Scalability Evaluation of ExplorViz with the Universal Scalability Law

Master's Thesis

Simon Bela Nicolay Fritz Alexander Ehrenstein

June 10, 2022

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
Sören Henning, M.Sc.
Alexander Krause-Glau, M.Sc.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 10. Juni 2022

Abstract

Modern distributed stream processing systems play an important role in cloud computing systems and Big Data applications. To cope with varying intensity of user load, an important characteristic that often is required for such systems is scalability. The Universal Scalability Law is a performance model to describe the scalability of universal systems. In this work, we examine to what extend the Universal Scalability Law can be integrated with the methodology of the Theodolite benchmarking framework for cloud-native stream processing systems. Theodolite assesses scalability based on service level objectives (SLOs). We find that the Universal Scalability Law can be used to make the execution of benchmarks more efficient with regard to the total execution time and to quantify the scalability based on the benchmark results. However, we find that due to the measurement method used by the Theodolite framework the interpretability of the model may be influenced by the underlying SLOs. We apply our extended version of Theodolite to benchmark the scalability of the software visualization and comprehension framework ExplorViz, which visualizes monitored applications using dynamic analysis. ExplorViz comprises a microservice-based architecture that uses the stream processing framework Kafka Streams. Our results show that some of the ExplorViz microservices scale linearly, but that the system scalability is mainly bounded by one microservice.

Contents

Chapter 1

Introduction

1.1 Motivation

The popularity gains of operating software in the cloud [Chou 2015] have impacted fundamental design choices for building software. Applications that are designed to be executed in cloud environments require the usage of specific technologies or architectures such as containerization or microservices. The specific aspects of such applications are also called cloud-native principles [Gannon et al. 2017]. Since cloud-native systems often consist of multiple, individually executed, and interconnected components, there are various options for the deployment configuration of such systems. However, it is important to choose the right configuration to ensure that all the requirements of the respective system are met. A useful characteristic that allows to identify appropriate configurations is scalability. Intuitively, scalability describes how the computational resources and the load that can be processed are related to each other. One method that can be used to gain more insight about the scalability is benchmarking [Sim et al. 2003]. However, due to the heterogeneous nature of cloud-native applications, benchmarking them is a complex task. Another class of systems that are complex to benchmark, are distributed stream processing systems [Karimov et al. 2018]. The Theodolite benchmarking framework [Henning and Hasselbring 2022; 2021, a; b] aims to simplify the benchmarking process for cloud-native stream processing applications, by providing methodology and tooling for service orchestration, experiment execution, and analysis.

In this work, we extend the methodology and implementation of Theodolite by the Universal Scalability Law (USL) performance model [Gunther 1993]. Our aim is to make the execution of benchmarks with Theodolite more efficient in terms of execution time and the benchmark results more interpretable. The USL allows to detect contention and coherency issues that affect the scalability of a system in form of model parameters that have a physical meaning. As a result, the USL can help software developers to understand why a certain scalability is observed. Moreover, with the USL, the scalability of systems can be predicted and the model parameters can be used to quantify the scalability in scalar values. This also allows to compare the scalability across different systems based on the estimated parameters.

Further, we plan to apply our extended version of Theodolite to benchmarking the scalability of the software visualization and comprehension tool ExplorViz [Fittkau et al.

2017; 2013; Hasselbring et al. 2020] which is based on the stream processing framework Kafka Streams [Wang et al. 2021]. The motivation for this is that the microservices of ExplorViz, to this point, lack an extensive empirical scalability evaluation. Since ExplorViz is designed to support the visualization of multiple applications simultaneously, including the visualization of the interaction of changing amounts of users, ExplorViz is required to be scalable. Consequently, our results provide the base for assessing whether ExplorViz is suitable for its aimed use cases from a scalability perspective.

1.2 Goals

In this section we describe the goals of this work. We plan to address the following aspects:

G1: Application of the Universal Scalability Law to Stream Processing

The USL was originally formulated for scalability analysis in performance engineering of multiprocessor systems. In this work, we plan to leverage the USL to stream processing in order to use it for the scalability analysis of cloud-native stream processing applications with Theodolite.

G2: Extension of Theodolite

Based on our research results concerning the application of the USL to stream processing, we plan to extend Theodolite by the USL in two ways. First, motivated by the research question "How can the scalability metric be measured more efficiently?" from Henning and Hasselbring [2020], we aim to make the benchmark execution more efficient by implementing a heuristic that is based on the USL. Second, we aim to implement a tool that allows analyzing the benchmark results in terms of the USL.

G3: Benchmarking the Scalability of ExplorViz

We plan to design benchmarks for assessing the scalability of the microservices architecture of ExplorViz. Afterwards, we execute the benchmarks with Theodolite. Our analysis includes applying the USL to the benchmark results and we focus on the microservices that are part of ExplorViz' trace analysis.

1.3 Document Structure

The remainder of this work is structured as follows: In Chapter 2, we explain the foundations and technologies this work is based on. This includes an introduction of the term scalability in more detail and a definition of the USL. Next, we discuss how the USL can be applied in distributed stream processing in Chapter 3. Afterwards, we present our implementation in more detail in Chapter 4 and we describe our concrete benchmarks of ExplorViz in Chapter 5. In Chapter 6, we evaluate our extensions of Theodolite and the results of benchmarking ExplorViz. In Chapter 7, we describe research that is related to this work, before we finally summarize the conclusions of this work in Chapter 8.

Chapter 2

Foundations and Technologies

2.1 Different Notions of Scalability

When analyzing system scalability, it is important to have a precise definition of scalability. In this section we discuss different aspects of scalability and we explain the formal definition of the scalability of cloud applications this work is based on.

2.1.1 Scalability in Cloud Computing

A formal definition of scalability is not trivial and depends on the context. Henning and Hasselbring [2022] define that a ″system is 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″. This definition is based on the definition of Herbst et al. [2013], who define scalability of cloud systems as "[..] the ability of a system to sustain increasing workloads with adequate performance provided that hardware resources are added″ and on the following three attributes of scalability described by Weber et al. [2014]:

▷ **Load Intensity** describes the amount of work a system has to handle.

▷ **Provisioned Resources** describe the minimum amount of computational resources that are required to process a certain load intensity.

▷ **Service Level Objects (SLOs)** define minimal service levels that should be met by the system under test.

Henning and Hasselbring [2022] state that scalability can also be defined from a capacity perspective. This means that scalability can be seen as a ″system's ability to increase its capacity by consuming more resources″ while certain ″quality criteria″ are met. According to the authors, these quality criteria correspond to the SLOs from Weber et al. [2014].

2.1.2 Resource Scaling

In the previous section, we discussed the meaning of the term scalability in the context of cloud applications. The mentioned definitions of scalability have in common that we can add resources to the system that is observed. However, the definitions do not imply how this can be done. To add resources there are two ways: vertical scaling and horizontal scaling [Michael et al. 2007].

Vertical Scaling

Vertical scaling refers to adding resources to individual computation nodes. This can, for example, be achieved by adding CPUs, CPU cores, RAM, or more powerful hardware. Vertical scaling is usually easy to implement, since it often does not require a specific system architecture. In most cases, it is sufficient to just redeploy the application on a server with adapted hardware. However, when adding more CPUs or CPU cores, the application needs to support parallel execution. In addition, vertical scaling cannot be done indefinitely as running high-end hardware is often not cost-efficient and there exist physical bounds.

Horizontal Scaling

Horizontal scaling refers to adding computing nodes to a distributed system. In contrast to vertical scaling, horizontal scaling can theoretically be done indefinitely and is usually much cheaper, since no high-end hardware is required. One of the downsides of horizontal scaling is that it cannot be done with arbitrary systems. More precisely, it requires specific system architectures that allow to run multiple instances of the same application. Moreover, having more instances may lead to consistency or availability issues [Gilbert and Lynch 2002].

2.1.3 Scalability Metrics

Based on the three scalability attributes from Section 2.1.1, Henning and Hasselbring [2022] define two scalability metrics. Before we can introduce these metrics, we present the following formal definition of SLOs:

Definition 2.1.1 (Service Level Objectives (SLOs) [Henning and Hasselbring 2022]). *Given a load type and a resource type, let L be a discrete set of load values and R be a discrete set of resource values. Then the set of SLOs is defined as*

$$S := \{s \mid s : L \times R \to \{true, false\}\}$$

It is assumed that there exists an ordering on the values of L and R. Usually, L and R contain integer values where, for example, L contains integers that represent the number of users of a system and R contains integers that represent the number of instances of an application. For convenience, given an SLO $s \in S$ we also write slo_s instead of s. For better notation, we also define an evaluation function for sets of SLOs:

Definition 2.1.2 (SLO Evaluation Function). *For a set of SLOs $S' \subset S$ we define the set-based SLO evaluation function $SLO_{S'} : L \times R \rightarrow \{0, 1\}$ for $l \in L$, $r \in R$ as follows:*

$$SLO_{S'}(l, r) = \begin{cases} 1 & \forall s \in S' : slo_s(l, r) = true \\ 0 & otherwise \end{cases}$$

We say that all SLOs in S' are fulfilled, if and only if $SLO_{S'}(l, r) = 1$.

Given these formalizations of SLOs, we are able to define the following scalability metrics for a given set $S' \subset S$ of SLOs:

Definition 2.1.3 (Resource Demand Metric [Henning and Hasselbring 2022]). *Given a set of SLOs $S' \subset S$, the Resource Demand Metric is a function $demand : L \rightarrow R$ which determines the minimum amount of resources of R that is required to process a given load of L while all SLOs S' are fulfilled. Let $l \in L$, then we define:*

$$demand(l) := min\{r \in R \mid SLO_{S'}(l, r) = 1\}$$

Definition 2.1.4 (Load Capacity Metric [Henning and Hasselbring 2022]). *Given a set of SLOs $S' \subset S$, the Load Capacity Metric is a function $capacity(r) : R \rightarrow L$ which, for some resource value, determines the maximum load intensity from a given set of load values L for which all SLOs S' are fulfilled. Let $r \in R$, then we define:*

$$capacity(r) := max\{l \in L \mid SLO_{S'}(l, r) = 1\}$$

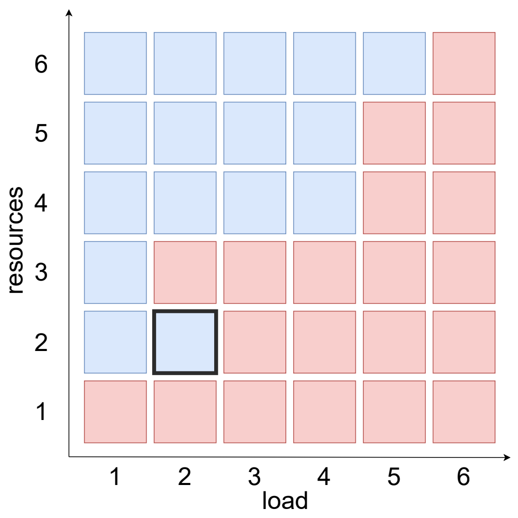
One can see that both metrics can be used to characterize scalability from different viewpoints. However, it is important to note that both metrics are only well-defined if, for the respective metric, the minimum or the maximum exists. In practice this depends on the chosen SLOs and appropriate ranges of load and resource values. Presuming that these attributes are always chosen carefully, we assume that the metrics are always well-defined.

We notice that the behavior of the scalability metrics is determined by the used SLOs. As mentioned, the SLOs should be defined carefully when applying the scalability metrics. However, the definitions of the scalability metrics do not state any additional constraints that are required to make an SLO suitable for applying the metrics. Some SLOs have properties that allow us to determine the scalability metrics more efficiently, though. One such useful property is the monotony of the SLO evaluation function, which may behave monotonously increasing in the resource dimension or monotonously decreasing in load dimension, which we describe in the following definition:

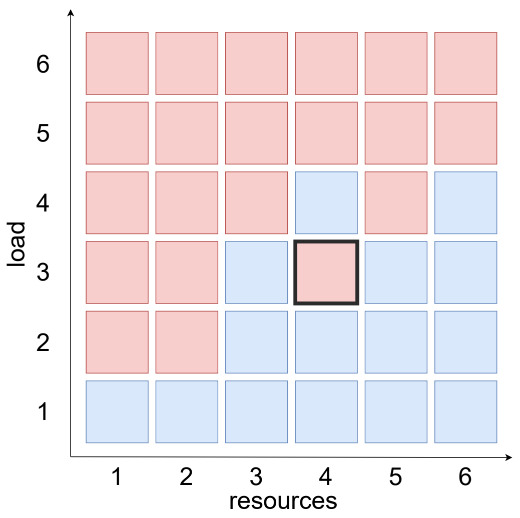
Definition 2.1.5 (Monotony of the SLO Evaluation Function). *Given a set of SLOs $S' \subset S$, we define the following:*

(1) The set-based SLO evaluation function $SLO_{S'}$ is monotonously increasing in the resource dimension if we have:

$$\forall l \in L, r_1, r_2 \in R : r_1 \leqslant r_2 \Rightarrow SLO_{S'}(l, r_1) \leqslant SLO_{S'}(l, r_2)$$



(a) Visualization of an SLO evaluation function that is not monotonously increasing in the resource dimension.



(b) Visualization of an SLO evaluation function that is not monotonously decreasing in the load dimension.

Figure 2.1. Visualization of the SLO evaluation function $SLO_{S'}$ for a set of SLOs S'. A red square indicates that we have $SLO_{S'}(l,r) = 0$ for the corresponding load value l and resource value r, meaning not all SLOs from S' are fulfilled. A blue square indicates that $SLO_{S'}(l,r) = 1$, meaning all SLOs from S' are fulfilled. In example (a), the SLO evaluation function is not monotonously increasing in the resource dimension, since we have $SLO_{S'}(2,2) = 1$ (black outlined square) but $SLO_{S'}(2,3) = 0$. However, if we had $SLO_{S'}(2,2) = 0$, the evaluation function would be monotonously increasing in the resource dimension. In example (b) the SLO evaluation function is not monotonously decreasing in the load dimension, since we have $SLO_{S'}(3,4) = 0$ (black outlined square) but $SLO_{S'}(4,4) = 1$. Though, if we had $SLO_{S'}(3,4) = 1$, the evaluation function would be monotonously decreasing in the load dimension

(2) The set-based SLO evaluation function $SLO_{S'}$ is monotonously decreasing in the load dimension if we have:

$$\forall r \in R, l_1, l_2 \in L : l_1 \leqslant l_2 \Rightarrow SLO_{S'}(l_1, r) \geqslant SLO_{S'}(l_2, r)$$

In Definition 2.1.5 (1), we define what is meant by the SLO evaluation function being monotonously increasing in the resource dimension. The situation is visualized in Figure 2.1a. The depicted SLO evaluation function is not monotonously increasing in the resource dimension since we have $SLO_{S'}(2,2) = 1$ and $SLO_{S'}(2,3) = 0$. If we had $SLO_{S'}(2,2) = 0$, it would be monotonously increasing in the resource dimension. Accordingly, Definition 2.1.5 (2) states what is meant by an SLO that is monotonously decreasing in the load dimension. Figure 2.1b shows a situation, where the SLO evaluation function is not monotonously decreasing in the load dimension, since $SLO_{S'}(3,4) = 0$ and $SLO_{S'}(4,4) = 1$.

However, if we had $SLO_{S'}(3,4) = 1$, it would be monotonously decreasing in the load dimension.

Depending of the system requirements, the SLO evaluation function may not always be monotonously increasing or decreasing in the load or resource dimension. However, meaningful SLOs should have this property in order to allow a meaningful interpretation of the scalability metrics. In addition, from a perspective of scalability benchmarking, a monotonous SLO evaluation function is advantageous, since the SLO evaluation changes only at one point from zero to one or vice versa. As a result, paying attention to the properties of the SLOs can allow to use more efficient algorithms to compute the scalability metrics during the benchmark execution.

2.2 The Universal Scalability Law

The Universal Scalability Law (USL) is a performance model for describing scalability. It was introduced by Gunther [1993, 2010] and applied to various systems [Schwartz 2017; Schwartz and Fortune 2010; Gunther et al. 2015]. One central characteristic is that the USL can be used to asses the scalability of universal systems. In this section, we will introduce the USL in more detail. For this, we first define the scaleup capacity:

Definition 2.2.1 (Scaleup Capacity [Gunther 2010]). *Given the throughput $X(N)$ of a multi-processor with N processes, we define the scaleup capacity $C(N)$ as*

$$C(N) = \frac{X(N)}{X(1)}$$

The scaleup capacity is equivalent to the normalized throughput.

Based on the scaleup capacity, Gunther [2010] defines the Universal Scalability Law, which describes scalability as the following non-linear rational function:

Definition 2.2.2 (Universal Scalability Law [Gunther 2010]).

$$C(N) = \frac{N}{1 + \alpha(N-1) + \beta N(N-1)}$$

The three terms in the denominator of the equation can be interpreted as follows:

1. **Concurrency**. The first term in the denominator describes the concurrency. Assuming, $\alpha = \beta = 0$, we have $C(N) = N$. Recalling the definition of scaleup capacity, this means that each additional processor is 100% efficient. This would imply linear scalability.

2. **Contention**. The second term in the denominator describes the contention. It captures the fraction of the execution time that reflects the waiting for shared resources. A typical example where contention can occur is database locking. The parameter α captures the degree of contention. If we have $\alpha = 0$, we have no contention, as the second term in

the denominator becomes zero. However, if we have $\alpha = 1$, we have a scaleup capacity of $C(N) = 1$. This means that regardless of the number of processors N, the scaleup capacity would be constant. Practically, this means that we have no scalability at all.

3. **Coherency**. The last term in the denominator describes the negative effects that arise from pairwise synchronization between the N processes. The larger β is, the more intense is the multiprocessor overhead from inter-process exchange. One example where coherency can occur is when some cache entry is invalidated and the updated value must be retrieved from other processes.

The definition of the USL describes the two model parameters α and β that capture the contention and the coherency of the examined system. As a consequence, the process for applying the USL is as follows [Gunther 2010]:

1. Compute $X(N)$ for each amount of processors N we are interested in, including for $N = 1$. This can be achieved by conducting experiments for the desired values of N.
2. Calculate $C(N)$, given the values for $X(N)$ and $X(1)$.
3. Estimate the model parameters α and β of the equation from Definition 2.2.2 using nonlinear statistical regression [Ritz and Streibig 2008].

We should note that for estimating both model parameters, the left side of the equation $C(N)$ must be known. Here, a practical problem arises, since the scaleup capacity depends on the throughput $X(1)$ of one processor, but this value is not always known. Gunther [2018] solves this by substituting the term $C(N)$ in the USL formula with its definition:

$$\begin{aligned} C(N) &= \frac{N}{1+\alpha(N-1)+\beta N(N-1)} \\ \Longleftrightarrow \frac{X(N)}{X(1)} &= \frac{N}{1+\alpha(N-1)+\beta N(N-1)} \\ \Longleftrightarrow X(N) &= \frac{X(1)N}{1+\alpha(N-1)+\beta N(N-1)} \end{aligned}$$

Based on the last equation, Gunther [2018] introduces an additional parameter $\gamma = X(1)$. As a result, the three parameter version of the USL can be defined as follows:

Definition 2.2.3 (Universal Scalability Law (Three Parameter Version) [Gunther 2018]). *Let $X(N)$ be the throughput for N processors. Then, the model parameters α, β, and γ characterize the scalability of a system based on the following equation:*

$$X(N) = \frac{\gamma N}{1+\alpha(N-1)+\beta N(N-1)}$$

If we look at Definition 2.2.3 we can see that the denominator is the same as in the original definition of the USL. The new model parameter γ occurs in the numerator and it estimates the value of the throughput of a single processor configuration of the system $X(1)$. Therefore, to apply the USL, it is now sufficient to identify the throughput values $X(N)$ for all the values of N one is interested in. Afterwards, we can directly use nonlinear regression in order to find the model parameters α, β, and γ. Gunther [2008] has shown that in case of $\gamma = 1$ and $\beta = 0$, the USL can be reduced to Amdahl's law [Amdahl 1967]. Amdahl's law states that the sequential parts of a program are the limiting factor with an increasing degree of parallelization. This gives us an asymptotic bound for the capacity of the examined system which only depends on the value of α and γ.

Definition 2.2.4 (Amdahl's Asymptote [Gunther 2010]). *For $\beta = 0$, we define Amdahl's asymptote as the following limit for the two parameter version of the USL*

$$\lim_{N\to\infty} \frac{N}{1+\alpha(N-1)+\beta N(N-1)} = \alpha^{-1}$$

and as the following limit for the three parameter version of the USL

$$\lim_{N\to\infty} \frac{\gamma N}{1+\alpha(N-1)+\beta N(N-1)} = \alpha^{-\gamma}$$

To this point, we have simply called the variable N the number of processors. However, as Holtman and Gunther [2008] state, there are two notions of scalability that result in a different interpretation of the independent variable N:

Hardware Scalability means that the hardware configuration of a system is scaled up or down. In this case, N can either be the amount of CPUs that is scaled up or down in one server, or the number of servers that is scaled up or down within a cluster. To avoid distortion in the experiments when measuring the capacity, every CPU in a server or each server in a cluster must provide the same degree of concurrency to the system.

Software Scalability means that the hardware configuration of a system stays fixed during the execution of experiments, while the amount of concurrent virtual users of the system is scaled up or down. In this case, N describes the amount of concurrent virtual users. A typical example for this would be a web shop that runs on a server with fixed resources while the amount of users that interact with the shop increases or decreases.

Figure 2.2 schematically shows scalability graphs for different values of the USL model parameters. It depicts how much throughput can be achieved with a certain amount of processors. When $\alpha = \beta = 0$ then the parameter γ corresponds to the rate of linear scalability (black line). This usually is what people want to achieve, as it means that for increased load, a proportional increase in the computational resources is sufficient, or inversely, adding resources proportionally increases the throughput. When $\alpha > 0$ and $\beta = 0$ (blue curve), the scalability is sublinear and bounded by Amdahl's asymptote by the value γ/α due to diminishing returns from contention. When $\alpha > 0$ and $\beta > 0$ (green curve), the system throughput first increases sublinear due to contention and at some point degrades again, because of negative returns from coherency.

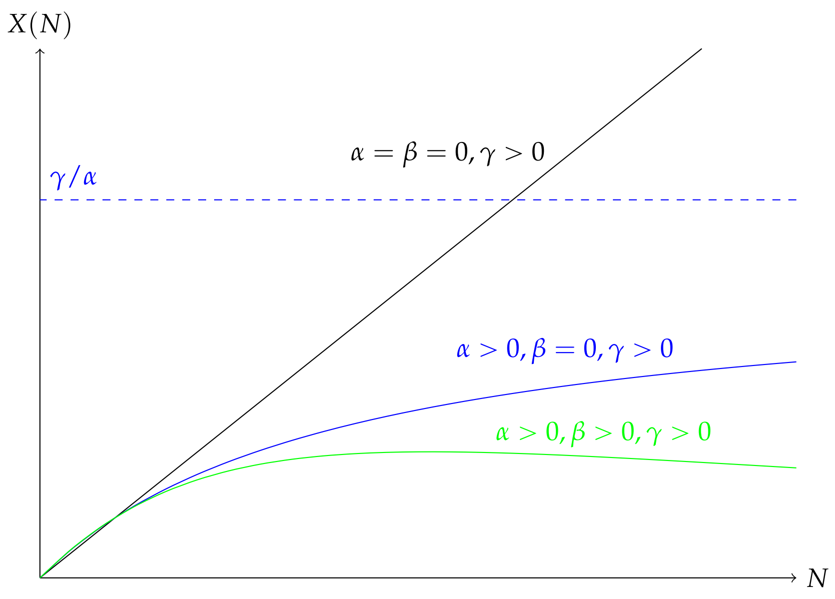


Figure 2.2. Visualization of scalability graphs with the USL and different values of the model parameters α, β, and γ.

2.3 The Distributed Stream Processing Framework Kafka Streams

Apache Kafka [Kreps et al. 2011] is an event streaming platform. Kafka provides a consumer and a producer API which allows to write and read key-value pairs, called messages, into and from channels. These channels are called topics. Each topic is subdivided into one or or multiple partitions which each consist of an ordered log of messages. These partitions are replicated across multiple Kafka brokers for fault tolerance.

Kafka Streams [Wang et al. 2021] is a framework based on Kafka which allows the distributed processing of continuous data streams [Hummer et al. 2013]. The stream processing logic in Kafka Streams is defined in a processing topology. It describes which stream processing operations are used and how they are combined. Each application that takes part in the stream processing may execute one or multiple topologies. According to the topology, Kafka Streams creates a set of tasks which the topology is subdivided into and that can be executed in parallel. The number of tasks is bounded by number of partitions of the Kafka topics that the topology is based on. According to the stream-table duality [Sax et al. 2018], the streaming model of Kafka Streams is based on streams and tables: streams represent the history of send messages, whereas tables represent the state of streams at a current time. Accordingly, Kafka Streams supports stateless operations, for example, transforming individual messages from a stream and writing it to another, and stateful operations based on tables.

2.4 The Container Orchestration Platform Kubernetes

Kubernetes is a container orchestration platform that allows the execution of arbitrary applications. The project is developed as part of the Cloud Native Computing Foundation (CNCF) [Cloud Native Computing Foundation 2018]. Kubernetes supports multiple container runtimes including Docker Engine [Merkel 2014], the CRI-O, and containerd runtimes that are also part of the CNCF [Batts 2019; Cloud Native Computing Foundation 2019].

The Kubernetes environment consists of a cluster of multiple nodes that run so-called *Pods*. Each Pod provides a logical environment to execute one or multiple *Containers* that need to be deployed together. Typically, each application, consisting of one or multiple containers, is deployed in one Pod. However, Kubernetes also provides a network layer allowing applications running in different Pods to communicate with each other. There are multiple API objects that can be used to define Pods and containers. These objects have different semantics. The most common are *Deployments* and *StatefulSets* which are used to define stateless and stateful applications respectively. A common method to deploy Kubernetes API objects is to define so-called Kubernetes manifest files. These are plain text files that contain the definition of API objects in the YAML format. Kubernetes API objects can be retrieved and manipulated through API endpoints, so-called resources. For example, there are endpoints to manage Pods, Deployments, and StatefulSets.

Moreover, it is possible to extend the Kubernetes API by custom endpoints called *CustomResources* that can be used to define the state of applications in the Kubernetes cluster. These custom resources can be used to implement Kubernetes Operators. An Operator is an application that is executed in Kubernetes itself and uses the information provided in custom resources to manage the state of applications in Kubernetes [Jarvinen 2019]. Kubernetes manifest files can be shared via the *Helm* package manager for Kubernetes which is also part of the CNCF [Butcher 2020]. Helm bundles multiple manifest files in a so-called *Chart* that can be shared with others.

2.5 The Theodolite Framework for Benchmarking Stream Processing Engines

Theodolite [Henning and Hasselbring 2021b] is a framework for benchmarking stream processing applications with a focus on scalability analysis in cloud environments. In the following, we discuss the methodology and the tools provided by the framework.

2.5.1 Theodolite Methodology

The Framework encompasses methodology from Henning and Hasselbring [2022, 2021a]. It relies on the scalability definitions that we discussed in Section 2.1. As stated, these definitions of scalability are based on the *Resource Demand Metric* and the *Load Capacity Metric*. According to the underlying definition of SLOs, Theodolite supports arbitrary load and resource dimensions. Moreover, the Theodolite methodology relies on two steps to make the benchmarking process more efficient:

1. **Discretization**
   In general, the load and resource values can be either continuous or discrete. However, during the benchmarking process, the amount of executed experiments is determined by the amount of different load and resource values that should be examined. This is due to the fact that for each combination of load and resource value, it has to be potentially assessed whether the system is able to process the load or not. Therefore, Theodolite expects that the load and resource dimensions are sets of integers. Consequently, continuous values must be discretized for the benchmarking with Theodolite. Moreover, a meaningful resolution of the load and resource values must be defined, in order to get accurate benchmark results. However, reducing the number of combination of load and resource values and therefore the duration of the benchmark execution comes at the cost of decreased accuracy.

2. **Heuristic Execution**
   After the load and resource values are discretized into a meaningful resolution, Theodolite uses heuristic algorithms in order to approximate the chosen scalability metric [Henning and Hasselbring 2020; 2022]. The heuristics are also called search strategies. During the execution of a search strategy, the combinations of load and resource values are examined in a certain order depending on the strategy used. For each combination of load and resource value, a so-called SLO experiment is executed and it we determine, whether all defined SLOs are fulfilled or not. Depending on the strategy, the results of previous SLO experiments are considered for choosing the next SLO experiment to execute. After the execution of the search strategy, an approximation of the used scalability metric is determined.

Table 2.1 gives an overview of the most central search strategies that are currently implemented by Theodolite.

First, we have the *FullSearch* search strategy. This strategy examines all combinations of load and resource values in individual SLO experiments. Since this gives full information about all possible experiments, it does not make any assumptions with respect to the monotony of the SLO evaluation functions or the scalability metric in order to approximate the scalability metric.

Second, we have the *LinearSearch* search strategy. The concrete behavior of this search strategy depends on the used scalability metric. When the *Resource Demand Metric* is used,

Table 2.1. Overview of the search strategies currently supported by Theodolite and their properties with regard to the set-based SLO evaluation function $SLO_{S'}$ and the used scalability metric. The function may be monotonously increasing in the resource dimension, or monotonously decreasing in the load dimension, or not monotonous at all. The scalability metric may be monotonously increasing.

	Assumptions	
Search Strategy	$SLO_{S'}$ monotonously increasing/decreasing	Scalability Metric monotonously increasing
FullSearch		
LinearSearch	✗	
BinarySearch	✗	
RestrictionSearch with		
FullSearch + LowerBound		✗
LinearSearch + LowerBound	✗	✗
BinarySearch + LowerBound	✗	✗

this strategy examines the resource values in increasing order for each load value. For each load value, the search stops when the first SLO experiment succeeds. The corresponding resource value is considered to be the approximated demand. Therefore, it assumes the SLO evaluation function to be monotonously increasing in the resource dimension for the given set of SLOs. In the case of the *Load Capacity Metric*, the *LinearSearch* search strategy examines the load values in increasing order for all resources values. For each resource value, it executes SLO experiments for increasing load values until the first SLO experiment does not succeed. Consequently, the resource value of the last successful SLO experiment is considered to be the approximated capacity. Therefore, it is assumed that the SLO evaluation function is monotonously decreasing in the load dimension for the given set of SLOs. Moreover, there is the *BinarySearch* search strategy. In contrast to the *LinearSearch* search strategy, it uses binary search to approximate the used scalability metric but it makes the same assumptions with regard to the monotony of the SLO evaluation function.

There is also the *RestrictionSearch* search strategy. This strategy uses either the *FullSearch*, *LinearSearch*, or *BinarySearch* search strategy in combination with a so-called restriction strategy. The restriction strategy is used to restrict the number of possibly executed SLO experiment for each load or resource value, depending on the scalability metric. In the current version of Theodolite, only the *LowerBound* restriction strategy exists, which in the case of the *Resource Demand Metric* restricts the set of resource values that are examined for the next load value to all the resource values greater than or equal to the demand that has been identified for the next lower load value. In the case of the *Load Capacity Metric*, the *LowerBound* restriction restricts the set of load values that are examined for the next resource value to the load values greater than or equal to the capacity that has been determined for the next lower resource value. Therefore, the *LowerBound* restriction strategy assumes the used scalability metric to be monotonously increasing, additionally to the assumptions of the used search strategy. Table 2.1 summarizes the information regarding the assumptions that the existing search strategies require in order to allow an accurate approximation of the used scalability metric.

In Section 2.1.1, we introduced SLOs. At the current stage, Theodolite supports the following SLOs for assessing the scalability of stream processing applications.

▷ **Lag Trend** is based on the *lag trend metric* [Henning and Hasselbring 2021a]. It measures the backpressure [Karimov et al. 2018] during the stream processing taking into account the amount of messages piling up within the messaging system. Based on these measurements, it uses linear regression to detect a trend in the amount of piled up messages across the runtime of an SLO experiment. The SLO is considered fulfilled as long as the lag trend, which is defined as the gradient of the linear regression model, does not exceed a certain threshold.

▷ **Dropped Records** counts the amount of messages that are dropped during an SLO experiment. This can happen if some messages lack certain properties required by the application. For example, a message may be dropped if it arrives too late at a processor and is therefore ignored. The dropped records SLO is fulfilled as long as the total amount of dropped records during an SLO experiment does not exceed a defined threshold.

Both SLOs have in common that they are based on a threshold that has to be specified for the respective benchmark. Hence, the main challenge for the user is to choose this threshold in a meaningful manner. However, when different load intensities, for example, varying from just a few hundreds to an intensity of hundreds of thousands, are investigated within a benchmark, choosing an absolute threshold may not be meaningful for all SLO experiments. The reason for this is that the values of the metric that the SLO relies on may be proportionally volatile to the load intensity. For example, the rate of records piling up within the messaging system might exceed a threshold of 2 000 when examining a load of multiple hundreds of thousand messages per second for a short period of time. However, this might not be related to a lack of the system under test to sustain the load, but to the volatility of the metric. Therefore, the lag trend SLO and the dropped records SLO are also provided as variants that take a ratio as parameter instead of a threshold [Vonheiden 2021]. According to this ratio, the SLO is evaluated with a threshold that is proportional to the load value examined in the respective SLO experiment.

2.5.2 Theodolite Benchmarking Tool

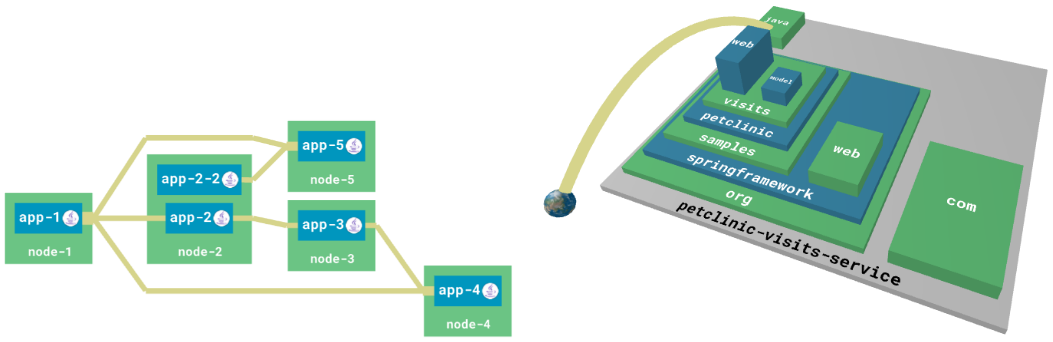
The Theodolite framework is specifically designed for benchmarking real world applications that use stream processing engines. Theodolite follows a cloud-native approach and can be installed in Kubernetes via the Kubernetes package manager Helm. Currently, tooling for benchmarking applications that use Kafka Streams or Apache Flink [Carbone et al. 2015] is provided. In addition, there is a set of pre-configured example benchmarks provided. The central component of the framework is the *Benchmarking Tool* which provides a Kubernetes Operator (see Section 2.4) in order to control benchmark executions [Henning et al. 2021]. Theodolite further provides an *Analysis Tools* component that allows evaluating the results of the benchmarks in the form of Jupyter Notebooks [Kluyver et al. 2016]. In addition, the framework provides a Java library for implementing load generators for Kafka. Since Theodolite provides a Kubernetes Operator, executing benchmarks requires the user to create Kubernetes custom resources. There are two custom resources that need to be created. First, all structural parameters for a benchmark have to be defined in the `Benchmark` custom resource, for example, the definition of the system under test and of the load generators. Second, a custom resource of the kind `Execution` has to be created. This custom resource refers to exactly one benchmark and contains all pieces of information for an individual execution of the benchmark. Theodolite uses the time series database *Prometheus* [Rabenstein and Volz 2015; Linux Foundation 2022] to collect metrics, which are mainly gathered via the *Java Management Extensions* (JMX) [Oracle 2022] from the Kafka brokers and the monitored stream processing applications.

For setting up a benchmark, it may be required to perform initialization of certain applications that are part of the benchmark infrastructure or the system under test. For example, it may be required to initialize a schema for a database. This can be achieved by defining so-called *Actions* in Theodolite's custom resources. Actions allow to execute shell commands within containers either before an SLO experiment is started, or after it has finished.

2.6 The Software Visualization and Comprehension Tool ExplorViz

ExplorViz [Fittkau et al. 2017; 2013; Hasselbring et al. 2020] is a software visualization and comprehension tool for complex applications. It uses dynamic analysis [Ernst 2003] in order to build a visual representation of the monitored applications. More specifically, ExplorViz uses tracing [Cornelissen et al. 2009] to collect information about control-flow of a program. During the trace analysis, so-called traces consisting of chains of method calls are collected to gain insight of into the behavior of a program during its execution. Tracing in ExplorViz is based on the OpenCensus library [OpenCensus Developers 2022], which uses a data structure called *Span*[1] to represent each method call. The visualization in ExplorViz follows the software city metaphor [Wettel and Lanza 2007], which aims at representing software in city-like structures that the user can navigate through. The goal of this approach is to reach a better comprehension of the software. Accordingly, two layers of visualizations are provided. From a high-level perspective, monitored applications are visualized together in a 2-dimensional landscape similar to UML diagrams. The landscape view is depicted in Figure 2.3a. In addition, there is a dedicated visualization

[1]https://opencensus.io/tracing/span/



(a) The landscape view of ExplorViz **(b)** The application view of ExplorViz

Figure 2.3. The visualization of a monitored application in ExplorViz [Krause-Glau 2022].

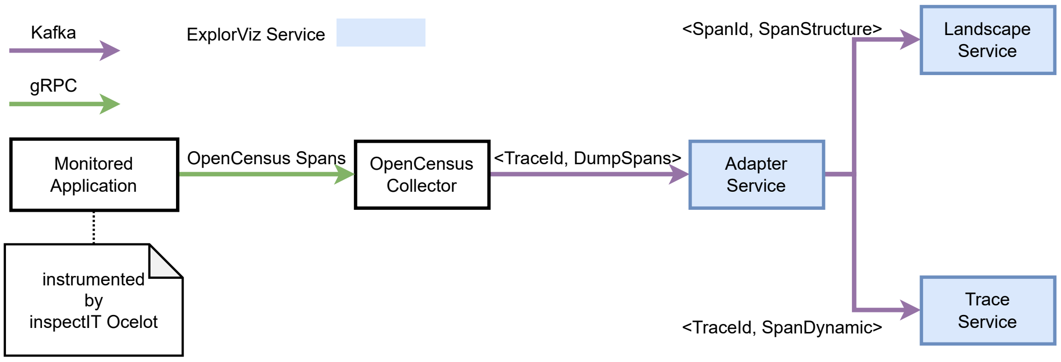


Figure 2.4. The microservices architecture of the ExplorViz trace analysis based on Krause et al. [2021].

per application that contains more details. Figure 2.3b shows the application level view, which is a 3-dimensional representation of the application consisting of a dynamic trace visualization that is embedded into the static hierarchial structure of packages and classes of the monitored application.

Figure 2.4 shows the system architecture of ExplorViz's trace analysis [Krause et al. 2021]. The trace analysis consists of three microservices: The *Adapter Service*, *Landscape Service*, and *Trace Service*. In the following, we will explain the process of the trace analysis in ExplorViz in more detail: Regarding the instrumentation and monitoring of a software system, ExplorViz uses Novatec's inspectIT Ocelot[2] to collect traces. Ocelot is attached as Java Agent to the monitored application and instruments it via byte code weaving based on the Java Instrumentation API. According to the instrumentation definition that

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

is configured at the Java Agent, Ocelot collects traces and breaks them down into spans. Each span corresponds to one method call in a trace. The spans are forwarded via gRPC to the OpenCensus Collector that forwards them to the ExplorViz *Adapter Service* via Kafka. At the *Adapter Service*, the spans are split into their structural data and into their dynamic data. The structural data contains static information of the span, for example, the name or the package that the associated method is contained in. The dynamic data contains the information of the span that are directly related to the execution the associated method, for example, the start and end timestamps of the execution. The structural span data is forwarded to the *Landscape Service* where it is persisted. When requested by the ExplorViz Frontend via HTTP, the structural data are integrated into a hierarchial representation of the monitored system and returned for visualization. The dynamic span data are forwarded to the *Trace Service* where the traces are reconstructed and persisted in order to be visualized in the frontend.

Chapter 3

Applying the Universal Scalability Law to Stream Processing Applications

In this chapter, we compare the underlying ideas of the scalability definition for stream processing applications from Henning and Hasselbring [2022] with the definition of scalability in terms of the USL. Based on this, we describe how the USL can be integrated to the methodology of Theodolite.

3.1 The Universal Scalability Law in the Context of Stream Processing

Originally, the USL was formulated to describe the scalability of multiprocessor systems. There are also examples of the application of the USL in distributed contexts [Gunther et al. 2015; Gunther 2019] or in cloud computing [Chen et al. 2022]. However, to the best of our knowledge, the USL has not been applied extensively in the context of distributed stream processing. Motivated by **G1**, we want to discuss how the USL can be applied to stream processing applications in cloud environments and analyze how the USL can be integrated into Theodolite's methodology. With this aim, we discuss in more detail how scalability is perceived from the perspective of the USL and in benchmarking with Theodolite. First, we discuss the assumptions that are prerequisites for applying the USL and we explain how we can ensure them for stream processing use cases. Afterwards, we discuss how the scalability metrics used by Theodolite can be utilized properly to apply the USL and what consequences arise with respect to the load and resource dimensions.

3.1.1 Assumptions for Applying the Universal Scalability Law

Gunther [2010] states that in order to apply the USL, the system under test must be optimized for concurrency. This means all CPU cycles of all processors of the system under test must be fully utilized. This view is inherent to the definition of throughput that the USL relies on. Gunther [2010] defines the throughput X_N for N processors as

$$X_N = \frac{C_N}{T_N}$$

where C_N is the number of executed transactions during the execution time of T_N for a system with N processors. Assuming that the computational resources are always fully utilized leads to the observation that each processor either performs useful work or does things, such as waiting for access to shared resources, or for network I/O or process synchronization to complete. As a result, a processor not performing useful work experiences contention or coherency.

3.1.2 Comparison of Capacity Metrics with Demand Metrics

In Section 2.1, we introduced the scalability metrics for cloud computing, that Theodolite is based on. That is, we described the demand and the capacity view of scalability with the according *Resource Demand Metric* and the *Load Capacity Metric*. Since the USL is entirely based on throughput, which is a capacity metric, we conclude that only the *Load Capacity Metric* could potentially be described with the USL.

3.1.3 Scalability Metrics for Stream Processing

Comparison of Throughput with the Load Capacity Metric

One possible approach to apply the USL to stream processing application is to use the throughput as scalability metric. This approach was demonstrated by Vikash et al. [2020], who examine different distributed stream processing systems in terms of the USL. However, considering only the throughput that can be achieved by a system may not lead to a meaningful characterization of scalability in all scenarios, since there may be additional requirements for the examined system. In contrast, the methodology from Theodolite provides more flexibility since the used *Load Capacity Metric* depends on SLOs that have to be defined with respect to the concrete benchmarking scenario and to the requirements for the examined system.

Relaxing the Assumptions of the Universal Scalability Law

As stated, the USL makes the assumption that the system under test is optimized for concurrency, i.e., that a sufficient amount of load is generated for the system meaning that all CPU cycles are fully utilized. However, the discrete measurement approach from Theodolite can result in this assumption being violated. The reason for this is that the utilization of the system under test depends on the applied load intensity in the SLO experiments. However, there can be SLO experiments where not all computational resources are fully utilized, but the SLOs may be violated. As a result, the assumptions from the USL are not meaningful to the measurement method of Theodolite. However, the SLOs can be interpreted as additional constraints to the throughput: If the SLOs are violated before the system is fully utilized, the unused computational resources can be considered to experience contention or coherency since applying enough load to utilize them would result in a violation of the SLOs. Summarizing, we can conclude that applying the USL in combination with the discrete measurement method from Theodolite can be achieved by dropping the assumption of full utilization of all CPU cycles in the system under test. Yet, this weakens the physical meaning of the USL model parameters since the parameters also capture properties of the SLOs that may be not related to the physical properties of the system under test.

Limits of Our Approach

The USL assumes that the scalability only depends on contention and coherency, i.e., the model parameters α and β. The value of γ only defines the scaling of the USL function on the vertical axis. However, there may be SLOs that behave in a way that result in values of the *Load Capacity Metric* that can not be described by the rational function of the USL. That is, the SLOs neither depend on contention nor coherency effects, but may be dependent on other variables. We note that this restriction is not related to the USL not being applicable to the system under test but to the incompatibility of our measurement method with the assumptions of the USL.

Conclusions

Based on the findings of the previous sections, we identify the following options to apply the USL to stream processing:

The first option is to use the throughput to measure scalability. As a consequence, we can apply the USL by means of its original interpretation, but we do not take any SLOs into account when assessing the scalability of our application.

The second option is to use the *Load Capacity Metric* from Theodolite, which is based on SLOs. However, it cannot be guaranteed that the USL fits the values of the metric, since it may depend on other variables than contention of coherency. Moreover, even if the model fits, the identified model parameters may have a different physical meaning since they originated from the behavior of the SLO.

In the remainder of this work, we will consider the second option, which combines the methodology of Theodolite with the USL.

3.1.4 Suitability of Theodolite's SLOs for Applying the Universal Scalability Law

In the previous sections, we concluded that applying the USL restricts us to use *Load Capacity Metric* in Theodolite. However, the concrete values of this metric depend on the used SLOs. While Theodolite supports arbitrary SLOs, there are two SLOs that are already implemented by the framework. In the following, we will take a closer look on what the usage of these SLOs implies for an application of the USL.

Lag Trend SLO

For stream processing, one meaningful SLO is the *lag trend* SLO, that we already described in Section 2.5. This SLO is fulfilled if the number of piled up messages does not increase with a rate greater than a specified threshold. The growth rate is determined with linear regression and it is also called lag trend. In an ideal scenario, the lag trend should exactly be zero if the system under test is able to sustain the load. However, it makes sense to allow a certain threshold to make the SLO less sensitive to measurement errors and outliers [Henning and Hasselbring 2021a]. Consequently, we can assume for now that if the threshold is chosen carefully, we can neglect it. Thus, we consider a *lag trend* SLO as being not fulfilled if the lag trend is greater than zero. Let us now consider an SLO experiment where the throughput of the system under test is X. We make the following observation.

Observation 1 . *The lag trend SLO is fulfilled, if and only if $l \leqslant X$.*

This is easy to recognize since as long as $l < X$, the system is not fully saturated. As a result, there are no messages piling up during the stream processing and the *lag trend* SLO is fulfilled. If we have $l = X$ the load is exactly equal to the capacity of the system under test, which means it is completely saturated, but the *lag trend* SLO still remains fulfilled as there are still no messages piling up. However, if $l > X$, the load exceeds the systems capacity while being fully saturated. As a result, records are piling up over the execution of the SLO experiment and the *lag trend* is not fulfilled anymore. Consequently, the largest load value of L for which the *lag trend* SLO is fulfilled approximates the value of the throughput of the system from the viewpoint of the USL. We can conclude that the USL can be applied based on the *Load Capacity Metric* if exclusively the *lag trend* SLO is used since it is only not fulfilled if the load applied to the system exceeds the systems throughput. However, the accuracy depends on the discretization of the load values, more specifically, on the distance between them. Moreover, it is assumed that no records get lost during the execution of the SLO experiment.

As a result, the *lag trend* SLO constitutes a special case where the USL can be applied and the USL model parameters retain their original physical meaning of contention and coherency since the resulting *Load Capacity Metric* approximates the throughput. We note that this is not the case for arbitrary SLOs, since they can depend on variables that are not related to throughput.

Dropped Records SLO

The other SLO that is currently supported by Theodolite is the *dropped records* SLO. It is violated if the number of dropped records during the stream processing is larger than a specified threshold. However, in contrast to the *lag trend* SLO, dropped records might occur even if the system under test is not fully saturated. This is due to the fact that their occurrence exclusively depends on the stream processing topology, which could be configured in a way that messages are dropped even if there are enough resources to process them. As a result, the *Load Capacity Metric* does not necessarily approximate the throughput when the *dropped records* SLO is used. Consequently, when applying the USL to the results, it should be noted that the meaning of the USL model parameters differs. That is α and β not only describe the physical properties of the system under test, but indicate how additional resources impact the fulfillment of the SLOs.

3.1.5 Load and Resource Dimensions

In the previous section, we discussed different properties of the scalability metrics and the SLOs that have to be considered when applying the USL. In this section, we discuss the requirements for the load and resource dimensions. We assume that both dimensions consist of ordered discrete values and each combination of load and resource values constitutes a possible SLO experiment. As discussed in the previous section, applying the USL presumes that the *Load Capacity Metric* is used during the benchmark execution. The reason for this is that the USL defines the scalability as a function of resources. While the accuracy of an USL model increases with a growing amount of known capacities, it also depends on the range of resource values that are examined to capture the characteristics of the scalability properly. Moreover, the discretization of the load values also influences the precision of the *Load Capacity Metric* and therefore the accuracy when applying the USL model. The higher the resolution of the load values, the more accurate are the results. According to the fact that Theodolite supports arbitrary integer resource dimensions, we can conclude that we can examine both hardware and software scalability, depending on whether the resource dimension describes the amount of physical processors (or instances), or virtual users. However, the case of software scalability is not particularly relevant for cloud applications, since one of the central benefits of the cloud is that it allows unlimited horizontal scale-out of properly sized instances. In general, we can subsume our findings of the last sections as follows:

- ▻ **Concurrency (USL) and resources [Henning and Hasselbring 2022] are terms that refer to the same variable.**

- ▻ **The USL can be used to describe the Load Capacity Metric (May not provide an accurate fit if the SLOs depend on variables that are not related to contention or coherency).**

- ▻ **When describing the Load Capacity Metric, the USL may capture unutilized resources in the model parameters.**

- ▻ **If the Load Capacity Metric approximates the throughput, the USL can be applied and the model parameters retain their original physical meaning.**

3.2 Extending the Theodolite Methodology to the Universal Scalability Law

In this section, we explain how the methodology of Theodolite can be extended to the USL. In Section 2.5, we explained that Theodolite's methodology is based on the three scalability attributes load, resources, and SLOs as well as on the *Resource Demand Metric* and the *Load Capacity Metric*. In Section 3.1, we discussed these aspects in the light of applying the USL. In this section, we discuss the consequences when extending the methodology of Theodolite to the USL.

As mentioned, applying the USL requires the results of a capacity metric. Therefore, applying the USL within the methodology of Theodolite is restricted to the usage of the *Load Capacity Metric*. We also discussed that for benchmarking cloud applications we are usually interested in the hardware scalability of systems. That is, on the one hand, we are interested into how the capacity of an application changes when the number of instances of the application is changed. On the other hand, we may also be interested in how increasing or decreasing the number of processors for a fixed amount of application instances impacts the capacity. As Theodolite supports arbitrary integer resource dimensions, we can benchmark both type of resource dimensions. Moreover, we can also examine software scalability. In this case, the resource dimension would constitute the number of virtual users.

Based on these considerations, the USL can be integrated into the benchmarking methodology of Theodolite in two ways: First, it is possible to extend the heuristic execution by a USL based search strategy that uses the USL to predict capacities during the benchmark execution with the aim to reduce the number of executed SLO experiments. Second, it is possible to extend the *Analysis Tools* component of Theodolite by a tool that allows to analyze the benchmark results with respect to the USL.

Chapter 4

Implementation

Motivated by **G2**, we describe all the parts of our implementation in this chapter. This includes integrating the USL into the implementation of Theodolite according to the extended methodology we presented in Chapter 3. While this constitutes the main part of our implementation, there is also a set of secondary extensions that we apply to Theodolite. These include the migration of the default Kafka implementation and the extension of the *Actions* mechanism of Theodolite. In the following, we describe our implementation in more detail.

4.1 Implementation Drivers

Taking into consideration the design decisions of the implementation of Theodolite, we identify the following drivers of our implementation.

4.1.1 Cloud Native Technologies

The implementation of Theodolite is divided into multiple components. Following cloud-native principles [Gannon et al. 2017], running Theodolite involves the execution of multiple containerized applications that communicate with each other via language-independent protocols and encodings. Each component that is implemented as part of our work should also follow cloud-native principles and be loosely coupled with the existing components of Theodolite.

4.1.2 Maintainability

Theodolite is implemented in a heterogeneous environment. Each component of Theodolite is implemented with the technologies that are suited the best for their specific domain. Having this separation between individual environments keeps the complexity of dependency management at a minimum and makes it easier for maintainers to work on specific components, as it is not required to install all the dependencies for the remaining components in the development environment. This separation of the different environments and codebases should be taken into consideration when introducing new technologies and components as part of this work.

4.2 Extending the Implementation of Theodolite by the Universal Scalability Law

In this section, we describe how we integrate the USL into the implementation of Theodolite. We extend Theodolite by the USL in two ways: First, we extend the heuristic execution. This can be realized with an online analysis approach that uses the USL to predict the results of individual SLO experiments during the benchmark execution in order to reduce the total amount of executed SLO experiments. To accomplish this, we introduce a new search strategy. Second, we extend the *Analysis Tools* component of Theodolite by a *USL Offline Analysis Tool* subcomponent that allows to analyze benchmark results with the USL.

4.2.1 Extending the Heuristic Benchmark Execution by the Universal Scalability Law

We extend the heuristic execution by introducing a new *UslSearch* search strategy which uses the USL in an online analysis approach. The main principle is to determine the capacity for a few resource values first and to use the gathered information to build an initial USL model. Then, for each of the remaining resource values, we use this model to predict the capacities and continue the search using the predicted capacities as starting points. Each time we determine the capacity for a new resource value, we use this new data point to improve the initial model. As a result, with each additional capacity found, the USL model is expected to become more accurate. Therefore, the amount of SLO experiments executed for determining the capacity for new resources value is expected to decrease with the amount of known capacities. In the following, we will first describe the structure of the underlying algorithm. Second, we will explain the application architecture of the related components, and finally, we introduce the most central aspects of our implementation.

Algorithmic Description

We can divide the algorithm into two phases. The first phase consists of determining the capacity for few resource values in order to build the initial USL model for the predictions. In particular, given a number of $N \geqslant 3$ resource values that are sorted increasingly, we first determine the capacity for the smallest resource value, the median, and the largest resource value using the already existing *BinarySearch* [Henning and Hasselbring 2022] search strategy as initial search strategy. We also call these three resource values the initial resource values. After that, phase two of the algorithm begins. The control flow of this phase is shown in Figure 4.1. It consists of $N - 3$ iterations, each for one of the remaining resource values in increasing order. The execution terminates if the iteration for the $(N - 1)$-th resource value is finished. For the k-th remaining resource value, an iteration is executed as follows: First, we generate our USL model from the capacities that are known to this point and we use this model to predict the capacity for the k-th resource value. Next, the

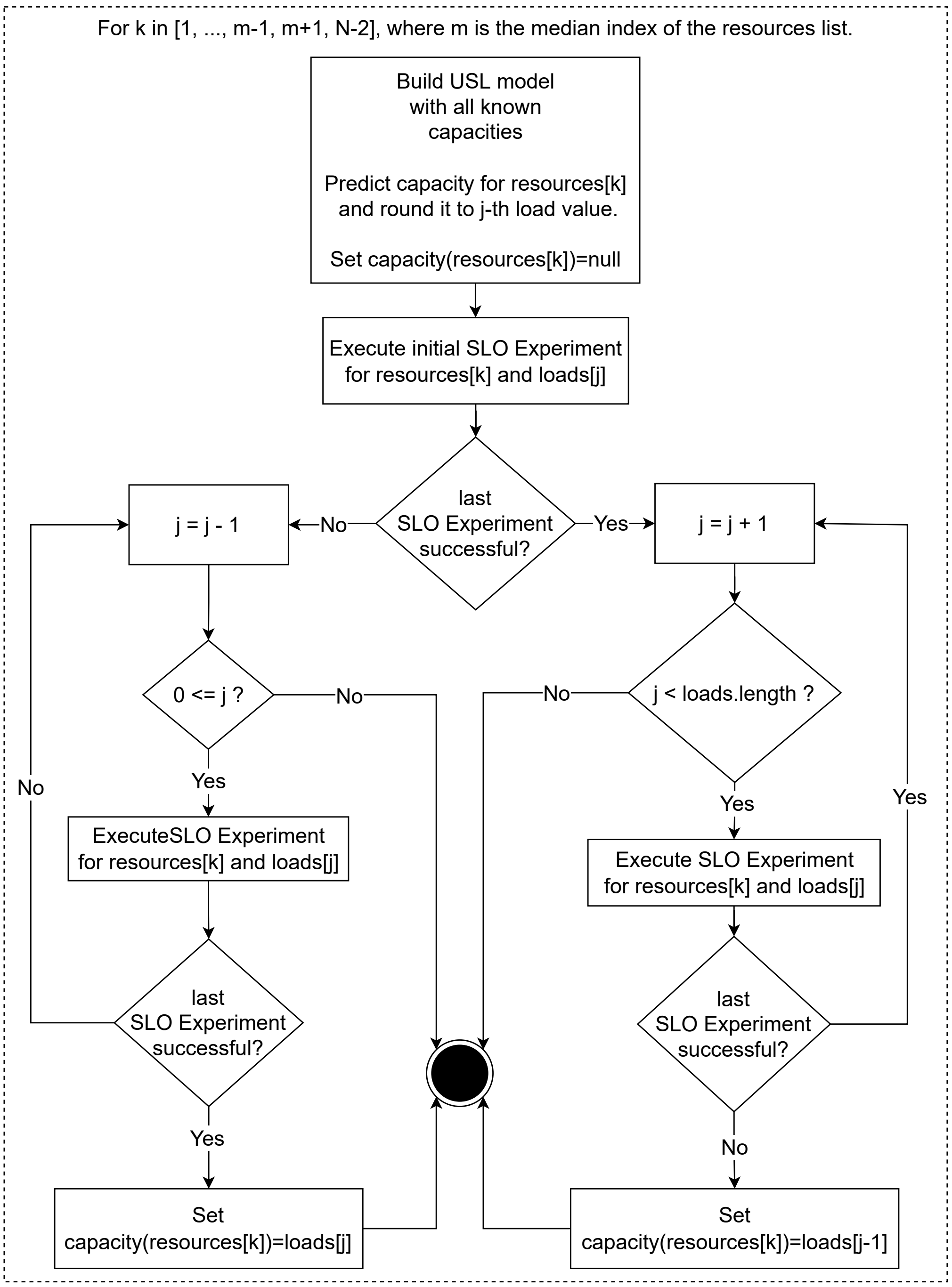

For k in [1, ..., m-1, m+1, N-2], where m is the median index of the resources list.
Build USL model with all known capacities
Predict capacity for resources[k] and round it to j-th load value.
Set capacity(resources[k])=null
Execute initial SLO Experiment for resources[k] and loads[j]
last SLO Experiment successful?
No
Yes
j = j - 1
j = j + 1
0 <= j ?
No
Yes
j < loads.length ?
No
Yes
ExecuteSLO Experiment for resources[k] and loads[j]
Execute SLO Experiment for resources[k] and loads[j]
last SLO Experiment successful?
No
Yes
last SLO Experiment successful?
Yes
No
Set capacity(resources[k])=loads[j]
Set capacity(resources[k])=loads[j-1]

Figure 4.1. Visualization of the control flow of the second phase of the *UslSearch* algorithm for *N* resources.

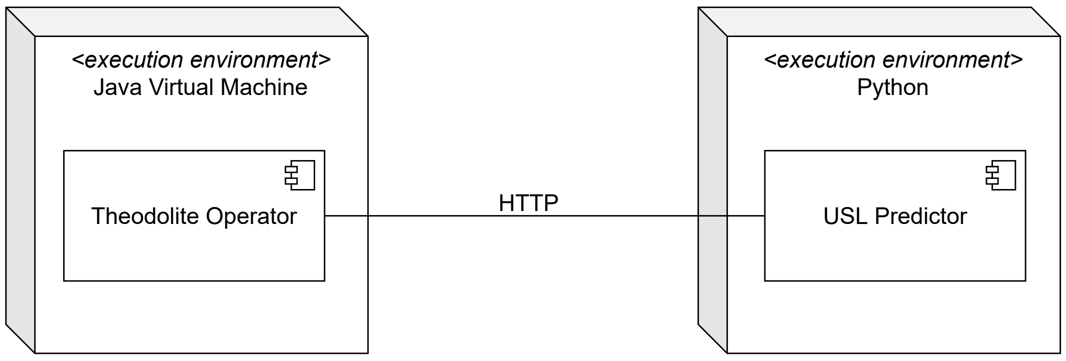


Figure 4.2. Visualization of the *USL Predictor* component that predicts the load for resources, based on the results from previous SLO experiments.

predicted capacity is rounded to the closest of the load values specified for the benchmark. Afterwards, an initial SLO experiment is executed for the rounded predicted load value. From this point, there are two options. In the first situation, all SLOs are fulfilled, resulting in a successful initial SLO experiment. In this case, we continue to iteratively examine all greater load values in increasing order by performing linear search until one SLO experiment does not succeed or all load values have been examined. The load of the last successful SLO experiment then corresponds to the capacity of the k-th resource value. In the second situation, at least one SLO is not fulfilled, meaning the initial SLO experiment is not successful. Analogously, we then continue with linear search for decreasing load values until an SLO experiment succeeds for a load value. If such an experiment exists, the corresponding load value is equal to the approximated capacity. Otherwise, we indicate that the capacity could not be determined for this resource value and we continue with the next resource value. However, if the range of the load and resource values are well-chosen, the algorithm should always be able to identify a successful SLO experiment. For the case where $N < 3$, we execute the initial search strategy for each of the resource values as we would not have enough data points to build a meaningful USL model otherwise.

System Architecture

As stated in Section 4.1, the drivers of our implementation are to follow the cloud-native principles of Theodolite and to ensure maintainability. This leads us to the design decision to introduce a new *USL Predictor* component that is used both to generate a USL model between the SLO experiments and make the predictions. The component is implemented in Python and makes use of the USL R library from Möding [2020] in combination with the rpy2 interface [Gaurier and Krassowski 2008], which allows to call R code from Python. Figure 4.2 gives a high-level description of how the *Theodolite Operator* interacts with the *USL Predictor*. The component exposes an HTTP endpoint that takes the input data for building the USL model and a set of resource values for which the capacity should be predicted. As an understandable and human readable encoding for data exchange, JSON is used for communication. We configured the Theodolite Helm Chart such that the *USL Predictor* is deployed as separate container in the same Kubernetes Pod where the *Theodolite Operator* is executed. Therefore, when installing Theodolite via the Helm Chart, the *USL Predictor* is enabled and deployed by default. It can also be disabled, though. Having the dedicated *USL Predictor* component allows the codebase regarding the USL model to be separated from the remaining Theodolite logic. As an advantage, there is no R or Python installation required because of the *USL Predictor*, during the development of only the *Theodolite Operator*.

Implementation

We introduce the *UslSearch* search strategy as a new class `UslSearch` in the Kotlin implementation of the *Theodolite Operator*. The corresponding class diagram can be found in Figure 4.3. The `SearchStrategy` is the base class for implementing new search strategies in order to approximate the scalability metric. Therefore, we introduce the `UslSearch` as a special type of `SearchStrategy`. The `UslSearch` follows the composite design pattern [Gamma et al. 1995]. That is, it references one initial `SearchStrategy` that is used to find the capacities for the three initial resource values. We use the existing `BinarySearch` as initial search strategy since for most cases this will reduce the number of required SLO experiments in order to determine the capacity for the initial resource values. Figure 4.4 displays the behavior of the `UslSearch`. Calls to other classes, not directly related to the structure of the algorithm, have been left out to allow a better comprehension of the implementation. As specified by the `SearchStrategy` class, the `UslSearch` implements the `applySearchStrategyByMetric` method. This method invokes the algorithm. The two phases of the algorithm are represented by individual calls to the method `applySearchStrategy`. This method performs a series of SLO experiments in order to determine the capacity for a given set of resources with the `SearchStrategy` that is specified as argument. The first call corresponds to phase one in which the `BinarySearch` object is used for finding the capacities of the initial resource values. The second call represents phase two in which the `UslSearch` object itself is used to determine the capacities of the remaining resources. During that phase, the *Usl Predictor* component is called via HTTP and the rounding of the predicted capacity values takes place.

Comparison to Existing Search Strategies

In Section 2.5 we introduced the search strategies *LinearSearch*, *BinarySearch*, and *RestrictionSearch* in combination with the *LowerBound* restriction strategy that are already supported by Theodolite. We also mentioned that choosing the right search strategy depends on the assumptions we are allowed to make with respect to the SLOs and the used scalability metric. For example, for the *RestrictionSearch* with the *LowerBound* restriction strategy,

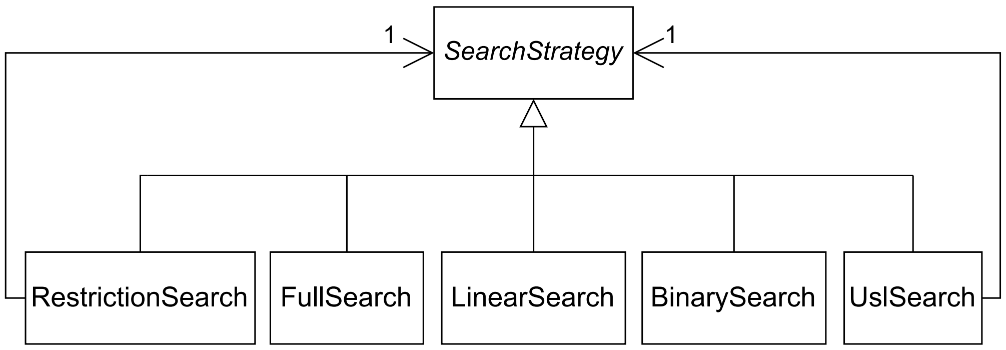


Figure 4.3. UML Class Diagram visualizing the changes induced by introducing the `UslSearch` search strategy. The remaining search strategies are already supported by Theodolite.

the assumption is that the scalability metric is monotonously increasing. For the *Load Capacity Metric* this means that for larger resource values, the capacity never decreases. However, this is a strong restriction, as there are many systems for which the capacity is not monotonously increasing. According to the USL, this is the case for any system where the coherency parameter β is greater than zero. Moreover, even if the system is not subject to coherency, the measured capacity may not be monotonously increasing due to volatility in the measurements. However, using the *RestrictionSearch* search strategy with *LowerBound* restriction reduces the number of executed experiments significantly since after each SLO experiment either the resource, or the load value is incremented. If we look at the algorithm of the `UslSearch`, we can recognize that the `UslSearch` does not assume the capacity function to be monotonously increasing. Rather, it only assumes the SLO evaluation function to be monotonously decreasing in the load dimension as it is assumed for the *LinearSearch* and *BinarySearch* search strategies. This observation is shown in Theorem 1.

Theorem 1 . *The UslSearch search strategy correctly approximates the well-defined Load Capacity Metric for any set of SLOs S' if the set-based SLO evaluation function $SLO_{S'}$ is monotonously decreasing in the load dimension.*

Proof. Let R be a discrete set of resource values and L be a discrete set of load values. Further, let S' be a set of SLOs and let $SLO_{S'}$ be monotonously decreasing in the load dimension. Let now $r \in R$. We have two cases:

Case 1: The resource value r is examined in the initial phase of the algorithm. Then the examination of r reduces to the initial search strategy. As this is *BinarySearch* and the set-based SLO evaluation function $SLO_{S'}$ is monotonously decreasing in the load dimension, the capacity is approximated correctly.

Case 2: The resource value r is not examined in the initial phase of the algorithm. Then the examination of the i-th load value starts with a rounded predicted capacity $l \in L$ from the *USL Predictor* component. Then we can make the following distinction:

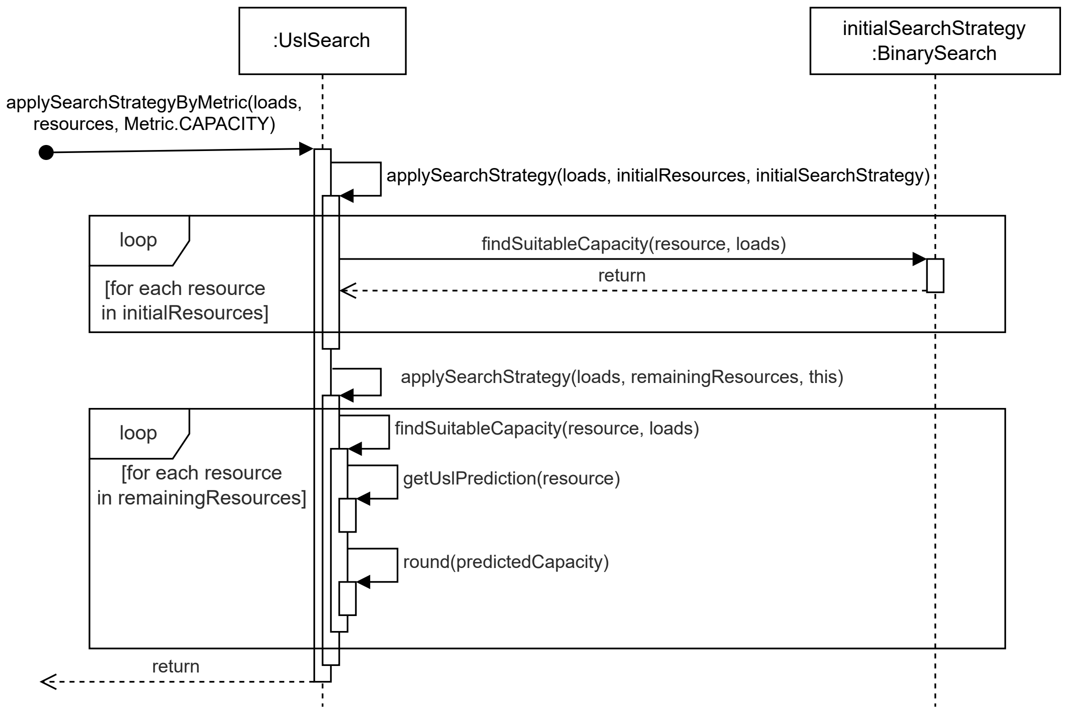

:UslSearch
initialSearchStrategy
:BinarySearch
applySearchStrategyByMetric(loads,
resources, Metric.CAPACITY)
applySearchStrategy(loads, initialResources, initialSearchStrategy)
loop
[for each resource
in initialResources]
findSuitableCapacity(resource, loads)
return
applySearchStrategy(loads, remainingResources, this)
loop
[for each resource
in remainingResources]
findSuitableCapacity(resource, loads)
getUslPrediction(resource)
round(predictedCapacity)
return

Figure 4.4. Visualization of the program logic of the `UslSearch` with *BinarySearch* as initial search strategy.

Case 2.1: $SLO_{S'}(l,r) = 1$. In this case, we perform linear search for increasing load values until all load values are examined, or one SLO experiment does not succeed. As $SLO_{S'}$ is monotonously decreasing in the load dimension, the capacity is approximated correctly.

Case 2.2: $SLO_{S'}(l,r) = 0$. In this case, we perform linear search for decreasing load values until one SLO experiment succeeds. Since we assumed the *Load Capacity Metric* to be well-defined, we know that such an SLO experiment exists. Due to the assumption that $SLO_{S'}$ is monotonously decreasing in the load dimension, the capacity is approximated correctly.

□

4.2.2 Extending the Analysis Tools by the Universal Scalability Law

In the previous section, we have introduced the USL in terms of using it to make the execution of benchmarks more efficient. However, we want to enable users of Theodolite to analyze the scalability of the system under test in terms of the USL regardless of the type

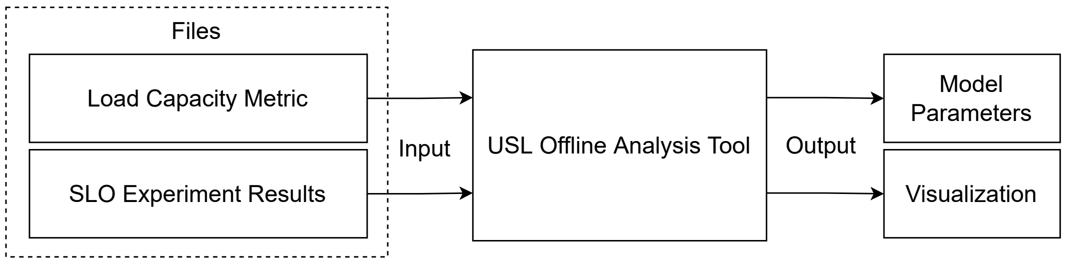


Figure 4.5. Overview of the input and output of the USL *Offline Analysis Tool* component

of search strategy used. Therefore, we extend the Theodolite *Analysis Tools* by a *USL Offline Analysis Tool* that allows to perform USL Analysis based on the benchmarking results. We provide our implementation in form of an R Markdown document as a user friendly execution environment. Our implementation is based on the USL R library from Möding [2020]. Figure 4.5 shows the usage of our *USL Offline Analysis Tool*. As the USL is based on the *Load Capacity Metric*, the application requires that for the benchmark execution with Theodolite, the metric `capacity` is specified in the `Benchmark` custom resource. This makes Theodolite use the *Load Capacity Metric* as scalability metric. After the execution of the N-th benchmark, the current version of Theodolite already generates the file `exp{N}_-capacity.csv` which contains the approximated values for the *Load Capacity Metric* for all resources of the benchmark encoded in the CSV format. In addition, Theodolite also generates the file `exp{N}-results` that contains information about all the executed SLO experiments and whether they were successful or not. The *USL Offline Analysis Tool* expects these files to be present in the same directory as the R Markdown document. When the R markdown document is executed, it takes the files contents and generates a USL model. Based on the model, it creates a visualization and a description of the model, which is exported to multiple files. The output contains the model parameters α, β, and γ, together with the model errors and a visualization of the USL model function, next to more model related information, such as Amdahl's asymptote. Moreover, it is reported how many SLO experiments were executed to determine the capacity per resource value, and how efficient the system under test behaves compared to linear scalability.

4.3 Further Extensions

In addition to the introduction of the USL to Theodolite as part of the heuristic execution and as part of the *Offline Analysis Tool*, we also extend Theodolite in the following ways in preparation of benchmarking ExplorViz: We address the migration of the Kafka implementation that is delivered per default with the Theodolite Helm Chart. Also, we enhance Theodolite's service orchestration capabilities by implementing a mechanism that allows the *Theodolite Operator* to delete arbitrary Kubernetes resources including custom resources.

4.3.1 Migrating the Kafka Implementation of Theodolite

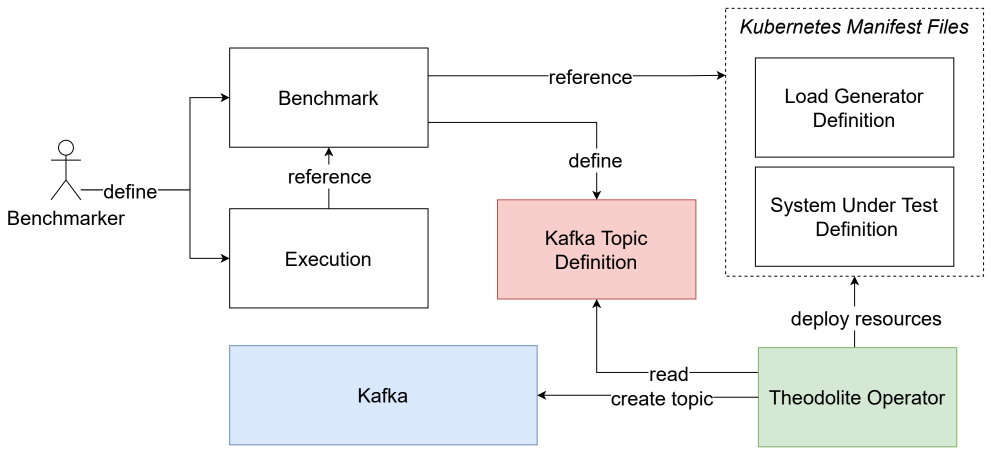
When installed via the Kubernetes package manager Helm, Theodolite comes with a Kafka Cluster that is ready to serve as message broker for stream processing. The management of Kafka topics, that is the creation of topics before an SLO experiment and their deletion afterwards, is entirely realized within Theodolite according to the information about the topics contained in Theodolite's Kubernetes custom resources. However, this adds a complexity to Theodolite that could be sourced out to a third party tool that is dedicated to managing Kafka in Kubernetes environments. This way, we aim to improve the stability of topic management.

Currently, Theodolite uses a Kafka implementation [Henning 2022] based on the Kafka implementation from the *Confluent Platform* [Confluent 2022a]. We replace this implementation with Strimzi Kafka, which is part of the CNCF as a Sandbox Project [Strimzi Developers 2019]. Strimzi provides a Kubernetes Operator that is able to deploy and manage Kafka brokers and related resources, for example, *Apache ZooKeeper* [Hunt et al. 2010]. Especially relevant for Theodolite, it provides management of topics via Kubernetes custom resources and exporting metrics for Prometheus. As a result, when using Strimzi, Theodolite does not require to use a third-party lag exporter to export the consumer lag to Prometheus. The consumer lag captures the backpressure [Karimov et al. 2018] that is used to compute the *lag trend metric* in order to apply *lag trend* SLO. Figure 4.6a depicts how Kafka topics are managed in the current version of Theodolite. The `Benchmark` custom resource directly contains information on what topics should be created (and deleted) for each SLO experiment. Before the load generator and the system under test, which are specified in Kubernetes manifest files, are created, the *Theodolite Operator* reads the information about the defined topics and creates them. After an SLO experiment, these topics are deleted by the *Theodolite Operator* respectively.

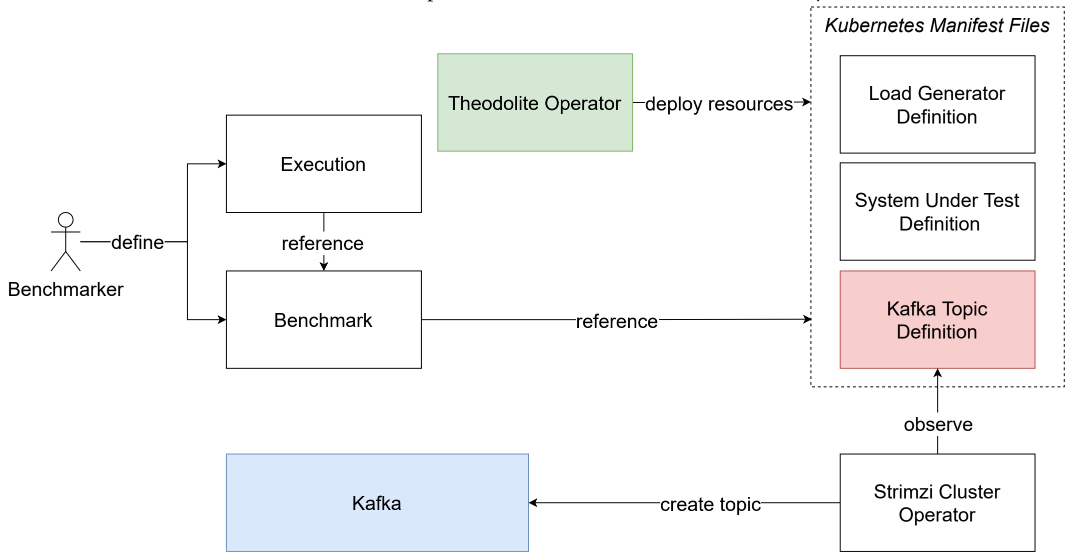
The new structure is visualized in Figure 4.6b. When we introduce the *Strimzi Cluster Operator* to Theodolite there is no need for the *Theodolite Operator* to manage the topics anymore. Instead, it is possible to reference the `KafkaTopic` custom resource from Strimzi as a Kubernetes manifest file from the user-defined benchmark in the same way as it is done for the load generator and the system under test. Due to the fact that the *Theodolite Operator* handles the execution of SLO experiments, it resolves all the referenced manifest files and deploys the specified resources as soon as an SLO experiment starts and it deletes them when an SLO experiment ends. In the case of a `KafkaTopic` resource being referenced, it also gets deployed or deleted accordingly. This is observed by the *Strimzi Cluster Operator*, which takes the necessary actions to create or delete the topic in the Kafka cluster. In the following, we describe the steps of the migration in more detail.

Step 1: Modification of the Benchmark Kubernetes Custom Resource from Theodolite

The information about the topics created and deleted before and after each SLO experiment is contained in the `Benchmark` custom resource of the *Theodolite Operator*. As this information



(a) Visualization of the topic creation without the *Strimzi Cluster Operator*.



(b) Visualization of the topic creation involving the *Strimzi Cluster Operator*.

Figure 4.6. Visualization of the Kafka topic creation before (a) and after (b) introducing the *Strimzi Cluster Operator*. In the current version of Theodolite (a), the Kafka topics are explicitly defined in the `Benchmark` custom resource of Theodolite and created by the *Theodolite Operator*. In contrast, when using the *Strimzi Cluster Operator*, the topics are defined as Kubernetes custom resources in manifest files.

is not required anymore when the management of topics is performed by Strimzi, we delete this information from the Theodolite's `Benchmark` custom resource definition.

Step 2: Define Kafka Cluster for Strimzi

The Strimzi Kubernetes Operator provides the `Kafka` Kubernetes custom resource which defines brokers, ZooKeeper nodes, and metric export for Prometheus. A minimal example of the `Kafka` custom resource is shown in Listing 4.1. Specifying such a custom resource as part of the Helm Chart of Theodolite replaces the previous implementation.

Listing 4.1. A Kubernetes manifest file that contains a minimal example for `Kafka` custom resource for Strimzi.

```yaml
apiVersion: kafka.strimzi.io/v1beta2 # version of the custom resource
kind: Kafka # type of the custom resource
metadata:
  name: theodolite-kafka # name of the custom resource
spec:
  kafka:
    replicas: 5 # use 5 Kafka brokers
    storage:
      type: ephemeral # no permanent storage
    listeners: # allow connection to Kafka brokers
      - name: plain
        port: 9092
        type: internal
        tls: false
  zookeeper:
    replicas: 3 # use 3 ZooKeeper nodes
    storage:
      type: ephemeral # no permanent storage
  kafkaExporter: {} # export consumer lag for Prometheus
```

Step 3: Replace Kafka Implementation in Helm Chart

We organize the Strimzi Kubernetes manifest files in the `template` directory of the Helm Chart and we make it possible to overwrite most of its contents via the Charts `values.yaml` file using Go templates. This is the preferred method for users to customize Helm Charts according to their requirements. I.e., it allows to specify the number of Kafka brokers and ZooKeeper nodes, the Kubernetes resource limits applied to the Pods running the Kafka and ZooKeeper, and the detailed configuration of the Kafka brokers.

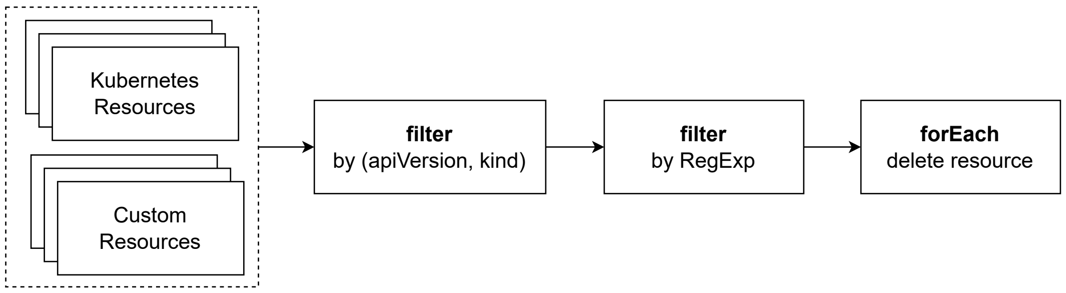


Figure 4.7. Overview over the operations that are performed when a `DeleteResourceAction` is executed.

We have integrated the new Kafka implementation as part of the release of Theodolite version `v0.7.0`.

4.3.2 Extending Theodolite's Action Mechanism

As stated in Section 2.5, Theodolite provides the *Actions* mechanism. Actions are small tasks that the *Theodolite Operator* executes in order to perform initialization steps before individual SLO experiments, or to reset the state of components after an experiment. However, in the current version of Theodolite, there is only one type of action that is supported. This type of action allows to execute shell commands inside running containers. However, a useful extension is to allow Theodolite to delete Kubernetes resources including custom resources. Specifically, our motivation to introduce this new action type is that it allows us to manage the creation and deletion of Kafka topics via the `KafkaTopic` custom resource from Strimzi. While the topics that are defined for a benchmark are already managed by the Theodolite Operator, there are cases where additional topics are created during the execution of an SLO experiment. For example, this can be the case when stateful operators are used in a Kafka Streams application. Not deleting these topics would result in consecutive SLO experiments restoring the state of previous experiments. Therefore, we extend the *Actions* mechanism by implementing a new type of action which allows to delete arbitrary Kubernetes resources. We call this new action type *Delete Resource Action*. This way, we are able to ensure that all topics that match certain conditions are deleted between SLO experiments.

The implementation of the new action consists of the extension of the `Benchmark` custom resource and additional application logic in the *Theodolite Operator*, which executes the action. In the current version, the `Benchmark` custom resource defines the two lists `beforeActions` and `afterActions` in which the individual actions are specified. We modify the definitions of both lists as follows: Each action can have the properties `exec` and `delete`. According to which of the two properties is set, the type of the action is determined. Setting the property `delete` sets the type of the action to *Delete Resource Action*. In this case, also

```yaml
apiVersion: theodolite.com/v1
kind: benchmark
metadata:
    name: example-benchmark
spec:
  sut:
    ...
    beforeActions:
      - delete:
          selector:
            apiVersion: kafka.strimzi.io/v1beta2
            kind: KafkaTopic
            nameRegex: "^input-topic-.*"
    afterActions:
      - delete:
          selector:
            apiVersion: kafka.strimzi.io/v1beta2
            kind: KafkaTopic
            nameRegex: "^input-topic-.*"
    ...
```

Listing 4.2. A Kubernetes manifest file that specifies a `Benchmark` custom resource that uses the new *Delete Resource Action*.

another property called `selector` must be set. This property has the following fields:

`apiVersion`: `String`
: Specifies the API version of the resource that should be deleted.

`kind`: `String`
: Specifies the Kubernetes kind of the resource that shoud be deleted.

`nameRegex`: `String`
: Specifies the regular expression that is matched with the `metadata.name` value of the resources to identify the resources that should be deleted.

An manifest file for a `Benchmark` custom resource that specifies both the lists `beforeActions` and `afterActions` is shown in Listing 4.2. The example shows a case where all topics that have a name beginning with the string `input-topic` are deleted before and after each SLO experiment,

We introduce the new action type as a new class `DeleteResourceAction` (see Listing 4.3) in the Kotlin implementation of the *Theodolite Operator*. Figure 4.7 visualizes the operations that are performed during the execution of the `DeleteResourceAction`. First, all Kubernetes

```kotlin
class DeleteResourceAction {

    lateinit var selector: DeleteResourceActionSelector

    fun exec(client: NamespacedKubernetesClient) {
        val regExp = selector.nameRegex.toRegex()
        val k8sManager = K8sManager(client)
        client
            .genericKubernetesResources(selector.apiVersion, selector.kind)
            .inNamespace(client.namespace)
            .list()
            .items
            .filter { regExp.matches(it.metadata.name) }
            .forEach{ k8sManager.remove(it) }
    }
}
```

Listing 4.3. The Kotlin implementation of the class *DeleteResourceAction*.

Resources including custom resources are retrieved via the Kubernetes API. Second, the resources are filtered by the Kubernetes specific metadata fields `apiVersion` and `kind`. Afterwards, the `metadata.name` field of the resources are matched with the user-specified regular expression of the field `nameRegex` in order to filter for the resources that should be deleted. In the last step, all filtered resources are deleted from the cluster via the fabric8 Kubernetes client[1].

In the Appendix of this work, we provide a detailed overview where the implementation of our extensions can be found.

4.4 Benchmark Implementation

In this section, we describe the implementation and configuration that is a prerequisite for the execution of our benchmarks.

4.4.1 Implementation and Configuration of the Load Generators

We implement dedicated load generators using Theodolite's load generator library that supports generating load for specific Kafka topics. We need to ensure that for each experiment the respective load generator is started with the according load definition. The type of the load generator depends on the benchmark. Depending on the benchmarked

[1]https://github.com/fabric8io/kubernetes-client

microservice of ExplorViz, the type of messages in the Kafka topics are either generated via the data serialization mechanism *Protobuf* [Google 2001], or the serialization framework *Apache Avro* [Apache Software Foundation 2022].

4.4.2 Setup of the System Under Test

First, we need to configure our system under test for all benchmarks. Depending on the benchmark it consists of one or multiple ExplorViz microservices and the required environment services. In addition to Kafka and ZooKeeper, the ExplorViz microservices require the following services as environment: *Confluent Schema Registry* [Confluent 2022b] and the databases *MongoDB* [MongoDB 2022] and *Cassandra* [Lakshman and Malik 2010]. The services Confluent Schema Registry, Kafka, and ZooKeeper are already provided by the Theodolite Helm Chart based on Strimzi Kafka. We extend the Helm Chart so that it also provides both of the remaining services. Moreover, we need to make sure that all the services are configured correctly, for example, that the databases have enough resources to sustain the loads during our benchmark execution. Also, we make sure that the databases are reset between individual SLO experiments during the benchmarks. To accomplish this, we deploy dedicated containers in Kubernetes that open client connections to the databases. These containers can be used by Theodolite to perform the initialization and reset of the databases with Theodolite's *Actions*. Finally, we also have to configure the system under test in a way that it can communicate with all the required services from the environment. For all required applications of the system under tests, load generators, and the environment services, we create Kubernetes manifest files that define how the respective containers should be orchestrated in Kubernetes.

4.4.3 Creation of Theodolite Custom Resources

In addition to all the Kubernetes manifest files that we need to define for our system under tests and the load generators, we also need to create a Kubernetes custom resource of the kind `Benchmark` per benchmark. This custom resource defines the static components of a benchmark. Central pieces of information contained are the type of the load and resource dimensions, the name by which the benchmark is identified, the Kubernetes manifest files that define the system under test and the load generator, whether the `capacity` or `demand` metric is used, and which SLOs should be applied. Moreover, for each execution of a benchmark, a Kubernetes custom resource of the type `Execution` is required by Theodolite. It references a `Benchmark` and contains additional information that is required for executing the benchmark. For example, it contains lists of concrete values for the load and resource dimensions types specified by the benchmark and it defines the duration of each SLO experiment.

Chapter 5

Benchmarking ExplorViz

In Chapter 4, we introduced extensions to Theodolite in preparation of our final goal **G3**, which addresses benchmarking ExplorViz. In this chapter, we present an overview of the structure of our benchmarks and how we execute them based on our extended version of Theodolite. This includes a detailed description of the load that is generated by our load generators and of how we configure the components that are deployed for the benchmark execution.

5.1 Benchmark Design

Our benchmarking process can broadly be divided into two phases: First, we benchmark the three ExplorViz microservices *Adapter Service*, *Landscape Service*, and *Trace Service* in isolation. The motivation for this is to first assess the scalability of the services individually and to use the findings for the design of more complex benchmarks involving all the microservices of the trace analysis at the same time. Second, we benchmark ExplorViz in such configurations where the trace analysis is executed as a whole system. Table 5.1 gives an overview of the planned benchmark categories. Each category represents multiple benchmarks with different configurations, for example, with different CPU resources granted to the system under test. In this section, we describe the individual benchmarks belonging to the four categories in more detail.

For convenience, we introduce the naming scheme **SUTName.[...Configuration]** to reference individual benchmarks, from the remaining part of this work. The string **SUTName** represents the name of the system under test, for example, the string **Adapter** indicates that the system under test of the benchmark is the *Adapter Service*. The string **[...Configuration]** acts as a placeholder for benchmark specific information that is sufficient to identify an individual benchmark within the respective category.

5.1.1 Benchmarking the ExplorViz Microservices in Isolation

Benchmark categories **C1–C3** represent the isolated benchmarks of the individual microservices of ExplorViz' trace analysis. Each of the three categories maps to benchmarks where one of the microservices *Adapter Service*, *Landscape Service*, and *Trace Service* is the system under test. In the following, we describe these three benchmark categories in more detail.

Table 5.1. The column *Category* displays the identifier of the benchmark category. The column *SUT* gives information about how the system under test is structured: For example, the combination of the letters *A, L, T* means that the system under test consists of the *Adapter Service*, *Landscape Service*, and *Trace Service*. The column *Resources [N]* shows the type of the resource dimension, or in USL terminology what the scaling variable N is: In all benchmarks, the resource dimension is either the number of instances of the system under test *I* or the amount of CPU cores limiting the maximal usage for the services of the system under test *CPU*. The column *Load* defines the load dimension used. In all benchmarks, the load dimension is the number of traces generated per second (TrPS). The column *Scal. Metric [X(N)]* shows the metric that is used to measure the scalability. In order to analyze our experiment results in terms of the USL, we use the *Load Capacity Metric* (*capacity*) in all of our benchmarks. In USL terminology, this corresponds to the capacity function $X(N)$. The column *SLOs* shows the set of SLOs used to assess whether individual SLO experiments are successful. In our benchmarks, we either use the *lag trend* SLO (*LT*) or the *dropped messages* SLO (*DR*).

Category	SUT	Resources $[N]$	Load	Scal. Metric $[X(N)]$	SLOs
C1	A	I	TrPS	capacity	LT
C2	L	I	TrPS	capacity	LT
C3	T	I	TrPS	capacity	LT, DR
C4	A, L, T	I, CPU	TrPS	capacity	LT, DR

C1: Adapter Service

The benchmarks for the *Adapter Service* correspond to the benchmark category **C1**. We executed four benchmarks for the *Adapter Service*. The benchmarks can be divided into two subcategories:

In the first, we configure the load generators in a way that each trace consists of exactly one span representing traces within the application monitored by ExplorViz that consist of exactly one method call. In the second subcategory, we configure the amount of spans per trace to be ten. We also say that the trace depth for the benchmarks is one for the first subcategory and ten for the second. A trace depth of ten is also the upper bound for ExplorViz, as traces that contain more spans are truncated during the trace analysis. For each of the two trace depths, we execute two benchmarks:

For the first benchmark, each instance is granted Kubernetes resource limits of 4 GiB RAM and 2 CPU cores. For the second, the resource limits where 4 GiB RAM and half a CPU core. Since the available execution time on each CPU is divided into 1 000 fractions (milliCPU) in Kubernetes, we can assign half a CPU core for a Kubernetes container by specifying resources of 500 milliCPU (short: 500m). As a result, we have the four benchmarks **Adapter.Span1.2cpu** and **Adapter.Span1.500m** for a trace depth of one, and **Adapter.Span10.2cpu** and **Adapter.Span10.500m** for a trace depth of ten.

For each of the benchmarks, we use the *lag trend ratio* SLO from Theodolite. This SLO automatically computes the threshold for a given a ratio parameter *r* relative to the load

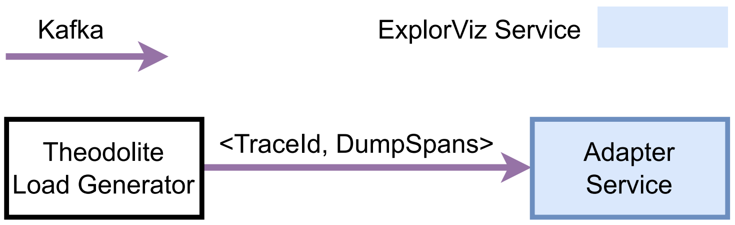


Figure 5.1. Visualization of the structure of the benchmark category **C1** that contains all benchmarks of the *Adapter Service* in isolation. The value of each message is of the type `DumpSpans` which is a wrapper for one of multiple spans. The key of each message corresponds to the random string trace identifier the first span belongs to.

per second l. The SLO is fulfilled as long as the consumer lag growth rate does not exceed the threshold $r \cdot l$. We set the ratio value to 0.025, which means that we tolerate a lag trend of no more than 2.5% of the load. This way, we ensure that the small amount of increase in the consumer lag we allow is proportional to the load of the SLO experiment. The *Adapter Service* only performs stateless stream processing operations and therefore we do not expect dropped records during the execution. As a result, we consider the *lag trend ratio* SLO to be sufficient for assessing the scalability of the *Adapter Service*.

Figure 5.1 displays the structure of the four benchmarks. The *Adapter Service* receives its input via Kafka in the `DumpSpans`[1] Protocol Buffer format generated by the Protobuf data serialization framework. Therefore, the load generator generates a certain amount of traces with the corresponding trace depth per second and serializes them with Protobuf before writing each message to Kafka. Each generated message is structured as follows: The value of each message is of the type `DumpSpans`, which is a wrapper around a list spans that have the type `Span`[2]. Therefore, each message may contain the information of one or multiple spans. The key of the message consists of the random string identifier of the trace that the first span in the message belongs to. We configure our load generator in a way the `DumpSpans` type always contains all spans of a generated trace. Since in a realistic tracing scenario all spans of a trace occur at a similar point of time during the program execution, it is likely that spans belonging to the same trace are dispatched together to Kafka in one message. For both benchmarks, we examined 3 up to 30 instances of the *Adapter Service* in steps of 3 instances as the resource dimension. Based on the findings of Henning and Hasselbring [2022], we execute each of the SLO experiments of all the benchmarks for a period of 240 seconds of which we consider the first 120 seconds as warmup time.

[1]https://github.com/yancl/opencensus-go-exporter-kafka/blob/master/proto/dump/v1/dump.proto
[2]https://github.com/census-instrumentation/opencensus-proto/tree/master/src/opencensus/proto/trace/v1

C2: Landscape Service

Benchmark category **C2** contains two benchmarks where the system under test is the *Landscape Service* exclusively. The *Landscape Service* is responsible for building a hierarchial representation of the monitored system. It achieves this by storing the observed method calls in flattened tree structure. To avoid multiple database accesses for methods that are called more than once, the *Landscape Service* stores already processed methods in a in memory cache.

In the first benchmark **Landscape.500m.RandomMethodNames**, we examine how well the *Landscape Service* is able to process spans of which the respective method has not been cached. This is realized by generating random alphanumeric method names with a length of 32 characters for each generated span within a trace.

With the second benchmark **Landscape.500m.ConstantMethodNames**, we examine how well the *Landscape Service* is able to handle calls of cached methods. This is realized by generating all spans for the same method. Analogous to the benchmarks from category **C1**, we choose the amount of traces per second as load dimension and the *lag trend ratio* SLO with a ratio value of 0.025 for deciding whether individual SLO experiments are successful or not.

For both benchmarks, we initially aimed to examine up to 30 instances in the resource dimension. However, we observed that when granting the *Landscape Service* Kubernetes resource limits of 4 GiB RAM and 2 CPU cores, we are not able to generate enough load to assess the capacity of high instances values. Therefore, we decided to decrease the resource limits to 4 GiB RAM and half a CPU core. With this setup, we are able to generate enough load to assess capacity from 3 up to 30 instances in steps of 3 instances for the benchmark where the method names were generated randomly to avoid cache hits. However, for the benchmark where all spans were generated for the same method, we were still not able to generate enough load to assess the maximum capacity for more than six instances. Therefore, we decided to examine deployments from only one to six instances.

The service topology of both benchmarks is depicted in Figure 5.2. The structure is similar to the structure of the benchmarks for the *Adapter Service*. The main difference is that the load needs to be generated on a different Kafka topic and with a different message format. More specifically, the value of each message has the type `SpanStructure`[3] which is generated by the data serialization system *Apache Avro*. It contains the identifier of the landscape the span belongs to, the fully-qualified name of the called method, and information regarding the host system where the method was executed. The key consists of the randomly generated string identifier of the span. For both benchmarks, we execute each of the SLO experiments for a period of 240 seconds of which we consider the first 120 seconds as warmup time.

[3]https://github.com/ExplorViz/adapter-service/blob/master/src/main/avro/AdapterEventProtocol.avdl

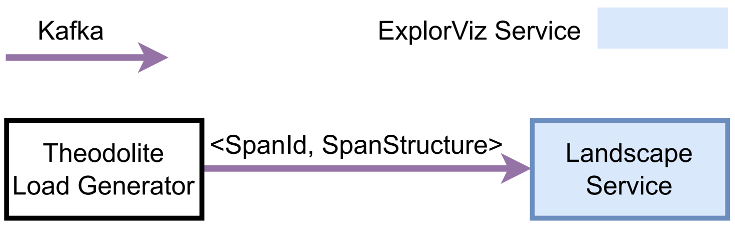


Figure 5.2. Visualization of the structure of the benchmark category **C2** that contains all isolated benchmarks of the *Landscape Service*. Each message corresponds to one span. The key of each message is the string identifier of the span and the value consist of the type `SpanStructure` which contains all structural information of the span.

C3: Trace Service

Benchmark category **C3**, contains the benchmarks where only the *Trace Service* is the system under test. The *Trace Service* is the only microservice of the ExplorViz trace analysis that uses stateful stream processing operators. More specifically, it uses an aggregation over a sliding time window to collect all the spans that are associated with individual traces.

The setup is similar to that of the benchmarks for the other two microservices. The main difference is that additionally to the *lag trend ratio*, we use the *dropped records ratio* SLO. This SLO detects whether there are skipped messages during the processing, for example, when they are expired with respect to the used time window. Similarly to the *lag trend ratio*, the *dropped records ratio* SLO is also based on specifying a ratio instead of specifying the threshold for the allowed amount of dropped records directly. Our motivation for using these two SLOs is as follows: The *Trace Service* uses a stateful aggregation over a sliding time window of the last ten seconds to collect all spans and compose them into traces. This means that spans that are to be processed too late are dropped. However, from the perspective of the *lag trend ratio* SLO, dropped records are not piling up within the topic. Therefore, the consumer lag decreases for dropped records in the same way as if the records were being processed as intended. The role of the *dropped records ratio* SLO is to detect such cases where the consumer lag does not exceed the tolerated bounds because of records being dropped. Therefore, the combination of these two SLOs is well suited for benchmarking stream processing applications in which dropped records are expected to occur. We configure the ratio for both the SLOs to be 0.05. The reason why we choose a larger value as ratio for the SLO is that by doing so, we reduce the sensitivity of the SLO for volatility in the underlying metrics when benchmarking smaller load values.

The benchmark category **C3** contains the two benchmarks **Trace.Span1** and **Trace.Span10** where traces of depth one and ten are examined respectively. For both benchmarks, we limit the CPU resources of each instance of the *Trace Service* to 4 GiB of RAM and 1 CPU core. Figure 5.3 visualizes the service topology of the benchmarks. The structure is similar to the benchmarks of the other two microservices except for the fact that another Kafka

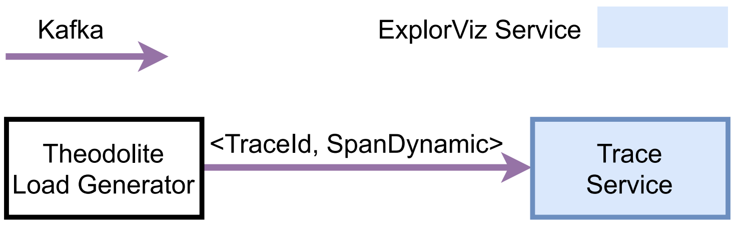


Figure 5.3. Visualization of the structure of the benchmark category **C3** that contains all isolated benchmarks of the *Trace Service*. The value of each message has the type `SpanDynamic`. The key of each message is the string identifier of the trace the corresponding span belongs to.

topic is used for the load generation and that the values of the input messages have the `SpanDynamic`[4] type which is also generated with Apache Avro. This type contains information about when the corresponding method was entered, when it's execution ended, and from which method it was called. We use the identifier of the trace the span belongs to as key of each message. For each of the two benchmarks, we execute each of the SLO experiments for a period of 420 seconds of which we consider the first 120 seconds as warmup time.

5.1.2 Benchmarking ExplorViz's Trace Analysis as a System

The benchmark category **C4** contains two benchmarks in which we examine the trace analysis as a whole system.

In the first benchmark, we examine different instance values for the three microservices. Therefore, we make use of the fact that Theodolite supports arbitrary integer load and resource dimensions. Let (I_A, I_L, I_T) be a 3-tuple that describes the ratio of different instance values of the three microservices *Adapter Service* (I_A), *Landscape Service* (I_L), and *Trace Service* (I_T). Let $R = (r_1, .., r_n)$ be the n different resource values for the benchmark in increasing order. We define the amounts of instances of the three microservices that are examined for the k-th resource value as $r_k(I_A, I_L, I_T) = (r_k I_A, r_k I_L, r_k I_T)$. For the benchmark, we decide to use a ratio of $(2, 1, 3)$, which corresponds to two instances of the *Adapter Service*, one instance of the *Landscape Service*, and 3 instances of the *Trace Service*. The resource values we examine are all the integer values from one to ten resulting in a maximum of 20 instances of the *Adapter Service*, 10 instances of the *Landscape Service*, and 30 instances of the *Trace Service*. In the remaining part of this work, we will refer to this benchmark as **TraceAnalysis.ProportionalResources**. We limit the number of CPU cores of each instance of the *Adapter Service* and *Landscape Service* to 500 milliCPU, and of the *Trace Service* to 1 CPU core. For all services, we allow each instance to use up to 4 GiB of RAM.

In the second benchmark, we fix the number of instances for all three microservices, and we examine the CPU resources as load dimension. In more detail, we examine a fixed

[4]https://github.com/ExplorViz/adapter-service/blob/master/src/main/avro/AdapterEventProtocol.avdl

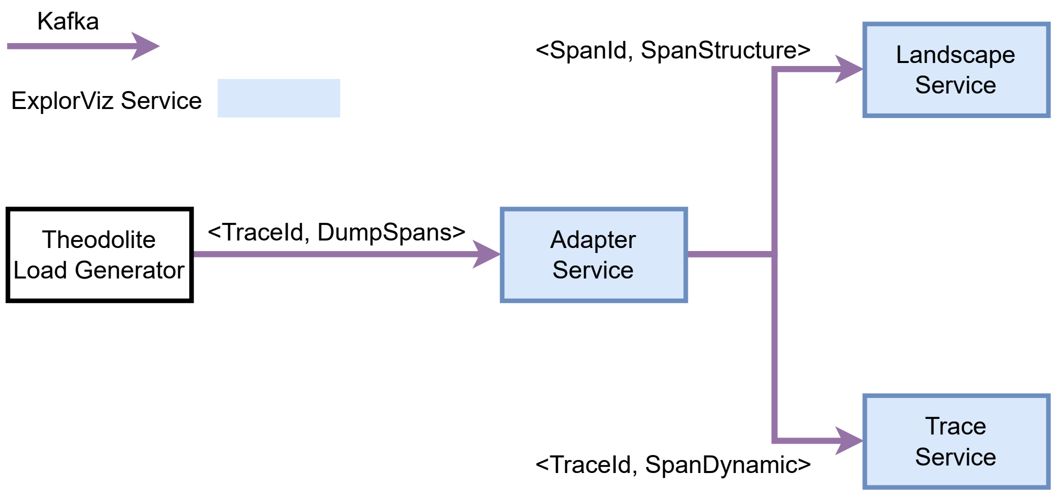


Figure 5.4. Visualization of the system under test and load generator setup for the benchmark category **C4** where the trace analysis is examined as a whole system. As for the benchmark category **C1**, the value of each generated message has the `DumpSpans` format and the key of the message is equal to the trace identifier of the first span that is contained in the corresponding `DumpSpans` object.

configuration of six instances of the *Adapter Service*, 3 instances of the *Landscape Service*, and nine instances of the *Trace Service* with load values between 350 milliCPU and 2 CPU cores with a resolution of 150 milliCPU. We refer to this benchmark as **TraceAnalysis.Cpu**. As for the benchmark category **C3**, we use the combination of the *lag trend ratio* and *dropped records* SLOs with a ratio value of 0.05 to assess the capacities for the load values for both benchmarks. For both benchmarks, the service topology is visualized in Figure 5.4. The load generator setup is the same as for the benchmarks for the *Adapter Service* in the benchmark category **C1** since it is required to generate the input data at the *Adapter Service*. From the *Adapter Service*, the spans are forwarded to the two downstream microservices. For both benchmarks, all generated traces have a depth of one span and we execute each of the SLO experiments for a duration of 420 seconds of which we consider the first 120 seconds as warmup time.

5.2 Benchmark Execution

In the previous section, we explained the structural design of all benchmarks, we execute. In this section, we give more information about the execution environment and the experiment setup.

5.2.1 Definition of Theodolite's Custom Resources

In order to be able to execute benchmarks of ExplorViz, we have to add the required configuration files for Theodolite. As mentioned in Section 2.5, Theodolite provides a declarative API via Kubernetes custom resources. With the help of this API, all services required for our benchmarks can be orchestrated by Theodolite in Kubernetes. For each benchmark, there are three main pieces of information that we need to provide in the custom resources: First, the references to the Kubernetes manifest files that define the load generator, second, a reference to the manifest files that define and the system under test, and third the definition of the SLOs.

5.2.2 Execution Environment

We execute our benchmarks on a 5 Node Cluster running Kubernetes 1.14.1. Each node is equipped with two Intel Xeon Gold 6130 (2.1 GHz, 16 Cores) CPUs with a total of 384 GB RAM.

5.2.3 Experiment Setup

In the following, we describe the general setup that most of these benchmarks have in common if not stated otherwise. We configure our benchmarks in a way that each instance of the load generator runs in a Kubernetes container with resource limits of 2 CPU cores and 4 GiB of RAM. In addition, we ensure that enough load generator instances are started in each SLO experiment in order to generate the required load. We execute our experiments with 10 Kafka brokers, running Kafka version 3.1. We set the configuration parameter `log.retention.ms` to `-1` to prevent old messages from being deleted during the benchmark execution and leave the remaining configuration parameters in the default configuration of the Theodolite Helm Chart. We configured the resource limits in Kubernetes so that each broker has 16 GiB RAM and 8 CPU cores. In addition, we configure each topic to have 250 partitions and each partition to be replicated across three Kafka brokers. Since we do not benchmark more than 250 instances, this allows that the full concurrency of all stream processing instances can be utilized. Also, we configure the load generators and all Kafka Streams applications of the system under test to commit processed messages every 5 seconds (`commit.interval.ms=5000`) and the Kafka Streams applications to run one Kafka Streams thread per instance. As search strategy, we mainly use the *UslSearch* search strategy and *RestrictionSearch* or *BinarySearch* as reference search strategies.

5.2.4 Replication Package

We provide all the resources required for the execution of our benchmarks in a replication package [Ehrenstein 2022] to allow the reproduction of our results.

Chapter 6

Experimental Evaluation

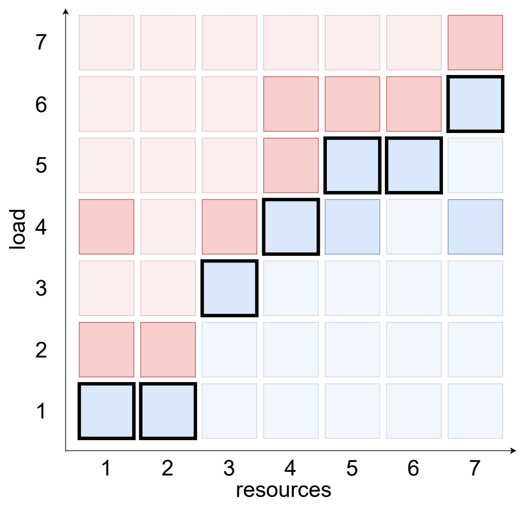
In this chapter, we evaluate the findings of this work. The chapter is divided into three main parts: In the first and second part, we evaluate the implementation of our extensions of Theodolite. Here, we focus on the new search strategy we introduced as extension of the heuristic benchmark execution of Theodolite and on the *USL Offline Analysis Tool*. In the third part, we evaluate the results of benchmarking the scalability of ExplorViz.

6.1 Evaluating the UslSearch Search Strategy

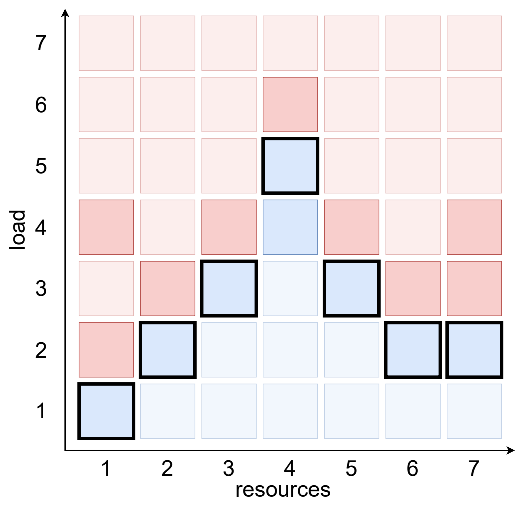
One central part of our implementation is the extension of the heuristic execution of benchmarks with Theodolite by using an online analysis approach based on the USL. We evaluate the implemented search strategy terms of correctness and efficiency.

6.1.1 Correctness

First, we address the evaluation of the implementation of the *UslSearch* search strategy in terms of correctness. For this, we consider the following two scenarios: The first scenario (**S1**) simulates a benchmark where the *Load Capacity Metric* is monotonously increasing. The second scenario (**S2**) simulates a benchmark where the *Load Capacity Metric* is not monotonously increasing. This means that at some point increasing the resources results in capacity degradation. With respect to the USL, this corresponds to a situation where $\beta > 0$. We implement a unit test for both of the scenarios. We add this unit test to the set of unit tests for the *Theodolite Operator*. For both scenarios, the load and resource dimensions are the integers between one and seven. Table 6.1 shows the expected results of the *Load Capacity Metric* for both scenarios. Figure 6.1 displays SLO experiments for both scenarios that were executed during the unit test and the resulting values for the *Load Capacity Metric*. If we compare the observed values to the expected capacities from Table 6.1, we can see that the *UslSearch* search strategy identifies the expected capacity values. As a result, we can conclude that the *UslSearch* can be used to determine the *Load Capacity Metric*, regardless of whether the metric is monotonously increasing.



(a) Visualization of the executed SLO experiments for the monotonously increasing *Load Capacity Metric* for the scenario **S1**.



(b) Visualization of the executed SLO experiments for the not monotonously increasing *Load Capacity Metric* for the scenario **S2**.

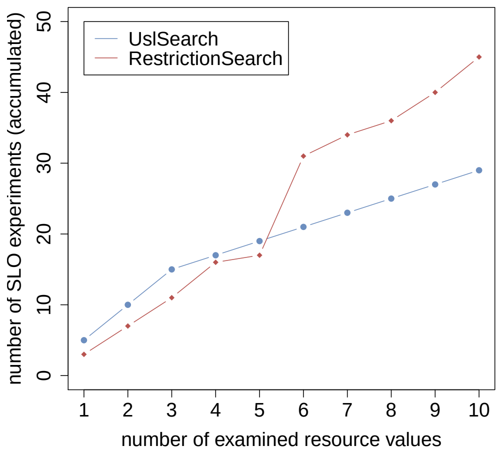
Figure 6.1. Visualization of executed SLO experiments for the unit tests of the *UslSearch* search strategy. Figure (a) shows the SLO experiments for the monotonously increasing *Load Capacity Metric* and figure (b) shows the SLO experiments for the not monotonously increasing *Load Capacity Metric*. Each square represents a possible SLO experiment. Each experiment is configured to either be successful (blue square) or not successful (red square). For the SLO experiments that were not executed by the *UslSearch*, the respective square is transparent. For each resource value, there is one experiment for a load value that corresponds to the capacity according to the *Load Capacity Metric*. This experiment is marked by a black outlined square.

Table 6.1. The expected values of the *Load Capacity Metric* the scenario where it is monotonously increasing (**S1**) and the scenario where it is not monotonously increasing (**S2**).

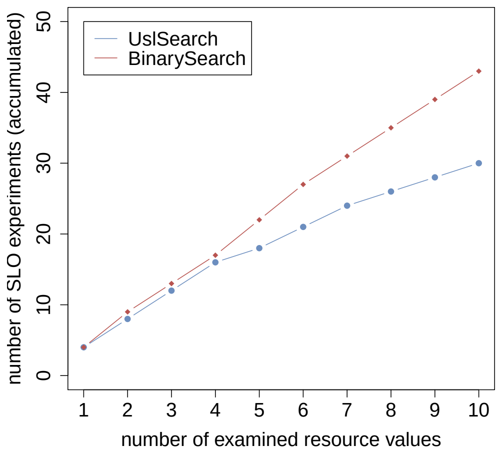
	Resources						
Scenario	1	2	3	4	5	6	7
S1	1	1	3	4	5	5	6
S2	1	2	3	5	3	2	2

6.1.2 Efficiency

Another aspect that we evaluate is the efficiency of the USL online analysis. This involves counting the number of executed SLO experiments for multiple executions of the same benchmark but with different search strategies and comparing the approximated capacity metrics. As a reference, we use the *RestrictionSearch* and *BinarySearch* search strategies.



(a) Number of executed experiments for the *Adapter Service* benchmark **Adapter.Span1.500m** with the *RestrictionSearch* search strategy as a reference.



(b) Number of executed experiments for the *Trace Service* benchmark **Trace.Span1** with the *BinarySearch* search strategy as a reference.

Figure 6.2. Comparison of the *UslSearch* search strategy with selected existing search strategies as a reference. The graph visualizes to the accumulated number of SLO experiments that are executed for with increasing amounts of examined resource values for the respective search strategy. The blue line shows the number of executed SLO experiments of the *UslSearch* search strategy and the red line shows the number of executed SLO experiments for the reference strategy.

Comparison of UslSearch with RestrictionSearch

In the first step, we compare the *UslSearch* search strategy with the *RestrictionSearch* search strategy for the benchmark **Adapter.Span1.500m**. We configure the *RestrictionSearch* to use the *LinearSearch* search strategy and the *LowerBound* restriction strategy. For each resource value, the *LinearSearch* examines the load values in increasing order starting from the smallest load value in the search interval. The search interval is defined by the *LowerBound* restriction. For each resource value, the search interval begins at the largest load value for which the SLO experiment of the next smaller resource value was successful. Therefore, this search strategy assumes that the *Load Capacity Metric* is monotonously increasing for the resource values. For both search strategies, we repeat each SLO experiment three times to increase the statistical meaningfulness.

For both executions, the accumulated number of executed SLO experiments is visualized in Figure 6.2a and the approximated *Load Capacity Metric* is described in Table 6.2. We can see that the total number of executed experiments with the *RestrictionSearch* search strategy is 45. It increases significantly by 14 SLO experiment executions when the 6th resource

Table 6.2. Comparison of the *Load Capacity Metric* values of the *UslSearch*, *RestrictionSearch*, and *BinarySearch* search strategies. The capacity values are underlined if the values differ by more than the distance between two adjacent load values. The capacity values are in bold font if they are equal.

	Adapter.Span1.500m		**Trace.Span1**	
Instances	*UslSearch*	*RestrictionSearch*	*UslSearch*	*BinarySearch*
3	**20 000**	**20 000**	2 800	3 000
6	50 000	40 000	3 800	4 200
9	70 000	60 000	3 800	4 600
12	**90 000**	**90 000**	4 200	4 000
15	110 000	none	4 400	4 200
18	**130 000**	**130 000**	4 000	4 400
21	**140 000**	**140 000**	4 200	4 000
24	150 000	140 000	4 200	3 800
27	**160 000**	**160 000**	4 000	3 800
30	170 000	190 000	**4 000**	**4 000**

value is examined. Looking at the table of the determined capacities, we can see that the reason for this is that for 15 instances, the *RestrictionSearch* is not able to determine the capacity, since the initial SLO experiment for a load of 90 thousand traces per second is not successful and the *RestrictionSearch* assumes a monotonously increasing *Load Capacity Metric*. Therefore, for the next resource value 18 the *RestrictionSearch* examines all load values specified for the benchmark starting from the smallest one. In comparison, when using the *UslSearch*, there are only 29 SLO experiments executed in total. Consequently, the *UslSearch* only requires 64% of the number of executed SLO experiments which corresponds to a speedup of 1.55. However, even if we consider the data point for 15 instances, where the *RestrictionSearch* requires 14 experiments to determine the capacity, as a outlier, the *UslSearch* is more efficient, since the speedup is still $31/27 \approx 1.15$.

Comparison of UslSearch with BinarySearch

Second, we compare the *UslSearch* search strategy with the *BinarySearch* search strategy for the benchmark **Trace.Span1**. For the execution where we use *UslSearch*, we repeat each SLO experiment three times to get statistically more meaningful results. For the execution where we use *BinarySearch*, we expect the number of executed SLO experiments to be much higher. Therefore, we execute each SLO experiment only once, as a tradeoff between execution time and accuracy.

The results of comparing the number of executed SLO experiments are shown in Figure 6.2b. The figure displays the accumulated number of executed SLO experiments in the order that the respective search strategy examines the resource values. We can see that when using *BinarySearch*, we execute four or five SLO experiments for each resource value to determine the capacity. For the ten different resource values, this gives us a total of 43 executed SLO experiments. In contrast, when using the *UslSearch* search strategy, the amount of required experiments is lower. For the instance values 9, 21, 24, and 27, only two SLO experiments are executed to determine the capacity. It is important to note that, for the general case, this is the minimum number of executed SLO experiments that is required to determine the capacity value for any resource value. Therefore, the *UslSearch* search strategy is optimally efficient for four of the ten resource values. In total, when using *UslSearch*, 30 SLO experiments are executed in order to determine the capacities for the ten resource values. This means that the *UslSearch* search strategy only requires about 70% of the number of SLO experiments compared to using *BinarySearch* which corresponds to a speedup of 1.43.

6.1.3 Summary

By looking at the approximated capacities in Table 6.2, we can see that for the benchmark **Adapter.Span1.500m**, where we compared the *UslSearch* with *RestrictionSearch*, the results are more congruent than for the benchmark **Trace.Span1**, where we compared the *UslSearch* with *BinarySearch*. More specifically, for the benchmark **Adapter.Span1.500m**, the approximated capacities differ only for two resource values by more than the distance of two adjacent load values. For the benchmark **Trace.Span1** this is the case for four resource values. Although, for both benchmarks, most of the approximated capacities are the same or the difference is exactly the distance between two adjacent load values at a maximum.

Considering that we repeated each SLO experiment three times (only one time when using *BinarySearch* for the benchmark **Trace.Span1**) these results show a similar approximation result. However, the *UslSearch* outperformed both the *RestrictionSearch* and the *BinarySearch* search strategies when it comes to the total number of executed SLO experiments, notably reducing the total execution time of the respective benchmark. We can conclude that the *UslSearch* search strategy is a efficient alternative to the existing search strategies as it provides accurate results with reduced total execution time. Further, our results give an answer to the open research question of how Theodolite's scalability metrics can be measured in a more efficient way [Henning and Hasselbring 2020].

6.2 Evaluating the USL Offline Analysis Tool

The implementation of the *USL Offline Analysis Tool* is mainly based on the USL R library [Möding 2020]. Therefore, the correctness of the implementation can be mainly reduced to the correctness of the underlying library. In the evaluation of the *USL Offline Analysis Tool* we focus on assessing how the component can be utilized to analyze the benchmark results in terms of the USL. The component builds an USL model based on the provided files that are generated by Theodolite and that contain the benchmark results. The output of the component contains the estimated model parameters α, β, and γ. Given these parameters, it is possible to compare the scalability across different systems. Natively, the *USL Offline Analysis Tool* requires that the benchmarks make use of the *Load Capacity Metric* as scalability metric. However, when using the *Resource Demand Metric*, the metric results can be manually transposed before applying the *USL Offline Analysis Tool*, for cases where both metrics are well-defined. As a result, it is possible to use the *USL Offline Analysis Tool* for benchmarks that were executed in the past and where the scalability was assessed by applying the *Resource Demand Metric*. The tool also provides information about the number of executed SLO experiments per resource value. This allows to assess the efficiency of the used search strategy. We successfully applied the *USL Offline Analysis Tool* based on the results of our benchmarks of ExplorViz, which are discussed in the next section. As a result, we conclude that the *USL Offline Analysis Tool* can provide useful information for evaluating benchmark results based on the USL.

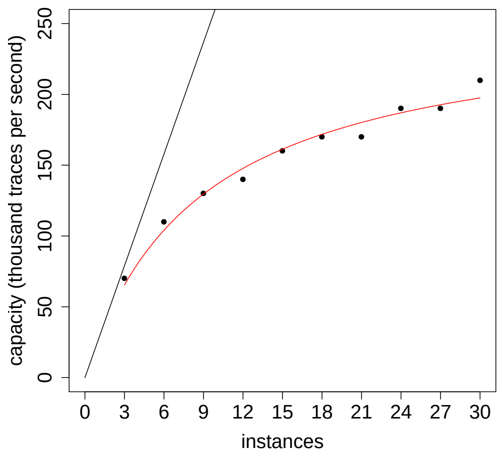
6.3 Scalability Evaluation of ExplorViz

In this section, we discuss the results of benchmarking the scalability of ExplorViz. First we focus on the benchmark categories **C1–C3** where we examined the microservices of the trace analysis individually. Afterwards, we present our findings regarding the benchmark category **C4** where we analyzed the trace analysis as a whole system.

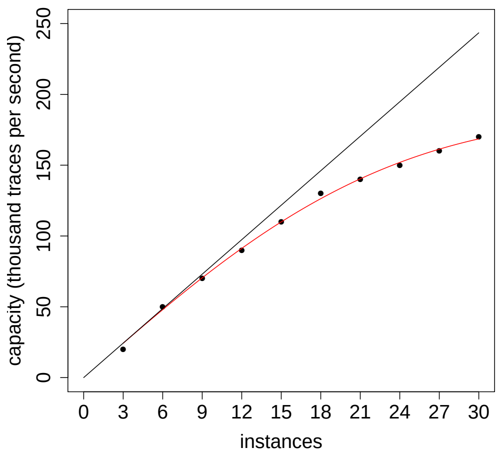
6.3.1 Adapter Service

Results for a Trace Depth of One

The results for the benchmark **Adapter.Span1.2cpu** are depicted in Figure 6.3a. The figure displays the scalability function obtained for the results of the benchmark by applying our *USL Offline Analysis Tool*. In addition, the USL model parameters are summarized in Table 6.3. The model identifies a contention of $\alpha \approx 0.1$. This means that the delay from contention is about 10%. The coherency parameter β is zero. Therefore the model was not able to identify coherency effects due to pairwise synchronization. This suggests, instances of the *Adapter Service* do not require a significant amount of pairwise synchronization. Moreover, read and write operations of the *Adapter Service* with respect to Kafka are not resulting in measurable coherency issues. The predicted throughput for one instance γ, computed for the model, is around 26 thousand traces per second. Table 6.3 also gives information about more key figures of the model: The capacity is predicted to be limited by an asymptotic value of approximately 255 thousand traces per second (Amdahl's asymptote). Thus, regardless of the number of instances, the system is predicted to not be able to process higher loads while fulfilling the SLOs of the benchmark. As we have $\beta = 0$, the approximated capacity metric is not concave and therefore does not have a local maximum. Instead, the capacity is maximized when the number of instances approaches infinity. Another information that can be extracted from the model is the efficiency rate,



(a) Scalability plot for the benchmark with resource limits of 2 CPUs and a trace depth of one **Adapter.Span1.2cpu**. We examined up to 300 thousand traces per second with steps of 10 thousand.



(b) Scalability plot for the benchmark with resource limits of 500 milliCPU and a trace depth of one **Adapter.Span1.500m**. We examined up to 250 thousand traces per second with steps of 10 thousand.

Figure 6.3. Scalability results for the *Adapter Service* for a trace depth of one span. The horizontal axes display the number of instances. The vertical axes display the capacity in traces emitted per second. The data points correspond to the determined capacities. The red curve displays the USL model. The black line corresponds linear scalability with a growth rate of γ.

visualized in Figure 6.4 (red curve). The efficiency rate describes the amount of useful work that is done by one instance [Möding 2020]. It is computed by taking the ratio of the measured capacity and the capacity which would occur in a system that scales linearly with a growth rate of γ. We observe that for increasing amount of instances, the efficiency rate decreases, except for in between 27 and 30 instances, where it slightly increases. A decreasing trend of the efficiency rate is plausible when taking into account the scalability plot from Figure 6.3a, since the scalability curve flattens while the numbers of instances increase, due to diminishing returns.

The scalability function determined by our *USL Offline Analysis Tool* for the benchmark **Adapter.Span1.500m** is visualized in Figure 6.3b and the corresponding model information is contained in Table 6.4. We can see that the contention is estimated to be $\alpha = 0$. Moreover, if we look at the 95% confidence interval for α and compare it to the confidence interval for the benchmark **Adapter.Span1.2cpu** we can see that they do not overlap. Therefore, we can conclude that with high probability, the contention in the system where the resource limits are set to 500 milliCPU is lower than in the system where each instance has limits of 2 CPU cores. Although we were not able to clearly identify the cause of this observation,

Table 6.3. Summary of the USL model parameters for the isolated benchmark of the *Adapter Service* **Adapter.Span1.2cpu** with resource limits of 2 CPUs and a trace depth of one.

Parameter	Value		95% Conf. Int.	
	estimate	error	lower	upper
α	0.10330	0.03376	0.04145	0.16522
β	0.00000	0.00000	-0.00118	0.00118
γ	26 310	3 515	19 865	32 752
scalability limit (Amdahl's asymptote)	254 600			
maximum capacity	not defined			

Table 6.4. Summary of the USL model parameters for the isolated benchmark of the *Adapter Service* **Adapter.Span1.500m** with resource limits of 500 milliCPU and a trace depth of one.

Parameter	Value		95% Conf. Int.	
	estimate	error	lower	upper
α	0.00000	0.00671	-0.01230	0.01230
β	0.00051	0.00015	0.00024	0.00078
γ	8 116	465	7 263	8 968
scalability limit (Amdahl's asymptote)	Infinity			
maximum capacity	181 600 (44.25 instances)			

the results may be explained by the fact that instances that are subject to higher resource limits tend to request a greater fraction of the CPU compared to instances that have lower resource limits. This behavior would result in increased contention with respect to the cluster resources becoming scarce. In order to validate this hypothesis, we observed the CPU utilization during the benchmark execution of randomly chosen SLO experiments. Indeed, this showed that for resource limits of 2 CPUs, the Kubernetes containers of the *Adapter Service* were granted approximately between 1 000 milliCPU and 1 600 milliCPU of CPU resources, while for containers with resource limits of 500 milliCPU, the granted resources never exceeded the limit. Under the assumption that our hypothesis is true, the results of the benchmark **Adapter.Span1.500m** reflect the contention of the *Adapter Service* alone more accurately, as the contention on a cluster-wide level is less significant to the capacity degradation. Looking at the coherency parameter β, we can see that in contrast to the benchmark with resource limits of 2 CPUs, there is a small coherency effect of approximately 0.0005 estimated. For γ, we can see that it is estimated to be around 8100 which is less than one third compared to the value of γ for the benchmark with resource limits of 2 CPUs. We can see the effect when we look at the two figures (a) and (b) in Figure 6.3. For resource limits of 2 CPUs, the initial growth rate is much larger and the

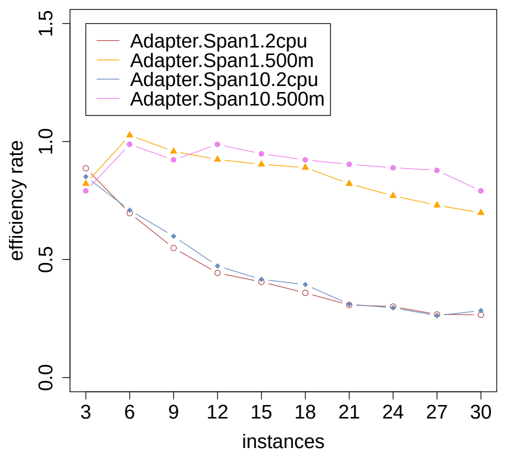


Figure 6.4. Efficiency rates for the benchmarks of the category **C1**. The efficiency rate is defined as the ratio between the measured capacity and the capacity that would occur in a linearly scalable system. It indicates how efficient the system under test is compared to linear scalability.

curve is steeper. However, compared to the black line of linear scalability, the scalability diverges to significantly sublinear due to diminishing returns. For the benchmark with resource limits of 500 milliCPU, the initial growth rate is less but due to no contention, the curve behaves more linearly with respect to the resource dimension. We also executed the benchmark **Adapter.Span1.500m** for resource values between 10 and 80 instances. The results of this variant of the benchmark are shown in Figure 6.5 and Table 6.5. We identified a value for $\alpha \approx 0.00253$, $\beta \approx 0.0002$ and $\gamma \approx 7574$. When compared to the results for resources up to 30 instances, we can see that the contention is now positive and the coherency has decreased. These differences be an indication for that the range of resource values up 30 instances was not sufficient to capture the model parameters accurately.

Results Trace Depth of Ten

For the benchmarks **Adapter.Span10.2cpu** and **Adapter.Span10.500m** the USL model functions are depicted in Figure 6.6a and Figure 6.6b. In general, we observe similar results compared to the benchmarks for a trace depth of one. For the benchmark with resource limits of 2 CPUs, we observe a contention of also $\alpha \approx 0.1$ and coherency of $\beta = 0$ and for the benchmark with resource limits of 500 milliCPU the parameters are estimated to be $\alpha = 0$ and $\beta \approx 0.0003$. Therefore, the value of β is a bit less than for the benchmark with a trace depth of one and resource limits of 500 milliCPU but the values for α are almost

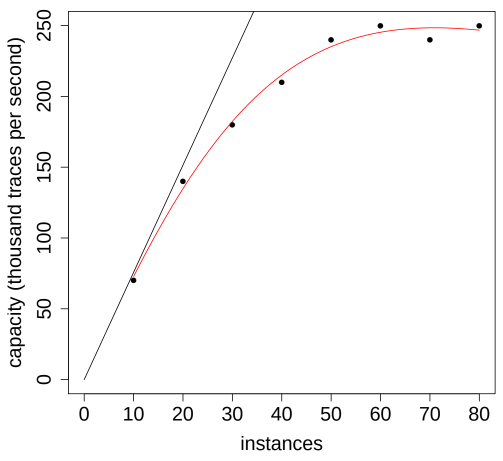
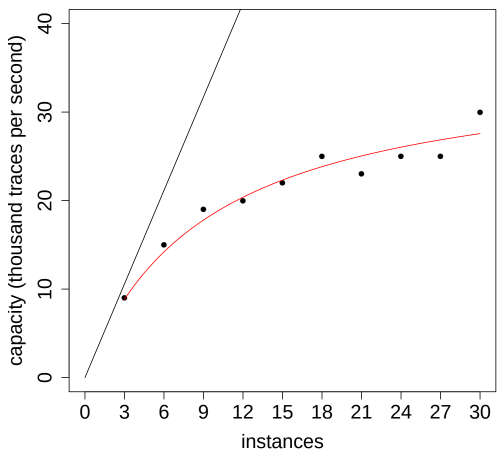


Figure 6.5. Scalability plot for the additional execution of the benchmark with resource limits of 500 milliCPU and a trace depth of ten **Adapter.Span1.500m**. In this benchmark, the range of examined resource values was up to 80 instances, instead of up to 30 instances.

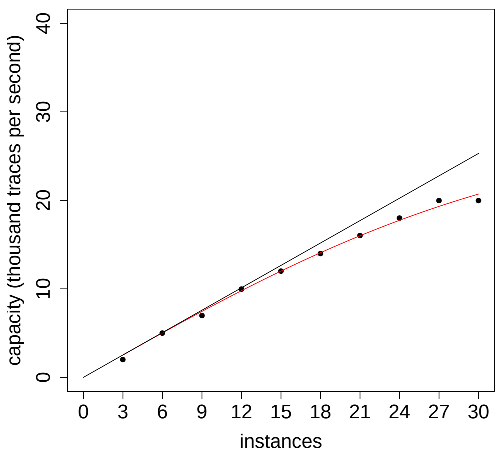
Table 6.5. Summary of the USL model parameters for the benchmark **Adapter.Span1.500m** in the variant where resource values between 10 and 80 instances were examined.

Parameter	Value		95% Conf. Int.	
	estimate	error	lower	upper
α	0.00253	0.00457	-0.00613	0.01119
β	0.00020	0.00003	0.00014	0.00026
γ	7 574	692	6 263	8 885
scalability limit (Amdahl's asymptote)	2 990 000			
maximum capacity	248 400 (70.86 instances)			

the same as for the benchmarks for a trace depth of one. The main difference becomes obvious when looking at the parameter γ. We can see that the initial growth rate is slightly more than ten times less than for the benchmarks with a trace depth of one. More exactly, for the benchmark **Adapter.Span10.2cpu**, $\gamma \approx 3\,500$ is estimated which is is about 7.5% compared to the rate for the benchmark with a trace depth of one and 2 CPU cores. For the benchmark **Adapter.Span10.500m** we have $\gamma \approx 840$ traces per second which is about 9.6% of the value for a trace depth of one. These results indicate that increasing the trace depth results in approximately proportionally reduced initial growth rate but does not significantly effect the contention and coherency.



(a) Scalability plot for the benchmark with resource limits of 2 CPUs and a trace depth of ten **Adapter.Span10.2cpu**. We examined up to 35 thousand traces per second with steps of 1 thousand.



(b) Scalability plot for the benchmark with resource limits of 500 milliCPU and a trace depth of ten **Adapter.Span10.500m**. We examined up to 30 thousand traces per second with steps of 1 thousand.

Figure 6.6. Scalability results for the *Adapter Service* for a trace depth of ten spans. The horizontal axes display the number of instances. The vertical axes display the capacity in traces emitted per second. The data points correspond to the determined capacities. The red curve displays the USL model. The black line corresponds linear scalability with a growth rate of γ.

Table 6.6. Summary of the USL model parameters for the isolated benchmark of the *Adapter Service* **Adapter.Span10.2cpu** with resource limits of 2 CPUs and a trace depth of ten.

Parameter	Value		95% Conf. Int.	
	estimate	error	lower	upper
α	0.09777	0.05075	0.00474	0.19080
β	0.00000	0.00097	-0.00178	0.00178
γ	3 524	728	2 188	4 860
scalability limit (Amdahl's asymptote)	36 050			
maximum capacity	not defined			

Discussion

Summarizing the benchmarking results of all benchmarks of the category **C1**, we can conclude that we observe contention and coherency. However, the increase in contention

Table 6.7. Summary of the USL model parameters for the isolated benchmark of the *Adapter Service* **Adapter.Span10.500m** with resource limits of 500 milliCPU and a trace depth of ten.

Parameter	Value		95% Conf. Int.	
	estimate	error	lower	upper
α	0.00000	0.01072	-0.01965	0.01965
β	0.00025	0.00025	-0.00020	0.00071
γ	843	80	696	991
scalability limit (Amdahl's asymptote)	Infinity			
maximum capacity	26 640 (62.66 instances)			

Table 6.8. Summary of the USL model parameters for the benchmark of the *Landscape Service* **Landscape.500m.RandomMethodNames** where all spans were generated for different methods, to avoid cache hits.

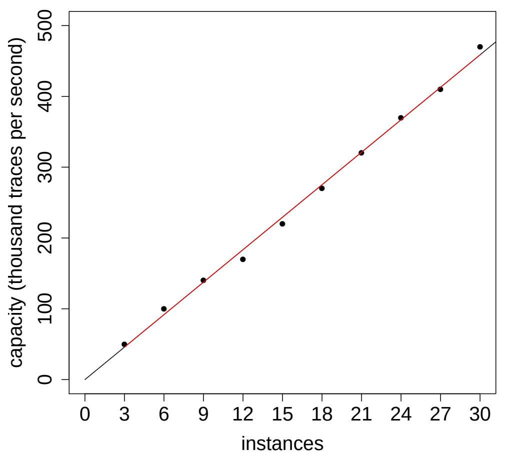
Parameter	Value		95% Conf. Int.	
	estimate	error	lower	upper
α	0.00000	0.00953	-0.01747	0.01747
β	0.00000	0.00023	-0.00042	0.00042
γ	15 290	1 364	12 790	17 790
scalability limit (Amdahl's asymptote)	Infinity			
maximum capacity	not defined			

for the benchmarks with CPU resources of 2 CPU cores, compared to the benchmarks with resources of 500 milliCPU, seems to be related to the clusters resources becoming scarce and not mainly caused by complete saturation of the *Adapter Service*. In addition, analyzing different trace depths leads us to the observation that an increase in the trace depth results in a proportionally decreased capacity. We can conclude that it should be payed attention to the clusters capacity, when granting CPU resources to containers, as increasing the CPU resources can lead to significant contention. However, for practical use cases it might be appropriate to balance the scalability with economical cost that arise from increasing the CPU limits or the number of instances.

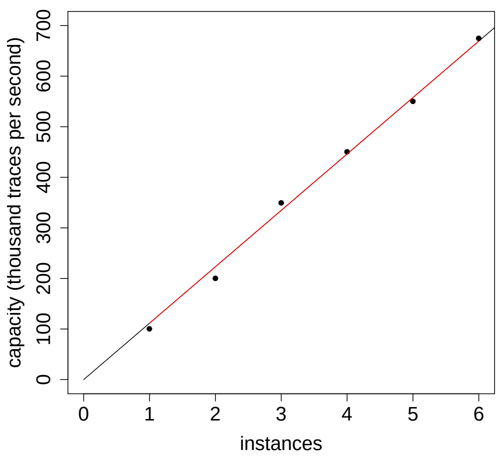
6.3.2 Landscape Service

Results

The model parameters of the benchmark **Landscape.500m.RandomMethodNames** are shown in Table 6.8. In this benchmark we avoided cache hits for spans in the *Landscape Service* by generating random method names. What stands out is that our results identify



(a) Scalability plot of the USL model for the benchmark **Landscape.500m.RandomMethodNames**. The examined number of traces per second was between 30 thousand and 520 thousand with steps of 10 thousand.



(b) Scalability plot of the USL model for the benchmark **Landscape.500m.ConstantMethodNames**. The examined number of traces per second was between 100 thousand and 700 thousand with steps of 25 thousand.

Figure 6.7. Scalability plots for benchmarking the *Landscape Service* in isolation. The horizontal axes display the number of instances of the *Landscape Service*. The vertical axes display the number of traces emitted per second with the load generator. The data points correspond to the capacity determined for the different numbers of instances.

Table 6.9. Summary of the USL model parameters for the benchmark **Landscape.500m.ConstantMethodNames** where all spans were generated for the same method.

Parameter	Value		95% Conf. Int.	
	estimate	error	lower	upper
α	0.00000	0.07762	-0.15641	0.15641
β	0.00000	0.01014	-0.02043	0.02043
γ	111 538	11 794	87 773	135 304
scalability limit (Amdahl's asymptote)	Infinity			
maximum capacity	not defined			

true linear scalability of the *Landscape Service*. This can be seen by that fact we have $\alpha = \beta = 0$. Therefore, Amdahl's asymptote is not defined and there is also no local maximum of the *Load Capacity Metric*. Looking at the plot of the model in Figure 6.7a we

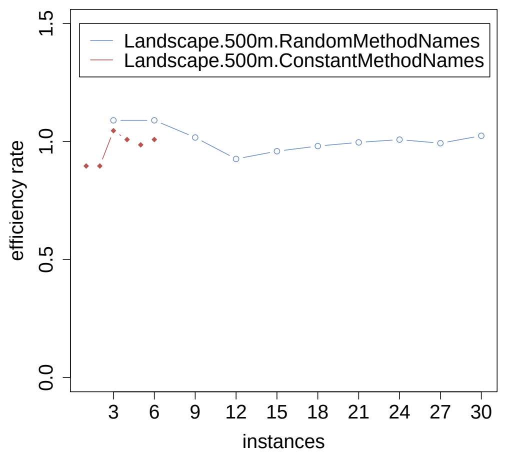


Figure 6.8. Efficiency rates for the benchmarks of the category **C2**.

can visually confirm the results, since the red and black line overlap. Since our model implies true linear scalability, we would expect the efficiency rate to equal one for all numbers of instances. However, when looking at the efficiency rate in Figure 6.8 we can see that the rate is very close to one, but not exactly one. This deviations can be explained by the model predicting a different capacity value than measured in our experiments. Based on these results, we conclude that the *Landscape Service* is linearly scalable for the estimated range of instances. However, there may be the point where either the hardware becomes contended when the limits of the cluster are reached, or when services of the environment, for example, Kafka or the Cassandra database become fully saturated. Still, in our benchmarks, the examined range of of instance values was not large enough to allow the USL model to identify this point, since we were not able to generate load with enough magnitude.

The results of the benchmark **Landscape.500m.ConstantMethodNames** are visualized in Figure 6.7b and the model parameters are contained in Table 6.9. Similar to the results for the other benchmark of the *Landscape Service*, we have the model parameters $\alpha = \beta = 0$ which indicates linear scalability. However, we observe that the rate γ is significantly larger for the benchmark where all the spans we generated for the same method. More precisely, we have $\gamma \approx 112\,000$ compared to $\gamma \approx 15\,000$ where the method names were generated randomly. This behavior is plausible since in the benchmark **Landscape.500m.ConstantMethodNames** all generated traces consist of spans for the same method. As a result, in each instance of the *Landscape Service*, the Cassandra database is accessed only once and all consecutive calls to the method result in a cache hit and no database query.

Table 6.10. Summary of the USL model parameters for the benchmark **Trace.Span1** for a trace depth of one span.

Parameter	Value		95% Conf. Int.	
	estimate	error	lower	upper
α	0.25046	0.04834	0.16185	0.33908
β	0.00285	0.00072	0.00153	0.00417
γ	1 430	150	1 155	1 705
scalability limit (Amdahl's asymptote)	5 710			
maximum capacity	4 206 (16.22 instances)			

Discussion

Summarizing, we can see that the *Landscape Service* shows clear indications of being linearly scalable. However, in our specific execution environment, the Kafka cluster was not able to handle higher throughput values than around 700 thousand messages per second. Therefore, we were only able to determine the capacity for up to 30 instances for the benchmark **Landscape.500m.RandomMethodNames** and up to six instances for the benchmark **Landscape.500m.ConstantMethodNames**. As a result, there may be contention or coherency issues that would only become measurable at higher instance values but are not detected by our USL models.

6.3.3 Trace Service

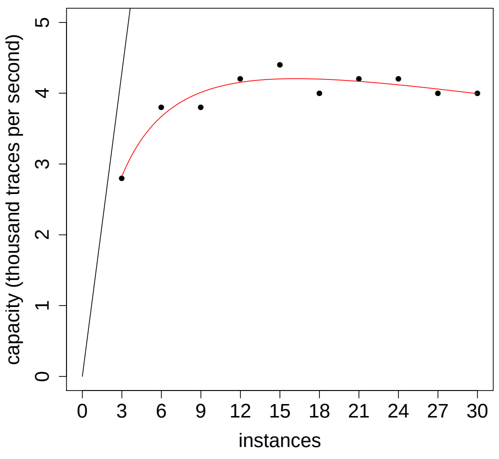
Results

We conducted two benchmarks for the *Trace Service*. In the first benchmark **Trace.Span1** we examined traces with a depth of one span. The resulting USL model is visualized in Figure 6.9a and the model parameters are summarized in Table 6.10. The α parameter of the USL model is estimated to be approximately 0.25 and β is estimated to be about 0.003. The initial growth rate γ is estimated by a value of approximately 1 400. According to the model, the maximum capacity is reached at around 4 200 traces per second with approximately 16 instances. Taking the visualization into consideration we can see that increasing the instances beyond that point does not result in a noticeably increased capacity. Since, we have $\beta > 0$, the model even estimates that the capacity decreases slightly for more than 16 instances.

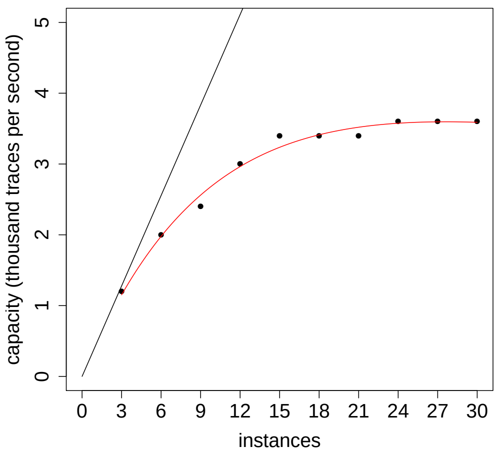
In the second benchmark for the *Trace Service* **Trace.Span10**, we examined traces with a depth of ten spans. The corresponding results are visualized in Figure 6.9b and the model parameters are described in Table 6.11. Compared to the benchmark with a trace depth of one, the model parameter $\alpha \approx 0.05$ is approximately 5 times smaller. Also, the parameters $\beta \approx 0.001$ and $\gamma \approx 400$ are significantly smaller. This leads to a more flattened curve

Table 6.11. Summary of the USL model parameters for the benchmark **Trace.Span10** for a trace depth of ten spans.

Parameter	Value		95% Conf. Int.	
	estimate	error	lower	upper
α	0.05089	0.01707	0.01960	0.08218
β	0.00125	0.00032	0.00066	0.00183
γ	426	37	359	493
scalability limit (Amdahl's asymptote)	8 369			
maximum capacity	3 596 (27.6 instances)			



(a) Scalability plot of the benchmark **Trace.Span1** for traces with a depth of one span. The examined number of traces per second was between 1 200 and 5 000 with steps of 200.



(b) Scalability plot of the benchmark **Trace.Span10** for traces with a depth of ten spans. The examined number of traces per second was between 1 000 and 5 000 with steps of 200.

Figure 6.9. Scalability plot of the USL model for the *Trace Service*. The vertical axes display the capacity in thousand traces per second. The horizontal axes display the number of instances of the *Landscape Service*.

of the capacity metric. Based on these results, we observe that an increased trace depth generally decreases the capacity measured in traces per second. However, the different maximum capacities are still of similar magnitude, i.e., around 4 200 traces per second for the benchmark **Trace.Span1** and to around 3 600 traces per second for the benchmark **Trace.Span10**.

If we look at the efficiency rates, visualized in Figure 6.10, we can observe an decreasing trend for both benchmarks. However, the efficiency is higher for the benchmark **Trace.Span10**. This is caused by the fact that the initial growth rate γ of the hypothetical linearly scalable system is higher for the benchmark **Trace.Span1**, while the measured capacities are roughly the same for both benchmarks. As a result, the instances in the benchmark **Trace.Span10** are more efficient when compared to a linear scalable system.

Discussion

The reasons for bounded capacity are likely the stateful operations performed by the *Trace Service*. The stream processing is based on a sliding time window that collects spans over the last ten seconds. After the ten seconds have passed, there is an additional grace period of two seconds, after which the window is considered final and all the contained spans are composed to the traces that they belong to. Spans are dropped if they are processed after the time window for the trace has ended. In the experiments we conducted, we observed that unsuccessful SLO experiments were mainly caused by the *dropped records ratio* SLO being violated. This can be explained as follows:

In our benchmarks, we observed that the consumer lag does not drop below a certain value, regardless of how many instances we have. We know that during the stream processing, messages are buffered in multiple situations [Kreps et al. 2011; Wang et al. 2021]. This may be one reason why, even if its trend is constant, we can observe some fluctuations in the consumer lag. Moreover, the intensity of these fluctuations seems to be correlated with the intensity of the load. An example for this can be seen in Figure 6.11. The figure displays the consumer lag for two SLO experiments from the benchmark **Trace.Span1**

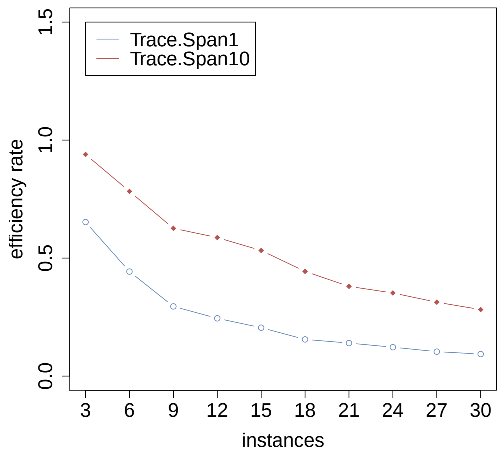


Figure 6.10. Efficiency rates for the benchmarks of the category **C3**.

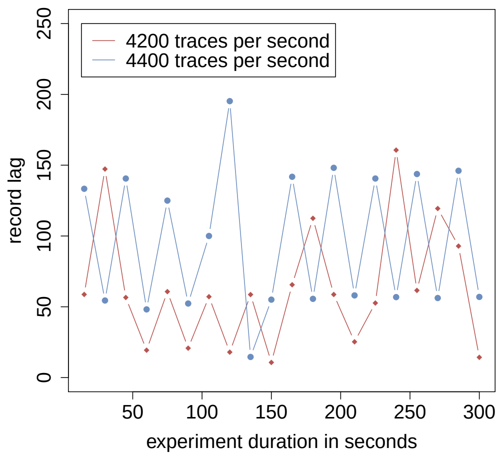


Figure 6.11. Consumer lag over the duration of an SLO experiment for 21 instances of the trace service. The red line corresponds to the successful SLO experiment for a load of 4 200 traces per second. The blue line corresponds to the unsuccessful SLO experiment for the next larger load value of 4 400 traces per second. Both SLO experiments are taken from the benchmark results of the benchmark **Trace.Span1**.

with different load values (4 200 and 4 400 traces per second) but with the same amount of resources (21 instances). In the SLO experiment with a load of 4 200 traces per second, we did not get enough dropped records to violate the *dropped records ratio* SLO. In contrast, for a load of 4, 400 traces per second, the *dropped records ratio* SLO was violated. We can see that the individual peaks of the consumer lag are higher for a load of 4 400 traces per second, compared to only a load of only 4 200 traces per second. This can explain the occurrence of a larger amount of dropped records, since a larger consumer lag can make more significant fraction of the messages in the topic arrive too late at the *Trace Service*. Consequently, if the load is increased to a value more than around 4 000 traces per second (depending on the benchmark), there may be the point where due to the fluctuations in the consumer lag some of the messages arrive too late at the *Trace Service* and as a result, we observe a sufficient amount of dropped records to violate the *dropped records* SLO.

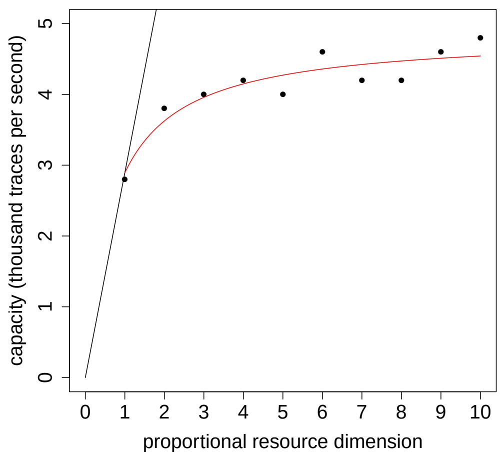
Another aspect that may be cause the observation of dropped records is that due to the task scheduling in Kafka Streams, the Kafka partitions may not be processed evenly during the execution of one SLO experiment. In our benchmarks for the *Trace Service*, we configured each of the Kafka topics to use 250 partitions. As a result, each of the maximum 30 instances of the *Trace Service* has to switch between the partitions that are assigned to the instances during the processing. While most of the messages may be processed within a reasonable amount of time, there may be few partitions that are not processed for a longer period of time. This may led to the occurrence of dropped records, even if the consumer lag does not increase significantly. However, since Kafka Streams tries to evenly assign the partitions to the stream processing applications, this effect would diminish if the number of partitions and the number of stream processing instances are the same (assuming one Kafka Streams thread per instance). As a result, it may be reasonable to repeat the respective experiments in a configuration, where the number of partitions is the same as the number of instances of the *Trace Service*.

We can summarize the benchmark results regarding the *Trace Service* as follows: Regardless of the trace depth, the scalability is bounded to a capacity of around 4 000 traces per second. For the benchmark **Trace.Span1** we have a maximum of approximately 4 200 traces per second which is slightly larger than around 3 600 traces per second for the benchmark **Trace.Span10**. This is plausible since a larger trace depth comes with increased computational complexity within the stream processing applications. Considering this, it is also not surprising that the initial growth rate γ is larger for the benchmark with a trace depth of one, compared to the benchmark with a trace depth of ten. Addressing the contention parameter α for both benchmarks, we observe that it is five times larger ($\alpha \approx 0.25$) for the benchmark **Trace.Span1** compared to the $\alpha \approx 0.05$ for benchmark **Trace.Span1**. However, as mentioned in Section 3.1.1, the usage of the *dropped records* SLO implies that the model parameters of the USL not exactly represent the physical properties of the system under test but also indicate how the fulfillment of the SLOs is affected by increasing the resources, in our case, the number of instances of the *Trace Service*. The high values for α indicate, that the capacity gain of increasing the number of instances of the *Trace Service* diminishes fast. But due to the measurement method, this may be related to the SLO becoming violated because of the occurrence of dropped records due to buffering or the varying speed at which the individual partitions are processed, instead of the access of the *Trace Service* instances to shared resources.

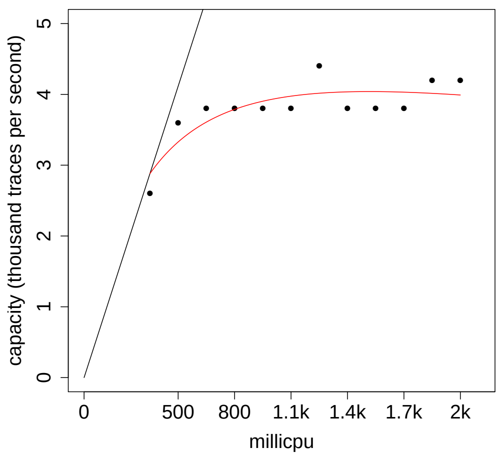
6.3.4 Trace Analysis as a System

Results

We examined the trace analysis of ExplorViz as a whole system in two benchmarks. For both benchmarks all generated traces had a depth of one. The USL model of the benchmark **TraceAnalysis.ProportionalResources** where we used the proportional resource dimension is depicted in Figure 6.12a and the corresponding summary of the model parameters can be found in Table 6.12. In general, we can see that the scalability is bounded by a capacity of around 4 000 to 5 000 traces per second. The USL predicts the limit to be at approximately 4 800 traces per second. Moreover, according to the model, the system shows significant contention, as we have $\alpha \approx 0.6$. However, the coherency is estimated to be $\beta = 0$. In general, the capacity is bounded to a similar value compared to the benchmarks where we examined the Trace service in an isolated manner. We conclude that it likely is the *Trace Service* which causes the capacity bound.



(a) Scalability plot of the benchmark of the whole Trace analysis with the proportional resource dimension **TraceAnalysis.ProportionalResources**. The examined number of traces per second was between 1 200 and 5 000 with steps of 200.



(b) Plot of the *Load Capacity Metric* for the benchmark of the whole Trace analysis with the CPU dimension **TraceAnalysis.Cpu**. The system consisted of six instances of the *Adapter Service*, three instances of the *Landscape Service*, and nine instances of the *Trace Service*. Before the USL was applied the resource dimension *milliCPU* was normalized such that 350 milliCPU correspond to $N = 1$. The examined number of traces per second was between 1 200 and 5 000 with steps of 200.

Figure 6.12. Scalability plot for the benchmarks of the whole trace analysis of ExplorViz. The vertical axes display the capacity in thousand traces per second and the horizontal axes display the number of instances, or the amount of milliCPU respectively.

The USL model of the benchmark **TraceAnalysis.Cpu** is visualized in Figure 6.12b. Before applying the *USL Offline Analysis Tool* there was one extra preprocessing step we conducted. That is, we normalized the resource dimension *milliCPU*, such that, 350 milliCPU correspond to $N = 1$ in terms of the USL. The reason for this is that the USL assumes that $N = 1$ is the minimal meaningful degree of concurrency. However, one milliCPU is not a meaningful value, since the services of our system under test require at least 350 milliCPU to start without an extensive amount of delay, or to even start at all. Consequently, we assumed that the minimal meaningful degree of concurrency is 350 milliCPU. The estimated parameter values are described in Table 6.13. We have $\alpha \approx 0.51$, $\beta \approx 0.023$, and $\gamma \approx 2881$. This leads to the observation that the amount of CPU resources has a very small impact on the capacity, since the capacity gain from increasing the amount of milliCPU diminishes fast. However, also for this benchmark the scalability is heavily bounded, likely by the *Trace Service*. When we look at the efficiency rates, visualized

Table 6.12. Summary of the USL model parameters for the benchmark of the whole trace analysis with proportional resource dimension **TraceAnalysis.ProportionalResources**.

Parameter	Value		95% Conf. Int.	
	estimate	error	lower	upper
α	0.59685	0.09872	0.41588	0.77782
β	0.00000	0.00771	-0.01413	0.01413
γ	2 894	208	2 513	3 275
scalability limit (Amdahl's asymptote)	4 849			
maximum capacity	not defined			

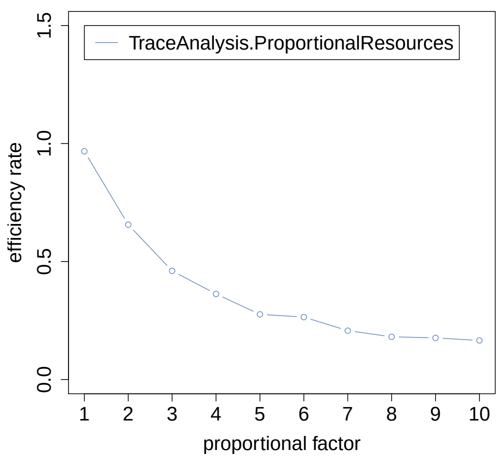
Table 6.13. Summary of the USL model parameters for the benchmark of the whole trace analysis with CPU resource dimension **TraceAnalysis.Cpu**.

Parameter	Value		95% Conf. Int.	
	estimate	error	lower	upper
α	0.51483	0.15045	0.24465	0.78502
β	0.02588	0.02254	-0.01460	0.06635
γ	2 881	214	2 496	3 265
scalability limit (Amdahl's asymptote)	5 595			
maximum capacity	4 040 (1 515.5 milliCPU)			

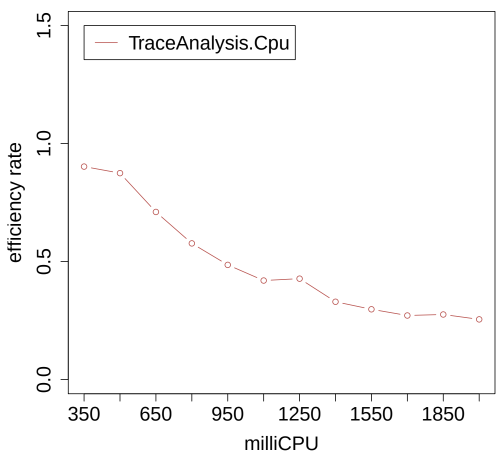
in Figure 6.13a for the benchmark **Trace.Span1** and in Figure 6.13b for the benchmark **Trace.Span10**, we can observe a decreasing trend for both benchmarks. As a result, for both benchmarks the systems becomes less efficient when compared to a linear scalable system, if the resources are increased.

Discussion

As a conclusion, when examining the trace analysis as a whole system, we can observe similar results compared to the benchmarks of the benchmark category **C3** where we benchmarked the *Trace Service* in isolation. We can observe that regardless of scaling the number of instances of the microservices or the amount of milliCPU granted as resource limits, the scalability is bounded by a capacity not more that around 4 800 traces per second for a trace depth of one. This value is similar to the identified maximum capacity the benchmarks of the category **C3**. Looking at the estimated model parameters, we can see that the contention α for the benchmarks is estimated to be between approximately 0.51 and 0.60 which is significantly more than in the results of the benchmarks in category **C3**. However, we have to recall the definition of the resource dimensions: Since the scalability



(a) Efficiency rate for the benchmark **TraceAnalysis.ProportionalResources** in the benchmark category **C4**. The actual amount of instances for the *Adapter Service* ($I_A = 2$), *Landscape Service* ($I_L = 1$), and *Trace Service* (I_T) is determined by multiplying the proportional factor with (I_A, I_L, I_T).



(b) Efficiency rate for the benchmark **TraceAnalysis.Cpu**. The vertical axis shows the efficiency rate and the horizontal axis displays the amount of milliCPU that was granted as resource limits for each of the instances of the microservices. The system consisted of six instances of the *Adapter Service*, three instances of the *Landscape Service*, and nine instances of the *Trace Service*.

Figure 6.13. Efficiency rates for the benchmarks of the benchmark category **C4**.

mainly seems to be bounded by the capacity of the *Trace Service*, the *Adapter Service* and the *Landscape Service* are not the limiting factor. However, due to the fact that in the case of benchmark **TraceAnalysis.ProportionalResources**, a resource value of one corresponds to three instances of the *Trace Service*, the contention is estimated higher than for the benchmark **Trace.Span1**, since the scalability becomes sublinear more quickly, although the actual data points are similar. A similar argument applies to the benchmark **TraceAnalysis.Cpu**. For this benchmark, we normalized the resource dimension such that $N = 1$ corresponds to 350 milliCPU, which was the minimum meaningful amount of resources that we could grant to the instances of the microservices. Considering a microservices system where all service show similar scalability results, we may observe that it is not always the same microservices that becomes the limiting factor. As a result, the normalization of the resource dimension would not distort the estimated parameters as much as in our results, where the scalability was always limited by the *Trace Service*.

6.3.5 Summarizing the Benchmark Results

We can summarize the results of benchmarking the scalability of ExplorViz as follows. When examining the *Adapter Service* in isolation we observe sublinear scalability because of a small contention and coherency. As a result, the scalability is bounded to around 250 thousand traces per second for a trace depth of one and to around 30 thousand traces per second for a trace depth of ten. In contrast, the *Landscape Service* is linearly scalable in our benchmarks, meaning the amount of load it can handle is proportional to the number of instances. During our benchmarks, this enabled us to process loads up to around 700 thousand traces per second. When looking at the scalability results of the *Trace Service*, we observe that its scalability is bounded because of the occurrence of *dropped records* for loads more than around 4 000 traces per second. When examining the trace analysis of ExplorViz as a whole system, we can confirm the results regarding the individual microservices and we observe that the scalability is bounded by the *Trace Service*.

Considering our findings with regard to benchmarking the scalability of ExplorViz, we can conclude that overall our approach with applying the USL as part of the methodology of Theodolite is convenient. However, there are some restrictions.

The formulation of the USL assumes that the system under test is utilized at 100% and that the throughput is only significantly limited by contention and coherency effects, which allows a physical interpretation of the model parameters. According to our findings in Chapter 3, using the *lag trend* SLO exclusively makes the *Load Capacity Metric* approximate the throughput. This allows a meaningful interpretation of the USL models for the benchmark results of the *Adapter Service* and the *Landscape Service*, since we only used the *lag trend* SLO. However, other SLOs including the *dropped records* SLO can be violated for smaller loads even if the throughput is not reached. As a result, the USL model parameter's meaning changes. That is, α may not only capture the serial fraction of the workload and β not only the overhead from synchronization among the processors. Instead, unutilized resources would be reflected in the model parameters. However, when applying the USL to these benchmark results, we did not see significant deviations between the measured data points and the USL. That is, the USL seems to still fit our results. Considering this, the values of the model parameters still give a relative characterization of the systems scalability since the higher the value of α, the faster the scalability becomes sublinear, i.e., the faster Amdahl's asymptote is approached. Respectively, the higher the value of β, the faster it becomes retrograde. Therefore, the USL can still be used to compare the scalability of different systems in these cases.

6.4 Threats to Validity

In this section, we address the internal and external threats to the validity of our evaluation.

6.4.1 Internal Threats

Selection of Reference Search Strategies

Addressing the comparison of the *UslSearch* search strategy with the existing search strategies, our results show that the *UslSearch* provides similar results even when executing less SLO experiments compared to the reference search strategies. We did not compare the *UslSearch* with all the existing search strategies, though. As a result, it is possible that the *UslSearch* would be outperformed by one of the remaining existing search strategies or in a benchmark where the *Load Capacity Metric* behaves differently.

Repetitions of SLO Experiments

It should be noted that we repeated most of our experiments three times. However, individual executions of the same SLO experiment may lead to different results. As a consequence, the results would become statistically more meaningful if we repeated the SLO experiments more often. However, this would result in a longer execution time of the benchmarks.

Sensitivity of SLOs for Less Intense Loads

In practice, the metrics that are used for computing the SLOs are affected by implementation details. One example for this is that during stream processing, the stream processing applications request multiple messages from the topics up to multiple hundreds of kilobytes at a time [Kreps et al. 2011]. Also, the stream processing applications periodically indicate the position of the last processed message in the ordered stream of messages in the topic [Wang et al. 2021]. This buffering during the stream processing can lead to unexpected measurements. For example, an increase in the consumer lag may be indicated before the offset of the most recent batch of messages is committed by a Kafka consumer. When observed over a longer period of time, however, the consumer lag follows a constant trend. As the buffer size is not proportional to the frequency with which the message arrive, our SLOs become more sensitive for volatile measurements for small load values. Therefore, in these scenarios, the *lag trend* SLOs may more often indicate a false assessment of the system under test being able to sustain the load.

Load and Resource Dimension Range and Resolution

With regard to our benchmark results, for all our determined USL models, the model parameters only identify estimates based on the examined ranges of the resource dimension.

As a result, our models may not be able to get the full picture and executing benchmarks for additional resource values may lead to different results. Further, the resolution of the load values impacts the accuracy of the model parameters that are estimated during the regression of the USL analysis.

6.4.2 External Threats

Specific Execution Environment

It should be noted that the results of our experiments only allow to assess the scalability of ExplorViz with regard to our specific execution environment. The results of the benchmark category **C1** where we examined the *Adapter Service* is exemplary for this since the estimated parameter values seem to be influenced by the cluster utilization. Moreover, the determined USL models do not allow to differentiate between multiple reasons that result in the parameter estimations. Hence, the USL models assess the scalability from a perspective where the system under test and the execution environment constitute one black box system. Consequently, the identification of the causes for the observations has to be done by analyzing the experiment results and by assessing the results of similar benchmarks.

Usage of Underlying Metrics

The identified USL models are based on the underlying metrics, i.e., on the JMX and Prometheus metrics that are used to compute the SLOs. However, the corresponding measurements often only provide an estimation of the real system state. Therefore, one central assumption of our approach is that the measurements are accurate enough to assess the scalability.

Chapter 7

Related Work

The current architecture of ExplorViz has not been evaluated in terms of scalability so far. Thus, the USL has also not been applied to ExplorViz yet. However, we identified some publications that cover approaches of integrating the USL with scalability benchmarking tools or that deal with the question how it can be applied to stream processing systems.

Heyman et al. [2014] present the *Scalar* distributed load generation and benchmarking platform that integrates an analysis of the scalability of the system under test based on USL. In contrast to Theodolite, which focuses on stream processing and cloud-native applications, the platform focuses on web applications. The platform is configurable via plugins that are used to orchestrate the system under test and manage the system state between individual experiments. The Scalar platform also provides a plugin that allows to analyze the experiment results in terms of the USL based on specified quality of service policies similar to SLOs. However, the USL is only used for offline analysis of the benchmark results and not for making the benchmark execution more efficient. Another plugin that is provided is responsible for both monitoring the usage of CPU resources and the network. In the current version, Theodolite does not monitor these resources. However, it may be useful to integrate similar capabilities to Theodolite with the aim to allow a better verification of the benchmarking process.

Vikash et al. [2020] use the USL to benchmark the scalability of different distributed stream processing systems in the context of Internet of Things (IoT) applications. In this regard, IoT sensors are simulated based on two datasets. The first simulates a forest monitoring system to detect hotspots and fire locations via sensors and the second consists of heterogeneous health care and patient treatment data that is labelled by time. In the benchmarks, the startup-time, the response-time, and the throughput are examined. For the measured throughput values, the systems scalability is analyzed with the USL. Although the measurement method is not described in detail, the throughput seems not to be determined in multiple experiments for discrete load values as, in the methodology of Theodolite. Instead, the throughput is maximized regardless of any SLOs to ensure full utilization of the stream processing system.

Chantzialexiou et al. [2018] introduce the *PilotStreaming* stream processing framework that provides a unified abstraction layer for the resource management of multiple execution platforms including high-performance computing (HPC) and serverless environments. To allow a better understanding of the performance characteristics of different deployment options in PilotStreaming, the *StreamInsight* framework is introduced [Luckow and Jha

2019]. This framework allows scalability analysis and prediction for different platforms and configurations. It consists of two components: The first component is responsible for data collection and the second component provides an analysis of the examined system with the USL. The used capacity metric is based on the definition of sustainable throughput [Karimov et al. 2018] which similarly to the approach from Theodolite considers the backpressure within the stream processing system when determining the capacity. Yet again, there are no SLOs specified and it is not further mentioned in detail how the capacity is measured. Moreover, it is not considered how the used measurement method based on the sustainable throughput may influence the interpretability of the resulting USL models with regard to the physical meaning of the model parameters.

Chapter 8

Conclusions and Future Work

In this chapter, we will summarize our findings and describe future research subjects.

8.1 Conclusions

The overall goal of our work was to benchmark the scalability of the software visualization and comprehension tool ExplorViz in terms of the USL performance model with the cloud-native benchmarking tool Theodolite. Within this aim, we identified three subordinate goals: First we aimed to apply the USL in scalability benchmarking of distributed stream processing applications in cloud environments. Based on this, our second goal was to extend the Theodolite benchmarking framework with regard to USL analysis and by various extensions that were prerequisite for benchmarking ExplorViz. Third, we planned to benchmark the scalability of ExplorViz′ trace analysis with our extended version of Theodolite.

With regard to our first goal, we identified that the USL can be applied to stream processing applications in the context of the scalability definition of Theodolite. However, the USL model parameters may not have an exact physical meaning if the *Load Capacity Metric* does not approximate the throughput of the system under test. Instead, they describe how adding resources impacts the fulfillment of the SLOs. With regard to the methodology of Theodolite, we identified that the USL can be used to reduce the total execution time of benchmarks when combined with the heuristic benchmark execution of Theodolite. In addition, the USL can be used to analyze the benchmark results in terms of scalability.

Based on these findings, we were able to address the second goal. We extended the implementation of heuristic execution in Theodolite by providing an additional search strategy that utilizes capacity predictions based on the USL in order to reduce the total execution time of benchmarks. In our evaluation, we compared our extension to the heuristic benchmark execution in form of a new search strategy with two of the existing search strategies. The results indicate that the new search strategy is more efficient than the existing search strategies while providing a comparable accuracy as the *LinearSearch* and *BinarySearch* search strategies and even more accuracy than the *RestrictionSearch* search strategy. We also implemented a tool that allows scalability analysis of the benchmark results in terms of the USL. The tool generates key figures and a visualization of the USL models. We used this tool in the analysis of the results when benchmarking ExplorViz and validated that it provides useful information. In preparation of executing our benchmarks, we also improved the Theodolite implementation in various ways including the migration of the default Kafka implementation and enabling the *Theodolite Operator* to delete arbitrary Kubernetes resources between individual experiments during the benchmarking.

Finally, we addressed our third goal. We used our extended Theodolite implementation for benchmarking the scalability of ExplorViz. In our benchmarks, we focused on the trace analysis within ExplorViz. We examined each of the microservices *Adapter Service*, *Landscape Service*, and *Trace Service* individually in multiple benchmarks as well as the trace analysis as a whole system. Our results show that the scalability is mainly bounded by the *Trace Service*, which was only able to process approximately between 3 600 and 4 800 traces per second at a maximum depending on the concrete configuration. However, the *Adapter Service* shows much higher capacities up to almost 300 thousand traces per second depending on the complexity of the generated load. The *Landscape Service* showed indications of being linearly scalable and we were able to process loads up to 700 thousand traces per second. When we benchmarked the trace analysis consisting of the three microservices as a whole system, we were able to confirm the results of benchmarking the microservices in isolation, meaning our results indicate that the *Trace Service* bounds the scalability of the trace analysis.

8.2 Future Work

Based on the results of this work, we identified some aspects that could be the subject of further research. First, it would be interesting to repeat our experiments in a cloud environment that allows more horizontal scale-out. The reason for this is that we observed indications for contention occurring on a cluster-wide level and that it would be interesting to assess whether this is caused by hardware resources becoming scarce, or by Kubernetes' management of hardware resources. Second, to assess the scalability of stream processing applications that are based on Kafka Streams accurately, it would be useful to have USL-based information about the scalability of Kafka. In particular, it would be useful to have more information about the read and write performance.

Another aspect we would like to further investigate is the influence of relation between the number of partitions and the numbers of instances of the *Trace Service*. By this, we could verify our presumption that having significantly more partitions than instances, i.e., stream processing threads, may result in dropped records occurring more frequently.

Moreover, we believe that there are additional ways to improve the methodology of Theodolite with regard to the USL: Even if the USL is based on the capacity metric, it may be possible to improve the heuristic execution also for benchmarks that use the *Resource Demand Metric* instead of the *Load Capacity Metric*. The basic idea is to extend Theodolite by a new search strategy that for each examined load value predicts the capacity for all of the resource values defined in the benchmark and then uses this information to choose the next SLO experiments that are executed to identify the demand. For such an implementation, the *Usl Predictor* component that we introduced can be reused. However, it should be carefully assessed under which assumptions such a search strategy would provide an accurate approximation of the *Resource Demand Metric*.

Finally, our results of benchmarking ExplorViz show that the scalability of the trace analysis is bounded by the *Trace Service*. Consequently, further considerations should analyze how to improve this microservice so that a more linear scalability can be achieved. In this regard, our benchmarks can be used for validation.

Bibliography

[Amdahl 1967] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). ACM, 1967, pages 483–485. DOI: 10.1145/1465482.1465560. (Cited on page 11)

[Apache Software Foundation 2022] Apache Software Foundation. *The Apache Avro website*. URL: https://avro.apache.org/ (visited on 05/18/2022). (Cited on page 41)

[Batts 2019] V. Batts. *Red Hat contributes CRI-O to the Cloud Native Computing Foundation*. 2019. URL: https://www.redhat.com/en/blog/red-hat-contributes-cri-o-cloud-native-computing-foundation (visited on 05/18/2022). (Cited on page 13)

[Butcher 2020] M. Butcher. *Celebrating Helm's CNCF graduation*. 2020. URL: https://helm.sh/blog/celebrating-helms-cncf-graduation/ (visited on 05/18/2022). (Cited on page 13)

[Carbone et al. 2015] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink™: stream and batch processing in a single engine. *IEEE Data Engineering Bulletin* 38 (2015). (Cited on page 16)

[Chantzialexiou et al. 2018] G. Chantzialexiou, A. Luckow, and S. Jha. Pilot-Streaming: a stream processing framework for high-performance computing. In: *2018 IEEE 14th International Conference on e-Science*. 2018, pages 177–188. DOI: 10.1109/eScience.2018.00033. (Cited on page 77)

[Chen et al. 2022] G. Chen, T. Johnson, and M. Cilimdzic. Quantifying cloud data analytic platform scalability with extended TPC-DS benchmark. In: *Performance Evaluation and Benchmarking*. Springer, 2022, pages 135–150. (Cited on page 21)

[Chou 2015] D. C. Chou. Cloud computing risk and audit issues. *Computer Standards and Interfaces* 42 (2015), pages 137–142. DOI: 10.1016/j.csi.2015.06.005. (Cited on page 1)

[Cloud Native Computing Foundation 2018] Cloud Native Computing Foundation. *Kubernetes is first CNCF project to graduate*. 2018. URL: https://www.cncf.io/blog/2018/03/06/kubernetes-first-cncf-project-graduate/ (visited on 05/18/2022). (Cited on page 13)

[Cloud Native Computing Foundation 2019] Cloud Native Computing Foundation. *CNCF announces containerd graduation*. 2019. URL: https://www.cncf.io/announcements/2019/02/28/cncf-announces-containerd-graduation/ (visited on 05/22/2022). (Cited on page 13)

[Confluent 2022a] Confluent. *The Confluent Platform documentation*. URL: https://docs.confluent.io/platform/current/platform.html (visited on 05/22/2022). (Cited on page 35)

[Confluent 2022b] Confluent. *The Confluent Schema Registry documentation.* URL: https://docs.confluent.io/platform/current/schema-registry/index.html (visited on 05/22/2022). (Cited on page 41)

[Cornelissen et al. 2009] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 35.5 (2009), pages 684–702. DOI: 10.1109/TSE.2009.28. (Cited on page 17)

[Ehrenstein 2022] S. Ehrenstein. Thesis artifacts for: scalability evaluation of ExplorViz with the Universal Scalability Law (2022). DOI: 10.5281/zenodo.6619236. (Cited on page 50)

[Ernst 2003] M. D. Ernst. Static and dynamic analysis: synergy and duality. In: *WODA 2003: Workshop on Dynamic Analysis.* 2003, pages 24–27. (Cited on page 17)

[Fittkau et al. 2017] F. Fittkau, A. Krause, and W. Hasselbring. Software landscape and application visualization for system comprehension with ExplorViz. *Information and Software Technology* 87 (2017), pages 259–277. DOI: 10.1016/j.infsof.2016.07.004. (Cited on pages 1, 17)

[Fittkau et al. 2013] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: the ExplorViz approach. In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT).* 2013, pages 1–4. DOI: 10.1109/VISSOFT.2013.6650536. (Cited on pages 2, 17)

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley, 1995. (Cited on page 31)

[Gannon et al. 2017] D. Gannon, R. Barga, and N. Sundaresan. Cloud-native applications. *IEEE Cloud Computing* 4.5 (2017), pages 16–21. DOI: 10.1109/MCC.2017.4250939. (Cited on pages 1, 27)

[Gaurier and Krassowski 2008] L. Gaurier and M. Krassowski. *An interface to use R from Python.* 2008. URL: https://rpy2.github.io/ (visited on 04/28/2022). (Cited on page 30)

[Gilbert and Lynch 2002] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM Sigact News* 33.2 (2002), pages 51–59. DOI: 10.1145/564585.564601. (Cited on page 6)

[Google 2001] Google. *Google's language-neutral, platform-neutral extensible mechanism for serializing structured data.* https://developers.google.com/protocol-buffers. Accessed: 2022-05-11. 2001. (Cited on page 41)

[Gunther 1993] N. Gunther. A simple capacity model of massively parallel transaction systems. In: *Int. CMG Conference.* 1993. (Cited on pages 1, 9)

[Gunther 2008] N. Gunther. *A general theory of computational scalability based on rational functions.* arXiv:0808.1431. 2008. (Cited on page 11)

[Gunther 2010] N. Gunther. *Guerrilla capacity planning: a tactical approach to planning for highly scalable applications and services*. 1st. Springer, 2010. DOI: 10.1007/978-3-540-31010-5. (Cited on pages 9–11, 21)

[Gunther 2018] N. Gunther. *USL scalability modeling with three parameters*. 2018. URL: http://perfdynamics.blogspot.com/2018/05/usl-scalability-modeling-with-three.html (visited on 04/17/2022). (Cited on page 10)

[Gunther 2019] N. Gunther. *Applying the Universal Scalability Law to distributed systems*. 2019. URL: https://www.youtube.com/watch?v=cuieNj3NrZ8&t. (Cited on page 21)

[Gunther et al. 2015] N. Gunther, P. Puglia, and K. Tomasette. Hadoop superlinear scalability: the perpetual motion of parallel performance. *Queue* 13.5 (2015), pages 20–42. DOI: 10.1145/2773212.2789974. (Cited on pages 9, 21)

[Hasselbring et al. 2020] W. Hasselbring, A. Krause, and C. Zirkelbach. ExplorViz: research on software visualization, comprehension and collaboration. *Software Impacts* 6 (2020). DOI: 10.1016/j.simpa.2020.100034. (Cited on pages 2, 17)

[Henning 2022] S. Henning. *Confluent Platform Helm Charts for Theodolite*. URL: https://github.com/SoerenHenning/cp-helm-charts/ (visited on 05/22/2022). (Cited on page 35)

[Henning and Hasselbring 2020] S. Henning and W. Hasselbring. Toward efficient scalability benchmarking of event-driven microservice architectures at large scale. In: *11th Symposium on Software Performance 2020*. Softwaretechnik-Trends 3. 2020, pages 28–30. (Cited on pages 2, 14, 55)

[Henning and Hasselbring 2021a] S. Henning and W. Hasselbring. How to measure scalability of distributed stream processing engines? In: *Companion of the ACM/SPEC International Conference on Performance Engineering*. ACM, 2021, pages 85–88. DOI: 10.1145/3447545.3451190. (Cited on pages 1, 14, 16, 24)

[Henning and Hasselbring 2021b] S. Henning and W. Hasselbring. Theodolite: scalability benchmarking of distributed stream processing engines in microservice architectures. *Big Data Res.* 25.C (2021). DOI: 10.1016/j.bdr.2021.100209. (Cited on pages 1, 13)

[Henning and Hasselbring 2022] S. Henning and W. Hasselbring. A configurable method for benchmarking scalability of cloud-native applications. *Empirical Software Engineering* (2022). In press. DOI: 10.1007/s10664-022-10162-1. (Cited on pages 1, 5–7, 14, 21, 25, 28, 45)

[Henning et al. 2021] S. Henning, B. Wetzel, and W. Hasselbring. Reproducible benchmarking of cloud-native applications with the Kubernetes operator pattern. In: *Symposium on Software Performance 2021*. 2021. (Cited on page 17)

[Herbst et al. 2013] N. Herbst, S. Kounev, and R. Reussner. Elasticity in cloud computing: what it is, and what it is not. In: *10th International Conference on Autonomic Computing (ICAC 13)*. USENIX Association, 2013, pages 23–27. (Cited on page 5)

[Heyman et al. 2014] T. Heyman, D. Preuveneers, and W. Joosen. Scalar: systematic scalability analysis with the Universal Scalability Law. In: *2014 International Conference on Future Internet of Things and Cloud*. 2014, pages 497–504. DOI: 10.1109/FiCloud.2014.88. (Cited on page 77)

[Holtman and Gunther 2008] J. Holtman and N. Gunther. Getting in the zone for successful scalability (2008). arXiv:0809.2541. (Cited on page 11)

[Hummer et al. 2013] W. Hummer, B. Satzger, and S. Dustdar. Elastic stream processing in the cloud. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 3.5 (2013), pages 333–345. DOI: 10.1002/widm.1100. (Cited on page 12)

[Hunt et al. 2010] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In: *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. 2010. (Cited on page 35)

[Jarvinen 2019] R. Jarvinen. Extending Kubernetes with the operator pattern. In: *33rd Large Installation System Administration Conference (LISA19)*. USENIX Association, 2019. (Cited on page 13)

[Karimov et al. 2018] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream data processing systems. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pages 1507–1518. DOI: 10.1109/ICDE.2018.00169. (Cited on pages 1, 16, 35, 78)

[Kluyver et al. 2016] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and J. development team. *Jupyter Notebooks? a publishing format for reproducible computational workflows*. IOS Press, 2016, pages 87–90. (Cited on page 17)

[Krause et al. 2021] A. Krause, M. Hansen, and W. Hasselbring. Live visualization of dynamic software cities with heat map overlays. In: *2021 Working Conference on Software Visualization (VISSOFT)*. IEEE, 2021. DOI: 10.1109/vissoft52517.2021.00024. (Cited on page 18)

[Krause-Glau 2022] A. Krause-Glau. *The ExplorViz live demo*. URL: https://www.explorviz.net/ (visited on 05/22/2022). (Cited on page 18)

[Kreps et al. 2011] J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing. In: *Proceedings of the NetDB*. 2011. (Cited on pages 12, 67, 74)

[Lakshman and Malik 2010] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* 44.2 (2010), pages 35–40. DOI: 10.1145/1773912.1773922. (Cited on page 41)

[Linux Foundation 2022] Linux Foundation. *The Prometheus website*. URL: https://prometheus.io/ (visited on 05/22/2022). (Cited on page 17)

[Luckow and Jha 2019] A. Luckow and S. Jha. Performance characterization and modeling of serverless and HPC streaming applications. In: *2019 IEEE International Conference on Big Data*. 2019, pages 5688–5696. DOI: 10.1109/BigData47090.2019.9006530. (Cited on page 77)

[Merkel 2014] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014.239 (2014). (Cited on page 13)

[Michael et al. 2007] M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski. Scale-up x scale-out: a case study using Nutch/Lucene. In: *2007 IEEE International Parallel and Distributed Processing Symposium*. 2007, pages 1–8. DOI: 10.1109/IPDPS.2007.370631. (Cited on page 6)

[Möding 2020] S. Möding. Analyze system scalability in R with the Universal Scalability Law (2020). (Cited on pages 30, 34, 55, 57)

[MongoDB 2022] MongoDB. *The MongoDB documentation website*. URL: https://www.mongodb.com/docs/ (visited on 05/22/2022). (Cited on page 41)

[OpenCensus Developers 2022] OpenCensus Developers. *The OpenCensus website*. URL: https://opencensus.io/ (visited on 05/23/2022). (Cited on page 17)

[Oracle 2022] Oracle. *The JMX technology website*. URL: https://www.oracle.com/java/technologies/javase/javamanagement.html (visited on 05/18/2022). (Cited on page 17)

[Rabenstein and Volz 2015] B. Rabenstein and J. Volz. Prometheus: a next-generation monitoring system (talk). In: Dublin: USENIX Association, 2015. (Cited on page 17)

[Ritz and Streibig 2008] C. Ritz and J. C. Streibig. *Nonlinear regression with R*. Springer, 2008. (Cited on page 10)

[Sax et al. 2018] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag. Streams and tables: two sides of the same coin. In: *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*. BIRTE '18. ACM, 2018. DOI: 10.1145/3242153.3242155. URL: https://doi.org/10.1145/3242153.3242155. (Cited on page 12)

[Schwartz 2017] B. Schwartz. Scalability is quantifiable: the Universal Scalability Law. In: *31st Large Installation System Administration Conference (LISA17)*. USENIX Association, 2017. (Cited on page 9)

[Schwartz and Fortune 2010] B. Schwartz and E. Fortune. Forecasting MySQL scalability with the Universal Scalability Law. *A Percona White Paper* (2010). (Cited on page 9)

[Sim et al. 2003] S. Sim, S. Easterbrook, and R. Holt. Using benchmarking to advance research: a challenge to software engineering. In: *25th International Conference on Software Engineering, 2003. Proceedings.* 2003, pages 74–83. DOI: 10.1109/ICSE.2003.1201189. (Cited on page 1)

[Strimzi Developers 2019] Strimzi Developers. *Strimzi Apache Kafka Operator joins the CNCF*. 2019. URL: https://strimzi.io/blog/2019/09/06/cncf/ (visited on 04/08/2022). (Cited on page 35)

[Vikash et al. 2020] Vikash, L. Mishra, and S. Varma. Performance evaluation of real-time stream processing systems for internet of things applications. *Future Generation Computer Systems* 113 (2020), pages 207–217. DOI: 10.1016/j.future.2020.07.012. (Cited on pages 22, 77)

[Vonheiden 2021] B. Vonheiden. Empirical scalability evaluation of window aggregation methods in distributed stream processing. Master's thesis. Kiel University, 2021. (Cited on page 16)

[Wang et al. 2021] G. Wang, L. Chen, A. Dikshit, J. Gustafson, B. Chen, M. J. Sax, J. Roesler, S. Blee-Goldman, B. Cadonna, A. Mehta, V. Madan, and J. Rao. Consistency and completeness: rethinking distributed stream processing in Apache Kafka. In: *Proceedings of the 2021 International Conference on Management of Data*. ACM, 2021, pages 2602–2613. (Cited on pages 2, 12, 67, 74)

[Weber et al. 2014] A. Weber, N. Herbst, H. Groenda, and S. Kounev. Towards a resource elasticity benchmark for cloud environments. In: *Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability*. HotTopiCS '14. ACM, 2014. DOI: 10.1145/2649563.2649571. (Cited on page 5)

[Wettel and Lanza 2007] R. Wettel and M. Lanza. Visualizing software systems as cities. In: *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. 2007, pages 92–99. DOI: 10.1109/VISSOF.2007.4290706. (Cited on page 17)

Appendix

List of Implementations

Migration of Kafka

Integrated in Theodolite version `v0.7.0`

Extension of Theodolite's Action Mechanism

Introduced in Merge Request: `https://git.se.informatik.uni-kiel.de/she/theodolite/-/merge_-requests/263`

Implementation of UslSearch search strategy and USL Offline Analysis Tool

Introduced in Merge Request: `https://git.se.informatik.uni-kiel.de/she/theodolite/-/merge_-requests/280`

Used Versions of the ExplorViz Microservices

ExplorViz Adapter Service

Source Code: `https://gitlab.com/sehrenstein/adapter-service/-/tree/scalability-benchmarking`

Docker Image: `registry.gitlab.com/sehrenstein/explorviz-benchmarks/explorviz-adapter`

ExplorViz Landscape Service

Source Code: `https://gitlab.com/sehrenstein/landscape-service/-/tree/scalability-benchmarking`

Docker Image: `registry.gitlab.com/sehrenstein/explorviz-benchmarks/explorviz-landscape`

ExplorViz Trace Service

Source Code: `https://gitlab.com/sehrenstein/trace-service/-/tree/scalability-benchmarking`

Docker Image: `registry.gitlab.com/sehrenstein/explorviz-benchmarks/explorviz-trace`

Implementation of the Load Generators

Source Code: `https://gitlab.com/sehrenstein/explorviz-benchmarks`

Raw Experiment Results and Replication Package (including Theodolite's Custom Resources)

Zenodo Archive: `https://doi.org/10.5281/zenodo.6619235`