Empirical Software Engineering (2022) 27: 143
https://doi.org/10.1007/s10664-022-10162-1

A configurable method for benchmarking scalability of cloud-native applications

Sören Henning[1] · **Wilhelm Hasselbring**[1]

Accepted: 30 March 2022 / Published online: 6 August 2022
© The Author(s) 2022

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. In this study, we present a 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, particularly suited for cloud-native applications. 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 asses whether specified service level objectives (SLOs) are achieved. To balance usability and reproducibility, our benchmarking method provides configuration options, controlling the trade-off between overall execution time and statistical grounding. We perform an extensive experimental evaluation of our method's configuration options for the special case of event-driven microservices. For this purpose, we use benchmark implementations of the two stream processing frameworks Kafka Streams and Flink and run our experiments in two public clouds and one private cloud. We find that, independent of the cloud platform, it only takes a few repetitions ($\leq$ 5) and short execution times ($\leq$ 5 minutes) to assess whether SLOs are achieved. Combined with our findings from evaluating different search strategies, we conclude that our method allows to benchmark scalability in reasonable time.

Keywords Scalability · Benchmarking · Performance engineering · Cloud-Native

---

Communicated by: Tse-Hsun (Peter) Chen, Cor-Paul Bezemer, André van Hoorn, Catia Trubiani and Weiyi Shang

This article belongs to the Topical Collection: *Software Performance*

✉ Sören Henning
soeren.henning@email.uni-kiel.de

Wilhelm Hasselbring
hasselbring@email.uni-kiel.de

[1] Software Engineering Group, Kiel University, 24098 Kiel, Germany

1 Introduction

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 (Gannon et al. 2017; Kratzke and Quint 2017). 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, dynamic orchestration, and microservices, which provide a new level of hardware abstraction, while still providing a high flexibility regarding the system's deployment. An entire ecosystem of such tools has grown in recent years under the umbrella of the Cloud Native Computing Foundation,[1] a suborganization within the Linux Foundation. Most prominent among these tools is probably Kubernetes (Burns et al. 2016), which has become the de-facto standard orchestration tool for cloud-native applications. Nowadays, all major cloud providers offer managed Kubernetes clusters. With such offerings, users only specify the desired cluster size (e.g., number of virtual or physical nodes and properties of nodes). For the actual operation of their applications (e.g., scaling, updating, or repairing), they only interact with the Kubernetes API. Recently, a further level of abstraction can be observed. For example, the Google Cloud Platform launched its *Kubernetes Autopilot* in 2021, which adjusts underlying node pools based on the current demand and only charges users for the containers they actually deploy.

A definition of the term "cloud-native" is provided by the Cloud Native Computing Foundation, which states that "cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds" (Cloud Native Computing Foundation 2018). This definition already includes the requirement for scalability. Similarly, microservice architectures (a common pattern for cloud-native applications (Balalaie et al. 2016)) are often adopted to cope with scalability requirements (Kratzke and Quint 2017; Soldani et al. 2018; Knoche and Hasselbring 2019). However, although scalability is often named as a crucial motivation for adopting cloud-native architectures or deployments, research is lacking a commonly accepted method to empirically assess and compare the scalability of cloud-native applications.

In empirical software engineering research, benchmarks are an established research method to compare different methods, techniques, and tools based on a standardized method (Sim et al. 2003; Tichy 2014; Hasselbring 2021). For traditional performance attributes such as latency or throughput, well-known (and often straightforward) metrics and measurement methods exists (Kounev et al. 2020). For scalability, in particular in the context of cloud-native applications, the situation is different: On the one hand, precise definitions exist and were refined over the last two decades (Jogalekar and Woodside 2000; Duboc et al. 2007; Weber et al. 2014). On the other hand, we observe that for cloud applications, no commonly accepted scalability benchmarking method exists, employed benchmarking methods are insufficiently described or not aligned with scalability definitions, and benchmarking methods for other application types cannot be transferred. 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.

With this paper, we aim to bridge the gap between research on defining scalability and experimental scalability evaluations of cloud-native applications. Our goal is to provide

[1] https://www.cncf.io

a solid framework, allowing researchers and practitioners to conduct empirical scalability evaluations. We propose, discuss, and evaluate a benchmarking method that can be used by benchmark designers (e.g., standardization organizations or research communities) and benchmarkers (e.g., software engineers, cloud vendors, or researchers), who wish to benchmark the scalability of different software artifacts or deployment options. Our study consists of two parts: a pre-study for engineering the benchmarking method and an experimental evaluation.

Pre-study We review scalability definitions and benchmarking best practices from industrial consortia and academia. We analyze which quality attributes of benchmarks have to be fulfilled by which benchmarking components and derive requirements for fulfilling them for the special case of scalability benchmarking. Based on the distinction between metric, measurement method, and tool architecture, we engineer a benchmarking method implementing the particular requirements.

We require that a scalability metric for cloud-native applications should be aligned with accepted definitions of scalability in cloud computing. We find that this can be achieved by two scalability metrics that describe scalability as functions. Our *demand* metric indicates how the required amount of provisioned resources evolves with increasing load intensities. Our *capacity* metric indicates how the processible load intensity evolves with increasing amounts of provisioned resources. Both metrics use the notion of service level objectives (SLOs) for quantifying whether a certain resource amount can handle a certain load intensity.

A scalability measurement method should provide statistically grounded results to allow for reproducibility. On the other hand, it should also not be too time consuming to remain usable. While measuring scalability according to our proposed metrics provides an accurate quantification of a system's scalability, it also requires many experiments to be performed. Combined with the need for sufficiently long experiment durations or repetitions, this may lead to long runtimes for benchmarking scalability. Our proposed measurement method provides several configuration options to balance the overall runtime and statistically grounding of its results.

For a scalability benchmarking tool, we require in particular usability to support reproducibility and verifiability of benchmarking studies. This includes a simple installation and the declarative description of benchmarks and their executions. Due to the complexity of such a benchmarking tool, the tool and corresponding benchmarks should not be coupled. We propose a benchmarking architecture fulfilling these requirements by adopting common patterns for managing cloud-native applications. Additionally, this architecture contains a data model for separately describing benchmarks and their executions, taking the different types of actors involved in benchmarking into account.

Experimental evaluation We experimentally evaluate whether our proposed method allows one to obtain statistically grounded results within reasonable execution times. We use an implementation of our proposed benchmarking tool architecture to experimentally evaluate the individual configuration parameters, which control the statistically grounding of benchmark results as well as the corresponding execution times. Test subjects of our experiments are two stream processing engines, which implement a set of four benchmarks. Our experiments are conducted in Kubernetes clusters running at two public cloud vendors (Google Cloud Platform and Oracle Cloud Infrastructure) and one private cloud. Specifically, we address the following research questions:

RQ 1 *For how long should experiments be executed that evaluate whether a certain combination of load intensity and provisioned resources fulfill the specified SLOs?*
RQ 2 *How many repetitions of such experiments should be performed?*
RQ 3 *How does the assessment of SLOs evolve with increasing resource amounts?*
RQ 4 *How does the assessment of SLOs evolve with increasing load intensities?*

We find that in most cases, it only takes a few repetitions ($\leq$ 5) and short execution times ($\leq$ 5 minutes) to assess whether a certain resource amount can handle a certain load intensity. When accepting a small error in the derived resource demand, also resource-load combinations for which this assessment is more extensive can be run with a few repetitions and short execution times. Moreover, the space of load intensities and resource amounts experiments are executed for can massively be reduced with search strategies for both our presented metrics.

Recommendation As benchmarking activities are often restricted in time or resources, we recommend to focus on expanding the evaluated load and resource space instead of exhaustively evaluate individual load and resource combinations. While individual experiments should be repeated and warm-up periods should be considered, evaluating more load intensities or resource amounts allows to gain a better impression of a system's scalability.

Contributions In summary, the main contributions of this paper are as follows:

- Requirements on scalability benchmarking of cloud-native applications, arranged by scalability metric, measurement method, and benchmarking tool architecture.
- A scalability measurement method, which performs isolated experiments for different load intensities and resource amounts in a configurable manner and evaluates whether specified service level objectives (SLOs) are met.
- An architecture for a cloud-native scalability benchmarking tool implementing the proposed measurement method. It supports different benchmarks, which do not need to be explicitly designed for scalability benchmarking.
- An extensive evaluation of different configuration options of our proposed method. A replication package and the collected data of our experiments is published as supplemental material (Henning and Hasselbring 2021b), such that other researchers may repeat and extend our work.

The remainder of this paper starts by providing the background and context of this study in Section 2. Based on this overview, we derive requirements for a scalability metric, a measurement method, and a benchmarking tool in Section 3. Afterwards, we propose scalability metrics in Section 4, a corresponding scalability measurement method in Section 5, and our scalability benchmarking architecture in Section 6. We experimentally evaluate our proposed benchmarking method in Section 7 and discuss related work in Section 8. Section 9 concludes this paper and points out future research directions.

2 Context and Background

In this section, we provide the context and background of this paper. Benchmarks consist of multiple components to empirically evaluate a quality, namely scalability in this work. Figure 1 shows these components and highlights the scope of this paper. The goal of this paper is not to present new benchmarks, i.e., SUTs and load generator, as those do already

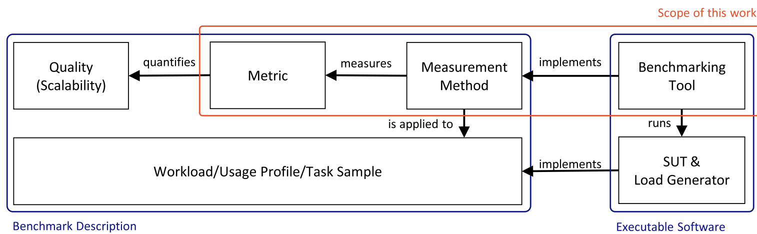


Fig. 1 Components of a benchmark and scope of this work

exists. Instead, we present a scalability benchmarking method, which consists of metrics, measurement method, and an architecture for executing such benchmarks. In the following, we briefly describe the components of benchmarks along with respective quality attributes in Section 2.1 and provide an overview of scalability definitions in Section 2.2.

2.1 Benchmarking in Empirical Software Engineering

2.1.1 Components of Benchmarks

The recently published ACM SIGSOFT Empirical Standard for Benchmarking (Ralph et al. 2021; Hasselbring 2021) names four essential components of a benchmark:[2]

- the quality to be benchmarked (e.g., performance, availability, scalability, security)
- the metric(s) to quantify the quality
- the measurement method(s) for the metric (if not obvious)
- the workload, usage profile and/or task sample the system under test is subject to (i.e., what the system is doing when the measures are taken)

In addition, benchmarks usually come with a benchmarking tool to automate the benchmarking process. A typical benchmarking tool architecture contains separate components for the load generation and SUT (Bermbach et al. 2017). 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 (Bermbach et al. 2017). Sometimes also a visualization component for passive observation is included (Bermbach et al. 2017).

2.1.2 Quality Attributes of Benchmarks

A set of five desired quality attributes for benchmarks is presented by v Kistowski et al. (2015), which represents the perspectives of the SPEC and TPC committees:

- **Relevance** How closely the benchmark behavior correlates to behaviors that are of interest to consumers of the results
- **Reproducibility** The ability to consistently produce similar results when the benchmark is run with the same test configuration

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

- **Fairness** Allowing different test configurations to compete on their merits with-out artificial limitations
- **Verifiability** Providing confidence that a benchmark result is accurate
- **Usability** Avoiding roadblocks for users to run the benchmark in their test environments

These and similar quality attributes can also be found by Sim et al. (2003), Huppler (2009), and Folkerts et al. (2013) as well as in textbooks on (cloud) benchmarking (Bermbach et al. 2017; Kounev et al. 2020).

2.1.3 Application-driven Benchmark Design

Often a distinction between microbenchmarking and application benchmarking is made (Kounev et al. 2020). We focus on benchmarking entire cloud-native applications or independently deployable components such as microservices as those can only be scaled at a whole. This implies that from a benchmarker's perspective, the SUT is a black box and we have to rely on metrics exposed by the orchestration platforms, middlewares, or the SUT itself.

2.2 Definition of Scalability

Initial definitions for scalability of distributed systems were presented by Bondi (2000) and Jogalekar and Woodside (2000), which were later generalized by Duboc et al. (2007).

2.2.1 Scalability in Cloud Computing

More recently, such definitions have been specified to target the peculiarities of scalability in cloud computing (Herbst et al. 2013; Lehrig et al. 2015; Brataas et al. 2017). A definition of scalability in cloud computing is, for example, given by Herbst et al. (2013), 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. (2014) further 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 amount 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 (Kleppmann 2017; Gorton 2022).

A similar definition, although formulated inversely, is used in multiple publications of the "CloudScale" project (Lehrig et al. 2015; Brataas et al. 2017; Brataas et al. 2021). They define scalability as "a system's ability to increase its capacity by consuming more resources" (Brataas et al. 2021), 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. (2014) and others call SLOs.

2.2.2 Vertical and Horizontal Scalability

A distinction is often made between horizontal and vertical scalability (Michael et al. 2007; Lehrig et al. 2015). While horizontal scaling refers to adding computing nodes to cope with increasing load intensities, vertical scaling means adding resources to a single node. A special case of vertical scaling in cloud computing is migrating from one VM type to another (Weber et al. 2014). 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. The scalability definition presented previously covers both horizontal and vertical scalability as both refer to different types of provisioned resources (Weber et al. 2014).

2.2.3 Scalability vs. Elasticity

Another quality that is often used in cloud computing is elasticity (Lehrig et al. 2015). Scalability and elasticity are related, but elasticity takes temporal aspects into account and describes how fast and how precisely a system adapts its provided resources to changing load intensities (Herbst et al. 2013; Islam et al. 2012). Scalability, on the other hand, is a prerequisite for elasticity, but is a time-free notion describing whether increasing load intensities can be handled eventually.

3 Requirements on Scalability Benchmarking

To identify requirements for scalability benchmarking method, we build upon the established benchmark quality attributes listed in Section 2.1.2. For each quality attribute, we identify the benchmark components (cf. Section 2.1.1) that can contribute most to implementing the attribute. Based on this, we derive for each benchmark component covered by this work (metric, measurement method, and tool architecture) a set of requirements that needs to be fulfilled for benchmarking scalability.

Table 1 describes the desired behavior of the individual benchmark components in order to implement the respective quality attribute. We conclude that relevance and fairness have mainly to be implemented at the level of the task sample and, to some extent, by the metric. The measurement method and the tool architecture, on the other hand, should primarily be designed for reproducibility, verifiability, and usability. In the following, we propose a set of requirements for scalability benchmarking for each of the benchmark components metric, measurement method, and tool architecture.

3.1 Requirements for Scalability Metrics

As described in Section 2, scalability is defined by the three attributes load intensity, service levels objectives (SLOs), and provisioned resources. We aim for a general scalability metric, which is applicable to different types of systems and scalability evaluations. Therefore, we require that this metric is not restricted to a specific type of any of these attributes.

Table 1 Quality attributes of benchmarks and benchmarking components required to implement them

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

Support for different load types Various types of load for a cloud-native application exist. 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 amount of messages written to a dedicated messaging system. Such load types can be further broken down to distinguish, for example, between the amount of concurrent users sending requests and the frequency users send requests with. 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. In previous work, we also highlighted domain-specific load types for the case of big data stream processing (Henning and Hasselbring 2021c).

Support for different SLOs The notion of SLOs in scalability definitions provides us 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 99.9 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 also 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 traditional distinction between vertical and horizontal scalability, this means upgrading the computing capabilities of existing nodes or expanding the node pool by additional nodes. In orchestrated cloud-native architectures, the assignment of application or service instances is usually abstracted and managed by a tool such Kubernetes. One option is to increase the requested CPU or memory resources for a so-called pod, which may contain multiple

containers. Kubernetes then ensures that the pod will be scheduled on a node, with provides sufficient resources. This can be seen as a form of "virtualized" vertical scaling. On the other hand, pods can also be scaled by increasing their number of replicas, which corresponds to "virtualized" horizontal scaling. Here, we use the term virtualized since it is not guaranteed that indeed more nodes or more powerful nodes are used. Nonetheless, even underlying hardware or VMs should be supported to be scaled such as machine sizes configured via the cloud provider.

3.2 Requirements for the Scalability Measurement Method

Robust statistical grounding Performance experiments exhibit a large variability in their results for various reasons (Maricq et al. 2018). 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 (Abedi and Brecht 2017). In order to obtain reproducible results, measurements should therefore be repeated and the confidence in the final results should be quantified (Papadopoulos et al. 2021). Scalability benchmarking conducts performance experiments to assess whether SLOs are achieved. This means that experiments in scalability evaluations should be executed for a sufficient amount of time as well as repeated multiple times.

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 to find a balance between statistically grounded results and a time-efficient execution.

3.3 Requirements for the Scalability Benchmarking Architecture

Operating a distributed software system in the cloud is a complex task, which indeed motivated the development of powerful orchestration tools such as Kubernetes (Burns et al. 2016). 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. 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 to achieve usability of the benchmarking tool. Hence, we require approaches, which are similar to those that are used for operating cloud-native applications.

Simple Installation Installing or deploying cloud-native tools is often complex as even for a single tool or service, several resources have to be deployed, such as *Deployments*, *Services*, *ConfigMaps* and many others in the case of Kubernetes. This becomes even more difficult if for different clusters or cloud providers, different adjustments have to be made. With our Theodolite benchmarking tool (Henning and Hasselbring 2021c), for example, we experienced that persistent volumes are created differently for different cloud providers, components must explicitly be deployed on certain node classes, or certain features of a

benchmarking setup should be disabled due to missing permissions. To simplify the installation of a benchmarking tool, we thus require the adoption of patterns and tools that are established for setting up cloud-native applications in production.

Declarative benchmark and experiment definition A common situation when operating software systems in orchestrated cloud platform is that the desired and previously configured system state deviates form the actual state. Orchestration tools such as Kubernetes address this by providing declarative APIs, which are used to describe the desired state of the system. The orchestration tools continuously compare the actual state to the desired state and perform the necessary reconfigurations. Accordingly, we also require that cloud-native benchmarking tools should be designed in a way such that users only describe what system they would like to benchmark with which configuration, while the benchmarking tool handles the actual execution.

Support for different benchmarks and SUTs There already exists a set of reference implementations and benchmarks for different types of systems, focusing on different attributes. For a generic benchmarking tool, we require that it should be able to support different benchmarks.

Support for different SUT configurations With benchmarks, often not only different systems or frameworks are compared, but also different configurations or deployment options. A generic benchmarking tool should support setting these configurations via its declarative API such that no new installation or even re-building of the benchmark implementation is required.

4 Scalability Metrics

We proposed and discussed the scalability metrics presented in this section at the *International Workshop on Load Testing and Benchmarking of Software Systems 2021* for the special case of stream processing systems (Henning and Hasselbring 2021a). From there, we received the feedback that these preliminary metrics could be widened in their scope to cover cloud-native applications in general.

Our metrics take up the three attributes of scalability in cloud computing presented in Section 2.2.1 and generalize them according to our requirements from Section 3. 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

$$\mathrm{slo}_s : L \times R \to \{\mathrm{false}, \mathrm{true}\}$$

with $\mathrm{slo}_s(l, r) = \mathrm{true}$ 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. In many cases, both functions are inverse to each other. However, we expect both metrics to have advantages, as discussed in our previous paper (Henning and Hasselbring 2021a).

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:[3]

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

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.

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 to easily determine 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.

According to the requirements identified in Section 3, both our metrics do not make any assumption on the type of load, resource, or SLO. This implies that these metrics allow to evaluate the same system with respect to different load and resources of varying dimensions. Typical load dimensions are, for example, the number of concurrent users at a system, the amount 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 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 (Brataas et al. 2017).

5 Scalability Measurement Method

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 a SUT react to a changing load, cf. Section 2.2.3). Measuring the throughput for a fixed, high load or increasing the load at runtime might cause wrong results (Henning and Hasselbring 2021a). Testing only at a fixed high load also fails to reveal deviations from expected trends, such as

[3] Note that we use a different definition of demand than used for operational analysis of queuing networks (Denning and Buzen 1978).

a linear increase in resource utilization with the load and nearly constant memory occupancy at constant load.

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. Afterwards, we discuss how the requirements for statistical grounding and time-efficient execution relate with each other.

5.1 SLO Experiments

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

Our measurement method deploys the SUT with r resources and generates the constant load l over some 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.

5.2 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.

Figure 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. Figure 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

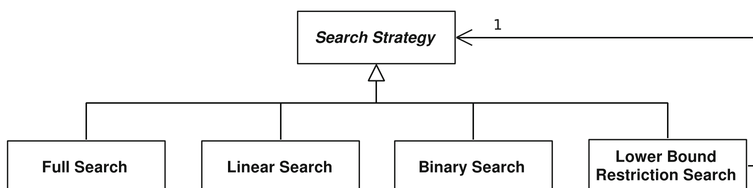


Fig. 2 UML class diagram of different search strategies

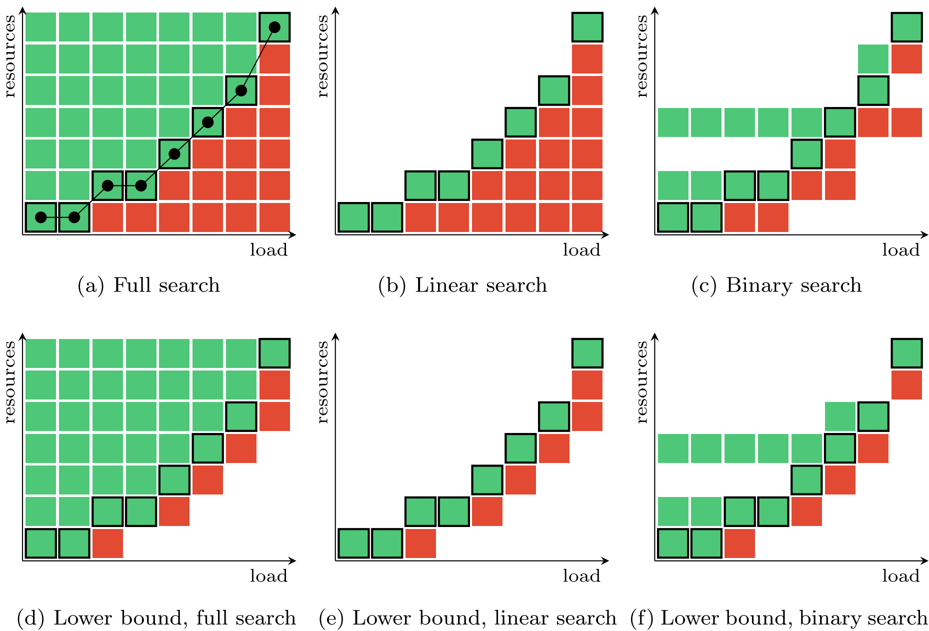


Fig. 3 Comparison of selected search strategies

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. 3a.

Full search The full search strategy (see Fig. 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 experiments, 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. 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. 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. 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 7.4 and show that it does not hold in all cases.

Lower bound restriction The *lower bound restriction* (see Fig. 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. 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:

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

In Section 7.4, we show that for the special case of event-driven microservices, we are safe to make this assumption.

5.3 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 2 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:

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

Table 2 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 measurement 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 δ	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 $\|L'\|$.

6 Scalability Benchmarking Architecture

In this section we propose an architecture for a benchmarking tool that implements our proposed measurement method and, thus, our proposed scalability metrics. Moreover, our architecture is designed according to the requirements defined in Section 3. We start by outlining our proposed benchmarking process in Section 6.1, which distinguishes the definition of benchmarks and their execution. Afterwards, we present a data model for defining these benchmarks and executions in Section 6.2. In Section 6.3, we present how benchmarks and executions defined with this model can be executed in cloud-native environments.

6.1 Overview of the Benchmarking Process

Figure 4 gives an overview of our proposed scalability benchmarking process. In general, we can observe two actors involved in benchmarking:

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. Moreover, they know about relevant load intensity types, resources types, and SLOs, regrading which scalability should be evaluated. Benchmark designers bundle all of this in *Benchmarks*. Benchmarks can be published as supplemental material of 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.

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. (2020). 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

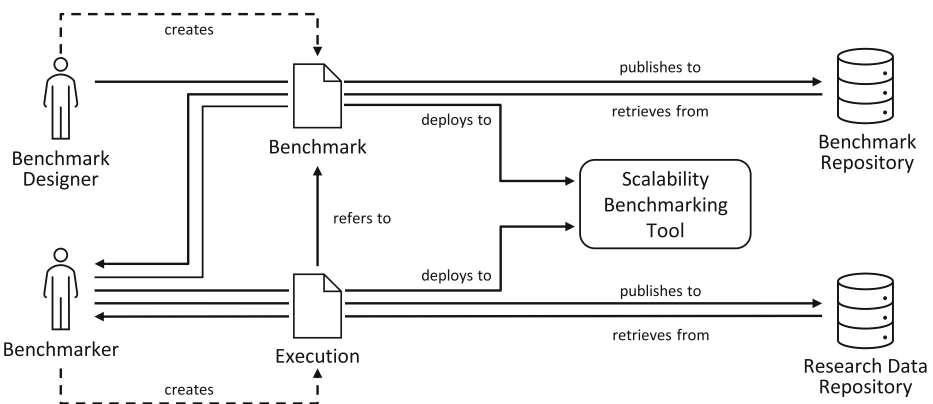


Fig. 4 Context diagram showing how actors interact with our proposed benchmarking tool architecture

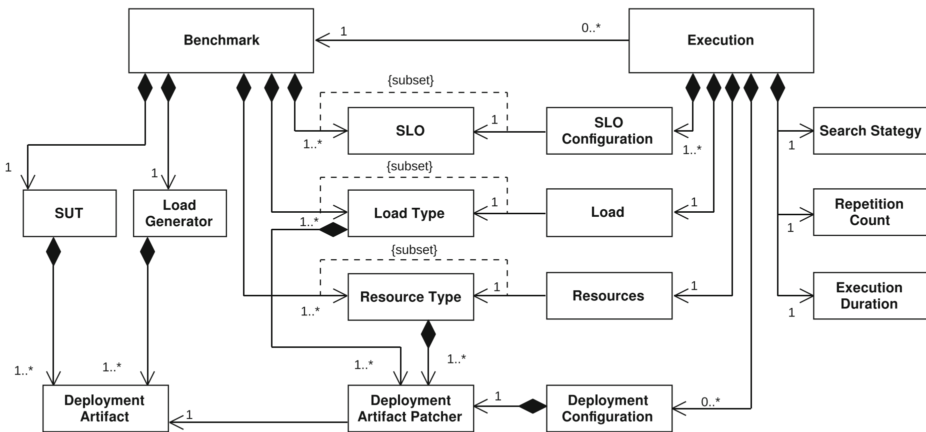


Fig. 5 UML class diagram of our scalability benchmarking data model

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.

6.2 Benchmarking Data Model

Based on the previously distinction between benchmarks and their executions, we propose a data model for defining them in a declarative way. Figure 5 visualizes the central elements of our data model and their relations as UML class diagram. In the following, we describe this data model starting from the central entities Benchmark and Execution.

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, definitions of Kubernetes resources such as Pods, Services, or ConfigMaps.[4]

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.

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*.

[4] 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*, which adjusts 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 data 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 or 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 Section 5, 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.

6.3 Benchmark Execution in Cloud-Native Environments

We propose a benchmarking tool architecture based on the operator pattern (Ibryam and Huss 2019). This pattern is increasingly used to reduce the complexity of operating applications by integrating domain knowledge into the orchestration process. Core of this pattern are the *operator* and, in the case of Kubernetes, so-called *Custom Resource Definitions (CRDs)*. Artifacts (or resources) of these CRDs can be created, altered, or remove by the user via the API of the orchestration tool. In a sophisticated reconciliation process, the operator continuously observes the currently deployed artifacts and may react to changes by creating, modifying or deleting other deployment artifacts.

Figure 6 shows our proposed architecture for a cloud-native scalability benchmarking tool. We envisage CRDs for the Benchmark and Execution entities of our benchmarking

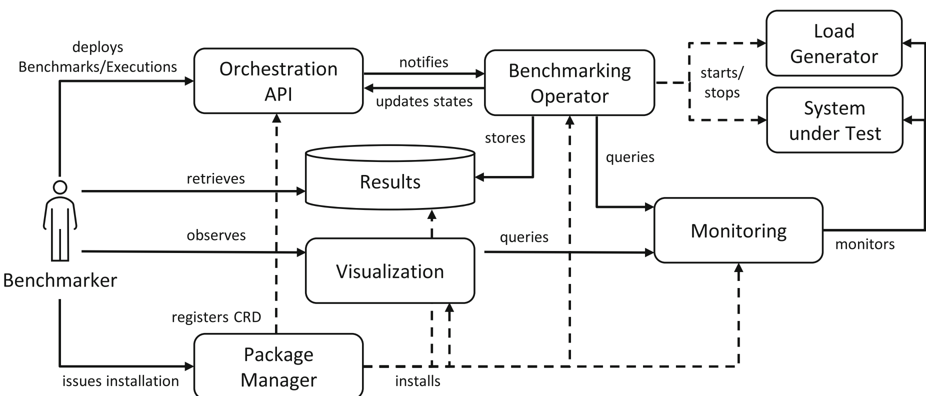


Fig. 6 Proposed benchmarking tool architecture based on the operator pattern

data 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 benchmark 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 component such as Prometheus[5] collects monitoring 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 visualization tool such as Grafana[6] might be used to let benchmarkers observe the execution of benchmarks.

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 such as Helm,[7] which installs the operator, the CRDs as well as dependent systems such as the monitoring and the visualization tool.

7 Experimental Evaluation

In this section, we perform an experimental evaluation of our proposed scalability benchmarking method to answer the research questions posed in Section 1. Specifically, we empirically evaluate the effect of our benchmarking method's configuration options (see Section 5.3) 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.

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. We focus on benchmarks for a specific type of cloud-native applications, namely event-driven microservices, to make the individual results comparable. However, our evaluation method is also intended to serve as a blueprint to repeat our evaluation for other SUTs.

After a detailed description of our experimental setup in Section 7.1, we conduct the following evaluations:

- In Section 7.2, we address RQ 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 7.3, we address RQ 2 and evaluate how many repetitions of an SLO experiment should be performed to decide with sufficiently high confidence whether evaluated SLOs are achieved.

[5] https://prometheus.io

[6] https://grafana.com/grafana/

[7] https://helm.sh

Table 3 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	*lag trend metric*	1
Stream processing engines	Kafka Streams, Flink	2
Cloud providers	Google (GCP), Oracle (OCI), private cloud (SPEL)	3
Total amount of SUTs		24

- In Section 7.4, we address RQ 3 and evaluate how the assessment of SLOs evolves with increasing resource amounts. This evaluation helps in determining whether the binary search strategy can applied with our *demand* metric and whether the lower bound restriction strategy can be applied with our *capacity* metric.
- In Section 7.5, we perform a similar evaluation to address RQ 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 7.6.

7.1 Experiment Setup

In the following, we present the general experiment setup for the following evaluations. First, we introduce our Theodolite scalability benchmarking tool, which implements our proposed benchmarking method. Afterwards, we describe the SUTs used for our evaluations. Our SUTs are implementations of benchmarks for event-driven microservices. Event-driven microservices are an emerging architectural style, in which microservices primarily communicate via asynchronous messaging. To parallelize data processing, such microservices employ distributed stream processing techniques (Fragkoulis et al. 2020). We consider 4 benchmarks, which are implemented by 2 stream processing engines and executed in 3 cloud environments. This results in 24 SUTs as summarized in Table 3.

7.1.1 The Theodolite Reference Implementation

Our cloud-native benchmarking tool Theodolite implements the architecture presented in Section 6 and, thus, our proposed benchmarking method. Theodolite is open source[8] research software (Hasselbring et al. 2020) with publicly available documentation.[9] We presented an early version of Theodolite in a previous publication (Henning and Hasselbring 2021c) along with benchmarks for distributed stream processing engines. Section 8 discusses how we extend our previous work.

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

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

We use the following technologies in Theodolite: Kubernetes as orchestration tool, Helm as package manager, Prometheus for monitoring the SUT, and Grafana for the visualization. These technologies are generally understood as being part of the cloud-native landscape.[10] In addition to our proposed architecture, Theodolite includes first-class support for configuring and monitoring a messaging system. Such a messaging system is used in almost all benchmarking studies for distributed stream processing and should therefore be also included in a scalability benchmarking tool (Henning and Hasselbring 2021c). For the implementation of our proposed architecture, this means that for an SLO experiment not only Kubernetes resources are deployed, but also corresponding messaging topics are created. We decided to not consider the messaging system as part of the SUT as, for example, in the case of Apache Kafka, its deployment takes a significant amount of time.

7.1.2 The Theodolite Stream Processing Benchmarks

Theodolite accepts benchmarks that are defined according to the data model depicted in Fig. 5. That is, a benchmark consists of a task sample (SUT and load generator) as well as supported load types, resource types, and SLOs. Theodolite comes with 4 benchmarks for event-driven microservices, which we use for our evaluation. In the following, we give an overview of these benchmarks and show how we employ them.

Task Samples In our previous paper (Henning and Hasselbring 2021c), we derived 4 task samples from common stream processing use cases for analyzing Industrial Internet of Things sensor data (Henning et al. 2021). The individual tasks samples are:

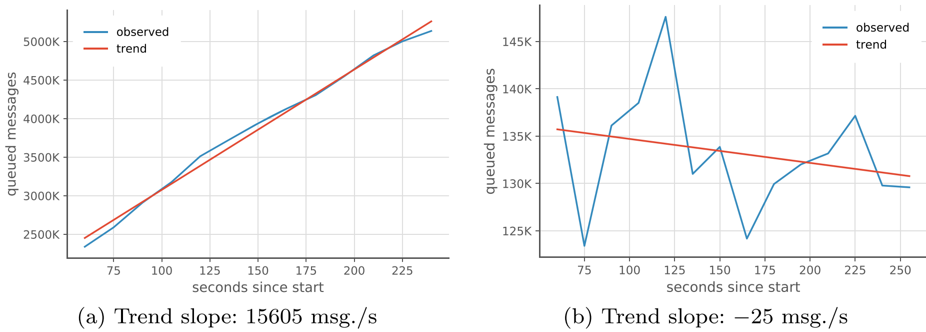
- UC1 Incoming messages are written to an external database. This task sample focuses solely on data processing and, therefore, is stateless.
- UC2 Incoming messages are aggregated by a message key within fixed-size, non-sliding time windows to reduce the total quantity of messages (downsampling).
- UC3 Incoming messages are aggregated by their key and a time attribute (e.g., day of week) in large sliding windows. Such computations are common to compute a seasonal trend over a long period of time (e.g., an aggregated weekly course over a period of several months).
- UC4 Incoming messages are aggregated to groups and groups of groups in a hierarchical fashion (Henning and Hasselbring 2020).

For a more detailed description, please refer to our original publication (Henning and Hasselbring 2021c).

We evaluate implementations of these tasks samples with the two stream processing engines Apache Kafka Streams (Wang et al. 2021) and Apache Flink (Carbone et al. 2015).

Load type A stream processing engine is usually subject to a load of messages coming from a central messaging system. In many of such systems, messages contain a key, which is used for data partitioning and, thus, the major means for parallelizing stream processing tasks (Fragkoulis et al. 2020). In our evaluation, we therefore focus on scaling with the amount of distinct message keys per unit of time. For benchmark UC1, UC2, and UC3, the load type corresponds to the amount of keys, where for each key one message per second is generated. For benchmark UC4, the load type corresponds to nested groups n (Henning and

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



(a) Trend slope: 15605 msg./s (b) Trend slope: −25 msg./s

Fig. 7 Examples of the monitored lag and the trend line computed using linear regression (Henning and Hasselbring 2021c)

Hasselbring 2021c), which results in 4^n keys, generating one message per second. For reasons of conciseness, we present the generated load also as 4^n messages per second instead of the group size n in the results tables of our evaluation.

Resource type The resource type used in our evaluation is the number of instances of the stream processing engine. This corresponds to what we referred to as "virtualized" horizontal scaling in Section 3. 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. Flink's coordinating *Jobmanager* pod is not scaled, which is the suggested deployment.[11]

SLO The SLO we use for this evaluation is based on the *lag trend metric* (Henning and Hasselbring 2021a). The lag of a stream processing job describes how many messages are queued in the messaging system, which have not been processed yet. The lag trend 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. Figure 7 illustrates the concept of the lag trend.

We use the lag trend metric to define an SLO, whose function evaluates to true if the lag trend does not exceed a certain threshold. The basic idea is: A fluctuating lag is acceptable (e.g., because message are consumed in batches) as long as the amount of queued messages does not increase over a longer period of time. Ideally, the threshold should be 0 as a non-positive lag trend means that messages can be processed as fast as they arrive. However, it makes sense to allow for a small increase as even when observing an almost constant lag, a slightly rising or falling trend line will be computed due to outliers.

We expect that in most cases, checking the lag trend alone suffices as an SLO for stream processing engines. The architectures of modern engines make it unlikely that SLOs such as a maximum tolerable processing latency can be fulfilled by scaling provisioned resources. An advantage of defining an SLO based on the lag is that it can be collected very efficiently

[11] https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/deployment/resource-providers/standalone/kubernetes/#standby-jobmanagers

Table 4 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 version	1.19.9-gke.1900	1.19.7	1.18.6
Kafka brokers	3	3	10

and does require data from the stream processing engine, which might be incorrect under high load. In our experiments, Prometheus queries the current lag every 15 s from Apache Kafka, the messaging system used in our experiments.

7.1.3 Evaluated Cloud Platforms

The experimental evaluations presented in this section are performed in 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. 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 4 summarizes the configuration of the Kubernetes clusters, we use in our evaluation.

7.1.4 Replication Package

We provide a replication package and the collected data of our experiments as supplemental material (Henning and Hasselbring 2021b), allowing other researchers to repeat and extend our work. Our replication package includes the Theodolite *Executions* (see Section 6.2) used in our experiments and interactive notebooks used for analyzing our experiment results. An additional online version of our notebooks is available as a web service.[12]

7.2 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 evaluate how the choice of warm-up period and

[12] https://mybinder.org/v2/zenodo/10.5281/zenodo.5596982

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.

7.2.1 Experiment Design

With this evaluation, we perform SLO experiments for all 24 SUTs depicted in Table 3. 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 access, 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 load-resource combinations 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 (Henning and Hasselbring 2021c). 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. Also for this case we find approximately matching instance counts, but due to higher instance numbers, we use two instances more and less to represent over or underprovisioning. Table 5 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 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 (Henning and Hasselbring 2021b) allows to evaluate our experiment results according to the described method for different warm-up durations, allowed errors and lag trend thresholds.

Table 5 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 ($inst^{\vee}$), overprovisioning ($inst^{\wedge}$), and resources that approximately match the demand ($inst^{\approx}$)

SUT			Load	Resource amounts		
cloud	Engine	bench.	mes./s	$inst^{\vee}$	$inst^{\approx}$	$inst^{\wedge}$
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
	KStreams	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
	KStreams	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
	KStreams	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

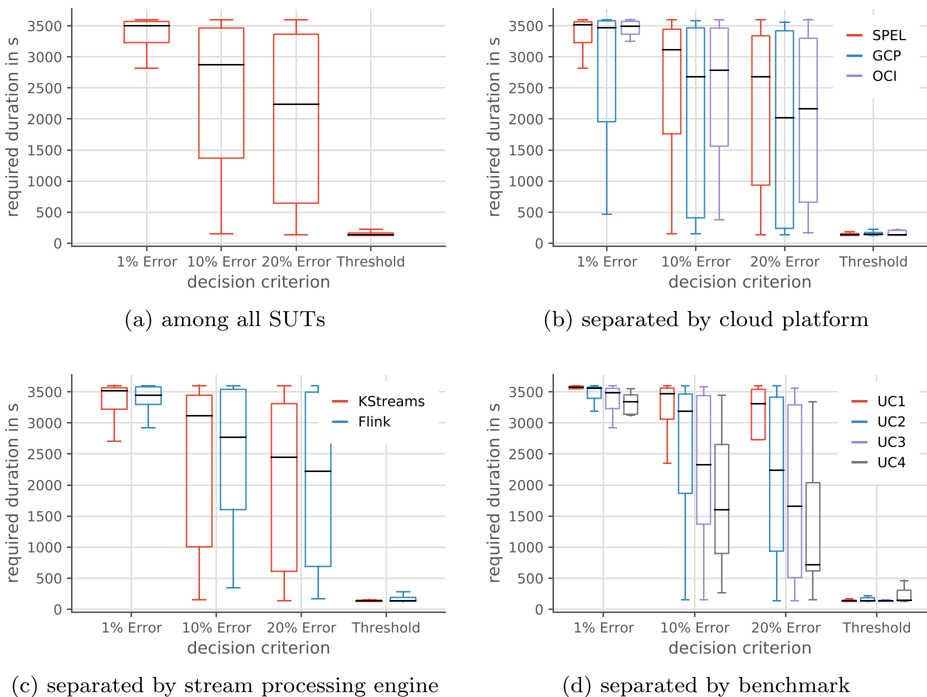
7.2.2 Results and Discussion

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. Figure 8 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 engine 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. 8, 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 5.3), 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 engine, and benchmark. While Fig. 8 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 7.1.2 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.

Figure 9 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 execution times 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 engine

(d) separated by benchmark

Fig. 8 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

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 6 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. 9, we see that certain resource-load combinations require significant longer 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.

Required experiment duration 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 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.

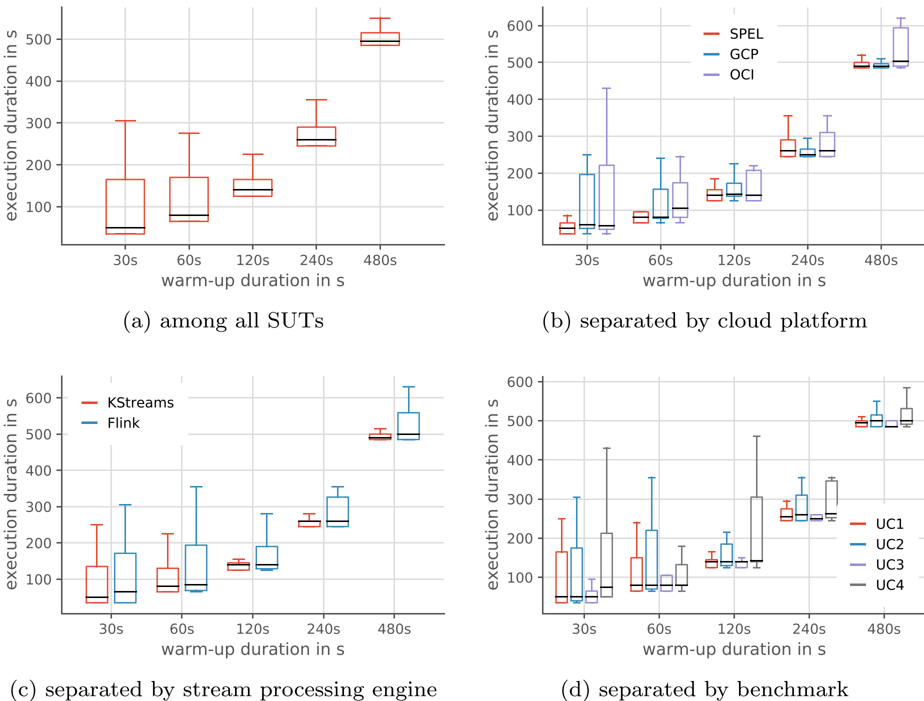


Fig. 9 Box plots showing the required execution duration among all SUTs and cloud providers for different warm-up durations. Whiskers are restricted to 1.5×IQR and outliers laying below or above the whiskers are omitted for readability

Table 6 Required experiment duration of SLO experiments for different SUTs, load intensities (mes./s), and resource amounts ($inst^{\vee}$, $inst^{\approx}$, and $inst^{\wedge}$) with a warm-up duration of 120 s

				req. execution time in s		
cloud	Engine	bench.	mes./s	$inst^{\vee}$	$inst^{\approx}$	$inst^{\wedge}$
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
	KStreams	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
	KStreams	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
	KStreams	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

7.3 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 of their result. In this section, we evaluate how many repetitions are required to decide with sufficiently high confidence whether SLOs are met.

7.3.1 Experiment Design

As in the previous evaluation, we perform SLO experiments for all 24 SUTs depicted in Table 3 with the same amounts of resources and load intensities (see Table 5). According to our results from the evaluation of warm-up and experiment duration, we run each SLO

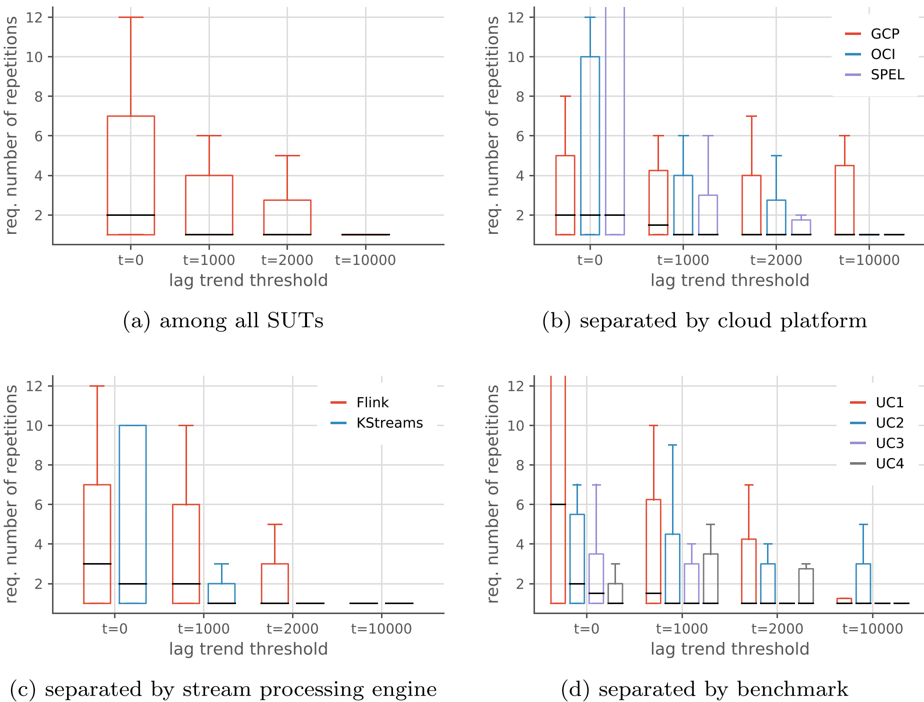


Fig. 10 Box-plots showing the required number of repetitions of SLO experiments for different thresholds. Whiskers are restricted to 1.5×IQR and outliers laying below or above the whiskers are omitted for readability

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. (2020) to apply the Central Limit Theorem.

For the majority of SUTs, we observed 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 (Kounev et al. 2020). 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 approximate the required number of repetitions n for such a 95% confidence interval with:

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

7.3.2 Results and Discussion

Figure 10 summarizes the approximated number of repetitions for different thresholds t as box plots. Our replication package (Henning and Hasselbring 2021b) allows to obtain 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 engine, and the benchmark. Generally, when looking at thresholds $t \geq 1000$, 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 (Leitner and Cito 2016). 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 7 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. 10, we can see that in certain cases very high numbers of repetitions (highlighted 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* function when accepting a small error in the function.

Required number of repetitions 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 5 repetitions are sufficient to determine a close approximation of the resource demand.

7.4 Evaluation of SLO 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 achieved before for lower resource amounts.

7.4.1 Experiment Design

In this evaluation, we 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. To not go beyond the scope of this paper, we focus on the two

Table 7 Required number of repetitions of SLO experiments for different SUTs, load intensities (mes./s), and resource amounts ($inst^{\vee}$, $inst^{\approx}$, and $inst^{\wedge}$). The threshold for the lag trend is $t = 2000$

Cloud	Engine	bench.	mes./s	Required repetitions		
				$inst^{\vee}$	$inst^{\approx}$	$inst^{\wedge}$
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
	KStreams	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
	KStreams	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
	KStreams	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

benchmarks UC2 and UC3 and restrict our experiments to the private cloud and the Google cloud. This results in 8 SUTs (see Table 8).

For each SUT, we conduct a set of isolated SLO experiments, in which we generate a constant load, equal to the loads of Table 5, and different resource amounts. We repeat each SLO experiment 5 times with 5 minutes of experiment duration including 2 minutes of warm-up. Table 8 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.

Table 8 SUT configuration for evaluations of the SLO with increasing resources

	SPEL				GCP			
	Kafka Streams		Flink		Kafka Streams		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	≤ 15	≤ 20	≤ 20	≤ 25	≤ 10	≤ 10	≤ 10	≤ 10
Duration	5 minutes, including 2 minutes warm-up							
Repetitions	5							

7.4.2 Results and Discussion

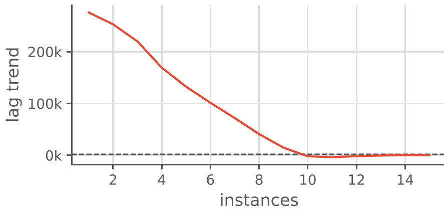
Figure 11 shows for each evaluated SUT how the median lag trend evolves with increasing resource amounts. Additionally, a horizontal line at a lag trend of 2000 is drawn to visualize a possible threshold for the *lag trend metric*.

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 partitions and the stream processing engines might work particularly efficiently if the partition count is a multiple of the instance count.

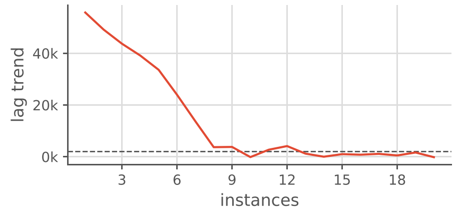
For the experiments in the public cloud, these effects can not clearly be observed. However, we observe that with a cluster size of 3 nodes, the lag trend increases again after some point when further instances are added. As we expect this to be due to exhausted node resources, we repeat the same experiments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

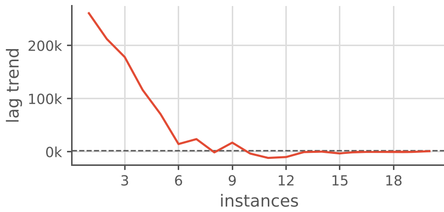
SLO assessment with increasing resources In general, the lag trend decreases with increasing resources. However, there are clearly certain resource configurations which perform better than others. This means, the binary search can not necessarily be used with the demand metric, while the lower bound restriction can not 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.



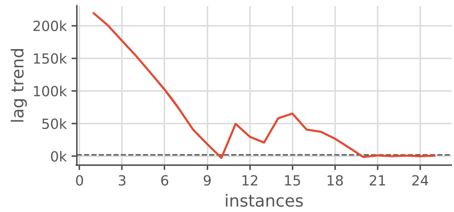
(a) SPEL, Kafka Streams, UC2



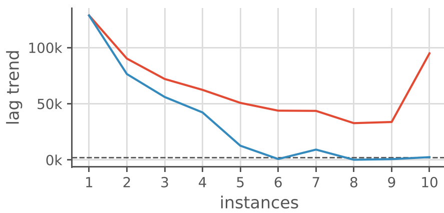
(b) SPEL, Kafka Streams, UC3



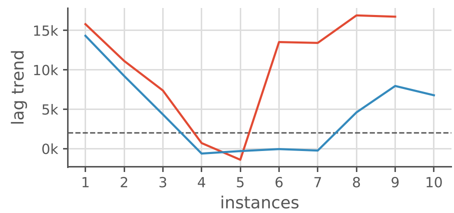
(c) SPEL, Flink, UC2



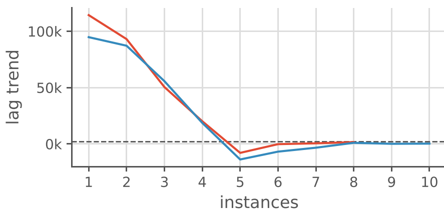
(d) SPEL, Flink, UC3



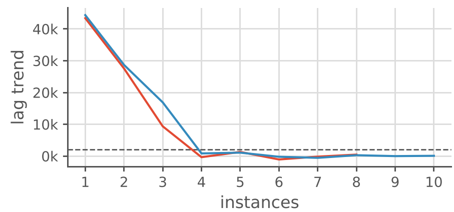
(e) GCP, Kafka Streams, UC2



(f) GCP, Kafka Streams, UC3



(g) GCP, Flink, UC2



(h) GCP, Flink, UC3

Fig. 11 Lag trend with increasing resource amounts for different SUTs. 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

7.5 Evaluation of SLO with Increasing Load

In this section, we now 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.

7.5.1 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 7.4 and is summarized in Table 9.

7.5.2 Results and Discussion

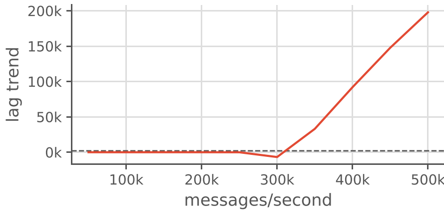
Figure 12 shows for each evaluated SUT how the median lag trend evolves with increasing load intensities. Again, a horizontal line at a lag trend of 2000 visualizes a possible threshold for the *lag trend metric*.

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

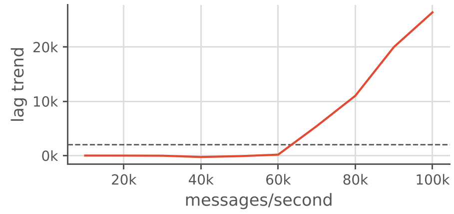
In summary, we conclude that the binary search strategy can be used to evaluate large sets of load intensities with the *capacity* metric, at least when benchmarking event-driven microservices with an SLO based on the *lag trend metric*. As furthermore the computed lag trend is monotonically increasing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

Table 9 SUT configuration for evaluations of SLO with increasing load

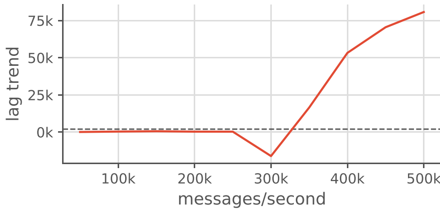
	SPEL				GCP			
	Kafka Streams		Flink		Kafka Streams		Flink	
	UC2	UC3	UC2	UC3	UC2	UC3	UC2	UC3
mes./s	≤ 500000	≤ 100000	≤ 500000	≤ 400000	≤ 250000	≤ 50000	≤ 250000	≤ 100000
Instances	10	13	11	20	6	5	5	6
Duration	5 minutes, including 2 minutes warm-up							
Repetitions	5							



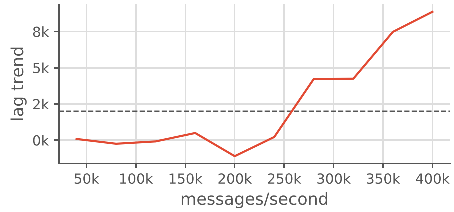
(a) SPEL, Kafka Streams, UC2



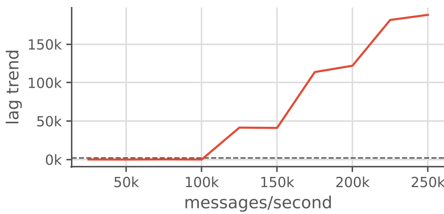
(b) SPEL, Kafka Streams, UC3



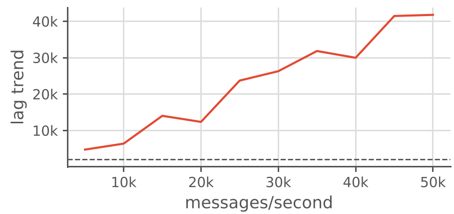
(c) SPEL, Flink, UC2



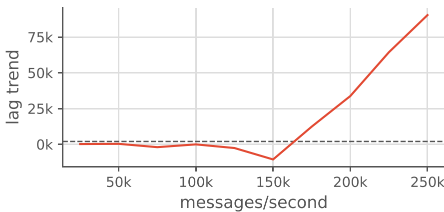
(d) SPEL, Flink, UC3



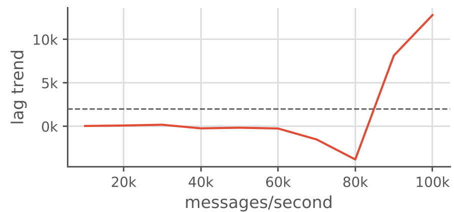
(e) GCP, Kafka Streams, UC2



(f) GCP, Kafka Streams, UC3



(g) GCP, Flink, UC2



(h) GCP, Flink, UC3

Fig. 12 Lag trend with increasing load for different SUTs

SLO assessment with increasing load 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.

7.6 Threats to Validity

The goal of this experimental evaluation was 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 to make only little assumptions regarding the underlying hardware or software infrastructure (Bermbach et al. 2017). Due to techniques such as containerization, cloud-native application 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 was to investigate the variability of results and hence the underlying processing capabilities of stream processing engines. Nevertheless, we reused 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 performed our experiments in a relatively short time frame. We performed 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 (Iosup et al. 2011), more recent research were not able to confirm this (Leitner and Cito 2016). 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 With our Theodolite benchmarking tool, we only evaluated one type of cloud-native applications, namely event-driven microservice that use distributed stream processing engines. Our results regarding required experiment duration and required number of repetitions should therefore not be generalized for other types of cloud-native applications, which potentially use other SLOs. Furthermore, we only considered two stream processing engines and focused on single types of load and resources. Similar limitations apply to the evaluated cloud environments. In the public clouds, we only evaluated virtual machines, while in the private cloud, we only evaluated bare metal servers. Furthermore, the nodes in the VM were 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.

8 Related Work

In this section, we discuss related work regarding our proposed scalability benchmarking metrics, measurement method, and tool architecture.

8.1 Scalability Metrics

Both our proposed metrics are functions. Lehrig et al. (2018) 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).

In traditional parallel and distributed systems research, it is common to describe 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 (Hoefler and Belli 2015): 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. This distinction cannot directly be transferred to cloud-native applications. Such 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 access whether a cloud-native applications is still functioning if the load increases. This is what Bondi (2000) defines as load scalability.

Also originating from traditional parallel and distributed systems research, the Universal Scalability Law (Gunther 2007; Gunther et al. 2015) 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 that it allows to easily rank different SUTs. However, to the best of our knowledge, there is no 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 their seminal work on cloud benchmarking, Kossmann et al. (2010) 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 to detect 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 paper, 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. (2015) and Becker et al. (2015) used systematic methods to derive scalability metrics. Lehrig et al. (2015) conducted a systematic literature review and found only one scalability metric evaluated in a practical setting at the time of the study (Tsai et al. 2011). This metric requires scaling to be quick, which contradicts most scalability definitions (Lehrig et al. 2015). Becker et al. (2015) use the goal question metric (GQM) method to derive two scalability metrics. The first metric "scalability range" describes the maximum load a 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 a SUT can achieve its SLOs if the load increases by a certain rate (load per unit of time). As highlighted in Section 2.2.3, using time for describing scalability is uncommon. Both metrics from Becker et al. (2015) do not consider the resource amounts needed to achieved its SLOs.

More recently, Brataas et al. (2017) proposed a scalability metric function relating load, resources, and SLOs. This metric matches our *capacity* metric, although formalized differently. In our previous work (Henning and Hasselbring 2021a), we emphasize that our *demand* metric is more strictly aligned to scalability definitions as it considers load to be the

input variable. However, the *capacity* metric may have the advantage to show the resource amounts with which adding further resources result in worse performance.

The *domain-based metric* was recently presented by Avritzer et al. (2020) to access 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 being added in order to satisfy SLO requirements. Instead, different resource amounts are considered as separate deployment options.

8.2 Scalability Measurement Methods

While several scalability evaluations describe scalability as functions of resources, many of them do not conduct isolated experiments for different load intensities (Brunner et al. 2015; Al-Said Ahmad and Andras 2019; Karakaya et al. 2017; Nasiri et al. 2019; Karimov et al. 2018). 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 (Henning and Hasselbring 2021a). Sometimes also the provisioned resources are auto-scaled in the background (Brunner et al. 2015; Al-Said Ahmad and Andras 2019), 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. (2014), 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 (Brataas et al. 2017; Cunha et al. 2017). As Brataas et al. (2017) use a scalability metric similar to our *capacity* metric, they pursue similar ideas for bounding the execution times of their experiments. In line with our linear search strategy, they stop testing higher load intensities once they detect that a load intensity cannot be handled anymore by a certain resource level. However, their paper discusses this topic only briefly and does not provide a systematic measurement method. As Brataas et al. (2017) 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. (2017) 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. (2011), Leitner and Cito (2016), Abedi and Brecht (2017), Maricq et al. (2018), or Laaber et al. (2019). He et al. (2019) and He et al. (2021) and Bulej et al. (2020) 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. Combining the methods of He et al. (2019) and He et al. (2021) 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. (2020) 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.

8.3 Scalability Benchmarking Architectures

Most benchmarking studies, which benchmark scalability in the cloud, do not provide a benchmarking tool or a corresponding architecture (Kuhlenkamp et al. 2014; Al-Said Ahmad and Andras 2019; Karimov et al. 2018). General architectures for cloud benchmarking tools are presented, for example, by Bermbach et al. (2017) and Iosup et al. (2014). Our architecture builds upon such architectures but is particular suited for cloud-native 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. (2017), but without distinguishing between benchmarks and executions. We expect the separation of benchmark and its execution to particularly foster reproducibility.

Recently, Avritzer et al. (2021) presented an architecture for measuring their *domain-based metric* (Avritzer et al. 2020). 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 data 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 our previous publication (Henning and Hasselbring 2021c), 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 engines. With this paper, we substantially extend the experiment control component of that architecture to increase usability. Specifically, we propose to apply the operator pattern along with a flexible data model for defining benchmarks and their executions in a declarative manner. In general, the focus of our previous publication was proposing specific benchmarks for microservices that use stream processing frameworks, while with this work, we present a general scalability benchmarking method for cloud-native applications.

To the best of our knowledge, we present the first scientific approach for applying the operator pattern to benchmark cloud-native applications. Recently, (Merenstein et al. 2020) proposed an architecture based on the operator pattern for benchmarking cloud-native storage infrastructure. Similar to our identified requirements, the authors highlight that such an architecture can improve usability, which is particularly important in complex cloud-native environments. Additionally, some benchmark operators emerged in the cloud-native community for benchmarking the performance of Kubernetes installations.[13] 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.

9 Conclusions and Future Work

This paper studied how scalability of cloud-native applications can be benchmarked. Based on a set of requirements derived as part of this study, we presented a benchmarking method,

[13] https://kubestone.io, https://github.com/cloud-bulldozer/benchmark-operator

consisting of two scalability metrics, a corresponding measurement method, and an architecture for a scalability benchmarking tool. Our scalability metrics are strictly aligned with definitions of scalability in cloud computing. They support arbitrary types of applications, which may be subject to different types of load, may react to these load by increasing different types of resource, and may have different SLOs. These scalability metrics can be measured with our proposed scalability measurement method. It supports configuration options to individually balance the trade-off between a time-efficient execution and statically grounded results. Our benchmarking tool architecture provides a platform for executing scalability benchmarks according to our proposed metrics and measurement method. Its design focuses on usability by adopting common cloud-native patterns.

In an extensive experimental evaluation, we analyze the trade-off between a time-efficient execution and statically grounded results for the case of cloud-native applications that employ stream processing engines. We run experiments in two public and one private cloud infrastructure and find that in most cases only little ($\leq$ 5) repetitions and short execution times ($\leq$ 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. Based on our experimental results, we recommend that benchmarkers focus on evaluating larger load and resource sets instead of on exhaustive repetitions of individual SLO experiments, as the former is more likely to present a fairly accurate picture of a systems scalability.

With this study, we lay the foundation for comprehensive scalability benchmarking of cloud native applications. Interesting future research directions to explore are, for instance, benchmarking different cloud-native technologies such as the stream processing engines studied in this paper, comparing scalability with different types of resources (e.g., vertical vs. horizontal scaling), or evaluating how costs in public clouds evolve with increasing load intensities and, thus, increasing resource demands.

Besides actual scalability benchmarking, future work may also extend the evaluation of our method. Possible extensions are evaluating how cloud configurations such as cluster size or node type influence the results as well as repeating our experiments at different times of day. Also an evaluation with other benchmarks, applications, load types, resource types, and SLOs would emphasize the applicability of our proposed method.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarations

Competing Interests The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Abedi A, Brecht T (2017) Conducting repeatable experiments in highly variable cloud computing environments. Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering.

Al-Said Ahmad A, Andras P (2019) Scalability analysis comparisons of cloud-based software services. Journal of Cloud Computing 8(1):1–17. https://doi.org/10.1186/s13677-019-0134-y

Avritzer A, Camilli M, Janes A, Russo B, Jahč J, Hoorn A, Britto R, Trubiani C (2021) PPTAM$^{\lambda}$: What, where, and how of cross-domain scalability assessment. In: 2021 IEEE 18th international conference on software architecture companion (ICSA-C). https://doi.org/10.1109/ICSA-C52384.2021. 00016

Avritzer A, Ferme V, Janes A, Russo B, van Hoorn A, Schulz H, Menasché D, Rufino V (2020) Scalability assessment of microservice architecture deployment configurations: A domain-based approach leveraging operational profiles and load tests. J Syst Softw 165:110564. https://doi.org/10.1016/j.jss.2020.110564

Balalaie A, Heydarnoori A, Jamshidi P (2016) Microservices architecture enables DevOps: Migration to a cloud-native architecture. IEEE Software 33(3):42–52. https://doi.org/10.1109/ms.2016.64

Becker M, Lehrig S, Becker S (2015) Systematically deriving quality metrics for cloud computing systems. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering. https://doi.org/10.1145/ 2668930.2688043

Bermbach D, Wittern E, Tai S (2017) Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective, 1st edn. Springer Publishing Company, Incorporated

Bondi AB (2000) Characteristics of scalability and their impact on performance. In: Proceedings International Workshop on Software and Performance. https://doi.org/10.1145/350391.350432

Brataas G, Herbst N, Ivanšek S, Polutnik J (2017) Scalability analysis of cloud software services. In: Proceedings International Conference on Autonomic Computing (ICAC). https://doi.org/10.1109/ICAC.2017.34

Brataas G, Martini A, Hanssen GK, Ræder G (2021) Agile elicitation of scalability requirements for open systems: A case study. J Syst Softw 182:111064. https://doi.org/10.1016/j.jss.2021.111064, https://www.sciencedirect.com/science/article/pii/S0164121221001618

Brunner S, Blöchlinger M, Toffetti G, Spillner J, Bohnert TM (2015) Experimental evaluation of the cloud-native application design. In: 2015 IEEE/ACM 8th international conference on utility and cloud computing (UCC). https://doi.org/10.1109/UCC.2015.87

Bulej L, Horký V, Tuma P, Farquet F, Prokopec A (2020) Duet benchmarking: Improving measurement accuracy in the cloud. In: Proceedings of the ACM/SPEC international conference on performance engineering. https://doi.org/10.1145/3358960.3379132

Burns B, Grant B, Oppenheimer D, Brewer E, Wilkes J (2016) Borg, Omega, and Kubernetes. Commun. ACM 59(5):50–57. https://doi.org/10.1145/2890784

Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K (2015) Apache Flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36(4)

Cloud Native Computing Foundation (2018) CNCF cloud native definition v1.0. https://github.com/cncf/toc/blob/main/DEFINITION.md

Cunha M, Mendonça NC, Sampaio A (2017) Cloud Crawler: a declarative performance evaluation environment for infrastructure-as-a-service clouds. Concurrency and Computation: Practice and Experience 29(1):e3825. https://doi.org/10.1002/cpe.3825

Denning PJ, Buzen JP (1978) The operational analysis of queueing network models. ACM Comput Surv 10(3):225–261. https://doi.org/10.1145/356733.356735

Duboc L, Rosenblum D, Wicks T (2007) A framework for characterization and analysis of software system scalability. In: Proceedings European software engineering conference/symposium on the foundations of software engineering. https://doi.org/10.1145/1287624.1287679

Folkerts E, Alexandrov A, Sachs K, Iosup A, Markl V, Tosun C (2013) Benchmarking in the cloud: What it should, can, and cannot be. In: Nambiar R, Poess M (eds) Selected topics in performance evaluation and benchmarking. Springer, Berlin, Heidelberg, pp 173–188

Fragkoulis M, Carbone P, Kalavri V, Katsifodimos A (2020) A survey on the evolution of stream processing systems. arXiv:2008.00842

Gannon D, Barga R, Sundaresan N (2017) Cloud-native applications. IEEE Cloud Computing 4(5):16–21. https://doi.org/10.1109/MCC.2017.4250939

Gorton I (2022) Foundations of Scalable Systems, 1st edn. O'Reilly

Gunther NJ (2007) Guerrilla capacity planning: A tactical approach to planning for highly scalable applications and services, 1st edn. Springer, Berlin

Gunther NJ, Puglia P, Tomasette K (2015) Hadoop superlinear scalability. Commun ACM 58(4):46–55. https://doi.org/10.1145/2719919

Hasselbring W (2021) Benchmarking as empirical standard in software engineering research. In: Proceedings of the Evaluation and Assessment in Software Engineering. https://doi.org/10.1145/3463274.3463361

Hasselbring W, Carr L, Hettrick S, Packer H, Tiropanis T (2020) Open source research software. Computer 53(8):84–88. https://doi.org/10.1109/MC.2020.2998235

He S, Liu T, Lama P, Lee J, Kim IK, Wang W (2021) Performance testing for cloud computing with dependent data bootstrapping. In: 2021 36th IEEE/ACM international conference on automated software engineering (ASE). https://doi.org/10.1109/ASE51524.2021.9678687

He S, Manns G, Saunders J, Wang W, Pollock L, Soffa ML (2019) 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. https://doi.org/10.1145/3338906.3338912

Henning S, Hasselbring W (2020) Scalable and reliable multi-dimensional sensor data aggregation in data-streaming architectures. Data-Enabled Discovery and Applications 4(1). https://doi.org/10.1007/s41688-020-00041-3

Henning S, Hasselbring W (2021) How to measure scalability of distributed stream processing engines? In: Companion of the ACM/SPEC International Conference on Performance Engineering. https://doi.org/10.1145/ 3447545.3451190

Henning S, Hasselbring W (2021) Replication package for: A configurable method for benchmarking scalability of cloud-native applications. https://doi.org/10.5281/zenodo.5596982

Henning S, Hasselbring W (2021) Theodolite: Scalability benchmarking of distributed stream processing engines in microservice architectures. Big Data Research 25:100209. https://doi.org/10.1016/j.bdr.2021.100209

Henning S, Hasselbring W, Burmester H, Möbius A, Wojcieszak M (2021) Goals and measures for analyzing power consumption data in manufacturing enterprises. Journal of Data, Information and Management 3(1):65–82. https://doi.org/10.1007/s42488-021-00043-5

Herbst NR, Kounev S, Reussner R (2013) Elasticity in cloud computing: What it is, and what it is not. In: Proceedings International Conference on Autonomic Computing (ICAC 13)

Hoefler T, Belli R (2015) 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. https://doi.org/10.1145/2807591.2807644

Huppler K (2009) The art of building a good benchmark. In: Nambiar R, Poess M (eds) Performance evaluation and benchmarking

Ibryam B, Huss R (2019) Kubernetes Patterns: Reusable Elements for Designing Cloud Native Applications, 1st edn. O'Reilly

Iosup A, Prodan R, Epema D (2014) IaaS cloud benchmarking: Approaches, challenges, and experience. In: Li X, Qiu J (eds) Cloud Computing for Data-intensive Applications. https://doi.org/10.1007/978-1-4939-1905-5_4

Iosup A, Yigitbasi N, Epema D (2011) On the performance variability of production cloud services. In: 2011 11th IEEE/ACM international symposium on cluster, cloud and grid computing. https://doi.org/10.1109/CCGrid.2011.22

Islam S, Lee K, Fekete A, Liu A (2012) How a consumer can measure elasticity for cloud platforms. In: Proceedings of the 3rd ACM/SPEC international conference on performance engineering. https://doi.org/10.1145/2188286. 2188301

Jogalekar P, Woodside M (2000) Evaluating the scalability of distributed systems. IEEE Transactions on Parallel and Distributed Systems 11(6):589–603. https://doi.org/10.1109/71.862209

Karakaya Z, Yazici A, Alayyoub M (2017) A comparison of stream processing frameworks. In: Proceedings International conference on computer and applications (ICCA). https://doi.org/10.1109/COMAPP.2017.8079733

Karimov J, Rabl T, Katsifodimos A, Samarev R, Heiskanen H, Markl V (2018) Benchmarking distributed stream data processing systems. In: Proceedings International conference on data engineering (ICDE). https://doi.org/10.1109/ICDE.2018.00169

Kleppmann M (2017) Designing Data-Intensive Applications, 1st edn. O'Reilly

Knoche H, Hasselbring W (2019) Drivers and barriers for microservice adoption – a survey among professionals in Germany. Enterprise Modelling and Information Systems Architectures (EMISAJ) – International Journal of Conceptual Modeling 14(1):1–35. https://doi.org/10.18417/emisa.14.1

Kossmann D, Kraska T, Loesing S (2010) An evaluation of alternative architectures for transaction processing in the cloud. In: Proceedings SIGMOD International Conference on Management of Data. https://doi.org/10.1145/ 1807167.1807231

Kounev S, Lange K-D, von Kistowski J (2020) Systems Benchmarking: For Scientists and Engineers, 1st edn. Springer Publishing Company, Incorporated
Kratzke N, Quint P-C (2017) Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. J Syst Softw 126:1–16. https://doi.org/10.1016/j.jss.2017.01.001
Kuhlenkamp J, Klems M, Röss O (2014) Benchmarking scalability and elasticity of distributed database systems. Proc VLDB Endow 7(12):1219–1230. https://doi.org/10.14778/2732977.2732995
Laaber C, Scheuner J, Leitner P (2019) Software microbenchmarking in the cloud. how bad is it really? Empirical Softw Engg 24(4):2469–2508. https://doi.org/10.1007/s10664-019-09681-1
Lehrig S, Eikerling H, Becker S (2015) Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics. In: Int Conf Quality of Software Architectures. https://doi.org/10.1145/2737182.2737185
Lehrig S, Sanders R, Brataas G, Cecowski M, Ivanšek S, Polutnik J (2018) CloudStore – towards scalability, elasticity, and efficiency benchmarking and analysis in cloud computing. Futur Gener Comput Syst 78:115–126. https://doi.org/10.1016/j.future.2017.04.018
Leitner P, Cito J (2016) Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. ACM Trans Internet Technol 16(3). https://doi.org/10.1145/2885497
Maricq A, Duplyakin D, Jimenez I, Maltzahn C, Stutsman R, Ricci R (2018) Taming performance variability. In: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation
Merenstein A, Tarasov V, Anwar A, Bhagwat D, Rupprecht L, Skourtis D, Zadok E (2020) The case for benchmarking control operations in cloud native storage. In: 12th USENIX workshop on hot topics in storage and file systems (HotStorage 20)
Michael M, Moreira JE, Shiloach D, Wisniewski RW (2007) Scale-up x scale-out: A case study using nutch/lucene. In: 2007 IEEE international parallel and distributed processing symposium. https://doi.org/10.1109/IPDPS.2007.370631
Nasiri H, Nasehi S, Goudarzi M (2019) Evaluation of distributed stream processing frameworks for iot applications in smart cities. Journal of Big Data 6(52). https://doi.org/10.1186/s40537-019-0215-2
Papadopoulos AV, Versluis L, Bauer A, Herbst N, Kistowski J, Ali-Eldin A, Abad CL, Amaral JN, Tuma P, Iosup A (2021) Methodological principles for reproducible performance evaluation in cloud computing. IEEE Trans Softw Eng 47(8):1528–1543. https://doi.org/10.1109/TSE.2019.2927908
Ralph P, bin Ali N, Baltes S, Bianculli D, Diaz J, Dittrich Y, Ernst N, Felderer M, Feldt R, Filieri A, de França BBN, Furia CA, Gay G, Gold N, Graziotin D, He P, Hoda R, Juristo N, Kitchenham B, Lenarduzzi V, Martínez J, Melegati J, Mendez D, Menzies T, Molleri J, Pfahl D, Robbes R, Russo D, Saarimäki N, Sarro F, Taibi D, Siegmund J, Spinellis D, Staron M, Stol K, Storey M-A, Taibi D, Tamburri D, Torchiano M, Treude C, Turhan B, Wang X, Vegas S (2021) Empirical standards for software engineering research. arXiv:2010.03525, Version 0.2.0
Sim SE, Easterbrook S, Holt RC (2003) Using benchmarking to advance research: A challenge to software engineering. In: 25th international conference on software engineering. IEEE. https://doi.org/10.1109/icse.2003.1201189
Soldani J, Tamburri DA, Van Den Heuvel W-J (2018) The pains and gains of microservices: A systematic grey literature review. J Syst Softw 146:215–232. https://doi.org/10.1016/j.jss.2018.09.082
Tichy WF (2014) Where's the science in software engineering? ubiquity symposium: The science in computer science. Ubiquity 2014:1–6. https://doi.org/10.1145/2590528.2590529
Tsai W-T, Huang Y, Shao Q (2011) Testing the scalability of SaaS applications. In: 2011 IEEE international conference on service-oriented computing and applications (SOCA). https://doi.org/10.1109/SOCA.2011.6166245
v Kistowski J, Arnold JA, Huppler K, Lange K-D, Henning JL, Cao P (2015) How to build a benchmark. In: Proceedings ACM/SPEC international conference on performance engineering. https://doi.org/10.1145/2668930.2688819
Wang G, Chen L, Dikshit A, Gustafson J, Chen B, Sax MJ, Roesler J, Blee-Goldman S, Cadonna B, Mehta A, Madan V, Rao J (2021) Consistency and completeness: Rethinking distributed stream processing in apache kafka. In: Proceedings of the 2021 International Conference on Management of Data. https://doi.org/10.1145/3448016.3457556
Weber A, Herbst N, Groenda H, Kounev S (2014) Towards a resource elasticity benchmark for cloud environments. In: Proceedings International workshop on hot topics in cloud service scalability. HotTopiCS '14. https://doi.org/10.1145/2649563.2649571

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.