments.

Basic producer and consumer Initial publications on microservices already report on using lightweight messaging buses for asynchronous communication between services [LF14; New15]. A common pattern is that microservices include basic producers and consumers [Bel20] to publish own events or subscribe to the events of other services, respectively [KÇC+21]. For example, to provide quick responses to user requests, services might hold copies of the state of other services, which is updated based on an event stream of the other service [HS17].

Distributed stream processing Scaling microservices, which include basic event consumers but integrate data of all instances via a database, is only feasible to a limited extent. As data volumes increase, such a database quickly becomes the microservice's bottleneck. Designing event-driven microservices around the patterns and dataflow models of modern stream processing frameworks can be an alternative for building highly scalable systems [Bel20]. Core principle is to model the internal architecture of microservices as dataflow graphs. All stateful operations within these graphs are performed on partitioned data streams, allowing for state locality (see the data partitioning dimension of the scale cube described in Section 4.2.2). In the following Section 4.4, we describe the fundamental models and patterns for distributed stream processing as well as state-of-the-art frameworks in more detail.

Function-as-a-Service (FaaS) In recent years, the FaaS programming model has become increasingly popular [CIM+19; LWS+19; ESvE+21]. It offers a *serverless* alternative for building event-driven microservices [Bel20; BCK+21; Roh22] by composing applications of small, stateless functions, which can also be chained to form more complex dataflow architectures. There seems to be no consensus on whether a microservice is implemented as a single function or is composed of multiple functions [ESvE+21]. FaaS offerings of public cloud providers (e.g., AWS Lambda or Google Cloud

Functions), allow engineers to deploy functions on managed infrastructure that are billed per invocation and run duration. As hardware resources are managed and scaled by the cloud provider, FaaS is often promoted for its virtually limitless scalability. We conduct an experimental evaluation to compare FaaS and stream processing implementations regarding their cost scalability in Chapter 15 of this thesis.

4.4 Distributed Stream Processing

While research on software systems for processing continuous streams of data dates back to the early 1990s, the advent of cloud computing and MapReduce [DG08] led to a second generation of stream processing systems [FCK$^+$20; CFK$^+$20]. Such "modern" systems are designed to run in a distributed fashion on commodity hardware in order to scale with massive amounts of data. Besides high throughput, these systems focus on low latency, fault tolerance, and coping with out-of-order streams.

In the following, we give an overview of the underlying dataflow models and processing patterns of modern stream processing frameworks, before we introduce the frameworks further considered in this thesis.

4.4.1 Stream Processing Models and Patterns

Modern stream processing frameworks process data in jobs, where a job is defined as a dataflow graph of processing operators. They can be started with multiple instances (e.g., on different computing nodes, containers, or with multiple threads). For each job, each instance processes only a portion of the data. Whereas isolated processing of data records is not affected by the assignment of data portions to instances, processing that relies on previous data records (e.g., aggregations over time windows) requires the management of state. Similar to the MapReduce programming model, keys are assigned to records and the stream processing frameworks guarantee that all records with the same key are processed by the same instance. Hence, no state synchronization among instances is required. If a processing operator changes the record key and a subsequent operator performs a stateful operation, the stream processing framework splits the dataflow graph into subgraphs, which can be processed independently by different instances. We refer to the recent surveys of Fragkoulis et al. [FCK+20] and Margara et al. [MCF+22] for detailed information on state-of-the-art stream processing models and patterns.

4.4.2 Modern Stream Processing Frameworks

In the following, we give a brief overview of modern distributed stream processing frameworks, particularly suited for implementing event-driven microservices.[6] For a detailed comparison, see the works of, for example, Hesse and Lorenz [HL15], Fragkoulis et al. [FCK+20], and van Dongen [vDon21].

In their textbook, Bellemare [Bel20] distinguishes between lightweight and heavyweight stream processing frameworks for implementing event-driven microservices. Lightweight frameworks are embedded as a programming library into the source code of microservices. The stream processing framework does not require any specific way to build or deploy the microservice. This allows the service to also perform other tasks beyond stream processing such as providing a REST API. Individual instances of a service discover each other (e.g., via features of the messaging system or Kubernetes) and perform the necessary coordination internally. Heavyweight frameworks on the contrary are provided as deployable software systems, which can be configured by one or more stream processing jobs to be executed. They are typically designed as a master–worker architecture. In this thesis, we mainly focus on lightweight frameworks. All modern stream processing frameworks can be deployed containerized on commodity hardware with Kubernetes.

Apache Flink Originating from a scientific research project [ABE+14], Apache Flink [CKE+15] has been extensively used, evaluated, and extended in research and became increasingly popular in industry. It offers one of the most elaborated dataflow models, providing precise control of time and state [CKE+15; CEF+17; ABC+21]. Moreover, Flink provides different

[6]The frameworks discussed in this section are often also referred to as stream processing engines or stream processing systems. In this thesis, we use the term framework as we focus on stream processing employed within microservices.

abstraction layers and a rich feature set regarding the integration with external systems. Flink clearly falls into the category of heavyweight frameworks [Bel20]. Its deployment consists of one or—for fault-tolerance—more coordinating *JobManagers* and a scalable amount of *TaskManagers*. Although heavyweight, we consider Flink in this thesis due to its widespread adoption and since we observe recent trends to more lightweight deployments of Flink.

Apache Kafka Streams Kafka Streams [SWW⁺18; WCD⁺21] is a stream processing framework built on top of Apache Kafka (see Section 4.3.1). It is available as a Java library and, thus, aligns with the idea of incorporating stream processing in standalone microservices. Compared to most other stream processing frameworks, it has a restricted set of features, in particular, concerning the integration with external systems. Kafka Streams only supports Kafka topics as data sources and sinks. According to the Kafka project, however, this can be compensated by the *Kafka Connect* project, which allows transferring data from external systems to Kafka topics and vice versa.

Apache Samza Similar to Kafka Streams, Apache Samza [NPP⁺17] can be embedded as a library in standalone applications. Individual instances of the same application use Apache Zookeeper and Apache Kafka for coordination, data transfer, and fault tolerance. Although still maintained, Samza is sometimes considered a predecessor of Kafka Streams [KK15]. Note that in this thesis, we use Samza as a runner for Apache Beam pipelines (see below), which allows implementing more complex use cases [ZXW⁺20].

Hazelcast Jet Hazelcast Jet [GTĎ⁺21] is a stream processing framework built on top of the Hazelcast IMDG distributed, in-memory object store. It can be embedded into Java applications and does not have any dependencies on an external system. Instead, individual instances discover each other, form a cluster, and handle coordination and data replication internally. Hazelcast Jet differs from other frameworks in its execution model, which is based on a concept similar to coroutines and cooperative threads [GTĎ+21]. With the release of Hazelcast 5.0 in 2021, Hazelcast Jet has been merged with Hazelcast IMDG into one unified product.

Apache Beam Apache Beam is not a stream processing system by itself, but instead, an SDK to implement stream processing jobs in a uniform model, which can be executed by several modern stream processing systems. Apache Beam implements Google's Dataflow model [ABC+15], which is also internally used by Google's cloud service *Google Cloud Dataflow*. A stream processing job implemented with Apache Beam is executed by a so-called runner. Runners can be seen as adapters for the actual stream processing systems. Besides a runner for Google Cloud Dataflow, Apache Beam provides also runners for several other systems, including the aforementioned Apache Flink, Apache Samza, and Hazelcast Jet. Previous research found that using Apache Beam as an abstraction layer comes with a significant negative impact on performance [HMG+19].

Other Stream Processing Frameworks Apache Spark [ZXW+16], Apache Storm [TTS+14], and the successor of the latter, Apache Heron [KBF+15], are considered heavyweight frameworks and, thus, fit less into the context of microservices [Bel20]. Spark differs from the frameworks discussed before in that it processes data streams in "micro-batches". Storm and Heron provide less sophisticated programming models and weaker fault tolerance mechanisms [FCK+20]. Moreover, there are several cloud services available for stream processing, with Google Cloud Dataflow being an example that has received much attention in research [ABB+13; ABC+15; ABC+21]. Although not the main focus of this work, our evaluations in Part IV also include experiments with Apache Spark (Chapter 14) and Google Cloud Dataflow (Chapter 15).

Part II

The Theodolite Benchmarking Method

Chapter 5

Research Design and Methods

In the chapters of this second part of the thesis, we address our first guiding goal, namely engineering a scalability benchmarking method for cloud-native applications. In this chapter, we introduce our employed research design and methods to address this goal.

We align our research design with the structuring of benchmarks into components as described in Section 3.3. Since our first goal is to derive a general scalability benchmarking method, we design a scalability metric and a measurement method. Specific task samples are not included in this goal as our benchmarking method should be applicable to different task samples. As also discussed in Section 3.3, we include engineering a scalability benchmarking tool architecture along with a corresponding implementation. Fig. 5.1 illustrates the benchmarking method components, covered by this part.

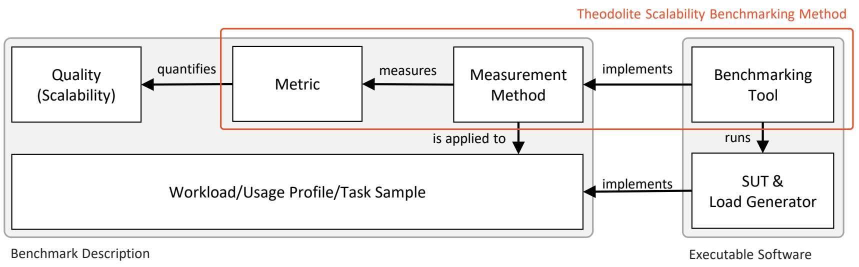


Figure 5.1. Part II of this thesis covers engineering a scalability benchmarking method for cloud-native applications, including scalability metrics, corresponding measurement methods, and an architecture for a corresponding benchmarking tool as highlighted by the red box.

For each of the three benchmarking method's components (the metric, the measurement method, and the tool architecture), we apply engineering research.[1] Specifically, we discuss design rationales, our proposed solution, and strength and weaknesses in the context of related work for each component. An empirical evaluation of our benchmarking method as a whole follows in Part IV. In the following, we describe how we derive design rationales (Section 5.1) and provide an overview of our benchmarking method's evaluations (Section 5.2).

5.1 Deriving Design Rationales

In order to derive design rationales for each component, we build upon the quality attributes of benchmarks described by von Kistowski et al. [vKAH+15] (see Section 3.4). For each quality attribute, we identify the benchmark components that can primarily contribute to implementing the attribute. We derive a mapping shown in Table 5.1, which describes the desired behavior of the individual benchmark components in order to implement the respective quality attribute.

Relevance can mainly be addressed by designing realistic task samples that cover interesting use cases of the SUTs and by a metric quantifying the qualities of the SUT that are of interest to the benchmarker. Reproducibility requires statistically grounded results, which must be achieved at the measurement method level. Additionally, a benchmarking tool simplifying benchmark execution may also help to increase repeatability, which is a prerequisite for reproducibility. Fairness can be obtained by neither using metrics nor task samples that favor one SUT over another. Like reproducibility, verifiability can be achieved by a statistically sound benchmarking method. Usability is mainly addressed by benchmarking tools that can easily be installed, require no or little user intervention when running the benchmark, and allow for adaption to different SUTs or execution environments. In particular, for application benchmarks, usability requires executing benchmarks in reasonable time, which has to be addressed by the measurement method.

[1] https://acmsigsoft.github.io/EmpiricalStandards/docs/?standard=EngineeringResearch

Table 5.1. Quality attributes of benchmarks (see Section 3.4) related to benchmarking components (see Section 3.3) required to implement them.

Quality Attribute	Metric	Measurement Method	Tool Architecture	Task Sample
Relevance	quantifies what the benchmarker really seeks to measure			represents a relevant use case for the SUTs to be benchmarked
Reproducibility		yields statistically grounded results	supports repeatability through simplified benchmark execution	
Fairness	provides an objective measure, independent of the SUT			is not tailored to the strengths and weaknesses of certain SUTs
Verifiability		yields statistically grounded results		
Usability		allows executing benchmarks in reasonable time	simplifies the execution of (potentially modified) benchmarks	

5.2 Evaluation Overview

As we detail in the following chapters, our proposed benchmarking method defines two metrics and is configurable for different load types, resources types, and SLOs. In Part IV of this thesis, we evaluate our scalability benchmarking method with different benchmarks and different configuration options.

Table 5.2 shows an overview of all evaluations conducted in this thesis. Besides our Theodolite benchmarks for event-driven microservices (see Chapter 11), we employ our method to different existing benchmarks or construct new ones for real-world software systems. We perform evalua-

Table 5.2. Overview of evaluations of our scalability benchmarking method, indicating the employed benchmarks, metrics, load types, resource types, and SLOs.

	Bench.		Metric		Load type			Res. type			SLO			
Evaluation	Theodolite	Others	Demand	Capacity	Messages/sec	Concurrent users	Domain-specific	Pod replicas	Pod resources	Cloud service cost	Lag trend	Dropped records	HTTP latency	Domain-specific
Chapter 12	✓		✓[a]	✓[a]	✓			✓			✓			
Chapter 13	✓		✓		✓		✓	✓	✓		✓	✓		
Chapter 14	✓	✓	✓		✓			✓			✓			
Chapter 15	✓		✓		✓					✓	✓			
Section 16.1		✓	✓				✓		✓				✓	✓
Section 16.2		✓		✓	✓			✓			✓	✓		
Section 16.3		✓	✓			✓		✓	✓				✓	

[a] We perform isolated experiments to evaluate the applicability of our scalability metric.

tions with both our metrics introduced in Chapter 6, the resource demand and the load capacity metric. Selected load types are messages per second for streaming-based benchmarks, concurrent users on a web application, and domain-specific ones detailed as detailed in the corresponding chapters. We evaluate scalability regarding the number of instances (i.e., Kubernetes Pods), the CPU and memory resources per instances, and the costs of cloud services. For streaming-based benchmarks we use the *lag trend* SLO and the *dropped records* SLO, introduced in Chapter 11. For HTTP-based benchmarks, we define SLOs based on the latency of HTTP requests. A domain-specific SLO based on application logs is used in Section 16.1. Further evaluations of our benchmarking method can be found by Mertens [Mer22], Boguhn [Bog22], Bensien [Ben21], Biernat [Bie20], and in our previous publication [HH21e].

Chapter 6

Scalability Metrics

This chapter addresses research question RQ 1.1 of this thesis and proposes our two Theodolite metrics for quantifying scalability of cloud-native applications (see Fig. 6.1). Both metrics describe the results of running scalability benchmarks according to our proposed Theodolite benchmarking method. The *demand* metric describes how the required amount of provisioned resources evolves with increasing load intensities. The *capacity* metric indicates how the processible load intensity evolves with increasing amounts of provisioned resources

This chapter builds upon our work previously published in the *Empirical Software Engineering* journal [HH22a] and presented at the *ACM/SPEC International Conference on Performance Engineering 2021* [HH21a]. It is organized as follows. After describing their design rationale in Section 6.1, we introduce our proposed metrics by a formal definition in Section 6.2. Section 6.3 provides a discussion on the strength and weaknesses of both

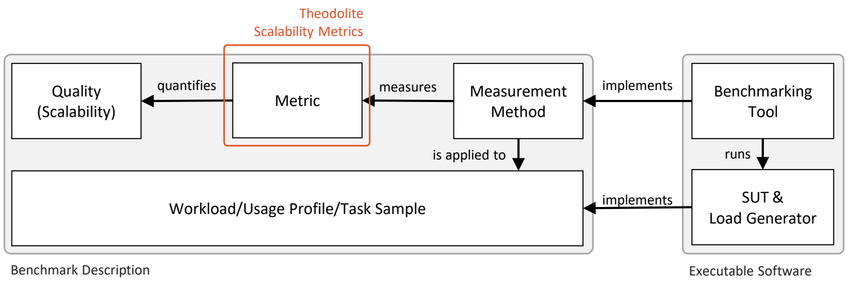


Figure 6.1. Chapter 6 of this thesis presents our Theodolite scalability metrics for cloud-native applications as highlighted by the red box.

metrics along with example benchmark results. Finally, Section 6.4 discusses our metrics in the context of related work.

6.1 Design Rationale

As described in Chapter 5, the major quality attributes of a benchmark's metric are relevance and fairness. To address relevance, a scalability metric should quantify scalability in accordance with its definition. We, therefore, require our metrics to take up the definition we introduced in Section 2.2, which defines scalability based on the notions of load, resources, and SLOs. To provide an objective, SUT-independent measure (i.e., fairness), we require the metrics to support arbitrary load types, resource types, and SLOs as detailed below.

Support for different load types Load on cloud-native applications can increase in various dimensions. For example, in the context of web-based systems, load is often considered as the number of requests arriving at a web server within some period of time, while in event-driven architectures it is often the number of messages written to a messaging system. Such load types can be further broken down to distinguish, for example, between the number of concurrent users sending requests and the frequency of users sending requests. Other typical load types are the size per message or request or, in the case of request–response systems (e.g., databases), the size of responses.

Support for different SLOs The notion of SLOs in scalability definitions provides us with a measure to check, whether a system is able to handle a certain load intensity. Typical SLOs are, for example, that no more than a certain percentage of requests or messages may be processed with a certain latency (e.g., maximum allowed latency at the 95th percentile) or that no more than a certain amount of requests is discarded. The choice of such SLOs always depends on the application domain and should not be defined by the scalability metric. Additionally, a metric should support multiple SLOs, which all have to be fulfilled.

Support for different resource types Depending on the desired deployment, the resources that can be added to sustain increasing workloads may be of different types. According to the distinction between vertical and horizontal scalability, this means upgrading the computing capabilities of existing deployment components or increasing the replica count of deployment components (see Section 2.3). In cloud-native applications, this distinction can often be further broken down. For example, only a subset of a system's components can be scaled or individual components can be scaled differently.

6.2 Formal Definition of Scalability Metrics

According to the previously presented design rationale, we start defining our scalability metrics by formalizing the three scalability attributes load, resources, and SLOs. We define the load type as the set of possible load intensities for that type, denoted as L. For example, when studying scalability regarding the number of incoming messages per unit of time, L would simply be the set of natural numbers. Similarly, we define the resource type as the set of possible resources, denoted as R. While for horizontal scalability, R is typically the set of possible instance numbers (e.g., container or VM instances), for vertical scalability, R is the set of possible CPU or memory configurations (e.g., for a container or VM). We also require that there exists an ordering on both sets L and R. We define the set of all SLOs as S and denote an SLO $s \in S$ as Boolean-valued function

$$\text{slo}_s : L \times R \rightarrow \{\text{false}, \text{true}\}$$

with $\text{slo}_s(l, r) = \text{true}$ if a system deployed with r resource amounts does not violate SLO s when processing load intensity l.

Based on the previous characterization of scalability, we propose two functions as metrics for scalability.

6.2.1 Resource Demand Metric

The first function maps load intensities to the resources, which are at least required for handling these loads. We denote the metric as demand: $L \rightarrow R$, defined as:

$$\forall l \in L : \text{demand}(l) = \min\{r \in R \mid \forall s \in S : \text{slo}_s(l, r) = \text{true}\}$$

The *demand* metric shows how the resource demand evolves with increasing load intensities. Ideally, the resource demand increases linearly. However, in practice higher loads often require excessively more resources or cannot be handled at all, independently of the provisioned resources.

6.2.2 Load Capacity Metric

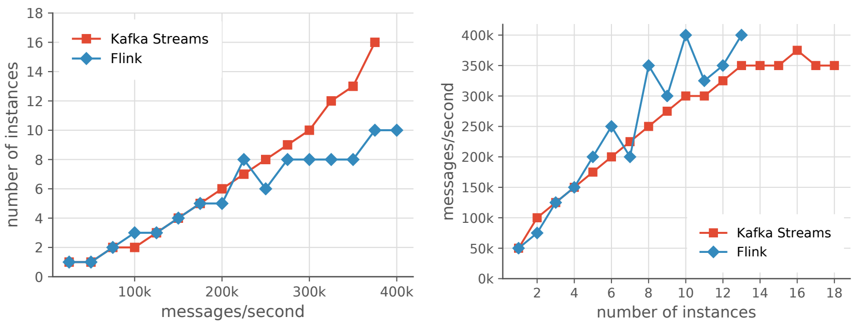
Our second metric maps provisioned resource amounts to the maximum load, these resources can handle. We denote this metric as capacity: $R \rightarrow L$, defined as:

$$\forall r \in R : \text{capacity}(r) = \max\{l \in L \mid \forall s \in S : \text{slo}_s(l, r) = \text{true}\}$$

Analogously to the *demand* metric, the *capacity* metric shows at which rate processing capabilities increase with increasing resources. It allows easily determining whether a system only scales up to a maximum resource amount (e.g., when a maximum degree of parallelism is reached). This is the case if increasing resources do not lead to higher load capacities.

6.3 Discussion

As required in Section 6.1, both our metrics do not make any assumption on the type of load, resource, or SLO. This implies that these metrics allow evaluating the same system with respect to different loads and resources of varying dimensions. Typical load dimensions are, for example, the number of concurrent users at a system, the number of parallel requests, or the size of requests. Also, multi-dimensional load and resource types (e.g., different VM configurations) could be evaluated, provided that there



(a) Resource demand metric **(b)** Load capacity metric

Figure 6.2. Visualizing the scalability of Kafka Streams and Flink with both our scalability metrics for an example benchmark execution [HH21a].

is an ordering on the load or resource values to be tested. For cloud configuration options, such an ordering usually exists in terms of the costs per configuration [BHI+17].

Fig. 6.2 shows plots of both scalability metrics for an example benchmark execution with Kafka Streams and Flink. It uses discrete sets for both the load intensities L and the provisioned resources R. L is defined as the number of sent messages per second with values between 25 000 and 400 000 and R is defined as the number of instances of the respective SUT with values between 1 and 18. A detailed description of the SLO used in this example can be found in Section 11.4.

6.3.1 Relation of Our Metrics

Intuitively, both our proposed scalability metrics are in many cases inverses of each other. More precisely, from their definitions, it follows that this is the case if and only if both functions are strictly monotonously increasing. In the example in Fig. 6.2, this is the case for Kafka Streams with loads between 100 000 and 300 000 messages per second and 2 to 10 instances.

As we usually need to evaluate scalability for discrete subsets of L and R (see Chapter 7), observing strictly monotonously functions is unlikely in practice. However, even if both metric functions are only monotonously increasing, they can still be considered inverses in a less strict sense as only the function's domain and image do not match (i.e, **ran** demand $\neq R$ and **ran** capacity $\neq L$, following Z notation [Spi89]). In Fig. 6.2, this is exemplified by Flink with loads between 25 000 and 200 000 messages/second and 1 to 5 instances.

However as Fig. 6.2 also shows, monotonicity is not always given in practice. For example, Flink's observed resource demand decreases from 225 000 to 250 000 messages/second and the capacity of 6, 8, and 10 instances is considerably higher than for 7, 9, and 11 instances, respectively. In such cases, both metrics provide different information as discussed in the following.

6.3.2 Comparison of Our Metrics

The major advantage of the demand metric is that it is more strictly aligned with the scalability definition in Section 2.2. It highlights that load is the independent variable. In practice, the load intensity, which a system has to sustain, emerges from external requirements (e.g., management, customers, users, etc.) and is out of control of a system's engineers. Hence, when benchmarking scalability from a client perspective, benchmarkers are interested in whether and how an SUT can handle increasing loads. On the other hand, evaluating scalability in function of resources as done with our capacity metric seems to be more common in the literature (see Section 6.4). It might thus appear more familiar for benchmarkers.

Besides these rather subjective differences, there are also differences concerning the informational value of the metrics. This particularly concerns cases, where a system only scales up to a certain load (i.e., SLOs cannot be fulfilled for higher loads, independent of the provisioned resources). In such cases, the demand metric is only partially defined, meaning that for loads larger than the scalability limit, no resource demand is provided. Fig. 6.2 exemplifies this for processing 400 000 messages/second with Kafka Streams. For the capacity metric, such a limit becomes more apparent as it is the maximum of the function and higher resource values result in a drop in capacity or at least no further increase (see Kafka Streams' capacity for more than 16 instances).

Moreover, the capacity metric is able to report if certain resource configurations are more efficient than others. For example, we observed in several experiments that some uneven instance numbers of Flink yield lower throughput than similar even numbers as also shown in Fig. 6.2. While this clearly appears by a zigzag pattern in the capacity function, the demand function is rather linear and hides this in the function steps. On the other hand, the demand metric reveals load intensities that can be processed much more efficiently or less efficiently. In Fig. 6.2, this is the case for 225 000 messages/second with Flink. For the situation of distributed stream processing systems, however, it is less likely that there is a fundamental reason for specific load intensities to be better processable.

6.4 Related Work

Both our proposed metrics are functions. Lehrig et al. [LSB+18] point out that scalability should be quantified as a function since capacity (or resource demand) does not increase at a constant rate when adding resources (or increasing the load).

As discussed in Section 2.1, traditional parallel and distributed systems research often describes scalability as a function mapping processors or computing nodes to the time they require to compute some problem. In particular in high-performance computing (HPC), the distinction between strong and weak scalability is common. Such metrics are not suitable for cloud-native applications. Cloud-native applications are usually not designed for solving a single, compute-intensive problem, but are subjected to a permanent load, such as requests from their users. Hence, the goal of scalability evaluations is to assess whether a cloud-native applications is still functioning if the load increases. This is what Bondi [Bon00] defines as load scalability.

Also originating from traditional parallel and distributed systems research, the Universal Scalability Law [Gun07; GPT15] is a general performance model for system scalability. Similar to our *capacity* metric, it describes scalability as capacity in function of processors. The Universal Scalability Law is based on the assumption that scalability of arbitrary systems can be described using a non-linear rational function with two system-specific coefficients, representing contention and coherency. Quantifying scalability by two coefficients has the significant advantage of allowing easy ranking different SUTs. However, to the best of our knowledge, there is no empirical study so far that evaluates how well these coefficients can be derived from empirical measurements when considering capacity as discrete values, as obtained with our proposed measurement method. In Section 16.2 of this thesis, we conduct a prototypical evaluation using the results of our capacity metric to fit an USL model.

In their seminal works on cloud benchmarking, Binnig et al. [BKK+09] and Kossmann et al. [KKL10] measure scalability of different cloud services by evaluating the number of successful requests while increasing the number of parallel requests. Similar to our *demand* metric, they thus consider the varying load as input variable. This allows detecting if an increasing load cannot be handled anymore at some point. In contrast to this work, the authors focus on cloud services, which are automatically scaled by the cloud provider. With such services, customers are not directly charged for the underlying hardware resources, but instead based on application-level metrics (such as requests per hour). For orchestrated cloud-native applications as studied in this thesis, resources are manually scaled to achieve the desired SLOs. Hence, we assume that a metric for such type of systems should quantify the resource demand.

Along with establishing precise definitions of scalability in cloud computing, Lehrig et al. [LEB15] and Becker et al. [BLB15] use systematic methods to derive scalability metrics. Lehrig et al. [LEB15] conduct a systematic literature review and found only one scalability metric evaluated in a practical setting at the time of the study [THS11]. This metric requires scaling to be quick, which contradicts most scalability definitions [LEB15]. Becker et al. [BLB15] use the goal question metric (GQM) method to derive two scalability metrics. The first metric "scalability range" describes the maximum load an SUT can handle without violating its SLOs. This metric can also be derived directly from both our scalability metrics. The second metric is called "scalability speed" and describes whether an SUT can achieve its SLOs if the load increases by a certain rate (load per unit of time). As highlighted in Section 2.4, using time for describing scalability is uncommon. Both metrics from Becker et al. [BLB15] do not consider the resource amounts needed to achieved its SLOs.

Herbst et al. [HKW$^+$15] and Brataas et al. [BHI$^+$17] use metrics, which are very similar to the ones presented in this chapter. Both studies build upon the same scalability definition as adopted in this thesis, thus relating load, resources, and SLOs. As part of their elasticity benchmarking method, Herbst et al. [HKW$^+$15] determine the resource demand of a system, which corresponds to our *demand* metric. In contrast to this thesis, the authors do not further discuss or formalize the metric. Brataas et al. [BHI$^+$17] evaluate scalability as function matching our *capacity* metric, although formalized differently. In our previous work [HH21a], we compare the *demand* and the *capacity* metric for the special case of stream processing systems.

Recently, Avritzer et al. [AFJ$^+$20] presented the *domain-based metric* to assess the scalability of microservice deployment options. It aims at measuring scalability as a single value, quantifying how increasing workload situations can be handled without violating SLOs. In contrast to our metric, the *domain-based metric* does not allow for resources to be added in order to satisfy SLO requirements. Instead, different resource amounts are considered to be separate deployment options.

Chapter 7

Scalability Measurement Method

This chapter presents our Theodolite measurement method to assess scalability of cloud-native applications according to the two metrics presented in the previous chapter (see Fig. 7.1). It addresses research question RQ 1.2 of this thesis. Our scalability measurement method approximates our scalability metrics by running isolated experiments for different combinations of generated load intensity and provisioned resources. It provides several configuration options to balance the overall runtime and statistically grounding of its results.

This chapter builds upon our work previously published in the *Empirical Software Engineering* journal [HH22a]. It is organized as follows. After describing its design rationale in Section 7.1, we outline the fundamental approach of our measurement method in Section 7.2. It consists of two main components: the execution of experiments to evaluate whether SLOs are met is presented in Section 7.3 and search strategies, which deter-

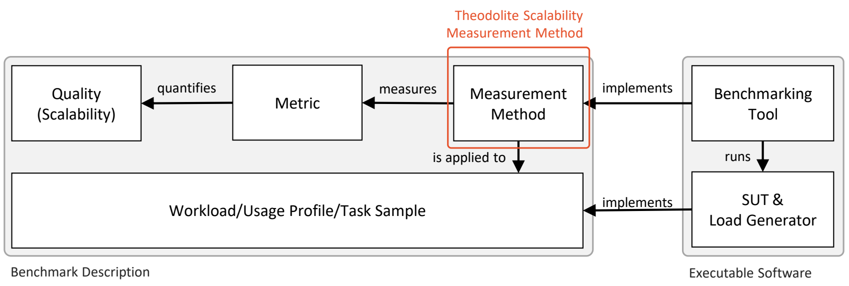


Figure 7.1. Chapter 7 of this thesis presents our Theodolite measurement method for benchmarking scalability of cloud-native applications as highlighted by the red box.

mine the SLO experiments to be executed, are presented in Section 7.4. Afterward, we discuss how the requirements for statistical grounding and time-efficient execution relate to each other in Section 7.5. We conclude this chapter by discussing our method with respect to related work in Section 7.6.

7.1 Design Rationale

From our mapping of benchmark quality attributes to benchmark components in Chapter 5, we can derive that our measurement method should provide statistically sound results (for repeatability and verifiability) and time-efficient execution (for usability). Both requirements conflict with each other as detailed in the following.

Robust statistical grounding As described in Section 3.6, performance experiments in the cloud should be repeated and the confidence in the final results should be quantified. Our scalability metrics are based on the assessment of SLOs, which requires conducting performance experiments. This means that experiments in scalability evaluations should be executed for a sufficient amount of time as well as sufficiently often repeated.

Time-efficient execution Increasing the statistical grounding of performance experiments as described above leads to longer execution times. Hence, the requirement for reproducibility conflicts with the requirement for usability and, thus, verifiability as with increasing execution time also costs increase. For a usable measurement method, we therefore require finding a balance between statistically grounded results and a time-efficient execution.

7.2 Fundamental Approach

Our scalability measurement method approximates our scalability metrics by running experiments with finite subsets of the considered load and resource types, $L' \subseteq L$ and $R' \subseteq R$. The sizes of the chosen subsets L' and

R' determine the resolution of the metrics, but also the overall runtime of the method. The basic idea of our measurement method is to run isolated experiments for various load $l \in L'$ and resource $r \in R'$ combinations, which serve to evaluate whether specified SLOs are met. We decided to run these experiments in isolation as scalability does not take temporal aspects into account (e.g., how fast can an SUT react to a changing load, cf. Section 2.4). Measuring the throughput for a fixed, high load or increasing the load at runtime might cause wrong results [HH21a].

In the following, we describe the two main components of our proposed measurement method: the execution of experiments to evaluate whether SLOs are met and search strategies, which determine the SLO experiments to be executed.

7.3 SLO Experiments

Formally, an SLO experiment determines whether for a given set of SLOs S, an SUT deployed with $r \in R'$ resources can handle a load $l \in L'$ in a sense that each $\forall s \in S : \text{slo}_s(l, r) = \text{true}$ (i.e., each SLO is fulfilled).

Our measurement method deploys the SUT with r resources and generates the constant load l over a configurable period of time. During this time, the SUT is monitored and data is collected, which is relevant to evaluate the SLOs. For example, for an SLO that sets a limit on the maximal latency of processed messages, monitoring would continuously measure the processing latency. The duration for which SLO experiments are executed should be chosen such that enough measuring data is available to draw statistically rigorous conclusions. On the other hand, this duration should not be unnecessarily long to achieve the required time-efficient execution and, thus, increase usability. To meet the requirement for statistically grounded results, measured values of an initial time period are discarded (warm-up period). Measurements during this time usually deviate from those of the further execution as, for example, optimizations are performed after start-up. Another measure to increase statistical rigor is to repeat SLO experiments with the same load and resource combination. To finally compute $\text{slo}_s(l, r)$, the monitored data points of all repetitions are aggregated in an SLO-specific way.

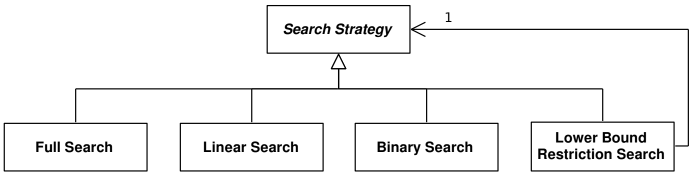


Figure 7.2. UML class diagram of different search strategies [HH22a].

7.4 Search Strategies

Our proposed scalability measurement method is configurable by a search strategy, which determines the SLO experiments that will be performed to accurately approximate the scalability metrics. In the case of the *demand* metric, the goal is to find the minimal required resources for each load intensity $l \in L'$. For the *capacity* metric, the maximal processible load intensity for each resource configuration $r \in R'$ should be found.

Fig. 7.2 gives an overview of selected search strategies, which we describe in the following for the case of the *demand* metric. Nonetheless, these strategies can easily be transferred to the *capacity* metric. Apart from these examples, also more complex strategies are conceivable (see, for example, the work of Ehrenstein [Ehr22]). Fig. 7.3 provides an illustrative example of our measurement method for each strategy. A colored cell corresponds to an SLO experiment for a certain load intensity and resource configuration, which is executed by the respective search strategy. Green cells represent that the corresponding SLO experiment determined that the tested resources are sufficient to handle the tested load. Red cells represent that the resources are not sufficient. Framed cells indicate the lowest sufficient resources per load intensity. The resulting *demand* function is plotted in Fig. 7.3a.

Full search The full search strategy (see Fig. 7.3a) performs SLO experiments for each combination of resource configuration and load intensity. Its advantage is that it allows for extensive evaluation after the benchmark has been executed. This also includes that based on the same SLO experi-

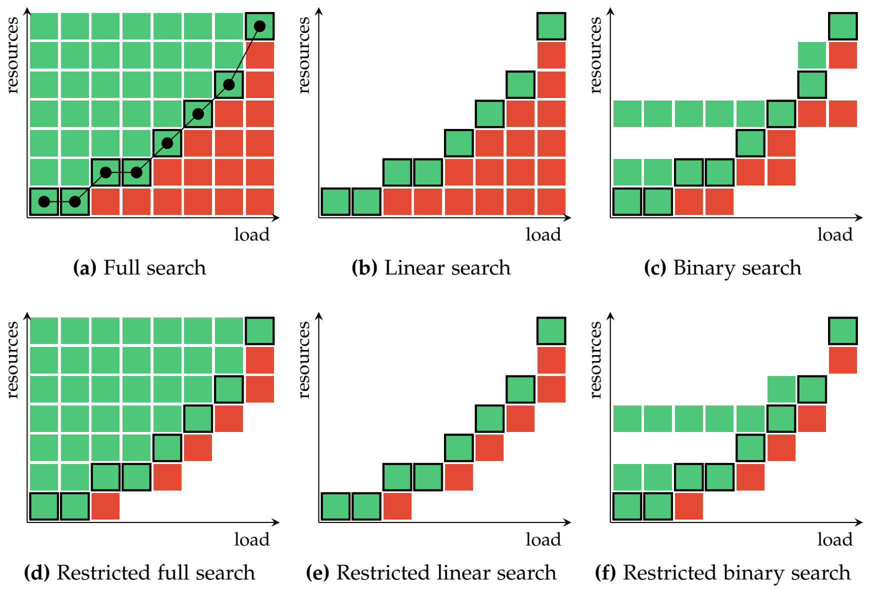


Figure 7.3. Comparison of selected search strategies [HH22a].

ments, both the *demand* and the *capacity* metric can be evaluated. However, this comes at the cost of significantly longer execution times.

Linear search The linear search strategy (see Fig. 7.3b) reduces the overall execution time by not running SLO experiments whose results are not required by the metric. That is, as soon as a sufficient resource configuration for a certain load intensity is found, no further resource configurations are tested for that load.

Binary search The binary search strategy (see Fig. 7.3c) adopts the well-known algorithm for sorted arrays. That is, the strategy starts by performing the SLO experiments for the middle resource configuration. Depending on whether this experiment was successful or not, it then continues searching in the lower or upper half, respectively. The binary search is particularly advantageous if the search space is very large (i.e, larger than in Fig. 7.3). However it is based on the assumption that with additional resources for the same load, performance does not substantially decrease. More formally, this strategy assumes:

$$\forall l \in L', r, r' \in R' : r' > r \wedge \text{slo}_s(l, r) = \text{true} \Rightarrow \text{slo}_s(l, r') = \text{true}$$

We evaluate this assumption for the special case of event-driven microservices in Section 12.4 and show that it does not hold in all cases.

Lower bound restriction The *lower bound restriction* (see Figs. 7.3d–f) is an example for a search strategy that uses the results of already performed SLO experiments to narrow the search space. It starts searching (with another strategy) beginning from the minimal required resources of all lower load intensities. Note that when combined with the binary search strategy, the lower bound restriction may also cause different experiments to be performed (see upper right of Fig. 7.3f). The lower bound restriction is based on the assumption that with increasing load intensity, the resource demand never decreases. More formally, this strategy assumes:

$$\forall l, l' \in L' : l' > l \Rightarrow \text{demand}(l') \geqslant \text{demand}(l)$$

In Section 12.4, we show that for the special case of event-driven microservices, we are safe to make this assumption.

7.5 Balancing Statistical Grounding and Time-Efficiency

The runtime of a scalability benchmark execution depends on the number of evaluated resource amounts $|R'|$, the amount of evaluated load intensities $|L'|$, the duration of an SLO experiment τ_e as well as the associated warm-up period τ_w, the number of SLO experiment repetitions ρ, and the applied search strategy δ. Likewise, these values also control the statistical grounding of the results. Table 7.1 summarizes the effect of each configuration option on statistical grounding, while the following formulas show the runtime Φ for both the *demand* and the *capacity* metric:

Table 7.1. Effect of configuration options on statistical grounding of results.

Symbol	Configuration option	Description
$\|R'\|$	Resource amounts	Higher amounts cause more fine-grained approximation of the scalability metric function.
$\|L'\|$	Load intensities	Higher amounts cause more fine-grained approximation of the scalability metric function.
τ_e	Experiment duration	Longer durations cause more stable results.
τ_w	Warm-up period duration	Higher values increase the certainty that early measurements that are not representative for the system under normal operation are excluded.
ρ	Repetitions	More repetitions rule out the effect of outliers.
δ	Search strategy	Strategies may be based on assumptions that do not always hold.
ϕ_δ	Runtime of strategy δ	Depending on the metric, the runtime of δ either depends on $\|R'\|$ or on $\|L'\|$.

$$\Phi_{\text{demand}} = |L'| \times \phi_\delta(|R'|) \times \rho \times (\tau_e + \tau_w)$$
$$\Phi_{\text{capacity}} = |R'| \times \phi_\delta(|L'|) \times \rho \times (\tau_e + \tau_w)$$

7.6 Related Work

While several scalability evaluations describe scalability as functions of resources, many of them do not conduct isolated experiments for different load intensities [BBT+15; AA19; KYA17; NNG19; KRK+18]. Instead, they continuously increase the load on a system and measure when SLOs cannot be fulfilled anymore. In our previous work, we highlighted limitations of this method for the case of stream processing [HH21a]. Sometimes also the provisioned resources are auto-scaled in the background [BBT+15; AA19], which serves a different purpose than our scalability benchmarking method. Different methods for scalability benchmarking of database systems in the cloud are discussed by Kuhlenkamp et al. [KKR14], which, however, do not include running isolated experiments for different load resource combinations.

Scalability evaluations similar to our proposed measurement method can be found for the case of Infrastructure-as-a-service (IaaS) clouds [HKW+15; BHI+17; CMS17]. As Herbst et al. [HKW+15] and Brataas et al. [BHI+17] use scalability metrics similar to ours, they also pursue similar ideas for bounding the execution times of their experiments. Herbst et al. [HKW+15] apply a binary search strategy to find the maximal load processing capacity for a set of resources. In line with our linear search strategy, Brataas et al. [BHI+17] stop testing higher load intensities once they detect that a load intensity cannot be handled anymore by a certain resource level. However, both papers discusses this topic only briefly and do not provide a systematic measurement method. As Brataas et al. [BHI+17] analyze scalability only with respect to increasing resources, they do not apply strategies such as our lower bound restriction. Although without explicitly stating scalability metrics and measurement methods, Cunha et al. [CMS17] employ a similar approach for evaluating horizontal and vertical scalability for IaaS cloud environments. They conduct three isolated experiments for different resource amounts and load intensities and evaluate whether service level objectives are achieved. The authors do not use any techniques to reduce the search space.

Related work quantifying the variability of short running performance experiments in the cloud can, for example, be found by Iosup et al. [IYE11], Leitner and Cito [LC16], Abedi and Brecht [AB17], Maricq et al. [MDJ+18], or Laaber et al. [LSL19]. He et al. [HMS+19; HLL+21] and Bulej et al. [BHT+20] propose methods for reducing the amount of experiment repetitions while preserving a high measurement accuracy. These methods differ from ours in that they aim to accurately measure the performance of a system, while for benchmarking scalability, we only need to accurately assess whether a system fulfills specified SLOs. (See our method's evaluations in Chapter 12 for further details.) Combining the methods of He et al. [HMS+19; HLL+21] with ours is basically possible, but would lead to benchmark execution durations of several weeks, which we consider impractical.

For measuring their *domain-based metric*, Avritzer et al. [AFJ+20] proposed a novel approach for deriving SLOs based on a low workload. Additionally, they use operational profiles for representative workloads, which may be extracted from monitoring data.

Chapter 8

Benchmarking Tool Architecture

In this chapter, we addresses research question RQ 1.3 of this thesis and propose the Theodolite architecture for a scalability benchmarking tool for cloud-native applications. It implements our proposed measurement method and, thus, measures our proposed scalability metrics (see Fig. 8.1).

This chapter builds upon our work previously published in the *Empirical Software Engineering* journal [HH22a]. It starts by describing our design rationale in Section 8.1. Section 8.2 outlines our proposed benchmarking process, which distinguishes the definition of benchmarks and their execution. Section 8.3 presents a corresponding meta model for defining these benchmarks and executions. Afterward, Section 8.4 describes how benchmarks and executions defined with this model can be executed in cloud-native environments. Section 8.5 presents our Theodolite benchmarking framework, which implements the architecture proposed in this chapter. Finally, Section 8.6 discusses related work.

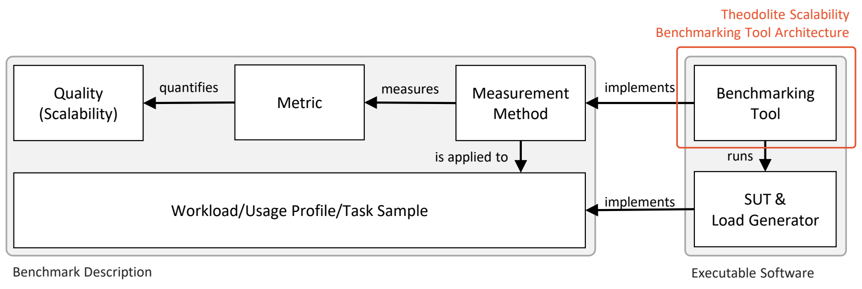


Figure 8.1. Chapter 8 of this thesis presents our Theodolite architecture for a cloud-native scalability benchmarking tool as highlighted by the red box.

8.1 Design Rationale

According to our mapping of benchmark quality attributes to benchmark components in Chapter 5, we found that a benchmarking tool should primarily be designed for usability. According to von Kistowski et al. [vKAH+15], good benchmarks avoid "roadblocks" for users to run the benchmark. Simplifying benchmark execution also supports repeatability of benchmarking studies, which is a prerequisite for reproducibility. As we aim for engineering an architecture for a benchmarking tool that supports executing different benchmarks, we extend the requirement from von Kistowski et al. [vKAH+15]: not only executing benchmarks should be simplified, but also defining them.

To provide high usability, we propose that a scalability benchmarking tool should be built around two design principles. First, it should operate on declarative descriptions of benchmarks and their executions and, second, it should integrate with the cloud-native ecosystem. We detail advantages of both principles in the following.

8.1.1 Declarative Benchmarks and Executions

As we describe in Section 8.2, we suggest distinguishing between benchmarks and their executions. Defining both in declarative files provides the following advantages.

Distributing, sharing, and archiving Declarative files that define the benchmark can easily be understood without requiring insights into, for example, the SUT implementation or metrics calculation. Additionally, declaring the benchmark execution in a file serves as documentation of the experimental setup. The corresponding files can be stored in version control systems or research data repositories providing trackability, auditability, and reusability.

Automating benchmark execution Operating a distributed software system in the cloud is a complex task, which explains the rise of powerful orchestration tools such as Kubernetes. Typical situations that have to be handled are, for example, unpredictable network connections, deviations in the underlying hardware or software infrastructure as well as complex requirements on the order of starting many interacting components. Orchestration tools address this by providing declarative APIs, which are used to describe the desired state of the system. They continuously compare the actual state to the desired state and perform the necessary reconfigurations. Such situations must also be accounted for when running benchmarks in orchestrated cloud platforms, where experiments should be executed for several hours without user intervention. Thus, benchmarkers should describe their desired experiments in a declarative way, while the benchmarking tool handles the actual execution.

Benchmarking different configuration options Benchmarks are often used to compare not only different systems or frameworks but also different configurations or deployment options. Allowing for such configurations in a declarative way without a new installation or even re-building the benchmark's implementation enhances usability.

8.1.2 Integration with the Cloud-Native Ecosystem

We propose to utilize existing tools and patterns of the cloud-native ecosystem to a great extent. This provides the following advantages regarding the deployment, operation, and observation of benchmarks.

Running existing benchmarks, SUT, and load generators There exists a variety of reference implementations and benchmarks for different types of cloud-native applications. Although often originally designed for evaluating other qualities, such applications can still serve as task samples for scalability evaluations. Using a standardized way of describing an SUT deployment (e.g., by Kubernetes resources) simplifies designing new scalability benchmarks for those applications. The same applies also for conducting scalability evaluations of real-world systems (i.e., in a non-competitive manner). Likewise, relying on standardized deployment descriptions allows using existing load generator tools with little effort. That means the benchmarking tool does not have to provide a load generator, but instead benchmark designers can use the one, which is best suited to the benchmark.

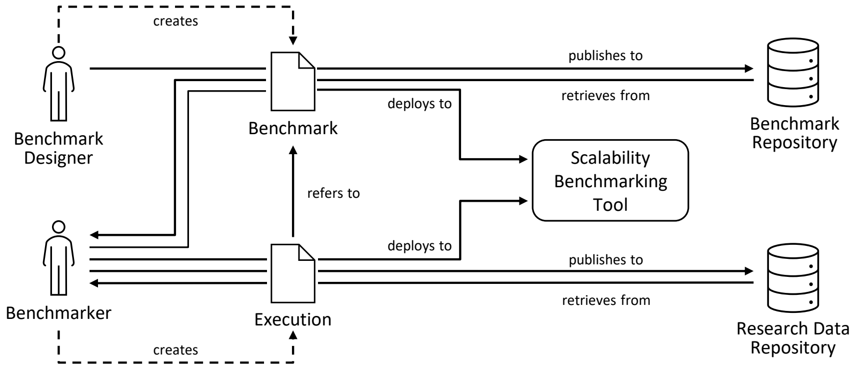


Figure 8.2. Context diagram showing how actors interact with our proposed benchmarking tool architecture [HH22a].

Utilizing existing performance metrics Cloud-native applications or corresponding frameworks and middlewares often provide performance and status metrics via established APIs and data formats (e.g., Prometheus). Using these metrics in scalability benchmarks does not only simplify defining benchmarks, it also fosters using similar metrics to those that are used in production.

Out-of-the-box tooling support The cloud-native ecosystem provides a plethora of tools for managing and observing various aspects of cloud-native applications. As these tools are usually developed in large community projects and often supported by large cloud vendors, they can generally be considered to be of high quality. Integrating a benchmarking tool for cloud-native applications with such tools also helps to provide a rich user experience.

8.2 Overview of the Benchmarking Process

Fig. 8.2 gives an overview of our proposed scalability benchmarking process. In general, we can observe two actors involved in benchmarking:

Benchmark designer Benchmark designers are, for example, researchers, engineers, or standardization committees, which are experts regarding a specific type of application or software service. They are able to construct representative and relevant task samples or workloads for that type of software systems. Moreover, they know about relevant load intensity types, resource types, and SLOs, regarding which scalability should be evaluated. Benchmark designers bundle all of this in Benchmarks. Benchmarks can be published as supplemental material to research papers, but ideally, they are versioned and maintained in public repositories (e.g., at GitHub). Benchmarks are stateless as they can be executed arbitrarily often.

Benchmarker Benchmarkers intend to compare and rank different existing SUTs, evaluate new methods or tools against a defined standard, or repeat previous experiments. A detailed description of the benchmarker actor can be found by Kounev et al. [KLvK20]. Benchmarkers retrieve existing Benchmarks from their public repositories and execute them in the desired cloud environment. For this purpose, they describe the experimental setup for running a single Benchmark in a so-called Execution. Benchmarkers deploy both the Execution and the corresponding Benchmark to the benchmarking tool, which applies our proposed scalability measurement method. Executions are then assigned a state, which is typically something like *Pending*, *Running*, *Finished*, or *Failed* if an error occurred. Executions can be shared, for example, as part of a research study that benchmarks the scalability of different SUTs. The same or other benchmarkers can then again retrieve and copy Executions, for example, to replicate benchmarking studies.

8.3 Benchmarking Meta Model

Based on the previous distinction between benchmarks and their executions, we propose a meta model for defining them in a declarative way. Fig. 8.3 visualizes the central elements of our meta model and their relations as UML class diagram. In the following, we describe this meta model starting from the central entities Benchmark and Execution.

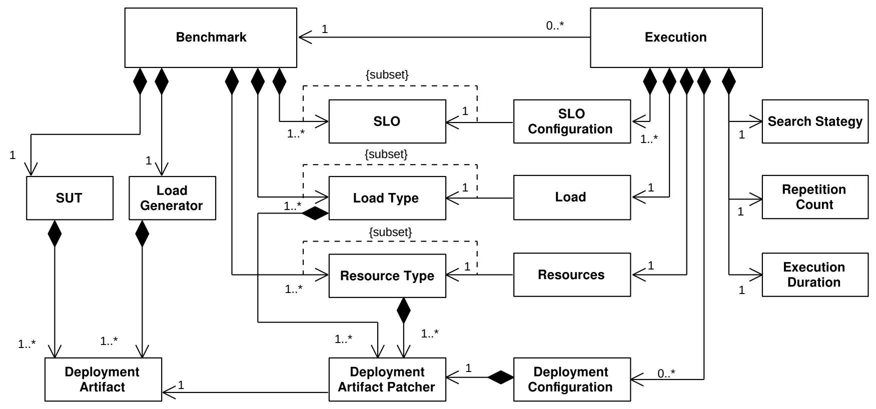


Figure 8.3. UML class diagram of our benchmarking meta model [HH22a].

Benchmark A Benchmark is a static representation of a SUT and an associated Load Generator, where SUT and Load Generator are represented as sets of Deployment Artifacts. Such Deployment Artifacts are, for example, objects of Kubernetes resources such as Pods, Services, or ConfigMaps.[1]

According to our scalability metrics, benchmarks support different SLOs, Load Types, and Resource Types. An SLO represents the computations on gathered monitoring data, which are necessary to check an SLO. This may include the queries to the monitoring system, statistical calculation on the returned data, thresholds, or warm-up durations. Load Types and Resource Types are both represented as sets of Deployment Artifact Patchers. These patchers are associated with a Deployment Artifact and modify it in a certain way when running an SLO experiment. For example, a Resource Types for scaling the number of replicas can use a Deployment Artifact Patcher that adjusts the number of replicas of a SUT's Deployment Artifact.

Existing cloud-native benchmarks for other qualities can be utilized to define scalability benchmarks by aggregating their deployment artifacts and specifying load types, resource types, and SLOs. Benchmarks do not have a life cycle and can be executed arbitrarily often by Executions.

[1]We decided to use the more general term *Deployment Artifact* to avoid confusion with the term resources from our scalability definitions.

Execution An Execution represents a one-time execution of a benchmark with a specific configuration. It evaluates a subset of the SLOs provided by the Benchmark, which can additionally be configured by an SLO Configuration. SLO Configurations adjust SLO parameters such as warm-up duration or thresholds. As specified by our measurement method, scalability is benchmarked for a finite set of load intensities of a certain load type and a finite set of resource amounts of a certain resource type. In our meta model, these sets are represented as Loads and Resources. Since the Benchmark declares its supported Load Types and Resource Types, the specified Loads and Resources refer to the corresponding Load Type and Resource Type, respectively (thus the *subset* constraints).

Furthermore, an Execution can configure the SUT and the Load Generator by Deployment Configurations. Such Deployment Configurations consist of a Deployment Artifact Patcher and a fixed value, which the corresponding deployment is patched with. This allows, for example, to evaluate different configurations of the same SUT, for example, via environment variables. An Execution supports the configuration options of our measurement method discussed in Chapter 7, namely a Search Strategy as well as a Repetition Count and an Experiment Duration for the SLO experiments. Warm-up period durations are SLO-specific and, thus, specified as part of SLO Configurations.

In contrast to Benchmarks, Executions have a life cycle. They can be planned, executed, or aborted. Each execution of a benchmark is represented by an individual entity. This supports repeatability as executions can be archived and shared.

8.4 Operator-Based Cloud-Native Architecture

We propose a benchmarking tool architecture based on the Operator pattern (see Section 4.1.3). Fig. 8.4 shows our proposed architecture for a cloud-native scalability benchmarking tool. We envisage Custom Resource Definitions (CRDs) for the Benchmark and Execution entities of our benchmarking meta model such that benchmarkers can deploy Benchmarks and Executions to the orchestration API. Whenever new Executions are created, the scalability benchmarking operator is notified and, if no other bench-

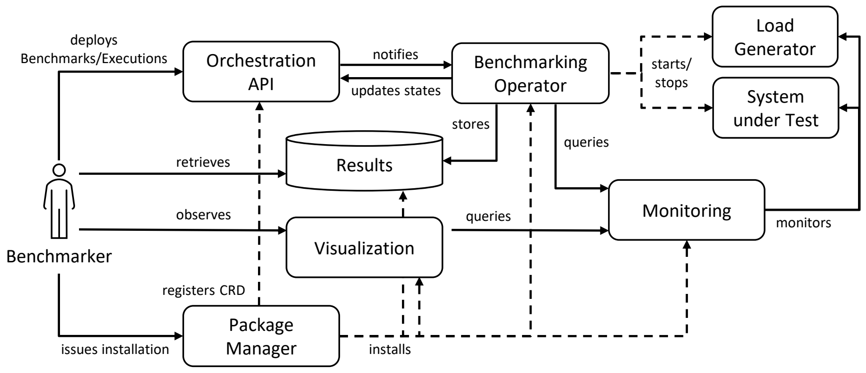


Figure 8.4. Proposed benchmarking tool architecture based on the operator pattern [HH22a].

mark is currently executed, it starts executing a benchmark according to the specified Execution. This means, it alters the Deployment Artifacts of the SUT and the load generator according the provided Deployment Configuration, applies the configured search strategy to decide which SLO experiments should be performed, and also adjusts the Deployment Artifacts according to the selected load and resources. It then deploys all (potentially adjusted) Deployment Artifacts for the specified duration and repeats this procedure multiple times according to the defined Execution.

During this time, a monitoring system [RSE+19] collects performance data of the SUT, which is then used by the operator to evaluate the specified SLOs. The operator stores all (raw) results persistently to allow for offline analysis, archiving, and sharing. Additionally, a cloud-native visualization tool might be used to let benchmarkers observe the execution of benchmarks (see Section 3.3).

This architecture causes a significant effort for the installation of the entire benchmarking infrastructure. To implement the requirement for a simple implementation, we propose to employ a cloud-native package management tool, which installs the operator, the CRDs as well as dependent systems such as the monitoring and the visualization tool.

8.5 The Theodolite Benchmarking Framework

Theodolite[2] is our framework for benchmarking the scalability of cloud-native applications. It implements the architecture proposed in this section and, thus, the benchmarking method proposed in this thesis. Theodolite is free and open-source research software [HCH+20], distributed via GitHub.[3] We provide documentation and starting guides via our tool's website and in two demonstration papers [HH22b; HWH23].

Theodolite has been peer reviewed and successfully evaluated by multiple independent experts of the SPEC Research Group, a sub-organization within the Standard Performance Evaluation Corporation (SPEC). Review criteria are important quality factors such as maturity, availability, and usability to ensure high quality and relevance to the community. Theodolite is now listed in the SPEC RG repository of peer-reviewed tools for quantitative system evaluation and analysis.[4]

Referring to our proposed architecture (see Fig. 8.4), we employ the following technologies in our Theodolite framework: Kubernetes as orchestration tool, Prometheus for monitoring the SUT, Grafana [Gra22] for the visualization, and Helm [Hel22] as package manager. These technologies are generally understood as part of the cloud-native landscape.[5] Theodolite stores all raw measurements used for the SLO assessment in CSV files. These can be used for an in-depth analysis with provided Jupyter notebooks [KRP+16; GP21].

8.6 Related Work

Many studies, which benchmark scalability in the cloud, do not provide a benchmarking tool or a corresponding architecture [AA19; KRK+18]. General architectures for cloud benchmarking tools are presented, for example, by Bermbach et al. [BWT17] and Iosup et al. [IPE14]. Our architecture builds upon such architectures but is particularly suited for cloud-native

[2] https://www.theodolite.rocks

[3] https://github.com/cau-se/theodolite

[4] https://research.spec.org/tools/overview/

[5] https://landscape.cncf.io

environments by relying on established cloud-native tools and patterns. Additionally, we allow for defining benchmarks declaratively.

Declarative benchmark description is also suggested by Cunha et al. [CMS17], but without distinguishing between benchmarks and executions. We expect the separation of benchmark and its execution to particularly foster reproducibility. More generally, advantages of declarative benchmarking and performance are highlighted by Walter et al. [WvHK+16] and Ferme and Pautasso [FP18]. Similar to our approach, Bermbach et al. [BKD+17] present a benchmarking framework providing a generic interface to execute arbitrary application benchmarks for cloud storage services.

Recently, Avritzer et al. [ACJ+21] presented an architecture for measuring their *domain-based metric* [AFJ+20]. In contrast to this study, the authors do not aim for executing independently provided benchmarks, but instead, perform scalability tests as part of a software's quality assurance. For this reason, and because scalability is studied with respect to a different metric, they do not provide a meta model, which is comparable to the one we presented in this study. Similar to our proposed architecture, the authors utilize cloud-native technologies for monitoring and visualization. However, they do not integrate their architecture in the underlying orchestration tool (Docker swarm in their case) as we do by adopting the operator pattern.

In a previous publication [HH21e], we presented a first prototypical architecture for an initial version of our scalability benchmarking method. The presented architecture did already rely on cloud-native technologies but was tailored to run specific benchmarks for stream processing frameworks. We later extend the experiment control component of that architecture substantially to increase usability [HH22a]. Specifically, we proposed to apply the operator pattern along with a flexible meta model for defining benchmarks and their executions in a declarative manner.

To the best of our knowledge, we presented the first scientific approach for applying the operator pattern to benchmark cloud-native applications [HWH21; HH22a]. Similar to ours, Merenstein et al. [MTA+20; MTA+21] proposed an architecture based on the operator pattern for benchmarking cloud-native storage infrastructure. In line with our argumentation, the authors highlight that such an architecture can improve usability, which is particularly important in complex cloud-native environments. Nikolaidis et al. [NCM+21b] present *Frisbee*, a Kubernetes operator for declaratively testing cloud-native applications. While first being promoted as benchmarking tool for comparing the recovery behavior of highly available systems [NCM+21a], it now aims for arbitrary end-to-end testing of containerized applications. Most similar to our proposed architecture is *AutoDECK*, an operator-based performance evaluation tool recently presented by IBM Research [CCT+22]. It integrates different types of performance benchmarks. Like our Theodolite tool, also Frisbee and AutoDECK provide interoperability with Prometheus and Grafana. Additionally, some benchmark operators emerged in the cloud-native community for benchmarking the performance of Kubernetes installations.[6] In contrast to our architecture, all of these operators do not distinguish between CRDs for benchmarks and their executions, which we expect to be a key feature for enabling reproducibility.

[6] `https://github.com/xridge/kubestone`, `https://github.com/cloud-bulldozer/benchmark-operator`

Part III

The Theodolite Benchmarks for Event-Driven Microservices

Chapter 9

Research Design and Methods

In the chapters of this thesis' third part, we address our second guiding goal, namely designing scalability benchmarks for event-driven microservices. We present our Theodolite benchmarks, which allow software engineers to evaluate and compare different stream processing frameworks, different configurations of these frameworks, alternative algorithms, and different deployment options regarding their scalability. In this chapter, we introduce our employed research design and methods to address this goal.

We build our proposed benchmarks on top of the scalability benchmarking method presented in Part II. With respect to the structuring of benchmarks into components as described in Section 3.3, this means we have to design the task sample components, whereas we can use the scalability metric and measurement presented in Chapter 6 and Chapter 7, respectively (see Fig. 9.1). We complement our task samples with

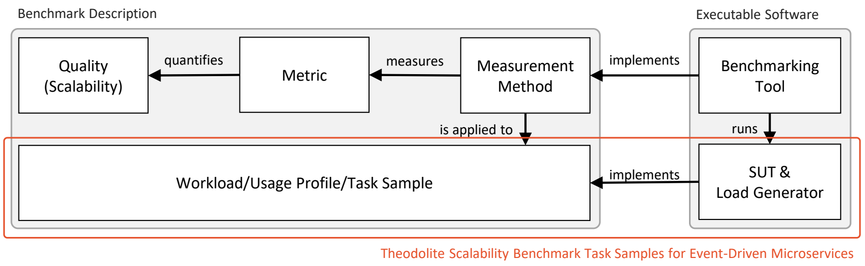


Figure 9.1. Part III of this thesis covers designing task samples for scalability benchmarks for event-driven microservices along with corresponding SUT and load generator implementations as highlighted by the red box.

specific implementations for different SUTs and load generators. These are bundled as specific benchmarks according to the meta model of our benchmarking tool architecture described in Chapter 8.

9.1 Design Rationale

As discussed in Chapter 5, the task sample component of a benchmark primarily has to address relevance and fairness. In the following, we propose three principles for designing task samples, which support relevance and fairness.

Based on real requirements Benchmarks designed according to real-world requirements are more likely to behave like the corresponding real-world systems. Hence, they can also be considered more relevant.

Use case oriented We propose to design scalability benchmarks according to typical use cases for stream processing within microservices as opposed to benchmarks modeled around capabilities of stream processing frameworks. This supports fairness and relevance of the benchmarks.

Described as abstract dataflow architectures As we introduced in Section 4.4, most modern stream processing frameworks employ similar dataflow models. However, specific implementations and default configurations differ significantly in some cases. Task samples should be described in abstract dataflow architectures not tailored to specific frameworks to ensure a fair comparison between frameworks.

9.2 Research Design Overview

We design our Theodolite benchmarks for use cases of event-driven microservices in the domain of Industrial Internet of Things (IIoT) analytics. In particular, we focus on the special case of analyzing electrical power consumption data in manufacturing enterprises. However, we expect that

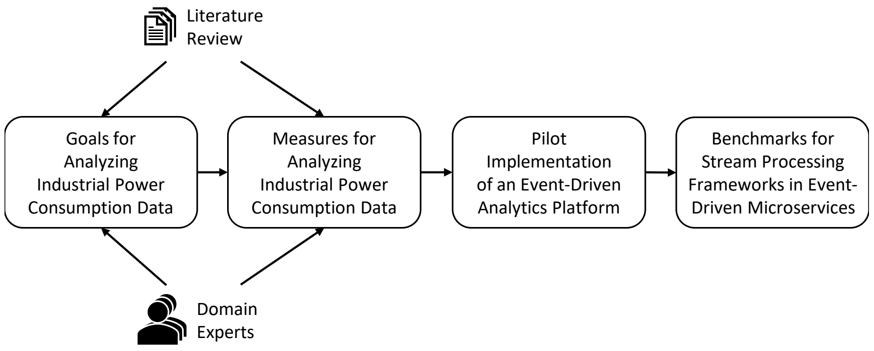

Literature Review
Goals for Analyzing Industrial Power Consumption Data
Measures for Analyzing Industrial Power Consumption Data
Pilot Implementation of an Event-Driven Analytics Platform
Benchmarks for Stream Processing Frameworks in Event-Driven Microservices
Domain Experts

Figure 9.2. Research design for designing relevant benchmarks for event-driven-microservices. The first three activities are discussed in Chapter 10, while the last activity is discussed in Chapter 11.

the presented use cases also apply to other types of IIoT data stream analytics.

Fig. 9.2 outlines our employed research design. Based on a literature review and knowledge from domain experts, we identify goals for analyzing industrial power consumption data and derive a set of software-based measures to tackle these goals. In a pilot implementation of an IIoT analytics platform, we show how our proposed measures can be implemented with an event-driven microservice architecture. From this architecture, we select a set of service functions, representing the use cases for our proposed benchmarks. For each identified use case, we present a corresponding dataflow architecture to be implemented with stream processing frameworks. In total, we present four dataflow architectures and provide corresponding implementations for four stream processing frameworks. We complement their benchmark definition by different load types, resource types and SLOs.

In the two remaining chapters of this part, Chapter 10 presents the identified goals and measures as well as our pilot implementation, while Chapter 11 presents the derived benchmarks.

Table 9.1. Overview of all scalability evaluations with our Theodolite benchmarks conducted in this thesis.

	Task sample				Framework								Cloud			
Evaluation	UC1	UC2	UC3	UC4	Flink	Beam/Flink	Beam/Samza	Kafka Streams	Hazelcast Jet	Others	Configurations	Algorithms	Private	Oracle (OCI)	Google (GCP)	Amazon (AWS)
Chapter 12	✓	✓	✓	✓	✓			✓					✓	✓	✓	
Chapter 13	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓			
Chapter 14			✓		✓			✓		✓[a]		✓	✓			
Chapter 15	✓		✓			✓	✓			✓					✓	✓

[a] We use implementations of OSPBench [vDvdP20] to construct a Theodolite benchmark for Apache Spark.

9.3 Evaluation Overview

Our Theodolite benchmarks are designed to evaluate different frameworks for event-driven microservices regarding their scalability. Moreover, they should allow evaluating different configuration options, deployment options and alternative algorithms. In Part IV of this thesis, we employ our benchmarks to perform various of such evaluations. Table 9.1 provides an overview of these evaluations, indicating the employed benchmarks, the evaluated stream processing frameworks, whether different configurations or algorithms are evaluated, and the used cloud provider. It shows that this thesis includes evaluations of all presented task samples (named UC1–UC4) and framework implementations as well as of different framework configurations, alternative algorithms, and cloud environments. As already shown in Section 5.2, we evaluate our Theodolite benchmarks with different load types, different resource types, and different SLOs. Further evaluations of our Theodolite benchmarks can be found by Boguhn [Bog22], Bensien [Ben21], Biernat [Bie20], and in our previous publication [HH21e].

Chapter 10

Industrial Internet of Things Analytics for the Case of Industrial Power Consumption

This chapter addresses research question RQ 2.1 of this thesis and identifies relevant use cases for event-driven microservices. As shown in our research design presented in the previous chapter, we focus on the domain of analyzing industrial power consumption. In two industrial pilot cases, we discuss how analyzing power consumption data can serve the goals reporting, optimization, fault detection, and predictive maintenance. Accompanied by a literature review, we propose to implement the measures real-time data processing, multi-level monitoring, temporal aggregation, correlation, anomaly detection, forecasting, visualization, and alerting in software to tackle these goals. In a pilot implementation of a power consumption analytics platform, we show how our proposed measures can be implemented with an event-driven microservice architecture.

This chapter summarizes our work previously published in the *Journal of Data, Information and Management* [HHB+21] and the *Software Impacts* journal [HH21d]. It is structured as follows. Section 10.1 presents the context of this study. Section 10.2 summarizes the results of our literature review. Section 10.3 briefly describes the current state of energy monitoring in our studied pilot cases. Section 10.4 presents the goals for analyzing power consumption data identified in our pilot cases, followed by our proposed measures for tackling these goals in Section 10.5. Section 10.6 shows how our proposed measures can be implemented in an analytics platform.

10.1 Context

The immense electrical power consumption of the manufacturing industry [Int19] is a considerable cost factor for manufacturing enterprises and a serious problem for environment and society. Corporate values, public relations, energy-related costs, and legal requirements are therefore leading to an increasing energy awareness in enterprises [SGO17]. At the same time, trends toward the Industrial Internet of Things, Industry 4.0, smart manufacturing, and cyber-physical production systems [SSH+18] allow for collecting energy data in real time and at machine level, from smart meters or machine-integrated sensors [SOM14; MAL19]. Furthermore, research on big data provides methods and technologies to analyze data of huge volume and high velocity, as it is the case with power consumption data [SCG+14; ZMY+18]. However, even though research suggests a variety of goals and measures for analyzing power consumption data, the full potential of available data is rarely exploited [SM15; BVS+11; CS19].

In our Industrial DevOps [HHL+19] research project Titan [HHL+21], we work on methods and tools for integrating and analyzing big data from Internet of Thing devices in industrial manufacturing. Analyzing power consumption data in two enterprises of the manufacturing industry serves as a case study. Both enterprises are project partners of wobe-systems and Kiel University in the Titan project.

10.2 Literature Review

Analyzing industrial energy data is an emerging field of research. In this section, we highlight the findings of our literature review regarding goals and measures for analyzing power consumption data as well as related work on implementing such measures.

10.2.1 Goals for Analyzing Power Consumption Data

A lot of research exists, in particular, on how energy data analysis can contribute to reducing the energy usage in manufacturing. For example, Vikhorev et al. [VGB13] point out that making energy data available for production operators promotes energy awareness. Cagno et al. [CWT+13] show that a lack of energy consumption information prevents implementation of energy-saving measures. Detailed information is especially required at process and machine level for optimizing energy consumption, as highlighted by Thollander et al. [TPC+15]. For systematic monitoring and optimizing energy consumption, enterprises are moving toward establishing an energy management [CS19]. Implementing an energy management requires revealing all energy consumptions within the enterprise [FM12]. Schulze et al. [SNO+16] identify organizational measures for implementing an energy management in industry.

Increasing availability of smart meters and Internet of Things (IoT) adoption in the manufacturing industry (Industry 4.0) enable enterprises to collect energy data in great detail [SM15; MS13; MAL19]. This includes commercial metering systems as well as prototypical low-cost systems as proposed by Jadhav et al. [JKP21]. Shrouf and Miragliotta [SM15] highlight several benefits of IoT adoption for energy data obtained from reviewing literature and information published by European manufacturing enterprises. Tesch da Silva et al. [TdCP+20] present a systematic literature review on energy management in Industry 4.0.

Implementing an energy management and analyzing monitored energy data assist an enterprise in understanding its energy consumption [MS13; VGB13; SGO17]. It provides insights into which devices, machines, and enterprise departments use how much power and during which times this power is consumed. Combined with information about the production processes, reports can thus be used to identify which processes consume how much power [HT09]. In this way, measures for optimizing energy consumption can be evaluated and saving potentials identified [BVS+11].

The literature focuses particularly on optimizing power consumption for economical and ecological reasons [BVS+11; MS13; SOG+14; SNO+16; SGO17]. Mohamed et al. [MAL19] report on opportunities provided by IoT energy data for improving energy efficiency and reducing energy costs. Shrouf and Miragliotta [SM15] focus on optimizing energy usage to reduce costs and improve reputation, for example, by reducing energy wastage and improving production scheduling. In their systematic literature review on energy management in Industry 4.0, Tesch da Silva et al. [TdCP+20] outline methods for improving energy efficiency and point out current limitations for their implementation.

To optimize the overall power consumption, it can be expedient to optimize the operation of machines in production individually. For example, Vijayaraghavan and Dornfeld [VD10] optimize the power consumption of machine tools to reduce the power consumption of an entire manufacturing system. Shrouf et al. [SOG$^+$14] optimize the production scheduling of a single machine for minimizing overall energy consumption costs.

A special optimization aspect is the reduction of peak loads [HT09; VGB13; SM15]. In addition to the basic price, which is fixed per month, and the price per kilowatt hour, large-scale power consumers such as manufacturing enterprises often have to pay a demand rate. The demand rate depends on the maximum demand that occurs within a billing period. In this way, grid operators expect to have a load as uniformly as possible in the electricity grid [AE08]. Demand peaks are therefore disproportionately more expensive for the customer. Thus, an optimization should aim to achieve a power consumption as constant as possible, i.e., to distribute the demand evenly over time (peak shaving). In order to achieve this, it is necessary to identify periods during which relatively much power is demanded. Likewise, it is important to discover which consumers are responsible for the demand and to what extent [HT09].

Other goals besides reporting and optimization can only rarely be found in the literature. Quiroz et al. [QMM$^+$18] report how power consumption, which deviates from its normal behavior, can be an indicator for a fault such as a mechanical defect or faulty operation. Analyzing the power consumption of machines can therefore be used to automatically detect such faults and to react accordingly [VD10; MAL19]. Further, analyzing power consumption data may allow prediction of future faults so that necessary maintenance actions can be taken [SOM14; MAL19].

10.2.2 Measures for Analyzing Power Consumption Data

Many studies consider near real-time processing of energy data to be necessary [VD10; VGB13; SCG$^+$14; HHM$^+$18; LN18]. Proposed implementations, therefore, often use stream processing techniques and tools (see also the following Section 10.2.3). Several studies point out that many types of power consumption analysis require consumption data at different levels [VGB13; SM15; KJJ$^+$20]. Whereas, for example, the effect of overall power consumption optimizations can be evaluated with data of the overall power consumption, detecting defects in machines requires acquiring data at machine level. Moreover, different stakeholders are often interested in power consumption reports of different granularity [SGO17]. In addition to aggregating the power consumption of multiple consumers to larger groups, it is often required to aggregate multiple measurements of the same consumer over time [SM15]. Analyzing energy data often yields significantly better results if, in addition to recorded power consumption, further information is included such as operational and planning data from the production as well as business data [VD10; SGO17].

Most approaches for energy analytics platforms and energy management systems include data visualizations [FM12; VGB13; SCG$^+$14; ZMY$^+$18]. Visualizations are often realized as information dashboards, which contain multiple components providing different types of visualization. Individual components show, for example, the current status of power consumption as numeric values or gauges [RM19; VGB13], the evolution of consumption over time in line charts [VGB13; SCG$^+$14; FM12], the distribution among subconsumers and categories (also in the course of time) [VGB13; MLB$^+$15], correlations of individual power consumers [SCG$^+$14; MBL$^+$17], particularly important values such as the peak load [VGB13], detected anomalies [CTC$^+$17], or forecasted power consumption [SY18].

Research also exists on forecasting power consumption or detecting anomalies in power consumption data. Both practices are closely related. Methods for forecasting and anomaly detection create models of the past power consumption, explicitly or implicitly, and project them into the future (forecasting [MTA$^+$15]) or compare them to the the actual power consumption (anomaly detection [CT14; LN18]). Common approaches use statistical methods such as ARIMA [CKK13] or kernel density estimation [AT16], machine learning methods such as artificial neural networks [DM17; ZXZ$^+$17], or a combination of both [CT14]. Whereas much forecasting and anomaly detection research exists on energy consumption of households [CKK13; LN18], buildings [AT16; CT14], and electricity grids [DM17; ZXZ$^+$17], approaches regarding power consumption of industrial production environments are rare due to their irregular nature [BTV+18]. Liu and Nielsen [LN18] show how alerts could be triggered when anomalies are detected.

10.2.3 Implementation of Measures

Software systems for implementing such measures are presented, for example, by Sequeira et al. [SCG+14] and Rackow et al. [RJD+15]. Yang et al. [YCL+20] propose such a system for assessing power consumption at a university campus. However, these systems only focus on a subset of measures proposed in this chapter.

A couple of software architectures for implementing energy data analysis are suggested. Several architectures [SCG+14; SGO17; HHM+18; LN18] follow the Lambda architecture pattern [MW15]. Such architectures deploy a speed layer for fast online processing and a batch layer for accurate offline processing of data. In our pilot implementation (see Section 10.6), we pursue a more recent architectural style of processing data exclusively online (also referred to as Kappa architecture) [Kre14] by utilizing Apache Kafka's capabilities for reprocessing distributed, replicated logs [WKS+15]. Additionally, we combine this with the microservice architecture pattern and design dedicated, encapsulated microservices per analytics task. Benefits of using microservices and, in particular, the associated concept of polyglot persistence for analyzing industrial energy usage are highlighted by Herman et al. [HHM+18] and in our previous publication [HHM19].

Big data analytics of energy consumption heavily relies on cloud computing [SOM14; HHM+18; SCG+14; MAL18; YCL+20]. Sequeira et al. [SCG+14] propose *cloud connector* software components for integrating data from energy meters. Recent studies suggest applying fog computing for integrating production data in general [QT19] and energy consumption data in particular [MAL19]. Our pilot implementation follows the suggestions of Pfandzelter and Bermbach [PB19] to deploy data analytics using stream processing in the cloud and data preprocessing and event processing in the fog. Szydlo et al. [SBS+17] present how data transformation at fog computing nodes can be implemented using flow-based programming [Mor10] and graphical dataflow modeling. Hasselbring et al.

[HWD21] present how we apply flow-based programming [Mor10] and graphical dataflow modeling in our Titan project.

10.3 Studied Pilot Cases

In this section, we give a brief overview of our two studied pilot cases. Both pilot cases are enterprises of the manufacturing industry.

The first studied enterprise is a newspaper printing company. It is characterized by high requirements on production speed and the fact that production downtimes are extremely critical. The company has to print and deliver daily newspapers for the next day within only a few hours during the night. If newspapers were printed too late, they would not be up to date anymore and could no longer be sold. Production failures would therefore be associated with significant economic loss. The characteristic production times, with peaks in the nights before working days, are reflected, for example, in the power consumption of the air compressors as depicted in Fig. 10.1. In addition to daily newspaper printing, the company prints advertising supplements, weekly newspapers, and customer magazines to utilize production capacity.

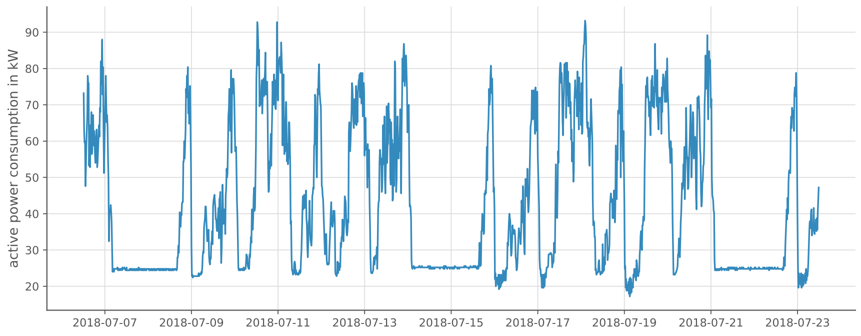


Figure 10.1. Power consumed for generating compressed air in the newspaper printing company over a period of 2.5 weeks [HHB+21]. The curve shows a weekly pattern and reflects the company's operating hours with constant low consumption at the weekend and peak loads at night.

The second studied enterprise is a manufacturer of optical inspection systems for non-man-size pipelines and wells. This enterprise is characterized by a high vertical range of manufacturing. Thus, its production environment operates a wide range of machines, some of which are largely automated, others are primarily user-controlled. Furthermore, the manufacturer operates a rather large data center which runs software for its administration, development, and production. In this work, we focus on power consumption of the production processes and not on the power consumption of inspection systems themselves.

Both enterprises already have the necessary physical infrastructure to record electrical power consumption in production and query it during operation. Electricity meters capture the required data at machine level and with high frequency. We therefore do not include approaches and techniques for acquiring power consumption data in this study. However, both companies do not yet exploit the full potential of the recorded and stored data. Currently, they analyze the data mainly by hand and only at certain times. Much of the information hidden in power consumption data is therefore not revealed yet. The reasons for this cannot be found in missing interest, but in a lack of applicable technologies. Currently, the production operators use software provided by metering device manufactures, for example, to visualize the stored data. However, this software does not meet all requirements. For example, the amount of visualized data is too large, making it hard to extract the crucial information. Another issue is the integration of different types of electricity meters. Although standardized protocols exist, many metering devices and systems do not apply them.

10.4 Goals for Analyzing Power Consumption Data

In this section, we identify motivations for analyzing power consumption data in our studied pilot cases. We classify these motivations into the four goal categories reporting, optimization, fault detection, and predictive maintenance. Our literature review (see Section 10.2) suggests that these goals also occur in other manufacturing enterprises as similar motivations can be found in related studies. In the following, we describe each goal category in detail.

10.4.1 Reporting

In both studied enterprises, comprehensive reporting is particularly required for an ISO 50001 [Int18] certification. The ISO 50001 standard specifies requirements for organizations and businesses for establishing, implementing, and improving an energy management system. It describes a systematic approach to support organizations in continuously improving their energy efficiency. In order to be certified to use an energy management system in compliance with ISO 50001, enterprises commission accredited certification bodies to perform regular independent audits [JF16]. These certifications are usually not required by law, but serve as evidence that a company is making efforts to save energy.

Both companies consider sustainability as an important pillar of their corporate philosophy. ISO 50001 certification allows them to demonstrate their efforts in saving energy to customers and other stakeholders. Moreover, in Germany, where both enterprises are located, ISO 50001 certification enables cost savings as such a certification is a prerequisite for manufacturing enterprises with high power consumption to reduce regulatory charges (e.g., reducing the EEG surcharge [Bun20]). Certification is even essential for the manufacturer of optical inspection systems. Its customers are mainly public authorities, which often require ISO 50001 certification in their calls for tender.

Reports for ISO 50001 certification are required to justify irregular or increasing power consumption. This is in particular challenging for the newspaper printing company, where power consumption highly depends on the production utilization and external influences. Hence, this company requires complex analyses for their reports, such as correlations with external data from the production and the environment. Moreover, the ISO 50001 standard requires that reports on the enterprise's energy consumption are available to customers, stakeholders, employees, and management.

10.4.2 Optimization

The ecological and economical motivations for optimizing energy consumption presented in our literature review (see Section 10.2) also apply to both our pilot cases. We identify the following types of optimizing power consumption in the enterprises studied.

Optimization of Overall Consumption A first step for optimizing the overall power consumption is identifying energy-inefficient machines and devices. This knowledge can then be used to replace them with more energy-efficient ones or retrofitting them accordingly. Furthermore, time periods should be detected in which devices consume energy, although it would not be necessary. Typical examples of unnecessary energy consumption are keeping machines in standby mode or lighting workplaces outside of working hours, but also less apparent saving potential is expected to be discovered.

Optimization of Peak Loads Being large-scale power consumers, both studied pilot cases have to pay a demand rate based on the maximum demand within a billing period. Since demand peaks are disproportionately more expensive, they aim for distributing the demand more evenly. As illustrated in Section 10.2.1, this requires identifying time periods and consumers with high power consumption This includes, on the one hand, the identification of large consumers in general but, on the other hand, also demand fluctuations of individual devices. Based on this information, production processes can be modified such that, for instance, multiple machines with a high inrush current are not started at the same time. Reducing the overall energy consumption is highly related to reducing peak loads. If measures are taken to replace devices, this has an effect on both optimization goals. For example, if devices that are unnecessarily operated standby during load peaks are turned off during these periods, not only demand peaks are reduced, but also the enterprise's power consumption in total.

Optimization at Machine Level Similar to what we present in Section 10.2, it is reasonable to optimize the operation of machines or production processes to optimize the overall power consumption or peak loads. A potential power saving measure in the newspaper printing company exists in the printing process. The number of newspapers produced per unit of time depends directly on the operating speed of the printing presses. To determine an optimal printing speed, several other factors are also taken into account, such as reliability, which decreases when increasing the production speed. With monitoring and analyzing the printing presses' power consumption, the company can also include energy-related costs when determining the production speed.

10.4.3 Fault Detection

The studied enterprises report that a power consumption of machines, deviating from their normal behavior can be an indicator for a fault such as a mechanical defect or faulty operation. Analyzing the power consumption can therefore be used to automatically detect such faults and to react accordingly. A typical case of anomalous power consumption is a strong increase, for example, when a defect occurs suddenly. A decrease of power consumption can also be such an indicator as parts of a machine may no longer be operated due to a defect. Less noticeable is a slight deviation over a longer period of time, for example, if several minor defects occur over time. Detecting deviations or a long-term trend in regularly fluctuating power consumption is even more challenging.

The central compressed air supply in the newspaper printing company is an example for fault detection using power consumption data. An extensive pipe network supplies various areas of the production environment and finally individual machines with compressed air. The compressed air distribution network leaks regularly, causing air to escape. These leaks do not necessarily become apparent directly, but should still be repaired. As leaks result in higher power consumption of the air compressors, power consumption data can provide an instrument for leak detection. However, since power consumption of the air compressors is subject to strong, irregular fluctuations (see Fig. 10.1), an increase in power consumption does not immediately become apparent. This may be solved by considering the power consumption only in idle times, for example, during the weekend. An increase in power consumption over several weekends may thus be

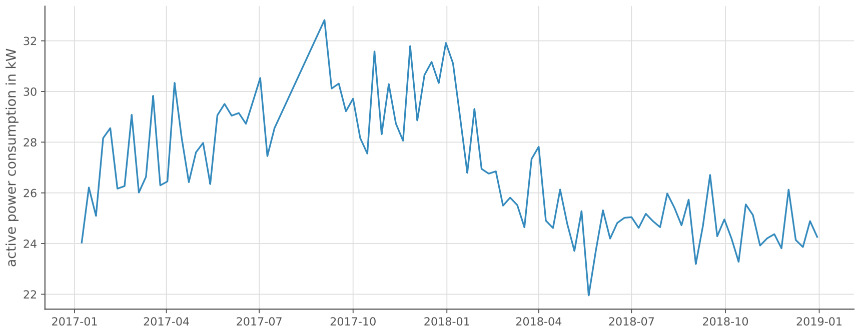


Figure 10.2. Stand-by power consumption for generating compressed air in the newspaper printing company at weekends over a period of two years [HHB+21].

an indication of a leak. Fig. 10.2 shows the average power consumption between Saturday 12:00 and Sunday 12:00 for each weekend in 2017 and 2018. The course shows a steady increase in 2017 due to leaks in the compressed air supply. In early 2018, the company repaired several leaks, causing a tangible reduction in power consumption.

10.4.4 Predictive Maintenance

With regular, time-based maintenance intervals, machines and devices are often maintained even though there is no actual need for it. This means that components and operating materials are replaced since their expected operating time expires, although they are still functioning and could actually continue operating. Predictive maintenance is an approach that aims for performing maintenance actions only if it would otherwise result in defects or limitations in performance or quality [YML+17]. The difficulty is therefore to decide when maintenance is really necessary. For this purpose, sensor data of the machine and its environment are collected and automatically analyzed [YML+17]. Our literature review (see Section 10.2) suggests that power consumption can be such data.

We distinguish between predictive maintenance and fault detection as while fault detection aims to detect errors after they occurred, predictive maintenance refers to the detection of errors before they occur.

Nevertheless, predictive maintenance is closely related to fault detection as occurring faults often cause further faults. Early fault detection may allow detecting future faults and taking appropriate preventive measures.

An example for predictive maintenance using power consumption data are cooling circuits as used in the studied enterprises. Such circulation systems typically include a filter through which coolant is pumped to remove impurities. These filters need to be replaced regularly. The electrical power consumption of the pump indicates the resistance within the circulation system and, thus, how polluted the filter is. Increased power consumption can therefore serve for detecting an upcoming filter change. Lower power consumption can also provide information. It may indicate that not enough coolant is in the circuit (referred to as dry run) and, thus, coolant needs to be refilled.

10.5 Measures for Analyzing Power Consumption Data

In this section, we discuss software-based measures for analyzing power consumption data that support in achieving the goals defined in the previous section. Based on our literature review in Section 10.2 and knowledge from domain experts within our studied pilot cases, we suggest the following measures: real-time data processing, multi-level monitoring, temporal aggregation, correlation, anomaly detection, forecasting, visualization, and alerting. Different use cases gauge goals differently and measures vary in their importance for the individual goals. We therefore rate the impact of each measure on each goal and visualize these impacts on radar charts shown in Fig. 10.3. In the following, we briefly describe each measure and characterize how each measure affects each goal.

10.5.1 Near Real-time Data Processing

Near real-time (also referred to as online) data processing describes approaches, where data are immediately processed after their recording. It contrasts batch (also referred to as offline) processing, which first collects recorded data and then processes all the collected data only at certain

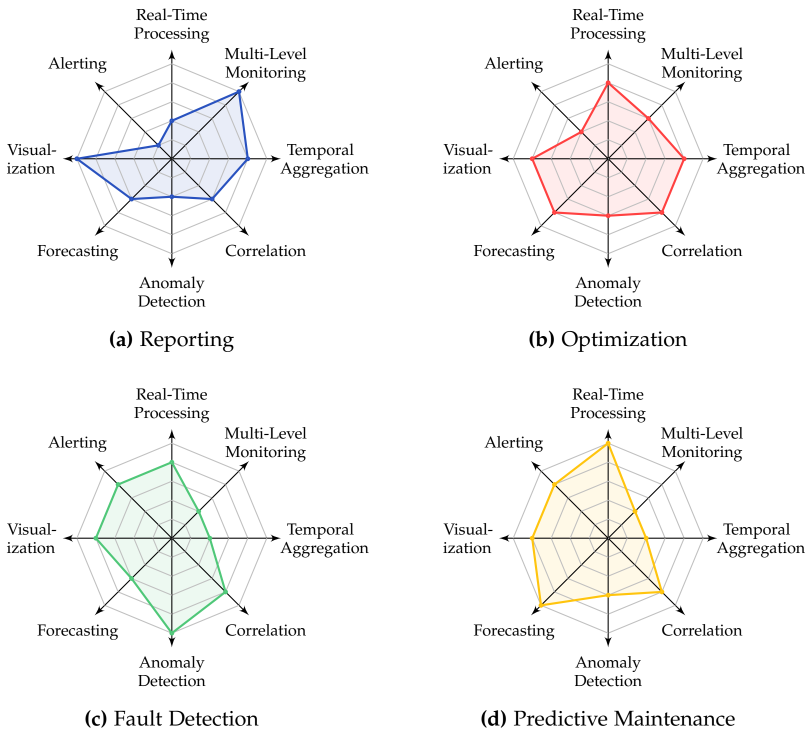


Figure 10.3. Impact rating of the proposed measures for the four goals presented in Section 10.4 [HHB+21]. The larger its distance from the radar chart's center is, the higher a measure's impact was weighted on the corresponding goal.

times. Whereas near real-time data processing is usually more difficult to design and implement than batch processing, it yields immediate results and, thus, allows reacting immediately on these results.

Data processing in near real-time primarily supports the goals optimization (see Fig. 10.3b), fault detection (see Fig. 10.3c), and predictive maintenance (see Fig. 10.3d) [VD10; SM15]. Power consumption can be efficiently optimized if the effectiveness of energy-saving actions are evaluated immediately. The sooner a fault is detected and reported, the faster it can be reacted to and, therefore, the more valuable its detection is. Predictive maintenance requires processing monitoring data in real time as otherwise the time for maintenance may be determined after the maintenance should have already been performed [SBA20]. Although a real-time overview of the enterprise's energy usage at any time is not required for ISO 50001 audits, it assists in reporting (see Fig. 10.3a) the power consumption, for example, to the management [MS13].

10.5.2 Multi-Level Monitoring

We suggest organizing power consumers in a hierarchical model, where groups of devices and machines are further grouped into larger groups [HH20b]. Multiple such models have to be maintained in parallel. For example, it is reasonable to organize devices by their type (e.g., all air compressors), but also to organize them by their physical location (e.g., a certain shop floor).

Besides monitoring groups of consumers, for example, via sub-distribution units, data for groups can also be obtained by aggregating the consumption of all its partial consumers. In particular, this is necessary for devices which, for reasons of redundancy, have more than one power supply. Here, the overall machine's power consumption is usually more important than the power consumption of the individual power supplies. Comparing the power consumption monitored by sub-distribution units with aggregated data of all known sub-consumers may reveal consumptions that were unknown so far.

Hierarchical models of power consumers particularly support reporting (see Fig. 10.3a) as they offer insights at which times which consumers or groups consume how much power. Power consumption can be optimized (see Fig. 10.3b) at machine level as well as on aggregated data (see Section 10.2). Furthermore, our literature review shows that fault detection (see Fig. 10.3c) and predictive maintenance (see Fig. 10.3d) may also be performed on different levels.

10.5.3 Temporal Aggregation

Temporal aggregation refers to summarizing multiple measurements of the same consumer over time to one data point. It serves for: (1) reducing the number of data points for storage and (2) simplifying data analysis by providing a more abstract view on the data. Therefore, temporal aggregation supports humans in comprehending the monitored power consumption data and, thus, reporting (see Fig. 10.3a) as well as manually identifying optimization potentials. Also automatic data processing for optimization (see Fig. 10.3b), fault detection (see Fig. 10.3c), and predictive maintenance (see Fig. 10.3d) may benefit from aggregated data. We distinguish two different kinds of temporal aggregation as described in the following.

Downsampling The first kind is to collect and aggregate all measurements in consecutive, non-overlapping, fixed-sized time windows (called downsampling). An appropriate size for such windows is, for example, 5 minutes so that every 5 minutes a new aggregation result is computed representing the average, minimal, and maximal power consumption over the previous 5 minutes. The number of data points can thus be massively reduced, which is required for several forms of storing, analyzing, and visualizing data. We suggest performing multiple such aggregations (e.g., for time windows of size 1 minute, 5 minutes, and 1 hour) and store their aggregation results for different durations. This allows storing more recent (and more interesting) data with more detail than data from the previous months or years.

Aggregating Temporal Attributes The second kind of temporal aggregation is to aggregate all data points having the same temporal attribute such as day of week or hour of day. The set of aggregated data points allows modeling or identifying seasonality. For example, aggregating all measurements recorded at the same day of week allows showing the average power consumption course over a week. Likewise, aggregating based on the hour of the day allows obtaining the average course of a day.

10.5.4 Correlation

Our literature review (see Section 10.2) shows how operational and planning data from the production as well as business data can be included in different types of power consumption analysis. Furthermore, it is reasonable to correlate the power consumption of different consumers as their power consumption may depend on each other if their production processes are dependent [BTV+18].

Correlating power consumption data with production data supports reporting (see Fig. 10.3a) as it allows for better understanding the power consumption. Management levels might be interested in a correlation with business data as this allows reporting about, for example, the energy costs per produced unit. In particular, correlation can serve as trigger for optimization (see Fig. 10.3b), fault detection (see Fig. 10.3c), and predictive maintenance (see Fig. 10.3d). If, for example, the power consumption of a machine increases rapidly while also the production speed increases, the increasing power consumption was most likely not caused by a fault. If, however, the production speed remains constant and no other production data justifies the increase, a fault detection could be triggered. Correlations of power consumption of different consumers are interesting for reporting, but also for optimizations, in particular for reducing load peaks.

10.5.5 Anomaly Detection

We suggest employing anomaly detection techniques to discover time periods, during which power consumption is unexpectedly high or low, like it is done for energy consumption of buildings or households in related work (see Section 10.2). This includes continuously computing anomaly scores for observed power consumption and comparing these anomaly scores with previously defined thresholds. We suggest applying anomaly detection both on monitored and aggregated data (see Section 10.5.2).

Primarily, anomaly detection serves as a measure for the goal of fault detection (see Fig. 10.3c). Faults in devices, machines, or production processes are deviations from the desired behavior and, thus, anomalous behavior of power consumption may indicate an occurring fault. Detecting anomalies in power consumption exclusively in relation to time is often not sufficient. The consumption of many devices is subject to external influences such as temperature [LN18] and, especially in production environments, the operating times of machines do not follow daily or weekly patterns [BTV+18]. Correlating power consumption with environmental, operational, and planning data (see Section 10.5.4) therefore assists in detecting anomalies.

Furthermore, anomaly detection allows identifying potential applications of optimization (see Fig. 10.3b) and predictive maintenance (see Fig. 10.3d). It also supports in explaining power consumption behavior in reporting (see Fig. 10.3a).

10.5.6 Forecasting

As highlighted in our literature review (see Section 10.2), analyzing power consumption data allows making predictions about the future power consumption. Similar to anomaly detection, predicting power consumption in industrial production poses additional challenges in contrast to predicting power consumption of households, buildings, or electricity grids. To cope with the irregular nature of industrial power consumption, correlation with environmental, operational, and planning data (see Section 10.5.4) promises to create more accurate models.

Forecasting the power consumption of machines in addition to the overall production environment supports optimization (see Fig. 10.3b) as it allows detecting load peaks before they actually occur. Thus, production operators may take appropriate countermeasures such as replanning production processes. Making predictions about the future status of the production environment is required for predictive maintenance. Thus, forecasting power consumption enables predictive maintenance (see Fig. 10.3d) based on power consumption data. Fault detection (see Fig. 10.3c) based on anomaly detection often relies on forecasts by comparing the actual consumption with the expected (i.e., forecasted) one. Furthermore, forecasting can be used in reporting (see Fig. 10.3a) as it supports planning and decision making for business and production operation.

10.5.7 Visualization

According to our literature review in Section 10.2, we propose to visualize analyzed power consumption data in information dashboards. This way, visualizations integrate individual measures proposed in this chapter and serve as a link between data analysis and the users. Dashboards should be dynamic and interactive in the sense that they are updating their visualized data continuously and let users interact with them [RM19].

For example, dashboards may start with a rough outline of the overall production's power consumption but allow users to zoom in and show specific machines and time periods in detail.

As state-of-the-art libraries and frameworks for data visualization are largely based on web technologies [BOH11; LMS+18], it is reasonable to implement dashboards as web applications. This has the additional advantage that the visualization is user-friendly accessible since it does not have further requirements on software or hardware infrastructure than a web browser.

First and foremost, dashboards enable reporting (see Fig. 10.3a) on power consumption. Appropriate visualization allows for understanding how power consumption is composed, observing changes in power consumption over time, and comparing the power consumption of different machines and production processes. Enterprises may provide different dashboards for different stakeholders to only show the information relevant for the corresponding target audience [SM15]. Visualizations assist in optimization (see Fig. 10.3b) as they allow identifying optimization potentials and enable operators to check whether optimization actions are effective. Furthermore, interactive visualizations can motivate, trigger, and enable energy saving actions [RM19]. A dashboard may also show information concerning fault detection (see Fig. 10.3c) and predictive maintenance (see Fig. 10.3d) and provide means to verify whether faults and maintenance actions are detected successfully [SM15].

10.5.8 Alerting

Since industrial production becomes increasingly automated [SSH+18; FMR+22], production operators should automatically be notified when faults are detected or maintenance actions have to be taken (see Fig. 10.3c and Fig. 10.3d). Depending on their frequency and severity, such notifications and alerts may be sent via email or communication tools. For reporting (see Fig. 10.3a) purposes, such notifications may additionally be displayed in a dashboard. Furthermore, operators may be notified if optimization potential is detected (see Fig. 10.3b), for example, by generating an alert if a load peak is about to occur.

10.6 Pilot Implementation of the Measures

In this section, we show how the measures proposed in Section 10.5 can be implemented with an event-driven software architecture. In our Titan project on Industrial DevOps [HHL+19; HHL+21], we develop methods and techniques for integrating Industrial Internet of Things data. A major emphasis of the project is to make produced data available to various stakeholders in order to facilitate a continuous improvement process. The Titan Control Center[1] is our open-source pilot application for integrating, analyzing, and visualizing industrial big data from various sources within industrial production [HH21d].

Fig. 10.4 shows the Titan Control Center architecture. It is composed of event-driven microservices as introduced in Section 4.3 with the microservices `Aggregation`, `History`, `Statistics`, `Anomaly Detection`, `Forecasting`, and `Sensor Management`. Microservices communicate with each other asynchronously via Apache Kafka. The Titan Control Center features two single-page applications that visualize analyzed data and allows for configuring the analyses. In a previous publication [HHM19], we show how these architecture decisions facilitate scalability, extensibility, and fault tolerance of the Titan Control Center.

The Titan Control Center is deployed following the concepts of edge and fog computing [GME+15; BMZ+12]. In particular suited for Internet of Things (IoT) data streams, edge and fog computing architectures preprocess data at the edges of the network (i.e., physically close to the IoT devices), whereas complex data analytics are performed in the cloud [PB19]. In order to facilitate scalability and fault tolerance, the Titan Control Center microservices for data analysis and storage are deployed in a cloud environment. This can be a public, private, or hybrid cloud, which allows elastic increasing and decreasing of computing resources. On the other hand, software components for integrating power consumption data into the Titan Control Center are deployed within the production. This includes querying or subscribing to electricity meters, format and unit conversions, filtering, but also aggregations to reduce the amount of data points. We employ the Titan Flow Engine [HHL+19; HWD21] for this purpose. It allows graphical modeling of data flows in industrial produc-

[1] `https://github.com/cau-se/titan-ccp`

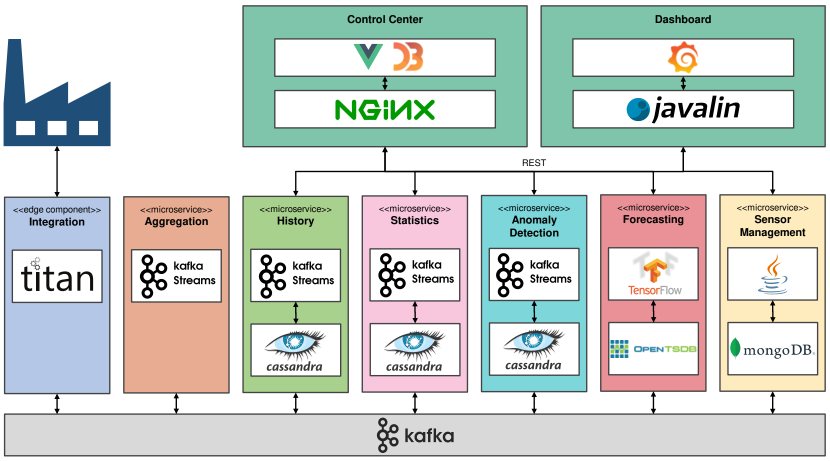


Figure 10.4. Microservice-based pilot architecture of the Titan Control Center for analyzing electrical power consumption [HHB+21].

tion according to flow-based programming [Mor10]. With the Titan Flow Engine, individual processing steps are implemented in so-called *bricks*, which are connected via a graphical user interface to *flows*. This enables production operators to reconfigure power consumption data flows, for example, to integrate new electricity meters, without requiring advanced programming skills.

In the following, we present how each measure proposed in Section 10.5 can be implemented using the Titan Control Center.

10.6.1 Near Real-Time Data Processing

Power consumption data is processed in near real time at all architectural levels of the Titan Control Center. This starts by the ingestion of monitoring data and immediate filter, convert, and aggregate operations in the Titan Flow Engine at the edge. The final integration step is sending the monitoring data to the messaging system. Following the publish–subscribe pattern, microservices subscribe to this data stream and are notified as soon as new data arrive. In the same way, individual microservices communicate with each other asynchronously. Within microservices, we process data using stream processing techniques. This implies that microservices continuously compute and publish new results as new data arrive. For implementing stream processing architectures in most of the microservices, we use Kafka Streams. As all computations are performed in near real time, the visualizations can also be updated continuously. Hence, the visualization applications (see Section 10.6.7) periodically request new data from the individual services.

10.6.2 Multi-Level Monitoring

The Aggregation microservice [HH19b] of the Titan Control Center computes the power consumption for groups of machines by aggregating the power consumption of the individual subconsumers. This microservice subscribes to the stream of power consumption measurements coming from sensors, aggregates these measurement continuously according to configured groups, and publishes the aggregation results via the messaging system as if they were real sensor measurements. In addition to sensor measurements, however, these data are enriched by summary statistics of the aggregation.

As proposed in Section 10.5.2, the Aggregation microservice supports aggregating sensor data in arbitrarily nested groups and multiple such nested group structures in parallel. In one of our studied enterprises, we integrate power consumption data of different kinds of sensors, which provide data in different frequencies. An important requirement for the Aggregation service was therefore to support different sampling frequencies. Furthermore, besides the focus on scalability throughout the entire Control Center architecture, an important requirement for this microservice is to reliably handle downtimes and measurements arriving out-of-order. Therefore, it allows configuring the required trade-off between accuracy, aggregation latency, and performance [HH20b].

The Sensor Management microservice of the Titan Control Center allows assigning names to sensors and arranging these sensors in nested groups. For this purpose, the Titan Control Center's visualization component provides a corresponding user interface. The Sensor Management service stores these configurations in a MongoDB [Mon22] database. It publishes changes of group configurations via the messaging system so that the Aggregation service (and potentially other services) are notified about these reconfigurations. The Aggregation service is designed in a way that, when receiving reconfigurations, it immediately starts aggregating measurements according to the new group structure. Further, as aggregations are performed on measurement time and not on processing time, it supports reprocessing historical data.

10.6.3 Temporal Aggregation

Both types of temporal aggregation discussed in Section 10.5.3 are supported by the Titan Control Center. As both types serve different purposes, they are implemented in individual microservices. Both services subscribe to input streams, which provide monitored power consumption from sensors as well as aggregated power consumption for groups of machines.

Downsampling The History microservice receives incoming power consumption measurements and continuously aggregates all data items within consecutive, non-overlapping, fixed-sized windows (often referred to as tumbling windows in stream processing frameworks [CKH19]). The results of these aggregations are stored to an Apache Cassandra [LM10] database as well as published for other services. The History service supports aggregations for multiple different window sizes in parallel, allowing to generate time series with different resolutions. To prevent the amount of stored data from becoming too large, time series of different resolutions are assigned different *times to live*. Thus, the Titan Control Center allows, for example, storing raw measurements captured with high frequency for only one day, but aggregated values in minute resolution for years. Window sizes and times to live can be individually configured according to requirements for trackability and availability of storage infrastructure.

Aggregating Temporal Attributes The Statistics microservice aggregates power consumption measurements by a temporal attribute (e.g., day of week) to determine an average course of power consumption, for example, per week or per day. These statistics are continuously recomputed, stored in a Cassandra database, and published for other services, whenever new input data arrives. In our studied pilot cases we found out that in particular the average consumptions over the day, the week, and the entire year allow detecting patterns in the consumption. Furthermore, aggregating temporal attributes such as the month of the year over one year allows observing how monthly peak loads evolve over time.

10.6.4 Correlation

The Titan Control Center provides different features for correlating power consumption data. One of these features is graphical correlation of power consumption of different machines or machine groups. Our visualization component (see Section 10.6.7) provides a tool, which allows a user to compare the power consumption of multiple consumers in time series plots (see Fig. 10.5). It displays multiple time series plots below each other, each containing multiple time series. The user can zoom into the plots and shift the displayed time interval. All charts are synchronized by the time domain, thus zooming or shifting one plot also effects the others [JFD+16]. This tool allows operators to analyze interesting points in time (such as outages or load peaks) in more detail.

Together with the newspaper printing company, we implemented a first proof of concept for correlating real-time production data with power consumption data. We correlated the printing machines' power consumption with their printing speed. For this purpose, we integrated the production management system using the Titan Flow Engine and visualized both types of data in our visualization component. Even though we were able to show the feasibility of such a real-time correlation, we identified that for in-depth analyses, power consumption data with higher accuracy is required. Similarly, we prototypically correlated the power consumption of air conditioning systems with weather data. We identified a high impact of the outside temperature on the power consumed for cooling and, thus, use weather data as a feature for our forecasting implementations (see Section 10.6.6).

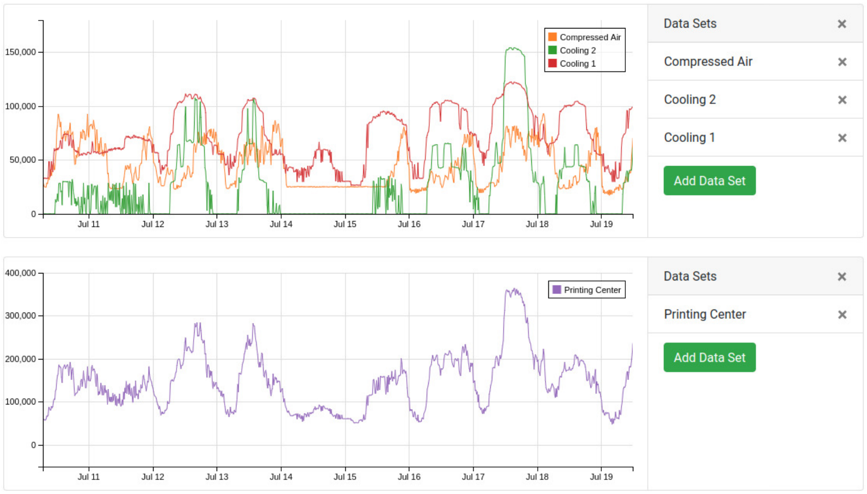


Figure 10.5. Screenshot showing the graphical correlation of power consumption using the Titan Control Center [HHB+21].

10.6.5 Anomaly Detection

The Titan Control Center envisages individual microservices for independent anomaly detection tasks and, hence, allows choosing an appropriate technique for each task. This includes individual techniques for different production environments and even for different machines.

With our pilot implementation, we already provide an Anomaly Detection microservice, which detects anomalies based on summary statistics of the previous power consumption. These statistics (e.g., per hour of week) are continuously recomputed by the Stats microservice (see Section 10.6.3) for each machine and machine group and published via the Control Center's messaging system. Our Anomaly Detection microservice subscribes to this statistics data stream and joins it with the stream of measurements (from real machines or aggregated groups of machines). Ultimately, this means each incoming measurement is compared to the most recent summary statistics of the corresponding point in time and machine. If the measured power consumption deviates too much from the average consumption of the respective hour and weekday, it is considered as anomaly. Instead of joining the measurements stream with the statistics stream, it can also be joined with the data stream published by the forecasting service (see Section 10.6.6). Using forecast data requires more complex model training, but might yield more accurate results.

All detected anomalies are again published to a dedicated data stream via the messaging system, allowing other microservices to access detected anomalies. Moreover, the microservice stores all detected anomalies in a Cassandra database.

10.6.6 Forecasting

Similar to anomaly detection, we envisage individual Forecasting microservices for different types of forecasts, for example, used for different power consumers. Forecasting benefits notably from the microservice pattern since technologies used for forecasting often differ significantly from the ones used for implementing web systems. The Titan Control Center supports arbitrary Forecasting microservices, each using its own technology stack. The only requirement for a Forecasting service is that it is able to communicate with other services via the messaging system.

Our pilot implementation already features a microservice that performs forecasts using an artificial neural network with TensorFlow [ABC+16]. This neural network is trained offline using historical data and mounted into the microservice at start-up. During operation, the Forecasting microservice subscribes to the stream of measurements (again monitored or aggregated) and feeds each incoming measurement into the neural network. The forecast results are stored in an OpenTSDB [Ope22] time series database [GJK+14] and published to a dedicated stream via the messaging system.

In a first proof of concept, we built and trained such neural networks together with the newspaper printing company. We selected a set of machines in the company with different power consumption patterns and trained individual networks per machine. These neural networks use not only the historical power consumption of their machines as input, but also the power consumption of other machines as well as environmental data, such as the outside temperature. We deploy individual instances of our

Forecasting microservice for each neural network, allowing for individual forecasts of each machine.

10.6.7 Visualization

As suggested in Section 10.5.7, the Titan Control Center features web applications for visualizing power consumption data. Since visualization serves as a measure to integrate the results of other measures, we also regard the visualization software components as integration of the individual analysis microservices. The Titan Control Center provides two single-page applications for visualization: a graphical user interface, tailored to the specific functions of the Titan Control Center, and a dashboard for simple, yet highly adjustable data visualizations. In the following, we describe both applications and their corresponding use cases.

Control Center The Titan Control Center user interface serves to provide a consistent access to all functionalities of the Titan Control Center. This includes visualizing the analysis results of microservices, but also control functions for configuring microservices. The user interface is implemented with Vue.js [You22] and D^3 [BOH11].

Fig. 10.6 shows a screenshot of the Titan Control Center's summary view. It consists of several components which collect and show the individual analysis results for the entire production. A time series chart displays the power consumption in course of time. This chart is interactive, allowing zooming and shifting the displayed time interval. Colored arrows indicate how the power consumption evolved within the last hour, the last 24 hours, and the last 7 days. A histogram shows a frequency distribution of metered values serving to detect potential for load peak reduction. A pie chart breaks down the total power consumption into subconsumers. Line charts display the average course of power consumption over the week or the day, as provided by the Statistics microservice (see Section 10.6.3). The visualizations are periodically updated with new data. This causes, for example, the time series diagram to shift forward continuously and the arrows to change color and direction.

Apart from this summary view, our pilot implementation also provides the described types of visualization for individual machines and groups

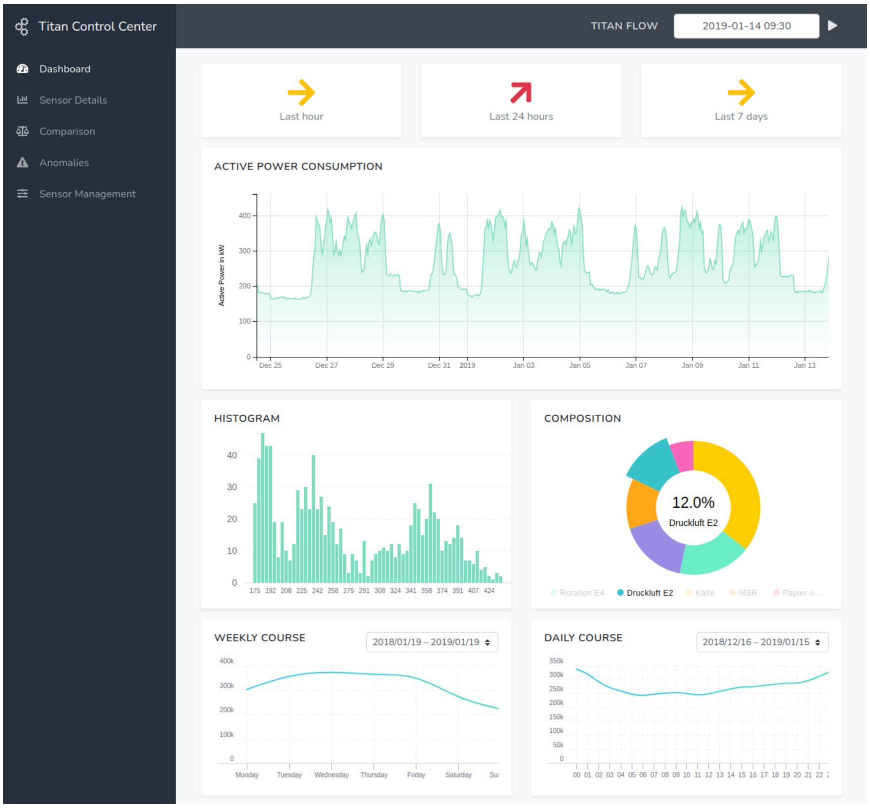


Figure 10.6. Screenshot of the Titan Control Center [HHB+21].

of machines. Starting from an overview of the total power consumption, a user can thus navigate through the hierarchy of all consumers. Furthermore, the single-page application allows graphically correlating data (see Section 10.6.4) and configuring machines and machine groups maintained by the Sensor Management service.

Dashboard The second application is a pure visualization dashboard implemented with Grafana [Gra22] (see Fig. 10.7). It provides a set of

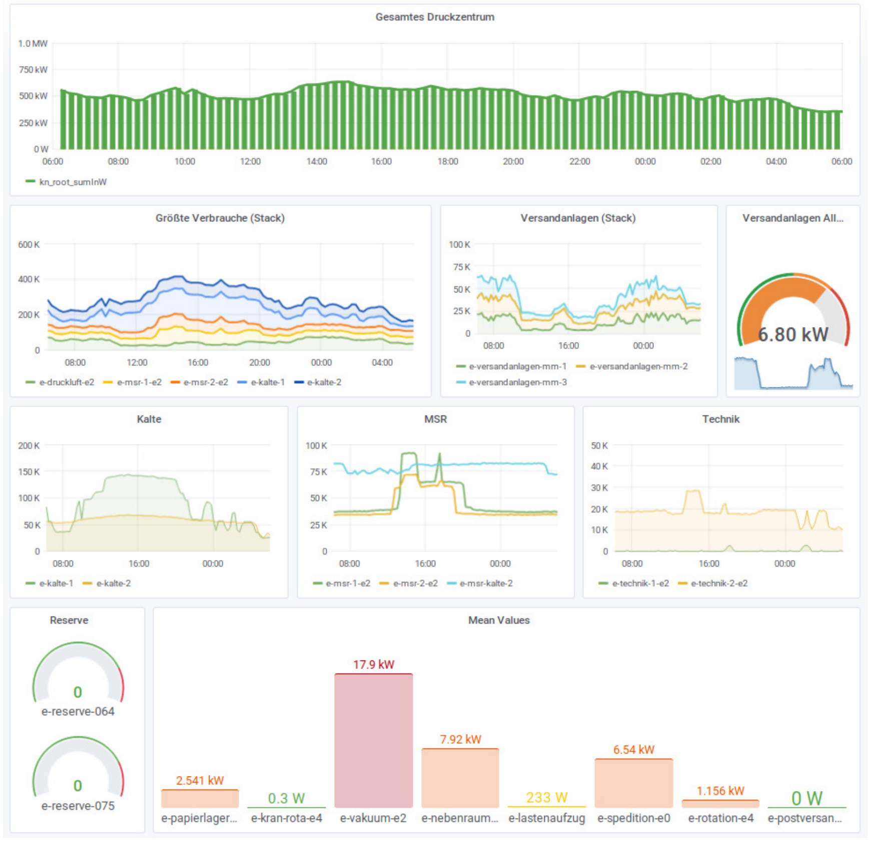


Figure 10.7. Screenshot of the Titan dashboard implemented with Grafana [Wet19].

common visualizations such as line charts, bar charts, and gauges. As presented in Fig. 10.7, we mainly display time series charts as bar or line charts. The dashboard is highly adjustable, meaning that users can add, modify, and rearrange chart components. Such adjustments can be performed graphically and only require usage of provided interfaces. Thus, especially IT savvy production operators can customize dashboards. Moreover, they can create own dashboards and share them among users.

In this way individual dashboards, for example, for management and production operators can be implemented.

In contrast to the Control Center, this dashboard does not provide any control functions (e.g., for sensor configuration) and no complex interactive visualizations (e.g., the comparison tool). Thus, it only serves as an extension to the Control Center, allowing for visual data analysis and reporting. In particular, this dashboard covers use cases, where power consumption data should be integrated in existing dashboards (as it is the case in one studied enterprise) or if dashboards should be customized by production operators.

10.6.8 Alerting

Alerting in the Titan Control Center is implemented using the Titan Flow Engine in the Integration component. All messages that are published to the messaging system can again be consumed by the Titan Flow Engine and processed in flows. This way, production operators can create and adjust alerting flows directly within the production environment. Our pilot implementation already provides a flow that sends an email whenever an anomaly in power consumption is reported. In dedicated bricks, the operator can filter the types of anomaly an alert should be generated for and configure how the email should be sent (e.g., message and receiver). The Flow Engine allows modeling flows that perform arbitrary actions in the production environment when alerts are received. This includes communications with machines again, for example, to show alerts on machine monitors.

Chapter 11

Scalability Benchmarks for Event-Driven Microservices

This chapter presents our Theodolite scalability benchmarks for event-driven microservices. It builds upon our work previously published in the *Big Data Research* journal [HH21e] and presented at the *ACM/SPEC International Conference on Performance Engineering 2021* [HH21a].

With this chapter, we address research questions RQ 2.2–2.4 of this thesis. Section 11.1 addresses research question RQ 2.2 and presents four dataflow architectures derived from typical use cases for event-driven microservices based on Chapter 10. Section 11.2 discusses relevant load types and Section 11.3 discusses relevant resource types. Thus, they address research question RQ 2.3. Section 11.4 presents SLOs for distributed stream processing and, hence, addresses research question RQ 2.4. Afterward, Section 11.5 gives an overview of our benchmark implementations for state-of-the-art stream processing frameworks and Section 11.6 discusses related work on stream processing benchmarks.

11.1 Dataflow Architectures

In this section, we identify four use cases of different complexity for event-driven microservices. Our use cases are derived from the Titan Control Center as presented in Section 10.6. Although addressed to analyzing industrial power consumption data, we suppose that these use cases also occur in other application domains. For each use case, we present a corresponding dataflow architecture (often referred to as operator graph or topology). Our presented architectures do not follow a specific dataflow

Table 11.1. Overview of dataflow characteristics observed in use cases.

Dataflow characteristics	UC1	UC2	UC3	UC4
Stateless operations	✓	✓	✓	✓
Tumbling window aggregations	✗	✓	✗	✓[a]
Sliding window aggregations	✗	✗	✓	✓[a]
Joins of different streams	✗	✗	✗	✓
Feedback loops	✗	✗	✗	✓

[a] Use case UC4 can be configured either to use tumbling windows or sliding windows.

model as different stream processing frameworks use different models [ABC+15; SWW+18]. However, most of these models are similar, allowing the described architectures to be implemented in most modern frameworks.

All our use cases share that they receive all data from a messaging system and publish all processing results back to that messaging system. We assume all input messages to be measurements from IIoT sensors consisting of the corresponding sensor identifier, a timestamp and the actual measurement. All messages are keyed by the sensor identifier. The dataflow architectures presented below focus only on required processing steps. In practice, microservices fulfilling these use cases are likely to contain additional processing steps, for example, for filtering and transforming intermediate data. Table 11.1 lists typical dataflow characteristics in stream processing and shows in which use cases these characteristics apply.

11.1.1 Use Case UC1: Database Storage

A simple, but common use case in event-driven architectures is that events or messages should be stored permanently, for example, in a NoSQL database. Using this database, an application can provide its data via an API as it is the case in Lambda and Kappa architectures [Lin17]. In the Titan Control Center, for example, the `Statistics` and the `History` microservices store processed data this way (see Section 10.6).

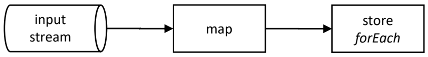


Figure 11.1. Dataflow architecture for UC1: Database Storage [HH21e].

A dataflow architecture for this use case is depicted in Fig. 11.1. The first step is to read data records from a messaging system. Then, these records are converted into another data format in order to match the often different formats required by the database. Finally, the converted records are written to an external database. Depending on the required processing guarantees, this is done either synchronously or asynchronously. Unless the interaction between database and stream processing framework should be benchmarked, we suggest not including a real database in implementations of these benchmarks. Otherwise, due to the simple, stateless stream processing topology, the benchmark would primarily test the database's write capabilities. See Chapter 15 for a benchmarking study with database writes.

11.1.2 Use Case UC2: Downsampling

A very common use case for stream processing architectures is downsampling, meaning to aggregate multiple messages within consecutive, non-overlapping time windows (called tumbling windows [CKH19]). Typical aggregations compute the average, minimum, or maximum of measurements within a time window or count the occurrence of same events. Such reduced amounts of data are required, for example, to save computing resources or to provide a better user experience (e.g., for data visualizations). With such window aggregations, the (potentially varying) message frequency of an input stream is reduced to a constant value. This is required for applying many machine learning methods, which require data of a fixed frequency. In the Titan Control Center, the History microservice continuously downsamples IIoT monitoring data and provides these data for other microservices (see Section 10.6).

A dataflow architecture for this use case is presented in Fig. 11.2. It first reads measurement data from an input stream and then assigns each measurement to a time window of fixed, but statically configurable size.

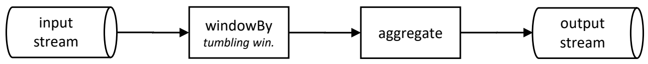


Figure 11.2. Dataflow architecture for UC2: Downsampling [HH21e].

Afterward, an aggregation operator computes the summary statistics sum, count, minimum, maximum, average, and population variance for a time window. Finally, the aggregation result containing all summary statistics is written to an output stream.

11.1.3 Use Case UC3: Time-Attribute-Based Aggregation

A second type of temporal aggregation is aggregating messages that have the same time attribute. Such a time attribute is, for example, the hour of day, day of week, or day in the year. This type of aggregation can be used to compute, for example, an average course over the day, the week, or the year. It allows demonstrating or discovering seasonal patterns in the data. The Statistics microservice of the Titan Control Center implements this use case (see Section 10.6).

This use case differs from UC2 in that the time attribute has to be extracted from the record's timestamp, whereas in UC2 the timestamp needs no further interpretation. Moreover, in this use case, multiple aggregations have to be performed in parallel (e.g., maintaining intermediate results for all 7 days of the week). Thus, the amount of different output keys increases by the factor of possible different time attributes. For example, when computing aggregations based on the day of week for a data stream with n different keys, the result stream contains data of $7n$ different keys.

In practice, not all messages that have ever been recorded should be considered in the aggregation, but usually only those of a certain past time period. For example, in industrial facilities, operators are interested in the average course of energy consumption over the day within the last 4 weeks. They do probably not want to include older data as the average course might change over time, for example, due to changing process planning and varying load over the year. Therefore, the aggregation based on time attributes is performed on a sliding window [CKH19], which only considers data of a fixed time period. In contrast to the tumbling

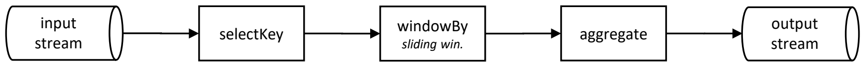


Figure 11.3. Dataflow architecture for UC3: Aggregation based on time attributes [HH21e].

window aggregation in UC2, this use case additionally requires computing results for multiple overlapping time windows, which further increases the amount of output data.

Fig. 11.3 depicts a dataflow architecture for this use case. The first step is to read measurement data from the input stream. Then, a new key is set for each message, which consists of the original key (i.e., the identifier of a sensor) and the selected time attribute (e.g., day of week) extracted from the record's timestamp. In the next step, the message is duplicated for each sliding window it is contained in. Then, all measurements of the same sensor and the same time attribute are aggregated for each sliding time window by computing the summary statistics sum, count, minimum, maximum, average, and population variance. The aggregation results per identifier, time attribute, and window are written to an output stream. Optionally, this dataflow architecture can be configured to also periodically emit intermediate results.

11.1.4 Use Case UC4: Hierarchical Aggregation

For analyzing sensor data, often not only the individual measurements of sensors are of interest, but also aggregated data for groups of sensors. When monitoring energy consumption in industrial facilities, for example, comparing the total consumption of machine types often provides better insights than comparing the consumption of all individual machines. Additionally, it may be necessary to combine groups further into larger groups and adjust these group hierarchies at runtime. A detailed description of these requirements, supplemented with examples, is provided in our previous publications [HH19b; HH20b]. In the Titan Control Center, the `Aggregation` microservice hierarchically aggregates IIoT data this way (see Section 10.6).

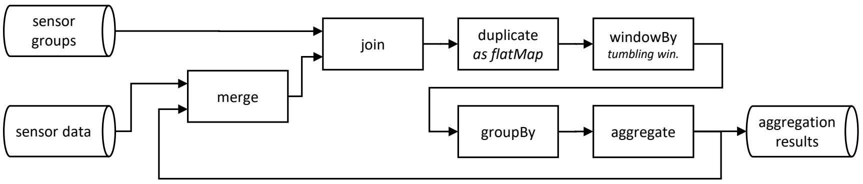


Figure 11.4. Dataflow architecture for UC4: Hierarchical Aggregation [HH21e].

Fig. 11.4 depicts a dataflow architecture for the use case of hierarchically aggregating data streams. The dataflow architecture requires two input data streams: a stream of sensor measurements and a stream tracking changes to the hierarchies of sensor groups. In the consecutive steps, both streams are joined, measurements are duplicated for each relevant group, assigned to time windows, and the measurements for all sensors in a group per window are aggregated. Finally, the aggregation results are exposed via a new data stream. Additionally, the output stream is fed back as an input stream in order to compute aggregations for groups containing subgroups. See our previous publication [HH20b] for a detailed architecture description. To also support unknown record frequencies, this dataflow architecture can be configured to use sliding windows instead of tumbling windows [HH20b].

11.2 Load Types

Load on a stream processing application implementing the previously presented dataflow architectures can be scaled in various dimensions. The most important load type is scaling with the number of distinct simulated sensors. All our proposed benchmarks use the sensor identifier as key on the input topic and keep it as part of the key within all dataflow architectures. Thus, within all architectures, processing is parallelized on the key. For use case UC4, the number of sensors is implicitly defined by the structure of the sensor hierarchy. Assuming a uniformly shaped hierarchy tree, scalability could be evaluated regarding the number of sensors within a group and the depth of nesting groups in groups. Due to our proposed benchmarking method and our associated Theodolite framework, supported load types do not have to be configured as part of the benchmark's implementation. Instead, every scalable property of a benchmark deployment can serve as load type. Other load types are, for example, the frequency of simulated sensors sending messages or the payload of messages. Scalability can also be examined regarding the complexity of operations within a dataflow architecture. For example, in use case UC3 one could scale with the size of windows while keeping the volume of input messages constant.[1]

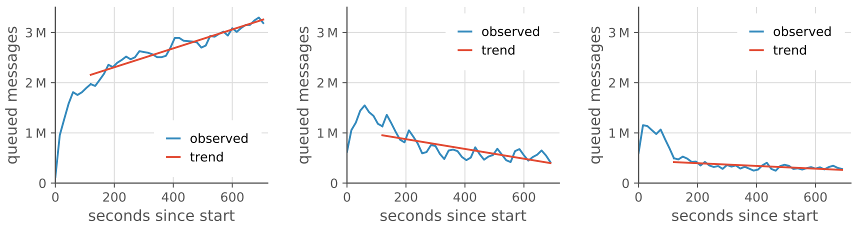
11.3 Resource Types

All modern stream processing frameworks as discussed in Section 4.4 use parallelization as the primary method to scale with increasing load. To exploit parallelization, either the number of instances or the number of CPUs and memory provided per instance can be scaled. For the latter, many stream processing frameworks also require scaling the size of thread pools and similar configurations accordingly. Thus, the number of instances and the resources per instance are also the most relevant resource dimensions for benchmarking scalability. However, thanks to our proposed benchmarking method, other resource dimensions can be evaluated as well.

11.4 Service Level Objectives

We propose two SLOs for evaluating streaming processing frameworks. The primary SLO is the consumer lag trend SLO, indicating whether the number of unprocessed and queued messages increases over time. In the case of dataflow architectures containing window aggregation, we propose to complement it with an SLO, capping the amount of discarded records per second.

[1]This expands our benchmarking method to consider workload instead of load as input variable. Workload is often defined as the product of load and work with work being the amount of operations that are performed for each incoming message [BLB15; BSL16].



(a) 6 SUT instances **(b)** 7 SUT instances **(c)** 8 SUT instances

Figure 11.5. Illustration of the consumer lag trend metric for an exemplary benchmark execution (Theodolite's UC3 benchmark implemented with Kafka Streams and 50 000 messages/second) with different numbers of SUT instances. Independent of the number of instances, we can observe a variable lag. However, computing a trend line (without considering the measurements from an initial warmup period), reveals that for 6 instances, the number of queued messages will steadily increase over time. Providing 7 instances leads to a decrease in queued messages after the warmup period, while 8 instances yields an almost constant trend line.

11.4.1 Consumer Lag Trend SLO

The consumer lag of a stream processing job describes how many messages are queued in the messaging system, which have not been processed yet. Our consumer lag trend metric describes the average increase (or decrease) of the lag per second. It can be measured by monitoring the lag and computing a trend line using linear regression. The slope of this line is the lag trend. Fig. 11.5 illustrates the concept of the lag trend.

We use the lag trend metric to define an SLO, whose function valuates to true if the lag trend does not exceed a certain threshold. Ideally, this threshold should be 0 as a non-positive lag trend means that messages can be processed as fast as they arrive. However, it could make sense to allow for a small increase since even when observing an almost constant lag, a slightly rising or falling trend line will be computed due to outliers.

We observed that in most cases, checking the lag trend alone suffices as an SLO. The architectures of modern stream processing frameworks make it unlikely that SLOs such as a maximum tolerable processing latency can be fulfilled by increasing the degree of parallelism. An advantage of defining an SLO based on the lag is that it can be collected very efficiently from the messaging system brokers. It does require data from the stream processing frameworks, which might not reliably update their metrics anymore under high load, leading to incorrect SLO results.

11.4.2 Dropped Records SLO

In certain cases, we observed that under high load the consumer lag does not substantially increase, but records were discarded due to lateness. In most stream processing frameworks, operations on time windows still accept out-of-order records for a configurable amount of time. If this time has elapsed, records are discarded and not further processed. Thus, records are still consumed from the messaging system, not causing a consumer lag increase, but results become incorrect. We observed this particularly for implementations of use case UC4 containing multiple repartitionings and a feedback loop.

To consider these cases when evaluating an SUT for a certain load and resource combination, we propose to include a second SLO, which sets a maximum allowed number of discarded records per second. In line with the corresponding metrics provided by stream processing frameworks, we refer to this SLO as dropped records SLO. As the metrics are provided by the stream processing framework, which might be incorrect under high load, it is important to only use this SLO in addition to the lag trend SLO.

11.5 Systems under Test and Load Generation

Along with our Theodolite benchmarking framework (see Section 8.5), we provide open-source implementations of the dataflow architectures presented in this section with Apache Flink, Hazelcast Jet, Apache Kafka Streams, and Apache Beam.[2] The implementations of the latter are available with the Apache Flink and the Apache Samza runners. Using other Beam runners as we do in Chapter 15 with Google Cloud Dataflow, requires only little effort.

[2] `https://www.theodolite.rocks/theodolite-benchmarks/`

In addition to the implementations of the dataflow architectures, we also provide a corresponding load generator for each use case. It is configurable by the number of simulated sensors and with the frequency each simulated sensor sends records. Multiple instances of the load generator can be started, which automatically form a cluster, perform a leader election, and divide the amount of data to be generated among themselves.

All our task sample implementations and the load generators are configured to use Apache Kafka as messaging system to read and write data from. Simulated sensor measurements are defined and encoded with the Apache Avro serialization format. For some implementations, we also integrate alternative transport mechanism as shown in Chapter 15. As required by Aderaldo et al. [AMP+17] (see Section 4.2.3), both task sample implementations and the load generators are continuously tested and built, as well as packaged and published as container images. We also provide Kubernetes manifests to deploy our implementations to a Kubernetes cluster.

For each task sample implementation, we provide a corresponding benchmark for our Theodolite benchmarking framework (see Section 8.5). These benchmarks consist of the Kubernetes manifests for the task sample implementation (SUT) and the load generator, configurations of Kafka topics, a load type defining the number of simulated sensors, two resource types defining the number of instances and the number resources per instances, and Theodolite definitions of our proposed SLOs.

11.6 Related Work

Over the last few years, a couple of benchmarks for stream processing frameworks have been proposed. In the following, we give an overview of such benchmarks and relate them to our Theodolite benchmark for event-driven microservices. Table 11.2 summarizes characteristics of the discussed benchmarks.

StreamBench [LWX+14] is one of the earliest benchmarks for modern stream processing frameworks. While originally only implemented for Spark and Storm, it has later been used to benchmark Apache Apex, Beam, Flink, and Samza as well [HMG+19; QWH+16]. As its name suggests,

Table 11.2. Overview of the characteristics and implementations of stream processing benchmarks.

			Messaging			Stream processing framework								Cloud-native		
Benchmark	Task samples	Open source	Kafka	Others	None	Flink	Spark	Storm	Samza	Kafka Streams	Hazelcast Jet	Others	Database	Containers	Kubernetes	Others
Theodolite (this thesis)	4	✓	✓			✓			✓[b]	✓	✓	✓[b]	[a]	✓	✓	
Beam Nexmark [Apa22]	13	✓	✓	✓		✓[b]	✓[b]		?[b]		?[b]	✓[b]				
ESPBench [HMP$^+$21]	5	✓	✓			✓[b]	✓[b]		?[b]		✓[b]	?[b]	✓			
OSPBench [vDvdP20]	5	✓	✓			✓	✓			✓				✓		✓
DSPBench [BGM$^+$20]	5	✓	✓				✓	✓					✓			
Shahverdi et al. [SAS19]	1	✓	✓			✓	✓	✓		✓	✓		✓			
Karimov et al. [KRK$^+$18]	2				✓	✓	✓	✓								
RIoTBench [SCS17]	4[c]	✓		✓				✓					✓			
YSB [CDE$^+$16]	1	✓	✓			✓	✓	✓					✓			
SparkBench [LTW$^+$15]	10	✓			✓		✓									
StreamBench [LWX$^+$14]	7		✓				✓	✓								
Linear Road [ACG$^+$04]	5				✓							✓				

[a] optional
[b] using Apache Beam
[c] RIoTBench's 4 application benchmarks are composed of 27 microbenchmarks

SparkBench [LTW+15] is a benchmark tailored to Apache Spark. The Yahoo Streaming Benchmark (YSB) [CDE+16] is frequently used and adapted in research [YJH+17; KYA17; NNG19; ZMK+19; CYH20; vDvdP20]. Worth highlighting is the work of Shahverdi et al. [SAS19], who extend YSB with implementations for the frameworks Kafka Streams and Hazelcast Jet. As discussed in Section 4.4, these frameworks are particularly suited for building event-driven microservices. RIoTBench [SCS17] provides four application benchmarks for Storm composed of 27 small task samples. Nasiri et al. [NNG19] adopt RIoTBench for Flink and Spark. Karimov et al. [KRK+18] present a benchmark with two task samples, derived from a real industrial context, yet without providing open-source implementations.

More recently, DSPBench [BGM+20], OSPBench [vDvdP20; vDon21], and ESPBench [HMP+21; Hes22] have been proposed. DSPBench contains 15 benchmarks, which resample typical stream processing applications, derived from reviewing the literature. OSPBench provides benchmarks for analyzing traffic sensor data. Besides evaluations of latency, throughput, and resource usage, van Dongen and van den Poel used OSPBench to also evaluate scalability [vDvdP21b] and fault recovery [vDvdP21a]. In contrast to most other benchmarks, OSPBench provides implementations for the rather new framework Kafka Streams, which is also intensively studied in this thesis. The Enterprise Stream Processing Benchmark (ESPBench) builds upon the Senska benchmark [HRM+18]. It is special in the sense that it integrates a relational database management system. In contrast to most other benchmarks, ESPBench's task samples are implemented with Apache Beam. While Hesse et al. [HMP+21] only perform evaluations with Spark, Flink, and Hazelcast Jet, we expect that also other Beam runners can be used to run the benchmark.

The Nexmark benchmark [TTP+10] has originally been proposed as the *Niagara Extension to the XMark benchmark* addressed to first-generation stream processing systems (see Section 4.4). The Apache Beam community adapted and extended Nexmark with implementations for Beam to benchmark the performance of different runners [Apa22]. Documentation and benchmark results are provided for the direct runner as well as for the Flink, the Spark, and the Google Cloud Dataflow runners. However, running the benchmark with other runners should be possible as well. Recently, there seems to be an effort to implement the Nexmark task samples with other frameworks in an open-source project.[3] However, currently this project only provides implementations for Apache Flink. Moreover, Gencer et al. [GTĎ+21] implemented the Nexmark benchmark for their performance evaluation of Hazelcast Jet.

Worth mentioning is also the Linear Road benchmark presented by Arasu et al. [ACG+04]. Although published years before all modern stream processing frameworks considered in this thesis (see Section 4.4) were released, it is still used in research [ZHD+17; ZMK+19; Sax20] and compared to newer benchmarks [BGM+20; HMP+21]. Pagliari et al. [PHU20] and Garcia et al. [GGS+22a; GGS+22b] present approaches to generate benchmarks.

Table 11.2 focuses on publications presenting new benchmarks. The differentiation between benchmarks and benchmarking or experimental studies is sometimes blurry. Many publications that present benchmarks perform also an experimental study with them. On the other hand, many experimental studies utilize existing benchmarks, but modify them. From Table 11.2, we can see that a lot of open-source benchmarks have been proposed. Apart from our Theodolite benchmarks, none of these benchmarks is particularly addressed to scalability. Often originating in data management research, many benchmarks are defined as "queries" over data streams [TTP+10; KRK+18; HMP+21]. Most benchmarks include a messaging system as a middleware component between workload generation and stream processing framework. In the vast majority of cases, this is Apache Kafka. Karimov et al. [KRK+18] exclude such a system to not let it become the benchmark's bottleneck. Our Theodolite benchmarks purposely include Kafka to represent more realistic event-driven microservice deployments. Flink, Spark, and Storm are by far the most supported frameworks. Only a few benchmarks exist for Samza, Kafka Streams, and Hazelcast Jet, which are frameworks particularly suited for implementing event-driven microservice. Our Theodolite benchmarks are the only ones providing implementations for all of them. While some benchmarks include an interaction with a database in their setup, others do not. With our benchmarks, a database can optionally be used as we do in Chapter 15. Besides our Theodolite benchmarks, there is only one

[3] `https://github.com/nexmark/nexmark`

other benchmark (OSPBench) that is provided as container images to be used in a cloud-native setting. No other benchmark provides Kubernetes manifests.

Karimov et al. [KRK$^+$18] highlight shortcomings of several experimental performance evaluations of stream processing frameworks. They define the sustainable throughput metric, which is the highest load that a system can handle without continuously increasing latency. Imai et al. [IPV17] defined this metric in a similar form. Venugopal and Theobald [VT20], Chu et al. [CYH20], and van Dongen and van den Poel [vDvdP20] perform experimental evaluations with it. Sustainable throughput can be seen as a specific form of our load capacity metric with the lag trend SLO.

Part IV

Experimental Evaluation

Chapter 12

Evaluating Variability of Benchmark Results in the Cloud

In this chapter, we perform an experimental evaluation of our proposed scalability benchmarking method. Specifically, we empirically evaluate the effect of our benchmarking method's configuration options (see Section 7.5) on the statistically grounding of its results. The overarching goal of this evaluation is to find configuration parameters such that the results are reproducible, while the overall execution time is kept as short as possible. Hence, we state the following evaluation questions:

EQ 1 For how long should SLO experiments be executed?

EQ 2 How many repetitions of such SLO experiments should be performed?

EQ 3 How does the assessment of SLOs evolve with increasing resource amounts?

EQ 4 How does the assessment of SLOs evolve with increasing load intensities?

We conduct our experiments for multiple SUTs, which implement different benchmarks, employ different software frameworks, and run in different cloud environments. This way, we also seek to find out whether the choice of configuration parameters should depend on the cloud provider, implementation, or benchmark. In this evaluation, we focus on our Theodolite benchmarks for event-driven microservices. However, our evaluation method is also intended to serve as a blueprint to repeat our evaluation for other SUTs.

This chapter builds upon our work previously published in the *Empirical Software Engineering* journal [HH22a]. After a detailed description of our experimental setup in Section 12.1, we conduct the following evaluations:

- In Section 12.2, we address EQ 1 and study the duration SLO experiments are executed for as well as their warm-up period duration. We evaluate how both durations should be chosen such that we can decide with sufficiently high confidence whether evaluated SLOs are achieved.
- In Section 12.3, we address EQ 2 and evaluate how many repetitions of an SLO experiment should be performed to decide with sufficiently high confidence whether evaluated SLOs are achieved.
- In Section 12.4, we address EQ 3 and evaluate how the assessment of SLOs evolves with increasing resource amounts. This evaluation helps in determining whether the binary search strategy can be applied with our *demand* metric and whether the lower bound restriction strategy can be applied with our *capacity* metric.
- In Section 12.5, we perform a similar evaluation to address EQ 4 and evaluate how the assessment of SLOs evolves with increasing load intensities. This evaluation helps in determining whether the binary search strategy can be applied with our *capacity* metric and whether the lower bound restriction strategy can be applied with our *demand* metric.

In all four sections, we first describe the employed experiment design, before we present and discuss the experiment results. Finally, we discuss threats to validity in Section 12.6.

12.1 Experimental Setup

In the following, we present the general experimental setup for the following evaluations. For all experiments, we use our Theodolite scalability benchmarking framework (see Section 8.5) with our Theodolite benchmarks for event-driven-microservices (see Chapter 11). We consider 4 benchmarks, which are implemented by 2 stream processing frameworks and executed in 3 cloud environments. This results in 24 SUTs as summarized in Table 12.1.

Table 12.1. Overview of SUTs studied in our evaluation

SUT component	Evaluated options	Σ
Benchmarks		
Task samples	Theodolite's UC1, UC2, UC3, UC4	4
Load type	Messages with distinct keys per second	1
Resource type	Number of instances (Kubernetes pods)	1
SLO	Consumer lag trend	1
Frameworks	Kafka Streams, Flink	2
Cloud providers	Google (GCP), Oracle (OCI), private cloud (SPEL)	3
Total number of SUTs		24

12.1.1 Benchmark Implementations and Configurations

We evaluate implementations of our Theodolite benchmarks with the stream processing frameworks Apache Flink and Apache Kafka Streams. Specific configurations of our dataflow architectures and the stream processing frameworks are aligned with our pilot evaluations of a previous publication [HH21e]. For benchmark UC1, UC2, and UC3, the load type corresponds to the number of keys, where for each key, we generate one message per second.

For benchmark UC4, the load type is the number nested groups n, which results in 4^n keys, generating one message per second. For reasons of conciseness, we present the generated load also as these 4^n messages per second in the results tables of our evaluation.

The resource type used in our evaluations is the number of instances of services. For Kafka Streams, this is simply the amount of pods, containing the same Kafka Streams application. All necessary coordination among instances to distribute tasks and data is then handled by the Kafka Streams framework. For Flink, the resource type is the amount of TaskManager pods. Additionally, an environment variable has to be set, which notifies the Flink instances about the desired parallelism, which in our case corresponds to the amount of TaskManager pods. Since we do not benchmark for fault tolerance, Flink's coordinating JobManager pod is not scaled.

Table 12.2. Configuration of Kubernetes clusters used for our evaluation, running at Google Cloud Platform (GCP), Oracle Cloud Infrastructure (OCI), and a private cloud (SPEL).

	GCP	OCI	SPEL
Nodes	3	3	5
CPU cores	4	4	2×16
RAM	16 GB	16 GB	384 GB
Machine type	e2-standard-4	VM.Standard.E2.4	Intel Xeon Gold 6130
Kubernetes	1.19.9-gke.1900	1.19.7	1.18.6
Kafka brokers	3	3	10

As discussed in Section 11.4, the consumer lag trend is the most important SLO. Hence, we focus on this SLO in the evaluations of this chapter.

12.1.2 Evaluated Cloud Platforms

The experimental evaluations presented in this section are performed at two public and one private cloud platforms. The two public cloud vendors are Google Cloud Platform (GCP) and Oracle Cloud Infrastructure (OCI), where we rely on the managed Kubernetes services with virtual machine nodes. We chose Google Cloud Platform as it is one of the largest cloud providers, whose Kubernetes offering can be regarded as most matured since Google significantly leads the Kubernetes development. Oracle Cloud Infrastructure is representative of a niche cloud provider, which provides a less sophisticated managed Kubernetes service. As private cloud infrastructure, we chose the Software Performance Engineering Lab (SPEL) at Kiel University.[1] In contrast to the public clouds, its Kubernetes cluster runs on 5 bare metal nodes with considerably more powerful hardware. Besides also representing a realistic deployment platform used in many industries, the private cloud serves as a reference, ruling our the influence of public cloud performance peculiarities. Table 12.2 summarizes the configuration of the Kubernetes clusters, we use in our evaluation.

[1] `https://www.se.informatik.uni-kiel.de/en/research/software-performance-engineering-lab-spel`

12.1.3 Replication Package

We provide a replication package and the collected data of our experiments as supplemental material [HH21b], allowing other researchers to repeat and extend our work. Our replication package includes the Theodolite Executions (see Section 8.3) used in our experiments and interactive notebooks used for analyzing our experiment results.

12.2 Evaluation of Warm-up and SLO Experiment Duration

Our scalability measurement method and, thus, our proposed benchmarking tool architecture is configurable by the duration, SLO experiments are executed for, and by the duration that is considered as warm-up period. We address EQ 1 and evaluate how the choice of warm-up period and duration influences the result of the SLO experiments. Goal of this evaluation is to minimize the experiment duration, without substantially scarifying the quality of the results.

12.2.1 Experiment Design

With this evaluation, we perform SLO experiments for all 24 SUTs depicted in Table 12.1. For each SUT, we aim to select SLO experiments, in which the resource amounts approximately correspond to the resource demand of the load. Those are probably the most difficult to assess, while combinations of high load and few resources or vice versa are likely to require less time to be evaluated.

We performed preliminary experiments to find reasonable combinations of load and resources to evaluate. For each SUT, we determine an approximation of the load that can be handled by 4–5 instances. We found those instance counts to be reliably working in all cloud platforms. To find suitable load-resource combinations, we run single, explorative experiments for a short time and manually observe the lag via Theodolite's dashboard [HH21e]. These results are not statistically grounded and do not necessarily represent the real resource demand. Instead, they represent a deployment, in which the provisioned resources approximately match the resource demand to bootstrap the following experimental evaluations. In addition to the resource amounts that approximately match the demand of a load, we perform experiments for one instance more and less, representing a slight over or underprovisioning. In our private cloud environment, we additionally perform these experiments with loads twice as high. For this case we find approximately matching instance counts as well, but due to higher instance numbers, we use two instances more and less to represent over or underprovisioning. Table 12.3 shows the load intensities and resource amounts that we use for the following evaluation.

To obtain an approximation of the true, long-term lag trend, we perform the SLO experiments in this evaluation over a period of one hour. According to our measurement method and the lag trend SLO, we let Theodolite monitor the lag during this time. For each experiment, we compute the lag trend over the entire experiment duration with different warm-up periods. The computed lag trends serve as reference values, used in the following approach to reduce the experiment duration.

For each experiment and evaluated warm-up period, we now evaluate how much shorter the experiment duration can be chosen such that the result of the SLO evaluation does not deviate from the reference value. For this purpose, we retroactively reduce the experiment duration by discarding the latest measurements. We evaluate two options as decision criterion for when no further measurements should be discarded:

1. We reduce the duration as long as the computed trend slope does not deviate by more than a certain error from the reference value.

2. We reduce the duration as long as the binary result of the SLO evaluation does not change. More specifically, we first determine whether the reference values exceeds a threshold t. Then, we reduce the duration as long as the lag trend does not rises above t or falls below t.

Our replication package [HH21b] allows evaluating our experiment results according to the described method for different warm-up durations, allowed errors, and lag trend thresholds.

Table 12.3. Chosen load intensities and resource amounts for the SUTs, used for the evaluation of experiment duration and repetition count. The load type corresponds to messages per second. Resource amounts are numbers of instances, representing underproviding ($\text{inst}^{\vee}$), overprovisioning ($\text{inst}^{\wedge}$), and resources that approximately match the demand ($\text{inst}^{\approx}$).

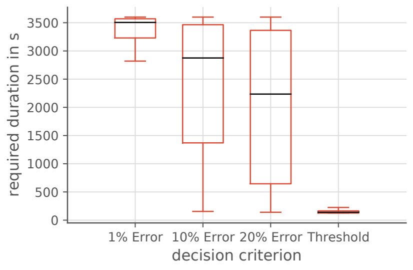
SUT			load	resource amounts		
cloud	framework	bench.	mes./s	$\text{inst}^{\vee}$	$\text{inst}^{\approx}$	$\text{inst}^{\wedge}$
GCP	Flink	UC1	200 000	4	5	6
		UC2	150 000	3	4	5
		UC3	60 000	5	6	7
		UC4	65 536	1	2	3
	Kafka Streams	UC1	300 000	4	5	6
		UC2	150 000	5	6	7
		UC3	20 000	4	5	6
		UC4	65 536	4	5	6
OCI	Flink	UC1	200 000	4	5	6
		UC2	150 000	3	4	5
		UC3	60 000	4	5	6
		UC4	65 536	1	2	3
	Kafka Streams	UC1	300 000	5	6	7
		UC2	150 000	4	5	6
		UC3	30 000	5	6	7
		UC4	65 536	4	5	6
SPEL	Flink	UC1	300 000	3	4	5
			600 000	8	10	12
		UC2	150 000	3	4	5
			300 000	6	8	10
		UC3	60 000	2	3	4
			240 000	8	10	12
		UC4	65 536	1	2	3
	Kafka Streams	UC1	300 000	4	5	6
			600 000	5	7	9
		UC2	150 000	3	4	5
			300 000	7	9	11
		UC3	30 000	4	5	6
			60 000	8	10	12
		UC4	65 536	3	4	5

12.2.2 Results and Discussion

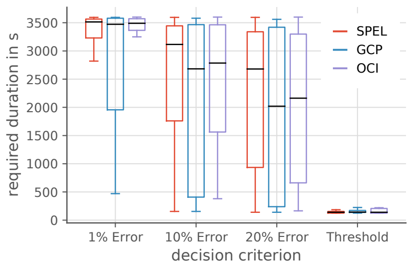
Our results show that when using a maximum allowed error as decision criterion, the time required to reach a stable value decreases with increasing allowed error. Fig. 12.1 illustrates this for errors of 1%, 10%, and 20% with a warm-up duration of 120 s. However, we observe the same trend also for other errors and warm-up durations. We cannot identify a significant impact of the cloud provider or the stream processing framework on the required execution time. Our results suggest that more complex stream processing benchmarks require shorter execution times, but this would need further experiments.

When using a maximum allowed error as decision criterion, we observe that even with an allowed error of 20%, the required execution time remains excessively high: For example with the data presented in Fig. 12.1, more than 50% of the experiments, require more than half an hour execution time for a single SLO experiment. Referring to the runtime formula of our method (see Section 7.5), this would quickly lead to a total runtime of several days. On the other hand, the time required to decide whether the lag trend exceeds a threshold is significantly lower, independently of the cloud provider, stream processing framework, and benchmark. While Fig. 12.1 illustrates this observation for a threshold of $t = 2000$ with a warm-up duration of 120 s, the results for other thresholds $t > 0$ and warm-up durations are quite similar. Thresholds close to $t = 0$ require longer experiment durations, but as described in Section 11.4 allowing for a small lag trend increase is sensible. As for our scalability metric we are ultimately only interested in whether the SLO is met, we only look at the executing times required to decide if the lag trend does not exceed the threshold.

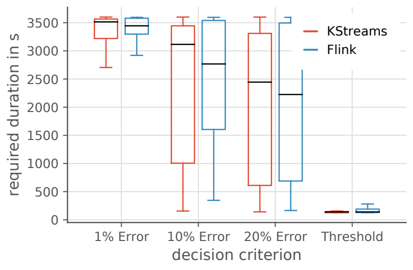
Fig. 12.2 summarizes the required execution times with a threshold of $t = 2000$ for different warm-up durations between 30 s and 480 s (multiples of the sampling interval) and all evaluated SUTs as box plots. We observe that in the vast majority of cases, warm-up periods of 60 s and 120 s result in required execution times of less than 5 minutes. Summarized over all SUTs, longer warm-up durations lead to less variability in the required execution duration, but also cause longer durations in most of the cases. We make similar observations independently of the chosen threshold.



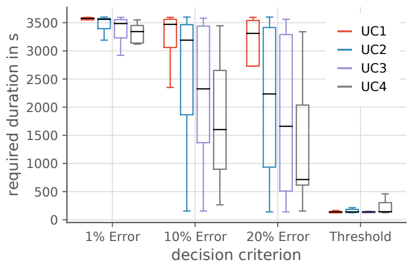
(a) among all SUTs



(b) separated by cloud platform



(c) separated by stream processing framework

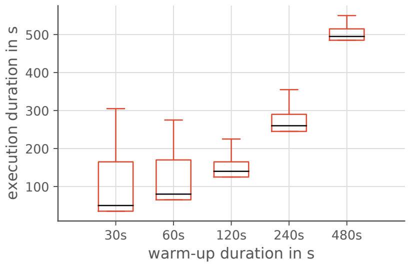


(d) separated by benchmark

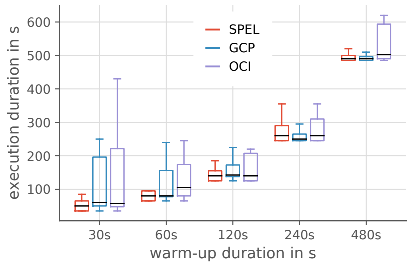
Figure 12.1. Box plots showing the required execution duration among all SUTs and cloud providers for different decision criterion. Whiskers are restricted to 1.5×IQR (interquartile range) and outliers lying below or above the whiskers are omitted for readability [HH22a].

From our experiment results, we consider a warm-up duration of 120 s to be a good trade-off. In contrast to 60 s warm-up, 120 s result in longer median execution times, but minimize the execution duration for the vast majority of experiments (see the upper whisker). When only looking at the private cloud or benchmark UC3, also significant shorter warm-up durations of 30 s could be chosen.

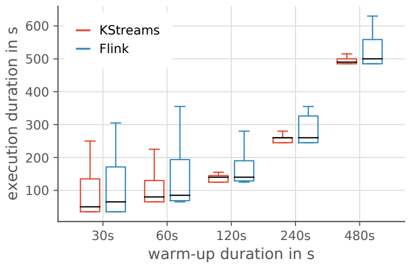
Table 12.4 shows the execution times for all evaluated SUTs for a warm-up duration of 120 s and a threshold of $t = 2000$. In line with Fig. 12.2, we see that certain resource-load combinations require significant longer



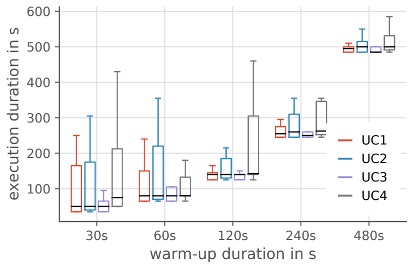
(a) among all SUTs



(b) separated by cloud platform



(c) separated by stream processing framework



(d) separated by benchmark

Figure 12.2. Box plots showing the required execution duration among all SUTs and cloud providers for different warm-up durations. Whiskers are restricted to 1.5×IQR and outliers laying below or above the whiskers are omitted for readability [HH22a].

execution times. However, we can observe that in these cases, testing the same load with slightly less or slightly more instances only requires a fraction of the time. Hence, with a significantly shorter execution time, we can get a good approximation of the resource demand.

12.2.3 Summary

Running experiments until we obtain a stable performance measurement result is impractical due to the immense time required for such evaluations. However, simply determining whether a SLO is met or not can

Table 12.4. Required experiment duration of SLO experiments for different SUTs, load intensities (mes./s), and resource amounts ($\text{inst}^{\vee}$, $\text{inst}^{\approx}$, and $\text{inst}^{\wedge}$) with a warm-up duration of 120 s.

SUT			load	execution time in s		
cloud	framework	bench.	mes./s	$\text{inst}^{\vee}$	$\text{inst}^{\approx}$	$\text{inst}^{\wedge}$
GCP	Flink	UC1	200 000	3085	225	140
		UC2	150 000	130	280	155
		UC3	60 000	265	150	125
		UC4	65 536	140	125	840
	Kafka Streams	UC1	300 000	140	125	145
		UC2	150 000	125	305	145
		UC3	20 000	140	130	140
		UC4	65 536	145	145	140
OCI	Flink	UC1	200 000	140	125	135
		UC2	150 000	130	215	205
		UC3	60 000	140	125	125
		UC4	65 536	140	125	1710
	Kafka Streams	UC1	300 000	155	220	125
		UC2	150 000	140	125	1560
		UC3	30 000	140	125	125
		UC4	65 536	140	350	460
SPEL	Flink	UC1	300 000	140	165	125
		UC2	150 000	185	125	180
		UC3	60 000	140	180	125
		UC4	65 536	155	170	415
	Kafka Streams	UC1	300 000	125	125	125
			600 000	140	140	125
		UC2	150 000	140	125	155
			300 000	140	140	125
		UC3	30 000	140	125	125
			60 000	2220	125	2185
		UC4	65 536	140	125	125

be done within 5 minutes in most cases, when using warm-up durations of 60 s to 120 s. Although certain resource-load combinations are more difficult to assess, execution times of up to 5 minutes provide still a good approximation of the resource demand and the load capacity.

12.3 Evaluation of Repetition Count

Our scalability measurement method and, thus, our proposed benchmarking tool architecture support repeating SLO experiments multiple times to increase the confidence in their results. In this section, we address EQ 2 and evaluate how many repetitions are required to decide with sufficiently high confidence whether SLOs are met.

12.3.1 Experiment Design

As in the previous evaluation, we perform SLO experiments for all 24 SUTs depicted in Table 12.1 with the same amounts of resources and load intensities (see Table 12.3). According to our results from the evaluation of warm-up and experiment duration, we run each SLO experiment for 5 minutes with the first 2 minutes considered as warm-up period. We perform 30 repetitions of each experiment as suggested, for example, by Kounev et al. [KLvK20] to apply the Central Limit Theorem.

For the majority of SUTs, we observe a normal distribution on the computed lag trend slopes. Deriving mean $\overline{x}$ and standard deviation s (with $N-1$ degrees of freedom) of the lag trend slopes for an SUT, we can now approximate how many repetitions are required to obtain a certain confidence interval for the true mean [KLvK20]. However, similar to the previous evaluation, we are ultimately only interested in whether the lag trend slope is above or below a threshold t. Thus, we do not need to approximate the number of repetitions to obtain a two-sided confidence interval with a certain error around the mean, but instead only consider a one-sided confidence interval of $(-\infty, t)$ or (t, ∞), respectively. We can approximate the required number of repetitions n for such a 95% confidence interval with:

$$n = \left(\frac{z_{0.05} s}{t - \overline{x}} \right)^2$$

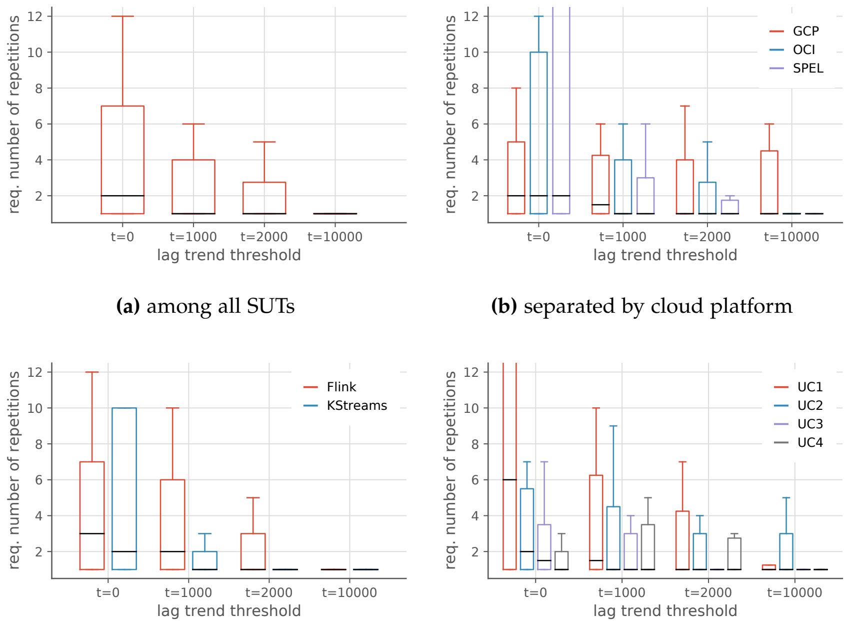
12.3.2 Results and Discussion

Fig. 12.3 summarizes the approximated number of repetitions for different thresholds t as box plots. Our replication package [HH21b] allows obtaining these values also for other thresholds. We observe that, independent of the chosen threshold, the required number of repetitions for most SUTs is very low: 50% of all SUTs only require 1–2 repetitions. Furthermore, the observed variability decreases with higher thresholds. While in most cases for $t = 0$ up to 12 repetitions are necessary, this value decreases to 6 repetitions for $t = 1000$ and 5 repetition for $t = 2000$. For $t = 10000$ even in the vast majority of cases only one repetition is necessary. However, a threshold of $t = 10000$ means that an increase of 10 000 messages per second is tolerable, which in some evaluated configurations already corresponds to half the generated load. This raises the chance of considering resource amounts as sufficient which in fact are not. A dependency on the threshold can be observed independently of the cloud platform, stream processing framework, and the benchmark. Generally, when looking at thresholds $t \geqslant 1000$, slightly more repetitions are required in the public clouds (with more repetitions in the Google cloud than in the Oracle cloud). A possible explanation is that performance in public clouds is often influenced by co-located tenants ("noisy neighbor") and, thus, is less stable [LC16]. In the private cloud, on the other hand, we exclusively control the entire hardware. However, the observed deviation between public and private cloud is rather low. Another explanation, thus, could simply be that considerably more computing resources are available in our private cloud, resulting in a lower hardware utilization. It is also noticeable that Flink requires more repetitions than Kafka Streams. We cannot identify a clear pattern suggesting that particular benchmarks require more repetitions than others.

Table 12.5 shows the approximated number of repetitions of all evaluated SUTs and load intensities for different numbers of instances and a threshold of $t = 2000$. In addition to the box plots presented in Fig. 12.3, we can see that in certain cases very high numbers of repetitions (high-

Table 12.5. Required number of repetitions of SLO experiments for different SUTs, load intensities (mes./s), and resource amounts ($\text{inst}^{\vee}$, $\text{inst}^{\approx}$, and $\text{inst}^{\wedge}$). The threshold for the lag trend is $t = 2000$.

SUT			load	required repetitions		
cloud	framework	bench.	mes./s	$\text{inst}^{\vee}$	$\text{inst}^{\approx}$	$\text{inst}^{\wedge}$
GCP	Flink	UC1	200 000	5	15	4
		UC2	150 000	1	3	1
		UC3	60 000	1	1	2
		UC4	65 536	1	11	3
	Kafka Streams	UC1	300 000	4	1	1
		UC2	150 000	8827	7	1
		UC3	20 000	2	1	1
		UC4	65 536	1	1	1
OCI	Flink	UC1	200 000	2	5	1
		UC2	150 000	19	2	1
		UC3	60 000	1	1	1
		UC4	65 536	1	115	1
	Kafka Streams	UC1	300 000	130	1	1
		UC2	150 000	1	1	9
		UC3	30 000	1	1	1
		UC4	65 536	10	1	1
SPEL	Flink	UC1	300 000	1	1	1
			600 000	5	7	1
		UC2	150 000	1	3	1
			300 000	3	4	1
		UC3	60 000	1	1	1
			240 000	1	1	1
		UC4	65 536	1	2	799
	Kafka Streams	UC1	300 000	1	1	1
			600 000	1	1	1
		UC2	150 000	1	1	2
			300 000	2	1	1
		UC3	30 000	1	1	1
			60 000	1	24	3
		UC4	65 536	1	1	1



(a) among all SUTs

(b) separated by cloud platform

(c) separated by stream processing framework

(d) separated by benchmark

Figure 12.3. Box-plots showing the required number of repetitions of SLO experiments for different thresholds. Whiskers are restricted to 1.5×IQR and outliers laying below or above the whiskers are omitted for readability [HH22a].

lighted in red) would be required in order to tell with sufficiently high confidence whether the lag trend slope is above or below the threshold. However, in almost all of these cases, we would only need a few repetitions when evaluating the same SUT with slightly more or fewer instances. We also observed this when choosing a different threshold. Transferred to our scalability measurement method, this means that only a few repetitions are required to obtain a good approximation of the resource demand. Therefore, only a few repetitions are required to determine the *demand* and *capacity* functions when accepting a small error in the function.

12.3.3 Summary

In most cases, only very little repetitions are required to assess whether an SLO is met or not. While determining the exact resource demand might require hundreds or thousands of repetitions, up to 3 or 5 repetitions are sufficient to determine a close approximation of the resource demand and the load capacity.

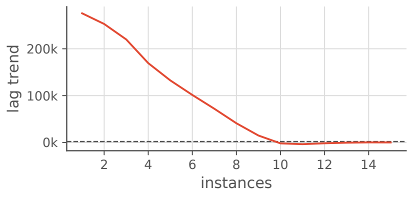
12.4 SLO Evaluation with Increasing Resources

In this section, we evaluate how the computed lag trend evolves with increasing the provisioned resource amounts, while keeping the generated load constant. The goal of this evaluation is to analyze whether SLOs might be violated for higher resource amounts when they have been fulfilled before for lower resource amounts.

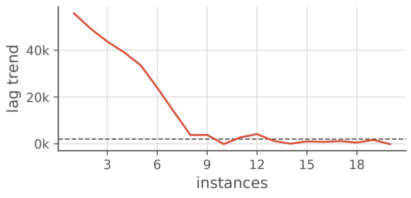
12.4.1 Experiment Design

In this evaluation, we address EQ 3 and evaluate selected SUTs in more detail. From our previous evaluation, we observed that the results for both public clouds do not differ significantly. The same applies for the benchmarks. Hence, we focus on the two benchmarks UC2 and UC3 and restrict our experiments to the private cloud and the Google cloud. This results in 8 SUTs (see Table 12.6).

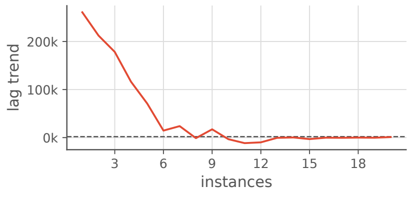
For each SUT, we conduct a set of isolated SLO experiments, in which we generate a constant load, equal to the loads of Table 12.3, and different resource amounts. We repeat each SLO experiment 5 times with 5 minutes of experiment duration including 2 minutes of warm-up. Table 12.6 summarizes the experiment set-up. For each SLO experiment, we compute the lag trend allowing us to analyze how the lag trend evolves with increasing resource amounts. In Google Cloud Platform, we additionally performed SLO experiments in a Kubernetes cluster with 6 instead of 3 nodes as explained in the following section.



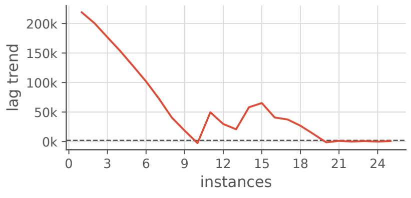
(a) SPEL, Kafka Streams, UC2



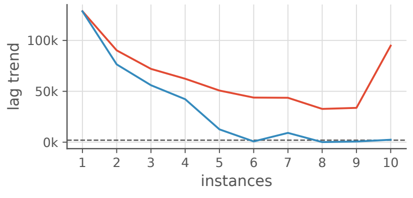
(b) SPEL, Kafka Streams, UC3



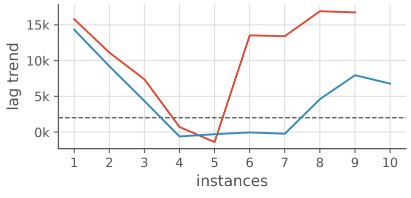
(c) SPEL, Flink, UC2



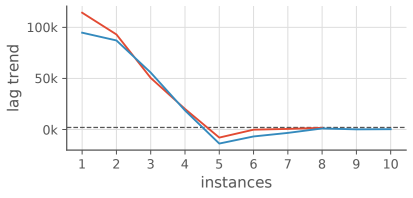
(d) SPEL, Flink, UC3



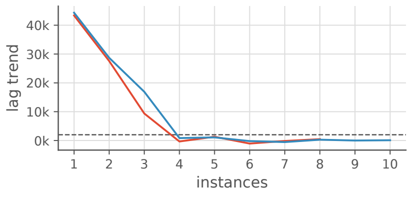
(e) GCP, Kafka Streams, UC2



(f) GCP, Kafka Streams, UC3



(g) GCP, Flink, UC2



(h) GCP, Flink, UC3

Figure 12.4. Lag trend with increasing resource amounts for different SUTs [HH22a]. In the Google cloud, the red line represents the lag trend for a 3 node cluster, while the blue line represents the 6 node cluster.

12.4.2 Results and Discussion

Fig. 12.4 shows for each evaluated SUT how the median lag trend evolves with increasing resource amounts. Additionally, a horizontal line at a lag trend of 2000 is drawn to visualize a possible threshold for the *lag trend metric*.

In general, we can observe that in the private cloud the lag trend decreases with increasing amounts of instances, until it reaches a value of approximately 0 and, thus, falls below the defined threshold. This marks the resource demand of the tested load intensity according to our *demand* metric. After that, the lag trend fluctuates considerably for 3 out of 4 SUTs, before it stabilizes at around 0. These fluctuations occur more strongly with Flink than with Kafka Streams and more strongly with UC3 than with UC2. In particular, we can observe that 10 instances seem to perform better than 9 and 11 instances. One possible reason for this may be found in the fact that we use 40 Kafka topic partitions and the stream processing frameworks might work particularly efficiently if the partition count is a multiple of the instance count.

For the experiments in the public cloud, these effects cannot clearly be observed. However, we observe that with a cluster size of 3 nodes, the lag trend increases again after some point when further instances are added. As we expect this to be due to exhausted node resources, we repeat the same experiments in a Kubernetes cluster with twice the number of nodes. From this, we can see that the same instance numbers result in lower

Table 12.6. SUT configuration for evaluations of the SLO with increasing resources.

	SPEL				GCP			
	KStreams		Flink		KStreams		Flink	
	UC2	UC3	UC2	UC3	UC2	UC3	UC2	UC3
mes./s	300 000	60 000	300 000	240 000	150 000	20 000	150 000	60 000
instances	$\leqslant 15$	$\leqslant 20$	$\leqslant 20$	$\leqslant 25$	$\leqslant 10$	$\leqslant 10$	$\leqslant 10$	$\leqslant 10$
duration	5 minutes, including 2 minutes warm-up							
repetitions	5							

lag trends and, especially, the lag trend remains below the threshold for higher instance numbers. Thus, we see our assumption confirmed that the increase for higher loads is caused by a high utilization of the cluster.

Regardless of the actual reasons for both observations, we can conclude that the lag trend is not always decreasing with higher resource amounts. Therefore, our proposed binary search strategy must be used with caution for the *demand* metric. For our *capacity* metric, this means that increasing resources might also lead to violations of SLOs such that the lower bound restriction cannot always be applied.

12.4.3 Summary

In general, the lag trend decreases with increasing resources. However, there are clearly certain resource configurations which perform better than others. This means, the binary search cannot necessarily be used with the demand metric, while the lower bound restriction cannot necessarily be used with the capacity metric. Additionally, benchmarkers should be aware of the underlying resource limits. Reaching these limits breaks monotonicity, making these search strategies not applicable.

12.5 SLO Evaluation with Increasing Load

In this section, we now address EQ 4 and evaluate how the lag trend slope evolves with increasing loads, while fixing the number of processing instances. The goal is to analyze whether SLOs might be violated for lower load intensities while they are achieved for higher loads.

12.5.1 Experiment Design

Similar to the previous evaluation, we now fix the amount of instances and perform a set of SLO experiments for increasing load intensities. The remaining setup corresponds to that of Section 12.4 and is summarized in Table 12.7.

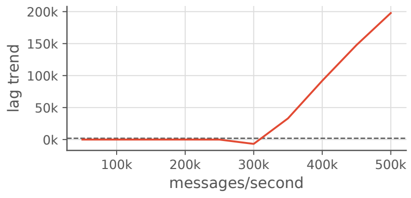
Table 12.7. SUT configuration for evaluations of SLO with increasing load.

	SPEL				GCP			
	Kafka Streams		Flink		Kafka Streams		Flink	
	UC2	UC3	UC2	UC3	UC2	UC3	UC2	UC3
mes./s	$\leqslant 500k$	$\leqslant 100k$	$\leqslant 500k$	$\leqslant 400k$	$\leqslant 250k$	$\leqslant 50k$	$\leqslant 250k$	$\leqslant 100k$
instances	10	13	11	20	6	5	5	6
duration	5 minutes, including 2 minutes warm-up							
repetitions	5							

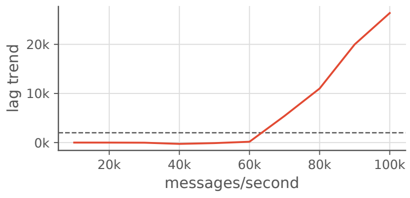
12.5.2 Results and Discussion

Fig. 12.5 shows for each evaluated SUT how the median lag trend evolves with increasing load intensities. Again, a horizontal line at a lag trend of 2000 visualizes a possible threshold for the *lag trend metric*.

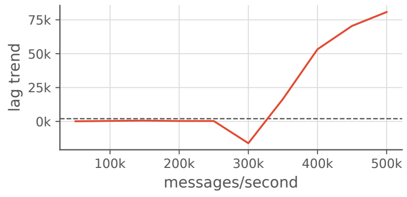
For 7 out of 8 evaluated SUTs, we observe that for low load intensities the lag trend stays reasonably constant and fluctuates only slightly around 0, until a certain load intensities is reached. In all of these cases, it does not exceed 2000, which suggests that $t = 2000$ is a reasonable order of magnitude for the threshold. For the Kafka Streams implementation of UC3 in Google Cloud Platform, the lag trend is always greater than the threshold since our evaluated load intensities are too high. For all other SUTs, we can first observe a slight drop in the lag trend once a certain load intensity is exceeded, which is followed by monotonically increase. This marks the capacity of the evaluated resource configuration according to our *capacity* metric, i.e., the maximal load it can process. The drop can be explained by the fact that the load is already high enough such that messages are massively queuing up while the SUT starts up. Once the SUT reaches its normal throughput, messages have already been accumulated and are then continuously processed, leading to a decrease in the lag. The drop of Flink deployments is stronger compared to Kafka Streams since Flink has a longer start-up time as we investigated manually. Again, the Kafka Streams implementation of UC3 in Google Cloud Platform is the only SUT, for which the the lag trend is not monotonically increasing.



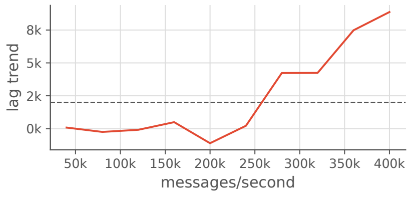
(a) SPEL, Kafka Streams, UC2



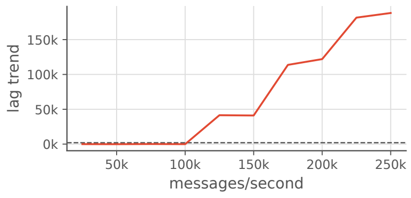
(b) SPEL, Kafka Streams, UC3



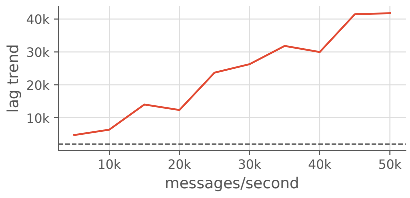
(c) SPEL, Flink, UC2



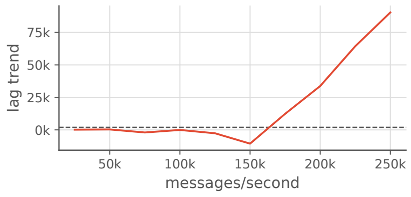
(d) SPEL, Flink, UC3



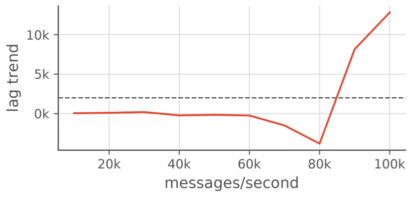
(e) GCP, Kafka Streams, UC2



(f) GCP, Kafka Streams, UC3



(g) GCP, Flink, UC2



(h) GCP, Flink, UC3

Figure 12.5. Lag trend with increasing load for different SUTs [HH22a].

More specifically, for load intensities of 20 000 and 40 000 messages per second, the lag decreases. Since, however, we cannot make this observation on other than the median data, we expect this to be outliers.

In summary, we conclude that the binary search strategy can be used to evaluate large sets of load intensities with the *capacity* metric, at least when benchmarking event-driven microservices with an SLO based on the *lag trend metric*. As furthermore the computed lag trend is monotonically increasing after exceeding the defined threshold, we expect also the lower bound restriction to be applicable for the *demand* metric.

12.5.3 Summary

When increasing the load, the lag trend remains below a threshold slightly higher than 0 up to a certain load intensity. For higher load intensities, the lag trend is monotonically increasing. This means, the preconditions for using the binary search with the *capacity* metric and the lower bound restriction with the *demand* metric are fulfilled.

12.6 Threats to Validity

The goal of this experimental evaluation is to assess how configuration options of our scalability benchmarking method influence their results. In the following, we report on the threats and limitations to the validity of our evaluation.

Threats to Internal Validity Cloud platforms in general allow only making little assumptions regarding the underlying hardware or software infrastructure [BWT17]. Due to techniques such as containerization, cloud-native applications abstract this even further. Hence, we only have little influence on the execution environment and cannot control possible influences on our result. Major part of this evaluation is to investigate the variability of results and hence the underlying processing capabilities of stream processing frameworks. Nevertheless, we reuse the same clusters among all our experiments such that we cannot rule out that a recreation of the cluster on potential other hardware or with other co-located VMs will cause different results. Furthermore, we only perform our experiments in a relatively short time frame. We perform experiments of the same type mostly in a sequence such that we cannot rule out that general performance variations over several hours bias our results. While early works on cloud benchmarking found that performance exhibits clear seasonal patterns [IYE11], more recent research were not able to confirm this [LC16]. Our cluster configuration of the private cloud and the public clouds is very different, which may make it difficult to compare them.

Threats to External Validity We only evaluate one type of cloud-native applications, namely event-driven microservice that use distributed stream processing frameworks. Our results regarding the required experiment duration and required number of repetitions should therefore not be generalized to other types of cloud-native applications, which potentially use other SLOs. Furthermore, we only consider two stream processing frameworks and focus on single types of load and resources. Similar limitations apply to the evaluated cloud environments. In the public clouds, we only evaluate virtual machines, while in the private cloud, we only evaluate bare metal servers. Furthermore, the nodes in the VM are of medium size resulting in an overall small cluster, while the powerful nodes in private cloud provide much more computing capacity. To increase the external validity of our results, it might be advisable to perform additional evaluations with other cluster sizes.

Chapter 13

Evaluating Scalability of Distributed Stream Processing Frameworks

In this chapter, we employ our Theodolite benchmarking method and our benchmarks for event-driven microservices to assess and compare the scalability of different stream processing frameworks and their configurations. Specifically, we state the following evaluation questions:

EQ 1 How do different stream processing frameworks compete regarding their scalability?

EQ 2 Are the previously discovered performance limitations of Apache Beam's abstraction layer [HMG+19] still present with more recent framework versions?

EQ 3 How do stream processing frameworks scale with increasing computational work?

EQ 4 Can vertical scaling be a viable alternative to horizontal scaling?

This chapter starts by describing our experimental setup in Section 13.1. Section 13.2 starts addressing EQ 1 by running baseline experiments for each benchmark and framework. This section also provides first results regarding EQ 2, which is further investigated in Section 13.3. Section 13.4 addresses EQ 3 and evaluates how different stream processing frameworks scale when increasing the duration of window aggregations. Section 13.5 addresses EQ 4 by evaluating how stream processing frameworks scale on a single node. Finally, Section 13.6 discusses threats to validity. We provide a preprint of a publication building upon this chapter, which is currently under peer review [HH23b].

13.1 Experimental Setup

For the evaluations in this chapter, we use the *SPEL* private cloud environment described in Section 12.1, yet with the newer Kubernetes version 1.23.7. Unless stated differently, we run 5 Kafka brokers, one on each node, with Kafka version 3.2. Each Kafka topic is configured to consist of 100 partitions. In the following, we summarize the configuration of the benchmarked stream processing framework, the selected benchmark task samples, and the benchmarking method. We provide a replication package and the collected data of all experiments as supplemental material [HH22c], allowing other researchers to repeat and extend our work.

13.1.1 Configuration of Frameworks

We benchmark the stream processing frameworks Apache Beam with the Flink and the Samza runner, Apache Flink, Hazelcast Jet, and Apache Kafka Streams. For a fair comparison, we evaluate all frameworks with mostly their default configuration. We enable committing read offsets to Kafka in all frameworks. This allows us to monitor the consumer lag via Kafka metrics, which is required to evaluate our lag trend SLO. Enabling offset committing is also often done in production deployments to increase observability. We set the commit interval to 5 seconds for all frameworks, which is the default configuration of Kafka consumers once offset committing is enabled. Kafka Streams has a default commit interval of 30 seconds as in Kafka Streams, the commit interval also controls fault tolerance (comparable to the checkpointing interval in other frameworks). For our experiments with Apache Beam and the Flink runner, we enable the *FasterCopy* option as we further discuss and evaluate in Section 13.3. Per default, we configure 1 CPU and 4 GB of memory for each pod, which is a common ratio of CPU and memory of cloud VMs.

13.1.2 Configuration of Task Samples

Unless otherwise stated, we use the following configuration of our benchmark dataflow architectures.

– Benchmark UC1 is configured to write each incoming message as a log statement to the standard output stream to simulate a database write operation (i.e., simulating a side effect in the dataflow architecture).

– Benchmark UC2 aggregates incoming messages over tumbling windows of one minute. Any out-of-order records arriving after the window has been closed are discarded.

– Benchmark UC3 aggregates records by their hour of day attribute over a time window of three days with a slide period of one day. That means each incoming record belongs to three time windows. Early results (i.e., before the end of the time window has passed) are emitted every 5 seconds. For Kafka Streams, such emission cannot explicitly be configured. However, Kafka Streams continuously forwards aggregation results based on the configured *commit interval* (which is also 5 seconds).

– We benchmark a simplified version of benchmark UC4, which omits the feedback loop. This allows for better predictability of the message volume and, hence, more comparable results.

13.1.3 Configuration of the Benchmarking Method

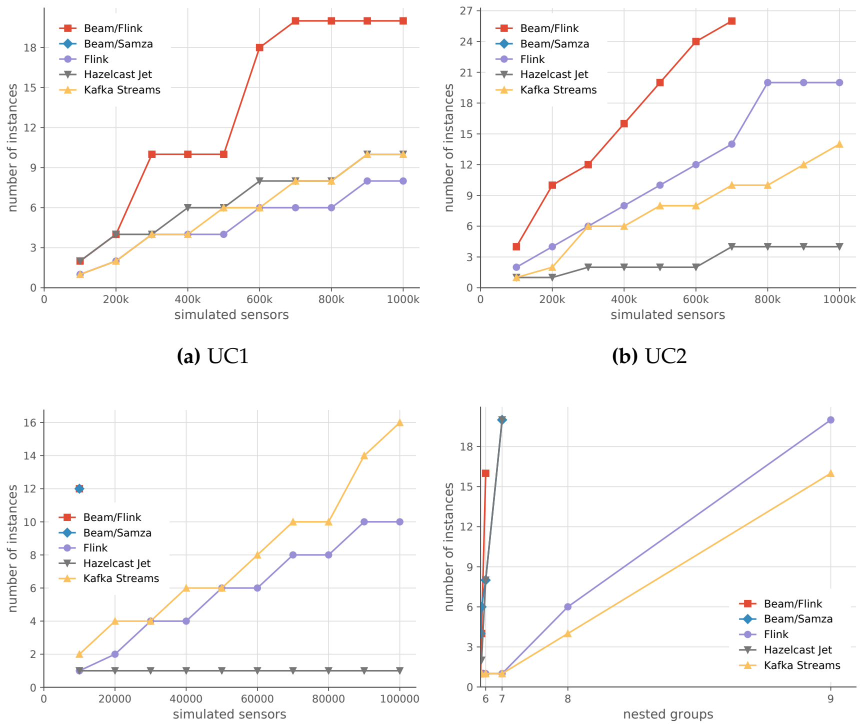
According to our experimental evaluation in Section 12.2, we run our experiments with benchmark UC1–UC3 for a duration of 5 minutes while considering the first 2 minutes as warm-up period. As benchmark UC4 shows a higher variability in the results, we run its experiments for 10 minutes including a 4 minutes warm-up period. We quantify scalability with our resource demand metric and use the *linear search* strategy in combination with the *lower bound restriction*. If not stated differently, we evaluate scalability in regards to increasing the number of simulated sensors as load type. In benchmark UC4, this is indirectly controlled by increasing the number of nested groups, with each group containing 4 sub-groups or sensors. This means for n nested groups, we simulate 4^n sensors. Each simulated sensor generates one measurement per second. Unless otherwise stated, we use the number of instances as resource type. In all benchmark executions, we use the lag trend SLO with a threshold of 1% of the generated message volume. Additionally, for benchmarks UC2 and UC4 (which aggregate data in short windows), we configure the dropped records SLO with again a threshold of 1% of the generated message volume. For the Apache Beam implementation with the Samza runner, no metrics concerning the number of dropped records are provided. With Beam's Flink runner, these metrics are only unreliably available.[1] This means that for these two SUTs, we cannot definitely be sure whether the determined resource demand for UC2 and UC4 is sufficient to process all records successfully, yet indicating a lower bound.

13.2 Baseline Experiments

We benchmark load intensities between 100 000 and 1 000 000 simulated sensors (and, thus, generated messages per second) for benchmark UC1 and UC2, 10 000 and 100 000 simulated sensors for UC3, and 5 to 9 nested groups (1 024–262 144 generated messages per second) for benchmark UC4.

Fig. 13.1 shows the resource demand results for all evaluated frameworks and benchmarks. The results for benchmark UC4 (in the following figures as well) are visualized with an exponential scale with base 4 at the horizontal axis since the number of generated messages grows exponentially with a linear increase in the number of nested groups. We can observe that in almost all experiments, Flink, Hazelcast Jet, and Kafka Streams, have a considerably lower resource demand than the Beam deployments. Only Hazelcast Jet in UC4 and the Beam Flink runner for low loads in UC1 are exceptions to this. As in some cases, the generated load intensities were too high for the Beam deployments, we repeat the corresponding experiments with lower load intensities (see Fig. 13.2 and Fig. 13.5).

[1]We asked a corresponding question regarding the metrics of both runners at Beam's mailing list, but did not receive an answer.



(a) UC1 **(b)** UC2

(c) UC3 **(d)** UC4

Figure 13.1. Scalability benchmark results according to our resource demand metric for the stream processing frameworks Apache Beam (with the Flink and Samza runners), Apache Flink, Hazelcast Jet, and Apache Kafka Streams.

Despite some outliers (see Beam/Flink in UC1 and Flink in UC2), all frameworks show linear scalability according to our resource demand metric, yet with different rates. Whereas both SUTs based on Beam show the steepest increase in required resources, the results of Flink, Kafka Streams, and Hazelcast Jet vary depending on the benchmark. In UC1 (see Fig. 13.1a), all frameworks behave similarly, with resource demands increasing slightly steeper for Hazelcast Jet compared to Kafka Streams and for Kafka Streams compared to Flink. For UC2 (see Fig. 13.1b), we see

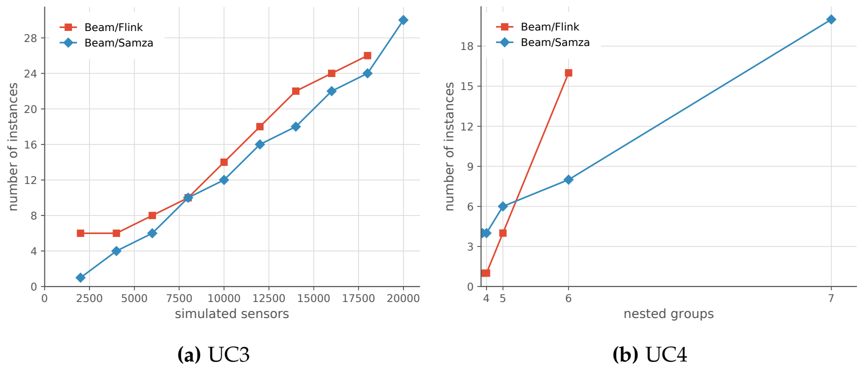
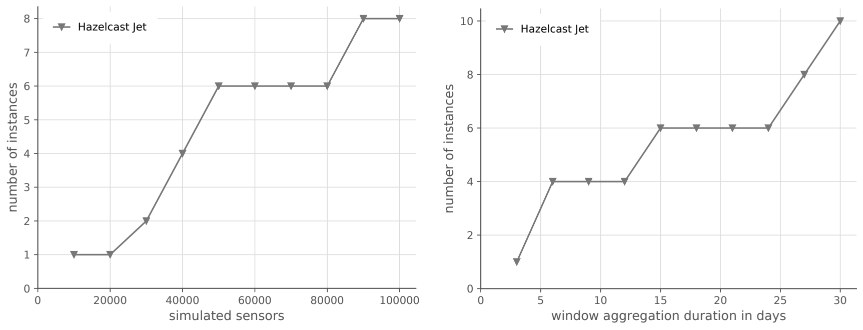


Figure 13.2. Repetition of scalability experiments shown in Figs. 13.1c–d with lower load intensities for Apache Beam with the Flink and the Samza runners.

a clear ranking with Hazelcast Jet showing the best results, followed by Kafka Streams and Flink. For UC3 (see Fig. 13.1c), Hazelcast Jet appears to be even more superior. A single Jet instance is sufficient for all evaluated load intensities. On the other hand, Flink requires up to 10 TaskManagers and Kafka Streams up to 16 instances. Overall, Kafka Streams' resource demand for UC3 increases at a steeper rate compared to Flink. To further inspect the scalability of Hazelcast Jet for UC3, we repeat these experiments with an aggregation duration of 30 days in contrast to 3 days as used in the other experiments. Fig. 13.3a shows that Hazelcast Jet also scales linearly in this case. With UC4 (see Fig. 13.1d), we observe a slightly flatter increase in resource demand for Kafka Streams compared to Flink. In contrast to the other benchmarks, Hazelcast Jet shows a significantly higher resource demand. Up to 30 instances are not able to handle load from more than 7 nested sensor groups.

For the frameworks used with Apache Beam, we observe a significantly steeper increase in resource demand of Samza compared to Flink in UC1 (cf. Fig. 13.1a and Fig. 13.5a) and UC2 (cf. Fig. 13.1b and Fig. 13.5b). For benchmark UC3 (see Fig. 13.2a), both frameworks scale at similar rates with Samza requiring slightly fewer instances. For benchmark UC4 (see Fig. 13.2b), it appears that the resource demand of Flink increases at a steeper rate. However, since we do not find a resource demand for loads



(a) 30 days aggregation period **(b)** 100 000 simulated sensors

Figure 13.3. Repetition of scalability experiments with Hazelcast Jet and benchmark UC3. (a) evaluates scalability regarding the number of simulated sensors with a 30 days aggregation period in contrast to the 3 days period in Fig. 13.1c. (b) evaluates scalability regarding the aggregation period with a constant load of 100 000 simulated sensors in contrast to 10 000 simulated sensors in Fig. 13.6a.

of more than 7 nested groups, we cannot safely conclude whether this is a clear trend. For lower load intensities (less than 6 nested groups), Flink requires fewer instances than Samza.

Worth mentioning is also the significant difference between native Apache Flink and the Flink runner of Apache Beam. In almost all experiments, the resource demand of Apache Beam with Flink is at least twice as high. For the more compute-intensive benchmarks UC3 and UC4, it is tremendously higher. The performance overhead of using Apache Beam as an abstraction layer has also been observed in related research [HMG+19].

13.3 Apache Beam Configuration

As we have seen in Section 13.2, Apache Flink and Apache Samza in combination with Apache Beam have a significantly higher resource demand compared to the other evaluated frameworks. In this section, we take a closer look at the scalability of the Apache Beam SUTs and evaluate how scalability is affected by different configuration options. In the following, we first look at the Apache Flink runner and, afterward, at the Apache Samza runner.

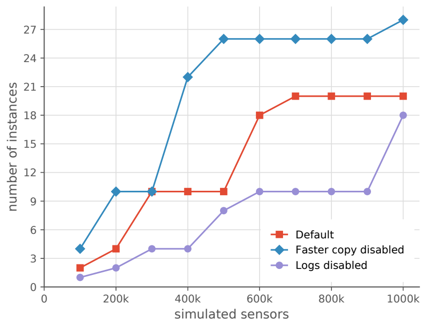
13.3.1 Apache Flink

In their master's thesis, Spæren [Spæ21] investigates possible reasons for the performance overhead of the Flink runner found by Hesse et al. [HMG+19]. They discovered unnecessary serialization and deserialization between operators and introduced the *FasterCopy* option, which disables these copy operations. This option is integrated in Beam since version 2.26. While the stream processing application must fulfill some requirements to run with the *FasterCopy* option, Bensien [Ben21] found that the Theodolite benchmarks fulfill these requirements. As additionally this option might become the default in future releases, we decided to turn on this option in our benchmark implementations by default. In this section, we evaluate how enabling and disabling *FasterCopy* affects scalability.

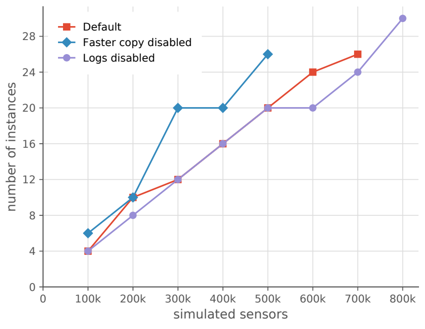
Additionally, we observed that Beam's Kafka consumers generate a lot of log messages if not configured differently. This contrasts with the other frameworks. As extensive logging can actually have an impact on performance (see the following Section 13.5), we evaluate whether disabling all logging results in lower resource demand.

Fig. 13.4 shows our results of running the scalability benchmarks with the *FasterCopy* option disabled and logging disabled compared to the experiments from Section 13.2. We can see that enabling *FasterCopy* results in significantly lower resource demands for UC1–UC3 (see Figs. 13.4a–c). This is in line with the performance improvements reported by Spæren [Spæ21]. For benchmark UC4 (see Fig. 13.4d), enabling *FasterCopy* seems to not have a great effect on resource demand. Whereas with 5 nested groups the determined required number of instances is slightly higher, it is slightly lower with 6 nested groups. A possible explanation is that the dataflow architecture of UC4 involves more data transfer among instances and, hence, actually requires serialization and deserialization between operators.

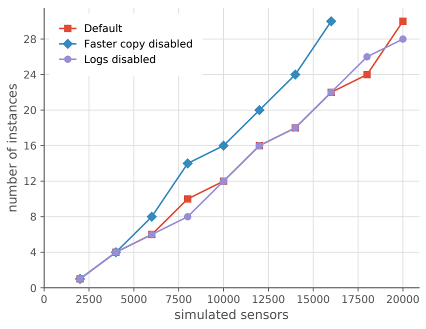
We can observe that disabling all logging has only a small impact on the resource demand of benchmark UC2–UC4 (see Figs. 13.4b–d), but significantly reduces the resource demand of benchmark UC1 (see



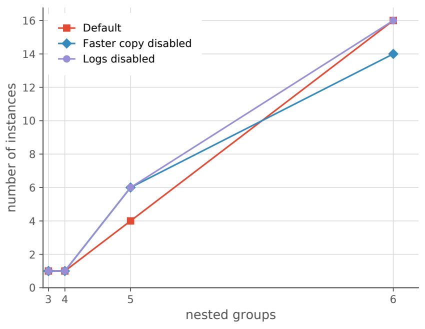
(a) UC1



(b) UC2



(c) UC3



(d) UC4

Figure 13.4. Scalability benchmark results according to our resource demand metric for different configurations of Apache Beam with the Apache Flink runner.

Fig. 13.4a). The latter is expected since benchmark UC1 logs each incoming message to simulate side effects such as writing records to a database. We can conclude that the extensive logging of Beam's Kafka consumer contributes very little to the overhead introduced by Apache Beam.

13.3.2 Apache Samza

In a blog post, software engineers at LinkedIn [ZXW+20] report how they tremendously improved the performance of Beam's Samza runner. Primar-

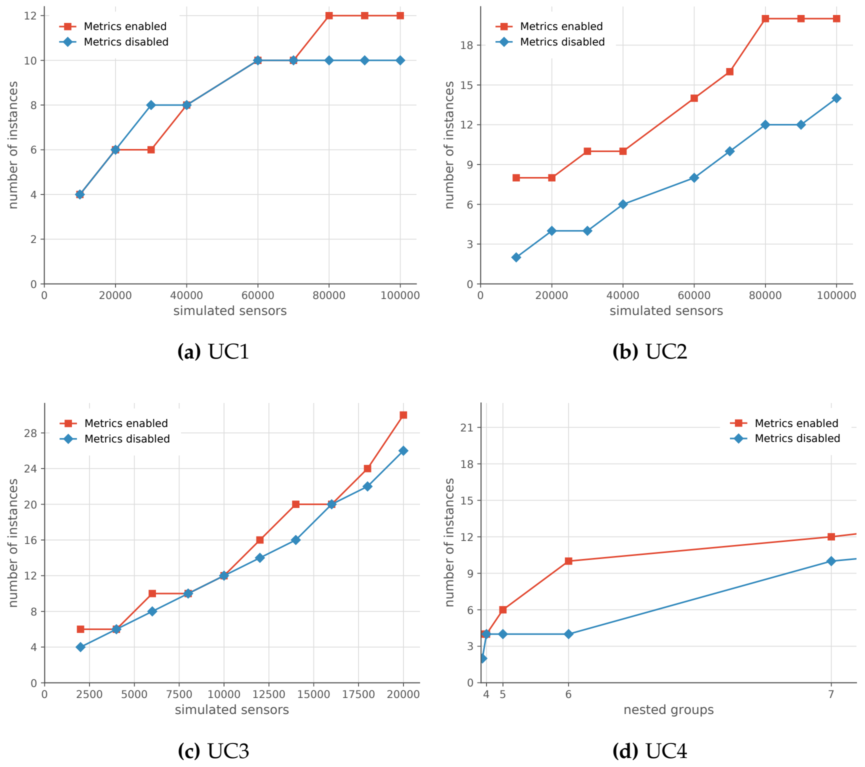


Figure 13.5. Scalability benchmark results according to our resource demand metric for different configurations of Apache Beam with the Apache Samza runner.

ily, this was achieved by exporting Beam metrics more efficiently. Moreover, the authors observed that performance could further be improved when disabling the Beam metrics entirely. Although this might not be an option in production [ZXW+20], we are still interested in how much performance could be further improved when disabling all Beam metrics.

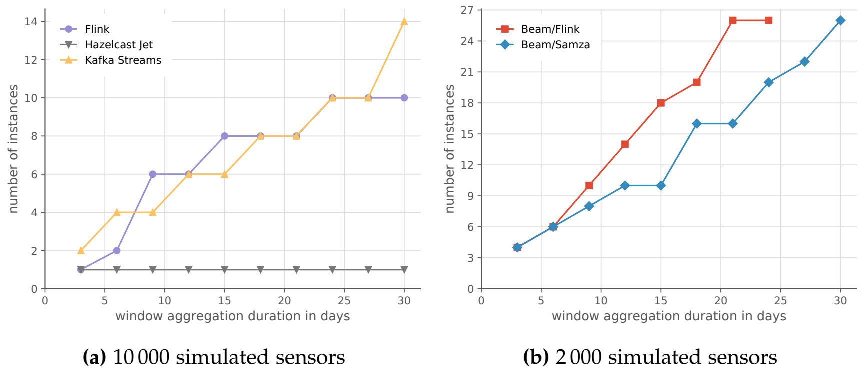
Fig. 13.5 shows our results for benchmarking scalability with Beam metrics enabled and disabled. We can observe that independent of the benchmark, disabling metrics results in a similar linear increase in resource demand, yet at a lower level. However, also with metrics disabled, the resource demand of Apache Beam with the Samza runner is considerable higher compared to most other frameworks (with metrics not disabled).

Worth mentioning is also the bachelor's thesis of Bensien [Ben21], who benchmarked Beam with the Samza runner in version 2.22 using Theodolite's UC1 benchmark. This Beam version did not include the performance improvements by Zhang et al. [ZXW+20] (released in Beam version 2.27). Bensien observed a resource demand more than twice as high with metrics enabled, compared to disabling them.

13.4 Scaling the Window Aggregation Duration

In Section 13.2, we configure benchmark UC3 with a window duration of 3 days to compute an average daily course. This is a trade-off to still benchmark generated data volume of reasonable size. However, it is likely that in practice, larger time windows are required to obtain more reasonable results. Therefore, in this section, we evaluate, how different stream processing frameworks scale with increasing benchmark UC3's window duration. We increase the window duration from 3 days to 30 days, while keeping the number of simulated sensors and, thus, the incoming message rate constant. This evaluation is an example of benchmarking scalability with respect to the work performed for each incoming message in contrast to scaling the load at the framework and, thus, addresses EQ 3.

Fig. 13.6 shows the results of these experiments. According to our previous results in Section 13.2, we simulate 10 000 sensors for our experiments with Flink, Hazelcast Jet, and Kafka Streams (see Fig. 13.6a) and 2 000 sensors for the Beam SUTs (see Fig. 13.6b). We can observe that again all frameworks scale approximately linearly. Remarkable is again the performance of Hazelcast Jet, which only requires a single instance, independently of the window size. We repeat these experiments with a higher load of 100 000 sensors. As shown in Fig. 13.3b, Hazelcast Jet also scales approximately linearly in this case. In contrast to scaling with the number of sensors (see Fig. 13.1c), Kafka Streams and Flink scale now with about the same rate of resource demand increase. Compared to the results shown Fig. 13.2a, Samza's resource demand increases less steeply compared to Beam's Flink runner.



(a) 10 000 simulated sensors **(b)** 2 000 simulated sensors

Figure 13.6. Scalability benchmark results of all stream processing frameworks according to our resource demand metric when increasing the duration of aggregation windows and, thus the number of windows maintained simultaneously.

13.5 Scaling on a Single Node

In this section, we address EQ 4 and evaluate whether vertical scaling can be a viable alternative to horizontal scaling for stream processing frameworks. We, therefore, scale our SUTs on a single node with both the number of instances and the amount of resources provided for a single SUT pod.

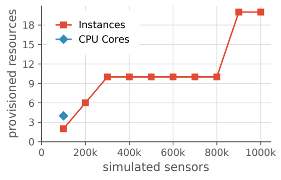
For the experiments of this section, we slightly modify our experimental setup. We only deploy 3 Kafka brokers, which run on different Kubernetes nodes. Of the other two nodes, we dedicate one to run the load generators and one to run the SUT instances. Although we run only 3 Kafka brokers and all load generator instances on the same node, we can confirm that the configured load is still successfully generated.

To not fully utilize the SUT node, which also runs some infrastructure and monitoring components, we deploy up to 20 instances with one CPU core each or up to 20 CPU cores for a single instance. For scaling with the number of instances, we keep the same configuration as in the previous experiments. For scaling with the resources per pod, we only refer to the number of CPU cores, but scale memory proportionally. (However, in all our experiments we never observed fully utilized pod memory.) In order to utilize multiple CPU cores, most stream processing frameworks have to be configured accordingly. For Flink, we scale the number of task slots of the TaskManager equally to the number of CPU cores. In Kafka Streams, we scale the number of threads equally to the number of CPU cores. For Samza, the documentation is inconsistent regarding scaling a standalone application on a single node. We decided to scale the *container thread pool* size equally to the number of CPU cores. Hazelcast Jet does not require additional configuration as an instance configures its cooperative thread pool automatically according to the number of CPU cores provided.

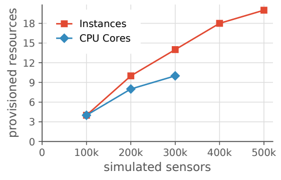
For most experiments, we generate the same load intensities as in the first experiment. However, we use the smaller load intensities from Section 13.3 for the Beam experiments and the 30 days window for Hazelcast Jet with benchmark UC3 as introduced in Section 13.4.

Fig. 13.7 shows the results of our experiments with a single node for benchmark UC1–UC3. Experimental results for benchmark UC4 can be found in our replication package [HH22c]. Almost all frameworks show approximately linear scalability when scaling the number of instances. We observe that Beam with the Samza runner does not scale with increasing the CPU resources per pod. Hence, we assume that scaling the container thread pool is not the right option to increase capacity on a single node. Whether other configuration options exist remains unclear. Further, we can observe that no framework is able to process load intensities higher than 200 000 messages per second with a single instance for benchmark UC1. The reason for this is that we simulate database writes by printing all incoming record to the standard output stream. Running a single instance of the frameworks causes all threads to write to the same stream, which is synchronized and becomes the bottleneck of our evaluation.

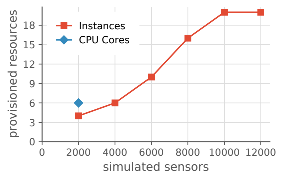
Kafka Streams seems to be more efficient when scaling with the number of instances, compared to scaling with the number of cores. For Flink and Hazelcast Jet, scaling with the number of cores is more efficient for the more complex dataflow in UC3, while with benchmark UC2 both types of scaling yield similar results. Remarkable are the results for scaling with the number of cores with Beam and the Flink runner. In contrast to native Flink, Beam with the Flink runner seems not to be scalable with respect to the number of cores. Moreover, the Kafka Streams implementation of benchmark UC3 scales neither with increasing the number of instances nor



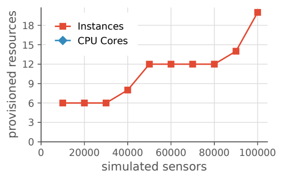
(a) Beam/Flink UC1



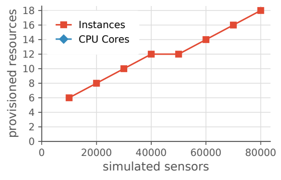
(b) Beam/Flink UC2



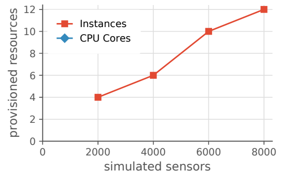
(c) Beam/Flink UC3



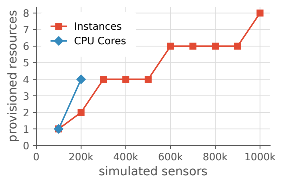
(d) Beam/Samza UC1



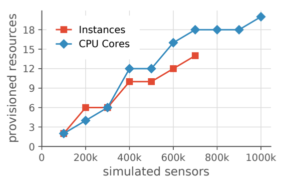
(e) Beam/Samza UC2



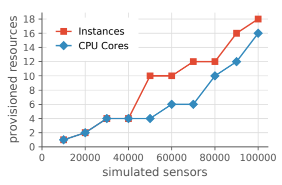
(f) Beam/Samza UC3



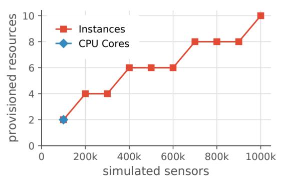
(g) Flink UC1



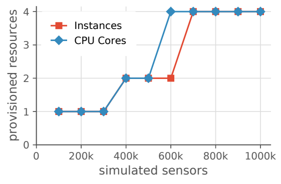
(h) Flink UC2



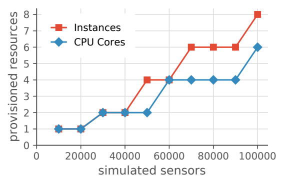
(i) Flink UC3



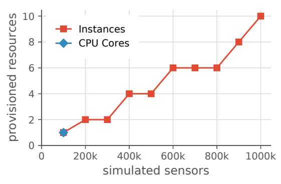
(j) Hazelcast Jet UC1



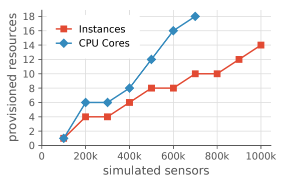
(k) Hazelcast Jet UC2



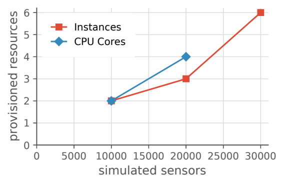
(l) Hazelcast Jet UC3



(m) Kafka Streams UC1



(n) Kafka Streams UC2



(o) Kafka Streams UC3

Figure 13.7. Scalability (resource demand metric) of frameworks on a single node.

with increasing the number of cores. As it scales linearly when running on multiple nodes (see Fig. 13.2a), our results suggest that underlying hardware resources become exhausted. However, from manually observing system-level metrics, we cannot observe anything conspicuous.

13.6 Threats to Validity

Despite careful research design, there exist threats and limitations to the validity of our evaluation, which we report below. In addition, the threats discussed in Section 12.6 largely apply to these evaluations as well.

Threats to Internal Validity To obtain statistically grounded benchmarking results, we build these evaluations upon our results of Chapter 12. Nevertheless, we only found that the selected configuration options provide good estimates. Hence, resource demands obtained in these evaluations should only be considered estimates. Repeating SLO experiments more often and for longer time periods as well as using our full search strategy is likely to produce very similar, but not necessarily identical results. Moreover, we evaluate a larger set of SUTs, load types, resource types, SLOs, and benchmarks in this section compared to the evaluation in Chapter 12. We address this limitation by carefully observing the benchmark execution but do not conduct as extensive experiments as in Chapter 12.

Threats to External Validity We conduct all experiments in this evaluation with our Theodolite benchmark task samples. As discussed in Chapter 9, these benchmarks represent relevant use cases. However, we cannot directly generalize our findings to arbitrary other applications, where other frameworks, configuration options, or deployment options might perform better than in our experiments. We conduct all experiments in this evaluation in a private cloud environment. In particular, we use comparatively large bare metal nodes. While this increases the internal validity (see Chapter 12) of our results, we cannot necessarily conclude that we would obtain the same results in public cloud environments. However, we expect that cloud-native abstraction layers such as containerization and resource limits mitigate these differences.

Chapter 14

Evaluating Scalability of Sliding Window Aggregation Methods

Aggregations over sliding time windows are a common task of stream processing systems. An example is Theodolite's UC3 benchmark (see Section 11.1.3), which computes statistics over a sliding window of several days. Other use cases of sliding window aggregations can be found in other benchmarks [KRK+18; BGM+20; vDvdP20]. Most state-of-the-art stream processing frameworks provide high-level APIs to define such aggregations, designed for scalability, fault-tolerance, and deterministic handling of out-of-order data. Additionally, performing sliding window aggregations more efficiently is a large field of research [CKH19; VGT+23].

In this chapter, we benchmark different methods for sliding window aggregations regarding their scalability. In particular, these are the native methods provided by the stream processing frameworks Apache Flink, Apache Kafka Streams, and Apache Spark. We complement this by an alternative method provided by Kafka Streams (see following the section) and Scotty [TGC+21], an extension for more efficient window aggregations, which is available for most stream processing frameworks. For this purpose, we employ our Theodolite benchmarking method with two task samples. First, we use our Theodolite UC3 benchmark introduced in Section 11.1.3. Second, we use the OSPBench benchmark [vDvdP20] as introduced in Section 11.6. With the latter, we also show how our scalability benchmarking method can be used with other stream processing benchmarks.

This chapter is structured as follows. Section 14.1 provides a brief overview of methods for sliding window aggregations. Section 14.2 describes our experimental setup. Section 14.3 presents and discusses the

Table 14.1. Naming of fixed-size window types in state-of-the-art stream processing frameworks.

Framework	Consecutive, fixed-size windows	Fixed window size with sliding period	Fixed window size, continuously sliding
Beam	Fixed time window	Sliding window	—
Flink	Tumbling window	Sliding window	—
Hazelcast Jet	Tumbling window	Sliding window	—
Kafka Streams	Tumbling window	Hopping window	Sliding window

results of our experiments. Section 14.4 concludes this chapter with a short discussion on threats to validity.

14.1 Methods for Sliding Window Aggregations

In most frameworks, sliding time windows are defined based on a window size and a sliding period [CKH19]. The window size defines a duration of a time period for which records are aggregated. The sliding period defines after which time a new window starts. For example a window size of 5 minutes and a sliding period of one minute means that every processed record is contained in 5 windows. Thus, strictly speaking, these windows do not actually "slide" continuously, but rather "hop" by a specified time period. For this reason, these windows are named *hopping windows* in Kafka Streams, with *sliding windows* having different semantics as shown in Table 14.1. For consistency with the literature and most frameworks, we use the term *sliding windows* for fixed window size with a sliding period.

Most modern stream processing frameworks aggregate records in time windows by creating a copy of each record for each window it belongs to. This copy is then keyed by the original key and the time window, allowing the stream processing framework to process all records to be aggregated by the same instance. The actual aggregation logic is performed in a user-defined way. Aggregation functions can be classified as distributive, algebraic, and holistic [GBL+96; TGC+21], having different implications on the state size.

The previously described method for sliding window aggregations has the disadvantage that often unnecessary or redundant computations are performed. Kafka Streams' sliding windows provide an alternative aggregation method [Ble20]. They do not have a defined sliding period. Instead, they provide semantics similar to sliding windows in other frameworks with the smallest possible sliding period (i.e., one millisecond for Kafka Streams). In contrast to those window types, however, Kafka Streams creates only up to two windows for each processed record, which can reduce the amount of computations significantly. To the best of our knowledge, this window aggregation method is unique to Kafka Streams.

Moreover, several approaches for more efficient sliding window aggregations have been proposed in research [CKH19; VGT+23]. The Scotty window processor [TGC+21] is such an approach, which applies the *stream slicing* technique. It divides the data stream into non-overlapping sections, called *slices*, aggregates data within slices, and further aggregates all slices belonging to the same window. Scotty is available as an open-source extension to several modern stream processing frameworks. A single-node performance evaluation of Scotty can be found by Traub et al. [TGC+21].

14.2 Experimental Setup

For the evaluations in this chapter, we use the *SPEL* private cloud environment described in Section 12.1. We deploy 10 Kafka brokers and configure 40 partitions per topic. In the following, we give a brief overview of our used benchmarks. A more detailed description can be found in the work of Vonheiden [Von21a] and the corresponding replication package [Von21b].

Theodolite's UC3 Benchmark We use extended Flink and Kafka Streams implementations of Theodolite's UC3 benchmark, which are configurable with the desired aggregation method. Our Flink implementation can optionally use the Scotty window operator instead of the default sliding window aggregation. Our Kafka Streams implementation can be configured to use Scotty, Kafka Streams' sliding windows, or the default hopping windows. In contrast to the experiments in Chapter 13, we use an earlier version of our implementations and configure no early emission of aggregation results. This is more aligned with our preliminary results [HH21e], but makes the results of Kafka Streams and Flink less comparable.

OSPBench We selected an alternative stream processing benchmark based on the following criteria. It should be available as open source, it should support Kafka as data source so we can use our lag trend SLO, it should provide implementations for multiple frameworks, and it should contain a sliding window aggregation, which is implemented with the native aggregation methods of the frameworks.[1] Furthermore, implementations should be available for multiple stream processing frameworks. Availability of container images is another plus, allowing to use our Theodolite framework without further adjustments of the benchmark itself. From our overview of benchmarks for stream processing frameworks (see Section 11.6), only OSPBench [vDvdP20] satisfies these criteria.

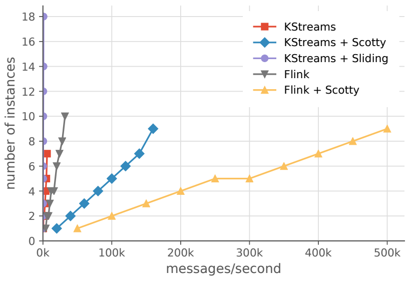
OSPBench provides open-source implementations for Apache Flink, Apache Kafka Streams, and Apache Spark, the latter with both the Spark Streaming API and the newer Structured Streaming API.[2] We select Flink, Kafka Streams and Spark Streaming as SUTs in this evaluation. We exclude Spark Structured Streaming since it does allow us to compute the lag trend via Kafka metrics [Von21a]. Within OSPBench, we select one of its aggregation pipelines [vDvdP21b] for our evaluation and create corresponding Theodolite benchmarks. These benchmarks use the number of generated messages per second as load type, the number of instances as resource type, and employ our lag trend SLO. As discussed by Vonheiden [Von21a], we made some adjustments to the OSPBench implementations as well as fixed errors with its load generators.

14.3 Results and Discussion

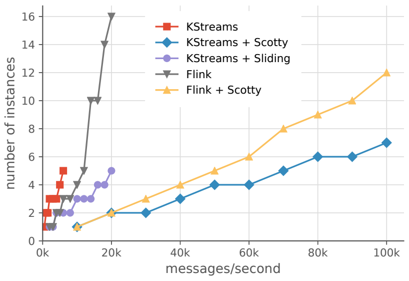
In the following we summarize our results of benchmarking the scalability of different methods for sliding window aggregations. For further discussion of the results, we refer to the work of Vonheiden [Von21a].

[1]The Yahoo Streaming Benchmark [CDE+16] uses a custom implementation for window aggregations.

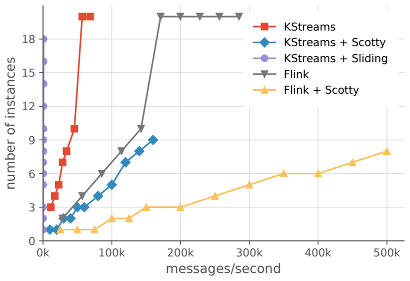
[2]`https://github.com/Klarrio/open-stream-processing-benchmark`



(a) Window size of 30 days and a sliding period of 1 day (i.e., 30 overlapping windows)



(b) Window size of 30 seconds and a sliding period of 1 second (i.e., 30 overlapping windows)



(c) Window size of 5 minutes and a sliding period of 1 minute (i.e., 5 overlapping windows)

Figure 14.1. Scalability benchmark results of different sliding window aggregation methods with Theodolite's UC3 benchmark according to our resource demand metric [Von21a].

14.3.1 Theodolite's UC3 Benchmark

Fig. 14.1 shows the experimental results for our experiments with Theodolite's UC3 benchmark. In all experiments, we cannot observe any scalability limits and all configurations scale approximately linear with increasing load. As already noticed in other experiments (e.g., Section 12.4), we again observe that Flink provides better performance for specific instance counts (e.g., 10 or 20 instances with 40 Kafka partitions).

The implementations with Scotty have a considerably lower resource demand compared to all native aggregation methods of the stream processing framework. For short windows, where the rate of incoming messages corresponds approximately to the slide period (see Fig. 14.1b), Kafka Streams' sliding window aggregation requires fewer instances compared to hopping window aggregation. For all other window sizes, the resource demand with Kafka Streams' sliding window increases significantly steeper.

Note that in these experiments the scalability of Kafka Streams' and Flink's default window aggregation methods cannot directly be compared as we use different configurations of the frameworks. Comparing both frameworks is addressed in the next section.

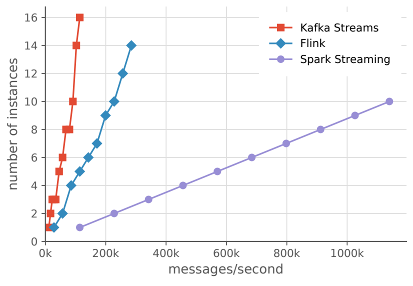
14.3.2 OSPBench

We compare sliding window aggregations of Kafka Streams, Flink, and Spark Streaming using our Theodolite benchmarks built on top of OSPBench. Fig. 14.2 shows the results of our benchmark execution. Independent of the window size and the sliding period, we can observe that Spark's resource demand increases the slowest, followed by Flink and Kafka Streams. The superior performance of Spark is presumably due to Spark's processing mode. In contrast to Flink and Kafka Streams, Spark uses micro-batching, resulting in significantly higher throughput but also increased latency [vDvdP20].

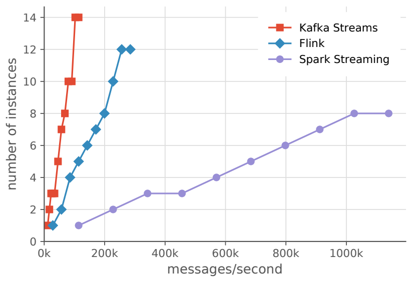
14.4 Threats to Validity

The threats to validity discussed in Section 12.6 and Section 13.6 largely apply to the evaluations of this chapter as well. In the following, we briefly report on additional threats and limitations to the validity of this chapter's evaluation.

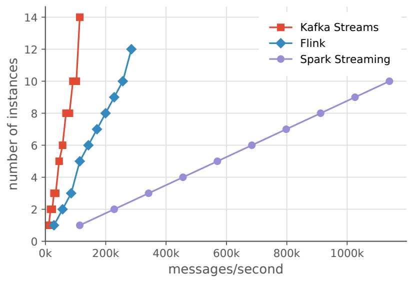
Threats to Internal Validity It should be noted that the benchmarked frameworks are based on different stream processing semantics and use different default configurations. This also applies to the different semantics of hopping and sliding windows in Kafka Streams. Moreover, Scotty



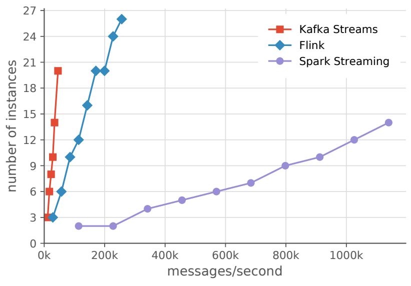
(a) Window size of 2.5 minutes and a sliding period of 0.5 minutes (i.e., 5 overlapping windows)



(b) Window size of 5 minutes and a sliding period of 1 minute (i.e., 5 overlapping windows)



(c) Window size of 10 minutes and a sliding period of 2 minutes (i.e., 5 overlapping windows)



(d) Window size of 5 minutes and a sliding period of 0.25 minutes (i.e., 20 overlapping windows)

Figure 14.2. Scalability benchmark results of different stream processing frameworks for a sliding window aggregation with OSPBench according to our resource demand metric [Von21a].

appears to rather align with Flink's dataflow model than with Kafka Streams. In our evaluations, we focus on using default configurations or configurations comparable to our previous evaluations [HH21e].

Threats to External Validity The OSPBench task sample uses a distributive aggregation and Theodolite's UC3 benchmark uses an algebraic aggregation. As we benchmarked no task sample with a holistic window aggregation, our results cannot be generalized to this type of aggregations.

Chapter 15

Evaluating Cost Scalability of Stream Processing and Function-as-a-Service

As described in Section 4.3.2, Function-as-a-Service (FaaS) is an alternative approach to implement event-driven microservices. In this chapter, we compare the scalability of distributed stream processing with FaaS. We use our scalability benchmarking method presented in this thesis along with our benchmarks for event-driven microservices.

Due to their fundamentally different deployment model, comparing FaaS and stream processing frameworks regarding their computing resource usage is neither practical nor desirable. From a cloud customer's perspective, underlying hardware resources are hidden and automatically scaled, meaning that cloud customers have no insights into how resource demands evolve. Moreover, the key advantage of serverless deployments is that cloud customers do not have to care about the actual resource usage of their deployments as they are billed per processed data volume and execution time. Instead, customers are rather interested in evaluating how costs evolve with increasing load on their system. This is also referred to as cost scalability [BHI+17].

This chapter builds upon our work previously presented at the *IEEE International Conference on Cloud Engineering 2022* [PHS+22b]. We start with an overview of our evaluation setup and methodology (Section 15.1). After an initial comparison of stream processing and FaaS (Section 15.2), we explore the space of implementation and deployment options. Specifically, we evaluate the impact of chosen transport method (Section 15.3), cloud service provider (Sections 15.4 and 15.5), FaaS runtime environment (Section 15.6), stream processing framework choice (Section 15.7), a serverless stream processing offering (Section 15.8), and a managed Kubernetes service (Section 15.9). Afterward, we discuss threats to validity in Section 15.10. Finally, we summarize our results and derive decision guidelines for choosing among stream processing and FaaS for implementing event-driven microservices (Section 15.11).

15.1 Experimental Setup and Methodology

For the evaluations of this chapter, we conduct an exploratory experiment design. We start with our baseline experiments comparing stream processing and FaaS. Afterward, we modify different properties of our benchmark implementation and deployment and compare the results with our baseline experiments. The respective setup and relevant modifications are described in the following sections. An overview of our experiment setups for stream processing and FaaS is given in Tables 15.1 and 15.2, respectively.

From our Theodolite benchmarks, we select task sample UC1 as an example for a stateless use case and UC3 as an example for a stateful one. In contrast to the previous evaluations, our Theodolite load generator sends generated messages via HTTP in an open workload model [BWT17] or via a serverless cloud messaging service. We configure benchmark UC1 to write all incoming messages to a real serverless cloud database. Similar to one of the evaluations in Chapter 14, we use a simplified version of our benchmark UC3 with 30 second windows as discussed in the following section. For our stream processing experiments, we use the Apache Beam implementations, already evaluated in Chapter 13. For the FaaS experiments, we implement the benchmarks accordingly.

We evaluate scalability regarding the number of simulated sensors and, thus, messages per second. As already introduces, the resource dimension of these evaluations is an hourly cost estimate. To yield such an estimate, we can leverage different kinds of information provided by cloud platforms or measurements. For experiments on FaaS platforms with pay-per-request pricing models, cost estimates can be derived by extrapolating from small-scale environments, as the cost can be expected to scale linearly with the

Table 15.1. Overview of stream processing deployments considered (changes over the baseline marked in **bold**)

	GCP (Baseline)	GCP Pub/Sub	AWS	GCP Samza	GCP Dataflow	GCP Autopilot
Cloud	GCP	GCP	**AWS**	GCP	GCP	GCP
Kubernetes	GKE	GKE	**EKS**	GKE	—	**GKE Autop.**
VM Type	`e2-standard-4`	`e2-standard-4`	**`m5.xlarge`**	`e2-standard-4`	`e2-standard-4`	—
Framework	Apache Flink	Apache Flink	Apache Flink	**Apache Samza**	**Dataflow**	Apache Flink
Transport	HTTP & Kafka	**Pub/Sub**	HTTP & Kafka	HTTP & Kafka	**Pub/Sub**	HTTP & Kafka
Database	Cloud Firestore	Cloud Firestore	**DynamoDB**	Cloud Firestore	Cloud Firestore	Cloud Firestore

Table 15.2. Overview of FaaS deployments considered (changes over the baseline marked in **bold**)

	GCP (Baseline)	GCP Pub/Sub	AWS	GCP Go	GCP NodeJS
Cloud	GCP	GCP	**AWS**	GCP	GCP
FaaS Engine	Cloud Functions	Cloud Functions	**AWS Lambda**	Cloud Functions	Cloud Functions
Memory	256 MB	256 MB	256 MB	256 MB	256 MB
Language	Java	Java	Java	**Go**	**NodeJS**
Transport	HTTP	**Cloud Pub/Sub**	HTTP	HTTP	HTTP
Database	Cloud Firestore	Cloud Firestore	**DynamoDB**	Cloud Firestore	Cloud Firestore

number of requests for current cloud pricing models. Additionally, any costs for database reads and writes can be derived by tracking database access and calculating the resulting cost based on the per-request cost of the database system used. To achieve a cost estimate for a stream processing deployment, we use our Theodolite scalability benchmarking method and run multiple experiments for different Kubernetes cluster sizes. For each generated load intensity, we determine the cluster size with the lowest cost, which is still able to process the generated load without violating specified SLOs. As in our previous experiments with benchmark UC3, we focus on the lag trend SLO.

We provide a replication package containing all implementations as well as the collected data of our experiments as supplemental material [PHS+22a], allowing other researchers to repeat and extend our work.

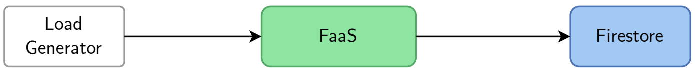
15.2 Baseline: Stream Processing and FaaS

As our baseline, we compare Google Cloud Functions and Apache Flink, running Apache Beam pipelines on Google Kubernetes Engine (GKE).

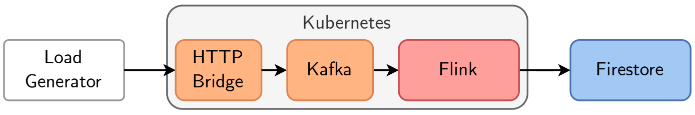
15.2.1 Implementation

In the stateless storage use case (UC1), client events are sent over HTTP and stored in Google Cloud Firestore (see Fig. 15.1a). We choose Firestore for its pay-as-you-go model that fits the serverless pricing model. As necessary for Apache Flink, HTTP events are enqueued in Apache Kafka by a middleware prior to processing (see Fig. 15.1b). Cloud Functions, on the other hand, can directly expose an HTTP endpoint.

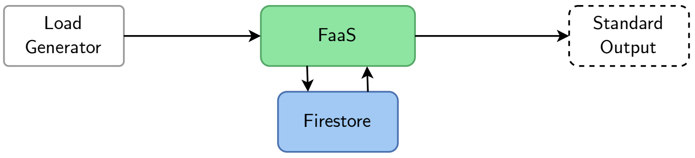
The stateful window aggregation application (UC3) also receives events over HTTP, but results are emitted to the output log of the respective platform. In a real application, a further stateless operation such as UC1 might be performed afterward, yet our goal here is to study the stateful operator in isolation. For our implementation with Flink, we use the built-in window aggregation mechanisms with RocksDB as state backend (see Fig. 15.1d). To support stateful window aggregation on stateless functions, we store intermediate window state in a Google Cloud Firestore collection



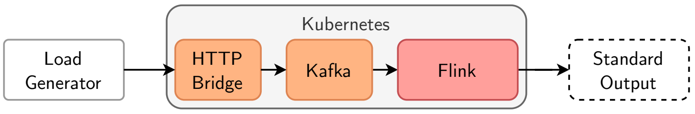
(a) UC1 as FaaS implementation



(b) UC1 as streaming implementation



(c) UC3 as FaaS implementation



(d) UC3 as streaming implementation

Figure 15.1. Implementations in our baseline benchmarks [PHS+22b]: To aggregate data across multiple events, the FaaS implementation is connected to a Firestore database to persist state. As Apache Beam running on top of Apache Flink cannot process HTTP requests directly, we add an HTTP bridge and Apache Kafka.

for each window (see Fig. 15.1c). Both implementations are configured to aggregate data over windows of 30 seconds, with a new window starting every 3 seconds. This results in 10 windows per emulated sensor that are maintained in parallel.

As Apache Flink and its operators are implemented in Java, we also use the Java 11 runtime for our cloud functions to account for effects caused by programming language or runtime. We set the function memory to 256 MB, which is the smallest amount that can support a function execution without running into memory errors. This also limits our per-function compute resources to 0.1667 vCPU.

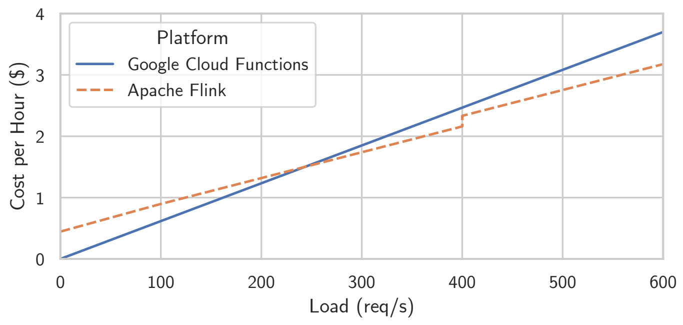
For our streaming implementation, we deploy Flink in a GKE cluster with different numbers of `e2-standard-4` virtual machines. The overall deployment consists of one coordinating Flink JobManager, varying numbers of Flink TaskManagers, a three-broker Apache Kafka cluster, a component redirecting incoming HTTP requests to Kafka as well as some additional components for monitoring and cluster management. To ensure a reasonable degree of fault tolerance, Flink is configured with a 30-second checkpointing interval and each Kafka partition is replicated across three brokers.

All experiments are conducted in the `europe-west-3` (Frankfurt) Google Cloud region, with the load generators deployed on `e2-highcpu-4` virtual machines on Google Compute Engine in the same region.

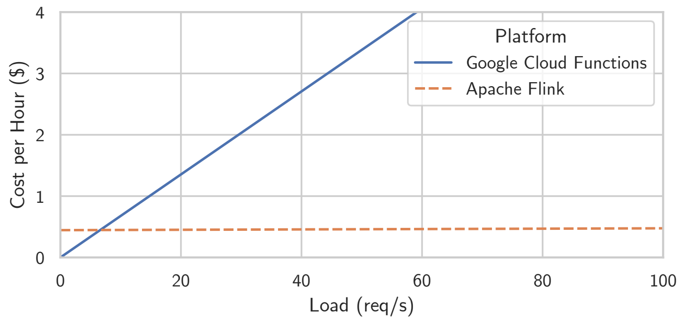
15.2.2 Results and Discussion

We show the results of our baseline evaluation in Fig. 15.2. For the application that we consider, costs scale linearly with request loads, yet at different rates. This is expected for functions, which are billed by request and where requests can be processed independently. In essence, FaaS is variable cost only. In stream processing, we instead observe a pattern of *steps*, which can be seen in Fig. 15.2a (and more pronounced at a larger scale in Fig. 15.5b). This is a result of a more coarsely grained allocation of resources, i.e., servers that need to be added to the cluster. Additionally, there is a minimum cost of running the cluster, which is the cost of a single server, a fixed rate for managing the Kubernetes cluster, and cost for the necessary load balancer. Overall, this means that stream processing costs here are a combination of fixed cost, variable cost per request, and variable cost which need to be added in batches, as shown in Fig. 15.3. This leads to the intersection of function and cluster costs at a specific request level (200 req/s for UC1 and 5 req/s for UC3): At a request rate below this level, the fixed cost of running a single-server cluster is higher than paying per request for FaaS functions. Beyond this request rate, the overhead of operating full servers in a cluster is negligible compared to the premium of serverless functions.

Interestingly, the break-even point is at a higher load rate for the stateless UC1 than for the stateful aggregation in UC3. For the cloud



(a) UC1 Costs



(b) UC3 Costs

Figure 15.2. The cost benchmark results of our baseline comparison of Apache Flink and Google Cloud Functions show how application costs scale with request load [PHS$^+$22b]. The overhead of operating a Kubernetes cluster for Apache Flink leads to higher costs compared to Cloud Functions at lower request loads. The request rate at which Cloud Functions become less economical than stream processing with Flink depends on the type of function: 200 req/s for UC1 and 5 req/s for UC3.

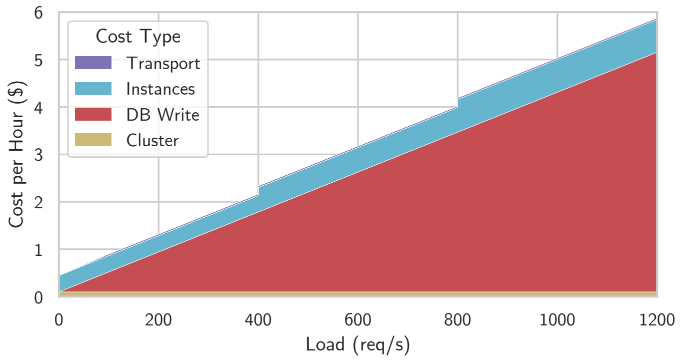


Figure 15.3. The cost breakdown of our baseline evaluation of UC1 with Apache Flink shows that total costs are composed of fixed costs (Kubernetes cluster and HTTP load balancer), costs per request (database writes), and costs increasing in batches (Kubernetes cluster nodes) [PHS+22b].

function implementation of UC3, the largest share of costs per request are caused by writes (62.2%) and reads (20.8%) to Cloud Firestore, as shown in Fig. 15.4. This database access is required to store intermediate state—in our implementation, each window is stored as a database entry, leading to ten read and write requests for each function invocation. In the streaming implementation, on the other hand, there is no such database access required since all state is maintained inside the Flink TaskManagers.

15.2.3 Takeaway for Platform Choice

Our baseline experiments show that FaaS is an economical choice over stream processing for stateless applications with low to medium event arrival rates, in our case from 0 to 200 requests per second. For stateful applications, where functions need to store intermediate state in a database, the cost of database access makes FaaS infeasible for anything but low-rate event processing.

15.3 Impact of Pub/Sub in FaaS and Streaming

While we use HTTP transport mechanism for simulated sensor data in our baseline evaluation, this does not necessarily reflect all IoT environments, where data distribution paradigms such as publish–subscribe are more common (see Chapter 10). We thus further quantify the impact of the communication pattern on stream processing and FaaS costs.

15.3.1 Implementation and Setup

We extend our baseline implementation with support for Google Cloud Pub/Sub[1]. For our function implementation, this requires adding an event trigger and application logic for event parsing. In our Apache Flink setup, we replace the previous HTTP middleware and the Apache Kafka deployment with a direct connection to Google Cloud Pub/Sub, using the *PubSubIO* connectors provided by Apache Beam. Instead of sending JSON objects as done with our HTTP implementation, we send binary encoded Apache Avro[2] records via Pub/Sub.

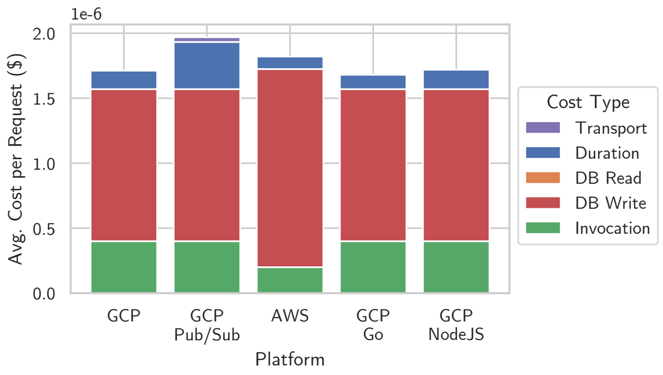
15.3.2 Results and Discussion

As shown in Fig. 15.4, using Cloud Pub/Sub has a noticeable effect on the execution duration of our FaaS implementations, especially in UC1, where processing costs increase by 154.6%. This effect is less pronounced for UC3, where duration increases by 8.6%. One possible explanation for this effect is an increased overhead caused by message parsing compared to HTTP, where request data is passed to our function directly as JSON rather than encoded. However, due to the relatively high costs of database access, this has only a small impact on total costs (12.9% increase for UC1 and 1.4% increase for UC3). At less than \$0.04 per 1,000,000 messages, the cost per Cloud Pub/Sub message is two orders of magnitude smaller than costs incurred by message processing.

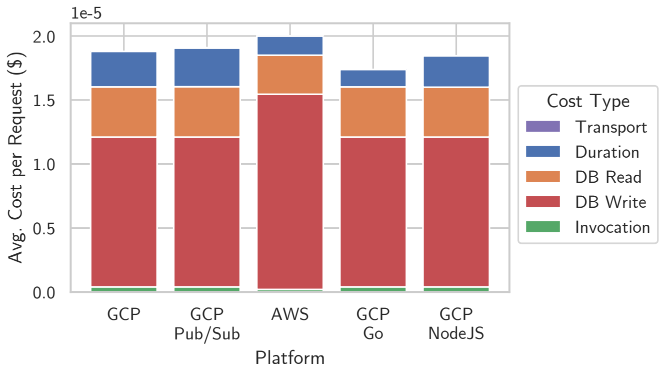
Figure 15.5 shows how costs increase with increasing load when using Cloud Pub/Sub in our Apache Flink implementation. Pub/Sub introduces

[1]`https://cloud.google.com/pubsub/`

[2]`https://avro.apache.org/`



(a) UC1



(b) UC3

Figure 15.4. FaaS cost per request by type [PHS+22b]. Breaking down the costs per requests of our cloud function implementations of the two applications in our benchmarks, we see that database access is the major cost factor. While this does not impact UC1, where both the FaaS and stream processing implementation write to Cloud Firestore and thus incur identical database access costs, storing intermediate state in UC3 accounts for 83.0% of the total cost of operating the FaaS implementation. Neither the choice of Cloud Platform, of programming language, nor of endpoint change this result significantly: AWS Lambda is 6.4% more expensive than our baseline as a result of increased DynamoDB access cost, while the choice of language runtime only changes function duration costs, which are marginal compared to Firestore access costs.

an additional cost factor to the overall deployment. These costs increase at a steeper rate than the costs for the Kubernetes cluster: While the share of Pub/Sub costs in total costs is 1.5% for UC1 and 2.9% for UC3 at a load intensity of 100 req/s, it grows to 2.6% and 17.5%, respectively, at a load of 1,000 req/s. On the other hand, these additional costs are compensated by the slightly higher loads which Flink can process with Pub/Sub before requiring an additional virtual machine. Figure 15.6 shows that, averaged over all evaluated load profiles, costs for processing messages from Pub/Sub are similar to redirecting HTTP requests via Kafka.

15.3.3 Takeaway for Transport Method Choice

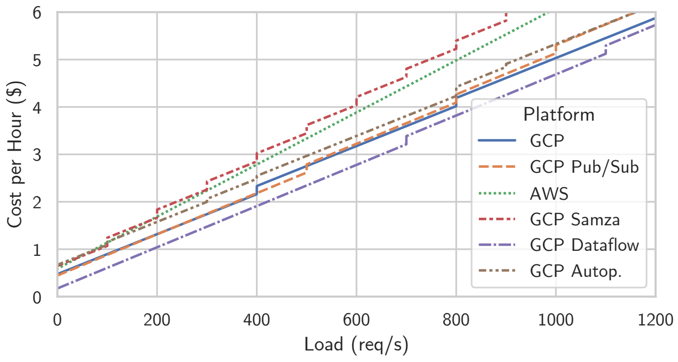
Our experiments show that there is no clear difference in costs when choosing Pub/Sub or HTTP, neither in stream processing nor in FaaS. However, small savings are possible when using a transport method that simplifies processing. Hence, it does not seem to be reasonable to add a dedicated message transform layer just to save costs.

15.4 Different FaaS Platforms

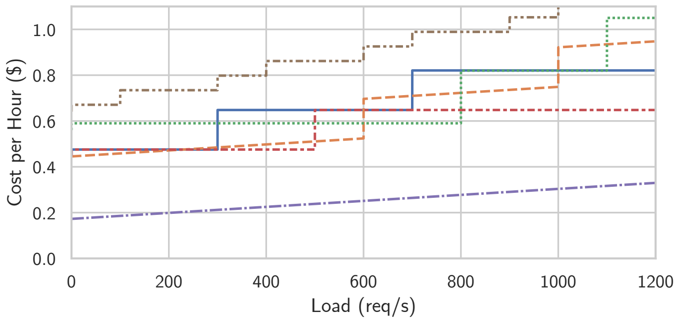
In our baseline FaaS evaluation, we use Google Cloud Functions, yet other cloud providers offer their own serverless platforms that may have different runtime behavior and pricing, impacting the cost results of our experiments. In this experiment, we thus compare our Google Cloud Function implementation with an implementation on AWS Lambda.

15.4.1 Implementation and Setup

We implement our benchmark for AWS Lambda with an AWS DynamoDB serverless database. To ensure comparability, we use the Java 11 runtime and conduct our experiments in the `eu-central-1` (Frankfurt) region. We again set the memory limit to 256 MB. Our load generator for this implementation runs in the same region on an `m5.xlarge` EC2 instance.



(a) UC1 Costs



(b) UC3 Costs

Figure 15.5. Costs increase approximately linearly for all evaluated streaming deployments [PHS+22b]. However, Google Cloud Dataflow has considerably lower costs than the other streaming frameworks.

15.4.2 Results and Discussion

As we expect the costs for function execution to scale linearly with event arrival rate, we consider the average cost for individual function execution which we show in Fig. 15.4. The average cost per function execution is 6.4% higher on AWS Lambda than on Google Cloud Functions for both applications, which is caused mainly by the more expensive database access in DynamoDB over Cloud Firestore.

15.4.3 Takeaway for Cloud Provider Choice in FaaS

In our experiments, the choice of FaaS provider had only a limited impact on the total cost of execution, yet we see that the cost difference can depend on the type of application as applications using other cloud platform services may encounter significant costs (which may vary between providers).

15.5 Different Kubernetes Engines

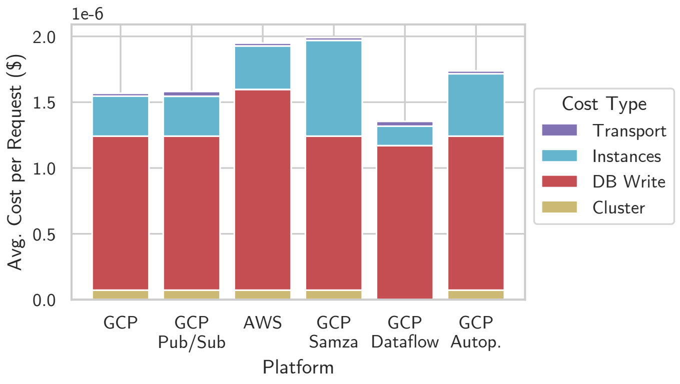
Similar to our evaluation of different FaaS Platforms, we also compare GKE and AWS Elastic Kubernetes Service (EKS).

15.5.1 Implementation and Setup

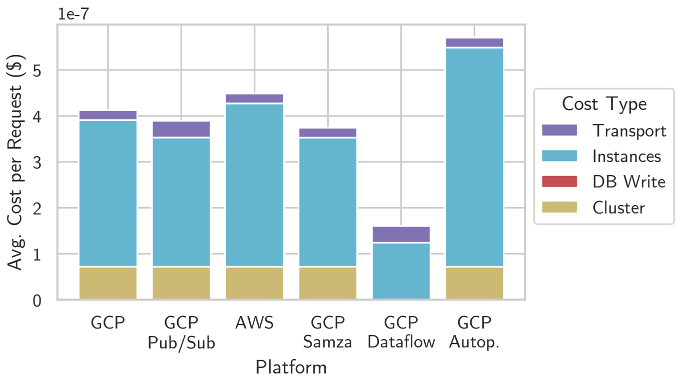
Deployment descriptions for Kubernetes are largely platform independent, allowing us to almost use the same deployment with EKS as with GKE. As in our evaluation of different FaaS Platforms, we write incoming events in our UC1 implementation to an AWS DynamoDB serverless database. Both our EKS cluster and the load generator for this implementation use `m5.xlarge` EC2 instances, running in the `eu-central-1` (Frankfurt) region.

15.5.2 Results and Discussion

As shown in Fig. 15.5a, the costs for our UC1 deployment on EKS increase at a steeper rate than in the GKE deployment. Averaged over all evaluated load profiles, EKS has 24.3% higher costs than GKE as shown in Fig. 15.6a. Interestingly, EKS has higher costs although the EKS deployment requires significantly less Flink TaskManager instances: Loads up to 1,100 req/s can be processed by a single TaskManager, compared to 8 instances required in the GKE deployment. However, higher costs per VM instance and especially higher costs per database write outweigh this superior performance. As we do not see such a difference in resource usage for UC3, we conclude that either DynamoDB provides faster writes than Firestore or Beam's DynamoDB writer is more resource efficient than the Firestore writer.



(a) UC1



(b) UC3

Figure 15.6. Stream processing cost per request by type [PHS+22b]. Averaging the cost per request over all evaluated load profiles, we see that, similar to FaaS, writing to a database is the largest cost factor for UC1 on all deployments. For UC3, costs are similar independent of the cloud provider, endpoint, and streaming framework, but instance costs are considerably lower for Dataflow and higher for GKE Autopilot.

In our implementation of the stateful application, we use only native Apache Beam functionality. As shown in Fig. 15.5b, costs increase in EKS at a similar rate as in GKE. Depending on the load intensity, at which VMs have to be added to the cluster, either GKE or EKS is cheaper. Averaged over all evaluated load profiles, EKS has 8.8% higher costs than GKE (see Fig. 15.6b). This is in accordance with the slightly higher costs per VM instance in AWS.

15.5.3 Takeaway for Cloud Platform Choice in Streaming

Similar to our findings from evaluating different FaaS platforms, the choice of cloud infrastructure for running a stream processing framework has a small but noteworthy impact on the total costs. The discrepancy results mainly from different costs for cloud resources, which even outweigh significant performance gaps.

15.6 Different Programming Languages in FaaS

In our baseline FaaS evaluation, we use the Java 11 runtime in order to account for effects of programming language or runtime performance when comparing to Apache Flink. Most modern FaaS platforms support a wider variety of runtimes, and the choice of language may have an indirect impact on execution cost when an implementation requires more resources or function executions take more time.

15.6.1 Implementation and Setup

To quantify the effect of runtime choice, we implement our benchmark in Node.js and Go. Node.js is one of the most popular choices for cloud functions, while Go is the only programming language supported by Google Cloud Functions that is compiled directly to machine code and may thus have the smallest performance overhead [CYH+20].

15.6.2 Results and Discussion

As shown in Fig. 15.4, the choice of programming language has only a small effect on the cost of function execution, with overall costs changing by -1.9% and -7.5% (Go) and 0.4% and -1.9% (Node.js) for UC1 and UC3, respectively. Although the duration of a function execution changes by -22.7% and -50.8% for UC1 and UC3 with Go, the effect on costs is insignificant compared to costs for database access. Surprisingly, the Node.js implementation is as efficient as our Java implementation. This might be caused by a more mature and optimized execution environment in Google Cloud Functions, as Node.js is one of the most popular languages for FaaS functions.

15.6.3 Takeaway for Language Choice in FaaS

As the majority of costs for the execution of a function are incurred by database access and not function duration, the choice of programming language has no considerable effect on the cost of our application. For stateless applications without database access, and especially for more complex functions where the largest share of costs is incurred by execution duration rather than function invocation, comparing implementation runtimes may nevertheless be beneficial.

15.7 Different Streaming Frameworks

We use Apache Flink for our baseline evaluation, which is a stream processing framework originating in academia and extensively studied in research. In this experiment, we compare this to Apache Samza, an open source stream processing framework developed in industry at LinkedIn (see Section 4.4). Samza is built around similar concepts as Flink and can also be used to run Apache Beam pipelines.

15.7.1 Implementation and Setup

Thanks to Apache Beam, we can use exactly the same implementation for Samza as we use for Flink. In contrast to Flink, Samza does not need a dedicated coordinator, but instead uses our existing Kafka/ZooKeeper deployments for coordination among instances.

15.7.2 Results and Discussion

In case of the stateless application, we found that Samza has a significantly higher resource demand than Flink, causing higher costs as shown in Fig. 15.5a. As processing 300 requests per second already requires 14 Samza instances, we extrapolated the costs for higher loads. We assume that this huge discrepancy is because we did not enable bundling, a Beam feature, which is used in Beam's *FirestoreIO* to write multiple records as batch. Bundling is disabled per default and its usage is not documented for Samza.

With the stateful application, Samza performs similar to Flink. As, however, Samza scales in smaller steps, the rather small load profiles studied here result in slightly lower costs for Samza as shown in Fig. 15.5b.

15.7.3 Takeaway for Framework Choice

In general, different stream processing framework can be operated at similar costs. However, different feature sets and inappropriate configuration options might cause cost pitfalls, particularly when interacting with other cloud services.

15.8 Serverless vs. Serverful Stream Processing

In our baseline evaluation, we compare serverless FaaS implementations with streaming implementations running in Kubernetes. Major cloud vendors also provide managed streaming offerings, which run streaming pipelines on top of hosted stream processing engines. While requiring the same development skills than with other stream processing frameworks, serverless stream processing services can be considered an in-between of self-operated application with a stream processing framework and FaaS in terms of operational complexity.

15.8.1 Implementation and Setup

To compare the costs of self-operating a stream processing framework with a fully-managed service, we run our Apache Beam implementations on Google Cloud Dataflow with varying numbers `e2-standard-4` instances. Similar to the other frameworks, Dataflow should be used with a durable data source instead of ingesting data directly via HTTP. As we consider using a serverless stream processing service along with a self-operated Kafka cluster to be less realistic for real-world systems, we focus on processing data from Google Cloud Pub/Sub and use the Flink experiments with Pub/Sub as baseline.

15.8.2 Results and Discussion

As shown in Fig. 15.5, Google Cloud Dataflow has significantly lower costs than our Apache Flink on Kubernetes deployment. Averaged over all evaluated load profiles (see Fig. 15.6), Dataflow has 85.6% of the costs for operating Flink for UC1 and only 41.2% for UC3. This is primarily due to the massively reduced costs for the virtual machines as with Dataflow, fewer instances are required to process the same load, e.g., the stateful application can be run with a single VM at all tested load rates. We observed that costs for Dataflow could be further reduced when using smaller instances such as `n1-standard-1` ones. Additionally, there are no general managing fees for Dataflow, while Google charges customers $0.10 per hour for managing a Kubernetes cluster. The impact of this fee on total costs decreases with increasing load (see Fig. 15.3). Since the largest cost driver in the stateless application are database writes, costs are reduced less than in the stateful application. An in-depth analysis of resource efficiency advantages in Dataflow is beyond the scope of this work, but possible reasons are:

- Dataflow might in general offer a better performance than other stream processing systems.

- Apache Beam might be optimized for Google Cloud Dataflow and, as shown in Chapter 13 and previous research [HMG+19], Flink provides much better performance when running native Flink pipelines instead of using Beam.

– Flink's default configuration might not be optimal and additional tuning is required to reach comparable performance.

– Resource utilization when running Flink in small Kubernetes clusters might not be optimal.

15.8.3 Takeaway for Platform Choice

Processing event streams with Google Cloud Dataflow has significantly lower costs in our experiments compared to our Flink deployment. Thus, serverless stream processing services can be a compelling alternative to running stream processing frameworks manually in Kubernetes, reducing both operational complexity and costs.

15.9 Serverless vs. Serverful Kubernetes

Recently, cloud providers started offering managed Kubernetes services, which charge users per container resource usage instead of for the underlying VM instances. A prominent example for such a service is GKE Autopilot.[3]

15.9.1 Implementation and Setup

As autoscaling of the Kubernetes cluster takes a considerable amount of time, running dedicated experiments with GKE Autopilot is unpractical. However, we can get a reasonable cost approximation by using the results of our baseline evaluation, in which we determined the required number of Flink TaskManagers per load profile on a sufficiently dimensioned cluster. Total costs are then the costs for the TaskManagers, combined with the constant costs for other components such as Kafka, HTTP Bridge, or monitoring.

[3] `https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview`

15.9.2 Results and Discussion

Independent of the load profile and the use case, GKE Autopilot has higher costs compared to GKE's default mode (see Fig. 15.5). The relative cost difference appears to decrease with higher loads. This can be explained by a minimal cost per container that is charged independent of the actual resource usage. Moreover, the cost difference is less pronounced in the stateless application, where costs are heavily influenced by database writes (see Fig. 15.6).

15.9.3 Takeaway for Kubernetes Service Choice

While serverless Kubernetes offerings reduce the management burden, they also have higher cloud service costs. Nevertheless, costs for running self-operated stream processing framework in a serverless Kubernetes cluster are still lower than for FaaS at medium and high loads.

15.10 Threats to Validity

In the following, we briefly report on threats and limitations to the validity of this chapter's evaluation.

Threats to Internal Validity The threats to internal validity discussed in Section 12.6 and Section 13.6 mostly apply to the stream processing experiments in this chapter. We run our experiments for FaaS platforms on a small scale and extrapolate the results to obtain cost estimates for larger load intensities. While we consider this a reasonable method due to the FaaS deployment model, we cannot rule out that performance diverges under high load. For example, cloud providers might throttle or prioritize customers depending on the load.

Threats to External Validity Also the threats to external validity discussed in Section 12.6 and Section 13.6 mostly apply to the stream processing experiments in this chapter. For example, we focus on two large cloud platforms and two rather simple task samples. We cannot directly generalize our results to arbitrary other execution environments or applications. Due to our exploratory evaluation design, we do not test all deployment options versus each other. For example, we conduct our evaluations of different frameworks or programming language only in one cloud environment. However, our results regarding the cost of stream processing and FaaS show very clear differences. It is unlikely that experiments in other execution environments or with other configurations yield fundamentally different results.

15.11 Summary of Results and Decision Guidelines

In our experiments, we quantitatively evaluated the choice between functions and stream processing for cloud event processing and have explored the impact of choosing cloud providers, endpoints, programming languages, and platforms. We see that the major influences on cost are the rate at which events arrive and the type of application. FaaS is the economic choice for applications that manage little to no state and process events with low to medium arrival rates. Stream processing is better suited for operations that require state, such as window aggregation, and for applications that process events on the order of thousands of events per second.

Beyond these considerations, we could not observe any considerable impact of other deployment parameters on costs. The choice of a specific communication pattern, such as publish–subscribe or HTTP, should thus be based not on cost but on functional differences. Similarly, the choice of cloud service provider did not influence costs significantly and might be influenced more by specific services that a provider offers.

Chapter 16

Case Studies

In the previous chapters, we evaluate our scalability benchmarking method with our proposed benchmarks for event-driven microservices. In this chapter, we now evaluate whether our proposed method can be applied to benchmark different types of existing cloud-native applications. We selected three systems as case studies, which are a commercial software for the promotional loan business (Section 16.1), an open-source research software for software visualization (Section 16.2), and a microservice reference application frequently used in research (Section 16.3). For each software system, we design a suitable scalability benchmark, execute this benchmark, and present and discuss benchmarking results.

16.1 Scalability Benchmarking of a Promotional Loan System

Enterprises and individuals can often, for example, in Germany, not directly request promotional loans. Instead they request loans via intermediary financial institutions (e.g., their house bank), which communicate with the actual loan creditor. The Promotional Loan System (PLS) of the b+m Informatik AG is a distributed software system providing this communication. It consists of components running at the house banks (the PIs) and a component running at the loan creditor (the ZI). The PLS supports different processes, for example, for loan applications or contract acceptance.

We use our Theodolite method to benchmark the scalability of the PLS. The goal of these evaluations is to assess how computing resources of the single ZI instance can be increased if the amount of data sent from connected PIs increases. Designed benchmarks should also be used to evaluate scalability of the software on a regular basis. In this section, we give a brief overview of the benchmark design, execution, and results. We refer to the work of Wetzel [Wet22] for further details.

16.1.1 Benchmark Design

We design a scalability benchmark for the PLS according to the meta model of our benchmarking tool architecture described in Chapter 8. It is defined as follows:

SUT A PLS deployments consists of multiple PI components and one ZI component. In production, each PI or ZI instance would be deployed with an individual PostgreSQL database. To save computing resources, we use a single PostgreSQL database for a set of 5 PIs in our benchmarks. Furthermore, the identity and access management provider Keycloak is used. For all our evaluations, we only use container images of the PLS and actually never had access to the PLS source code.

We enhanced observability of the PLS by deploying it with cloud-native tools such as Open Service Mesh (OSM), Prometheus, Loki and Promtail. OSM injects an *Envoy* proxy to each PI or ZI instance, which monitors incoming and outgoing network traffic and exposes corresponding performance metrics in Prometheus' data format. These metrics are periodically queried by Prometheus and, thus, can be used by Theodolite for evaluating SLOs. Loki, along with Promtail, provides similar opportunities for the application's log stream. It allows defining queries such as the rate of log messages containing a specific substring using an API, compatible with Prometheus's query language.

Load Generator Our benchmark is designed around the *message* domain entity of the PLS. Messages provide a semi-structured way to exchange information between the PIs and the ZI. Two users are required to create, control, approve, and release a message on a PI, before the PI sends it to the ZI (*four-eye principle*). Messages can be enriched by file attachments.

In our benchmarks, we simulate two types of users of the web-based user interface. One type of user creates messages and submits them for approval. The other type of users releases them. Before creating or releasing a set of messages, users perform a login. Afterward, they logout again. This results in a quite complex sequence of HTTP requests to the PIs and Keycloak, including redirects, cookie management, etc. [Wet22]. For both user types, we create trace-based workloads [BWT17] for the JMeter load generator tool, which we deploy within our cluster.

Load Dimension We focus on two dimensions of increasing load on the PLS. First, we increase the number of messages sent per minute. Second, we increase the size of messages' file attachments. Both are reported to be of special interest by b+m Informatik.

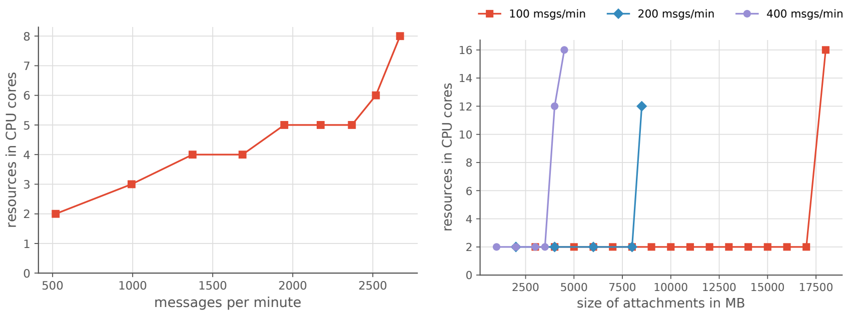
Resource Dimensions The PLS ZI component is not designed to run in a distributed fashion. Instead we focus on scaling the computing CPU and memory resources of the single ZI instance to cope with increasing load.

SLOs Since there are no explicitly described SLOs for the PLS, we have to define custom metrics to assess whether the PLS is functioning as expected. We define three SLOs: (1) The ZI must receive on average at least 90% of the created messages. (2) The 90th percentile latency of the PI-ZI communication must be less or equal than 20 milliseconds. (3) No more than 20 requests must return an HTTP error code. We specify these SLOs as corresponding queries to Prometheus and Loki.

16.1.2 Benchmark Execution

We use the *SPEL* private cloud environment described in Section 12.1 for executing the benchmarks. In an exploratory pre-study, we run individual SLO experiments for multiple hours to assess the performance variability of our PLS deployment.[1] Within these experiments, we found a memory leak in the software only becoming apparent when generating a high load over multiple hours. We reported our finding to b+m Informatik and were provided with an updated PLS version, which we used in the following.

[1]To systematically determine the required experiment duration and number of repetitions, our evaluations described in Chapter 12 can be used as a blueprint.



(a) Increasing the message rate

(b) Increasing the size of file attachments

Figure 16.1. Scalability of the ZI component of the PLS software according to our resource demand metric [Wet22].

In a set of evaluations, we benchmark the scalability of the PLS regarding both defined load dimensions. From the pre-study, we conclude that 18 minutes experiment duration with the measurements of the first 6 minutes discarded as warm-up period as well as 5 repetitions are reasonable configurations. We scale the provisioned number of CPU cores for the ZI from 1 to 16.

16.1.3 Benchmark Results

Fig. 16.1 shows selected experimental results for both evaluated load dimensions in terms of Theodolite's demand metric. We can see that the PLS scales reasonable linearly for up to 2 500 generated messages per minute (6 cores, see Fig. 16.1a). For higher load intensities, disproportionately more cores are required. Using 12 or 16 cores does not yield an increase in processable messages.

Increasing the size of file attachments can only be handled to a very limited extent with additional resources (see Fig. 16.1b). Depending on the message frequency, attachments up to approximately 3,5 MB, 8 MB, or 18 MB can be processed by 2 cores. A constant load of larger files requires significantly more resources or cannot be handled at all.

16.2 Scalability Benchmarking of the Open-Source Research Software ExplorViz

ExplorViz [FKH17; HKZ20] is an open-source research tool for software visualization and program comprehension. It uses dynamic analysis to build a continuously updating 3D representation of a monitored software system. ExplorViz is designed as an event-driven microservice architecture running in the cloud [KH22]. The most data-intensive components are the Adapter microservice, the Landscape microservice, and the Trace microservice. The Adapter service receives incoming monitoring traces and publishes events to Kafka topics, which are consumed by the Landscape and the Trace service. All three microservices use the Kafka Streams stream processing framework.

In this case study, we show how we design and execute a scalability benchmark for ExplorViz. The ExplorViz maintainers are primarily interested in evaluating how ExplorViz scales with increasing amounts of monitoring data and being able to repeat these evaluations on a regular basis. Additionally, we evaluate whether the results of our benchmarking method can be applied to the Universal Scalability Law (USL, see also Section 6.4) [Gun07; GPT15]. In this section, we give a brief overview of the benchmark design, execution, and results. For further details, we refer to the work of Ehrenstein [Ehr22].

16.2.1 Benchmark Design

Our Theodolite benchmarks for ExplorViz are defined as follows:

SUT ExplorViz is provided as a set of container images. We use these images to define corresponding Kubernetes manifest for all components. We define four benchmarks with different SUTs: (1) the Adapter microservice, (2) the Landscape microservice, (3) the Trace microservice, and (4) the entire trace analysis of ExplorViz consisting of all three microservices.

Load Generator We implement ExplorViz-specific load generators for each microservice with our Theodolite load generator framework, briefly introduced in Section 11.5. It creates a constant rate of traces of configurable size. Depending on the ExplorViz microservice under test, it either creates *Protocol Buffers* or *Apache Avro* messages.

Load Dimension In all benchmarks, the load dimension is the number of generated traces per second.

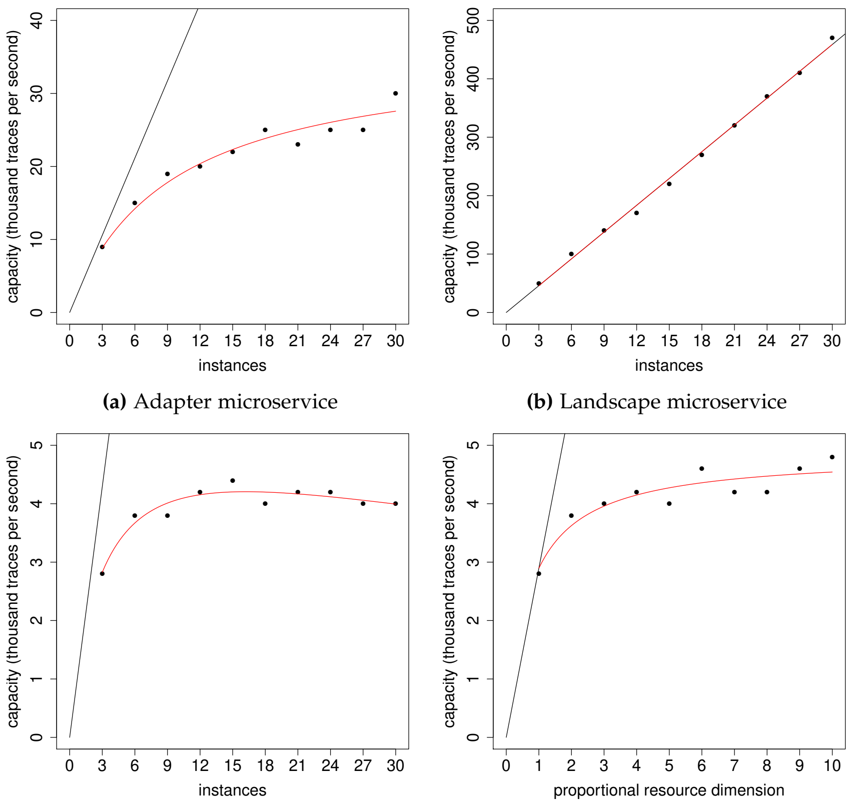
Resource Dimensions In our benchmarks for individual ExplorViz microservices, we scale with the number of instances. In the benchmark with all trace analysis microservices as SUT, we support scaling all services according to a specific ratio as well as scaling the CPU resources per service instance with a fixed number of instances per service. In both cases, this ratio is two instances of the Adapter microservice and three instances of the Trace microservice per one instance of the Landscape service.

SLO Since the SUTs studied in this section are event-driven microservices, we can apply our lag trend SLO and our dropped records SLO as defined in Section 11.4. Both SLOs are configured with a threshold of 5% of the generated load intensity.

16.2.2 Benchmark Execution

We use the *SPEL* private cloud environment described in Section 12.1 for executing the benchmarks. We deploy up to 30 microservice instances for the benchmarks with a single microservice as SUT and up to 60 instances for the benchmark with the entire trace analysis as SUT (20 instances of the Adapter microservice, 10 instances of the Landscape microservice, and 30 instances of the Trace microservice). Depending on the benchmark, we generate up to 500 000 traces per second.

In contrast to the other case studies, we use our resource capacity metric. This allows us to prototypically evaluate scalability with the USL. We use the determined load capacity for each evaluated number of instances to fit the non-linear rational function of the USL. This function provides us two coefficients, representing contention and coherency of the SUT.



(a) Adapter microservice

(b) Landscape microservice

(c) Trace microservice

(d) All trace analysis microservices

Figure 16.2. Scalability of ExplorViz microservices benchmarked with our load capacity metric and the Universal Scalability Law [Ehr22].

16.2.3 Benchmark Results

Fig. 16.2 shows selected scalability benchmarking results for each SUT. The red curve shows the fitted USL function. The black line corresponds to linear scalability with a growth rate of the USL's γ coefficient [Ehr22].

The Adapter microservice scales sublineraly. We observe mainly contention and very small coherency effects (see Fig. 16.2a). However, as argued by Ehrenstein [Ehr22], the contention seems to result from utilizing the underlying hardware and not to be related to the architecture of the Adapter service. The Landscape microservice scales linearly (see Fig. 16.2b). We do not see any contention or coherency effects within the evaluated load and resource ranges. For the Trace service, we can clearly observe both contention and coherency (see Fig. 16.2c). The load capacity increases with up to 15 instances, but decreases again with more instances. An in-depth analysis of the Trace service's architecture and potentially further experiments are necessary to evaluate whether this results from accessing shared resources or too strict SLOs [Ehr22]. The scalability results of the entire trace analysis consisting of all three microservices (see Fig. 16.2d) look quite similar to the results for the Trace service. This clearly indicates that the Trace microservice is the bottleneck of the entire trace analysis.

16.3 Scalability Benchmarking of the TeaStore Microservice Reference Application

The TeaStore [vKES+18] is a microservice reference application for benchmarking, performance modeling, and resource management research. It resamples a web shop for tea, allowing customers, for example, to browse the shop catalog, receive product recommendations, or place orders. The TeaStore consists of six microservices, which synchronously communicate via REST interfaces, and a MariaDB database. It has been used in several studies, for example, for research on microservice performance testing [EBS+20], detecting performance regressions [LCL+21], and improving performance of containerized deployments [GGB+20].[2] We thus consider it a reasonable case study for Theodolite.

In this section, we briefly show how a Theodolite benchmark for the TeaStore can be designed and executed. We provide a step-by-step guide as part of our Theodolite documentation.[3] Further, we show scalability

[2]An exhaustive list of scientific publications leveraging the TeaStore can be found at the project's website: `https://github.com/DescartesResearch/TeaStore#the-teastore-in-action`

[3]`https://www.theodolite.rocks/example-teastore.html`

benchmark results, which have previously been presented at the *13th Symposium on Software Performance* [HWH23].

16.3.1 Benchmark Design

Our Theodolite benchmark for the TeaStore is defined as follows (see also Fig. 16.3):

SUT The TeaStore comes with Kubernetes files, which we can directly use in our benchmark. While the TeaStore integrates application-level monitoring with Kieker [vHWH12; HvH20], we require higher-level metrics such as latency of requests between services. Therefore, we deploy the TeaStore along with Open Service Mesh (OSM) as in Section 16.1 to monitor incoming and outgoing network traffic. To actually see scaling effects, we restrict the containers of the WebUI, Image, Auth, and Recommender microservices to 0.5 CPU cores and 1 GB memory.

Load Generator For generating load on the TeaStore, we deploy JMeter within our cluster and use the *browse* profile [vKES+18], provided as part of the TeaStore.

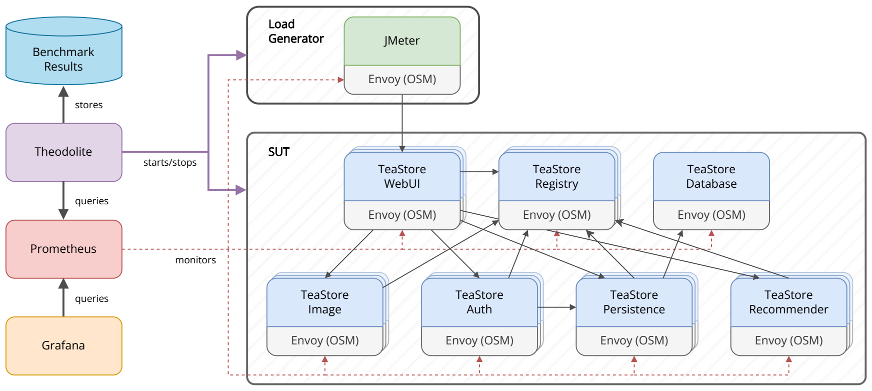


Figure 16.3. Benchmark deployment of the Theodolite stack, the TeaStore as SUT, and the JMeter load generator [HWH23].

Load Dimension The load with which we scale is the number of concurrent users configured in JMeter. Each user creates a sequence of requests to the WebUI service, resulting in approximately 25–30 requests per user and second.

Resource Dimensions We choose two types of resource scaling: (1) We scale the number of instances for each of the WebUI, Image, Auth, and Recommender services (benchmarking horizontal scalability). (2) We stick to one instance per service, but scale the amount of provided CPU cores and memory for each service (benchmarking vertical scalability).

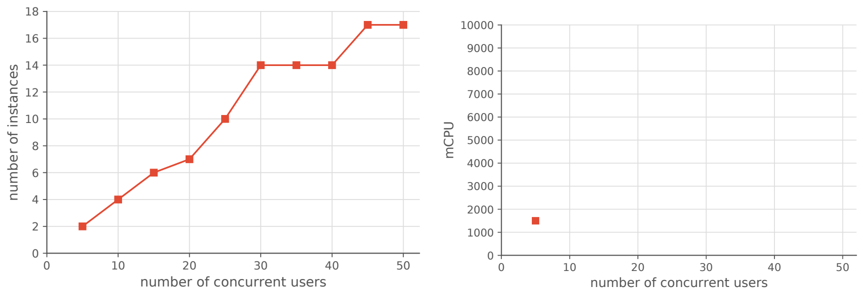
SLO To consider a certain load intensity to be handleable by a certain amount of instances, we require that the 95th percentile latency of requests to the WebUI service does not exceed 200 ms.

16.3.2 Benchmark Execution

Again we use the *SPEL* private cloud environment described in Section 12.1 to execute the benchmarks. For all benchmark executions, we use the *load capacity* metric with Theodolite's lower bound, linear search strategy. In manual experiments, we found that after 10 minutes of warmup the WebUI response latencies are quite stable. Thus, we run each experiment of a certain load intensity with a certain amount of resources for 20 minutes, while discarding the measurements of the first 10 minutes. For benchmarking horizontal scalability, we vary the number of pod instances from 1 to 20, while for vertical scalability we vary the pod's CPU resources from 0.5 to 8 cores and, proportionally, the pod's memory from 1 GB to 16 GB. We generate load with 5 to 50 concurrent users, resulting in a runtime of over 12 hours per benchmark execution.

16.3.3 Benchmark Results

Fig. 16.4 shows the results of our scalability benchmark. For benchmarking horizontal scalability, we can see that the required number of pods (for each microservice) scales approximately linearly with the amount of concurrent users. In our benchmark execution for vertical scalability, we observed



(a) horizontal scalability **(b)** vertical scalability

Figure 16.4. Results for benchmarking the horizontal and vertical scalability of the TeaStore according to our resource demand metric [HWH23].

that 5 concurrent users could be served by providing 1.5 CPU cores and 3 GB memory to each service. Higher amounts of concurrent users could not be handled, irrespective of the provisioned resources. While a detailed analysis is beyond the scope of this case study, we suspect that higher loads could still be processed by tuning the number of threads or accepted connections.

Part V

Conclusions and Future Work

Chapter 17

Conclusions

In this thesis, we addressed the lack of a suitable scalability benchmarking method for cloud-native applications and, in particular, for event-driven microservices. We defined two overarching goals, namely, to design a scalability benchmarking method for cloud-native applications and to design benchmarks that allow assessing and comparing the scalability of different stream processing frameworks, configuration options, and deployment options for event-driven microservices.

In this chapter, we summarize our contributions in terms of a scalability benchmarking method (Section 17.1), scalability benchmarks for event-driven microservices (Section 17.2), and evaluation results (Section 17.3).

17.1 The Theodolite Scalability Benchmarking Method

We proposed our Theodolite scalability benchmarking method. It allows experimentally evaluating the scalability of arbitrary cloud-native applications. Based on the distinction between individual benchmark components and benchmark quality attributes, our proposed benchmarking method consists of scalability metrics, a measurement method, and an architecture for a benchmarking tool. Regarding the design of the individual benchmark components, we raised three research questions RQ 1.1–1.3.

We addressed RQ 1.1 by designing our Theodolite scalability metrics, namely the resource demand and the load capacity metric, to quantify scalability. Based on established scalability definitions, both metrics quantify scalability using the notions of load, resources, and SLOs. Our resource demand metric quantifies how the minimum amount of required resources evolves as load increases, while the load capacity metric quantifies how the maximum processible load evolves as provisioned resources increase.

Our Theodolite measurement method addresses RQ 1.2. It defines a method for systematically conducting experiments to measure scalability according to our Theodolite metrics. The fundamental principle is to run isolated experiments for different load intensities and provisioned resources, which assess whether specified SLOs are fulfilled. Our method provides different search strategies and other configuration options to balance time-efficient execution and statistical grounding.

We designed a software architecture for a scalability benchmarking tool to address RQ 1.3. Our architecture is built around the common distinction of actors involved in benchmarking, benchmark designers and benchmarkers. Based on this distinction, we present a meta model to describe and relate scalability benchmarks and their execution. We propose to build a corresponding benchmarking tool as Kubernetes Operator. This allows for seamless integration into the cloud-native ecosystem and for defining scalability benchmarks and benchmark executions in declarative files to foster usability and reproducibility.

Our Theodolite benchmarking framework is an open-source implementation of our proposed architecture and, hence, our proposed benchmarking method. Theodolite has been-peer reviewed and successfully evaluated by the SPEC Research Group for its high quality and relevance to the software performance engineering community.

17.2 The Theodolite Scalability Benchmarks for Event-Driven Microservices

Building upon our Theodolite benchmarking method for cloud-native applications, we designed a set of scalability benchmarks for event-driven microservices, a special type of cloud-native applications. Our Theodolite benchmarks provide four task samples to assess the scalability of different stream processing frameworks, configuration options, and deployment options for event-driven microservices. We raised four research questions RQ 2.1-2.4 concerning the identification of relevant task samples, corresponding dataflow architectures for stream processing frameworks, load and resource dimensions to be evaluated, and respective SLOs.

We addressed RQ 2.1 by identifying real use cases for event-driven microservices from an IIoT analytics platform. From these use cases, we derived four dataflow architectures to be implemented by stream processing frameworks, hence, addressing RQ 2.2. These dataflow architectures serve as a basis for our benchmark task samples. Receiving a stream of simulated sensor measurements as input, individual dataflow architectures fulfill the tasks of: (1) writing all measurements to a database, (2) downsampling the measurement frequency, (3) aggregating measurements based on a time attribute, and (4) hierarchically aggregating measurements in groups.

As an answer to RQ 2.3, we identified the number of simulated sensors (i.e., messages with distinct keys per second) as a particularly relevant load dimension and the number of instances and the number of CPU cores as particularly relevant resource dimensions. We addressed RQ 2.4 by presenting the lag trend SLO and the dropped records SLO. However, our proposed benchmarking method allows benchmarking scalability with respect to arbitrary other load dimensions, resource dimensions, and SLOs.

We provide open-source implementations of the presented dataflow architectures for the state-of-the-art stream processing frameworks Apache Flink, Hazelcast Jet, Apache Kafka Streams, and the Apache Beam abstraction layer. Specific implementations for the latter are provided for Apache Flink and Apache Samza. Moreover, we provide benchmark definitions to be used with our Theodolite benchmarking framework (i.e., Theodolite Kubernetes YAML files) for all implementations and identified load dimensions, resource dimensions, and SLOs.

17.3 Experimental Evaluation Results

We conducted extensive experimental evaluations of and with our scalability benchmarking method and our benchmarks for event-driven microservices.

We analyzed the trade-off between a time-efficient execution and statically grounded results for the case of event-driven microservices. We ran experiments in two public and one private cloud infrastructure and found that in most cases only little ($\leqslant 5$) repetitions and short execution times ($\leqslant 5$ minutes) are necessary to assess whether certain resource amounts can handle a load intensity. Additionally, our results show that for both our scalability metrics, search strategies can be used to massively reduce the amount of individual experiments.

We employed our benchmarking method along with our stream processing benchmarks to assess the scalability of state-of-the-art stream processing frameworks, particularly suited for implementing event-driven microservices. We found that all benchmarked frameworks provide approximately linear scalability. However, we noticed considerable differences among the frameworks in terms of the rate at which resource demands increase. Moreover, no framework is clearly superior. Rather, it depends on the use case, which framework scales with the lowest resource demand.

Our benchmarking method does not only allow comparing different stream processing frameworks, but also evaluating different configuration options and algorithms developed by research. We employed one of our Theodolite benchmarks as well as another benchmark provided by research to evaluate scalability of different sliding window aggregation methods. While all methods provide linear scalability, we found that the Scotty window processor can reduce resource demands significantly.

We employed our Theodolite benchmarking method as well as our stream processing benchmark to compare cost scalability of stream processing frameworks running in Kubernetes with FaaS offerings. We found that in terms of pure costs for the cloud services, FaaS is superior for applications that are subject to small load intensities and maintain small state. Once load intensities increase and utilize at least a single stream processing instance, stream processing systems can be operated at lower costs. This observation holds independent of the cloud provider and technology employed for implementation.

In three case studies, we found that our benchmarking method can also be applied to evaluate scalability of other types of cloud-native applications. Specifically, our Theodolite method allows using existing deployment specifications without requiring access to the application's source code.

Chapter 18

Future Work

We complete this thesis by pointing out future research directions. In this chapter, we primarily show how our benchmarking method and our benchmarks allow future work to conduct experimental scalability evaluations (Section 18.1). Moreover, we outline future work regarding our scalability benchmarking method (Section 18.2) and our scalability benchmarks for event-driven microservices (Section 18.3). We conclude this thesis with a small outlook (Section 18.4).

18.1 Experimental Scalability Evaluations with Theodolite

As summarized in Section 17.1, we designed a scalability benchmarking method for cloud-native applications. It lays the foundation for future research, defining new benchmarks, adopting existing ones (e.g., addressing performance), and executing them.

With this thesis, we put a focus on event-driven microservices, a special type of cloud-native applications, and designed corresponding scalability benchmarks for stream processing frameworks (see Section 17.2). This allows for a wide range of further research on scalability of event-driven microservices. Building upon our experimental evaluations in Part IV, future work could involve benchmarking the scalability of different deployment and configuration options, different algorithms, or load intensities of other dimensions.

Frequently executing scalability benchmarks, for example, as part of the continuous integration or continuous deployment process could assist quality assurance and regression testing. While continuous benchmarking has been occasionally proposed in research [WEH15; GLB19], it still seems not to be established, in particular, at a larger scale than microbenchmarking. We suspect that one cause is usability hurdles. Implementing a benchmarking tool as Kubernetes operator as proposed in this thesis and considering benchmarks as Infrastructure as Code, however, may significantly improve usability [HWH21]. Placing benchmarks defined in Kubernetes manifest files along with application deployment files in Git repositories, may allow recently emerging cloud-native GitOps [BH22] tools to automatically run these benchmarks in the continuous deployment process.

18.2 Future Work on Scalability Benchmarking Methods

With Theodolite, we presented a systematical and automated method accompanied by a corresponding tool to evaluate whether specified load intensities can be handled by specified resource amounts without violating specified SLOs. We see great potential for using Theodolite beyond such two-dimensional scalability evaluations. For example, a large field of research concerns approaches to search the vast space of cloud deployment options for a cost-optimal one [FFH13; BEG+19]. Combined with search-based software engineering methods [HMZ12], Theodolite could contribute to this research.

Another promising future research direction is integrating Theodolite with elasticity benchmarking approaches of, for example, auto-scalers. Established elasticity benchmarking methods rely on an approach similar to ours to determine the optimal resource demand [HKW+15]. Theodolite and its benchmarks could be used to conduct the necessary experiments for cloud-native applications and, in particular, for event-driven microservices in an automated fashion.

In our case study in Section 16.2, we prototypically used the scalability results of our load capacity metric to fit a Universal Scalability Law (USL) model. Although the resulting model coefficients might require slightly alternative interpretations [Ehr22], our initial results look promising. The major advantage of deriving such coefficients is that they improve interpretation and ranking of results. Before analyzing cloud-native application with the USL at a larger scale, however, additional experiments and a statistically sound evaluation should be performed. If these are successful, we can also speed up benchmark execution using an USL-based search strategy. Ehrenstein [Ehr22] built a prototypical extension of such a search strategy for our Theodolite benchmarking framework. It selects load intensities and resource amounts, for which SLO experiments are executed, to incrementally refine the USL model.

18.3 Future Work on Scalability Benchmarks for Event-Driven Microservices

With our Theodolite scalability benchmarks, we focused on four task samples. In Chapter 10, we identified further types of continuous sensor data analysis not included in these task samples. Future research could design corresponding task samples based on our presented goals and measures. In particular, we found forecasting and anomaly detection on sensor data streams to be of high relevance. With our Titan Control Center, we already presented how corresponding microservices could be designed [HHB+21; HH21d]. We suggest using these architectures as a basis to complement our Theodolite benchmarks by two additional ones for forecasting and anomaly detection. In contrast to the existing ones, such dataflow architectures call an external service or perform a compute-intensive operation (forecasting) and join two data streams based on the messages timestamp (anomaly detection).

18.4 Outlook

As introduced in Chapter 1, an end to scalability requirements is not in sight. Although new technologies will emerge, cloud-native and event-driven architectures will likely remain highly relevant research topics for years to come. With this work, we contribute to a solid foundation for empirical scalability evaluations of software methods, architectures, and technologies proposed in future research.

Bibliography

[AA19] A. Al-Said Ahmad and P. Andras. "Scalability analysis comparisons of cloud-based software services". In: *Journal of Cloud Computing* 8.1 (2019), pp. 1–17. DOI: 10.1186/s13677-019-0134-y.

[AB17] A. Abedi and T. Brecht. "Conducting repeatable experiments in highly variable cloud computing environments". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE '17. 2017, pp. 287–292. DOI: 10.1145/3030207.3030229.

[ABB+13] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. "Millwheel: fault-tolerant stream processing at internet scale". In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1033–1044. DOI: 10.14778/2536222.2536229.

[ABC+15] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1792–1803. DOI: 10.14778/2824032.2824076.

[ABC+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. "TensorFlow: a system for large-scale machine learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. 2016, pp. 265–283.

[ABC+21] T. Akidau, E. Begoli, S. Chernyak, F. Hueske, K. Knight, K. Knowles, D. Mills, and D. Sotolongo. "Watermarks in stream processing systems: semantics and comparative analysis of Apache Flink and Google Cloud Dataflow". In: *Proceedings of the VLDB Endowment* 14.12 (Oct. 2021), pp. 3135–3147. DOI: 10.14778/3476311.3476389.

[ABE+14] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. "The stratosphere platform for big data analytics". In: *The VLDB Journal* 23.6 (Dec. 2014), pp. 939–964. DOI: 10.1007/s00778-014-0357-y.

[ACG+04] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. "Linear Road: a stream data management benchmark". In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB '04. 2004, pp. 480–491.

[ACJ+21] A. Avritzer, M. Camilli, A. Janes, B. Russo, J. Jahič, A. v. Hoorn, R. Britto, and C. Trubiani. "PPTAM$^\lambda$: what, where, and how of cross-domain scalability assessment". In: *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*. 2021, pp. 62–69. DOI: 10.1109/ICSA-C52384.2021.00016.

[AE08] M. Albadi and E. El-Saadany. "A summary of demand response in electricity markets". In: *Electric Power Systems Research* 78.11 (2008), pp. 1989–1996. DOI: 10.1016/j.epsr.2008.04.002.

[AF09] M. L. Abbott and M. T. Fisher. *The art of scalability: scalable web architecture, processes, and organizations for the modern enterprise*. 1st. Addison-Wesley Professional, 2009.

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. "A view of cloud computing". In: *Communications of the ACM* 53.4 (Apr. 2010), pp. 50–58. DOI: 10.1145/1721654.1721672.

[AFJ+20] A. Avritzer, V. Ferme, A. Janes, B. Russo, A. van Hoorn, H. Schulz, D. Menasché, and V. Rufino. "Scalability assessment of microservice architecture deployment configurations: a domain-based approach leveraging operational profiles and load tests". In: *Journal of Systems and Software* 165 (2020), p. 110564. DOI: 10.1016/j.jss.2020.110564.

[Amd67] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). 1967, pp. 483–485. DOI: 10.1145/1465482.1465560.

[AMP+17] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi. "Benchmark requirements for microservices architecture research". In: *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*. 2017, pp. 8–13. DOI: 10.1109/ECASE.2017.4.

[Apa22] Apache Software Foundation. *Nexmark benchmark suite*. 2022. URL: https://beam.apache.org/documentation/sdks/java/testing/nexmark/.

[AT16] S. Arora and J. W. Taylor. "Forecasting electricity smart meter data using conditional kernel density estimation". In: *Omega* 59 (2016). Business Analytics, pp. 47–59. DOI: 10.1016/j.omega.2014.08.008.

[Bas13] V. R. Basili. "A personal perspective on the evolution of empirical software engineering". In: *Perspectives on the Future of Software Engineering: Essays in Honor of Dieter Rombach*. Ed. by J. Münch and K. Schmid. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 255–273. DOI: 10.1007/978-3-642-37395-4_17.

[BBT+15] S. Brunner, M. Blöchlinger, G. Toffetti, J. Spillner, and T. M. Bohnert. "Experimental evaluation of the cloud-native application design". In: *2015 IEEE/ACM 8th International Confer-*

ence on Utility and Cloud Computing (UCC). 2015, pp. 488–493. DOI: 10.1109/UCC.2015.87.

[BCK+21] D. Bermbach, A. Chandra, C. Krintz, A. Gokhale, A. Slominski, L. Thamsen, E. Cavalcante, T. Guo, I. Brandic, and R. Wolski. "On the future of cloud engineering". In: *2021 IEEE International Conference on Cloud Engineering (IC2E)*. 2021, pp. 264–275. DOI: 10.1109/IC2E52221.2021.00044.

[BDD+18] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara. "From monolithic to microservices: an experience report from the banking domain". In: *IEEE Software* 35.3 (2018), pp. 50–55. DOI: 10.1109/MS.2018.2141026.

[BEG+19] A. Bauer, S. Eismann, J. Grohmann, N. Herbst, and S. Kounev. "Systematic search for optimal resource configurations of distributed applications". In: *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. 2019, pp. 120–125. DOI: 10.1109/FAS-W.2019.00040.

[Bel20] A. Bellemare. *Building event-driven microservices*. 1st. O'Reilly Media, Inc., 2020.

[Ben21] J. R. Bensien. "Scalability benchmarking of stream processing engines with Apache Beam". Bachelor's Thesis. Kiel University, 2021.

[Ber17] D. Bermbach. "Quality of cloud services: expect the unexpected". In: *IEEE Internet Computing* 21.1 (Jan. 2017), pp. 68–72. DOI: 10.1109/MIC.2017.1.

[BGM+20] M. V. Bordin, D. Griebler, G. Mencagli, C. F. R. Geyer, and L. G. L. Fernandes. "DSPBench: a suite of benchmark applications for distributed data stream processing systems". In: *IEEE Access* 8 (2020), pp. 222900–222917. DOI: 10.1109/ACCESS.2020.3043948.

[BGO+16] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. "Borg, Omega, and Kubernetes". In: *Communications of the ACM* 59.5 (Apr. 2016), pp. 50–57. DOI: 10.1145/2890784.

[BH22] F. Beetz and S. Harrer. "GitOps: the evolution of DevOps?" In: *IEEE Software* 39.4 (2022), pp. 70–75. DOI: 10.1109/MS.2021.3119106.

[BHI+17] G. Brataas, N. Herbst, S. Ivansek, and J. Polutnik. "Scalability analysis of cloud software services". In: *2017 IEEE International Conference on Autonomic Computing (ICAC)*. 2017, pp. 285–292. DOI: 10.1109/ICAC.2017.34.

[BHJ16a] A. Balalaie, A. Heydarnoori, and P. Jamshidi. "Microservices architecture enables DevOps: migration to a cloud-native architecture". In: *IEEE Software* 33.3 (May 2016), pp. 42–52. DOI: 10.1109/ms.2016.64.

[BHJ16b] A. Balalaie, A. Heydarnoori, and P. Jamshidi. "Migrating to cloud-native architectures using microservices: an experience report". In: *Advances in Service-Oriented and Cloud Computing*. Ed. by A. Celesti and P. Leitner. 2016, pp. 201–215.

[BHP+06] S. Becker, W. Hasselbring, A. Paul, M. Boskovic, H. Koziolek, J. Ploski, A. Dhama, H. Lipskoch, M. Rohr, D. Winteler, S. Giesecke, R. Meyer, M. Swaminathan, J. Happe, M. Muhle, and T. Warns. "Trustworthy software systems: a discussion of basic concepts and terminology". In: *ACM SIGSOFT Software Engineering Notes* 31.6 (Nov. 2006), pp. 1–18. DOI: 10.1145/1218776.1218781.

[BHT+20] L. Bulej, V. Horký, P. Tuma, F. Farquet, and A. Prokopec. "Duet benchmarking: improving measurement accuracy in the cloud". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '20. 2020, pp. 100–107. DOI: 10.1145/3358960.3379132.

[Bie20] N. A. Biernat. "Scalability benchmarking of Apache Flink". Bachelor's Thesis. Kiel University, 2020.

[BKD+17] D. Bermbach, J. Kuhlenkamp, A. Dey, A. Ramachandran, A. Fekete, and S. Tai. "BenchFoundry: a benchmarking framework for cloud storage services". In: *Service-Oriented Computing: 15th International Conference, ICSOC 2017, Malaga, Spain,* *November 13–16, 2017, Proceedings*. 2017, pp. 314–330. DOI: 10.1007/978-3-319-69035-3_22.

[BKK+09] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. "How is the weather tomorrow? towards a benchmark for the cloud". In: *Proceedings of the Second International Workshop on Testing Database Systems*. DBTest '09. 2009. DOI: 10.1145/1594156.1594168.

[BLB15] M. Becker, S. Lehrig, and S. Becker. "Systematically deriving quality metrics for cloud computing systems". In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. 2015, pp. 169–174. DOI: 10.1145/2668930.2688043.

[Ble20] A. S. Blee-Goldman. *KIP-450: sliding window aggregations in the DSL*. 2020. URL: https://cwiki.apache.org/confluence/display/KAFKA/KIP-450%3A+Sliding+Window+Aggregations+in+the+DSL.

[BMH+21] G. Brataas, A. Martini, G. K. Hanssen, and G. Ræder. "Agile elicitation of scalability requirements for open systems: a case study". In: *Journal of Systems and Software* 182 (2021), p. 111064. DOI: 10.1016/j.jss.2021.111064.

[BMZ+12] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. "Fog computing and its role in the Internet of Things". In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. 2012, pp. 13–16. DOI: 10.1145/2342509.2342513.

[Bog20] L. Boguhn. "Forecasting power consumption of manufacturing industries using neural networks". Bachelor's Thesis. Kiel University, 2020.

[Bog22] L. Boguhn. "Benchmarking the scalability of distributed stream processing engines in case of load peaks". Master's Thesis. Kiel University, 2022.

[BOH11] M. Bostock, V. Ogievetsky, and J. Heer. "D^3 data-driven documents". In: *IEEE Transactions on Visualization and Computer Graphics* 17.12 (2011), pp. 2301–2309. DOI: 10.1109/TVCG.2011.185.

[Bon00] A. B. Bondi. "Characteristics of scalability and their impact on performance". In: *Proceedings of the 2nd International Workshop on Software and Performance*. WOSP '00. 2000, pp. 195–203. DOI: 10.1145/350391.350432.

[BSL16] G. Brataas, E. Stav, and S. Lehrig. "Analysing evolution of work and load". In: *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. 2016, pp. 90–95. DOI: 10.1109/QoSA.2016.18.

[BTV+18] S. Bischof, H. Trittenbach, M. Vollmer, D. Werle, T. Blank, and K. Böhm. "HIPE: an energy-status-data set from industrial production". In: *Proceedings of the Ninth International Conference on Future Energy Systems*. e-Energy '18. 2018, pp. 599–603. DOI: 10.1145/3208903.3210278.

[Bun20] Bundesamt für Wirtschaft und Ausfuhrkontrolle (BAFA). *Merkblatt stromkostenintensive Unternehmen 2022*. Brochure. 2020. URL: http://www.bafa.de/SharedDocs/Downloads/DE/Energie/bar_merkblatt_unternehmen.pdf.

[BVS+11] K. Bunse, M. Vodicka, P. Schönsleben, M. Brülhart, and F. O. Ernst. "Integrating energy efficiency performance in production management – gap analysis between industrial needs and scientific literature". In: *Journal of Cleaner Production* 19.6 (2011), pp. 667–679. DOI: 10.1016/j.jclepro.2010.11.011.

[BWT17] D. Bermbach, E. Wittern, and S. Tai. *Cloud service benchmarking: measuring quality of cloud services from a client perspective*. 1st. Springer Publishing Company, Incorporated, 2017. DOI: 10.1007/978-3-319-55483-9.

[CCT+22] S. Choochotkaew, T. Chiba, S. Trent, T. Yoshimura, and M. Amaral. "AutoDECK: automated declarative performance evaluation and tuning framework on kubernetes". In: *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. 2022, pp. 309–314. DOI: 10.1109/CLOUD55607.2022.00053.

[CDE+16] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. “Benchmarking streaming computation engines: Storm, Flink and Spark Streaming”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2016, pp. 1789–1792. DOI: 10.1109/IPDPSW.2016.138.

[CEF+17] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. “State management in Apache Flink®: consistent stateful distributed stream processing”. In: *Proceedings of the VLDB Endowment* 10.12 (Aug. 2017), pp. 1718–1729. DOI: 10.14778/3137765.3137777.

[CFK+20] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos. “Beyond analytics: the evolution of stream processing systems”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’20. 2020, pp. 2651–2658. DOI: 10.1145/3318464.3383131.

[CIM+19] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. “The rise of serverless computing”. In: *Communications of the ACM* 62.12 (Nov. 2019), pp. 44–54. DOI: 10.1145/3368454.

[CKE+15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. “Apache Flink: stream and batch processing in a single engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[CKH19] P. Carbone, A. Katsifodimos, and S. Haridi. “Stream window aggregation semantics and optimization”. In: *Encyclopedia of Big Data Technologies*. Ed. by S. Sakr and A. Y. Zomaya. Springer, 2019, pp. 1615–1623. DOI: 10.1007/978-3-319-77525-8_154.

[CKK13] P. Chujai, N. Kerdprasop, and K. Kerdprasop. “Time series analysis of household electric consumption with arima and arma models”. In: *Proceedings of the International MultiConference of Engineers and Computer Scientists*. 2013, pp. 295–300.

[Clo18] Cloud Native Computing Foundation. *CNCF cloud native definition v1.0*. 2018. URL: https://github.com/cncf/toc/blob/main/DEFINITION.md.

[Clo22] Cloud Native Computing Foundation. *CNCF annual survey 2021*. 2022. URL: https://www.cncf.io/reports/cncf-annual-survey-2021.

[CMS17] M. Cunha, N. C. Mendonça, and A. Sampaio. "Cloud Crawler: a declarative performance evaluation environment for infrastructure-as-a-service clouds". In: *Concurrency and Computation: Practice and Experience* 29.1 (2017), e3825. DOI: 10.1002/cpe.3825.

[CS19] C. Cooremans and A. Schönenberger. "Energy management: a key driver of energy-efficiency investment?" In: *Journal of Cleaner Production* 230 (2019), pp. 264–275. DOI: 10.1016/j.jclepro.2019.04.333.

[CT14] J.-S. Chou and A. S. Telaga. "Real-time detection of anomalous power consumption". In: *Renewable and Sustainable Energy Reviews* 33 (2014), pp. 400–411. DOI: 10.1016/j.rser.2014.01.088.

[CTC+17] J.-S. Chou, A. S. Telaga, W. K. Chong, and G. E. Gibson. "Early-warning application for real-time detection of energy consumption anomalies in buildings". In: *Journal of Cleaner Production* 149 (2017), pp. 711–722. DOI: 10.1016/j.jclepro.2017.02.028.

[CWT+13] E. Cagno, E. Worrell, A. Trianni, and G. Pugliese. "A novel approach for barriers to industrial energy efficiency". In: *Renewable and Sustainable Energy Reviews* 19 (2013), pp. 290–308. DOI: 10.1016/j.rser.2012.11.007.

[CYH+20] R. Cordingly, H. Yu, V. Hoang, D. Perez, D. Foster, Z. Sadeghi, R. Hatchett, and W. J. Lloyd. "Implications of programming language selection for serverless data processing pipelines". In: *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data*

Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech). 2020, pp. 704–711. DOI: 10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00120.

[CYH20] Z. Chu, J. Yu, and A. Hamdull. "Maximum sustainable throughput evaluation using an adaptive method for stream processing platforms". In: *IEEE Access* 8 (2020), pp. 40977–40988. DOI: 10.1109/ACCESS.2020.2976738.

[DG08] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (Jan. 2008), pp. 107–113. DOI: 10.1145/1327452.1327492.

[DL20] A. Davoudian and M. Liu. "Big data systems: a software engineering perspective". In: *ACM Computing Surveys* 53.5 (Sept. 2020). DOI: 10.1145/3408314.

[DLR13] L. Duboc, E. Letier, and D. S. Rosenblum. "Systematic elaboration of scalability requirements through goal-obstacle analysis". In: *IEEE Transactions on Software Engineering* 39.1 (2013), pp. 119–140. DOI: 10.1109/TSE.2012.12.

[DM17] G. M. U. Din and A. K. Marnerides. "Short term power load forecasting using deep neural networks". In: *2017 International Conference on Computing, Networking and Communications (ICNC)*. Jan. 2017, pp. 594–598. DOI: 10.1109/ICCNC.2017.7876196.

[DRW07] L. Duboc, D. Rosenblum, and T. Wicks. "A framework for characterization and analysis of software system scalability". In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC-FSE '07. 2007, pp. 375–384. DOI: 10.1145/1287624.1287679.

[DvH17] T. F. Düllmann and A. van Hoorn. "Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ICPE '17 Companion. 2017, pp. 171–172. DOI: 10.1145/3053600.3053627.

[EBS+20] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović, and A. van Hoorn. "Microservices: a performance tester's dream or nightmare?" In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '20. 2020, pp. 138–149. DOI: 10.1145/3358960.3379124.

[Ehr19] S. B. N. F. A. Ehrenstein. "Distributed sensor management for an Industrial DevOps monitoring platform". Bachelor's Thesis. Kiel University, 2019.

[Ehr22] S. B. N. F. A. Ehrenstein. "Scalability evaluation of ExplorViz with the Universal Scalability Law". Master's Thesis. Kiel University, 2022.

[ESvE+21] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup. "Serverless applications: why, when, and how?" In: *IEEE Software* 38.1 (2021), pp. 32–39. DOI: 10.1109/MS.2020.3023302.

[EZL89] D. Eager, J. Zahorjan, and E. Lazowska. "Speedup versus efficiency in parallel systems". In: *IEEE Transactions on Computers* 38.3 (1989), pp. 408–423. DOI: 10.1109/12.21127.

[FAS+13] E. Folkerts, A. Alexandrov, K. Sachs, A. Iosup, V. Markl, and C. Tosun. "Benchmarking in the cloud: what it should, can, and cannot be". In: *Selected Topics in Performance Evaluation and Benchmarking*. Ed. by R. Nambiar and M. Poess. 2013, pp. 173–188.

[FBS22] S. Forti, U. Breitenbücher, and J. Soldani. "Trending topics in software engineering". In: *ACM SIGSOFT Software Engineering Notes* 47.3 (July 2022), pp. 20–21. DOI: 10.1145/3539814.3539820.

[FBW+19] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann. "Microservices migration in industry: intentions, strategies, and challenges". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 481–490. DOI: 10.1109/ICSME.2019.00081.

[FCK+20] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos. *A survey on the evolution of stream processing systems*. 2020. DOI: 10.48550/arxiv.2008.00842.

[FFH13] S. Frey, F. Fittkau, and W. Hasselbring. "Search-based genetic optimization for deployment and reconfiguration of software in the cloud". In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 512–521. DOI: 10.1109/ICSE.2013.6606597.

[FKH17] F. Fittkau, A. Krause, and W. Hasselbring. "Software landscape and application visualization for system comprehension with ExplorViz". In: *Information and Software Technology* 87 (2017), pp. 259–277. DOI: 10.1016/j.infsof.2016.07.004.

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer Publishing Company, Incorporated, 2014. DOI: 10.1007/978-3-7091-1568-8.

[FM12] T. Fiedler and P.-M. Mircea. "Energy management systems according to the ISO 50001 standard — challenges and benefits". In: *2012 International Conference on Applied and Theoretical Electricity (ICATE)*. Oct. 2012, pp. 1–4. DOI: 10.1109/ICATE.2012.6403411.

[FMR+22] K. Feichtinger, K. Meixner, F. Rinker, I. Koren, H. Eichelberger, T. Heinemann, J. Holtmann, M. Konersmann, J. Michael, E.-M. Neumann, J. Pfeiffer, R. Rabiser, M. Riebisch, and K. Schmid. "Industry voices on software engineering challenges in cyber-physical production systems engineering". In: *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2022, pp. 1–8. DOI: 10.1109/ETFA52439.2022.9921568.

[FP18] V. Ferme and C. Pautasso. "A declarative approach for performance tests execution in continuous software development environments". In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. 2018, pp. 261–272. DOI: 10.1145/3184407.3184417.

[GBL+96] J. Gray, A. Bosworth, A. Lyaman, and H. Pirahesh. "Data cube: a relational aggregation operator generalizing GROUP-BY, CROSS-TAB, and SUB-TOTALS". In: *Proceedings of the*

Twelfth International Conference on Data Engineering. 1996, pp. 152–159. DOI: 10.1109/ICDE.1996.492099.

[GBS17] D. Gannon, R. Barga, and N. Sundaresan. "Cloud-native applications". In: *IEEE Cloud Computing* 4.5 (2017), pp. 16–21. DOI: 10.1109/MCC.2017.4250939.

[GGB+20] S. Gholami, A. Goli, C.-P. Bezemer, and H. Khazaei. "A framework for satisfying the performance requirements of containerized software systems through multi-versioning". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '20. 2020, pp. 150–160. DOI: 10.1145/3358960.3379125.

[GGS+22a] A. M. Garcia, D. Griebler, C. Schepke, and L. G. Fernandes. "SPBench: a framework for creating benchmarks of stream processing applications". In: *Computing* (2022). DOI: 10.1007/s00607-021-01025-6.

[GGS+22b] A. M. Garcia, D. Griebler, C. Schepke, and L. G. L. Fernandes. "Evaluating micro-batch and data frequency for stream processing applications on multi-cores". In: *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 2022, pp. 10–17. DOI: 10.1109/PDP55904.2022.00011.

[GHP+19] M. Grambow, J. Hasenburg, T. Pfandzelter, and D. Bermbach. "Is it safe to dockerize my database benchmark?" In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC '19. 2019, pp. 341–344. DOI: 10.1145/3297280.3297545.

[GJK+14] T. Goldschmidt, A. Jansen, H. Koziolek, J. Doppelhamer, and H. P. Breivold. "Scalability and robustness of time-series databases for cloud-native monitoring of industrial processes". In: *2014 IEEE 7th International Conference on Cloud Computing*. 2014, pp. 602–609. DOI: 10.1109/CLOUD.2014.86.

[GKK+12] J. Grundy, G. Kaefer, J. Keung, and A. Liu. "Guest editors' introduction: software engineering for the cloud". In: *IEEE Software* 29.2 (2012), pp. 26–29. DOI: 10.1109/MS.2012.31.

[GLB19] M. Grambow, F. Lehmann, and D. Bermbach. "Continuous benchmarking: using system benchmarking in build pipelines". In: *2019 IEEE International Conference on Cloud Engineering (IC2E)*. 2019, pp. 241–246. DOI: 10.1109/IC2E.2019.00039.

[GME+15] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. "Edge-centric computing: vision and challenges". In: *ACM SIGCOMM Computer Communication Review* 45.5 (Sept. 2015), pp. 37–42. DOI: 10.1145/2831347.2831354.

[GMW+20] M. Grambow, L. Meusel, E. Wittern, and D. Bermbach. "Benchmarking microservice performance: a pattern-based approach". In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. SAC '20. 2020, pp. 232–241. DOI: 10.1145/3341105.3373875.

[Gor17] I. Gorton. "Hyperscalability – the changing face of software architecture". In: *Software Architecture for Big Data and the Cloud*. Ed. by I. Mistrik, R. Bahsoon, N. Ali, M. Heisel, and B. Maxim. Boston: Morgan Kaufmann, 2017, pp. 13–31. DOI: 10.1016/B978-0-12-805467-3.00002-8.

[Gor22] I. Gorton. *Foundations of scalable systems*. 1st. O'Reilly Media, Inc., 2022.

[GP21] B. E. Granger and F. Pérez. "Jupyter: thinking and storytelling with code and data". In: *Computing in Science & Engineering* 23.2 (2021), pp. 7–14. DOI: 10.1109/MCSE.2021.3059263.

[GPT15] N. J. Gunther, P. Puglia, and K. Tomasette. "Hadoop superlinear scalability". In: *Communications of the ACM* 58.4 (Mar. 2015), pp. 46–55. DOI: 10.1145/2719919.

[Gra21] J. Grabitzky. "A showcase for the Titan Control Center". Bachelor's Thesis. Kiel University, 2021.

[Gra22] Grafana Labs. *Grafana*. 2022. URL: https://grafana.com/grafana.

[Gra93] J. Gray. *The benchmark handbook for database and transaction systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[GTĎ+21] C. Gencer, M. Topolnik, V. Ďurina, E. Demirci, E. B. Kahveci, A. Gürbüz, O. Lukáš, J. Bartók, G. Gierlach, F. Hartman, U. Yılmaz, M. Doğan, M. Mandouh, M. Fragkoulis, and A. Katsifodimos. "Hazelcast Jet: low-latency stream processing at the 99.99th percentile". In: *Proceedings of the VLDB Endowment* 14.12 (July 2021), pp. 3110–3121. DOI: 10.14778/3476311.3476387.

[Gun07] N. J. Gunther. *Guerrilla capacity planning: a tactical approach to planning for highly scalable applications and services*. 1st. Springer-Verlag Berlin Heidelberg, 2007. DOI: 10.1007/978-3-540-31010-5.

[Han19] A. Hansen. "Exploring an energy-status-data set from industrial production". Bachelor's Thesis. Kiel University, 2019.

[Has16] W. Hasselbring. "Microservices for scalability: keynote talk abstract". In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ICPE '16. 2016, pp. 133–134. DOI: 10.1145/2851553.2858659.

[Has18] W. Hasselbring. "Software architecture: past, present, future". In: *The Essence of Software Engineering*. Ed. by V. Gruhn and R. Striemer. Springer, 2018, pp. 169–184. DOI: 10.1007/978-3-319-73897-0_10.

[Has21] W. Hasselbring. "Benchmarking as empirical standard in software engineering research". In: *Evaluation and Assessment in Software Engineering*. EASE '21. 2021, pp. 457–462. DOI: 10.1145/3463274.3463361.

[HB15] T. Hoefler and R. Belli. "Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. 2015. DOI: 10.1145/2807591.2807644.

[HCH+20] W. Hasselbring, L. Carr, S. Hettrick, H. Packer, and T. Tiropanis. "Open source research software". In: *Computer* 53.8 (2020), pp. 84–88. DOI: 10.1109/MC.2020.2998235.

[HCL20] H. Herodotou, Y. Chen, and J. Lu. "A survey on automatic parameter tuning for big data processing systems". In: *ACM Computing Surveys* 53.2 (Apr. 2020). DOI: 10.1145/3381027.

[Hel22] Helm Authors. *Helm*. 2022. URL: https://helm.sh.

[Hen19] S. Henning. "Monitoring electrical power consumption with Kieker". In: *Softwaretechnik-Trends* 39.3 (Nov. 2019). (Proceedings of the 9th Symposium on Software Performance (SSP 2018)), pp. 31–33.

[Hes22] G. Hesse. "A benchmark for enterprise stream processing architectures". PhD thesis. Universität Potsdam, 2022. DOI: 10.25932/publishup-56600.

[HH19a] S. Henning and W. Hasselbring. *Replication package for: scalable and reliable multi-dimensional aggregation of sensor data streams*. Zenodo, 2019. DOI: 10.5281/zenodo.3540895.

[HH19b] S. Henning and W. Hasselbring. "Scalable and reliable multi-dimensional aggregation of sensor data streams". In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019, pp. 3512–3517. DOI: 10.1109/BigData47090.2019.9006452.

[HH20a] S. Henning and W. Hasselbring. *Replication package for: scalable and reliable multi-dimensional sensor data aggregation in data-streaming architectures*. Zenodo, 2020. DOI: 10.5281/zenodo.3736689.

[HH20b] S. Henning and W. Hasselbring. "Scalable and reliable multi-dimensional sensor data aggregation in data-streaming architectures". In: *Data-Enabled Discovery and Applications* 4.1 (2020). DOI: 10.1007/s41688-020-00041-3.

[HH20c] S. Henning and W. Hasselbring. "Toward efficient scalability benchmarking of event-driven microservice architectures at large scale". In: *Softwaretechnik-Trends* 40.3 (Nov. 2020). (Proceedings of the 11th Symposium on Software Performance (SSP 2020)), pp. 28–30.

[HH21a] S. Henning and W. Hasselbring. "How to measure scalability of distributed stream processing engines?" In: *Companion of the ACM/SPEC International Conference on Performance Engineering*. ICPE '21. 2021, pp. 85–88. DOI: 10.1145/3447545.3451190.

[HH21b] S. Henning and W. Hasselbring. *Replication package for: a configurable method for benchmarking scalability of cloud-native applications*. Zenodo, 2021. DOI: 10.5281/zenodo.5596982.

[HH21c] S. Henning and W. Hasselbring. *Replication package for: Theodolite: scalability benchmarking of distributed stream processing engines in microservice architectures*. Zenodo, 2021. DOI: 10.5281/zenodo.4476083.

[HH21d] S. Henning and W. Hasselbring. "The Titan Control Center for Industrial DevOps analytics research". In: *Software Impacts* 7 (2021). DOI: 10.1016/j.simpa.2020.100050.

[HH21e] S. Henning and W. Hasselbring. "Theodolite: scalability benchmarking of distributed stream processing engines in microservice architectures". In: *Big Data Research* 25 (2021), p. 100209. DOI: 10.1016/j.bdr.2021.100209.

[HH22a] S. Henning and W. Hasselbring. "A configurable method for benchmarking scalability of cloud-native applications". In: *Empirical Software Engineering* 27.6 (2022). DOI: 10.1007/s10664-022-10162-1.

[HH22b] S. Henning and W. Hasselbring. "Demo paper: benchmarking scalability of cloud-native applications with Theodolite". In: *2022 IEEE International Conference on Cloud Engineering (IC2E)*. 2022, pp. 275–276. DOI: 10.1109/IC2E55432.2022.00037.

[HH22c] S. Henning and W. Hasselbring. *Replication package for: benchmarking scalability of stream processing frameworks deployed as event-driven microservices in the cloud*. Zenodo, 2022. DOI: 10.5281/zenodo.7497281.

[HH23a] S. Henning and W. Hasselbring. "Benchmarking scalability of cloud-native applications". In: *Software Engineering 2023*. 2023.

[HH23b] S. Henning and W. Hasselbring. *Benchmarking scalability of stream processing frameworks deployed as event-driven microservices in the cloud*. 2023. DOI: 10.48550/arXiv.2303.11088.

[HHB+21] S. Henning, W. Hasselbring, H. Burmester, A. Möbius, and M. Wojcieszak. "Goals and measures for analyzing power consumption data in manufacturing enterprises". In: *Journal of Data, Information and Management* 3.1 (2021), pp. 65–82. DOI: 10.1007/s42488-021-00043-5.

[HHL+19] W. Hasselbring, S. Henning, B. Latte, A. Möbius, T. Richter, S. Schalk, and M. Wojcieszak. "Industrial DevOps". In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 2019, pp. 123–126. DOI: 10.1109/ICSA-C.2019.00029.

[HHL+21] W. Hasselbring, S. Henning, B. Latte, I. Stemmler, M. Wojcieszak, and U. Glockmann. *Abschlussbericht KMU-innovativ: Verbundprojekt Titan Industrial DevOps Plattform für iterative Prozessintegration und Automatisierung*. Tech. rep. Kiel: Selbstverlag des Instituts für Informatik, Kiel, 2021.

[HHM+18] J. Herman, H. Herman, M. J. Mathews, and J. C. Vosloo. "Using big data for insights into sustainable energy consumption in industrial and mining sectors". In: *Journal of Cleaner Production* 197 (2018), pp. 1352–1364. DOI: 10.1016/j.jclepro.2018.06.290.

[HHM19] S. Henning, W. Hasselbring, and A. Möbius. "A scalable architecture for power consumption monitoring in industrial production environments". In: *2019 IEEE International Conference on Fog Computing (ICFC)*. 2019, pp. 124–133. DOI: 10.1109/ICFC.2019.00024.

[Hil90] M. D. Hill. "What is scalability?" In: *ACM SIGARCH Computer Architecture News* 18.4 (Dec. 1990), pp. 18–21. DOI: 10.1145/121973.121975.

[HKR13] N. R. Herbst, S. Kounev, and R. Reussner. "Elasticity in cloud computing: what it is, and what it is not". In: *International Conference on Autonomic Computing*. ICAC '13. 2013, pp. 23–27.

[HKW+15] N. R. Herbst, S. Kounev, A. Weber, and H. Groenda. "BUNGEE: an elasticity benchmark for self-adaptive IaaS cloud environments". In: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 2015, pp. 46–56. DOI: 10.1109/SEAMS.2015.23.

[HKZ20] W. Hasselbring, A. Krause, and C. Zirkelbach. "ExplorViz: research on software visualization, comprehension and collaboration". In: *Software Impacts* 6 (2020), p. 100034. DOI: 10.1016/j.simpa.2020.100034.

[HL15] G. Hesse and M. Lorenz. "Conceptual survey on data stream processing systems". In: *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. 2015, pp. 797–802. DOI: 10.1109/ICPADS.2015.106.

[HLL+21] S. He, T. Liu, P. Lama, J. Lee, I. K. Kim, and W. Wang. "Performance testing for cloud computing with dependent data bootstrapping". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021, pp. 666–678. DOI: 10.1109/ASE51524.2021.9678687.

[HMG+19] G. Hesse, C. Matthies, K. Glass, J. Huegle, and M. Uflacker. "Quantitative impact evaluation of an abstraction layer for data stream processing systems". In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 2019, pp. 1381–1392. DOI: 10.1109/ICDCS.2019.00138.

[HMP+21] G. Hesse, C. Matthies, M. Perscheid, M. Uflacker, and H. Plattner. "ESPBench: the enterprise stream processing benchmark". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '21. 2021, pp. 201–212. DOI: 10.1145/3427921.3450242.

[HMS+19] S. He, G. Manns, J. Saunders, W. Wang, L. Pollock, and M. L. Soffa. "A statistics-based performance testing methodology for cloud applications". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. 2019, pp. 188–199. DOI: 10.1145/3338906.3338912.

[HMZ12] M. Harman, S. A. Mansouri, and Y. Zhang. "Search-based software engineering: trends, techniques and applications". In: *ACM Computing Surveys* 45.1 (Dec. 2012). DOI: 10.1145/2379776.2379787.

[HRM⁺18] G. Hesse, B. Reissaus, C. Matthies, M. Lorenz, M. Kraus, and M. Uflacker. "Senska – towards an enterprise streaming benchmark". In: *Performance Evaluation and Benchmarking for the Analytics Era*. Ed. by R. Nambiar and M. Poess. 2018, pp. 25–40. DOI: 10.1007/978-3-319-72401-0_3.

[HS17] W. Hasselbring and G. Steinacker. "Microservice architectures for scalability, agility and reliability in e-commerce". In: *Proceedings of the IEEE International Conference on Software Architecture Workshops*. 2017, pp. 243–246. DOI: 10.1109/ICSAW.2017.11.

[HS19] M. Hausenblas and S. Schimanski. *Programming Kubernetes: developing cloud-native applications*. O'Reilly Media, Inc., 2019.

[HT09] C. Herrmann and S. Thiede. "Process chain simulation to foster energy efficiency in manufacturing". In: *CIRP Journal of Manufacturing Science and Technology* 1.4 (2009), pp. 221–229. DOI: 10.1016/j.cirpj.2009.06.005.

[Hup09] K. Huppler. "The art of building a good benchmark". In: *Performance Evaluation and Benchmarking*. Ed. by R. Nambiar and M. Poess. 2009, pp. 18–30. DOI: 10.1007/978-3-642-10424-4_3.

[HvH20] W. Hasselbring and A. van Hoorn. "Kieker: a monitoring framework for software engineering research". In: *Software Impacts* 5 (2020), p. 100019. DOI: 10.1016/j.simpa.2020.100019.

[HvHK⁺17] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger. "Performance engineering for microservices: research challenges and directions". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ICPE '17 Companion. 2017, pp. 223–226. DOI: 10.1145/3053600.3053653.

[HWD21] W. Hasselbring, M. Wojcieszak, and S. Dustdar. "Control flow versus data flow in distributed systems integration: revival of flow-based programming for the Industrial Internet of Things". In: *IEEE Internet Computing* 25.4 (2021), pp. 5–12. DOI: 10.1109/MIC.2021.3053712.

[HWH21] S. Henning, B. Wetzel, and W. Hasselbring. "Reproducible benchmarking of cloud-native applications with the Kubernetes Operator Pattern". In: *Symposium on Software Performance 2021*. 2021. URL: http://ceur-ws.org/Vol-3043.

[HWH23] S. Henning, B. Wetzel, and W. Hasselbring. "Cloud-native scalability benchmarking with Theodolite: applied to the TeaStore benchmark". In: *Softwaretechnik-Trends* 43.1 (Feb. 2023). (Proceedings of the 13th Symposium on Software Performance (SSP 2022)), pp. 23–25.

[IH19] B. Ibryam and R. Huss. *Kubernetes patterns: reusable elements for designing cloud native applications*. 1st. O'Reilly Media, Inc., 2019.

[ILF+12] S. Islam, K. Lee, A. Fekete, and A. Liu. "How a consumer can measure elasticity for cloud platforms". In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ICPE '12. 2012, pp. 85–96. DOI: 10.1145/2188286.2188301.

[Int18] International Organization for Standardization. *Energy management systems – Requirements with guidance for use*. Standard. Geneva, CH: International Organization for Standardization, Aug. 2018.

[Int19] International Energy Agency. *World energy balances 2019*. 2019, p. 793. DOI: 10.1787/3a876031-en.

[IPE14] A. Iosup, R. Prodan, and D. Epema. "IaaS cloud benchmarking: approaches, challenges, and experience". In: *Cloud Computing for Data-Intensive Applications*. Ed. by X. Li and J. Qiu. New York, NY: Springer New York, 2014, pp. 83–104. DOI: 10.1007/978-1-4939-1905-5_4.

[IPV17] S. Imai, S. Patterson, and C. A. Varela. "Maximum sustainable throughput prediction for data stream processing over public clouds". In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2017, pp. 504–513. DOI: 10.1109/CCGRID.2017.105.

[IYE11] A. Iosup, N. Yigitbasi, and D. Epema. "On the performance variability of production cloud services". In: *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2011, pp. 104–113. DOI: 10.1109/CCGrid.2011.22.

[JF16] B. Jovanović and J. Filipović. "ISO 50001 standard-based energy management maturity model – proposal and validation in industry". In: *Journal of Cleaner Production* 112 (2016), pp. 2744–2755. DOI: 10.1016/j.jclepro.2015.10.023.

[JFD+16] A. Johanson, S. Flögel, C. Dullo, and W. Hasselbring. "OceanTEA: exploring ocean-derived climate data using microservices". In: *Proceedings of the Sixth International Workshop on Climate Informatics*. NCAR Technical Note NCAR/TN. Sept. 2016, pp. 25–28.

[JKP21] A. R. Jadhav, S. Kiran M. P. R., and R. Pachamuthu. "Development of a novel IoT-enabled power-monitoring architecture with real-time data visualization for use in domestic and industrial scenarios". In: *IEEE Transactions on Instrumentation and Measurement* 70 (2021), pp. 1–14. DOI: 10.1109/TIM.2020.3028437.

[JW00] P. Jogalekar and M. Woodside. "Evaluating the scalability of distributed systems". In: *IEEE Transactions on Parallel and Distributed Systems* 11.6 (2000), pp. 589–603. DOI: 10.1109/71.862209.

[KBF+15] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. "Twitter Heron: stream processing at scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. 2015, pp. 239–250. DOI: 10.1145/2723372.2742788.

[KBS19] M. Kleppmann, A. R. Beresford, and B. Svingen. "Online event processing". In: *Communications of the ACM* 62.5 (Apr. 2019), pp. 43–49. DOI: 10.1145/3312527.

[KÇC$^+$21] I. Karabey Aksakalli, T. Çelik, A. B. Can, and B. Tekinerdoğan. "Deployment and communication patterns in microservice architectures: a systematic literature review". In: *Journal of Systems and Software* 180 (2021), p. 111014. DOI: 10.1016/j.jss.2021.111014.

[KF19] A. Katsifodimos and M. Fragkoulis. "Operational stream processing: towards scalable and consistent event-driven applications". In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology*. 2019, pp. 682–685. DOI: 10.5441/002/edbt.2019.86.

[KH19] H. Knoche and W. Hasselbring. "Drivers and barriers for microservice adoption – a survey among professionals in Germany". In: *Enterprise Modelling and Information Systems Architectures (EMISAJ) – International Journal of Conceptual Modeling* 14.1 (2019), pp. 1–35. DOI: 10.18417/emisa.14.1.

[KH22] A. Krause-Glau and W. Hasselbring. "Scalable collaborative software visualization as a service: short industry and experience paper". In: *2022 IEEE International Conference on Cloud Engineering (IC2E)*. 2022, pp. 182–187. DOI: 10.1109/IC2E55432.2022.00026.

[KJJ$^+$20] F. M. Kanchiralla, N. Jalo, S. Johnsson, P. Thollander, and M. Andersson. "Energy end-use categorization and performance indicators for energy management in the engineering industry". In: *Energies* 13.2 (Jan. 2020), p. 369. DOI: 10.3390/en13020369.

[KK15] M. Kleppmann and J. Kreps. "Kafka, Samza and the Unix philosophy of distributed data". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[KKL10] D. Kossmann, T. Kraska, and S. Loesing. "An evaluation of alternative architectures for transaction processing in the cloud". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. 2010, pp. 579–590. DOI: 10.1145/1807167.1807231.

[KKR14] J. Kuhlenkamp, M. Klems, and O. Röss. "Benchmarking scalability and elasticity of distributed database systems". In: *Proceedings of the VLDB Endowment* 7.12 (Aug. 2014), pp. 1219–1230. DOI: 10.14778/2732977.2732995.

[Kle17] M. Kleppmann. *Designing data-intensive applications*. 1st. O'Reilly Media, Inc., 2017.

[KLvK20] S. Kounev, K.-D. Lange, and J. von Kistowski. *Systems benchmarking: for scientists and engineers*. 1st. Springer Publishing Company, Incorporated, 2020.

[KNR11] J. Kreps, N. Narkhede, and J. Rao. "Kafka: a distributed messaging system for log processing". In: *Proceedings of the International Workshop on Networking Meets Databases*. 2011.

[Koc20] T. Koch. "Scalable and interactive real-time visualization of time series data". Bachelor's Thesis. Kiel University, 2020.

[KQ17] N. Kratzke and P.-C. Quint. "Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study". In: *Journal of Systems and Software* 126 (2017), pp. 1–16. DOI: 10.1016/j.jss.2017.01.001.

[Kra22] N. Kratzke. "Cloud-native observability: the many-faceted benefits of structured and unified logging—a multi-case study". In: *Future Internet* 14.10 (2022). DOI: 10.3390/fi14100274.

[Kre14] J. Kreps. *Questioning the Lambda architecture*. 2014. URL: https://www.oreilly.com/radar/questioning-the-lambda-architecture.

[KRK$^+$18] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. "Benchmarking distributed stream data processing systems". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. Apr. 2018, pp. 1507–1518. DOI: 10.1109/ICDE.2018.00169.

[KRP+16] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and Jupyter development team. "Jupyter Notebooks – a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. 2016, pp. 87–90. URL: https://eprints.soton.ac.uk/403913/.

[KYA17] Z. Karakaya, A. Yazici, and M. Alayyoub. "A comparison of stream processing frameworks". In: *2017 International Conference on Computer and Applications (ICCA)*. Sept. 2017, pp. 1–12. DOI: 10.1109/COMAPP.2017.8079733.

[LC16] P. Leitner and J. Cito. "Patterns in the chaos—a study of performance variation and predictability in public IaaS clouds". In: *ACM Transactions on Internet Technology* 16.3 (Apr. 2016). DOI: 10.1145/2885497.

[LCL+21] L. Liao, J. Chen, H. Li, Y. Zeng, W. Shang, C. Sporea, A. Toma, and S. Sajedi. "Locating performance regression root causes in the field operations of web-based systems: an experience report". In: *IEEE Transactions on Software Engineering* (2021), pp. 1–1. DOI: 10.1109/TSE.2021.3131529.

[LEB15] S. Lehrig, H. Eikerling, and S. Becker. "Scalability, elasticity, and efficiency in cloud computing: a systematic literature review of definitions and metrics". In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*. QoSA '15. 2015, pp. 83–92. DOI: 10.1145/2737182.2737185.

[LF14] J. Lewis and M. Fowler. *Microservices*. 2014. URL: https://martinfowler.com/articles/microservices.html.

[LHW19] B. Latte, S. Henning, and M. Wojcieszak. "Clean code: on the use of practices and tools to produce maintainable code for long-living software". In: *Proceedings of the Workshops of the Software Engineering Conference 2019*. Vol. Vol-2308. Feb. 2019, pp. 96–99. URL: http://ceur-ws.org/Vol-2308.

[Lil00] D. J. Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge University Press, 2000. DOI: 10.1017/CBO9780511612398.

[Lin17] J. Lin. "The Lambda and the Kappa". In: *IEEE Internet Computing* 21.5 (2017). DOI: 10.1109/MIC.2017.3481351.

[LM10] A. Lakshman and P. Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (Apr. 2010), pp. 35–40. DOI: 10.1145/1773912.1773922.

[LMS+18] D. Li, H. Mei, Y. Shen, S. Su, W. Zhang, J. Wang, M. Zu, and W. Chen. "ECharts: a declarative framework for rapid construction of web-based visualization". In: *Visual Informatics* 2.2 (2018), pp. 136–146. DOI: 10.1016/j.visinf.2018.04.011.

[LN18] X. Liu and P. S. Nielsen. "Scalable prediction-based online anomaly detection for smart meter data". In: *Information Systems* 77 (2018), pp. 34–47. DOI: 10.1016/j.is.2018.05.007.

[LSB+18] S. Lehrig, R. Sanders, G. Brataas, M. Cecowski, S. Ivanšek, and J. Polutnik. "CloudStore – towards scalability, elasticity, and efficiency benchmarking and analysis in cloud computing". In: *Future Generation Computer Systems* 78 (2018), pp. 115–126. DOI: 10.1016/j.future.2017.04.018.

[LSL19] C. Laaber, J. Scheuner, and P. Leitner. "Software microbenchmarking in the cloud. how bad is it really?" In: *Empirical Software Engineering* 24.4 (Aug. 2019), pp. 2469–2508. DOI: 10.1007/s10664-019-09681-1.

[LTW+15] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. "SparkBench: a comprehensive benchmarking suite for in memory data analytic platform spark". In: *Proceedings of the 12th ACM International Conference on Computing Frontiers*. CF '15. 2015. DOI: 10.1145/2742854.2747283.

[LWS+19] P. Leitner, E. Wittern, J. Spillner, and W. Hummer. "A mixed-method empirical study of Function-as-a-Service software development in industrial practice". In: *Journal of Systems and Software* 149 (2019), pp. 340–359. DOI: 10.1016/j.jss.2018.12.013.

[LWX+14] R. Lu, G. Wu, B. Xie, and J. Hu. "Stream Bench: towards benchmarking modern distributed stream computing frameworks". In: *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. 2014, pp. 69–78. DOI: 10.1109/UCC.2014.15.

[LZJ+21] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shan, J. Shen, and M. Ali Babar. "Understanding and addressing quality attributes of microservices architecture: a systematic literature review". In: *Information and Software Technology* 131 (2021), p. 106449. DOI: 10.1016/j.infsof.2020.106449.

[LZS+21] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski. "Data management in microservices: state of the practice, challenges, and research directions". In: *Proceedings of the VLDB Endowment* 14.13 (Sept. 2021), pp. 3348–3361. DOI: 10.14778/3484224.3484232.

[MAL18] N. Mohamed, J. Al-Jaroodi, and S. Lazarova-Molnar. "Energy cloud: services for smart buildings". In: *Sustainable Cloud and Energy Services: Principles and Practice*. Ed. by W. Rivera. Springer International Publishing, 2018, pp. 117–134. DOI: 10.1007/978-3-319-62238-5_5.

[MAL19] N. Mohamed, J. Al-Jaroodi, and S. Lazarova-Molnar. "Leveraging the capabilities of Industry 4.0 for improving energy efficiency in smart factories". In: *IEEE Access* 7 (2019), pp. 18008–18020. DOI: 10.1109/ACCESS.2019.2897045.

[MBL+17] M. Masoodian, I. Buchwald, S. Luz, and E. André. "Temporal visualization of energy consumption loads using timetone". In: *2017 21st International Conference Information Visualisation (IV)*. July 2017, pp. 146–151. DOI: 10.1109/iV.2017.13.

[MBM+21] N. C. Mendonça, C. Box, C. Manolache, and L. Ryan. "The monolith strikes back: why istio migrated from microservices to a monolithic architecture". In: *IEEE Software* 38.5 (2021), pp. 17–22. DOI: 10.1109/MS.2021.3080335.

[MCF+22] A. Margara, G. Cugola, N. Felicioni, and S. Cilloni. *A model and survey of distributed data-intensive systems*. 2022. DOI: 10.48550/arxiv.2203.10836.

[MDJ+18] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci. "Taming performance variability". In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. 2018, pp. 409–425.

[Mer22] L. A. Mertens. "Reengineering Theodolite with the Java Operator SDK". Bachelor's Thesis. Kiel University, 2022.

[MG11] P. M. Mell and T. Grance. *Sp 800-145. the NIST definition of cloud computing*. Tech. rep. Gaithersburg, MD, USA, 2011. DOI: 10.6028/NIST.SP.800-145.

[MLB+15] M. Masoodian, B. Lugrin†, R. Bühling, and E. André. "Visualization support for comparing energy consumption data". In: *2015 19th International Conference on Information Visualisation*. July 2015, pp. 28–34. DOI: 10.1109/iV.2015.17.

[MMS+07] M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski. "Scale-up x scale-out: a case study using Nutch/Lucene". In: *2007 IEEE International Parallel and Distributed Processing Symposium*. 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370631.

[Mon22] MongoDB, Inc. *MongoDB*. 2022. URL: https://www.mongodb.com.

[Mor10] J. P. Morrison. *Flow-based programming, 2nd edition: a new approach to application development*. Paramount, CA: CreateSpace, 2010.

[MP19] D. Méndez Fernández and J.-H. Passoth. "Empirical software engineering: from discipline to interdiscipline". In: *Journal of Systems and Software* 148 (2019), pp. 170–179. DOI: 10.1016/j.jss.2018.11.019.

[MS13] G. Miragliotta and F. Shrouf. "Using Internet of Things to improve eco-efficiency in manufacturing: a review on available knowledge and a framework for IoT adoption". In: *Advances in Production Management Systems. Competitive Manufacturing for Innovative Products and Services*. Ed. by C.

Emmanouilidis, M. Taisch, and D. Kiritsis. 2013, pp. 96–102. DOI: 10.1007/978-3-642-40352-1_13.

[MTA+15] F. Martínez-Álvarez, A. Troncoso, G. Asencio-Cortés, and J. Riquelme. "A survey on data mining techniques applied to electricity-related time series forecasting". In: *Energies* 8.11 (2015), pp. 13162–13193. DOI: 10.3390/en81112361.

[MTA+20] A. Merenstein, V. Tarasov, A. Anwar, D. Bhagwat, L. Rupprecht, D. Skourtis, and E. Zadok. "The case for benchmarking control operations in cloud native storage". In: *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. July 2020.

[MTA+21] A. Merenstein, V. Tarasov, A. Anwar, D. Bhagwat, J. Lee, L. Rupprecht, D. Skourtis, Y. Yang, and E. Zadok. "CNSBench: a cloud native storage benchmark". In: *Conference on File and Storage Technologies*. FAST 21. Feb. 2021, pp. 263–276. URL: https://www.usenix.org/conference/fast21/presentation/merenstein.

[MW15] N. Marz and J. Warren. *Big data: principles and best practices of scalable realtime data systems*. 1st. USA: Manning Publications Co., 2015.

[NCM+21a] F. Nikolaidis, A. Chazapis, M. Marazakis, and A. Bilas. "Frisbee: a suite for benchmarking systems recovery". In: *Workshop on High Availability and Observability of Cloud Systems*. HAOC '21. 2021, pp. 18–24. DOI: 10.1145/3447851.3458738.

[NCM+21b] F. Nikolaidis, A. Chazapis, M. Marazakis, and A. Bilas. *Frisbee: automated testing of cloud-native applications in Kubernetes*. Sept. 2021. DOI: 10.48550/arxiv.2109.10727.

[New15] S. Newman. *Building microservices*. 1st. O'Reilly Media, Inc., 2015.

[New21] S. Newman. *Building microservices*. 2nd. O'Reilly Media, Inc., 2021.

[NNG19] H. Nasiri, S. Nasehi, and M. Goudarzi. "Evaluation of distributed stream processing frameworks for IoT applications in smart cities". In: *Journal of Big Data* 6.52 (2019). DOI: 10.1186/s40537-019-0215-2.

[NPP+17] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. “Samza: stateful scalable stream processing at LinkedIn”. In: *Proceedings of the VLDB Endowment* 10.12 (Aug. 2017), pp. 1634–1645. DOI: 10.14778/3137765.3137770.

[Ope22] OpenTSDB Authors. *OpenTSDB*. 2022. URL: http://opentsdb.net.

[Ous11] J. Ousterhout. “Is scale your enemy, or is scale your friend? technical perspective”. In: *Communications of the ACM* 54.7 (July 2011), p. 110. DOI: 10.1145/1965724.1965748.

[PB19] T. Pfandzelter and D. Bermbach. “IoT data processing in the fog: functions, streams, or batch processing?” In: *2019 IEEE International Conference on Fog Computing (ICFC)*. 2019, pp. 201–206. DOI: 10.1109/ICFC.2019.00033.

[PHS+22a] T. Pfandzelter, S. Henning, T. Schirmer, W. Hasselbring, and D. Bermbach. *Replication package for: streaming vs. functions: a cost perspective on cloud event processing*. Zenodo, 2022. DOI: 10.5281/zenodo.7495024.

[PHS+22b] T. Pfandzelter, S. Henning, T. Schirmer, W. Hasselbring, and D. Bermbach. “Streaming vs. functions: a cost perspective on cloud event processing”. In: *2022 IEEE International Conference on Cloud Engineering (IC2E)*. 2022, pp. 67–78. DOI: 10.1109/IC2E55432.2022.00015.

[PHU20] A. Pagliari, F. Huet, and G. Urvoy-Keller. “NAMB: a quick and flexible stream processing application prototype generator”. In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 2020, pp. 61–70. DOI: 10.1109/CCGrid49817.2020.00-87.

[PJ16] C. Pahl and P. Jamshidi. “Microservices: a systematic mapping study”. In: *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1*. CLOSER 2016. 2016, pp. 137–146. DOI: 10.5220/0005785501370146.

[PJZ18] C. Pahl, P. Jamshidi, and O. Zimmermann. "Architectural principles for cloud software". In: *ACM Transactions on Internet Technology* 18.2 (Feb. 2018). DOI: 10.1145/3104028.

[Pro22] Prometheus Authors. *Prometheus*. 2022. URL: https://prometheus.io.

[PVB$^+$21] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. v. Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tůma, and A. Iosup. "Methodological principles for reproducible performance evaluation in cloud computing". In: *IEEE Transactions on Software Engineering* 47.8 (2021), pp. 1528–1543. DOI: 10.1109/TSE.2019.2927908.

[QMM$^+$18] J. C. Quiroz, N. Mariun, M. R. Mehrjou, M. Izadi, N. Misron, and M. A. M. Radzi. "Fault detection of broken rotor bar in LS-PMSM using random forests". In: *Measurement* 116 (2018), pp. 273–280. DOI: 10.1016/j.measurement.2017.11.004.

[QT19] Q. Qi and F. Tao. "A smart manufacturing service system based on edge computing, fog computing, and cloud computing". In: *IEEE Access* 7 (2019), pp. 86769–86777. DOI: 10.1109/ACCESS.2019.2923610.

[QWH$^+$16] S. Qian, G. Wu, J. Huang, and T. Das. "Benchmarking modern distributed streaming platforms". In: *2016 IEEE International Conference on Industrial Technology (ICIT)*. 2016, pp. 592–598. DOI: 10.1109/ICIT.2016.7474816.

[RbAB$^+$21] P. Ralph, N. bin Ali, S. Baltes, D. Bianculli, J. Diaz, Y. Dittrich, N. Ernst, M. Felderer, R. Feldt, A. Filieri, B. B. N. de França, C. A. Furia, G. Gay, N. Gold, D. Graziotin, P. He, R. Hoda, N. Juristo, B. Kitchenham, V. Lenarduzzi, J. Martínez, J. Melegati, D. Mendez, T. Menzies, J. Molleri, D. Pfahl, R. Robbes, D. Russo, N. Saarimäki, F. Sarro, D. Taibi, J. Siegmund, D. Spinellis, M. Staron, K. Stol, M.-A. Storey, D. Taibi, D. Tamburri, M. Torchiano, C. Treude, B. Turhan, X. Wang, and S. Vegas. *Empirical standards for software engineering research*. Version 0.2.0. Mar. 2021. DOI: 10.48550/arXiv.2010.03525.

[RJD+15] T. Rackow, T. Javied, T. Donhauser, C. Martin, P. Schuderer, and J. Franke. “Green Cockpit: transparency on energy consumption in manufacturing companies”. In: *Procedia CIRP* 26 (2015). 12th Global Conference on Sustainable Manufacturing – Emerging Potentials, pp. 498–503. DOI: 10.1016/j.procir.2015.01.011.

[RM19] T. Rist and M. Masoodian. “Promoting sustainable energy consumption behavior through interactive data visualizations”. In: *Multimodal Technologies and Interaction* 3.3 (2019), p. 56. DOI: 10.3390/mti3030056.

[Roh22] T. Rohrmann. “Rethinking how distributed applications are built”. In: *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*. DEBS ’22. 2022, p. 4. DOI: 10.1145/3524860.3544410.

[RSE+19] R. Rabiser, K. Schmid, H. Eichelberger, M. Vierhauser, S. Guinea, and P. Grünbacher. “A domain analysis of resource and requirements monitoring: towards a comprehensive model of the software monitoring domain”. In: *Information and Software Technology* 111 (2019), pp. 86–109. DOI: 10.1016/j.infsof.2019.03.013.

[SAS19] E. Shahverdi, A. Awad, and S. Sakr. “Big stream processing systems: an experimental evaluation”. In: *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. 2019, pp. 53–60. DOI: 10.1109/ICDEW.2019.00-35.

[Sax20] M. J. Sax. “Performance optimizations and operator semantics for streaming data flow programs”. PhD thesis. Humboldt-Universität zu Berlin, 2020. DOI: 10.18452/21424.

[SBA20] R. Sahal, J. G. Breslin, and M. I. Ali. “Big data and stream processing platforms for Industry 4.0 requirements mapping for a predictive maintenance use case”. In: *Journal of Manufacturing Systems* 54 (2020), pp. 138–151. DOI: 10.1016/j.jmsy.2019.11.004.

[SBS+17] T. Szydlo, R. Brzoza-Woch, J. Sendorek, M. Windak, and C. Gniady. "Flow-based programming for IoT leveraging fog computing". In: *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. 2017, pp. 74–79. DOI: 10.1109/WETICE.2017.17.

[SCG+14] H. Sequeira, P. Carreira, T. Goldschmidt, and P. Vorst. "Energy Cloud: real-time cloud-native energy management system to monitor and analyze energy consumption in multiple industrial sites". In: *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. Dec. 2014, pp. 529–534. DOI: 10.1109/UCC.2014.79.

[SCS17] A. Shukla, S. Chaturvedi, and Y. Simmhan. "RIoTBench: an IoT benchmark for distributed stream processing systems". In: *Concurrency and Computation: Practice and Experience* 29.21 (2017), e4257. DOI: 10.1002/cpe.4257.

[SEH03] S. E. Sim, S. Easterbrook, and R. C. Holt. "Using benchmarking to advance research: a challenge to software engineering". In: *25th International Conference on Software Engineering*. 2003, pp. 74–83. DOI: 10.1109/icse.2003.1201189.

[SGO17] F. Shrouf, B. Gong, and J. Ordieres-Meré. "Multi-level awareness of energy used in production processes". In: *Journal of Cleaner Production* 142 (2017), pp. 2570–2585. DOI: 10.1016/j.jclepro.2016.11.019.

[SM15] F. Shrouf and G. Miragliotta. "Energy management based on Internet of Things: practices and framework for adoption in production management". In: *Journal of Cleaner Production* 100 (2015), pp. 235–246. DOI: 10.1016/j.jclepro.2015.03.055.

[SNO+16] M. Schulze, H. Nehler, M. Ottosson, and P. Thollander. "Energy management in industry – a systematic review of previous findings and an integrative conceptual framework". In: *Journal of Cleaner Production* 112 (2016), pp. 3692–3708. DOI: 10.1016/j.jclepro.2015.06.060.

[SOG+14] F. Shrouf, J. Ordieres-Meré, A. García-Sánchez, and M. Ortega-Mier. "Optimizing the production scheduling of a single machine to minimize total energy consumption costs". In: *Journal of Cleaner Production* 67 (2014), pp. 197–207. DOI: 10.1016/j.jclepro.2013.12.024.

[SOM14] F. Shrouf, J. Ordieres, and G. Miragliotta. "Smart factories in Industry 4.0: a review of the concept and of energy management approached in production based on the Internet of Things paradigm". In: *2014 IEEE International Conference on Industrial Engineering and Engineering Management*. Dec. 2014, pp. 697–701. DOI: 10.1109/IEEM.2014.7058728.

[Spæ21] T. Spæren. "Performance analysis and improvements for Apache Beam". Master's Thesis. University of Oslo, 2021.

[Spi89] J. M. Spivey. *The Z notation: a reference manual*. USA: Prentice-Hall, Inc., 1989.

[SSH+18] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, and M. Gidlund. "Industrial Internet of Things: challenges, opportunities, and directions". In: *IEEE Transactions on Industrial Informatics* 14.11 (2018), pp. 4724–4734. DOI: 10.1109/TII.2018.2852491.

[Sto18] B. Stopford. *Designing event-driven systems*. 1st. O'Reilly Media, Inc., 2018.

[STV18] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel. "The pains and gains of microservices: a systematic grey literature review". In: *Journal of Systems and Software* 146 (2018), pp. 215–232. DOI: 10.1016/j.jss.2018.09.082.

[SW02] C. U. Smith and L. G. Williams. *Performance solutions: a practical guide to creating responsive, scalable software*. USA: Addison Wesley Longman Publishing Co., Inc., 2002.

[SWW+18] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag. "Streams and tables: two sides of the same coin". In: *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*. BIRTE '18. 2018, pp. 1–10. DOI: 10.1145/3242153.3242155.

[SY18] S. Singh and A. Yassine. "Big data mining of energy time series for behavioral analytics and energy consumption forecasting". In: *Energies* 11.2 (Feb. 2018), p. 452. DOI: 10.3390/en11020452.

[TBP$^+$22] L. Thamsen, J. Beilharz, A. Polze, and O. Kao. *The methods of cloud computing*. Tech. rep. Technische Universität Berlin, Feb. 2022.

[TdCP$^+$20] F. S. Tesch da Silva, C. A. da Costa, C. D. Paredes Crovato, and R. da Rosa Righi. "Looking at energy through the lens of Industry 4.0: a systematic literature review of concerns and challenges". In: *Computers & Industrial Engineering* 143 (2020), p. 106426. DOI: 10.1016/j.cie.2020.106426.

[TGC$^+$21] J. Traub, P. M. Grulich, A. R. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. "Scotty: general and efficient open-source window aggregation for stream processing systems". In: *ACM Transactions on Database Systems* 46.1 (Mar. 2021). DOI: 10.1145/3433675.

[THS11] W.-T. Tsai, Y. Huang, and Q. Shao. "Testing the scalability of SaaS applications". In: *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. 2011, pp. 1–4. DOI: 10.1109/SOCA.2011.6166245.

[Tic14] W. F. Tichy. "Where's the science in software engineering? ubiquity symposium: the science in computer science". In: *Ubiquity* 2014 (Mar. 2014), pp. 1–6. DOI: 10.1145/2590528.2590529.

[Tic98] W. F. Tichy. "Should computer scientists experiment more?" In: *Computer* 31.5 (1998), pp. 32–40. DOI: 10.1109/2.675631.

[TLP17] D. Taibi, V. Lenarduzzi, and C. Pahl. "Processes, motivations, and issues for migrating to microservices architectures: an empirical investigation". In: *IEEE Cloud Computing* 4.5 (2017), pp. 22–32. DOI: 10.1109/MCC.2017.4250931.

[TPC+15] P. Thollander, S. Paramonova, E. Cornelis, O. Kimura, A. Trianni, M. Karlsson, E. Cagno, I. Morales, and J. P. Jiménez Navarro. "International study on energy end-use data among industrial SMEs (small and medium-sized enterprises) and energy end-use efficiency improvement opportunities". In: *Journal of Cleaner Production* 104 (2015), pp. 282–296. DOI: 10.1016/j.jclepro.2015.04.073.

[Tsa20] C. Tsatia Tsida. "Analyzing environmental data with the Titan platform". Master's Thesis. Kiel University, 2020.

[TTP+10] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. *NEXMark – a benchmark for queries over data streams (draft)*. 2010. URL: https://datalab.cs.pdx.edu/niagara/pstream/nexmark.pdf.

[TTS+14] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. "Storm@twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. 2014, pp. 147–156. DOI: 10.1145/2588555.2595641.

[VD10] A. Vijayaraghavan and D. Dornfeld. "Automated energy monitoring of machine tools". In: *CIRP Annals* 59.1 (2010), pp. 21–24. DOI: 10.1016/j.cirp.2010.03.042.

[vDon21] G. van Dongen. "Open stream processing benchmark: an extensive analysis of distributed stream processing frameworks". PhD thesis. Ghent University, 2021.

[vDvdP20] G. van Dongen and D. van den Poel. "Evaluation of stream processing frameworks". In: *IEEE Transactions on Parallel and Distributed Systems* 31.8 (2020), pp. 1845–1858. DOI: 10.1109/TPDS.2020.2978480.

[vDvdP21a] G. van Dongen and D. van den Poel. "A performance analysis of fault recovery in stream processing frameworks". In: *IEEE Access* 9 (2021), pp. 93745–93763. DOI: 10.1109/ACCESS.2021.3093208.

[vDvdP21b] G. van Dongen and D. van den Poel. "Influencing factors in the scalability of distributed stream processing jobs". In: *IEEE Access* 9 (2021), pp. 109413–109431. DOI: 10.1109/ACCESS.2021.3102645.

[VGB13] K. Vikhorev, R. Greenough, and N. Brown. "An advanced energy management framework to promote energy awareness". In: *Journal of Cleaner Production* 43 (2013), pp. 103–112. DOI: 10.1016/j.jclepro.2012.12.012.

[VGT+23] J. Verwiebe, P. M. Grulich, J. Traub, and V. Markl. "Survey of window types for aggregation in stream processing systems". In: *The VLDB Journal* (Feb. 2023). DOI: 10.1007/s00778-022-00778-6.

[vHoo14] A. van Hoorn. *Model-driven online capacity management for component-based software systems*. Kiel Computer Science Series 2014/6. Kiel, Germany: Department of Computer Science, Kiel University, 2014.

[vHWH12] A. van Hoorn, J. Waller, and W. Hasselbring. "Kieker: a framework for application performance monitoring and dynamic software analysis". In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ICPE '12. 2012, pp. 247–248. DOI: 10.1145/2188286.2188326.

[vKAH+15] J. von Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao. "How to build a benchmark". In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. 2015, pp. 333–336. DOI: 10.1145/2668930.2688819.

[vKES+18] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev. "TeaStore: a micro-service reference application for benchmarking, modeling and resource management research". In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Sept. 2018, pp. 223–236. DOI: 10.1109/MASCOTS.2018.00030.

[Von21a] B. Vonheiden. "Empirical scalability evaluation of window aggregation methods in distributed stream processing". Master's Thesis. Kiel University, 2021.

[Von21b] B. Vonheiden. *Master thesis replication package for: empirical scalability evaluation of hopping window aggregation methods in distributed stream processing*. Zenodo, 2021. DOI: 10.5281/zenodo.5764902.

[VT20] V. E. Venugopal and M. Theobald. "Benchmarking synchronous and asynchronous stream processing systems". In: *Proceedings of the 7th ACM IKDD CoDS and 25th COMAD*. CoDS COMAD 2020. 2020, pp. 322–323. DOI: 10.1145/3371158.3371206.

[Wal14] J. Waller. *Performance benchmarking of application monitoring frameworks*. Kiel Computer Science Series 2014/5. Kiel, Germany: Department of Computer Science, Kiel University, 2014.

[WCD+21] G. Wang, L. Chen, A. Dikshit, J. Gustafson, B. Chen, M. J. Sax, J. Roesler, S. Blee-Goldman, B. Cadonna, A. Mehta, V. Madan, and J. Rao. "Consistency and completeness: rethinking distributed stream processing in Apache Kafka". In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD/PODS '21. 2021, pp. 2602–2613. DOI: 10.1145/3448016.3457556.

[WEH15] J. Waller, N. C. Ehmke, and W. Hasselbring. "Including performance benchmarks into continuous integration to enable DevOps". In: *ACM SIGSOFT Software Engineering Notes* 40.2 (Mar. 2015), pp. 1–4. DOI: 10.1145/2735399.2735416.

[Wet19] D. B. Wetzel. "Entwicklung eines Dashboards für eine Industrial DevOps Monitoring Plattform". Bachelor's Thesis. Kiel University, 2019.

[Wet22] D. B. Wetzel. "Scalability benchmarking of a promotional loan system". Master's Thesis. Kiel University, 2022.

[WHG+14] A. Weber, N. Herbst, H. Groenda, and S. Kounev. “Towards a resource elasticity benchmark for cloud environments”. In: *Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability*. HotTopiCS ’14. 2014. DOI: 10.1145/2649563.2649571.

[WHH03] C. Wohlin, M. Höst, and K. Henningsson. “Empirical research methods in software engineering”. In: *Empirical Methods and Studies in Software Engineering: Experiences from ESERNET*. Ed. by R. Conradi and A. I. Wang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 7–23. DOI: 10.1007/978-3-540-45143-3_2.

[WKS+15] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein. “Building a replicated logging system with Apache Kafka”. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1654–1655. DOI: 10.14778/2824032.2824063.

[WvHK+16] J. Walter, A. van Hoorn, H. Koziolek, D. Okanovic, and S. Kounev. “Asking “what?”, automating the “how?”: the vision of declarative performance engineering”. In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ICPE ’16. 2016, pp. 91–94. DOI: 10.1145/2851553.2858662.

[YCL+20] C.-T. Yang, S.-T. Chen, J.-C. Liu, R.-H. Liu, and C.-L. Chang. “On construction of an energy monitoring service using big data technology for the smart campus”. In: *Cluster Computing* 23 (2020), pp. 265–288. DOI: 10.1007/s10586-019-02921-5.

[YJH+17] S. Yang, Y. Jeong, C. Hong, H. Jun, and B. Burgstaller. “Scalability and state: a critical assessment of throughput obtainable on big data streaming frameworks for applications with and without state information”. In: *Euro-Par 2017: Parallel Processing Workshops*. Vol. 10659. Lecture Notes in Computer Science. 2017, pp. 141–152. DOI: 10.1007/978-3-319-75178-8_12.

[YML+17] J. Yan, Y. Meng, L. Lu, and L. Li. "Industrial big data in an Industry 4.0 environment: challenges, schemes, and applications for predictive maintenance". In: *IEEE Access* 5 (2017), pp. 23484–23491. DOI: 10.1109/ACCESS.2017.2765544.

[You22] E. You. *Vue.js*. 2022. URL: https://vuejs.org.

[ZHD+17] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze. "Revisiting the design of data stream processing systems on multi-core processors". In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 2017, pp. 659–670. DOI: 10.1109/ICDE.2017.119.

[Zim17] O. Zimmermann. "Microservices tenets". In: *Computer Science – Research and Development* 32.3 (July 2017), pp. 301–310. DOI: 10.1007/s00450-016-0337-0.

[ZLC+23] X. Zhou, S. Li, L. Cao, H. Zhang, Z. Jia, C. Zhong, Z. Shan, and M. A. Babar. "Revisiting the practices and pains of microservice architecture in reality: an industrial inquiry". In: *Journal of Systems and Software* 195 (2023), p. 111521. DOI: 10.1016/j.jss.2022.111521.

[ZMK+19] S. Zeuch, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl. "Analyzing efficient stream processing on modern hardware". In: *Proceedings of the VLDB Endowment* 12.5 (Jan. 2019), pp. 516–530. DOI: 10.14778/3303753.3303758.

[ZMY+18] Y. Zhang, S. Ma, H. Yang, J. Lv, and Y. Liu. "A big data driven analytical framework for energy-intensive manufacturing industries". In: *Journal of Cleaner Production* 197 (2018), pp. 57–72. DOI: 10.1016/j.jclepro.2018.06.170.

[ZXW+16] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. "Apache Spark: a unified engine for big data processing". In: *Communications of the ACM* 59.11 (Oct. 2016), pp. 56–65. DOI: 10.1145/2934664.

[ZXW+20] Y. Zhang, B. Xia, K. Wu, and X. Liu. *Building a better and faster Beam Samza runner*. 2020. URL: https://engineering.linkedin.com/blog/2020/building-a-better-and-faster-beam-samza-runner.

[ZXZ+17] J. Zheng, C. Xu, Z. Zhang, and X. L. Li. "Electric load forecasting in smart grids using long-short-term-memory based recurrent neural network". In: *2017 51st Annual Conference on Information Sciences and Systems (CISS)*. Mar. 2017, pp. 1–6. DOI: 10.1109/CISS.2017.7926112.