FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Evaluation of service compliance with SLO before production release
Student:	Bc. Peter Žáčik
Supervisor:	Ing. Martin Beránek
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

One of the most challenging issues facing backend service development is compliance with the objectives introduced by the SLO (Service-level Objectives). Those objectives could be error rate, latency, availability in time and more. Deciding if service release will follow objectives once in production is generally tricky.

1. Analyze the current state of proactive service testing before releasing to production.
2. Implement a tool that generates tests of common SLOs (availability, latency, error rate, ...) from API description (Swagger, proto files, ...):
   a. Generated tests must be usable in the Continous Delivery system and mark a service invalid once it doesn't pass the given objectives.
   b. The test format should be a configuration of commonly used benchmarking tools (K6, Locust, Gatling benchmark).
3. Test the tool implementation on an example application deployment with basic SLO values of 99.99% availability, 1% error rate and 100 ms latency:
   a. The application has to be able to simulate the SLO values.

It is not required to develop a benchmark tool, yet it is imperative to bridge the gap between benchmark development and SLO service description. The tool has to help validate the service and free the developer from the time of benchmark preparation.

Electronically approved by Ing. Michal Valenta, Ph.D. on 29 September 2022 in Prague.

Master’s thesis

Evaluation of service compliance with SLO before production release

Bc. Peter Žáčik

Department of Software Engineering
Supervisor: Ing. Bc. Martin Beránek

May 4, 2023

Acknowledgements

I would like to thank my supervisor, Ing. Bc. Martin Beránek for the time he dedicated to this project and his helpful insight into the world of practical software engineering.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 4, 2023 …………………

Czech Technical University in Prague
Faculty of Information Technology
© 2023 Peter Žáčik. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Žáčik, Peter. *Evaluation of service compliance with SLO before production release.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Tato práce prezentuje implementaci softwarového nástroje poskytujícího automatické vyhodnocování cílů SLO. Nový nástroj, Perfcheck, využívá specifikace OpenAPI ke generování přizpůsobitelného benchmarkového testu pro webové služby. Perfcheck navíc poskytuje integraci s platformou Google Cloud. Nástroj je použitelný v Continuous Delivery (CD) systému. V práci je popsán proces návrhu a implementace nástroje Perfcheck. Nástroj je Open source a je k dispozici pro instalaci na více platformách.

Klíčová slova service-level agreement, service-level objective, testování softwaru, výkon aplikací, latence, implementace, automatizace, statistická analýza, Golang, Google Cloud, OpenAPI

Abstract

This thesis presents the implementation of a software tool providing automatic service-level objective (SLO) evaluation. The new tool, Perfcheck, utilizes the OpenAPI specification to generate a customizable benchmark test for HTTP web services. Alternatively, Perfcheck integrates with the Google Cloud Platform. The tool is usable in a Continuous Delivery (CD) system. The process of designing and implementing Perfcheck is described in the thesis. The tool is open source and is available for installation on multiple platforms.

Keywords service-level agreement, service-level objective, software testing, application performance, latency, implementation, automatization, statistical analysis, Golang, Google Cloud, OpenAPI

Contents

Introduction **1**

1 Goal **3**

2 Software testing **5**
2.1 Functional testing 5
2.2 Non-functional testing 6
2.3 SLA, SLO, SLI 6
2.3.1 SLO examples 7
2.3.2 Error budget 7
2.3.3 SLO evaluation 8
2.4 Pre-release service testing 9
2.4.1 Staging environment 9
2.4.2 Blue-green deployments 9
2.4.3 Canary deployments 10
2.4.4 Service evaluation in a serverless environment 10

3 State-of-the-art **11**
3.1 Service monitoring 11
3.1.1 Kubernetes Dashboard 11
3.1.2 Prometheus 12
3.1.3 NGINX Amplify 12
3.1.4 Google Cloud Platform SLOs 12
3.1.5 Other cloud providers 13
3.2 Service evaluation 13
3.2.1 Artillery 14
3.2.2 Gatling 15
3.2.3 JMeter 16
3.2.4 Grafana k6 17
3.2.5 Locust 19

3.2.6 Benchmark tools summary 19

4 Analysis **21**
4.1 Target audience 21
4.2 Target SLIs 21
4.3 Requirements 22
4.3.1 Functional requirements 22
4.3.2 Non-functional requirements 23
4.4 Use cases 24

5 Design and Architecture **27**
5.1 Benchmarking tool 27
5.2 Language 27
5.2.1 Go templates 28
5.3 Input and Output 28
5.3.1 `generate` 30
5.3.2 `test` 31
5.3.3 `get-template` 31
5.4 SLO parsing 31
5.4.1 OpenAPI SLOs 31
5.4.2 Google Cloud SLOs 34
5.5 SLO evaluation 34
5.6 Architecture 35
5.7 Release management 35
5.7.1 GoReleaser 36

6 Implementation **39**
6.1 Environment 39
6.2 License 40
6.3 Command-line interface 40
6.4 SLO parsing 41
6.4.1 OpenAPI SLOs 43
6.4.2 Google Cloud SLOs 43
6.5 Benchmark template 43
6.5.1 Thresholds 44
6.5.2 Virtual users and duration 44
6.5.3 The response size metric 45
6.5.4 Parameter generation 45
6.6 Test execution 46
6.7 Statistical analysis 46
6.8 Documentation 49
6.9 Release management 50

7 Examples **53**

7.1 OpenAPI 53
7.2 Google Cloud 54
7.3 Statistical analysis 55
7.4 Deployment pipeline 56
7.5 Examples summary 57

Conclusion **59**

Bibliography **61**

A Acronyms **65**

B Contents of enclosed media **67**

C Example GitHub Actions pipeline **69**

List of Figures

3.1 JSON representation of a GCP service-level objective 13
3.2 Artillery: YAML definition of a test scenario. 14
3.3 Artillery: YAML definition of a test configuration. 14
3.4 Artillery: YAML definition of SLOs. 15
3.5 Gatling: simple Java benchmark. 16
3.6 JMeter UI: definition of an HTTP request 17
3.7 k6: test of a single GET endpoint. 17
3.8 k6: test of a single POST endpoint with custom body and headers. 18
3.10 Locust: simple benchmark test implementation. 19
3.9 k6: latency threshold definition. 19

4.1 UML use case diagram 25

5.1 Go template: Input structure and execution logic (`main.go`). . . . 29
5.2 Go template: Template definition for a k6 benchmark. 29
5.3 Go template: Final generated benchmark. 30
5.4 OpenAPI vendor extensions at the top level of the document. . . . 32
5.5 OpenAPI vendor extensions at the endpoint level. 33
5.6 The Perfcheck package structure. 35
5.8 Installation using the Go command-line interface. 35
5.7 UML Activity Diagram: Perfcheck packages. 36

6.1 Initialization of the command-line interface. 41
6.2 Definition of a CLI command with a single flag. 41
6.3 Parsers: Generic representation of the API 42
6.4 Parsers: Generic representation of a single API endpoint (path). The `Metric` type is an alias for a Go string. 42
6.5 Parsers: Generic representation of a endpoint parameter generation logic. 43
6.6 Perfcheck: latency threshold template. 44
6.7 Perfcheck: generated latency thresholds. 45

6.8 Perfcheck: custom response size metric definition. 45
6.9 Perfcheck: parameter generation sub-template. 46
6.10 Perfcheck: the `generateFromPattern` function. 47
6.11 Perfcheck: the benchmark execution. 48
6.12 Perfcheck: Example usage of a statistical SLO. 49
6.13 Perfcheck: One-sided one sample t-test execution. 49
6.14 GoReleaser: Linux packages configuration 50
6.15 GoReleaser: Snapcraft store configuration 51
6.16 GoReleaser: Docker image configuration 51

7.1 Example: application entrypoint with global SLO specification. . . 54
7.2 Example: application entrypoint with global SLO specification. . . 55
7.3 Google Cloud: SLO definition for a Cloud Run application. 56
7.4 Example: application endpoint with a statistical latency requirement 56

List of Tables

2.1 Examples of service-level objectives 8

4.1 Functional requirements coverage. 24

5.1 OpenAPI parameter generation for Perfcheck. 34

Introduction

Performance is an often overlooked aspect of software products. Being difficult to define, measure, and guarantee, performance may not be a priority and efforts are instead placed on new features and pleasing aesthetics. Upper management may find it difficult to express exact performance requirements without technical knowledge or prior experience, which could lead to negligence by the development team. Many projects could benefit by being able to increase availability and reduce loading times or the number of outages. Poor performance discourages both new and existing consumers from using the product, which is undesirable when user engagement is a key indicator of company success.

In situations where high performance is essential to the product, one must be able to precisely specify the acceptance criteria of the product. One way to define performance requirements is through a Service Level Agreement (SLA) document. The components of the Service Level Agreement describe individual objectives (service-level objectives, SLO) and respective metrics that need to be monitored and kept above or below a certain threshold (service-level indicator, SLI). Not fulfilling the objectives results in penalties against the provider of the product. Having the ability to predict and monitor the affected metrics is beneficial for both the customer and the supplier of the product.

It is time consuming for developers to write benchmark tests that verify the fulfillment of service-level objectives. Although tools and frameworks are available to implement simple or advanced benchmarks, most require knowledge of a certain programming language and a mastery of complex configuration mechanisms. It is even more tricky to decide whether a service complies with the SLOs before being deployed to production. In this case, benchmark tests must be executed manually before the release in a staging environment, or included in the deployment pipeline and executed automatically. Automation is essential for a continuous deployment process. Therefore, one could also benefit from the tests being autogenerated, either from the source code,

documentation, or a description of an interface.

Backend services commonly offer a public description of their Application Programming Interface (API), which helps consumers of the service familiarize themselves with their functionality. The API description is used primarily to define the required inputs and expected outputs of individual endpoints. The formats of the API descriptions are heavily standardized and well known by developers. Therefore, it is often used for the generation of client-side code. However, it could also be used to specify non-functional requirements, either globally or for parts of the service. An external tool would then be able to extract the requirements, generate benchmark tests for the API, and run them in a given environment. As a result, human efforts would decrease and the reliability of the testing process will improve.

Chapter **1**

Goal

The goal of this thesis is to design, implement, and test a software tool which will reduce development efforts related to non-functional service testing. The new tool will provide a way to easily create and run benchmark tests for a given service. The use of the tool should be language-agnostic and will enable the description of the requirements for the service within the implementation. All benchmarks will be automatically generated, based on service level objectives (SLOs). It would only be necessary to include the service level objectives in the application programming interface (API) description.

The tool will be usable in continuous integration (CI) and continuous delivery (CD) workflows. A service that does not conform to the selected requirements will be marked invalid by the workflow. The results of the benchmark tests will be easily available to service developers and could be advantageous in improving the service. Once the workflow has been completed successfully, the service can be considered compliant with the SLO requirements and can proceed to the next release step.

The following are the steps necessary to complete the task:

- familiarization with current software testing techniques, described in Chapter 2,
- familiarization with the current possibilities of benchmark testing and description of specific shortcomings of existing benchmark tools, described in Chapter 3,
- analysis of requirements and use cases, described in Chapter 4
- outlining the architecture and design of the new tool, and deciding which technologies are suitable for development, described in Chapter 5,
- implementation of the tool, described in Chapter 6
- evaluation of the functionality and usability of the new tool, with the use of examples, described in Chapter 7.

Chapter 2

Software testing

Software testing is the process of evaluating the validity of a software product by identifying defects or errors that negatively affect the users of the product. The concept of software testing has evolved greatly since it was first introduced, but it is not an exact science [1]. Many different approaches are used in the software development industry, each covered by a wide selection of tools.

2.1 Functional testing

Functional testing is the act of checking whether a given service follows a predefined set of functional (business) requirements. It is used to verify that the application is working as intended [2].

It is rarely possible to completely verify the behavior of a service. Techniques such as program verification can be used, but their use is problematic for complex applications. Normally, the test case developer must cover as many possible execution paths as possible in order to consider the application as correct.

Functional testing is typically applied at multiple levels, with each level focusing on a different aspect of a standard software application. Possible examples are listed below [2]:

- unit testing,
- integration testing,
- system (end-to-end) testing,
- acceptance testing.

2.2 Non-functional testing

Non-functional testing is the act of measuring and evaluating how *well* a system performs the required tasks. Non-functional requirements typically define thresholds for various metrics that need to be followed. Ignoring non-functional testing might lead to the service being rendered unusable or impractical for the user, even if the system behavior is correct. Possible examples of non-functional requirements are a certain level of security, performance, usability, or accessibility [3].

2.3 SLA, SLO, SLI

Non-functional requirements are usually defined by the customer in a **service-level agreement** (SLA) document. The SLA is a documented agreement that binds the supplier and the customer of the product and contains the objectives that the service must achieve to comply with the agreement. If the objectives are not met, sanctions can be placed against the service provider. A common way to define objectives is to specify relevant metrics and appropriate thresholds that must be followed [3].

Service-level objectives (SLOs) are targets within the SLA, agreed between the customer, the supplier, and stakeholders, and are used to measure how the service performs. Used in combination with alerting mechanisms, SLOs help prevent breaches of the SLA document[3].

The status of an SLO is evaluated by **service-level indicators** (SLIs), which track a measurable quantity. SLIs are used to determine whether the objectives in the respective document are met. Common examples of SLIs are [3]:

- **Latency** is the time between the initiation of the request and the acquisition of the first part of the response (time to first byte, TTFB). It can be interpreted as the delay caused by the network connection plus the server computation [4]. Latency requirements can be based on several statistics, such as the mean, median or a certain percentile [5].
- **Error rate** is the ratio of the number of failed requests to the number of all requests. In the HTTP protocol, a failed request is one that did not receive a response or received a response with a 5xx status code. [3].
- **Availability** is the ability of the system to perform the required task at a given point in time. Availability windows are usually calendar-based or hour-baseds [3]. For example, a service may need to be available during regular work hour. The availability thresholds can be expressed as a percentage of requests that receive a correct response during the defined availability window. When taking into account only simple objectives, the error rate is inversely proportional to availability [3].

- **Response size** is the size in bytes of the response body. The requirements can be based on average size, maximum size, or a similar metric.

- **Throughput** is the number of requests per second that the system can handle before becoming saturated. When a system is saturated, performance will decrease for any additional users [3]. As such, it defines the maximum number of concurrent users. It is commonly used in conjunction with latency [6].

- **Mean time between failure (MTBF)** is the average time delay between two consecutive system outages.

- **Mean time to repair (MTTR)** is the average time delay between a system outage and its recovery.

- **Freshness** is the proportion of accessed data that was updated more recently than a certain threshold. It is often related to data caching [3].

- **Durability** is the probability that the data received from a service will be up-to-date for a specified period of time [6]

- **Correctness** is the ratio of correctly and incorrectly performed actions within the service. It is usually a property of the service, not of the infrastructure [6].

Some of the above-mentioned service-level indicators may overlap. It is important for the customer to specify what affects the business and what does not. It is rare that more than a few SLIs are used [3].

2.3.1 SLO examples

Examples of SLOs are shown in Table 2.1.

2.3.2 Error budget

It is highly desirable to monitor the operation of the services in production, to remain within the **SLO error budget** [3]. *The error budget for an SLO represents the total amount of time that a service can be noncompliant before it is in violation of its SLO* [5]. The SLO error budget enables tracking how many unwanted SLI measurements are allowed to occur until the end of the current compliance period, which might be helpful in managing release tasks and preventing sanctions against the supplier [3].

Indicator	Threshold	Statistic	Description
Latency	100 ms	mean	The average latency must be less than 100 milliseconds.
Latency	50 ms	99%	99% of requests must be faster than 50 milliseconds.
Error rate	0.01	-	Less than 1% of requests can fail.
Throughput	1 million, 1 s	95 %	95% of throughput requests must be handled under 1 second
Mean time to repair	60 min	mean	The average time to repair must be less than 60 minutes.
Availability	99%	-	The system must be available 99% of the time.

Table 2.1: Examples of service-level objectives

2.3.3 SLO evaluation

It is possible to define two distinct SLO evaluation methods that specify when an SLI metric is collected [7]:

- **Request-based evaluation** compares the number of requests that meet the evaluation criteria versus the number of requests that do not, in a given period.
- **Windows-based evaluation** compares the number of evaluation periods that meet a ”goodness” criterion with those that do not. For example, in an evaluation period of one week, one could check that 99 % of one-hour intervals were good enough to meet the SLO objectives.

Furthermore, it is necessary to specify the *compliance period* for the SLOs. Compliance periods are time intervals that specify when an SLO is evaluated [3, 7].

- **Rolling window** evaluation uses a sliding period with defined length and granularity. For example, a rolling window of 7 days has a daily evaluation. On a new day, the compliance and error budgets are recalculated for the length of the rolling window [7]. Rolling windows are more closely aligned with the user experience of the service [3].
- **Calendar window** evaluation measures compliance over a fixed period of time. The error budget is reset when a new period begins [7]. The calendar window approach is more closely aligned with business planning tasks [3].

This thesis focuses on the non-functional testing of services, using service-level objectives and the API description as the input, and verifying whether the service conforms to the specified objectives, before releasing to production.

2.4 Pre-release service testing

For most software, it is vital to run tests automatically before releasing them to production. Functional tests can be easily performed in a non-production environment, with no effect on the results. It is expected that the system's functional behavior does not depend on the resources available in its environment [1].

However, non-functional testing usually depends on the service's environment. Evaluation of non-functional requirements before releasing to production is not trivial and requires having computational resources similar to production resources available in the testing phase. This can be achieved by one of the following techniques [3]:

- staging (pre-production) environment,
- blue-green deployments,
- canary deployments.

2.4.1 Staging environment

A staging, or pre-production, environment is typically a replica of the production environment from a given point in time. It should include all services, data, and resources found in a production environment. To accurately evaluate non-functional requirements, the performance test must be able to generate a load similar to real-world traffic [3]. In practice, ephemeral environments are often used to reduce the costs associated with the maintenance of a complete environment [8].

An advantage of using a staging environment is that the execution of any kind of test does not affect the production application. A test may insert unwanted entries into the database that would then need to be deleted. In a staging environment, this is usually not a problem. The disadvantage of using a staging environment is the need to replicate all of the services running in production. These may include several business-related services, databases, message queues, or caches. Additionally, APIs of external dependencies may need to be mocked, in order for the staging environment to function properly.

2.4.2 Blue-green deployments

Blue-green deployment is a method of releasing changes safely to production. Two versions of a production service are deployed at the same time. With

blue-green deployments, only one of these servers is handling traffic. The new version can be monitored and tested before receiving production traffic [3].

2.4.3 Canary deployments

Canary deployments are partial, time-limited releases directly to the production environment. Once evaluated, they are completely deployed or reversed back to the original (*control*) deployment [3].

In canary deployments, only a small subset of users are affected [3]. The requirement for using blue-green or canary deployments is having multiple instances of a single service, which allows for routing different users to different versions of the program.

2.4.4 Service evaluation in a serverless environment

In certain serverless environments (Cloud Run, AWS Fargate), it is possible to deploy a new instance of an application independently of the live version, without needing to manually allocate resources. Therefore, it is feasible to perform service benchmark testing on a computationally equivalent replica with little effort. This approach is similar to a blue-green deployment with no affected end users. After the successful evaluation of the new revision, production traffic can be re-routed from the previous version [9, 10].

Chapter 3

State-of-the-art

This chapter outlines the current methods of service monitoring and proactive service evaluation with respect to service-level objectives.

3.1 Service monitoring

Most popular cloud providers (AWS, GCP, Azure, Vercel) have the option to monitor managed cloud services, such as containers, Kubernetes clusters, serverless functions, databases, API gateways, or static file servers. They provide metrics, alerts, and visualization of the performance of an application. Monitoring self-hosted systems is possible with the help of dedicated software, for example, Kubernetes Dashboard, Prometheus, or NGINX Amplify. The functionality of these tools is analyzed in the following sections, with regard to the ability to monitor running services, displaying analytical dashboards, and specifying SLOs.

3.1.1 Kubernetes Dashboard

Dashboard is a web-based Kubernetes user interface. You can use Dashboard to deploy containerized applications to a Kubernetes cluster, troubleshoot your containerized application, and manage the cluster resources. You can use Dashboard to get an overview of applications running on your cluster, as well as for creating or modifying individual Kubernetes resources (such as Deployments, Jobs, DaemonSets, etc.). For example, you can scale a Deployment, initiate a rolling update, restart a pod or deploy new applications using a deploy wizard. [11]

Service monitoring is possible using Kubernetes Dashboard, but it does not include alerts. It is also not possible to define and monitor custom SLOs for deployments.

3.1.2 Prometheus

Prometheus is an open-source monitoring system designed to collect, store, and analyze metrics data. It also includes an alerting mechanism. A custom query language, PromQL, is used to extract data from Prometheus. It is also possible to define custom SLOs in Prometheus using an external generator. [12].

3.1.3 NGINX Amplify

NGINX Amplify is a SaaS tool for monitoring NGINX proxy servers. It provides real-time monitoring and diagnostics of an application's performance. Alerts can be configured in NGINX Amplify. It does not provide an option to define SLOs. [13]

3.1.4 Google Cloud Platform SLOs

Google Cloud Platform (GCP) is a collection of cloud services offered by Google Cloud that include computing, storage, analytics, monitoring, and machine learning [14]. The most popular computing services include: Compute Engine, Kubernetes Engine, and Cloud Run.

Computing services with an exposed API running on GCP can be monitored using the GCP Monitoring service. GCP Monitoring collects telemetry from the services, provides analytics, and has the option to define custom SLOs [15].

GCP automatically assesses compliance with defined SLOs and displays the remaining error budget based on the SLO specifications, such as the SLI metric, the evaluation method and the compliance period [7]. SLOs defined in Cloud Monitoring are available via the Google Cloud API or various language-specific SDKs. This makes it possible to collect the SLOs and work with them in an external service. Within GCP, it is not possible to verify SLO compliance before a release.

Figure 7.3 shows an example of an SLO defined in GCP, represented in JSON. The SLO defines a request-based requirement for 1-second latency in 99 % of requests. The compliance period is one calendar day.

```
{
  "name": "projects/[PROJECT_ID]/services/
  [SERVICE_ID]/serviceLevelObjectives/
  [SLO_ID]",
  "displayName": "99% - Latency - Calendar day",
  "goal": 0.99,
  "calendarPeriod": "DAY",
  "serviceLevelIndicator": {
    "basicSli": {
      "latency": {
        "threshold": "1s"
      }
    }
  }
}
```

Figure 3.1: JSON representation of a GCP service-level objective

3.1.5 Other cloud providers

In AWS and Azure, it is not possible to set up service-level objectives and monitoring in a similar fashion to Google Cloud Platform SLOs.

3.2 Service evaluation

Evaluating service performance before production release is possible by using benchmarking tools. The tools are typically configured using a script (*Load-Test-as-Code*) or a custom configuration format. Afterwards, the benchmark is performed by executing the required HTTP communication. Results of the benchmark are typically available to the developer in a structured format. The most popular open-source options, based on the size of the user base and the activity of project development, are [16]:

- Artillery
- Gatling
- Apache JMeter
- Grafana k6
- Locust

3.2.1 Artillery

Artillery is a load test platform, designed to integrate with other monitoring tools and CI/CD tools [17]. Test scenarios and configurations are defined using a YAML format. An example of a simple test scenario is shown in Figure 3.2. In this scenario, items are first listed using a GET request. Then, the details of the first item are viewed using another GET request.

```yaml
scenarios:
  - name: "List items and view details of first"
    flow:
      - get:
          url: "/items"
          capture:
            - json: "$.results[0].id"
              as: "id"
      - get:
          url: "/items/{{ id }}"
```

Figure 3.2: Artillery: YAML definition of a test scenario.

To simulate real-world traffic, a test scenario may be executed with multiple concurrent virtual users (VUs). Each test can consist of multiple stages, with a specific number of VUs, or a ramping strategy for users [17]. Figure 3.3 shows the configuration of two consecutive testing stages.

```yaml
config:
  target: https://example.com
  phases:
    - duration: 10
      arrivalRate: 1
    - duration: 10
      arrivalRate: 10
```

Figure 3.3: Artillery: YAML definition of a test configuration.

Artillery will automatically collect metrics from requests defined in the scenarios and display them in the standard output of the test execution [17]. To define and verify SLOs, artillery uses a threshold concept. Figure 3.4 shows the definition of SLOs for latency in milliseconds, using the built-in

http.response_time metric. A custom percentile is defined using the p99 and p95 specifiers. If the service successfully complies with the SLOs, the test exits successfully, otherwise it displays the problem and exists with an error code.

```yaml
config:
  ensure:
    thresholds:
      - "http.response_time.p99": 250
      - "http.response_time.p95": 100
```

Figure 3.4: Artillery: YAML definition of SLOs.

3.2.2 Gatling

Gatling is a load-testing tool, designed to verify the objectives of HTTP interfaces. Gatling is executable in a Java environment. The test scenario may be defined in a Java, Kotlin, or Scala program [18]. Figure 3.5 shows a simple Java benchmark with the definition of the target endpoint, the number of users, and the frequency of calling the service. Gatling also provides a way to define the testing scenario using a visual user interface. Gatling does not provide a way to specify SLOs. Instead, a report is generated from each run of the benchmark.

```java
import io.gatling.javaapi.core.*;
import io.gatling.javaapi.http.*;

import static io.gatling.javaapi.core.CoreDsl.*;
import static io.gatling.javaapi.http.HttpDsl.*;

public class BasicSimulation extends Simulation {

  HttpProtocolBuilder httpProtocol = http
    .baseUrl("http://example/com");

  ScenarioBuilder scn = scenario("BasicSimulation")
    .exec(http("request")
      .get("/"))
    .pause(1);

  {
    setUp(
      scn.injectOpen(atOnceUsers(10))
    ).protocols(httpProtocol);
  }
}
```

Figure 3.5: Gatling: simple Java benchmark.

3.2.3 JMeter

Apache JMeter is an open-source Java-based benchmarking software. Compared to other popular benchmarking tools, Jmeter does not implement the *Load-Test-As-Code* approach. Instead, JMeter provides a window-based user interface. It enables the definition of test scenarios and displays metrics in a user-friendly way [19]. Figure 3.6 displays the JMeter user interface.

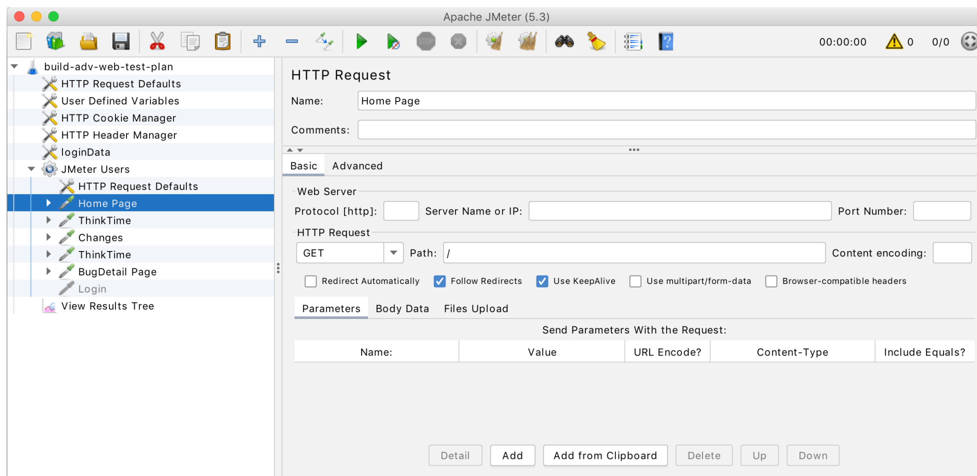

Figure 3.6: JMeter UI: definition of an HTTP request

Source: https://github.com/apache/jmeter

3.2.4 Grafana k6

Grafana k6 is an open-source benchmarking tool that makes it easy to execute load-testing scripts in a NodeJS environment. The test is expressed using a JavaScript program that uses multiple k6 libraries [20]. The developer only defines the details of the HTTP requests. At execution, k6 measures key performance metrics. An example test of a single endpoint is shown in figure 3.7.

```
import http from 'k6/http';

export default function () {
  http.get('https://example.com');
}
```

Figure 3.7: k6: test of a single GET endpoint.

Figure 3.8 shows how to execute more complex requests in k6, such as a POST request with headers and a body, where additional information must be included.

```
import http from 'k6/http';

export default function () {
  const body = {
    example: 'string'
  };

  const headers = {
    'Content-Type': 'application/json'
  }

  http.post('https://example.com', body, { headers });
}
```

Figure 3.8: k6: test of a single POST endpoint with custom body and headers.

By default, k6 collects the following metrics [20]:

- `http_reqs`: number of total HTTP requests executed,
- `http_req_blocked`: time spent waiting for a TCP connection slot,
- `http_req_connecting`: time spent establishing a TCP connection,
- `http_req_tls_handshaking`: time spent exchanging TLS secrets
- `http_req_sending`: time spent sending data,
- `http_req_waiting`: time to first byte (TTFB),
- `http_req_receiving`: time spent receiving data,
- `http_req_duration`: total time of a request,
- `http_req_failed`: ratio of failed requests.

Using metrics `http_req_duration` and `http_req_failed`, it is possible to check both the latency and error rate of a service. To implement pass/fail criteria for a k6 benchmark, thresholds for the test metric must be defined. Thresholds are a method to encode SLOs in k6 [20]. An example of a 200 *ms* threshold for the average latency of a request is shown in Figure 3.9.

```python
from locust import HttpUser, task

class AppUser(HttpUser):
    @task
    def hello_world(self):
        self.client.get("/foo")
```

Figure 3.10: Locust: simple benchmark test implementation.

```javascript
import http from 'k6/http';

export const options = {
  thresholds: {
    http_req_duration: ['avg < 200']
  },
};

export default function () {
  http.get('https://example.com');
}
```

Figure 3.9: k6: latency threshold definition.

To simulate real-world traffic, k6 has the option of specifying the number of concurrent requests to the service, using virtual users. Additionally, multiple stages can be executed with varying durations and numbers of virtual users to simulate increasing and decreasing traffic.

3.2.5 Locust

Locust is an open-source load-testing framework, provided as a Python package. The test configuration and scenarios are expressed as a Python script [21]. An example of a simple testing scenario is shown in Figure 3.10. In this configuration, Locust will add a new user of the service every second, with each of them regularly accessing the service. Compared to other popular benchmarking tools, Locust does not provide an option to specify SLOs.

3.2.6 Benchmark tools summary

All of the above-mentioned tools provide a similar set of capabilities, such as:

- Test scenarios - the test may consist of multiple consecutive requests to the service.
- Test stages - the scenario may be executed in stages with varying numbers of virtual users or a custom ramping strategy.
- Test thresholds - the test will succeed or fail based on whether the tracked metrics comply with the specified thresholds. In most cases, thresholds can directly express service-level objectives for the service.

Chapter 4

Analysis

In this chapter, an analysis of a new software tool for automated benchmark testing is performed. The tool will parse the SLO definitions embedded in the API description of a service or in an SLO monitoring platform. It will automatically generate and run a benchmark test for the service. In practice, this should free the developer from manual benchmark preparation. The target audience, functional and non-functional requirements, and use cases of the tool are described. The name *Perfcheck* (=performance check) was chosen for this tool.

4.1 Target audience

Software testing is mainly performed by the developers of the software in question or a dedicated tester. This tool aims to help developers verify their own software services. Full access rights and technical knowledge of the tested service may be required to use the tool. Deployment architecture and other infrastructure specifics are required to include the tool in a continuous delivery pipeline.

4.2 Target SLIs

This tool will be able to verify **latency**, **error rate**, and **response size** of a service or its individual endpoints. To include and evaluate other types of SLOs, the developer will be able to customize the benchmark configuration.

In theory, latency, error rate, response size, and throughput are the only SLIs independent of the current internal state of the system and therefore can be tested in a clean environment. Testing throughput is similar to specifying the number of virtual users and monitoring performance. Therefore, throughput will not be included in the SLI options for the new tool. Other SLIs, such as availability, freshness, MTBF, MTTR, or durability, depend on

the current point in time and the internal state of the system [3]. These are not easily tested with automatic benchmark generation. Furthermore, metrics such as quality or correctness may require additional business logic knowledge to provide a proper evaluation [3].

4.3 Requirements

This section defines the requirements for Perfcheck in order to be a useful and usable tool for software developers.

4.3.1 Functional requirements

- **FR01: Parse API description from an external source**
  The service developer must be able to specify the source of the API description. The API description is used to collect metadata about the service and metadata about the individual endpoints. The supported formats must be OpenAPIv2 and OpenAPIv3. The supported protocols must be at least HTTP as HTTPS.

- **FR02: Parse the SLO description from an external source**
  The service developer must be able to specify the source of the service-level objectives. The SLOs are necessary to generate a benchmark test for a service. The supported sources must be OpenAPIv2, OpenAPIv3, and Google Cloud SLOs. The format of the SLO specification for OpenAPI will be defined by Perfcheck.

- **FR03: Benchmark generation**
  The tool should generate a benchmark as a configuration of an existing benchmarking framework, for example, k6.

- **FR04: Benchmark execution**
  The benchmark must be executable in any supported environment with network access to the API of the service and enough computational resources to perform the test.

- **FR05: Benchmark results availability**
  The benchmark results must be available to the developer via a JSON file. The results might be used for further analysis of the performance.

- **FR06: Benchmark customization**
  The generated benchmark test must be customizable by the developer prior to execution, either by modifying the template or modifying the final benchmark test.

- **FR07: Statistical evaluation**
  The tool must be able to statistically assess compliance with SLOs based

on the benchmark data. Mainly, it must decide whether compliance with the SLOs based on limited testing data is statistically significant. The affected metrics are the average latency, the error rate, and the average response size. The results of the statistical evaluation must be taken into account when including the service in a CI/CD pipeline.

4.3.2 Non-functional requirements

- **NFR01: Installation availability**
  The tool must be available for installation at any time for Linux, Windows, and MacOS environments. Installation using a command-line interface is sufficient. Both prebuilt packages and an option to build from source should be available for all supported platforms. A docker image containing the pre-installed tool should be publicly available.

- **NFR02: Language interoperability**
  The functionality of the tool must not depend on the implementation specifics of the target service. It must be language and framework agnostic.

- **NFR03: Configurability**
  Tool configuration must be performed using command-line arguments or environment variables.

- **NFR04: Input extensibility**
  The tool should follow good software development patterns to be extensible in terms of input sources. Additional formats of API descriptions and SLO definitions may be added.

- **NFR05: Output extensibility**
  The tool should follow good software development patterns, to be extensible in terms of defining new output targets for the results of the benchmark.

- **NFR06: SLO extensibility**
  Developers should be able to extend the tool to support other types of SLOs, such as response size, availability, throughput, mean time between failure (MTBF), or mean time to repair (MTTR).

- **NFR08: Source availability**
  The tool should be open source. The code must be available on GitHub, with proper issue tracking, pull request strategy, and member access. A proper license must be attached to the source code.

Use case / requirement	FR1	FR2	FR3	FR4	FR5	FR6	FR7
UC01			√	√			
UC02			√	√			
UC03	√	√	√	√	√		√
UC04	√	√	√	√	√		√
UC05						√	
UC06						√	

Table 4.1: Functional requirements coverage.

4.4 Use cases

- **UC01: Install Perfcheck in a local environment**
  A developer installs Perfcheck in his personal environment.
- **UC02: Include Perfcheck in a CI/CD pipeline**
  A developer or DevOps engineer installs Perfcheck in a continuous integration/deployment pipeline.
- **UC03: Execute a benchmark using the OpenAPI definition**
  The developer or an automatic job executes the benchmark by providing the URL of the OpenAPI definition with custom annotations defined by Perfcheck.
- **UC04: Execute a benchmark using Google Cloud**
  The developer or an automatic job executes the benchmark by providing the information required to access the Google Cloud SLO definition.
- **UC05: Customize the benchmark template prior to generation**
  The developer edits the benchmark template to obtain a customized benchmark test.
- **UC06: Customize the benchmark prior to execution**
  The developer or an automatic job edits the resulting benchmark, in order to include service-specific requirements.

Use cases are visualized in the UML use case diagram in Figure 4.1. Table 4.1 shows the coverage of functional requirements by use cases. All requirements must be covered by at least one use case.

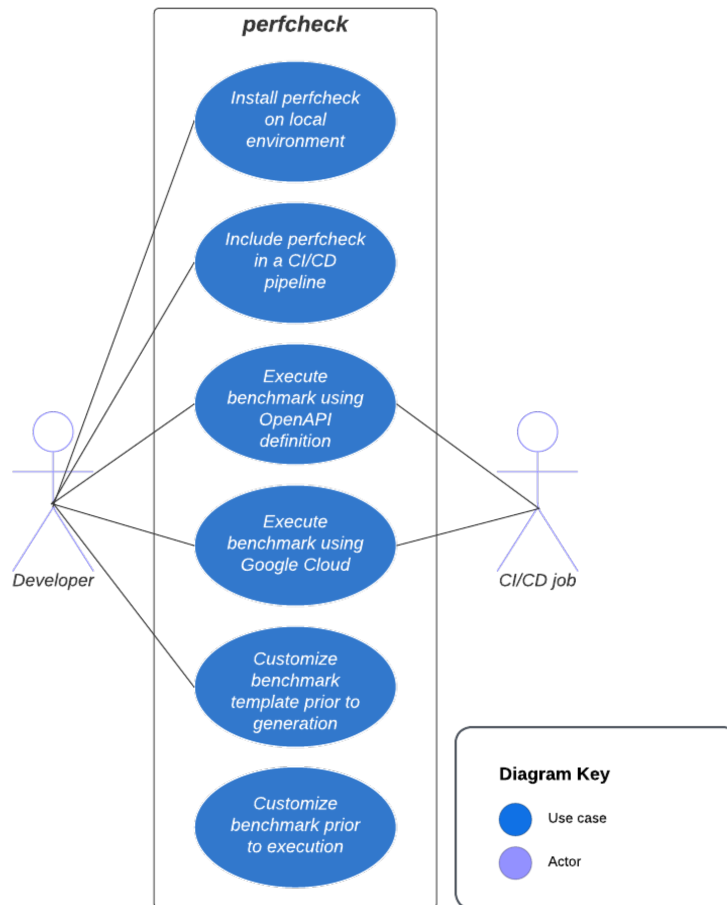

Figure 4.1: UML use case diagram

Chapter 5

Design and Architecture

This chapter introduces the methods and tools used to implement `Perfcheck`, proposes the overall architecture of the new tool, and describes the design choices behind the implementation of the tool. The design is the blueprint for the implementation of the product.

5.1 Benchmarking tool

The tool used to execute the generated benchmarks will be k6 [20]. This tool was chosen because of the use of the JavaScript language for the benchmark definition. According to the FR06 (benchmark customization) requirement, the benchmark must be customizable by the developer. Using a well-known language for the benchmark configuration enables more developers to access this functionality. JavaScript currently stands as the most commonly used programming language, according to the 2022 Stack Overflow Developer Survey [22]. As a result, benchmark customizability should be accessible to most developers. Furthermore, because of the open source nature of this project, more people may contribute to the future development of the project.

5.2 Language

The programming language of choice for the project is Golang (Go). Go is a low-level, statically typed, compiled imperative language created by Google [23]. Go is typically used to develop web applications, cloud services, and networking services. Examples of applications written in Go are Docker, Kubernetes, Prometheus, NATS, Terraform, or IPFS.

Go was chosen for its ease of use, wide library ecosystem, performance, and most importantly, the Go `text/template` package. According to the FR03 (benchmark generation) requirement, the tool must generate a test configuration of an external benchmarking tool. Therefore, it might be appropriate

to use a generic template for the test which would be filled in based on the input data. The go `text/template` is one of the most popular choices for text template processing [23].

In addition, the Golang ecosystem is very well suited to cover the NFR01 (installation availability) requirement. The release management technique is described in Section 5.7.1.

5.2.1 Go templates

The built-in `text/template` package implements a templating engine to generate textual output. As such, it can be used to automatically generate configuration files for a benchmarking tool, given a uniform SLO definition structure. Therefore, it could even be used to generate configurations for multiple benchmarking systems with no additional logic.

Go templates generate text based on input parameters. Executing a template means applying it to a Golang data structure (map, array, struct), which is available in the template definition using references. Any text enclosed in double curly braces is a dynamic content of the template. Inside the braces, the input data structure can be accessed. The full input is available using the dot (.) symbol. Nested values are accessible via the dot notation on the object. An example of a template execution is shown with a Go structure as input. Figure 5.1 displays the input structure for the template, Figure 5.2 displays the definition of a k6 benchmark template, and Figure 5.3 displays the output benchmark.

To cover FR06, the template file will be available for modification by the developer, before template execution. To cover FR07, the output file can be modified before running the benchmark test itself.

5.3 Input and Output

This tool is going to be used primarily by software developers; therefore, a command-line interface will be sufficient. A command-line interface will be uniform in a local environment and in a CI/CD pipeline, reducing the time spent learning the tool. Parameters (flags) can be passed either as command-line arguments, as environment variables, or from a configuration file. While testing in a local environment, using command-line arguments or configuration files is more efficient. In an CI/CD pipeline, environment variables are recommended to protect sensitive information. Popular CI/CD services have the option of variable masking [24].

An external Go module, such as `urfave/cli` [25], may be used to parse subcommands, arguments, flags, and generate documentation for the command-line interface.

The tool will have the following subcommands:

```go
import (
    "os"
    "text/template"
)

type LatencySLO struct {
    Latency int
    Target  float64
    Url     string
}

// template input
slo := LatencySLO{500, 99, "https://example.com"}

template, err := template.ParseFiles("benchmark.tmpl")
if err != nil {
    panic(err)
}

// execute and print result
template.Execute(os.Stdout, slo)
```

Figure 5.1: Go template: Input structure and execution logic (`main.go`).

```javascript
import http from "k6/http";

export const options = {
  thresholds: {
    http_req_duration: ["p({{ .Target }}) < {{ .Latency }}"]
  },
};

export default function () {
  http.get("{{ .Url }}");
}
```

Figure 5.2: Go template: Template definition for a k6 benchmark.

```
import http from "k6/http";

export const options = {
  thresholds: {
    http_req_duration: ["p(99) < 500"]
  },
};

export default function () {
  http.get("https://example.com");
}
```

Figure 5.3: Go template: Final generated benchmark.

- `generate`: Generate a benchmark from specified sources,
- `test`: Generate a benchmark from specified sources and execute it,
- `get-template`: Print the default template in a file to allow modifications.

All sub-commands have the `--config` flag, which specifies the path to a configuration file. The priorities for the configuration methods are:

1. Command-line arguments
2. Configuration file (if specified)
3. Environment variables

With this approach, variables from the configuration file or from the environment can be easily overwritten by the command-line arguments, if needed.

5.3.1 generate

The `generate` subcommand creates a benchmark configuration from specified sources. All flags, with their full name, shorthand, and environment variable name, are listed below:

- `--source, -s, [$SOURCE]`: Source of the API and SLO definitions. The allowed values are `openapi` and `gcloud`.
- `--docsUrl, -d, [$DOCS_URL]`: URL of the OpenAPI documentation, used if the source is set to `openapi`.

- `--gcloudProjectId, [$GCLOUD_PROJECT_ID]`: Google Cloud project ID, used if the source is set to `gcloud`.
- `--gcloudServiceId, [$GCLOUD_SERVICE_ID]`: Google Cloud monitoring service ID, used if the source is set to `gcloud`.
- `--gcloudServiceUrl, [$GCLOUD_SERVICE_URL]` URL of the Google Cloud target service, used if the source is set to `gcloud`.
- `--template, -t, [$TEMPLATE]`: Template file for the benchmark. If not specified, the default template is used.
- `--outFile, -o, [$OUT_FILE]`: Output file for the benchmark, the default value is `benchmarks/benchmark.js`.

5.3.2 `test`

The `test` subcommand uses the same syntax as the `generate` subcommand, with two additional flags that control the execution of the test.

- `--k6DataFile, [$K6_DATA_FILE]`: Output file for the k6 benchmark JSONL data (measurements). The default value is `data/k6.jsonl`.
- `--alpha, [$ALPHA]`: The significance level for statistical tests. The default value is 0.05.

5.3.3 `get-template`

The `get-template` sub-command prints the default template to a file specified by a flag or an environment variable:

- `--outputFile, -o, [$OUTPUT_FILE]`: Output file for the benchmark. The default value is `benchmark.tmpl`.

5.4 SLO parsing

Parsing the SLO definition from an external source is one of the core functionalities of the application. Two distinct sources must be supported according to FR02: OpenAPI documentation and Google Cloud SLOs.

5.4.1 OpenAPI SLOs

Both the OpenAPIv2 and OpenAPIv3 standards allow developers to include additional information in the OpenAPI document. Such properties are named *vendor extensions* [26]. This is achieved by adding a custom property prefixed with `-x` to the JSON or YAML representation of the document. Vendor extensions can be included at multiple different levels [26]:

```
    {
        "swagger": "2.0",
        "info": {
            "title": "Example API"
        },
        "paths": {},
        "x-perfcheck": {
            "stages": [
                { "duration": "1s", "target": 5 },
                { "duration": "3s", "target": 10 }
        ]
    }
}
```

Figure 5.4: OpenAPI vendor extensions at the top level of the document.

- at the top level of the document,
- in the info section,
- at operation (endpoint) level.
- in operation parameters,
- in operation responses,
- in security schemes.

For Perfcheck, it is relevant to define custom information at the top level of the document, at the endpoint level, and in the endpoint parameters. Top-level information will be used to define the number of virtual users for the test or the sequence of test stages. In addition, top-level vendor extensions are used to define SLO requirements for the entire application. This information is equivalent to the `vus` and `stages` options in k6. Figure 5.4 shows an example of defining multiple testing stages in an OpenAPI document, using the `x-perfcheck` key.

Adding vendor extensions at the endpoint level is achieved by defining an `x-perfcheck` property within the definition of an API operation. Figure 5.5 displays an example of defining SLOs for a single GET endpoint. The semantics of the SLO definition are equivalent to the threshold definitions of k6 [20], with additional options to support the statistical evaluation of the results of the k6 test.

In most cases, calling an endpoint requires the inclusion of additional parameters. The types of parameters for a typical HTTP request are:

```
{
    "swagger": "2.0",
    "info": {
        "title": "Example API",
    },
    "paths": {
        "/": {
            "get": {
                "summary": "Hello World",
                "x-perfcheck": {
                    "latency": [ "avg < 50" ]
                }
            }
        }
    }
}
```

Figure 5.5: OpenAPI vendor extensions at the endpoint level.

- path parameters,
- query parameters,
- cookie parameters,
- HTTP headers,
- HTTP body.

An example of an endpoint that does not require any parameters is the index file of a website. In cases where parameters need to be included, Perfcheck needs to know how to generate them, based on the endpoint description. By default, OpenAPI allows the developer to specify one example value for a parameter, which might be sufficient in some cases. If more complex logic is required, the developer will be able to define how the value should be generated. This is achieved by adding the `x-perfcheck` vendor extension to the "parameters" section of an OpenAPI operation. This property will contain a string that specifies the generation method. Possible methods for parameter generation are listed in Table 5.1. This operation takes precedence over the default example.

Method	Description	Example usage	Example output
`pick`	Pick a random example	`pick("foo", "bar")`	`"foo"`
`bool`	Random boolean	`bool()`	`true`
`string`	Random string with a specified length	`string(8)`	`"as64gh9+"`
`uuid`	Random UUID with a specified version	`uuid(4)`	`"f886...2ce"`
`range`	Random integer from a range with a specified lower and upper limit	`range(0, 50)`	`27`

Table 5.1: OpenAPI parameter generation for Perfcheck.

5.4.2 Google Cloud SLOs

Parsing SLO definitions from Google Cloud Monitoring is straightforward. However, Google Cloud supports only the definition of SLOs for the entire application, not for a specific endpoint. Therefore, when using Google Cloud as the source, Perfcheck is limited to testing only the root endpoint of an application. In some cases, such as single-page applications (SPAs) or simple APIs, this is sufficient.

5.5 SLO evaluation

The tool will use the built-in SLO evaluation methods of k6. Objectives are always evaluated using a certain statistic obtained from the benchmarking data. By default, k6 includes the following statistics [20]:

- rate: the ratio between the number of successful requests and the number of failed requests,
- mean: the average value of a metric,
- median: the median value of a metric,
- n-th percentile: percentage of values that must fall below or above the threshold,
- min, max: minimum or maximum value of the metric.

Furthermore, Perfcheck needs to employ statistical methods to evaluate SLO compliance more precisely. The benchmark test is a sample of limited size; therefore, the prediction can be improved by implementing statistical hypothesis testing. In theory, this approach should eliminate statistical errors and increase the reliability of the test.

The results of the SLO evaluation must be available to developers after the test execution. With this information, developers can focus on fixing the application shortcomings. Results in text will be printed on the standard output of the console. Most CI/CD systems have the ability to display the console output of individual jobs.

Individual events from k6 will be stored directly in a file, which can then be used to further process the results or send them to an external service, such as a data warehouse.

5.6 Architecture

To be testable and extendable, the functionality of the tool is split into multiple components. This section describes the architecture of the internal components of Perfcheck. In Go, separation of concerns is achieved through the use of nested packages [23]. Figure 5.6 displays the package structure of Perfcheck.

Figure 5.6: The Perfcheck package structure.

The responsibility of individual packages is displayed in the UML activity diagram 5.7.

5.7 Release management

The `Perfcheck` package needs to be available for installation by developers either by using the Go environment or by downloading a pre-compiled binary if the Go environment is unavailable. By making the project publicly available on GitHub, the project is automatically published for installation using the Go command-line interface. Other developers can install the latest version of the tool using the shell command shown in figure 5.8. Go automatically downloads the source and builds the correct binary for the operating system.

```
> go install github.com/zacikpet/perfcheck:latest
```

Figure 5.8: Installation using the Go command-line interface.

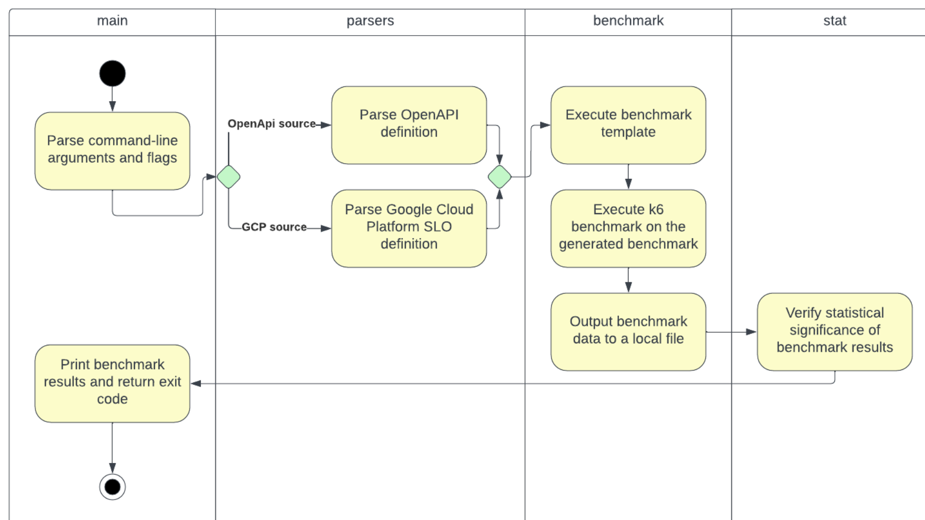

Figure 5.7: UML Activity Diagram: Perfcheck packages.

Pushing new changes to the master branch on GitHub updates the latest version of the package. Developers can choose a specific version by appending a tag name of a revision instead of the "latest" keyword.

If the Go command-line interface is not available, developers must be able to install the tool using other package management systems. Releasing Go projects for multiple platforms is easily achievable using the GoReleaser tool [27].

5.7.1 GoReleaser

GoRelaser is a tool that is used to cross-compile Golang packages, deploy them to multiple platforms, and create Docker images with a pre-installed binary [27]. With GoRelaser, Perfcheck will be released on the following platforms:

- Linux x86 64bit
- Linux i386
- Linux ARM 64bit
- Darwin x86 64bit
- Darwin ARM 64bit
- Windows x86 64bit

- Windows i386
- Windows ARM 64bit

Afterwards, installation is possible by downloading the correct binary from the GitHub release page or using the default OS package manager, such as brew for MacOS or apt for Ubuntu [27].

Chapter 6

Implementation

In this section, the process of implementing, documenting, and releasing Perfcheck is described. Only critical sections of the code are presented; additional context may be required to execute the code. The eclipse symbol (...) represents the left-out lines.

6.1 Environment

The first step of creating a Go application is to start with a clean GitHub project. The Go release system only works within repositories that are publicly available on the Internet. As such, the GitHub repository must be marked as public. The URL of the repository must be provided when creating a new Go application. The Go module is initialized using the following command:

```
$ go mod init github.com/zacikpet/perfcheck
```

Afterwards, any developer will be able to download and build the module using the Go command-line interface. The command displayed above also creates a *go.mod* file that maintains the project metadata and a list of direct and indirect dependencies. The installation and upgrading of dependencies is achieved with the following command, with the *-u* flag signaling to also patch indirect dependencies of the package.

```
$ go get -u example.com/package@version
```

Below is a list of direct dependencies of Perfcheck:

- `cloud.google.com/go/monitoring@1.13.0`: a part of the Google Cloud SDK for service monitoring
- `github.com/aclements/go-moremath@0.0.0`: a package that provides advanced math functions

- `github.com/pb33f/libopenapi@0.5.1`: support for OpenAPI documentation writing and parsing

- `github.com/urfave/cli/v2@2.25.0`: a command-line arguments parser

- `google.golang.org/api@0.114.0`: a set of packages that provide low-level access to Google APIs.

Installing dependencies also creates the *go.sum* file, which maintains the checksums of all dependencies and allows for exact installations on any system. Running the `go get` command without any arguments automatically downloads the correct versions of all dependencies listed in the *go.sum* file.

6.2 License

Regarding NFR08 (source availability), the new tool should be open-source. As such, it must include a permissive license. The dependencies of the project are provided under either the MIT License or the BSD 3-Clause Revised License. This project does not redistribute the dependencies; therefore, the license choice is not limited by their licenses. The MIT license is selected for this project and included in the root directory.

6.3 Command-line interface

The command line interface creation capability for an application is provided by the `github.com /urfave/cli/v2` package. All of the required logic is located in the *main.go* file, the application's entrypoint.

A CLI application is initialized and executed by the code in Figure 6.1, with the individual commands listed in the `Commands` array. Figure 6.2 defines the `get-template` command with the required list of flags and the action function. Flags and their respective shorthand names and environment variable names are also defined in this section.

Using this approach, `github.com/urfave/cli/v2` also generates a documentation page for each command and for the entire application. The documentation page is displayed to developers by appending a `--help` flag to any command.

```go
func main() {

    app := &cli.App{
        Name:  "perfcheck",
            Usage: "Automatic benchmarks of APIs",
            Commands: []*cli.Command{}
    }

    app.Run(os.Args)
}
```

Figure 6.1: Initialization of the command-line interface.

```go
{
    Name:  "get-template",
    Usage: "Output default template to file",
    Flags: []cli.Flag{
        &cli.StringFlag{
            Name:    "outFile",
            Aliases: []string{"o"},
            Usage:   "Name of output file",
            Value:   "benchmark.js.tmpl",
            EnvVars: []string{"OUT_FILE"},
        },
    },
    Action: func(ctx *cli.Context) error {
        return benchmark.GetTemplate(ctx.String("outFile"))
    },
},
```

Figure 6.2: Definition of a CLI command with a single flag.

6.4 SLO parsing

This section describes the implementation of the `perfcheck/parsers` package. The main functionality of this package is to collect SLO information from various sources and provide a generic interface. The dependents of this package do not need to differentiate between the sources of the SLOs. The

generic representation of the API is shown in figures 6.3, 6.4, and 6.5. The structures are initialized by the individual SLO parsing mechanisms.

```
type Api struct {
   BaseUrl string
   Paths   []Path
   Config  Config
}
```

Figure 6.3: Parsers: Generic representation of the API

```
type Path struct {
        Method   string
        Pathname string
        Detail   PathDetail
}

type PathDetail struct {
        Latency      []Metric
        ErrorRate    []Metric
        ResponseSize []Metric
        Params       Params
}
```

Figure 6.4: Parsers: Generic representation of a single API endpoint (path). The `Metric` type is an alias for a Go string.

```
type Params struct {
        Path  map[string]ParamDescription
        Query map[string]ParamDescription
}

type ParamDescription struct {
        Example  any
        Pattern  *string
}
```

Figure 6.5: Parsers: Generic representation of a endpoint parameter generation logic.

6.4.1 OpenAPI SLOs

The OpenAPI documentation is parsed into a Go structure, defined in Figure 6.3, using the `libopenapi` library [28]. Using this library, Perfcheck extracts the information necessary to generate the benchmark. This includes the protocol scheme, the base URL, the base SLO, and a list of operations with their respective SLOs and parameter generation logic.

6.4.2 Google Cloud SLOs

Service-level objectives defined within the Google Cloud are collected using the `cloud.google.com/go/monitoring` library [29]. The `monitoring` library is part of the Google Cloud SDK, which enables developers to work with, create, and edit Google Cloud resources. Proper authentication is required for Perfcheck to collect the required information. The Google Cloud SDK can read the credentials generated by the following shell command.

```
> gcloud auth
```

This command stores Google credentials in an available location on the system. Afterwards, the Google Cloud client libraries running on the given machine are authenticated to Google Cloud. Therefore, Perfcheck does not need to explicitly create or manage authentication tokens. In a CI/CD pipeline, a similar method that employs Google Cloud service accounts may be used.

6.5 Benchmark template

The default template contains a k6 benchmark with latency, error rate, and response size threshold checks. Domain-specific information is filled in by the

```
export const options = {
    thresholds: {
    {{- range .Paths }}
        {{- $pathname := .Pathname }}
        'http_req_duration{group:::{{ $pathname }}}': [
        {{- range .Detail.Latency -}}
            {{- if .IsK6Supported -}}
                '{{- . -}}',
            {{- end -}}
        {{- end -}}
        ],
        ...
    {{- end }}
    },
    ...
}
```

Figure 6.6: Perfcheck: latency threshold template.

`parsers` package. The template is executed on the structure defined in figures 6.3, 6.4, and 6.5.

6.5.1 Thresholds

In k6, the SLOs are defined using thresholds. The Perfcheck template specifies a threshold for each path individually using the k6 groups system [20]. In the generated benchmark, each group corresponds to a single endpoint, with the name of the group derived from the endpoint path. The section of the template, which specifies latency thresholds, is shown in figure 6.6. The thresholds are defined in the global *options* object of k6. An example output, after the template has been applied to an API with two endpoints, each with a different latency requirement, is shown in figure 6.7. Objectives that are not supported by k6 are not included in the final benchmark.

6.5.2 Virtual users and duration

The k6 syntax for defining the number of VUs and the duration of the test can be used directly within the global `x-perfcheck` property. During generation, the values are simply included in the *options* object of the final benchmark.

```
export const options = {
    thresholds: {
        'http_req_duration{group:::/foo}': ['avg < 100',],
        'http_req_duration{group:::/bar}': ['avg < 50',],
    }
}
```

Figure 6.7: Perfcheck: generated latency thresholds.

```
import { Trend } from 'k6/metrics';

const responseBytes = new Trend('response_bytes');

export default function () {
    ...
    const response = http.get(url); // make HTTP request

    responseBytes.add(res.body.length); // populate metric
    ...
}
```

Figure 6.8: Perfcheck: custom response size metric definition.

6.5.3 The response size metric

The response size metric is not included in k6 by default. It is possible to define custom metrics of various types in k6. Figure 6.8 shows the definition of the `response_bytes` metric and the population of the metric after each request. The metric is of type `Trend`, which allows one to calculate different statistics on the values [20]. Afterwards, the new custom metric can be included in the threshold definition, similarly to built-in metrics.

6.5.4 Parameter generation

Some requests need to be provided with additional parameters in order to run the test. The generation options are described in Table 5.1. The function to produce parameters is included directly in the benchmark file. Therefore, the generation itself is performed during the execution of the benchmark. Figure 6.9 shows the subtemplate that generates parameters of any type, with the help of the `generateFromPattern` function.

```
{{ define "params" }}
  {{- range $name, $value := . }}
    {{- if $value.Pattern }}
      {{ $name }}: generateFromPattern('{{ $value.Pattern }}'),
    {{- else if $value.Example }}
      {{ $name }}: '{{ $value.Example }}',
    {{- end }}
  {{- end }}
{{- end }}
```

Figure 6.9: Perfcheck: parameter generation sub-template.

The `generateFromPattern` function parses the generation methods described in Table 5.1 and substitutes the parameters. The code of the function, which makes use of regular expressions to match the pattern string, is shown in figure 6.10.

The parameter generation sub-template is used in the definition of query parameters, path parameters, HTTP headers, and JSON body properties.

6.6 Test execution

Once the benchmark is generated, it is executed or provided to the user for further modification. If the `test` subcommand is used, the template is executed using the k6 binary installed on the system. If k6 is not installed, Perfcheck exits with an error code.

The benchmark execution itself is a subprocess of Perfcheck. Therefore, the output from k6 can be redirected to the standard output of Perfcheck. Figure 6.11 shows the detection and execution of the k6 binary.

6.7 Statistical analysis

Statistical analysis is performed after running the k6 benchmark. Precise data from the benchmark are needed to verify statistical significance. Benchmark data are stored in a JSON line (JSONL) file. A JSONL file contains a JSON object per line of the file. Each line corresponds to a single event that contains a timestamp, request details, and request metrics. After filtering the required events, the metrics are used as a data set for analysis.

With statistical analysis, Perfcheck is able to verify compliance with SLOs that use average (mean) as their statistic. These SLOs are as follows:

- average latency lower than a certain threshold,

```
import { uuidv4, randomString } from 'https://jslib.k6.io...';

const generateFromPattern = (pattern) => {
  const uuidPattern = /string\((\d+)\)/
  const stringPattern = /string\((\d+)\)/
  const boolPattern = /bool\(\)/
  const rangePattern = /range\((\d+),(\d+)\)/

  if (uuidPattern.test(pattern)) {
    const version = parseInt(pattern.match(uuidPattern)[1]);
    if (version === 4) return uuidv4();
    throw new Error('Only UUID v4 is supported');

  } else if (stringPattern.test(pattern)) {
    const length = parseInt(pattern.match(stringPattern)[1]);
    return randomString(length);

  } else if (boolPattern.test(pattern)) {
    return Math.random() < 0.5;

  } else if (rangePattern.test(pattern)) {
    const [_, min, max] = pattern
        .match(rangePattern)
        .map(x => parseInt(x);

    return Math.round(Math.random() * (max - min) + min);
  }
}
```

Figure 6.10: Perfcheck: the `generateFromPattern` function.

```
import (
        "fmt"
        "os"
        "os/exec"
)
...
_, err := exec.LookPath("k6")
if err != nil {
    fmt.Fprintln(os.Stderr, "k6 binary is not present in
        your path. Install it or add it to your path.")
    panic(err)
}

cmd := exec.Command(
    "k6", "run", benchmark.Name(),
    "--out", fmt.Sprintf("json=%s", outFile)
)

cmd.Stdout = os.Stdout // redirect output
cmd.Stderr = os.Stderr // redirect err

err = cmd.Run()
```

Figure 6.11: Perfcheck: the benchmark execution.

- average response size lower than a certain threshold.

Example usage of these SLOs is shown in figure 6.12. A new `avg_stat` indicator is implemented, which is not included in the k6 benchmark. Instead, it is only used for the purpose of statistical evaluation.

To verify that the mean of an unknown population is below a specified value, a **one-sided, one-sample t-test** can be used [30]. The random variable X is defined as either the latency or the response size of the system. The following hypotheses are chosen for the test:

- H_0: $\overline{X} \geq \mu_0$
- H_A: $\overline{X} < \mu_0$

The significance level (α) is 0.05 by default. The user can customize the strictness of the test by changing the α value, using the `--alpha` command-line flag. If the null hypothesis is rejected, the random variable is assumed to

```
{
    "swagger": "2.0",
    "info": {
        "title": "Example API"
    },
    "paths": {},
    "x-perfcheck": {
        "latency": ["avg_stat" < 50]
    }
}
```

Figure 6.12: Perfcheck: Example usage of a statistical SLO.

```
import (
    "github.com/aclements/go-moremath/stats"
)

res, err := stats.OneSampleTTest(
    makeSample(data),
    target,
    stats.LocationLess
)
```

Figure 6.13: Perfcheck: One-sided one sample t-test execution.

be below the specified threshold, and the system will pass the test. If the null hypothesis is not rejected, Perfcheck will mark the service as invalid.

The `github.com/aclements/go-moremath` library is used to perform the one-sample t-test. Figure 6.13 shows the implementation. The third argument, `stats.LocationLess` is used to specify the directionality of the test [31].

6.8 Documentation

The source repository includes a *README* file that contains the steps necessary to include Perfcheck in a service testing process. The Go package system automatically generates more detailed documentation for the tool and can be found on https://pkg.go.dev. Descriptive comments are required to properly generate the documentation.

```yaml
nfpms:
  - package_name: perfcheck
    homepage: https://github.com/zacikpet/perfcheck
    maintainer: Peter Žáčik
    description: >
        This tool is used to evaluate the compliance
        of a service with service-level objectives
        specified in the API description of
        the service.
    license: MIT
    formats:
      - apk
      - deb
      - rpm
      - termux.deb
      - archlinux
    recommends:
      - k6
    release: "1.0.0"
```

Figure 6.14: GoReleaser: Linux packages configuration

6.9 Release management

Releasing Perfcheck is achieved using GoReleaser. By specifying a configuration file in the root directory, GoReleaser can compile and package the application for multiple platforms. The default options include builds for Linux, Windows, and MacOS [27]. These builds are automatically published on the GitHub release page. Linux packages in the *.apk*, *.deb*, *.rpm* formats are also published. All Linux package formats are specified in Figure 6.14.

In addition, GoReleaser is used to generate a package for the Snapcraft store, which is available on all the most common Linux distributions [27]. Figure 6.15 contains the `snapcrafts` section of the GoReleaser configuration file.

Lastly, as per NFR01 (Installation availability), a Docker image with the pre-installed Perfcheck binary is generated and published to DockerHub. Figure 6.16 displays the required GoReleaser section. The name of the Docker image is derived from the GitHub user name and repository name. The resulting image is available on DockerHub.

```yaml
snapcrafts:
  - name: perfcheck
    publish: true
    summary: Automatic SLO compliace evaluation
    description: >
        This tool is used to evaluate the compliance
        of a service with  service-level objectives
        specified in the API description of
        the service.
    grade: stable
    confinement: classic
    license: MIT
    base: core18
```

Figure 6.15: GoReleaser: Snapcraft store configuration

```yaml
dockers:
  - image_templates:
      - zacikpet/perfcheck
```

Figure 6.16: GoReleaser: Docker image configuration

Chapter **7**

Examples

This chapter contains the example usage of Perfcheck with all of the available features. For demonstration, a new HTTP web service is created using the Gin Web Framework [32]. The endpoints within this service simulate various values of SLIs, to verify the correctness of the testing process. The values are simulated in a manner that allows easy human verification. Perfcheck is expected to exit successfully only if all specified SLOs are met.

7.1 OpenAPI

Using the `swaggo/swag` package, the OpenAPI documentation is automatically generated from comments in the source code [33]. Therefore, the developer can include the service-level objectives directly in the service source code. Alternatively, the documentation can be generated dynamically. Figure 7.1 displays the entrypoint of a Gin web framework application. A global SLO is specified for the response size metric. The information required to perform the load test is also included.

Two example endpoints are defined in the service, with service-level objectives for latency and error rate. Figure 7.2 displays the definition of the two example endpoints, with their respective service-level objectives. One of the endpoints requires a UUID path parameter that is automatically generated by Perfcheck. Both endpoints simulate delay, in order to present the correctness of Perfcheck.

Subsequently, the OpenAPI documentation is generated using the `swaggo/swag` package. Once the service is running, the visual documentation is available on the `/swagger/index.html` path. The JSON representation of the documentation is available on the `/swagger/doc.json` path. To run Perfcheck, the application must be running and must be available via a network. In this case, the service is listening to `localhost:8080`. The following command is used to execute Perfcheck:

```go
// @title Example API
// @x-perfcheck {
//   "users": 20,
//   "duration": 3,
//   "responseSize": ["avg" < 1024]
/  }

func main() {
  r := gin.Default()

  host := os.Getenv("HOST")
  port := os.Getenv("PORT")
  scheme := os.Getenv("SCHEME") // http or https

  docs.SwaggerInfo.Host = fmt.Sprintf("%s:%s", host, port)
  docs.SwaggerInfo.Schemes = []string{scheme}

  r.GET("/example-a", ExampleA)
  r.GET("/example-b/:foo", ExampleB)

  r.Run(fmt.Sprintf(":%s", port))
}
```

Figure 7.1: Example: application entrypoint with global SLO specification.

```
$ perfcheck test \
    --source openapi \
    --docsUrl localhost:8080/swagger/doc.json
```

The k6 output is printed directly on the console, followed by the results of the statistical evaluation. As expected, the first endpoint has passed the test, whereas the second test has not, because it does not comply with the latency requirement.

7.2 Google Cloud

To demonstrate integration with the Google Cloud Platform, a new application is created in the Cloud Run service. A similar setup to 7.1 is used. However, the application contains only one endpoint that listens on the root path. No OpenAPI documentation is required; therefore, the service does not include comments. A service-level objective for the application is defined us-

```
1  // @Summary Example A
2  // @Router /example-a [get]
3  // @x-perfcheck { "latency": ["avg < 200"] }
4  func ExampleA(g *gin.Context) {
5          time.Sleep(time.Millisecond * time.Duration(100))
6          g.JSON(http.StatusOK)
7  }
8
9  // @Summary Example B
10 // @Router /example-b/:foo [get]
11 // @x-perfcheck {
12 //    "latency": ["avg < 100"],
13 //    "params": { "path": { "foo": "uuid(4)" } }
14 // }
15 func ExampleA(g *gin.Context) {
16         time.Sleep(time.Millisecond * time.Duration(500))
17         g.JSON(http.StatusOK)
18 }
```

Figure 7.2: Example: application entrypoint with global SLO specification.

ing the Cloud Run Monitoring subsystem, shown in Figure 7.3. Google Cloud authentication is required to perform the load test. The following series of commands can be used to execute Perfcheck:

```
gcloud login
perfcheck test \
    --source gcloud \
    --docsUrl localhost:8080/swagger/doc.json
```

7.3 Statistical analysis

Statistical analysis of an SLO is demonstrated by the definition of an endpoint with the `avg_stat` indicator metric. Figure 7.4 shows the endpoint definition with the required annotations. The objective is a latency of less than 100 milliseconds. The endpoint pauses the execution of the handler for a random duration between 0 and 201 milliseconds, resulting in an average latency slightly higher than the required value. This demonstration does not take into account network latency, as the server and Perfcheck are running on the same machine.

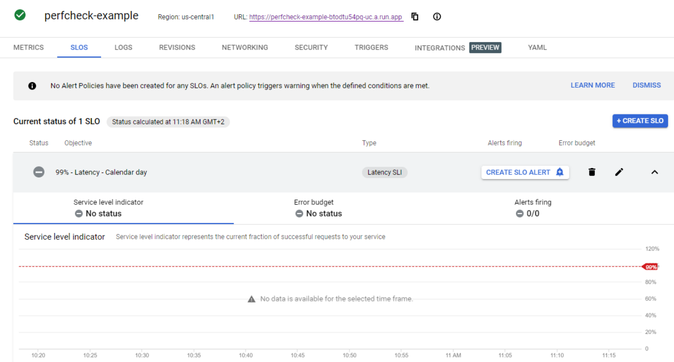

Figure 7.3: Google Cloud: SLO definition for a Cloud Run application.

```
// @Router  / [get]
// @x-perfcheck { "latency": ["avg_stat < 100"] }
func Test(g *gin.Context) {
        sleep := rand.Intn(201)
        time.Sleep(time.Millisecond * time.Duration(sleep))
        g.JSON(http.StatusOK, nil)
}
```

Figure 7.4: Example: application endpoint with a statistical latency requirement

The test is executed with 20 VUs for a duration of 1 second. In one run, the average latency is found to be $\overline{X} = 97.18\ ms$. The total number of requests to the server is $n = 3289$. Applying the one-sided one-sample t-test, as defined in Section 6.7, the p-value is calculated to be approximately 0.235. The p-value is higher than the default significance level α. Therefore, the null hypothesis is not rejected, resulting in a failed test. This result is in line with the fact that the average value generated by the `rand.Intn(201)` function call should be higher than 100. However, a regular test of the average latency would arrive at the same conclusion.

7.4 Deployment pipeline

The inclusion of Perfcheck in a deployment pipeline is demonstrated using GitHub Actions. GitHub Actions is a system capable of running CI/CD pipelines directly in the GitHub source repository [34]. Individual pipeline steps are defined in the *.github/workflows/deploy.yaml* file. In this example,

Perfcheck is used in the testing phase of a blue-green deployment. A sample Google Cloud Run application from Section 7.2 is used. The following are the steps executed by the pipeline:

1. Download source code
2. Authenticate to Google Cloud
3. Set up Google Cloud SDK
4. Install k6
5. Deploy the new (green) version of the application with no production traffic
6. Install and run Perfcheck on the green version
7. Route production traffic to the new version.

The full definition of the GitHub actions deployment pipeline is listed in the Appendix C.

7.5 Examples summary

All of the essential features are covered in the examples. In conclusion, the tool performs correctly with respect to the functional requirements defined in Section 4.3.1

Conclusion

This thesis describes how the goal of implementing software a tool that provides automatic SLO verification was achieved. The new tool, Perfcheck, enables developers to specify service-level objectives in the API description of a service and utilizes the k6 load testing tool to evaluate compliance. An emphasis is placed on correct evaluation of SLO compliance by employing basic statistical methods.

An overview of current software testing techniques was presented. Common load testing and monitoring tools were introduced, highlighting their most important features.

The use cases, functional requirements, and non-functional requirements for the new tool are introduced. Unlike other load testing tools, Perfcheck does not require knowledge of an additional programming language. Rather, developers only have to modify the existing API description. In addition, Perfcheck can integrate with the Google Cloud ecosystem. Service-level objectives defined in the Google Cloud Monitoring subsystem can be collected and a load test will be generated.

The design and architecture of the new tool is detailed in the thesis. The principles and methods of software engineering are applied to the development. The programming language, tools, and external libraries required to implement the tool in a concise, extensible, and modular manner were selected. The new tool is open source, enabling contributions and improvements by other developers. Due to the wide knowledge of the Go programming language, the participation of other developers is straightforward.

The implementation of the tool itself is described in the thesis. Code examples are used to illustrate the most critical sections of the program. The release management of the tool is refined and the tool is successfully deployed.

Examples of usage are attached that showcase the principal features of the new tool, along with a sample integration in a CI/CD pipeline.

In conclusion, all the tasks of the assignment were completed. However, there are ways Perfcheck can be improved in the future. Possible new features

include additional types of SLO, support for more service protocols, and perfecting the load test generated by the new tool. Therefore, further work is expected on the project.

Bibliography

[1] Myers, G. J.; Sandler, C.; et al. *The art of software testing*. John Wiley & Sons, 2011.

[2] Jorgensen, P.; DeVries, B. *Software Testing: A Craftsman's Approach*. 05 2021, ISBN 9781003168447, doi:10.1201/9781003168447.

[3] Beyer, B.; Murphy, N. R.; et al. *The Site Reliability Workbook: Practical Ways to Implement SRE*. O'Reilly Media, Inc., first edition, 2018, ISBN 1492029505.

[4] Mozilla Corporation. Understanding Latency. Available from: `https://developer.mozilla.org/en-US/docs/Web/Performance/Understanding_latency`

[5] Google LLC. Designing SLOs. Available from: `https://cloud.google.com/service-mesh/docs/observability/design-slo`

[6] Beyer, B.; Jones, C.; et al. *Site reliability engineering: How Google runs production systems*. " O'Reilly Media, Inc.", 2016.

[7] Google LLC. Creating an SLO. Available from: `https://cloud.google.com/stackdriver/docs/solutions/slo-monitoring/ui/create-slo`

[8] Ward, C. Ephemeral Environments for Testing. Jun 2020. Available from: `https://humanitec.com/blog/ephemeral-environments-for-testing`

[9] Google LLC. Cloud run. Available from: `https://cloud.google.com/run`

[10] Amazon Web Services LLC. AWS Fargate. Available from: `https://aws.amazon.com/fargate/`

[11] Kubernetes. Kubernetes Dashboard. Available from: https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

[12] Rabenstein, B.; Volz, J. Prometheus: A Next-Generation Monitoring System (Talk). Dublin: USENIX Association, May 2015.

[13] NGINX, Inc. NGINX Amplify. Available from: https://www.nginx.com/products/nginx-amplify/

[14] Google LLC. Creating an SLO. Available from: https://cloud.google.com/why-google-cloud

[15] Google LLC. Monitoring. Available from: https://console.cloud.google.com/monitoring

[16] Lönn, R. Open source load testing tool review 2020. Mar 2020. Available from: https://k6.io/blog/comparing-best-open-source-load-testing-tools/

[17] Artillery Software, Inc. Artillery. Available from: https://www.artillery.io/docs

[18] Gatling, Corp. Gatling. Available from: https://gatling.io/

[19] The Apache Software Foundation. Apache JMeter. Available from: vhttps://jmeter.apache.org/

[20] Grafana Labs. k6. Available from: https://k6.io/docs/

[21] Locust. Locust. Available from: https://locust.io/

[22] Stack overflow developer survey 2022. Available from: https://survey.stackoverflow.co/2022/

[23] The Go Programming Language. 2023. Available from: https://go.dev/

[24] Gitlab CI/CD variables. Available from: https://docs.gitlab.com/ee/ci/variables/

[25] Buch, D. Welcome to urfave/CLI¶. Oct 2022. Available from: https://cli.urfave.org/

[26] OpenAPI Extensions. 2023. Available from: https://swagger.io/docs/specification/openapi-extensions/

[27] Becker, C. A. GoReleaser. Available from: https://goreleaser.com/

[28] Shanley, D. libopenapi. Apr 2023. Available from: https://pkg.go.dev/github.com/pb33f/libopenapi

[29] Google Monitoring. Available from: https://pkg.go.dev/cloud.google.com/go/monitoring/apiv3

[30] Ross, A.; Willson, V. L. *One-Sample T-Test*. Rotterdam: SensePublishers, 2017, ISBN 978-94-6351-086-8, pp. 9–12, doi:10.1007/978-94-6351-086-8_2. Available from: https://doi.org/10.1007/978-94-6351-086-8_2

[31] Clements, A. gomoremaths. Jan 2021. Available from: https://pkg.go.dev/github.com/aclements/go-moremath

[32] Gin web framework. 2022. Available from: https://gin-gonic.com/

[33] The swaggo/swag package. 2023. Available from: https://pkg.go.dev/github.com/go-openapi/swag

[34] GitHub actions documentation. 2023. Available from: https://docs.github.com/en/actions

Appendix A

Acronyms

SLA Service-level agreement

SLO Service-level objective

SLI Service-level indicator

API Application programming interface

HTTP Hypertext Transfer Protocol

JSON JavaScript object notation

UML Unified Modeling Language

MTBF Mean time between failure

MTTR Mean time to repair

TCP Transmission Control Protocol

TTFB Time to first byte

VU Virtual user

SDK Software development kit

YAML YAML Ain't Markup Language

SPA Single-page application

UI User Interface

Appendix B

Contents of enclosed media

```
perfcheck..........................................................
├── pkg ............................................... Sub-packages
│   ├── benchmark ...............................................
│   ├── parsers .................................................
│   └── stat ....................................................
├── templates ....................... Template definition and execution
│   └── templates.go ............................................
├── .goreleaser.yaml ................ The GoRelaser configuration file
├── Dockerfile........................... A Dockerfile for GoReleaser
├── go.mod ................................... Go module configuration
├── go.sum ............................. List of dependecy checksums
├── LICENSE .............................. The text of the MIT license
├── main.go .............................. The application entrypoint
└── README.md...................... Instructions on how to run the tool
perfcheck-example.............................. Example application
├── .github/workflows/deploy.yaml ......... GitHub Actions workflow
├── docs ...................... Auto-generated OpenAPI documentation
├── go.mod ................................... Go module configuration
├── go.sum ............................. List of dependency checksums
└── main.go ................................... Application entrypoint
```

Appendix C

Example GitHub Actions pipeline

```yaml
on:
  push:
    branches:
      - master

env:
  PROJECT_ID: <PROJECT_ID>
  SERVICE_NAME: <GCP_CLOUD_RUN_SERVICE_NAME>
  REGION: us-central1
  MONITORING_SERVICE: <GCP_MONITORING_SERVICE_ID>

jobs:
  deploy:
    runs-on: ubuntu-latest

    permissions:
      contents: 'read'
      id-token: 'write'

    steps:
      - uses: 'actions/checkout@v3'

      - name: 'Authenticate to Google Cloud'
        uses: 'google-github-actions/auth@v1'
        with:
          workload_identity_provider: <OIDC_PROVIDER>
          service_account: <GCP_SERVICE_ACCOUNT_ID>
```

```

      - name: 'Set up Cloud SDK'
        uses: 'google-github-actions/setup-gcloud@v1'
        with:
          version: '>= 363.0.0'

      - name: Install k6
        run: |
          <GPG_SETUP>
          sudo apt-get update
          sudo apt-get install k6

      - name: Deploy green
        id: deploy
        uses: 'google-github-actions/deploy-cloudrun@v1'
        with:
          service: 'perfcheck-example'
          source: '.'
          tag: green

      - name: Setup Go environment
        uses: actions/setup-go@v4.0.0

      - name: Install perfcheck
        run: go install github.com/zacikpet/perfcheck@latest

      - name: Run perfcheck
        run: >
            perfcheck test
            --source gcloud
            --gcloudProjectId=$PROJECT_ID
            --gcloudServiceId=$MONITORING_SERVICE
            --gcloudServiceUrl ${{ steps.deploy.outputs.url }}

      - name: Route production traffic
        run: |
          export GREEN=$(
            gcloud run services describe $SERVICE_NAME \
                --platform managed \
                --region $REGION \
                --format=json \
                | jq --raw-output ".spec.traffic[] |\
                select (.tag==\"green\")|.revisionName"
```

```
71             )
72           export BLUE=$(
73             gcloud run services describe $SERVICE_NAME \
74                 --platform managed \
75                 --region $REGION \
76                 --format=json \
77                 | jq --raw-output ".spec.traffic[] |\
78                 select (.tag==\"blue\")|.revisionName"
79           )
80
81           gcloud run services update-traffic $SERVICE_NAME \
82             --update-tags=blue=$GREEN \
83             --platform managed \
84             --region $REGION
85
86           gcloud run services update-traffic $SERVICE_NAME \
87             --to-revisions=$GREEN=100 \
88             --platform managed \
89             --region $REGION
```