Benchmarking Scalability of Load Generator Tools

Christopher Konkel

Bachelor's Thesis
May 1, 2023

Software Engineering Group
Department of Computer Science
Kiel University

Advised by
Prof. Dr. Wilhelm Hasselbring
Dr.-Ing. Sören Henning

Selbstständigkeitserklärung

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

Weiterhin erkläre ich, dass die digitale Fassung dieser Arbeit, die dem Prüfungsamt per E-Mail zugegangen ist, der vorliegenden schriftlichen Fassung entspricht.

Kiel,

Abstract

Scalability is one of the most important quality characteristics of modern day applications. It is a non-trivial effort to predict performance changes and scaling capabilities of complex systems under increasing load. As a solution, a wide variety of load generator tools can be used to simulate high amounts of load on a system under test. But in a testing environment with limited hardware resources, the system under test is not the only component of interest in terms of system resource requirements, as load generator tools can also require significant amounts of hardware resources. Theodolite, a framework for benchmarking horizontal and vertical scalability of cloud-native applications, also uses a load generator for its benchmarking process.

In this thesis, we compare the vertical scalability of the custom load generator used by Theodolite to the three load generator tools k6, Gatling and JMeter using Theodolite benchmarking methodology. We evaluate the tools based on the niche use case of the custom load generator, simulating input of collected sensory data. We evaluate the HTTP and Kafka load format. To achieve this, we first configure the three tools to match the selected use case in both load formats and then integrate them into the Theodolite environment. Second, we define new benchmarks for Theodolite to use the selected load generators in its benchmarking process. Third, we execute the newly defined benchmarks and evaluate the results of the experiments. We find significant differences regarding the vertical scalability of the selected tools, with scalability differences of the same tool depending on the load format.

Contents

Chapter 1

Introduction

1.1 Motivation

Load generator tools have all kinds of use cases. From generating load on a web server for performance testing, the benchmarking of distributed stream processing engines [Henning and Hasselbring 2021b] to the implementation of a denial of service attack generator [Grabovsky et al. 2018]. Theodolite [Henning and Hasselbring 2022b], a scalability benchmarking tool for cloud-native applications, also uses load generators in its benchmarking method. Theodolite focuses on analysing performance of a system under test [Henning and Hasselbring 2022a]. The load generators stress the system under test with configurable amounts of workload, allowing to evaluate the behaviour of the monitored system in different scenarios.

Within benchmarking processes, the system under test often is the primary focus in every evaluation. But in a testing environment with limited hardware resources, the system under test is not the only component of interest in terms of system resource requirements, as load generator tools can also require significant amounts of hardware resources.

This thesis will focus on evaluating and comparing the scalability of different load generator tools by integrating them into the Theodolite benchmarking method. To achieve this, the scope of this comparison will be narrowed down to the niche use case of Theodolite's own load generator. Still, the expectation of this evaluation is to receive enough information about each load generator to identify tool recommendations.

1.2 Goals

The following goals define the scope of this thesis.

G1: Create Example Configurations for Different Load Generator Tools and the Integration in the Theodolite Environment

The team behind the Theodolite framework provides a custom load generator that simulates the input of collected sensory data. The first goal of this work is to configure the selected load generators in this evaluation to generate the same load format as the custom load generator. All load generators should support both the Kafka and HTTP message format.

This goal also includes setting up the load generators in the Theodolite environment in a way that they can be benchmarked.

G2: Define New Benchmarks for the Theodolite Operator to Enable the Evaluation of Load Generator Scalability

The second goal is to instruct the Theodolite operator to execute the load generator benchmarking experiments. To achieve this, we define the necessary benchmarks and executions. This task is non-trivial as it is not the intended use case of Theodolite to benchmark the load generator itself.

G3: Compare the Previously Listed Load Generator Tools

The final goal is to use the new benchmarks to gain valuable insights into the scalability of the different load generator tools. Even though the evaluation will focus on a niche use case (i.e., reproducing the behaviour of the custom theodolite load generator), the expectation is to receive enough information about each load generator to formulate usage considerations primarily regarding load quantities.

1.3 Document Structure

Chapter 2 introduces foundations and technologies that are relevant for understanding and following through the rest of this thesis. Chapter 3 specifies the setup for the unconventional load generator benchmarking approach using Theodolite. Chapter 4 will present the benchmarking results and discuss them. Related work is discussed in Chapter 5. Finally, Chapter 6 concludes the evaluation and presents possible future work.

Chapter 2

Foundations and Technologies

2.1 Benchmarking Software Systems

The term benchmarking has several definitions, two of which are quoted below and used as the definition in this work. Henning and Hasselbring [2022a] follow the definition that benchmarks "are an established research method to compare different methods, techniques, and tools based on a standardized method". V. Kistowski et al. [2015] defines a benchmark as a "[...] tool for the competitive evaluation and comparison of competing systems or components according to specific characteristics, such as performance, dependability, or security".

The requirements for a benchmark specification are formulated by V. Kistowski et al. [2015] as followed:

1. **Relevance** The benchmarking scenario should be close to the scenario of interest

2. **Reproducibility** Benchmark results have to be reproducible. The same test configurations have to produce similar results.

3. **Fairness** A benchmark should treat all test configurations equally, i.e. executing all test configurations without artificial limitations.

4. **Verifiability** The result of a benchmark must be accurate.

5. **Usability** Other users should be able to run the benchmarks in their test environments with minimal unnecessary effort.

2.2 Scalability of Software Systems

The scalability of a software system is "the ability of [a] system to sustain increasing workloads by making use of additional resources" [Herbst et al. 2013; Henning and Hasselbring 2021a]. The term scalability can be, broadly speaking, split into two categories: Vertical and horizontal scalability [Lehrig et al. 2015].

Horizontal Scalability

Horizontal scalability is the ability to effectively use more machines added to a network to handle the increasing load. This method is therefore not limited by the upgradeability of a single machine, but adds to the complexity of the software as inherent problems of distributed systems have to be overcome. If horizontal scaling is not applicable, vertical scaling can be used.

Vertical Scalability

Unlike horizontal scalability, vertical scalability is the ability to use the increased capacity of a machine, achieved by adding more (hardware) resources. This is often the case for systems that are designed to be scalable from the beginning. A single machine can only be upgraded to a certain extent, as the upgradeability of hardware is limited. The benchmarking process in this thesis will primarily evaluate vertical scalability of load generator tools.

2.3 The Container Orchestration Platform Kubernetes

Kubernetes is an open-source orchestration tool for deploying containerized applications. The Kubernetes project got open-sourced by Google in 2014 and has become the de-facto standard orchestration tool for cloud-native applications [Burns et al. 2016; Henning and Hasselbring 2022a]. It offers features like, among others, the deployment, scaling and management of containerized applications.

In the context of Kubernetes, this thesis uses the terms Pods, Containers, Deployments and ConfigMaps.

1. **Pod**: A Pod is the smallest "unit" of an application and consists of at least one container.
2. **Deployment**: A Deployment is a higher level definition of a set of Pods that can be managed together.
3. **ConfigMap**:[1] A Kubernetes resource used to store data in key-value pairs.

Kubernetes is used to deploy Theodolite and additional resources in order to execute the benchmarks. *kubectl*, the official command line tool for Kubernetes, is used for the interaction with the Kubernetes API.

2.4 Prometheus

Prometheus is an open-source systems monitoring tool [Cloud Native Computing Foundation 2016], collecting and storing metric data from all kinds of systems. It stores collected

[1]https://kubernetes.io/docs/concepts/configuration/configmap/

data as time series, i.e. streams of timestamped values belonging to the same metric and the same set of labeled dimensions.[2] A metric is a numeric measurement for something. As Prometheus records metrics as streams, changes over time can be tracked. Metric names specify a feature to be measured, e.g. total HTTP requests received by a system. Labels can then be used to further categorize the data within a metric, e.g. the type of HTTP request received.

To access collected and stored data, Prometheus provides a custom query language PromQL.[3] It offers, similar to other query languages, a set of tools to select and aggregate stored data. Theodolite uses Prometheus to collect and store raw data from components inside the cluster, that is then evaluated to generate results (see Figure 2.2).

2.5 The Scalability Benchmarking Framework Theodolite

Theodolite [Henning and Hasselbring 2022b] is a scalability benchmarking framework for cloud-native applications. It can be used to measure the horizontal and vertical scalability of a system under test [Henning and Hasselbring 2022a].

Theodolite runs as a Kubernetes Operator inside a Kubernetes cluster. This allows, besides other improvements, the usage of declarative files to define Benchmarks and their Executions (in the form of custom Kubernetes resources).

Using the Theodolite Operator, the lifecycle of benchmarks can be managed with established Kubernetes tooling. To run a benchmark, the user needs to provide a `Benchmark custom resource`, an `Execution custom resource` and necessary dependencies. Each execution consists of isolated experiments for different load intensities and provisioned resource amounts to determine scalability results.

`Benchmark resources`, `Execution resources` and other terminology used in the Theodolite context will be described in the following. The resources then get explained in Section 3.3 in greater detail.

2.5.1 The `Benchmark Custom Resource`

A `Benchmark resource` specifies the application that will be benchmarked, but does not contain the configuration for actually running the benchmark. The most relevant properties for this thesis are:

System under Test (SUT) The application to be benchmarked.

Load Generator (LG) The application that generates a configurable amount of workload on the SUT.

[2]https://prometheus.io/docs/concepts/data_model/
[3]https://prometheus.io/docs/prometheus/latest/querying/basics/

Patcher Patcher configure load generator and SUT resources, e.g. instances, hardware resource limits, ...

Service Level Objectives (SLOs) A definition that provides a way to quantify whether a certain load intensity can be handled by a certain amount of provisioned resources [Henning and Hasselbring 2022c].

2.5.2 The `Execution Custom Resource`

`Execution` resources contain the configuration for running a specific instance of a benchmark. As these hold the setup for the benchmarking process, benchmarking is only started once an execution has been deployed to the cluster. Relevant configuration properties are:

Resource Dimensions A discrete numeric range describing different amounts of resources available for the SUT, e.g., CPU cores. Represented on the y-axis in Figure 2.1.

Load Dimensions A discrete numeric range describing different intensities of load on the SUT, e.g., requests per second. Represented on the x-axis in Figure 2.1.

Search Strategy A strategy to select which experiment (selection of resource amount and load intensities) should be run next, depending on the outcome of the previous experiment [Henning and Hasselbring 2022a]. The Benchmarks in this thesis only use the lower bound linear search. More information on search strategies can be found in the documentation.[4]

2.5.3 The Theodolite Operator Architecture

The Theodolite operator gets notified about Kubernetes resource changes and handles further actions accordingly, e.g. starting/stopping load generator instances or SUTs and querying results. Both the load generator and the SUT are monitored by Prometheus. The operator then queries the results from the Prometheus storage, acts upon results according to the Benchmark configuration and finally stores the results for the benchmarker to retrieve. At any time during the experiment, the benchmarker can access the raw collected data from Prometheus using an integrated Grafana dashboard.[5] Figure 2.2 shows the architecture of the Theodolite operator including external services.

[4]https://www.theodolite.rocks/concepts/search-strategies
[5]https://grafana.com

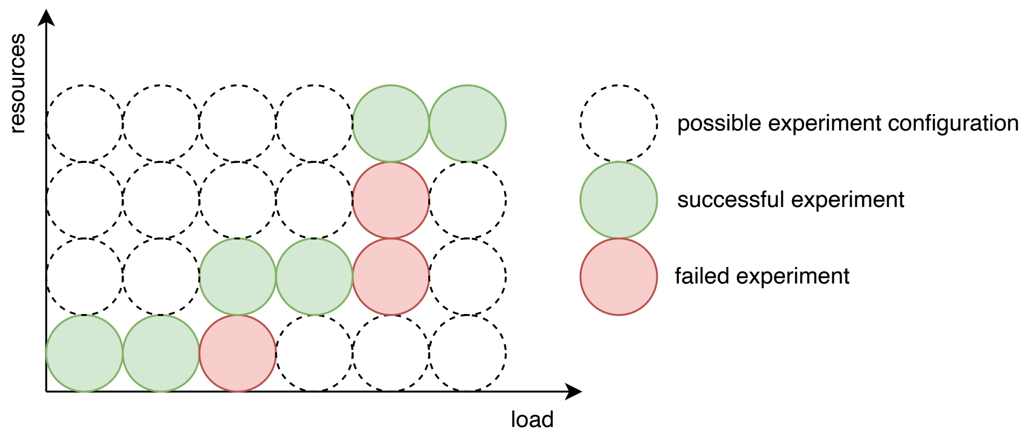


Figure 2.1. A visualization of the lower bound linear search strategy used to determine the execution (order) of experiments.

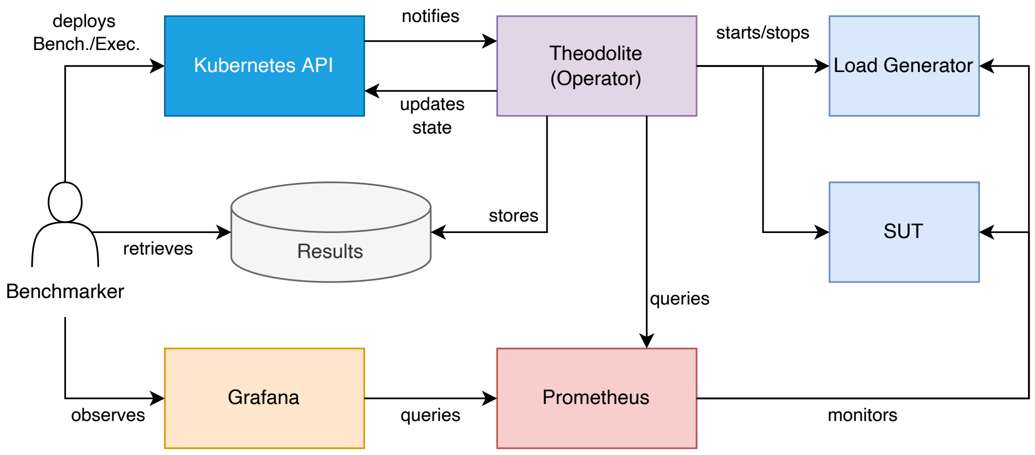


Figure 2.2. Interactions between the benchmarker and Theodolite's components [Henning and Hasselbring 2022b].

2.6 Load Generator Tools

As stated in Section 2.5.1, load generator tools (LGs) generate a configurable amount of workload on a specified endpoint. A wide variety of load generator tools for benchmarking purposes is available.[6] For the scope of this thesis, the following 4 load generators will be part of the comparison:

1. **Theodolite's Load Generator (Theodolite LG)**:[7] A simple custom load generator, also created by the Theodolite team, for the specific use case of simulating input of collected sensory data. Not to be confused with the Theodolite operator.
2. **JMeter** [Apache Software Foundation 2021]: The choice of JMeter represents popular but older load testing tools. They primarily benefit from established documentation, community extensions and community support.
3. **k6** [Grafana Labs 2021]: Representing modern load testing tools. Specifically claims simulation of lots of traffic, even on lower-end machines.
4. **Gatling** [Gatling Corp 2015]: Another modern load generator, chosen for this comparison as a modern alternative to JMeter.

This selection of tools covers both a good quantity ratio to detect anomalies and outliers and a range of different languages used for the implementation (Java, Scala and Golang).

2.7 The Messaging System Apache Kafka

Apache Kafka [Kreps et al. 2011] is a highly scalable, fault tolerant, distributed system that can act as a message broker between producers and consumers. Communication is achieved by publishing/subscribing to specific data topics [The Kafka Authors 2022]. Many big companies use Kafka for their messaging needs, as the system is highly configurable and supports client integration in most popular programming languages. For this reason, Kafka messages will be one of the two message formats used in the evaluation of this thesis.

[6]https://github.com/denji/awesome-http-benchmark
[7]https://www.theodolite.rocks/theodolite-benchmarks/load-generator.html

Table 2.1. The chosen load generator tools for the comparison in this thesis including a characteristics overview.

Name	Written In	HTTP Support	Kafka Support
Theodolite Load Generator	Java	Native	Native
APACHE JMeter™	Java	Native	Custom Sampler Code
k6	Golang	Native	Community Extension
Gatling	Scala	Native	Community Extension

Chapter 3

Benchmarking Setup

The goal of the benchmarking process is to evaluate vertical scalability of different load generator tools. To do so, we will be using the results of the Theodolite LG as a baseline for the other contestants.

The Theodolite LG was built to simulate input of collected sensory data. The corresponding message type of simulated data is specified as `ActivePowerRecords`, defined in the data serialization system Avro.[1] They consist of an identifier for simulated power sensor, a timestamp in epoch milliseconds and a simulated value in watts Listing 3.1.

Listing 3.1. Avro definition of the message type `ActivePowerRecord`.

```
  record ActivePowerRecord {
      /**
       * identifier for simulated power sensor.
       */
      string identifier;
      /**
       * timestamp in epoch milliseconds.
       */
      long timestamp;
      /**
       * simulated value in Watts (configurable constant value).
       */
      double valueInW;
    }
```

These records then get serialized for compatibility with the configured LG load format. For HTTP, the records are serialized to JSON and used as payload in HTTP Post messages. For Kafka, the records are serialized with the Confluent Schema Registry,[2] a standard serialization format for kafka topic message data, and sent to Kafka.

With Theodolite's LG beeing the baseline, the first setup step is to configure each load generator to mimic the simulated data of the Theodolite LG. The second step is to enable the LGs to run within the Theodolite environment. We handle this with the use of

[1]https://avro.apache.org

[2]https://docs.confluent.io/platform/current/schema-registry/

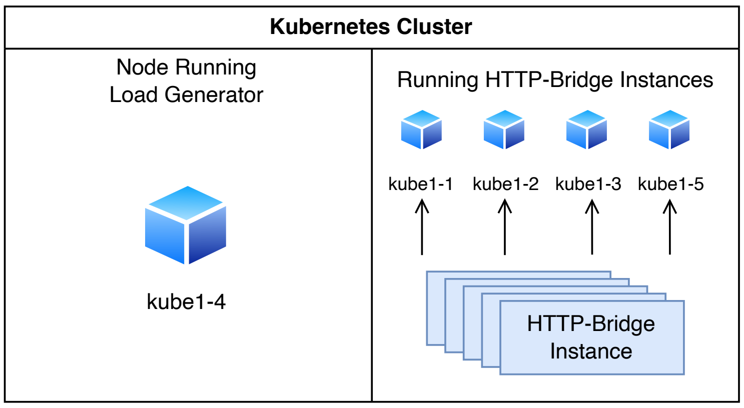


Figure 3.1. Kubernetes cluster structure for HTTP experiments. Note that the SUT instances run on different nodes than the load generator.

Kubernetes `Deployment resources` and `ConfigMap resources`.

Afterwards, we define `Benchmark resources` and `Execution resources` to use the Theodolite framework for our purposes.

All configuration files and other resources can be found in the artifact repository for this thesis [Konkel 2023].

3.1 Cluster Setup

Before discussing the benchmarking setup, we want to discuss the overall cluster setup. The Kubernetes cluster consists of five nodes, `kube1-1` to `kube1-5`, used to run executions.

Load generators run in HTTP configuration send their requests to HTTP-bridge instances (SUT) in the cluster. A HTTP-bridge instance is a Deployment with simple HTTP endpoints that receive requests and send a success response. Besides generating metrics for the use of their endpoints, these instances do not implement any additional functionality.

When the LGs run in Kafka configuration, they send their requests to typical Kafka instances (SUT).

To prevent unintended performance variations, a single node never hosts both a load generator and a request receiving instance. To fulfill the requirements for benchmark specifications (Section 2.1), especially the specification attribute reproducibility, we decided to use a partitioned Deployment approach that is showcased in Figure 3.1 for the HTTP configuration and Figure 3.2 for the Kafka configuration.

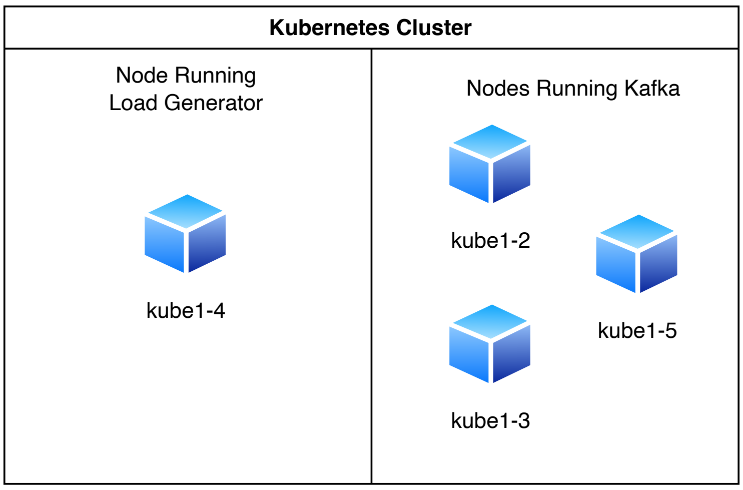


Figure 3.2. Kubernetes cluster structure for Kafka experiments. Note that the SUT instances run on different nodes than the load generator.

3.2 Load Generator Setup

In the following sections, we present the basic setup and configuration for each load generator. Gatling, k6 and the Theodolite LG use an open-workload approach [Hashemian et al. 2012]. Only JMeter uses an closed-workload approach due to its implementation of Thread Groups. They also contain variables set from outside of the configuration. Those variables are explained in Section 3.3.1.

3.2.1 Gatling

Gatling uses the concept of virtual users and scenarios. Important definitions for the Gatling environment are:

Virtual User (VU) VUs execute tasks given to them.

Scenario Scenarios are workload definitions for VUs. They are given to Gatling in the form of script files.

Injection Profile Injection profiles are the way that VUs are started.

Simulation The actual configuration of the test. The simulation describes the injection profile of VUs and which scenario those users will run.

Listing 3.2 shows the definition of the HTTP simulation (written in Java), Listing 3.3 the definition of the Kafka simulation (written in Scala).

Listing 3.2. Shortened configuration of the Gatling LG for the HTTP load format.

```java
public class TheodoliteHttpSimulation extends Simulation {
    String url = System.getProperty("url");
    HttpProtocolBuilder httpProtocol = http.baseUrl(url);

    ScenarioBuilder httpload = scenario("Users").exec(http("http␣load-
        testing").post("/gatling").body(StringBody(session -> {
                return "{\"identifier\":␣\"" + session.userId() % 50 + "
                    \",␣\"timestamp\":␣" + System.currentTimeMillis() + ",
                    ␣\"valueInW\":␣1234.56}";
                    })).header("content-type", "application/json"));
    {
        setUp(
            httpload.injectOpen(constantUsersPerSec(users).during(duration
                ).randomized())
        ).protocols(httpProtocol);
    }
}
```

The HTTP simulation first defines a scenario that creates HTTP Post messages which carry the JSON data as payload. The simulated identifier `session.userId()% 50` has no real impact on the experiment and is only used in this configuration to keep the values equal to the Kafka counterpart. After that, the injection profile is set to inject a constant number of users per second for a set duration. As Gatling supports this feature natively, the `randomized()` call injects VUs at random intervals to prevent injection peeks. Injection peeks can occur when the load generator injects all VUs at the same time in a constant interval. Lastly, VUs are instructed to execute the HTTP scenario.

To implement the Kafka format, a third party Kafka plugin is used.[3] The Kafka message consists of a key identifier and the serialized record in form of a byte array. The simulated key (and identifier) `session.userId % threads` is configured to match the amount of partitions in the Kafka topic. This is necessary to ensure that Kafka stores the messages correctly. Besides that, it is important to notice that this Gatling Kafka simulation uses a local schema registry instead of the one in the cluster, due to compatibility issues. As we are using rather small messages, this should not impact either overall performance or message size.

[3]https://github.com/Tinkoff/gatling-kafka-plugin

Listing 3.3. Shortened configuration of the Gatling LG for the Kafka load format.

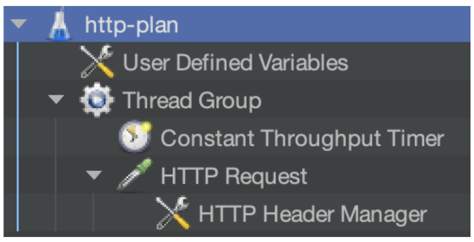
```
class TheodoliteKafkaSimulation extends Simulation {
    val kafkaConf: KafkaProtocol = kafka.topic("gatling-kafka-topic")
    val schemaParser = new Parser
    val valueSchemaJson = "{[...]}"
    val valueSchemaAvro = schemaParser.parse(valueSchemaJson)
    val writer = new SpecificDatumWriter[GenericData.Record](
        valueSchemaAvro)

    val scn: ScenarioBuilder = scenario("Basic")
    .exec(
        kafka("BasicRequest")
        .send[String,Array[Byte]](session => (session.userId % threads
            ).toString, session => {
        val avroRecord = new GenericData.Record(valueSchemaAvro)
        avroRecord.put("identifier", (session.userId % threads).
            toString)
        avroRecord.put("timestamp", System.currentTimeMillis())
        avroRecord.put("valueInW", 1234.56)
        val out = new ByteArrayOutputStream
        [...]
        })
    )
    setUp(scn.inject(constantUsersPerSec(requestsPerSecond.toDouble).
        during(duration).randomized)).protocols(kafkaConf)
}
```

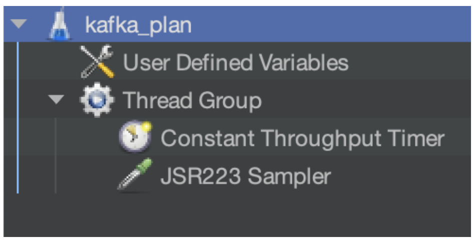
3.2.2 JMeter

The JMeter architecture is based on the concept of emulated users [Apache Software Foundation 2021]. JMeter test plans get organized in Thread Groups that get executed in configurable order. Within a Thread Group, each emulated user represents a Thread. As the test plans for this thesis are simple, we only use one Thread Group per plan. An emulated user can generate requests for a target system in a specified way. To generate the desired load, JMeter offers several plugins[4] that can be used in combination. Each plugin provides some functionality, that can be combined to simulate complex scenarios. In the following, we list used plugins and describe the configuration to create the specific load for our test cases.

[4]https://jmeter.apache.com/usermanual/component_reference.html



(a) Test plan configuration for the HTTP load format



(b) Test plan configuration for the Kafka load format

Figure 3.3. Overview of the JMeter test plan structure. Left side shows the minimal required setup for constant HTTP requests. Right side shows the minimal required setup for Kafka messages, using custom JSR223 Sampler code.

1. **HTTP Request** allows sending HTTP requests and handles the response.
2. **HTTP Header Manager** allows adding or override the HTTP request header.
3. **JSR223 Sampler** supports the execution of custom code written in Groovy and other programming languages[Apache Software Foundation 2021]. In this work, this sampler is used for the Kafka load implementation.
4. **Constant Throughput Timer** allows the instruction of emulated users to execute requests with a fixed rate per minute.

As with Gatling, we provide two configurations for the JMeter load generator. One configuration covers the HTTP load format, the other the Kafka load format. Figure 3.3 shows the structure of both test plans. For the HTTP test plan, we first configure the HTTP request type to POST and define the payload of the body as the data serialized to JSON (shown in Listing 3.4).

Listing 3.4. HTTP POST payload of the JMeter HTTP test plan

```
{
"identifier": "${__groovy(${__threadNum} % 50)}",
"timestamp": ${__time()},
"valueInW": 1234.56
}
```

Again, the identifier mimics the value boundary of the Kafka configuration for consistency. We then define the HTTP header content-type to `application/json` within the HTTP Header Manager. Lastly, we set the amount of requests per minute inside the Constant Throughput Timer.

For the Kafka test plan, we first configure the JSR223 Sampler to generate the desired Kafka messages (shown in Listing 3.5). We use a custom configured JSR223 Sampler instead of third party plugins like pepper-box,[5] because of compatibility issues despite using the required java version. As both Gatling and k6 had to use a local schema registry due to compatibility issues, we decided to also use this approach with JMeter for the sake of comparison. The implementation creates Kafka messages by serializing the corresponding Avro record as a byte array and using it as the payload for the actual message. We then configure the amount of requests per minute inside the Constant Throughput Timer, as with the HTTP test plan.

Listing 3.5. Shortened JSR223 Sampler configuration of the JMeter Kafka test plan.

```
        def user = String.valueOf((ctx.getThreadNum() + 1) % 50);
        def schemaParser = new Parser();
        def valueSchemaJson = [...];

        def avroRecord = new GenericData.Record(valueSchemaAvro);
        avroRecord.put("identifier", user);
        avroRecord.put("timestamp", System.currentTimeMillis());
        avroRecord.put("valueInW", 1234.56);
        [...]
        def msg = out.toByteArray();

        def producer = new KafkaProducer<>(kafkaProps);
        def record = new ProducerRecord<>("jmeter-kafka-topic", user, msg)
        producer.send(record);
```

3.2.3 k6

Just like Gatling, k6 also uses the concept of virtual users. k6 is configured using JavaScript script files. Important definitions for the k6 environment are:

Virtual user (VU) VUs execute tasks given to them.

Init code Code, which prepares the test.

VU code Code, which is executed by the virtual users and makes requests.

Listing 3.6 shows the HTTP script implementation, Listing 3.7 the Kafka script. For the HTTP script, we first set the global configuration settings inside the init code region. `executor` configures the load generator to simulate a constant arrival rate, again following the open-workload approach [Hashemian et al. 2012]. `preAllocatedVUs` and `maxVUs` set the lower and upper boundary for the amount of active virtual users. The VU code then defines the HTTP POST message similar to the other load generators and sends the request. For

[5]https://github.com/GSLabDev/pepper-box

the Kafka script, a third party Kafka extension is used.[6] The init code part is identical to the HTTP script. In the VU code, another local schema registry is defined. k6 also uses a local schema registry due to compatibility issues. After that, we define the message using appropriate data types and serialization and finally send it to Kafka.

Listing 3.6. Shortened HTTP configuration script for k6.

```
        // --- Init Code ---
        export const options = {
            scenarios: {
                SensorData: {
                executor: 'constant-arrival-rate',
                duration: __ENV.DURATION,
                // Iterations of function per 'timeUnit'.
                rate: __ENV.ITERATIONS_PER_TIMEUNIT,
                // To start 'rate' iterations per second.
                timeUnit: '1s',
                preAllocatedVUs: __ENV.PRE_ALLOCATED_VUS,
                maxVUs: __ENV.MAX_VUS,
                },
            },
        };

        // --- VU Code ---
        export default function () {
        const url = __ENV.URL;
        const valueInW = 1234.56;
        const payload = JSON.stringify({
            identifier: String(exec.vu.idInInstance % 50),
            timestamp: Date.now(),
            valueInW: 1234.56,
        });
        const params = {
            headers: {
            'Content-Type': 'application/json'
            },
        };

        http.post(url, payload, params);
        }
```

[6]https://github.com/mostafa/xk6-kafka

Listing 3.7. Shortened Kafka configuration script for k6.

```
// --- Init Code ---
export const options = {
    scenarios: {
        SensorData: {[...]},
    },
};

// --- VU Code ---
const topic = "k6-kafka-topic";
const writer = new Writer([...]);
const schemaRegistry = new SchemaRegistry();
const valueSchema = [...];

export default function () {
let message = [
    {
        key: schemaRegistry.serialize({
        data: String(exec.vu.idInInstance % 50),
        schemaType: SCHEMA_TYPE_STRING,
        }),
        value: schemaRegistry.serialize({
        data: {
            identifier: String(exec.vu.idInInstance % 50),
            timestamp: Date.now(),
            valueInW: 1234.56,
        },
        schema: { schema: valueSchema },
        schemaType: SCHEMA_TYPE_AVRO,
        }),
    },
    ];
    writer.produce({ messages: message });
}
```

3.2.4 Theodolite Load Generator

Because the Theodolite LG is custom made for use within the Theodolite environment, it is configured entirely within `Deployment resources`. Listing 3.8 shows the required environment variables inside the `Deployment resources` to configure the load generator. How environment variables works and the structure of these resources are explained in the next section.

Listing 3.8. Cutout of a `Deployment resource` used to configure the Theodolite LG. This part only shows the required environment variables to configure the load generator.

```
env:
- name: NUM_SENSORS
value: "150000"
- name: THREADS
value: "10"
- name: VALUE
value: "1234.56"
- name: KAFKA_BOOTSTRAP_SERVERS
value: "theodolite-kafka-kafka-bootstrap:9092"
- name: SCHEMA_REGISTRY_URL
value: "http://theodolite-kafka-schema-registry:8081"
- name: TARGET
value: "http"
- name: HTTP_URL
value: "http://theodolite-http-bridge:8080/post"
```

3.3 Theodolite Setup

Benchmark executions in Theodolite require the definition of each component in Figure 3.4. The following subsections describe each component in greater detail to get a better understanding of the inner workings in the benchmarking process.

3.3.1 `Deployment Resources` and Dependencies

Theodolite can spin up and control the different load generator tools using `Deployment resources`. These files contain the configuration for each load generator and required dependencies.

Listing 3.9 shows the structure of a `Deployment resource`. We first define the name of the Deployment. We then specify the amount of instances that should be deployed to a specific node that is defined in the lines underneath. Then, the container itself gets configured. The docker image contains most of the parts that a load generator needs to run.

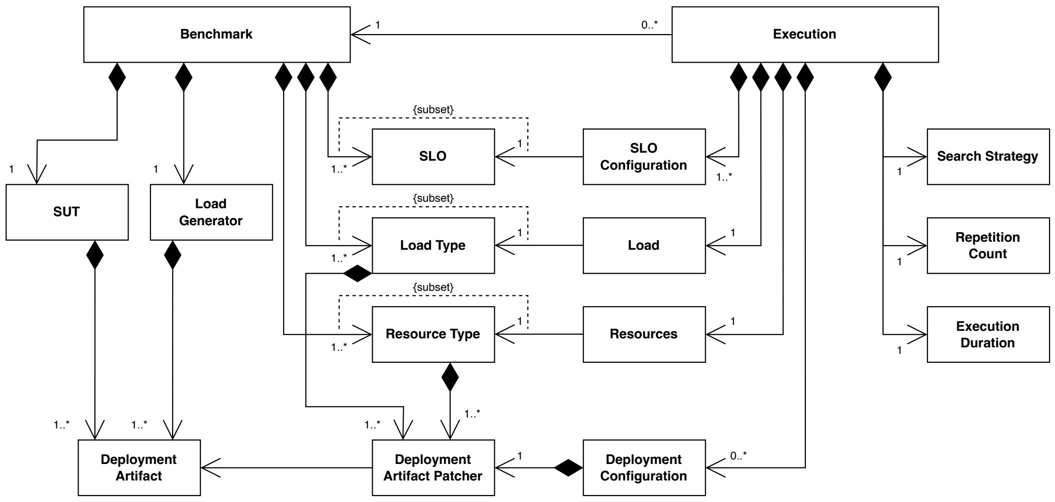


Figure 3.4. UML class diagram of Theodolite's scalability benchmarking data model [Henning and Hasselbring 2022a].

It contains the application (the LG) itself and all dependencies, i.e. libs and other extension files (Kafka client jar, ...).

After that, the command keyword defines the command line command that should be executed in the container once started. For the Theodolite LG, this line is not used, as it only supports our niche use case and its values can be configured by only using environment variables. For the other load generators, this line configures the execution of the load generator and parses environment variables into the container. These environment variables are defined in the following lines, where each variable receives a name and a default value.

The value of each environment variable can be dynamically configured using the combination of `Benchmark resources` and `Execution resources`. This is the mechanism that makes each execution configuration dynamic. How those resources modify environment variables gets explained in the following subsections. Lastly, the combination of `volumeMounts` and `volumes` allows to mount the content of ConfigMaps into the container as a directory. In our setup, these mounts contain the required test plans or script files for the laod generators to access within the container.

3.3.2 `Benchmark` Resources

Listing 3.10 shows most (not all) of the necessary configuration attributes to set up a benchmark. After setting the name, the `Benchmark resource` defines the SUT and load generator to be used. To do so, Deployments for both the SUT and the load generator are stored in the cluster using ConfigMaps. They can then be accessed and deployed by

Theodolite.

As noted earlier, even though our load generators are the focus of our comparison, they still define the load generator component within the benchmarking configuration. This means that both the HTTP and Kafka endpoint define the SUT component, while the LGs define the load generator component in Figure 3.4.

`resourceTypes` and `loadTypes` define the testing dimensions for the benchmark. Each type definition instructs patchers, components which take a value as input and modify a Kubernetes resource,[7] to change both the values of environment variables as well as hardware resources available to the Deployment. Usually, the resource types define patchers to modify the SUT instances and the load types define patchers to modify the load generator. As we are focusing on the scalability of the load generator, we use both types to modify two (instead of one) parameters of the load generator. The remaining lines configure the SLO (as explained in Section 2.5.1). A PromQL query is defined to either query the received HTTP requests or incoming Kafka messages, depending on the type of configuration.

`warmup` sets a time interval from the beginning of the experiment to the specified value in which no data is collected. This time interval is set to ignore the starting phase of the load generator (and other resources), as this could negatively impact the results.

[7]https://www.theodolite.rocks/creating-a-benchmark.html#load-and-resource-types

Listing 3.9. The shortened content of a `Deployment resource` file, annotated with explanations of the most important code lines.

```yaml
kind: Deployment
metadata:
  name: k6-http    # name of deployment.
spec:
  replicas: 1    # the amount of instancs to spin up.
    spec:
      nodeSelector:
        kubernetes.io/hostname: kube1-4    # specify the node to run
            the deployment on.
      containers:
        - name: k6-http
          image: grafana/k6    # docker image to run in the container.
          command: [...]    # command to be executed within the
              running container.
          env:    # definition of environment variables.
            - name: NUM_USERS
              value: "28000"    # default value for a given variable.
            - name: REQUESTS_PER_SECOND
              value: "10"
          volumeMounts:    # the following lines mount the content of
              a configmap into the container as a directory.
          - name: k6-script
            mountPath: /home/k6/script
      volumes:
        - name: k6-script
          configMap:
            name: k6-http-script
```

Listing 3.10. The shortened content of a `Benchmark resource` file, annotated with explanations of the most important code lines.

```
kind: benchmark
metadata:
name: k6-http-benchmark    # name of benchmark.
sut:    # definition of sut for the benchmark.
    resources:
    - configMap:
        name: http-bridge-deployment
loadGenerator:    # definition of the load generator for the benchmark.
    resources:
    - configMap:
        name: k6-http-deployment
resourceTypes:
    - typeName: "CPUResources"
    patchers:
        - type: ResourceLimitPatcher
        resource: "k6-http-deployment.yaml"
        properties:
            container: k6-http
            limitedResource: cpu
            factor: 1000
            format: m
loadTypes:
    - typeName: RequestsPerSecond
    patchers:
        - type: "EnvVarPatcher"
        resource: "k6-http-deployment.yaml"
        properties:
            container: k6-http
            variableName: REQUESTS_PER_SECOND
slos:
    name: "IncomingHttpRequestsPerSecond"    # name of the slo.
    prometheusUrl: "http://prometheus-operated:9090"
    properties:
        promQLQuery: "sum(rate(jetty_server_requests_seconds_count{uri='/
            k6'}[30s]))"
        warmup: 60    # in seconds.
        queryAggregation: min
        repetitionAggregation: median    # calculate median over the
            results of repititions of same execution.
        operator: gte    # data gets evaluated using the '>=' operator.
        threshold: 10    # in received requests per second.
```

3.3.3 `Execution Resources`

"An Execution represents a one-time execution of a benchmark with a specific configuration" [Henning and Hasselbring 2022a]. An example of the content inside of an `Execution Resources` is shown in Listing 3.11. As a `Benchmark resource` may define multiple load and resource types, the `Execution resource` has to choose between them.[8] It selects one load and one resource type and defines a list of valid numeric values for each of them. These are the values that override both the defaults in the `Benchmark` and `Deployment resource` (when selected) and get parsed into the SUT and load generator as environment variables. The `Execution resource` also selects a subset of SLOs and can override default SLO properties. Lastly, a total duration for each experiment as well as the number of repetitions per experiment are set.

[8]https://www.theodolite.rocks/creating-an-execution.html#selecting-load-type-resource-type-and-slos

Listing 3.11. The shortened content of a `Execution resource` file, annotated with explanations of the most important code lines.

```
kind: execution
metadata:
name: k6-http-execution
spec:
benchmark: k6-http-benchmark
load:
    loadType: "RequestsPerSecond"    # select one(!) loadType of the
        benchmark configuration.
    loadValues: [10000,15000,20000]
resources:
    resourceType: "CPUResources"    # select one(!) resourceType of the
        benchmark configuration.
    resourceValues: [1,2,4]
slos:
    - name: "IncomingHttpRequestsPerSecond"
    properties:
        threshold: ""    # overrides default benchmark threshold.
        thresholdRelToLoad: 0.95    # set slo threshold relative to load
            value.
        warmup: 120    # override default value in benchmark; in seconds.
execution:
    strategy:    # set search strategy
    name: "RestrictionSearch"
    restrictions:
        - "LowerBound"
    searchStrategy: "LinearSearch"
    duration: 300    # total run duration per experiment in seconds.
    repetitions: 2    # repetitions per experiment.
configOverrides: []
```

Chapter 4

Evaluation

As stated in the previous chapter, the goal of the benchmarking process is to evaluate vertical scalability of different load generator tools.

To achieve this, we set up the load generator tools to generate appropriate workload and function within the Theodolite environment. We also configured the necessary `Benchmark` and `Execution resources` to enable the use of the Theodolite framework for our benchmarking purposes. Before discussing final results, the following section will explain the execution attributes of interest, the order of execution within the benchmarking process and give an overview of how the results are visualized.

4.1 Methodology

For our purposes, we modify available CPU resources of single load generator instances to evaluate the increase in performance with an increase in available resources.

4.1.1 Benchmarking Vertical Scalability

Once all components of Figure 3.4 exist in the cluster, the Theodolite operator will start the execution. It selects the first pair of resource and load values from the execution configuration (Listing 3.11) and deploys the SUT and load generator accordingly. Again, we define this single run of value pairs as an experiment.

As explained in Section 3.3.2, we test the selected load generators by utilizing both resource and load type to modify parameters of the LG Deployment. For our purposes, we are interested in single load generator instances with varying load intensities (messages per second) and hardware constraints (available CPU resources, measured in CPU units[1]).

The Theodolite operator queries data using PromQL queries. In the `Benchmark resource` example, we get the the desired result of received requests per second by applying the `rate()` function to the total number of received requests. As there are multiple instances of both the HTTP and Kafka SUT, the `sum()` function is used to aggregate the data from multiple instances.

Every execution uses the SLO threshold set to 95% of the configured load value (relative value). This means that a single experiment counts as successfull if at least 95% of the

[1]https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu

configured requests/messages per second are received at the corresponding endpoint.

After gathering the experiment data, the next experiment is selected based on the chosen search strategy. For our purposes, we chose the lower bound linear search for all executions, visualized in Figure 2.1.

Experiment duration and warmup time are set to 5 minutes and 2 minutes, respectively. According to Henning and Hasselbring [2022a], these values are a good fit to determine whether a SLO is met or not without wasting too much time on a single experiment. They also found that less than 5 repetitions are sufficient.

4.1.2 Result Visualization Overview

We use two types of plots to present the final benchmarking results. Both visualizations are created using jupyter notebooks [Kluyver et al. 2016]. The first type of plot is shown in Figure 4.1 and directly results returned by Theodolite. To create these plots, we use a modified version of Theodolite's analysis notebooks.[2] The notebook can read and analyze the result data stored in `.csv` files for every benchmark execution. Configured amount of workload is represented on the x-axis, in requests (or messages) per second. The y-axis represents the hardware resources available to the container, measured in CPU units, where 1 CPU unit = 1 CPU Core = $1000m$ CPU [milliCPU]. Each point in the plot represents a successfull experiment. This means that for each point in the plot, the median over received requests (or messages) per second of all repetitions of an experiment stayed at or above 95% of the configured load.

The second type of plot is shown in Figure 4.2 and represents data manually queried from Prometheus. First, use Grafana to store parts of a plot in one of the dashboards as a `.csv` file. This data is then processed by an additional jupyter notebook that generates the plot. The x-axis represents relative time values in seconds, always starting at 0. Absolut time markers are not provided, because we only use these values to refer to data in the plot. The y-axis represents the rate of received requests/messages at a given time during an experiment, also measured in requests (or messages) per second.

4.2 Results and Discussion

For the analysis of the final benchmarking results, we first discuss the results of each individual load generator before taking a look at the bigger picture.

4.2.1 Theodolite Load Generator

We start with the evaluation of the Theodolite LG, because it acts as the baseline for comparison of the other load generator tools. Note that each plot represents the best result(s) out of multiple executions.

[2]https://www.theodolite.rocks/running-benchmarks.html#accessing-benchmark-results

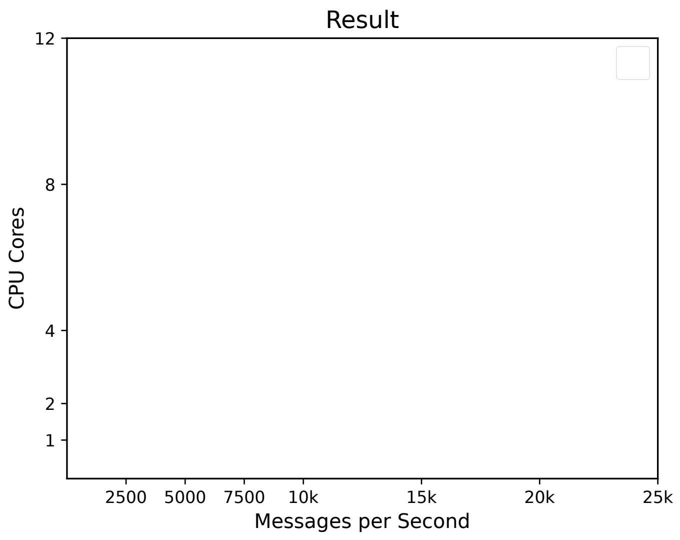


Figure 4.1. Plot frame for the visualization of Theodolite benchmark execution results. The x-axis represents the configured workload of the load generator, the y-axis CPU resources available to the Kubernetes container (in CPU units).

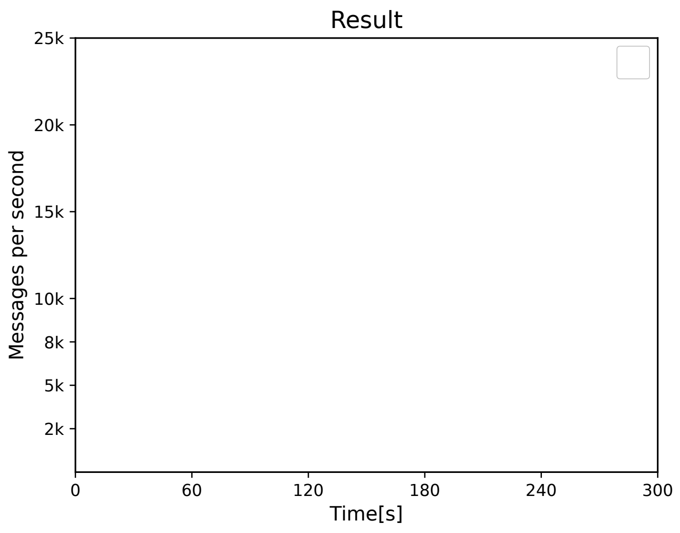


Figure 4.2. Plot frame for the visualization of manually queried data during benchmark executions. The x-axis represents relative time values of the queried interval, the y-axis rate of received requests/messages at a given time during an experiment.

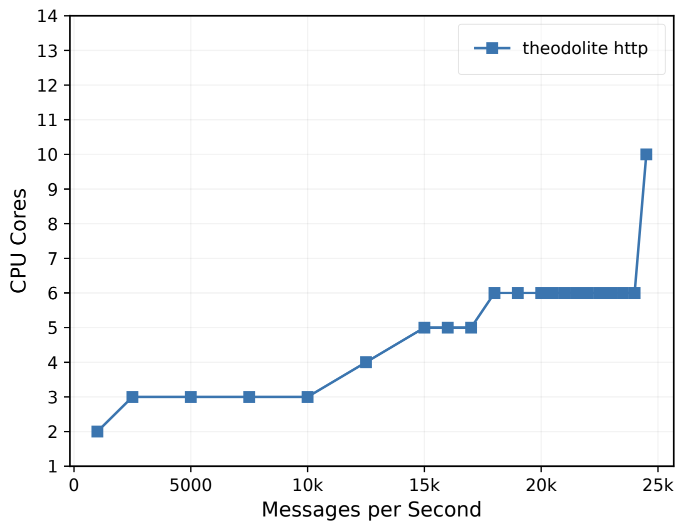


Figure 4.3. Theodolite LG results for the HTTP configuration.

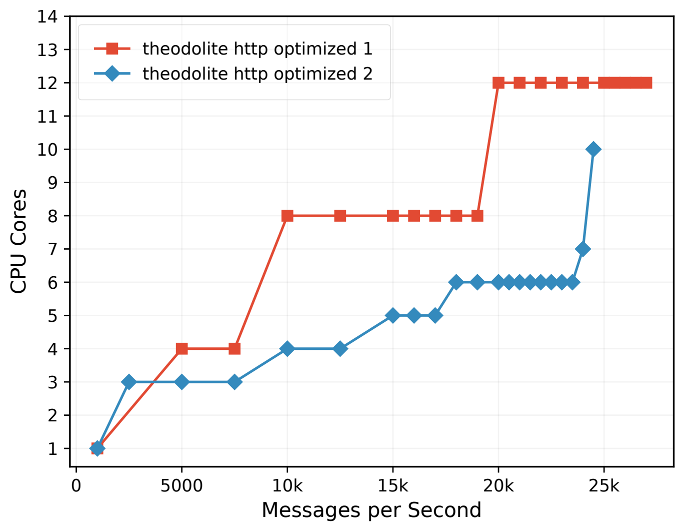


Figure 4.4. Optimized Theodolite LG results for the HTTP configuration.

The first observation when looking at the results of the Theodolite load generator is the distinction between normal and optimized executions. As explained earlier, the execution setup for load generators modifies workload and available CPU resources over the span of multiple experiments. All other parameters remain constant, including the configured amount of threads of a LG instance. In an experiment configuration with low available CPU resources, this could produce misleading results, because the container could start spending the majority of uptime time on thread switches instead of actual execution.

To bypass this problem, we differentiate between normal and optimized executions. The latter uses an additional Patcher to scale the number of threads of a load generator instance relative to its available CPU resources. Using this optimization, we prevent misleading results and provide as good of a baseline as possible for further comparison. A different ratio of threads and CPU cores, i.e. one additional thread for every 2 CPU cores, was considered but neglected because of the lack of performance improvements.

As not all load generator tools in this comparison support the concept of threads, we only apply this optimization to the Theodolite LG.

Figure 4.3 shows the results of the non-optimized configuration of the Theodolite LG running the HTTP benchmark. In the interval from 1,000 to 10,000 requests per second ($[1k - 10k]rps$), the load generator successfully provides the configured load with hardware requirements not exceeding 3 CPU cores. Up to $2.5k$ rps, just 2 CPU cores are sufficient. Within the interval $[10k - 20k]rps$ the CPU requirements increase from 3 to 6 cores almost linearly. The execution reaches its upper limit just short of $25k$ rps with a hardware requirement of 10 CPU cores.

In contrast, Figure 4.4 shows the optimized configuration. The plot contains two results representing the two best executions achieved. It is intended that `theodolite http optimized 1` has a steeper increase of hardware requirements, as it is an earlier experiment that only tested the CPU resource values $[1, 2, 4, 8, 12]$. We still include it, because it reaches the highest achieved throughput in the Theodolite LG HTTP configuration, surpassing $25k$ rps. `theodolite http optimized 2` represents the best overall Theodolite LG HTTP configuration regarding CPU resource usage and performance.

Notice that the results of both execution variations (normal and optimized) do not differ by a significant amount.

Figure 4.5 and Figure 4.6 show the normal and optimized version of the Theodolite LG Kafka variation, respectively. `theodolite kafka optimized 2` again represents a more recent experiment testing a finer range of resource values, while `theodolite kafka optimized 1` is another earlier experiment only testing the $[1, 2, 4, 8, 12]$ CPU resource values interval. The latter performs better in both maximum throughput and resource usage. This time, we observe more obvious differences between the normal and optimized version. `theodolite kafka optimized 1` generates load in the $[50k - 120k]$ messages per second (mps) and $[120k - 150k]mps$ intervals more efficient than the normal counterpart and reaches the highest observed throughput of these experiments, surpassing $160k$ mps as peak load generation capability.

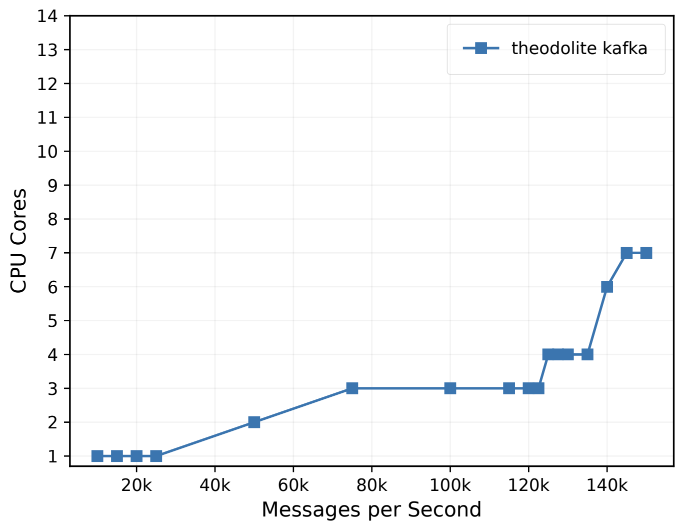


Figure 4.5. Theodolite LG results for the Kafka configuration.

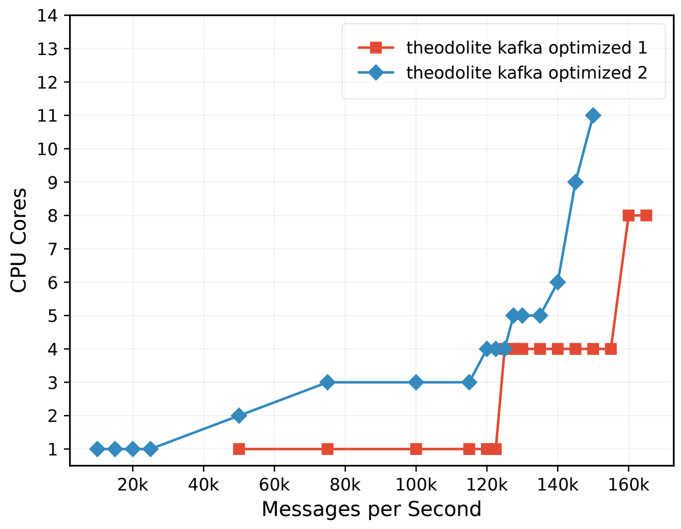


Figure 4.6. Optimized Theodolite LG results for the Kafka configuration.

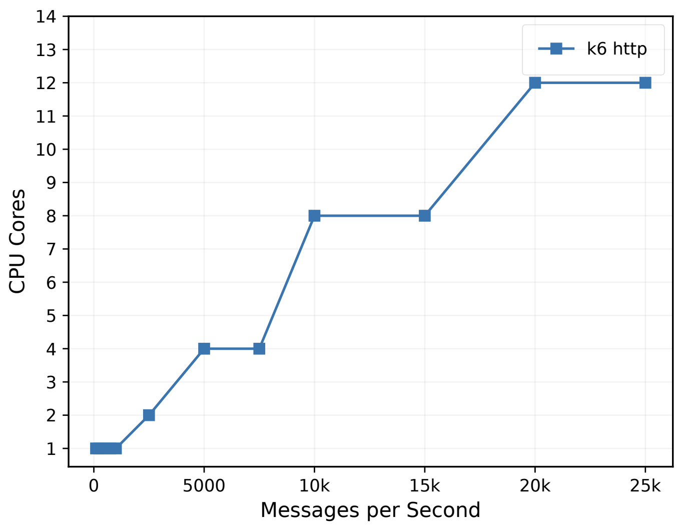


Figure 4.7. k6 results for the HTTP configuration.

4.2.2 k6

The results for k6 executing the HTTP configuration are quite similar to those of the Theodolite LG. Looking at Figure 4.7, `k6 http` reaches a peak load of 25*k rps*, close to the highest generated workload of both `theodolite http` and `theodolite http optimized` (1 and 2). It also shows similar resource scalability steps to `theodolite http optimized 1` in the tested interval.

Results of k6 Kafka executions (Figure 4.8) on the other hand differ greatly from the previously discussed results of Theodolite Kafka executions. k6 only generated roughly a tenth of the load of the Theodolite LG and required the highest available CPU resource configuration for doing so. `k6 kafka 1` represents the execution with the highest achieved throughput, while `k6 kafka 2` shows the most efficient run. The suspected reason for the performance result is the usage of a community extension to support the Kafka load format.

4.2.3 Gatling

Looking at the results of the Gatling HTTP executions Figure 4.9, we encounter the first of two unexpected results. Even though Gatling is a more modern load testing tool, the best execution only achieved a peak load generation of 1*k rps*. Multiple failed executions were recorded as well, in which the generated load would break down to 0 *rps* after an initial spike above the configured workload. Despite several debugging attempts and research of fixes for the most common problems when using Gatling, no better executions have been achieved.

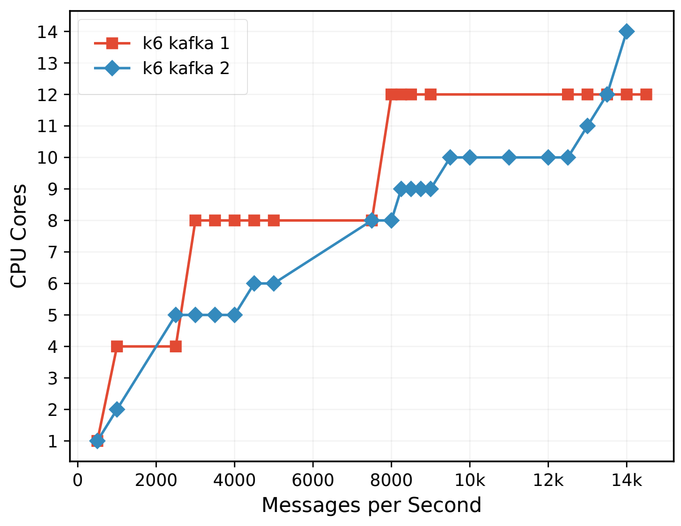


Figure 4.8. k6 results for the Kafka configuration.

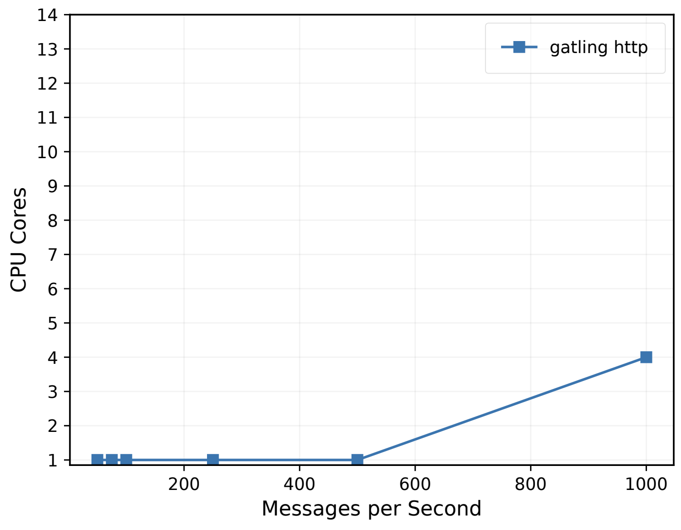


Figure 4.9. Gatling results for the HTTP configuration.

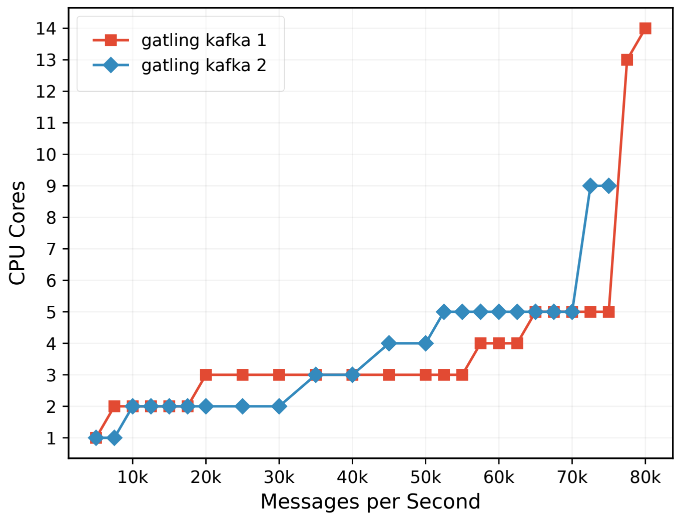


Figure 4.10. Gatling results for the Kafka configuration.

However, executing the Kafka configuration of Gatling Figure 4.10 yields great results. The top two executions compete closely with one another, with `gatling kafka 1` achieving workload peaks of $80k$ mps. Further analyzing `gatling kafka 1`, we observe a constant CPU resource requirement in the $[20k - 55k]mps$ interval. This is especially great in comparison to the higher resource requirements of k6 while generating significantly less load.

4.2.4 JMeter

Figure 4.11 shows the results for JMeter running the HTTP executions. It is obvious that with a peak generated workload of 500 rps, the JMeter HTTP runs are not really a competition for both the Theodolite LG and k6. But compared with the runs of Gatling, the results are quite similar, especially when comparing them on the workload interval of all load generators. The main difference is that Gatling reached the configured load before the rps crashed down to 0 indicating some error, while JMeter consistently underperformed past the workload of 500 rps.

Last and in this comparison certainly least, the results of the JMeter Kafka executions. This is the second unexpected result of the analysis. As the missing plot for JMeter Kafka runs indicates, none of the executions yieled successfull results when executed in the cluster. Despite various debugging attempts and research to fix the failing experiments as well as lowering the configured workload all the way down to 1 mps, the failure persistet. Figure 4.12 shows the successfull execution of the Jmeter Kafka configuration in a local cluster setup only used for testing purposes. Manually inspecting Kafka messages verified that the configuration works as intended within the local environment. Since no successfull

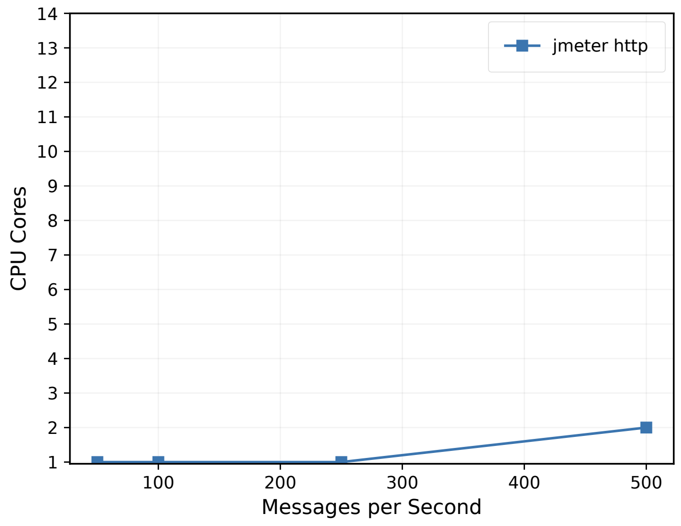


Figure 4.11. JMeter results for the HTTP configuration.

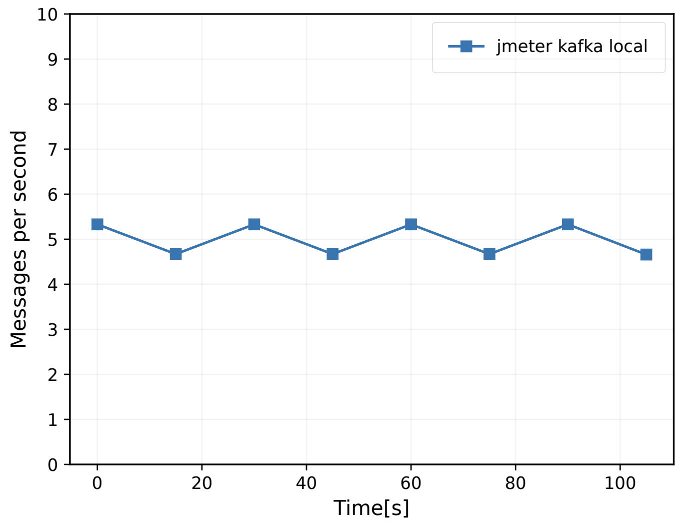


Figure 4.12. Manually queried data exported from the Grafana dashboard of a local cluster running on a laptop, used for testing purposes only. This data has not been generated in the cluster setup described in Section 3.1. This plot shows the generation of Kafka messages using JMeter in a local testing environment.

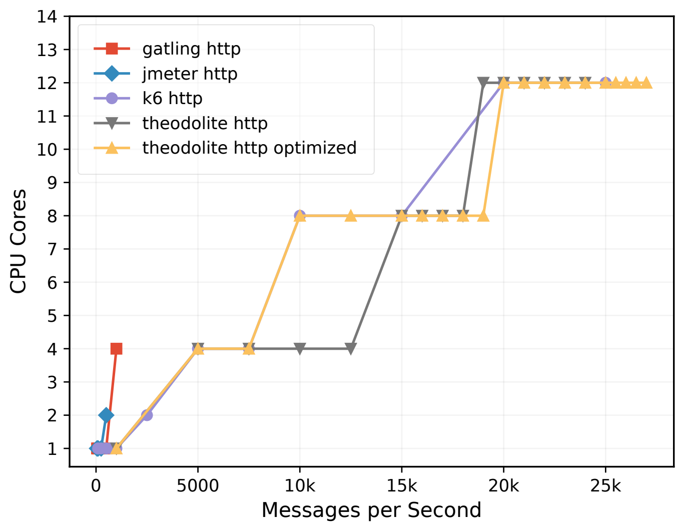


Figure 4.13. Results for all HTTP configurations combined, picking the best execution for each load generator prioritizing highest generated workload.

executions have been achieved in the cluster, we conclude no valid findings for the JMeter Kafka configuration.

4.2.5 Overall Comparison

Looking at the overall HTTP results (Figure 4.13 and Figure 4.14), both the Theodolite LG and k6 are close in comparison. Both tools scaled well with increasing CPU resources and reached comparable peak generated workloads, even though the Theodolite LG required less overall CPU resources to reach these values. Gatling is hard to compare, because the reason for the sudden stop of load generation remains unclear. JMeter performed stable during experiments and therefore scales worse with increasing CPU resources than its competitors.

The Kafka results (Figure 4.15) show a clear best performing tool. The Theodolite LG performed best in the use case it is specifically tailored for. Gatling showed a resonable performance, even though it requires significantly more CPU resources to generate less load. k6 did not scale as well with increasing resources when running the Kafka configuration, but still managed to perform decent. JMeter is hard to compare, as the reason for the execution failure remains unknown.

The results more or less stay the same when analyzing them from a multi-instance/distributed-Deployment perspective. It is expected to gain higher load testing throughput when using more than one load generator instance, as all tools in this comparison support distributed

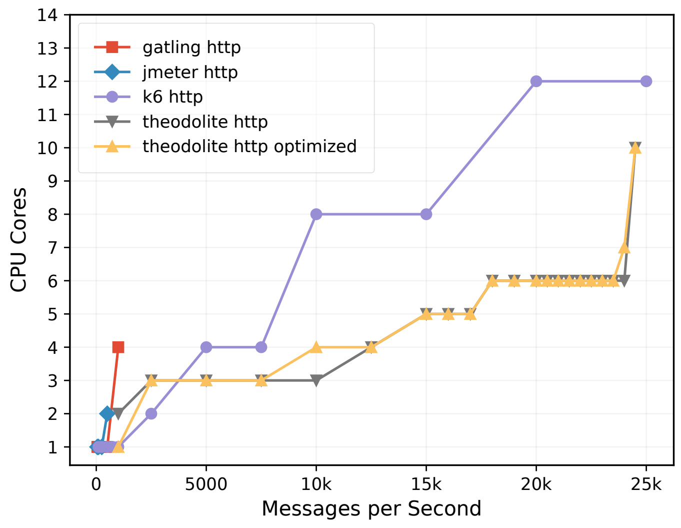


Figure 4.14. Results for all HTTP configurations combined, picking the best execution for each load generator prioritizing minimal CPU resource usage.

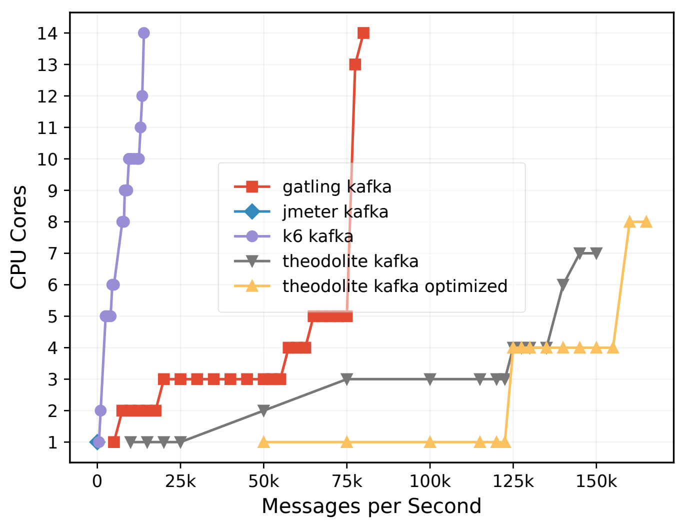


Figure 4.15. Results for all Kafka configurations combined, picking the best overall execution for each load generator.

testing in one way or another and therefore cover the basics challenges that come with horizontal scaling. However, vertical scalability can yield valuable inside about the number of LG instances required for a certain workload. This knowledge can be of special interest if set amounts of load have to be generated and available hardware resources are limited.

4.3 Threats to Validity

Load generator tools are complex tools that can be optimized in a wide variety of ways. However, this process takes a lot of time and requires in-depth knowledge about each tool and a lot of experience to utilize these possibilities. All load generators got configured by a bachelor level student, so they primarily follow default settings and minimal documentation examples customized to the specific use case. Configurations for each tool may not use optimizations that are considered standard practice in their respective domain.

Another considered factor is limiting the generated load by using endpoints that cannot handle the incoming load, therefore impacting results. To prevent this from happening, we increased the number of instances of the HTTP SUT during initial testing until no more changes with the increase of instances were identified. For the Kafka executions, the preexisting Kafka configuration within Theodolite was used, not indicating limiting functionality either.

Even though we conducted multiple executions for each configuration, therefore running several repetitions per experiment, the sample size is still small. Also considering all benchmarks were performed on one Kubernetes cluster with limited resources, we cannot guarantee that the results are free from any individual fluctuations.

Furthermore, the results only account for the described use case in this evaluation. They may vary for other use cases, especially when changing the load type (no HTTP/Kafka) or for significant changes in request/message size.

Chapter 5

Related Work

Henning and Hasselbring [2022a] presented the scalability benchmarking method used for the evaluation in this thesis.

Šmeral [2015] compared several tools for performance testing of web applications, including Gatling and JMeter. In their conducted tests, the Load Generation tools were executed on a dedicated physical machine equipped with two 4-core processors and 24GB memory. They evaluated the tools based on several web load formats, the closest to our comparison being the HTTP GET load format. Besides achieving overall higher throughput with their setup, the results also show comparable performance of Gatling and JMeter, with Gatling performing slightly better.

Lönn [2020], involved in the development of the load generator k6, presented a detailed comparison of a whole list of load generator tools with regards to maximum possible generated web load (HTTP) and memory usage. In the experiments, the load generator tools ran on a fanless 4-core Celeron server with 8GB memory. Therefore, the results represent the performance in testing environments with limited available resources, i.e. single low end machines. The results again show similar performance of both Gatling and JMeter with JMeter performing slighty better. k6 performed significantly better than both other tools. Only comparing the HTTP load format, our experiments yield the same results, with k6 performing best and Gatling and Jmeter performing quite similar.

Apte et al. [2017] presented AutoPerf, an automated load testing tool for web applications. In their work, they compare the scalability of their own approach with the existing load generator tools JMeter and Tsung, an open-source load testing tool written in Erlang.[1] Overall, they also found that JMeter performed worst when compared to the other existing load generator tool and their own approach.

[1]http://tsung.erlang-projects.org

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We defined three goals for this work in the first chapter. The first goal was to configure the three selected load generators k6, Gatling and JMeter to match the use case of the custom Theodolite LG. This goal also included the integration of the external tools in the Theodolite environment, so that they could be benchmarked using Theodolite methodology. We achieved this by providing scripts/test plans for every load generator and defined `Deployment resources` to prepare them for execution inside the cluster.

The second goal was to enable the Theodolite Operator to perform benchmarks using our selected load generators to generate results regarding the vertical Scalability of each tool. We defined new `Benchmark` and `Execution resources` for the Theodolite framework to use the configured LGs in experiments instead of the Theodolite LG. This allowed us to perform comparable load generator scalability benchmarks using established Kubernetes and Theodolite tooling.

The final goal was to execute the scalability benchmarks and analyze the results. We were able to find similarities and differences between the tools. Besides overall comparison of the tools, we also observed significant differences in the vertical scalability of each individual tool when comparing the results of both load formats HTTP and Kafka.

6.2 Future Work

In this thesis, we only compared vertical scalability of the selected load generator tools with regards to CPU resource contraints. It would be interesting to see if and how the results change when modifying more than one hardware resource constraint. One approach could be only changing the available memory, another the combination of both CPU and memory variations. The latter could also yield interesting results regarding the ideal proportion of available CPU and memory resources.

Another interesting comparison could be made when testing with different request/message sizes within the same load format, or expanding the evaluation and testing new load formats (not HTTP and Kafka).

Lastly, the evaluation could be continued by comparing multi-instance variations of the benchmark executions, verifying or denying the assumptions made and testing higher workload configurations.

Bibliography

[Apache Software Foundation 2021] Apache Software Foundation. *Apache JMeter*. 2021. URL: https://jmeter.apache.org (visited on 04/26/2023). (Cited on pages 8, 15, 16)

[Apte et al. 2017] V. Apte, T. Viswanath, D. Gawali, A. Kommireddy, and A. Gupta. AutoPerf: Automated Load Testing and Resource Usage Profiling of Multi-Tier Internet Applications. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE '17. L'Aquila, Italy: Association for Computing Machinery, 2017, pages 115–126. DOI: 10.1145/3030207.3030222. URL: https://doi.org/10.1145/3030207.3030222. (Cited on page 41)

[Burns et al. 2016] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, Omega, and Kubernetes. *Commun. ACM* 59.5 (Apr. 2016), pages 50–57. URL: https://doi.org/10.1145/2890784. (Cited on page 4)

[Cloud Native Computing Foundation 2016] Cloud Native Computing Foundation. *Prometheus*. 2016. URL: https://prometheus.io (visited on 04/28/2023). (Cited on page 4)

[Gatling Corp 2015] Gatling Corp. *Gatling*. 2015. URL: https://gatling.io (visited on 04/26/2023). (Cited on page 8)

[Grabovsky et al. 2018] S. Grabovsky, P. Cika, V. Zeman, V. Clupek, M. Svehlak, and J. Klimes. Denial of service attack generator in apache jmeter. In: *2018 10th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*. 2018, pages 1–4. DOI: 10.1109/ICUMT.2018.8631212. (Cited on page 1)

[Grafana Labs 2021] Grafana Labs. *k6*. 2021. URL: https://k6.io (visited on 04/26/2023). (Cited on page 8)

[Hashemian et al. 2012] R. Hashemian, D. Krishnamurthy, and M. Arlitt. Web workload generation challenges – an empirical investigation. *Software: Practice and Experience* 42.5 (2012), pages 629–647. DOI: https://doi.org/10.1002/spe.1093. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.1093. (Cited on pages 13, 17)

[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*. ICPE '21. Virtual Event, France: Association for Computing Machinery, 2021, pages 85–88. DOI: 10.1145/3447545.3451190. URL: https://doi.org/10.1145/3447545.3451190. (Cited on page 3)

[Henning and Hasselbring 2021b] S. Henning and W. Hasselbring. Theodolite: scalability benchmarking of distributed stream processing engines in microservice architectures. *Big Data Research* 25 (2021), page 100209. DOI: https://doi.org/10.1016/j.bdr.2021.100209. URL: https://www.sciencedirect.com/science/article/pii/S2214579621000265. (Cited on page 1)

[Henning and Hasselbring 2022a] S. Henning and W. Hasselbring. A configurable method for benchmarking scalability of cloud-native applications. *Empirical Software Engineering* 27.6 (Aug. 2022), page 143. DOI: 10.1007/s10664-022-10162-1. (Cited on pages 1, 3–6, 21, 25, 28, 41)

[Henning and Hasselbring 2022b] S. Henning and W. Hasselbring. Demo paper: benchmarking scalability of cloud-native applications with Theodolite. In: *2022 IEEE International Conference on Cloud Engineering (IC2E)*. 2022, pages 275–276. DOI: 10.1109/IC2E55432.2022.00037. (Cited on pages 1, 5, 7)

[Henning and Hasselbring 2022c] S. Henning and W. Hasselbring. *Theodolite documentation*. 2022. URL: www.theodolite.rocks (visited on 05/14/2022). (Cited on page 6)

[Herbst et al. 2013] N. R. 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)*. San Jose, CA: USENIX Association, June 2013, pages 23–27. URL: https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst. (Cited on page 3)

[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. In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Edited by F. Loizides and B. Scmidt. IOS Press, 2016, pages 87–90. DOI: 10.3233/978-1-61499-649-1-87. URL: https://doi.org/10.3233/978-1-61499-649-1-87. (Cited on page 28)

[Konkel 2023] C. Konkel. *Thesis artifacts for: benchmarking scalability of load generator tools*. May 2023. DOI: 10.5281/zenodo.7882286. (Cited on page 12)

[Kreps et al. 2011] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: a distributed messaging system for log processing. In: *Proceedings of the NetDB*. Volume 11. 2011, pages 1–7. (Cited on page 8)

[Lehrig et al. 2015] S. Lehrig, H. Eikerling, and S. Becker. Scalability, elasticity, and efficiency in cloud computing: a systematic literature review of definitions and metrics. In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*. QoSA '15. Montréal, QC, Canada: Association for Computing Machinery, 2015, pages 83–92. DOI: 10.1145/2737182.2737185. URL: https://doi.org/10.1145/2737182.2737185. (Cited on page 3)

[Lönn 2020] R. Lönn. *Open source load testing tool review 2020*. 2020. URL: https://k6.io/blog/comparing-best-open-source-load-testing-tools/ (visited on 03/04/2020). (Cited on page 41)

[Šmeral 2015] R. Šmeral. Modern performance tools applied. Master's thesis. Masaryk University, Faculty of InformaticsBrno, 2015. URL: https://theses.cz/id/xmalg6/. (Cited on page 41)

[The Kafka Authors 2022] The Kafka Authors. *Kafka event streaming platform*. 2022. URL: https://kafka.apache.org/documentation/#gettingStarted/ (visited on 11/25/2022). (Cited on page 8)

[V. Kistowski et al. 2015] J. V. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao. How to build a benchmark. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. Austin, Texas, USA: Association for Computing Machinery, 2015, pages 333–336. DOI: 10.1145/2668930.2688819. (Cited on page 3)