

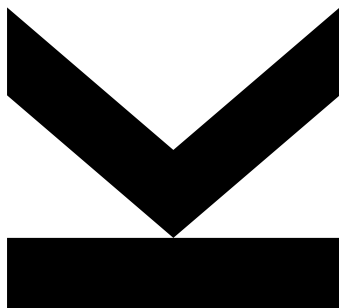
Author
Elias Berger
12209123

Submission
LIT Cyber-Physical
Systems Lab

Thesis Supervisor
Dr. Sören Henning
Univ.-Prof. Mag. Dr.
Rick Rabiser

January 2026

An Extensible Architecture for Integrating Observability Systems in Theodolite



Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Computer Science

Abstract

With modern applications getting increasingly more complex, monitoring and observability solutions are becoming increasingly used. Since the widespread adoption of microservices, scalability is an important measure of a system. The scalability benchmarking framework Theodolite is designed to test the scalability of such systems in controlled experiments. However it is still limited to Prometheus as its only data collection tool. The purpose of this thesis is to create an extension interface for different tools and to implement an alternative to Prometheus. Dynatrace was chosen for this second implementation of an observability backend. The newly implemented features were tested using the OpenTelemetry Demo application, which simulates an astronomy shop and users accessing it. These test results demonstrate the efficacy of the implementation and that different applications can now be experimented with using Dynatrace.

Contents

Abstract	ii
List of Figures	v
Listings	vi
1 Introduction	1
1.1 Motivation	1
1.2 Objectives and Approach	1
1.3 Thesis Structure	2
2 Background and Related Work	3
2.1 The Container Orchestrator Kubernetes	3
2.2 Site Reliability Engineering Metrics	3
2.3 The Theodolite Framework for Benchmarking Cloud Scalability	4
2.3.1 Metrics	4
2.4 Dynatrace and the Dynatrace Query Language	5
2.5 The OpenTelemetry Telemetry Data Collection Framework	5
3 An Interface for Different Observability Backends	7
3.1 Project Structure	7
3.1.1 Structure of SLO Classes	8
3.2 Limitations of the Current Architecture	8
3.3 New Design of Benchmark Structure	8
3.4 Properties of the New Implementation	10
3.4.1 The MetricRequestExecutor Interface	11
3.4.2 The MetricRequestExecutorFactory Class	13
3.4.3 The MetricQueryResponse Interface	13
3.4.4 The MetricQueryResponseFactoryMethod Interface	13
4 A Metric Backend Implementation for Dynatrace	14
4.1 The DqlRequestExecutor Class	14
4.2 The DqlPollResponse Class	15
5 Experimental Evaluation	18
5.1 The OpenTelemetry Demo	18
5.2 Experiment Setup	21
5.2.1 Used Hardware	21
5.2.2 Software Configuration	21
5.3 Results	24
5.4 Discussion	24
6 Related Work	26
7 Conclusion	27

Contents	iv
8 Future Work	29
Bibliography	30

List of Figures

2.1	Basic Structure of Theodolite's interactions	4
2.2	An example of a DQL query, showcasing the pipeline from raw logs to a table from https://docs.dynatrace.com/docs/discover-dynatrace/platform/grail/dynatrace-query-language/dql-reference#sequential-data-processing	5
3.1	Class diagram of existing classes related to the checking of SLOs	9
3.2	Sequence diagram of the process of retrieving metric data and checking SLOs	9
3.3	General architecture diagram for data collection and querying	10
3.4	Updated class diagram showing SLO-related classes	11
3.5	Updated sequence diagram of the process of retrieving metric data and checking SLOs	12
4.1	Sequence diagram showing the process of a DQL request	15
5.1	Architecture diagram for the evaluation of the Dynatrace specific implementation	19
5.2	A view of the OpenTelemetry Demo frontend	19
5.3	OpenTelemetry Demo microservice interactions [1]	20
5.4	Depiction of response time to load and resources	25
5.5	Maximum processable load for provisioned resources	25

Listings

3.1	Example YAML definition for metrics from Prometheus	7
3.2	Updated YAML definition for metrics from Dynatrace	10
3.3	Showcasing of the Singleton Factory	13
3.4	DqlPollResponse implementing the factory method	13
4.1	Method for handling dynamic JSON fields in the DqlPollResponse class .	16
4.2	JSON response for a DQL timeseries query	17
5.1	Benchmark definition for the SUT	22
5.2	Benchmark definition for the EnvVarPatcher modifying Locust users . . .	22
5.3	Benchmark definition showing the usage of the ReplicaPatcher for the frontend	23
5.4	Execution.yaml specification for this evaluation	23

Chapter 1

Introduction

1.1 Motivation

Scalability is widely regarded as a fundamental characteristic of cloud environments and applications. The term scalability in this case refers to a system being able to handle increasing amounts of load by utilizing more provided resources as described by Herbst et al. [14] and Lehrig et al. [15]. Theodolite [8] serves as a tool to test the scalability of these systems in controlled experiments specified by the user. To achieve this, Theodolite uses third-party observability platforms to estimate how well a system scales. However, Theodolite is currently restricted to just one of these tools, which is called Prometheus, and this is limiting the potential capabilities of Theodolite. Prometheus allows the collection of metrics from monitored sources, but does not include the collection of logs or traces (record of the lifecycle of a request). By having the flexibility of switching to different observability platforms, such as Dynatrace, we open up Theodolite to have access to the features of the platform. This work aims to achieve this through recommended software design principles described by Gamma et al. [7], such as the Abstract Factory Pattern, and apply these principles to extend Theodolite's capabilities. Specifically, the goal is to enable Theodolite to replace Prometheus with other Data Collection Services, such as Dynatrace's `OpenTelemetryCollector`, and query the processed data from Dynatrace. Therefore, enabling Theodolite's already existing features to be used with Dynatrace and potentially other Backends in the future. A list of goals to achieve this will be stated in the next section.

1.2 Objectives and Approach

The main objectives of this thesis are:

G1: Enabling the Use of Different Monitoring Backends With Theodolite

For this, a common interface is created that allows a backend to function with the existing Theodolite architecture. The currently implemented Prometheus backend should remain functional after the redesign, with no apparent changes noticeable to the user of Theodolite.

G2: Implementing a System Capable of Querying Dynatrace

On top of the newly implemented interface, Dynatrace should be added as an available backend for the system. Dynatrace is a commercial monitoring and observability backend

with an extended feature set. Specifically, relevant features of Dynatrace are as follows:

- Compatibility with the OpenTelemetry standard.
- The ability to use Dynatrace as a monitoring backend if it is already used in the experiment architecture.
- Log & Trace analysis, which is not enabled by the default Prometheus implementation.

G3: Evaluation of the New Implementation

To evaluate the functionality of the implemented interface, an experiment should be conducted. The evaluation setup consists of an OpenTelemetryCollector sending telemetry data to Dynatrace. As a system to be evaluated, the Astronomy Shop OpenTelemetry Demo was used.

1.3 Thesis Structure

The remainder of this thesis is structured like the following:

- Chapter 2 discusses important terms and fundamental knowledge to understand this thesis, as well as related work regarding existing frameworks.
- Chapter 3 provides an overview of the existing project and how it can be modified/extended to meet the goals.
- Chapter 4 highlights how the querying of the data is implemented for the Dynatrace platform.
- Chapter 5 shows results of the evaluation of the newly added functionality.
- Chapter 6 highlights previous work done on Theodolite and additional related work.
- Chapter 7 concludes the thesis and outlines future work.

Chapter 2

Background and Related Work

2.1 The Container Orchestrator Kubernetes

The use of container orchestration tools like Kubernetes continues to rise [5] due to the important qualities they provide to large-scale systems. Containerization of applications enables separation of applications from the underlying operating systems and environments of the machines they run on [4]. This separation is crucial for simplifying deployment due to environments remaining the same during development as well as deployment. Different containers don't interact with each other unless explicitly specified. Failure causes can be isolated from the host, enabling easier failure localization. Containerization also enables management software to manage applications rather than having to manage machines. Furthermore, developers do not have to think about specific interactions with different operating systems, and also simplify managing underlying infrastructure, as large systems can be operated using only a handful of operating systems. Due to the application-centric approach of containerization, metrics such as CPU consumption and memory usage can be tied to applications individually, improving application monitoring. Container interfaces enable communication with orchestration tools such as Kubernetes to monitor the health of applications/containers and restart them if needed, providing self-healing capabilities to running systems. Kubernetes handles the deployment of such application containers inside another resource management container called *Pod*. It is also possible to add additional supporting containers to the Pod (Sidecar Pattern), which run independently of the main application container.

2.2 Site Reliability Engineering Metrics

In their book about Site Reliability Engineering (SRE), Beyer et al. [2] define three terms:

- **SLI** Service Level Indicators are defined as quantitative measures/metrics of a service to be examined. Some examples would be: availability (how many requests to the service succeed) or request latency (how long an individual request takes to get a response back). SLIs are often aggregated over a time window.
- **SLO** Service Level Objectives take the previously defined SLI and set target values or target ranges to compare to. The result of an SLO is a Boolean value depending on whether the measured SLI falls below/above a value or inside/outside of a range. This comparison value has to be set by the designer of the system and can be a complex choice. An example of an SLO could be that the request latency is below 200ms.

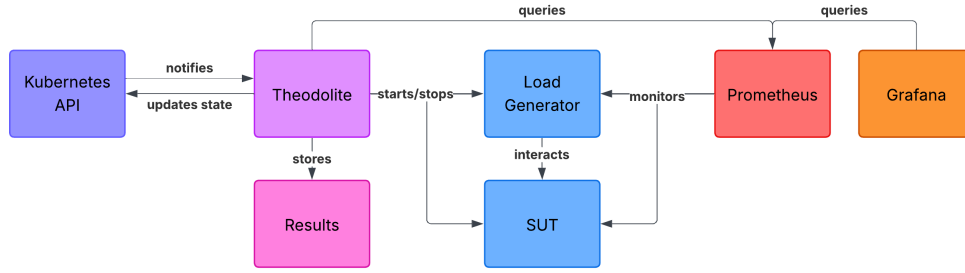


Figure 2.1: Basic Structure of Theodolite's interactions

- **SLA** Service Level Agreements define the consequences if an SLO is not met. This is not always necessary for some systems, as there can be no consequences to an SLO not being fulfilled. Examples could be a monetary fine to be paid.

2.3 The Theodolite Framework for Benchmarking Cloud Scalability

Theodolite is a framework designed to benchmark the horizontal and vertical scalability of cloud-native applications [8] running in Kubernetes. In this case, horizontal scaling refers to the addition of more computing nodes to a system, and vertical scaling to an increase in computing or processing power of the individual nodes [15]. Theodolite allows the user to specify benchmarks and executions as custom Kubernetes resources, which are defined in the framework as Kubernetes CRDs (Custom Resource Definitions) [13]. These benchmarks define a system under test, a Prometheus query (in the current version) to collect data from the system under test, as well as a load generator that interacts with the SUT to simulate users accessing the system. The basic structure of these interactions can be seen in Figure 2.1.

2.3.1 Metrics

Currently, Theodolite defines the scalability of a system depending on three factors [19].

- **Load Intensity** is the input the system receives. This could be the number of requests a system is receiving.
- **Provisioned Resources** are the systems' provided resources to handle a certain load. An example would be the number of containers provided by the system.
- **Service Level Objectives (SLOs)** determine if the system has failed or succeeded in processing a load intensity with its provisioned resources. Latency thresholds could exemplify such an SLO.

Henning and Hasselbring [8] define two metrics to measure the scalability of a system based on the above factors.

Resource Demand describes how many resources are at least required for some given load intensities, while all SLOs are still fulfilled.

Load Capacity describes how much load can at most be processed for some given provisioned resources, while all SLOs are still fulfilled.

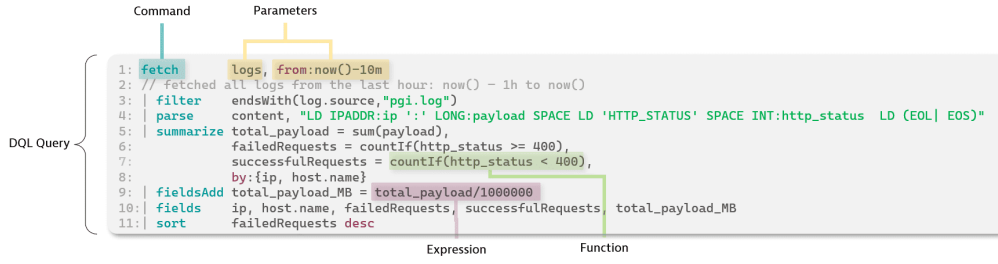


Figure 2.2: An example of a DQL query, showcasing the pipeline from raw logs to a table from <https://docs.dynatrace.com/docs/discover-dynatrace/platform/grail/dynatrace-query-language/dql-reference#sequential-data-processing>

More formally, we can define the ordered set of load intensities as $\mathbf{L}$, the ordered set of provided resources as $\mathbf{R}$ and the set of SLOs as $\mathbf{S}$. An SLO is defined as a function that maps to a boolean value $\text{slo}_s : \mathbf{R} \times \mathbf{L} \rightarrow \{\text{false}, \text{true}\}$ and $\text{slo}_s(\mathbf{r}, \mathbf{l}) = \text{true}$ with $\mathbf{r} \in \mathbf{R}, \mathbf{l} \in \mathbf{L}$ if the system fulfills the SLO with the given provided resources and load intensity.

Resource Demand can then be described as a function $\text{demand} : \mathbf{L} \rightarrow \mathbf{R}$

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

Load Capacity can then be described as a function $\text{capacity} : \mathbf{R} \rightarrow \mathbf{L}$

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

2.4 Dynatrace and the Dynatrace Query Language

Dynatrace is a platform used for monitoring, observing, and optimizing the performance of infrastructure and applications. It specifically focuses on cloud-native and microservice systems, allowing users to detect and resolve issues through the entire technology stack. In this research, Dynatrace serves as a monitoring backend, to which data is sent by an OpenTelemetryCollector, is processed, and then queried by Theodolite with the Dynatrace Query Language (DQL).

The Dynatrace Query Language [6] enables fetching data and processing it through a pipeline of subsequent commands. Each command in the pipeline returns a tabular output, which is then passed to the next command in the pipeline. An example of a DQL query can be seen in Figure 2.2. This work intends to use DQL to query data provided by an OpenTelemetryCollector running in the Benchmarking Cluster. The DQL provides features for Data Aggregation, which enables the output of an entire query to result in a single value, which could then be used to check if the defined SLOs are met. To keep consistency with the current implementation, this feature will not be used, and a series of values will be aggregated by Theodolite instead.

2.5 The OpenTelemetry Telemetry Data Collection Framework

OpenTelemetry is a project with the aim of providing an open source and vendor-/tool-agnostic observability framework [1, 20]. It focuses on the generation, export, and collec-

tion of telemetry data. The three main categories of data that OpenTelemetry collects are:

1. logs: logging messages which are emitted by the observed system
2. metrics: statistical representations of system state
3. traces: timed records of operations executed over multiple systems

OpenTelemetry collects this data from the individual sources and makes it possible to find correlations between them. Collecting this data serves as the building blocks for further analysis, which can be vendor-agnostic tools like Dynatrace. The terms telemetry and analysis together form what is called observability. Telemetry data is collected and then analysed by some other component to result in meaningful information about the state and ongoing processes of the observed system. In the case of this research, a telemetry collection tool would be the OpenTelemetryCollector and Dynatrace serves as an analysis tool.

Chapter 3

An Interface for Different Observability Backends

This chapter discusses the current Theodolite project structure and drafts for changes, as well as the implementations of the latter. The use of different monitoring backends should be enabled by the design. This refers to research goal **G1** mentioned in Section 1.2.

3.1 Project Structure

The existing Theodolite implementation uses Prometheus to collect metrics used for further evaluation [9]. Prometheus is an open source project that focuses on monitoring cloud infrastructure and provides metrics for Theodolite. It automatically discovers Kubernetes resources, such as Pods, and collects metrics and data from them. Furthermore, applications/services can provide HTTP endpoints for Prometheus to query directly. These endpoints can provide application-specific metrics if needed. The Prometheus server can be queried with the PromQL language, providing the collected metrics. These metrics are then aggregated according to the benchmark YAML configuration and used as an SLI in Theodolite. An example of a Theodolite benchmark YAML definition for the PromQL query, query aggregation, and SLO validation can be seen in Listing 3.1.

```
1  slos:
2    - sloType: "generic"
3      name: "uiLatency"
4      prometheusUrl: "http://prometheus-operated:9090"
5      offset: 0
6      properties:
7        externalSloUrl: "http://localhost:8082"
8        promQLQuery: "histogram_quantile(0.95,sum(irate(
9          osm_request_duration_ms_bucket{destination_name='teastore_webui'}[1m]))
10         by (le, destination_name))"
11        warmup: 600 #in seconds
12        queryAggregation: max
13        repetitionAggregation: median
14        operator: lte
15        threshold: 200
```

Listing 3.1: Example YAML definition for metrics from Prometheus

3.1.1 Structure of SLO Classes

The existing classes provide the functionality for executing queries against a Prometheus server running in the Theodolite Kubernetes cluster. The `AnalysisExecutor` is responsible for fetching data for a time frame of a previously executed experiment by calling the `MetricFetcher`, as well as saving the results and calling the `SloChecker` with them. The `MetricFetcher` calls a Prometheus server at the address provided in the configuration, which returns in JSON format. The JSON is mapped to the `PrometheusResponse` class with the Jackson library. The Creation of the `SloChecker` is handled by the `SloCheckerFactory`. The `SloChecker` checks if the aggregated metrics (SLI) fulfill the SLO according to the YAML configuration provided to Theodolite. This is done by first aggregating data according to `repetitionAggregation` and `queryAggregation` and then comparing the resulting value to `threshold` according to the `operator`. A class diagram of the discussed classes can be seen in Figure 3.1. The corresponding sequence diagram showcasing the interaction between classes in Figure 3.2.

3.2 Limitations of the Current Architecture

The current implementation does not support adding new ways to query data from sources other than Prometheus. The reworked code structure should enable the addition of new data sources, such as Dynatrace, using DQL. Furthermore, the current implementation defines the `prometheusUrl` (the URL where theodolite requests metrics from) and `offset` (a user-defined time offset for the metric queries) in the benchmark definition. This should ideally be moved into environment variables, since benchmarks should be sharable between users, and each user most likely needs a unique configuration for these variables. This removes the requirement for the user to edit the shared files and to only do local configuration through environment variables. Additionally, the property `promQLQuery` needs to be renamed. The following sections will elaborate on the details of the new implementation.

3.3 New Design of Benchmark Structure

To accommodate the aforementioned requirements, the new structure has removed some properties and renamed others. An updated section of the benchmark definition is shown in Listing 3.2. The used query evaluates the 90th percentile of request durations in a demo app, which will be elaborated on in Chapter 5

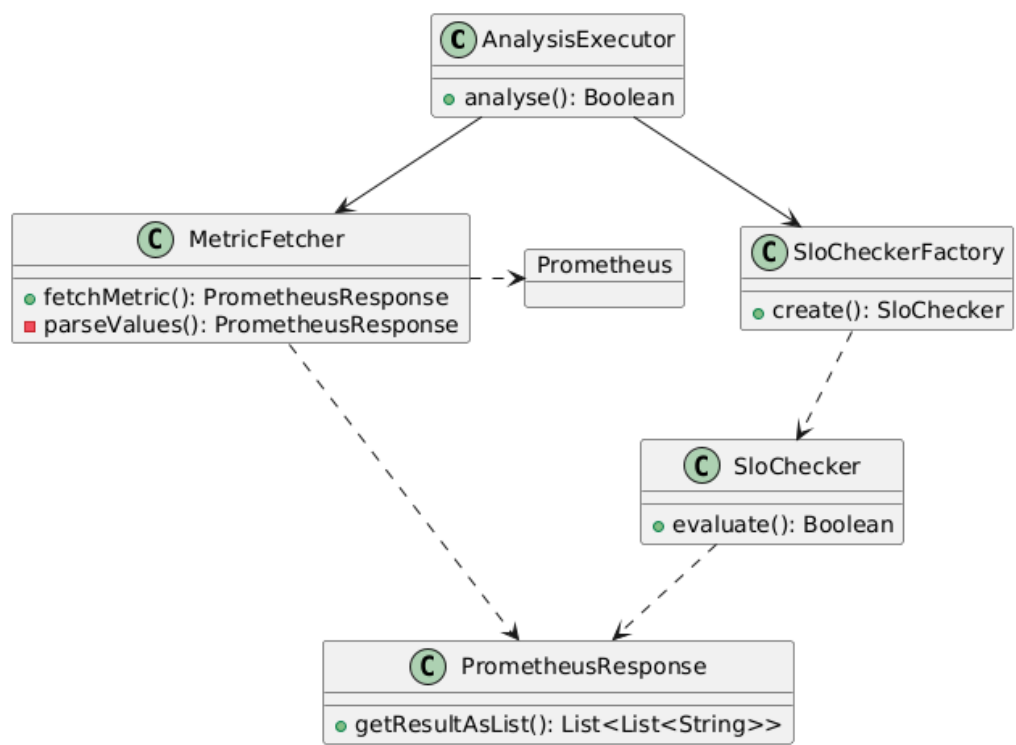


Figure 3.1: Class diagram of existing classes related to the checking of SLOs

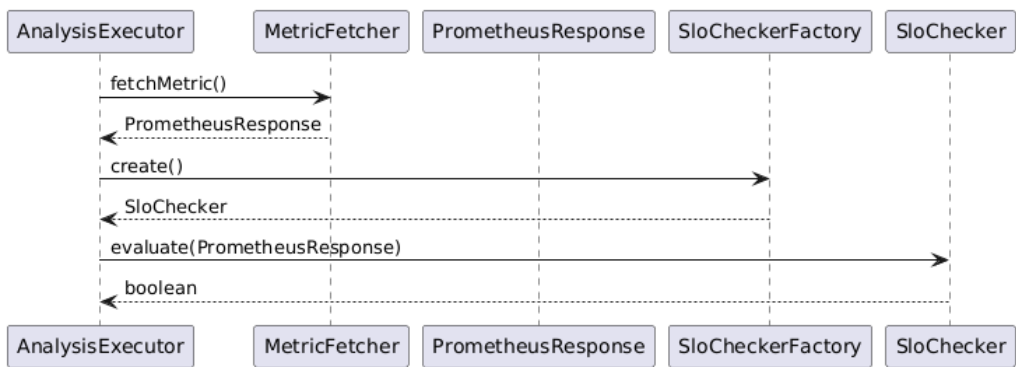


Figure 3.2: Sequence diagram of the process of retrieving metric data and checking SLOs

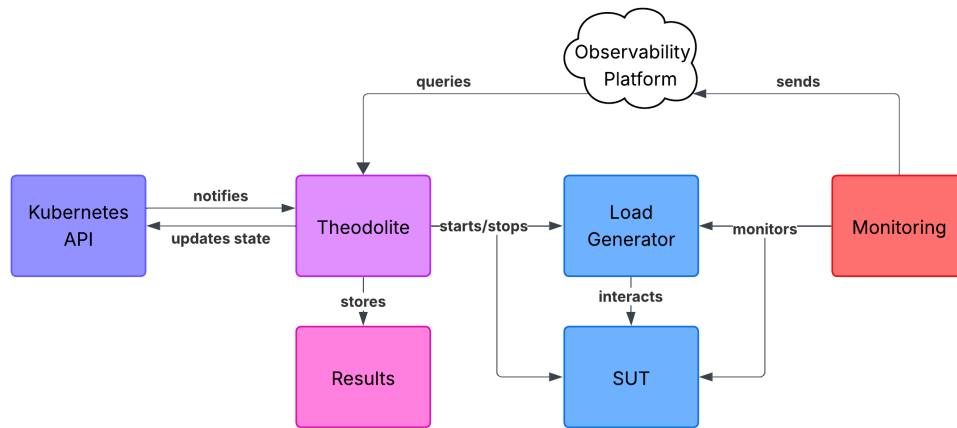


Figure 3.3: General architecture diagram for data collection and querying

```

1 slo:
2   - name: "p90duration"
3     sloType: "generic"
4     properties:
5       externalSloUrl: "http://localhost:80"
6       query: "fetch spans, samplingRatio: 10, scanLimitGBytes: 50
7         | filter request.is_root_span == true AND isNotNull(endpoint.name)
8         | makeTimeseries {p90 = percentile(duration, 90)},bins: 120,
9         by: { dt.system.sampling_ratio }"
10      provider: "dynatrace"
11      warmup: 60 # in seconds
12      queryAggregation: "mean"
13      repetitionAggregation: "median"
14      operator: "lte"
15      threshold: 500000000
  
```

Listing 3.2: Updated YAML definition for metrics from Dynatrace

3.4 Properties of the New Implementation

The new implementation should enable extensibility for future data sources. One implemented example would be querying Dynatrace servers with DQL, which will be discussed in the next chapter. For the collection and querying of metrics, the system architecture has to be modified. Instead of the Prometheus Server collecting data inside the cluster, different tools can be used, such as the `OpenTelemetryCollector` [3], which is used in Chapter 5. A variety of data is sent to the Dynatrace servers, such as metrics, traces, and logs. In the case of the query in Listing 3.2, the spans are queried, which are sub-components of traces, representing individual units of work. The collected data is then processed internally by Dynatrace and queried by Theodolite with DQL. An architecture overview for the new class structure can be seen in Figure 3.3.

The querying with DQL is achieved through interfaces such as the `MetricRequestExecutor`, `MetricQueryResponse` and Factories such as the

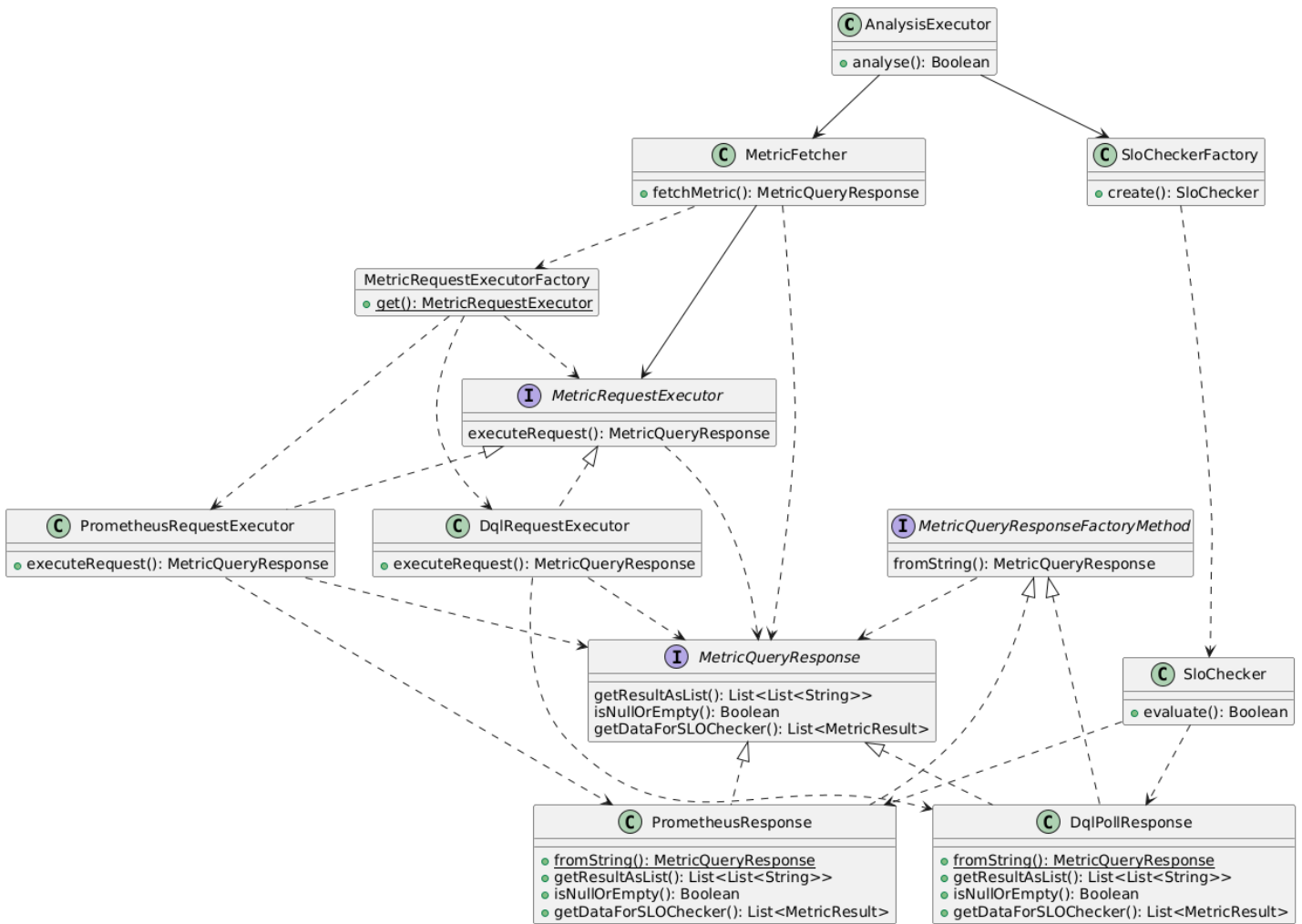


Figure 3.4: Updated class diagram showing SLO-related classes

`MetricRequestExecutorFactory` and `MetricQueryResponseFactoryMethod`, which can be seen in Figure 3.4. The following sections will elaborate on the mentioned classes.

3.4.1 The `MetricRequestExecutor` Interface

To provide a common Interface for the existing implementation and future extension, the `MetricRequestExecutor` interface features the method `executeRequest(uri: URI, query: String, offsetStart: Instant, offsetEnd: Instant, stepSize: Duration, timeout: Duration): MetricQueryResponse`. Any class implementing this interface, such as the `PrometheusRequestExecutor` or `DynatraceRequestExecutor`, is responsible for

1. Requesting metrics from a server using the supplied query
2. Returning an object implementing the `MetricQueryResponse` interface

The implementations of the `PrometheusRequestExecutor` and `PrometheusResponse` were transferred from the existing implementations. A second `RequestExecutor`, communicating with the Dynatrace backend, will be discussed later in Section 4.1.

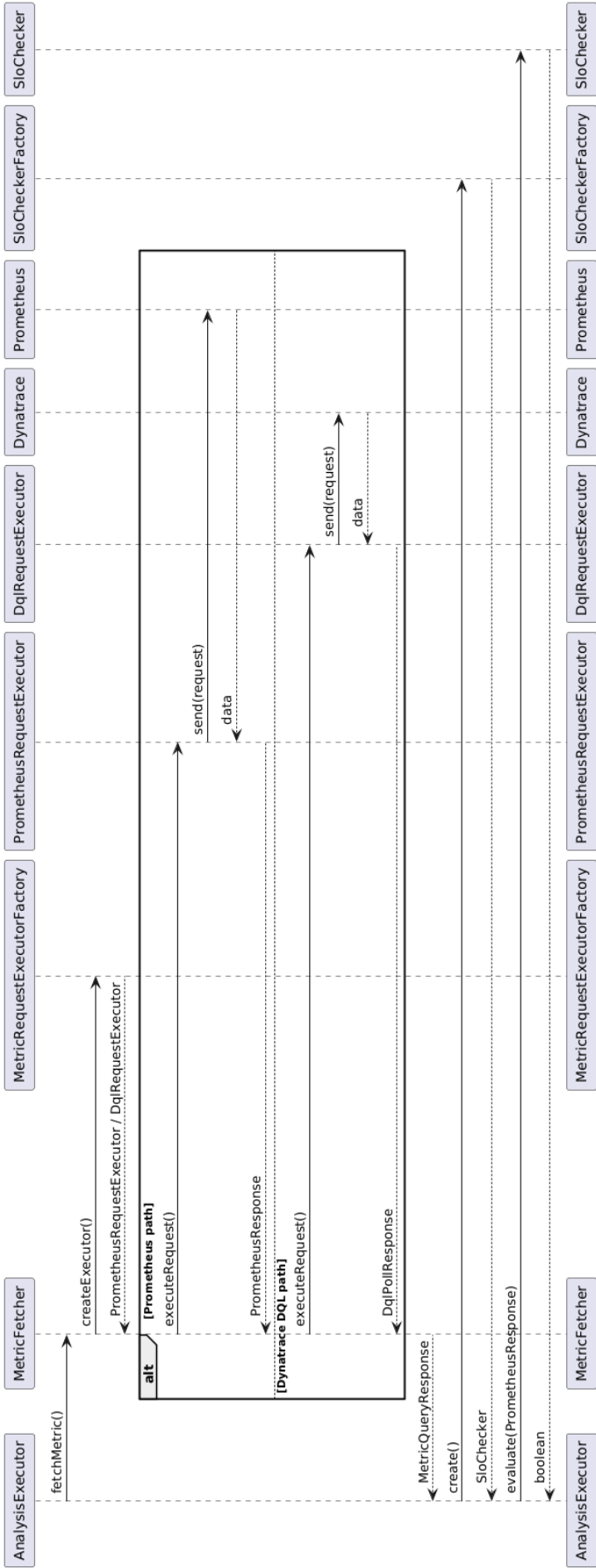


Figure 3.5: Updated sequence diagram of the process of retrieving metric data and checking SLOs

3.4.2 The MetricRequestExecutorFactory Class

Implemented as a Singleton, ensuring there is only one instance of this class, the `MetricRequestExecutorFactory` is responsible for the creation of a `MetricRequestExecutor` at runtime. Through the usage of the method `getOrPut` seen in Listing 3.3, the creation of a `RequestExecutor` is only executed when it is needed, avoiding the creation of all `Executors` at application startup [7].

```

1  object MetricRequestExecutorFactory {
2
3      private val executors: MutableMap<MetricFetcher.Kind,
        MetricRequestExecutor> = mutableMapOf()
4
5      fun get(kind: MetricFetcher.Kind): MetricRequestExecutor =
6          executors.getOrPut(kind) {
7              when (kind) {
8                  MetricFetcher.Kind.PROMETHEUS -> PrometheusRequestExecutor()
9                  MetricFetcher.Kind.DYNATRACE -> DQLRequestExecutor()
10             }
11         }
12     }

```

Listing 3.3: Showcasing of the Singleton Factory

3.4.3 The MetricQueryResponse Interface

The `MetricQueryResponse` interface serves as a common interface for the existing code. Specifically, the `AnalysisExecutor` needs the returned objects to be able to create lists for CSV files and for the `SloChecker`.

3.4.4 The MetricQueryResponseFactoryMethod Interface

Both the `PrometheusResponse` and the `DqlPollResponse` classes implement Kotlin companion objects [18], which implement the `MetricQueryResponseFactoryMethod` interface. A Kotlin companion object's methods can be accessed statically without the need for the parent object to be instantiated. Listing 3.4 shows the concrete implementation inside the `DqlPollResponse` class. In this case, the companion object specifically creates its parent. This serves as the primary way to create responses. This implementation pattern was chosen to enable the creation of responses with the `jacksonObjectMapper` from a JSON string, alongside providing a common interface for creation, ensuring extensibility [7].

```

1  companion object : MetricQueryResponseFactoryMethod<MetricQueryResponse> {
2      override fun fromString(json: String): DqlPollResponse{
3          return jacksonObjectMapper().registerModule(JavaTimeModule()).
            readValue(json)
4      }
5  }

```

Listing 3.4: `DqlPollResponse` implementing the factory method

Chapter 4

A Metric Backend Implementation for Dynatrace

To use Dynatrace with our newly implemented interfaces, two classes have to be implemented for the program to function. The `DqlRequestExecutor` discussed in Section 4.1 and the `DqlPollResponse` discussed in Section 4.2.

4.1 The `DqlRequestExecutor` Class

To execute DQL requests, the `DqlRequestExecutor` class is introduced, handling the authentication and querying. An authentication token must first be acquired to be able to send a query. This is done by sending an HTTP request to the authentication endpoint of the Dynatrace servers. A few parameters have to be specified for this request, including

Parameter	Value
client_id	required Obtained by generating an OAuth client in the Dynatrace App and serves as an identifier for the Dynatrace environment
client_secret	required Secret from the OAuth client
grant_type	required Is always set to <code>client-credentials</code>
scope	required Defines the permissions granted to the access token.
resource	required Set to <code>urn:dtaccount:{account-UUID}</code>

¹

These parameters are URL encoded and sent to the authentication endpoint of the Dynatrace environment. The response for this request has the following values:

Parameter	Value
token_type	required The type of token issued. Typically, the string <code>Bearer</code> .
resource	required Specifies the target resource or account context for which the token is valid.
access_token	required The actual token used to authenticate API requests, issued by the authorization server.
expires_in	recommended Specifies the token's lifetime in seconds.
scope	optional Defines the permissions granted to the access token.

²

¹<https://docs.dynatrace.com/docs/manage/identity-access-management/access-tokens-and-oauth-clients/oauth-clients#request-a-token>

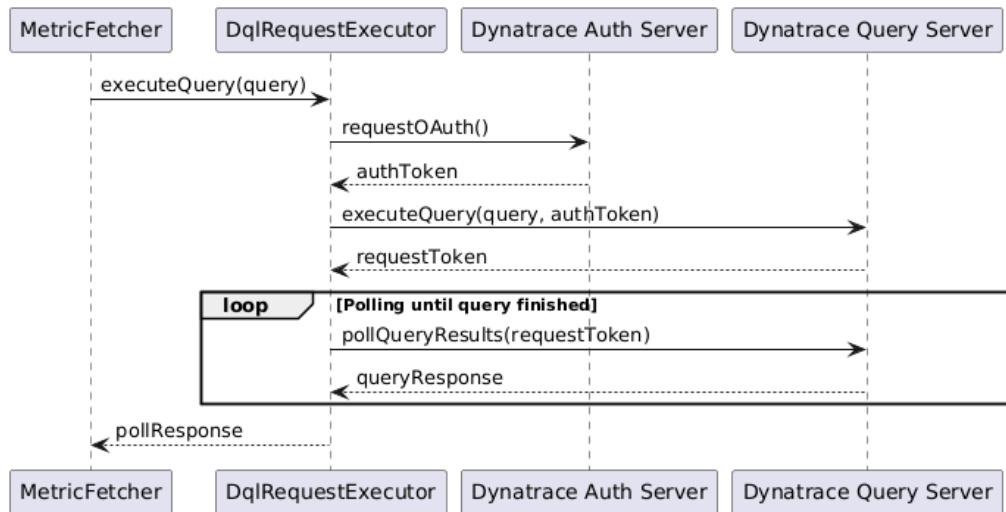


Figure 4.1: Sequence diagram showing the process of a DQL request

After obtaining the token, it is used in the HTTP authentication header for the query execution request. A successful query request will return another token that identifies the query as it is being processed at Dynatrace servers. This token is used to repeatedly poll the status of the query and obtain the result when it is finished.

To summarize, the class

1. requests an authentication token
2. executes the DQL query specified in the benchmark definition
3. polls query results

A visualization of this process can be seen in Figure 4.1

4.2 The DqlPollResponse Class

This class implements the interface `MetricQueryResponse` and provides a skeleton for response deserialization and implements the utility methods described in the `MetricQueryResponse` interface. A DQL response has a JSON body with varying formats; however, we assume the queries defined in the benchmark definition by the users of Theodolite are timeseries and have roughly the same shape. An example of a typical query response is shown in Listing 4.2. This response is then mapped onto the `DqlPollResponse` class as shown before in Listing 3.4. The `jacksonObjectMapper` will map key-value pairs of the JSON to the fields in the `DqlPollResponse` class. However, since the response can change dynamically based on the user-specified DQL query, some extra handling has to be done. The dynamic fields in the context of the example shown in Listing 4.2 are `dt.system.sampling_ratio` and `p90`. The method shown in Listing 4.1 is responsible for handling the dynamic fields and is annotated with the `@JsonAnySetter` annotation. This will automatically be called by the `jacksonObjectMapper` for every field not specified in the `DqlPollResponse` class. Further filtering is done to categorize the field as an array containing the values from the Theodolite experiment or a single value named after the group these array values belong to.

²<https://docs.dynatrace.com/docs/manage/identity-access-management/access-tokens-and-oauth-clients/oauth-clients#response>

```
1 @JsonAnySetter
2 fun handleDynamicFields(key: String, value: JsonNode) {
3     when {
4         value.isTextual -> {
5             grouping[key] = value.asText()
6         }
7         value.isArray -> {
8             val array = value as ArrayNode
9             val doubles = array.map { node ->
10                 if (node.isNull) null else node.asDouble()
11             }
12             metrics[key] = doubles
13         }
14         else -> {
15             throw IllegalArgumentException("Unexpected value type for key '
16                 $key': $value")
17         }
18     }
```

Listing 4.1: Method for handling dynamic JSON fields in the DqlPollResponse class

```
1  {
2    "records": [
3      {
4        "timeframe": {
5          "start": "2025-08-25T18:23:10.000000000+02:00",
6          "end": "2025-08-25T18:38:30.000000000+02:00"
7        },
8        "interval": "1000000000",
9        "dt.system.sampling_ratio": "10",
10       "p90": [
11         null,
12         4348087713.358979,
13         3535062127.9805803,
14         374538840.40694743,
15         260128251.29652727,
16         262745064.6606992,
17         ...
18       ]
19     }
20   ],
21   "types": [
22     {
23       ...
24     }
25   ],
26   "metadata": {
27     ...
28   }
29 }
```

Listing 4.2: JSON response for a DQL timeseries query

Chapter 5

Experimental Evaluation

This chapter evaluates the functionality of the newly implemented features by utilizing an existing microservice architecture. The experiment/system needs to be able to send telemetry data to Dynatrace servers via the OpenTelemetryCollector so the results can be queried afterwards. The following section provides further details on the chosen system under test and its functionalities. A load generator also has to be chosen to test the system.

5.1 The OpenTelemetry Demo

In order to run an experiment with Theodolite, a system under test has to be specified, which has to be in some way scalable. The OpenTelemetry project offers a demo, which includes a realistic distributed system specifically developed to collect data with OpenTelemetry. It even comes with a load generator to simulate real user traffic to the demo project. The system is designed to be an online webstore, which consists of multiple microservices running in a cluster. These microservices include:

- Accounting
- Advertising
- Shopping Cart
- Checkout
- Currency
- Email
- Flagd
- Fraud Detection
- **Frontend**
- Frontend Proxy
- Image Provider
- Kafka
- **Load Generator**
- **OtelCollector**
- Payment
- Product Catalog

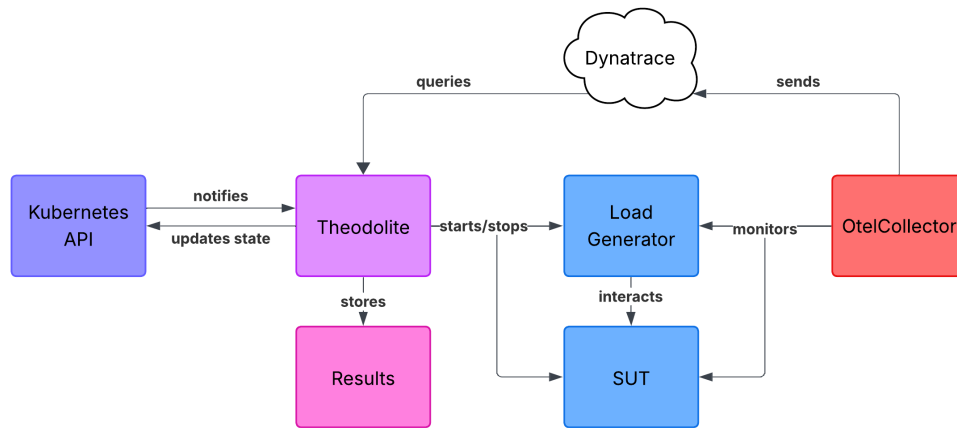


Figure 5.1: Architecture diagram for the evaluation of the Dynatrace specific implementation

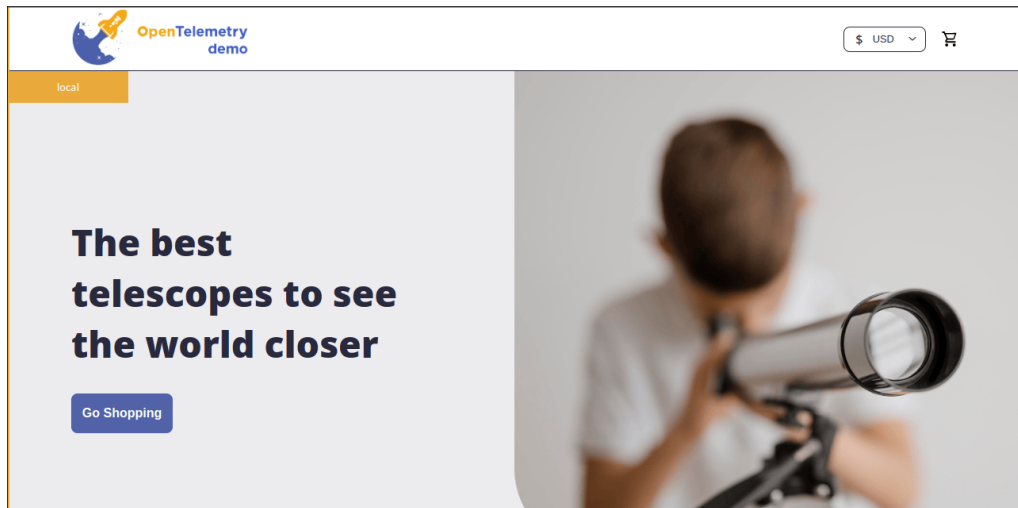


Figure 5.2: A view of the OpenTelemetry Demo frontend

- Quote
- Recommendation
- Shipping
- Valkey Cart

Relevant services for this research paper are highlighted.

As mentioned above, the demo comes with a configured `OpenTelemetryCollector`, which sends telemetry data to Dynatrace servers, such as metrics, traces, and logs. To provide visualization, the complete architecture is seen in Figure 5.1, a section of the frontend can be seen in Figure 5.2, and the underlying project structure and service interaction in Figure 5.3.

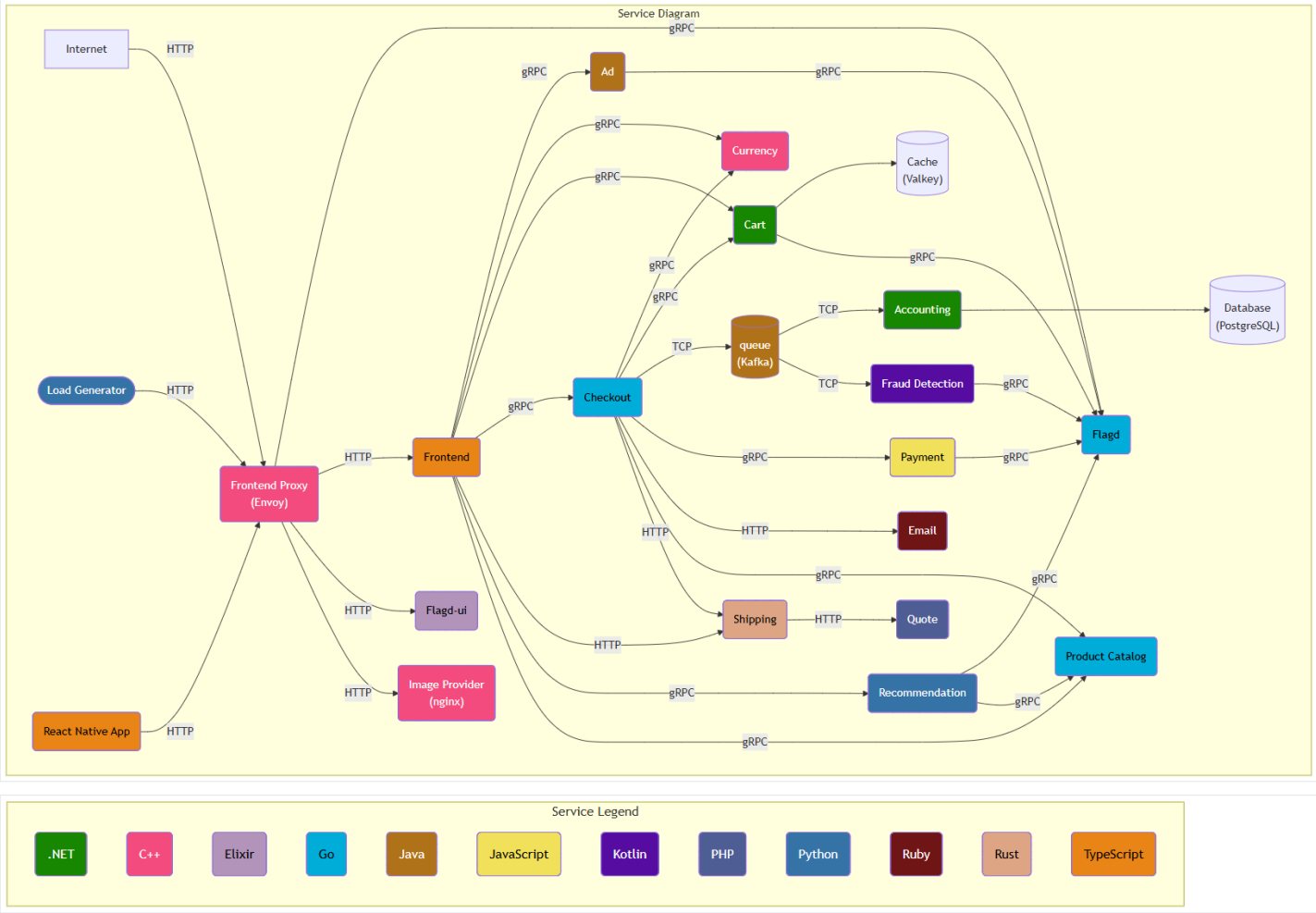


Figure 5.3: OpenTelemetry Demo microservice interactions [1]

5.2 Experiment Setup

This section describes the hardware and software setup, which was used to conduct an experiment to evaluate the newly implemented code. It specifies the used hardware as well as the configuration of the Kubernetes cluster, pods, and YAML files required for Theodolite.

5.2.1 Used Hardware

The experiment was first conducted once on an ASUS Vivobook 15 Pro with the following hardware:

Kind	Specification
Processor	AMD Ryzen 5 5600H, 3301Mhz, 6 Cores, 12 Logical Processors
RAM	16 GB with 3200 MT/s
Storage	SSD with 2254 MB/s maximum read speed

However, the results seemed to show some inconsistencies, as response times were very volatile, so the experiment was repeated and conducted successfully on the following hardware:

Kind	Specification
Processor	AMD Ryzen 7 3700X, 3600Mhz, 8 Cores, 16 Logical Processors
RAM	32 GB with 2133 MT/s
Storage	WD_BLACK SN7100 NVMe SSD with 7250 MB/s maximum read speed

5.2.2 Software Configuration

To enable Theodolite and the OpenTelemetry Demo to run, a Kubernetes cluster is needed. In the case of this evaluation, the kind¹ tool was utilized, which enables the user to start a Kubernetes cluster in a local Docker² container. The DNS configuration of the kind cluster was changed from the default to 8.8.8.8 and 4.4.4.4, as it was found that the DNS requests seem to be very inconsistent and slow with the default configuration. Theodolite was installed³ to the cluster with standard configuration via the helm⁴ tool (a package manager for Kubernetes). Furthermore, the OpenTelemetry Demo⁵ was also installed with a configuration that exports its data to Dynatrace.

To allow for the operation of the OpenTelemetry Demo project with Theodolite, the two deployment configurations, load intensity, and provisioned resources have to be addressed. The SUT needs to be scalable to allow more load from a load generator to be handled, and the load intensity from the load generator also needs to be able to be modified. This is achieved by deletion of the frontend pod and

¹<https://kind.sigs.k8s.io/>

²<https://www.docker.com/>

³<https://www.theodolite.rocks/installation.html>

⁴<https://helm.sh/>

⁵<https://www.dynatrace.com/news/blog/transform-opentelemetry-data-into-actionable-insights/>

the load-generator pod, to allow Theodolite to orchestrate these components. Listing 5.1 shows a section of the benchmark definition file used in this evaluation.

```

1 spec:
2   sut:
3     resources:
4       - configMap:
5         name: "otel-demo-configmap"
6         files:
7           - "frontend-deployment.yaml"
8   loadGenerator:
9     resources:
10      - configMap:
11        name: "otel-demo-configmap"
12        files:
13          - load-generator-deployment.yaml

```

Listing 5.1: Benchmark definition for the SUT

Since the load generator is implemented with Locust⁶, which simulates users actively using the web store of the OpenTelemetry Demo, only an environment variable of the load generator container has to be modified. Theodolite already provides the functionality with an `EnvVarPatcher` allowing for the modification of environment variables between experiments. A concrete specification for this evaluation is seen in Listing 5.2

```

1   loadTypes:
2     - typeName: "Users"
3       patchers:
4         - type: "EnvVarPatcher"
5           resource: "load-generator-deployment.yaml"
6           properties:
7             container: "load-generator"
8             variableName: "LOCUST_USERS"

```

Listing 5.2: Benchmark definition for the `EnvVarPatcher` modifying Locust users

In order to make the SUT handle the load from the load generator, the frontend pod is scaled to multiple instances. This is because it was found to be the main bottleneck for increasing user numbers. The other microservice pods are not scaled with theodolite in this case. Theodolite is able to scale pods via the `ReplicaPatcher`, which creates multiple instances of the provided deployment. The benchmark definition for the `ReplicaPatcher` is shown in Listing 5.3

⁶<https://locust.io/>

```

1 resourceTypes:
2   - typeName: "Instances"
3     patchers:
4       - type: "ReplicaPatcher"
5         resource: "frontend-deployment.yaml"

```

Listing 5.3: Benchmark definition showing the usage of the ReplicaPatcher for the frontend

The remainder of the benchmark definition specifies the SLOs for the experiments. An excerpt of this was already shown in Listing 3.2. The DQL query fetches all spans from the corresponding time frame of an experiment. Spans are individual units of work in a system, such as HTTP requests or database queries. A trace in OpenTelemetry is just a collection of these spans. Furthermore, the query extracts the request duration from these spans and aggregates the data in a timeseries. This function splits data into bins, each 120 seconds apart, and aggregates all values in that bin by determining the 90th percentile.

Scalability of the system is tested by providing different values for load intensity and resource capacity. This is specified in the `execution.yaml` file, where further values like the search strategy and duration of experiments can be specified. The chosen strategy in the case of this evaluation is `FullSearch`, so that values for every combination of Users and Instances are present. That way, the whole experiment can be evaluated.

To specify how long Theodolite waits before starting the load generator for each experiment, the variable `loadGenerationDelay` is used. The `duration` field determines the runtime for a single experiment. A full excerpt of the `execution.yaml` file is provided in Listing 5.4.

```

1 apiVersion: theodolite.rocks/v1beta1
2 kind: execution
3 metadata:
4   name: otel-demo
5 spec:
6   benchmark: otel-demo
7   load:
8     loadType: "Users"
9     loadValues: [5, 10, 15, 20, 25, 30]
10  resources:
11    resourceType: "Instances"
12    resourceValues: [1, 2]
13  slos:
14    - name: p90duration
15      properties: {}
16  execution:
17    metric: capacity
18    strategy:
19      name: "FullSearch"
20    duration: 600 # in seconds
21    loadGenerationDelay: 30 # in seconds
22    repetitions: 1

```

Listing 5.4: Execution.yaml specification for this evaluation

5.3 Results

As expected, the results show an increase in the response time as the system gets overloaded. This is then mitigated by providing more resources to the system, which can be seen clearly in Figure 5.4. The graph shows the response time of the system in seconds with respect to the load provided by the load generator. In this case the this refers to the number of Locust users concurrently accessing the frontend. The bars in red show results for 1 instance (frontend Kubernetes pod) and 2 instances in the blue color. It shows that the response time seems to scale approximately linearly with increasing load (Locust users) for both 1 and 2 instances. Furthermore, the response time is much lower with an increase in resources/instances. For a doubling in resources, the response time seems to be roughly halved.

Additionally, the threshold provided in the benchmark definition judges whether the system has failed the SLO or not. This results in a maximum load capacity for every resource count, shown in Figure 5.5. The X-axis depicts the number of instances/resources, in this case, we only have either one or two, and the Number of Locust users on the Y-axis. It can be seen that the largest number of Locust users that are processable with 1 instance are 5 and 15 for 2 instances. Therefore, this graph provides information about how the system scales with more load.

5.4 Discussion

By utilizing the OpenTelemetry Demo, the efficacy of the newly implemented features was depicted. This was achieved by allowing Theodolite to manage the creation and deletion of the load generator and the frontend Kubernetes deployments of the OpenTelemetry Demo. The different load and resource combinations were tested in the experiment, during which the OpenTelemetryCollector sent data to Dynatrace servers. Afterwards, Theodolite queried these results from the Dynatrace servers using the newly implemented code. Finally, the results were illustrated in graphs showing a nice linear increase in response time with increasing load and a decrease in response time with increased provisioned resources.

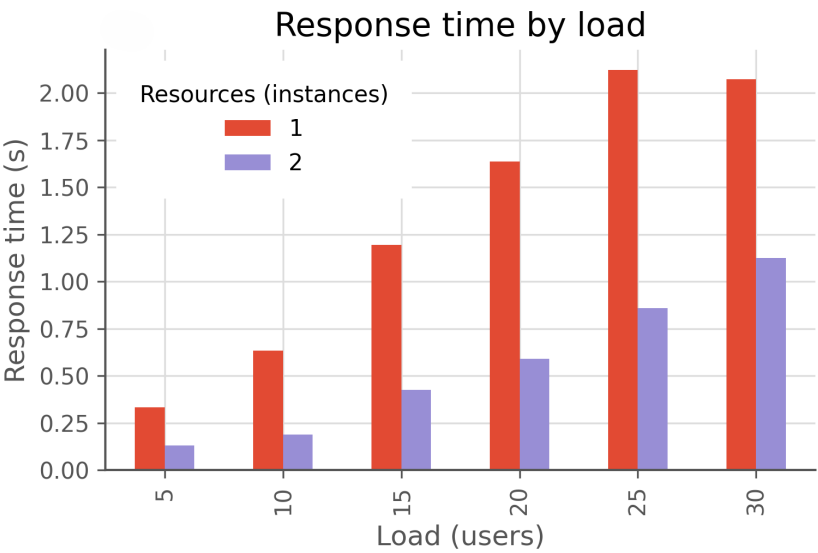


Figure 5.4: Depiction of response time to load and resources

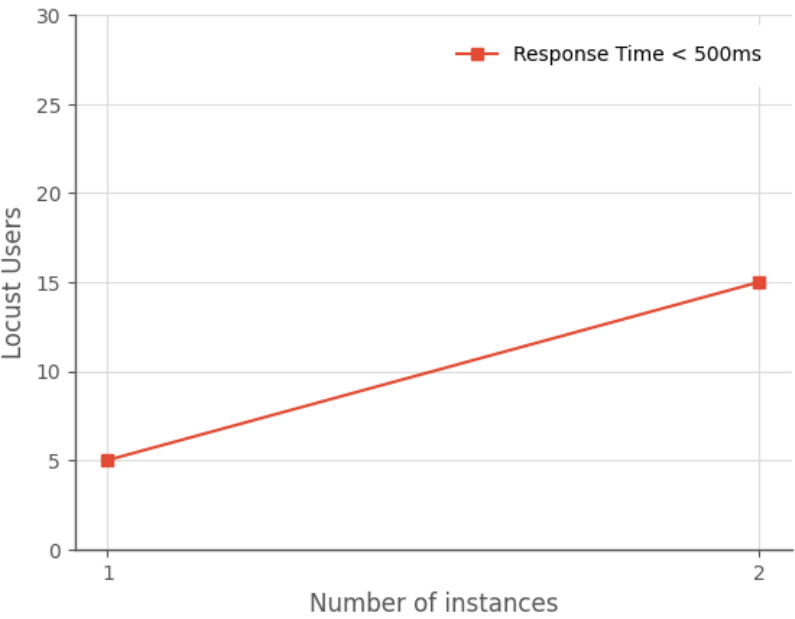


Figure 5.5: Maximum processable load for provisioned resources

Chapter 6

Related Work

In their 2021 paper [11], Henning and Hasselbring first introduced the Theodolite framework, which serves as the basis of this research. Along with another paper [10], the basic metrics that are used for the scalability experiments are defined.

Namely, the demand metric, which states the amount of resources that are at least required to process a given load. As well as the capacity metric, which maps resources to the maximum load they can process.

Furthermore, the papers “Cloud-Native Scalability Benchmarking with Theodolite: Applied to the TeaStore Benchmark” [12] and “Demo Paper: Benchmarking Scalability of Cloud-Native Applications with Theodolite” [9] discuss how Theodolite manages to benchmark systems by utilizing the Kubernetes Operator Pattern, which allows benchmarks to be created by declarative APIs. This is also where the distinction between benchmark and execution YAML files is first introduced, which were used and shown throughout this paper. The purpose of this separation is to allow benchmarks to be defined by a so-called benchmark designer, describing the SUT and load generator, and additional configurations of the benchmark. Executions refer to a concrete benchmark and describe a single execution of this benchmark. They describe the load and resource values, SLOs, the search strategy, and patchers. This was utilized in this paper for the modification of simulated users and the creation of multiple instances of the frontend.

Concerning the evaluation of a project with OpenTelemetry, Reichelt et al. [17] have also used the OpenTelemetry Demo to evaluate their approach. The paper attempts to bridge the gap between Kieker (an observability framework) and the OpenTelemetry standard, allowing Kieker to ingest and process the telemetry data collected by services like the OpenTelemetryCollector. This approach is very similar to the one showcased in this thesis, showing how the OpenTelemetry Demo application can be used to test and evaluate a variety of frameworks and applications.

A significant portion of research investigates observability and monitoring of cloud systems, where prior research investigates metrics collection, tracing, log analysis, and query languages for extracting performance insights from monitoring data. PromQL in Prometheus [16] being a prominent example. A systematic search through ACM Digital Library, IEEE Xplore, and Scopus yielded no results for research discussing the Dynatrace Query Language, highlighting that there is still room for public research regarding DQL.

Chapter 7

Conclusion

This work set out to achieve three research goals mentioned at the start in Chapter 1, which include

- **G1:** Enabling the use of different monitoring Backends with Theodolite
- **G2:** Implementing a system capable of querying Dynatrace
- **G3:** Evaluation of the new implementation

Goal **G1** was achieved by analyzing the current structure of the project and creating a common interface for different monitoring backends to interact with Theodolite. A special focus was placed on making the system easily extensible by applying common software design patterns. It is now possible to integrate new backends to the system with little modification to the core Theodolite system. Of course, the implementation is backwards compatible, still enabling the user to use Prometheus as if nothing had changed.

Goal **G2** was achieved by using this newly created interface to implement a secondary data source for Theodolite by accessing Dynatrace servers. This was done by authenticating to Dynatrace servers and then using the Dynatrace Query Language to query telemetry data. The HTTP responses of the servers were mapped onto data classes with the `jacksonObjectMapper`. Dynatrace enables the work with logs and traces, which are not currently supported by the base Prometheus implementation, extending the potential experiments able to be conducted with Theodolite. Furthermore, Theodolite can now operate in environments where Dynatrace is already part of the observability stack.

Goal **G3** was achieved by using the OpenTelemetry Demo, which comes with a full-fledged microservice demo application that serves as the system under test and a configured instance of the `OpenTelemetryCollector`, which sends its telemetry data to Dynatrace. An experiment was conducted using this application to collect data and query it with Dynatrace servers, showing the efficacy of the implementation. It demonstrated that the implementation can successfully query and process the data from Dynatrace, confirming that the implementation functions as intended, and new scalability experiments can be conducted using a non-Prometheus backend.

Together, the achieved goals provide a flexible and extendable architecture, laying a foundation for further integrations of additional monitoring backends.

The source code of the implementation can be found under <https://git.se.informatik.uni-kiel.de/elias.berger/theodolite/-/tree/dql-extension/theodolite/src/main/kotlin/rocks/theodolite/kubernetes/slo>.

Furthermore, the relevant YAML files for conducting the experiment with the OpenTelemetry Demo are placed in <https://git.se.informatik.uni-kiel.de/elias.berger/theodolite/-/tree/dql-extension/theodolite/examples/otel-demo-dynatrace>.

Chapter 8

Future Work

The project provides a basis for continued improvement and experimentation, such as:

- A cleaner separation of SLOs and SLIs would be desirable in future implementations, allowing for better clarity of the configuration. Currently, this is defined only as one configuration for an SLO, including the SLI configuration. Introducing distinct configuration entities for SLOs and SLIs would improve readability and make the overall configuration easier to maintain and extend.
- The system could be extended by integrating additional metric sources beyond the currently supported ones. This could include Kieker ¹ or Zipkin ² as examples for observability backends
- Benchmarking based on data only available with Dyntrace and not Prometheus. Trace and log data are not supported by the default Prometheus implementation, which is now possible using OpenTelemetry and Dynatrace.

¹<https://kieker-monitoring.net/>

²<https://zipkin.io/>

Bibliography

- [1] OpenTelemetry Authors. 2025. OpenTelemetry. Retrieved 10/20/2025 from <https://opentelemetry.io/docs/>.
- [2] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering: How Google Runs Production Systems*. (1st ed.). O'Reilly Media, Inc.
- [3] Alex Boten and Charity Majors. 2022. *Cloud-Native Observability with Open-Telemetry*. Packt Publishing Ltd, USA.
- [4] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM*, 59, 5, (April 2016), 50–57. DOI: 10.1145/2890784.
- [5] CNCF. 2025. CNCF-Annual-Report-2024. Retrieved 05/28/2025 from https://www.cncf.io/wp-content/uploads/2025/04/CNCF-Annual-Report-2024_v2.pdf.
- [6] Inc. Dynatrace. 2025. Dynatrace Query Language. Retrieved 05/15/2025 from <https://docs.dynatrace.com/docs/discover-dynatrace/references/dynatrace-query-language>.
- [7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., USA.
- [8] Sören Henning and Wilhelm Hasselbring. 2022. A configurable method for benchmarking scalability of cloud-native applications. *Empirical Softw. Engg.*, 27, 6, (November 2022), 42 pages. DOI: 10.1007/s10664-022-10162-1.
- [9] Sören Henning and Wilhelm Hasselbring. 2022. Demo Paper: Benchmarking Scalability of Cloud-Native Applications with Theodolite. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 275–276. DOI: 10.1109/IC2E55432.2022.00037.
- [10] Sören Henning and Wilhelm Hasselbring. 2021. How to Measure Scalability of Distributed Stream Processing Engines? In *Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE '21)*. Association for Computing Machinery, Virtual Event, France, pp. 85–88. DOI: 10.1145/3447545.3451190.
- [11] Sören Henning and Wilhelm Hasselbring. 2021. Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines in Microservice Architectures. *Big Data Research*, 25, 100209. DOI: <https://doi.org/10.1016/j.bdr.2021.100209>.
- [12] Sören Henning, Benedikt Wetzels, and Wilhelm Hasselbring. 2023. Cloud-Native Scalability Benchmarking with Theodolite: Applied to the TeaStore Benchmark. *Softwaretechnik-Trends*, 43, 1, (February 2023), 23–25. (Proceedings of the 13th Symposium on Software Performance (SSP 2022)).
- [13] Sören Henning, Benedikt Wetzels, and Wilhelm Hasselbring. 2021. Reproducible Benchmarking of Cloud-Native Applications With the Kubernetes Operator Pattern. In *Symposium on Software Performance 2021*.

- [14] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. 2013. Elasticity in Cloud Computing : What it is, and What it is Not. In *Proceedings of the 10th International Conference on Autonomic Computing, ICAC '13, San Jose, California, USA, 26th - 28th June 2013*. Advanced Computing Systems Association (USENIX), pp. 23–27.
- [15] Sebastian Lehrig, Hendrik Eikerling, and Steffen Becker. 2015. 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)*. Association for Computing Machinery, Montréal, QC, Canada, pp. 83–92. DOI: 10.1145/2737182.2737185.
- [16] Julien Pivotto and Brian Brazil. 2023. *Prometheus: Up & Running*. O'Reilly Media, Inc., USA.
- [17] David Georg Reichelt, Shinhyung Yang, and Wilhelm Hasselbring. 2025. Interoperability From OpenTelemetry to Kieker: Demonstrated as Export from the Astronomy Shop. *Softwaretechnik-Trends Band 45, Heft 4*.
- [18] JetBrains s.r.o. 2025. Kotlin. Retrieved 08/27/2025 from <https://kotlinlang.org/docs/object-declarations.html#object-declarations-overview>.
- [19] Andreas Weber, Nikolas Herbst, Henning Groenda, and Samuel Kounev. 2014. Towards a Resource Elasticity Benchmark for Cloud Environments. In *Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability (Hot-TopiCS '14)*, Article 5. Association for Computing Machinery, Dublin, Ireland, 8 pages. DOI: 10.1145/2649563.2649571.
- [20] Ted Young and Austin Parker. 2024. *Learning OpenTelemetry*. O'Reilly Media, Inc., USA.