HAMK
Häme University
of Applied Sciences

Enhancing API reliability in Kalibro.io

Bachelor's thesis

Degree Programme in Computer Applications

Spring 2024

Mohammed Al-Saadi

Tietojenkäsittelyn koulutus / Computer Applications Tiivistelmä
Tekijä Mohammed Al-Saadi Vuosi 2024
Työn nimi Enhancing API reliablity in Kalibro.io
Ohjaaja Elina Kulmala

Tämän opinnäytetyön tarkoitus on suorittaa suorituskykytestaus Kalibro.io-työkalun kaikkiin jo valmiisiin palveluihin. Kyseisen verkkotyökalun on kehittänyt Calevala Interactive Oy. Työkalu auttaa yrityksiä parantamaan liiketoimintaansa vertaamalla verkkosivustojaan kilpailijoidensa verkkosivuihin. Työkalu tarjoaa monia palveluita, kuten verkkosivujen värien analysoinnin, sivuston käyttäjäystävällisyyden ja mobiiliyhteensopivuuden tarkistamisen sekä verkkosivuston saavutettavuuden varmistamisen.
Koska Kalibro.io-työkalu käsittelee paljon HTTP-pyyntöjä, Calevala Interactive Oy:n oli tarpeen tarkistaa, pystyvätkö sen palvelut käsittelemään erilaisia käyttäjäkuormia tuhansien pyyntöjen kuormituksessa. Tämä oli varmistettava ennen työkalun tuotantoa, jotta työkalu olisi vakaa ja toimisi ilman ongelmia.

Tämä opinnäytetyö on toiminnallinen tapaustutkimus. Se perustuu pääasiassa kirjoittajan henkilökohtaisiin löydöksiin ja Internet-lähteisiin, kuten K6:n viralliseen verkkosivustoon, e-kirjoihin ja muihin verkkosivustoihin.

K6-suorituskykytestaustyökalun ja sen dokumentaation avulla oli helppoa tunnistaa neljä suurta suorituskykytestaustyyppiä, jotka ovat kuormitus-, stressi-, piikki- ja kestävyystestit. Jokaisella niistä on erilainen lähtökohta. Kuormitustestaus tarkistaa järjestelmän suorituskyvyn normaalin, oletetun käyttäjämäärän käsittelyssä, kun stressitestaus taas tarkastelee järjestelmän suorituskykyä äärimmäisessä kuormituksessa, joka kasvaa jatkuvasti tapahtuman ajankohdan mukaan aina sen murtumispisteeseen asti. Piikkitestauksella tarkistetaan suoriutumista suuren käyttäjämäärän äkillisessä lisääntymisessä ja kestävyystestillä arvioidaan järjestelmän suorituskykyä käyttäjien pitkäaikaisella kuormituksella.
Calevala Interactive Oy oli alustavasti tyytyväinen suorituskykytestien tuloksiin, sillä kaikki palvelut toimivat hyvin, mutta parantamisen varaa on vielä, erityisesti HTTP-kestoaika. Työkalu on kuitenkin edelleen kehitteillä, mikä tarkoittaa, että sen valmistuttua on suoritettava uusi testi järjestelmän lopullisen kuormituksen ja suorituskyvyn määrittämiseksi. Opinnäytetyön tuloksena on vaiheittainen kuvaus Kalibro.io:n suorituskykytestauksesta K6-työkalulla. Se voi ohjata lukijoita suorituskyvyn testausprosessin läpi. Se voi myös auttaa lukijoita ymmärtämään nopeasti työkalun peruskäsitteen ja K6:n edut.

HAMK
Häme University
of Applied Sciences

Degree Programme in Computer Applications		Abstract
Author	Mohammed Al-Saadi	Year 2024
Subject	Enhancing API relaiblity in Kalibro.io	
Supervisors	Elina Kulmala	

The purpose of this thesis is to execute performance testing on all of the ready services in the kalibro.io tool. The online tool was developed by Calevala Interactive Ltd. It helps business owners improve their business by comparing their websites to those of their competitors. The tool provides many services, such as analyzing website colors, checking if the website is user-friendly, mobile-friendly, website accessibility, and more. Since the tool deals with lots of HTTP requests, it was necessary for the company to check if the services would be able to handle a different load of users with thousands of requests. That had to be ensured before the tool went to production to make sure the tool was stable and could perform without any issues.

This thesis is a practical and research case study. It is mainly based on the author's personal experience and Internet sources such as the K6 official website, e-books, and other websites.

With the help of the K6 performance testing tool and its documentation, it was easy to conduct four major performance testing types: load, stress, spike, and soak tests. Each one of them has a different scenario. For instance, load testing to check the normal, assumed amount of users the system can handle, stress testing to check system performance under great stress load that increases continuously depending on the scenario time all the way to its breaking point; spike testing for a sudden increase of a big amount of users and soak testing to evaluate the system performance and durability with a load of users for a long period of time.

Calevala Interactive Ltd. was satisfied with the initial performance testing results since all the services performed well. However, there is still room for improvement, specifically in HTTP duration time. The tool is still under development, which means other tests must be conducted when it is complete to determine the system's real load and performance.

The thesis results are a step-by-step description of performance testing for kalibro.io using the K6 tool. It can guide readers through the process of performance testing. It can also help readers understand the basic concept of the tool and the benefits of K6.

Keywords: API testing, K6 tool, load testing, stress testing, soak testing.
Pages: 32 pages and appendices 1 pages

Glossary

API: Application Programming Interface

VS-Code: Visual Studio Code

CPU: Central Processing Unit

EC2: Amazon Elastic Compute Cloud

OS: Operating System

VUs: Virtual Users

HTTP: Hypertext Transfer Protocol

Content

Figures

1 Introduction

Software testing history goes back to the early 1950s when debugging guaranteed software program reliability. The definition of software testing was “what programmers do to discover bugs in their applications” (Lewis et al., 2009, p. 3). This implies finding bugs and mistakes during the software development phase (Amazon, n.d.). Over the years, various turning points have characterized software testing improvement. In 1958, Gerald M. Weinberg established the first Software Testing Group. Software testing was mentioned in a NATO report in 1968. Auto Tester presented the first software testing tool in the late twentieth century. The current century focuses on using artificial intelligence tools, and many other types of testing, consisting of unit testing, integration testing, and overall performance testing (Geeksforgeeks, 2022).

Software testing is a phase that must be planned and implemented during the software development life cycle. It makes sure that the software performs as planned. Testing evaluates software to identify differences between actual and expected results. Identifying bugs before software is deployed to the end user is necessary. It prevents business owners from probable disasters that could compromise the software's usability, reliability, and performance. In addition, it reduces the cost of the software (Bierig et al., 2022, p. 3).

This thesis was produced for Calevala Interactive Ltd. The company was developing a site audit tool (Kalibro.io) that helps business owners improve their businesses by evaluating how their website performs compared to competitors. In the early stages of the software planning, the team designed and implemented many services. One of the services provided for the customer was to check if any other website uses the business domain as a referral, check if other business or websites uses their brand name, check if the website is user-friendly, get all internal and external links for the website, and many other services. All those services were later canceled because they mainly depended on web scraping to extract the required data, which is not permissible on many websites. Calevala’s team is currently developing new services that do not depend on web scraping and follow all regulations, guaranteeing that no concerns will arise in the future.

This paper focused on the reliability of the Kalibrio.io API, with all ready endpoints subjected to overall performance testing. The test included a full evaluation of the services with load, stress, spike, and soak tests; each test performed a specific task to check how services were performing under different conditions, which, in the long run, does the best for customers in real-world scenarios.

Research questions.

- How does performance testing contribute to software development outcomes?
- How has the K6 tool improved Kalibro.io services?

2 Software Testing Techniques

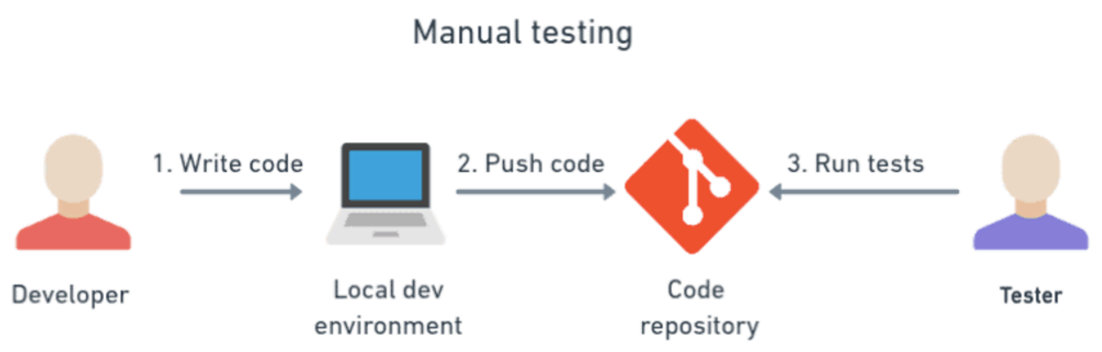
Manual testing and automated testing are the two primary techniques for testing software applications. This chapter briefly explains both techniques.

2.1 Manual Testing

Manual testing is a software testing technique in which the developer executes the test cases manually without using an automated tool. The test cases are executed by the tester manually according to a predefined test plan, which ensures that the software behaves as mentioned in the requirement document (Tran, 2022).

Manual testing can detect software defects/bugs. A defect is the difference between the expected output and the output generated by the software. Manual testing is usually performed before automated testing, and it happens while the software is being developed. Its disadvantages are that it requires effort and time but guarantees bug-free software. To perform Manual testing, one must know manual testing techniques, not automated testing tools. Manual testing is essential because one of the fundamentals of software testing is that "100% automation is not possible." (Java point, n.d). Figure 1 highlights the Manual testing flow.

Figure 1 Manual testing flow image



2.2 Automated Testing

Automated testing is a software testing technique that uses automation tools like Selenium to conduct pre-scripted tests on a system before it is released to the end users to ensure the software functions as expected without any serious flaws or problems (Alsmadi, 2012, p. 1). Automated testing is used for a wide range of applications, like web applications, mobile apps, and more. In some cases, humans performing manual tests on software might not be a good idea; it consumes time and effort. Additionally, humans most likely make mistakes, which leads to delays in product release time, reduces software reliability, and higher costs for business owners (Geeksforgeeks, 2024).

The main purpose of using automated tests is that they are performed quickly and efficiently without the need for manual intervention. In terms of how the test is performed, automated testing typically involves creating test scripts or scenarios designed to mimic user actions on an application. A testing tool can run these scripts automatically and report any errors or bugs detected in the system. This can save time and resources and improve the overall accuracy of the testing process (Rahman, 2019).

Figure 2 Automated testing flow image

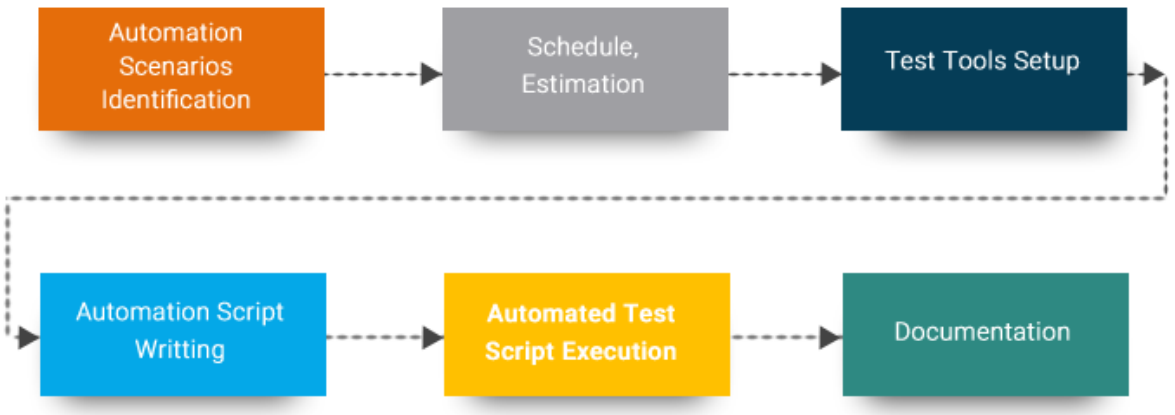

Automation Scenarios Identification
Schedule, Estimation
Test Tools Setup
Automation Script Writting
Automated Test Script Execution
Documentation

Figure 2 highlights the Automated testing flow. It starts with identifying the test scenario, setting up the tools and environment, writing the test script, running the test, and documenting the results.

3 Software Testing Types

Software testing types are the different ways and strategies testers use to fully check a software or application for bugs and anything that might compromise the software's reliability and performance. Each type focuses on a specific test objective and different test strategy (Testautomationresources, n.d.). Each tester or company has its own way, techniques, and Tools for conducting software testing that suits the project. All fall into two main categories: functional testing and non-functional testing (Chandrasekara & Herath, 2019, p. 2). This chapter will mainly highlight Non-Functional testing since it is the type used in this thesis.

Figure 3 Software testing categories image

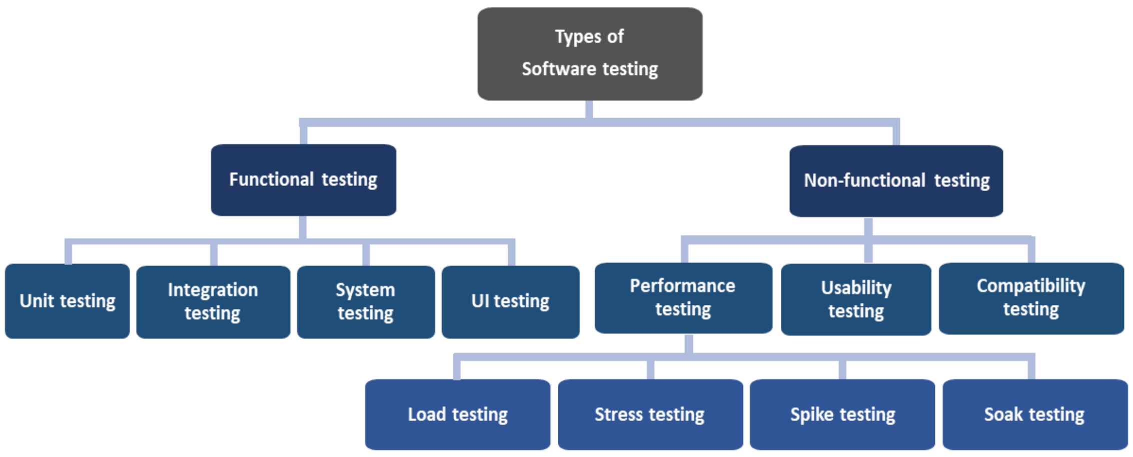


Figure 3 highlights Software testing types.

3.1 Functional Testing

Functional testing is a software testing process in which a system is validated against collected requirements, which determine the development of the software. More simply, functional testing can be seen as a way to ensure that all functional components of the software work as required (Chandrasekara & Herath, 2019, p. 2). This is used to verify that the software produces the output the end user requires. During this process, each feature of the software is adapted to your company's needs. The main goal of functional testing is to test all the application functions, the primary entry function, and the GUI screen flow (Geeksforgeeks, 2023; Testsigma, n.d). Functional testing can be divided into many techniques. This chapter won't go through them since it's not used in this thesis.

Figure 4 Functional testing flow image

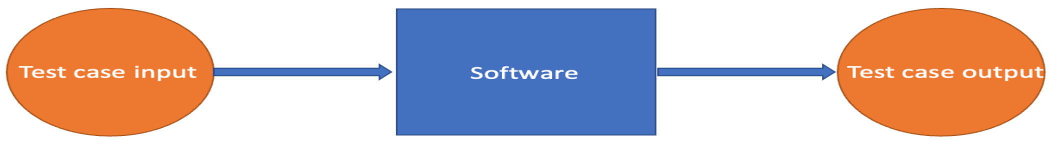


Figure 4 highlights the functional testing flow: Plan a test case, implement it, and analyze the results.

3.2 Non-Functional Testing

Non-functional testing is a type of software testing that checks a system's non-functional aspects, such as performance, usability, readability, and more. It helps test the system's readiness. Non-functional testing defines a system's way of operating rather than pointing out its specific behaviour. This totally contrasts with functional testing, which tests against the functional requirements (Chandrasekara & Herath, 2019, p. 3).

Non-functional testing is used to check and evaluate non-functional parameters that are not covered in functional testing. It is equally important as functional testing. Under non-functional testing, all non-functional parameters, such as an application's speed, scalability, performance, reliability, and efficiency, are tested (Nayyar, n.d; Geeksforgeeks, 2024). Non-functional testing can be divided into many techniques. This chapter will explore performance testing and its types, which is the most important type of non-functional testing technique and is the type used in this thesis.

Performance testing: is a type of non-functional testing and one of the most important phases of any product launch as it verifies and validates the overall product performance. It ensures that the software is up to a given workload (Chandrasekara & Herath, 2019, p. 3). The purpose is to identify bottlenecks, check system performance under different user loads and conditions, and make sure that the system can handle a certain number of users or transactions (Tricentis, 2024; Geeksforgeeks, 2023). Put simply, performance testing checks how a system behaves and responds to different situations. A system may run with no issues with only 1,000 concurrent users, but is it going to work with 100,000 users?

Performance testing includes many types. Load, stress, soak, and spike testing. Load testing is to identify the maximum number of users or transactions the system can handle, stress testing is to identify the system's breaking point and any potential issues that may appear under heavy load conditions, soak testing is to evaluate the behavior of a system when a significant workload has given continuously. Finally, we have the spike test to evaluate the behavior of a system when the load is suddenly increased (Geeksforgeeks, 2023). This chapter will dive a bit, explaining the four major types of performance testing mentioned above.

Load Testing assesses the system's current performance in terms of concurrent users or requests per second. It is run to determine whether the system is meeting the performance goals, assessing the system's current performance under typical and peak load (Gregorio, 2023; P, 2023).

Spike testing is a variation of a stress test, but it does not gradually increase the load. Instead, it spikes to an extreme load over a very short window of time. Run a stress test to Determine how the system will perform under a sudden surge of traffic and whether it will recover once the traffic has decreased (Gregorio, 2023; P, 2023).

Stress Testing is used to determine the limits of the system and verify its stability and reliability under extreme conditions. A stress test is run to Determine how the system will behave under extreme conditions, determine the system's breaking point and failure mode, and determine whether the system will recover without manual intervention after the stress test is over (Gregorio, 2023; P, 2023).

Soak testing is used to validate the reliability of the system over a long time. Run a soak test to Verify that the system doesn't suffer from bugs or memory leaks, which result in a crash or restart afterward. Verify that expected application restarts don't lose requests. Make sure the database doesn't exhaust the storage space and stop. Ensure the external services you depend on don't stop working after a certain number of requests are executed (Gregorio, 2023; P, 2023).

Figure 5 Performance testing process flow image

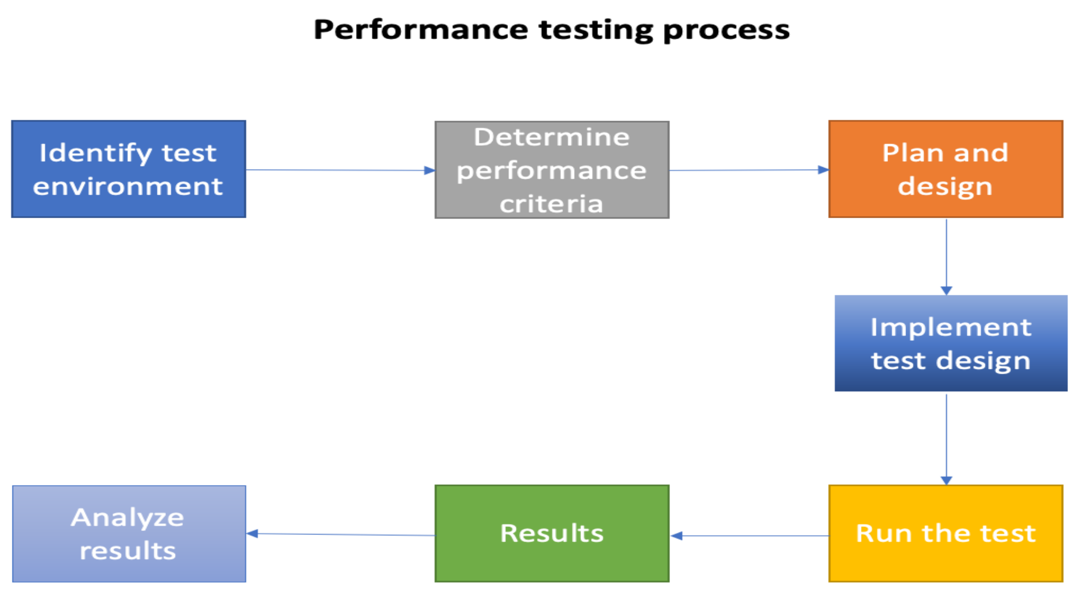

Performance testing process
Identify test environment
Determine performance criteria
Plan and design
Implement test design
Run the test
Results
Analyze results

Figure 8 highlights the Performance testing process flow: identifying the test environment, planning the test, implementing the code, running the test, getting results, and then analyzing the results.

4 Software Testing Tools

Software testing tools are developed to provide functionality and operations that will assist developers or testers in effectively testing software for both automated and manual tests. These tools help in different areas, like functional and non-functional testing. It enables testers to perform automated testing, cross-browser testing, mobile testing, compatibility testing, reliability, usability, and other factors to evaluate how well the system performs and what kind of user experience it offers (Prasad, 2004, p. 98).

These tools can ensure that software applications meet the required quality, security, and performance standards, ensuring higher user satisfaction and better business outcomes. Ultimately, the use of software testing tools can improve the quality of software applications thesis (Tricentis, 2021). This chapter will mainly focus on and highlight the tools used in this.

4.1 Development and Scripting Tools

The Development and Scripting Tools section will discuss the essential tools for writing and managing test scripts. This includes IDEs like Visual Studio Code (VS-Code), which make coding and debugging simpler and faster. These tools are important for testers and developers, helping them work more efficiently and effectively. The scripting tool used for the thesis is VS-Code this subchapter will briefly explain the tool.

Visual Studio Code: is an open-source code editor by Microsoft. It was released in 2015 and has become one of the most popular integrated development environments among developers. VS Code is considered a code editor. Its built-in capabilities include syntax highlighting, code completion corrections, code navigation, code refactoring, running, and debugging. All these improved capabilities make VS Code more than a basic code editor. it has a fast yet simple editing capability and is light in weight. It comes with inbuilt support and extension support for a wide range of programming and scripting languages, including JavaScript, typescript, python, ruby, go, HTML CSS, docker, node.js, yaml, and many more (visualstudio, 2024).

4.2 K6 Performance Testing Tools

Testing tools are software applications that help developers identify and measure the performance and reliability of their systems under different load condition scenarios. These tools are needed to simulate a load of thousands of users to measure the system's response time, throughput, bottlenecks, and issues affecting the application's overall performance and user experience (geeksforgeeks, 2023). Otherwise, it would be hard and time-consuming to manually simulate a load of thousands of users when the software is still in the development phase. Various tools are available in the market, including open-source and commercial tools. Some popular performance testing tools are JMeter and K6, each with its own unique features and capabilities. This thesis will focus mainly on K6 since it is the tool used to perform performance testing on kalibro.io services.

K6 is an open-source load-testing tool designed and released by Grafana Labs and its community in 2017. The tool is built to help developers create and execute performance testing for APIs, microservices, and websites. K6 aims to make performance testing more accessible and user-friendly for testers. K6 is designed and programmed with JavaScript, a pulper and widely used programming language with a big community to help when needed (Grafana, n.d). Additionally, it provides a simple command-line interface and a powerful scripting engine that enables developers to write complex and simple test cases.

K6 can perform four major types of performance testing: load, stress, spike, and soak. Load testing is to identify the normal load of users the system can handle, stress testing is to identify the system's breaking point, soak testing is used to evaluate the system when a significant workload has been given continuously, and finally, spike test is used to evaluate the system when the load is suddenly increased (Geeksforgeeks, 2023). Also, K6 includes many unique features, such as running tests on the local machine or in the cloud and generating reports while the test is executed. The tool is highly scalable and can easily simulate thousands of virtual users (Grafana, n.d).

Since its release, the K6 tool has become popular among developers and used by companies like Netflix, Microsoft, and Amazon(K6, 2023). The K6 community is also getting bigger, with many developers contributing to the project and sharing their experiences and best practices. Overall, K6 is a powerful and easy-to-use load-testing tool designed to help developers and DevOps teams ensure the performance and reliability of their applications.

Figure 6 K6 workflow

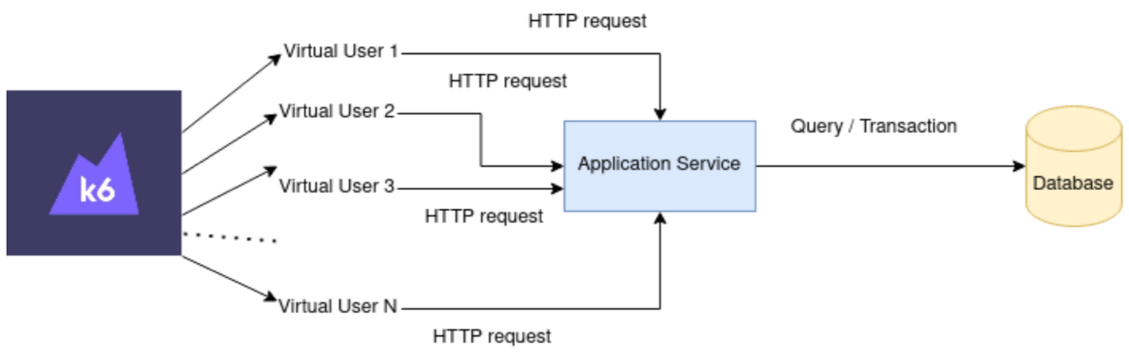


Figure 11 highlights the K6 workflow. Simulates many users that are sending requests to the application server. To check the performance.

5 Test Infrastructure Setup and Objectives

This chapter highlights the initial steps required to set up and plan the performance testing outlined in this thesis. It begins by defining the research objectives, which guide the goals and methods of our testing efforts. Next, it describes the services offered by Kalibro.io, followed by the installation of K6, a tool used for testing processes.

5.1 Research Objectives

The main objective of this case study was to conduct a detailed performance on the Kalibro.io site audit tool. As performance testing was the primary focus, the objectives were to use industry-standard tools such as K6 to effectively assess Kalibro responsiveness, robustness, and scalability under real-world scenarios. Specifically, Load testing evaluates the system with a normal number of concurrent users, stress testing pushes the system to the breaking point to determine user capacity, and spike testing accounts for sudden changes in user capacity to ensure the software will perform as expected soak to evaluate the system performance under extended period to identify a memory leak, server crash and more.

Planning for these test cases began after the author started the internship at Calevala Interactive Ltd. The work started with phase one, collecting all the necessary information and data on Kalibro.io services. Data gathering was not difficult under the guidance of the development team. The second phase was to collect data on the tools used to perform the software testing; once the second phase was complete, the author began testing the services in the local environment.

This thesis applied an action research study to answer the research questions. The goal was to understand how Kalibro currently worked and identify any bugs or malfunctions that could compromise the reliability and scalability of the services during the testing process. The performance and UI testing provided a holistic assessment of Kalibro's capabilities for real-world usage under load. The results highlighted current strengths as well as any areas needing improvement in the tool to deliver a smooth, responsive user experience even at high concurrency.

5.2 Kalibro.io Services Description

This chapter highlights the services provided by Kalibro.io before designing and implementing performance testing cases, so the reader has a full idea of what is being tested.

Six services were ready to be tested. Those services were signup and sign-in to create an account and log into the kalibrio.io dashboard. IP Location data service provides detailed geographical and network information based on an IP address, including the country, city, exact coordinates, and the associated time zone and region.

Who is data service providing comprehensive Who is information for domain names, including registration details, contact information, and domain status, this assists in identifying domain ownership.

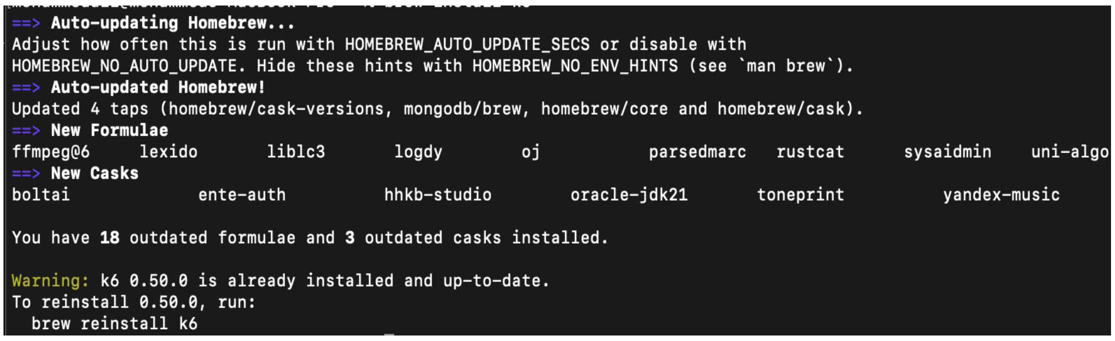
Color data service for getting all the color analysis data related to all the URLs that the specific customer has added to his/her account in the Kalibro.io system. Data includes analysis data for colors used in the web page, different color tones, individual colors, color tone ratio, main color tone, descriptive adjectives and business sectors associated with the main color tone, and total individual colors on the page.

Classification data service to get all the classification data related to all the URLs that the specific customer has added to his/her account in the Kalibro.io system. The classification is based on the screenshot taken from the web page that the URL refers to. The screenshot was taken using the Firefox browser. Classification is made with the help of machine learning models trained with the YOLOv5 -algorithm.

5.3 K6 Installation

Installing K6 is straightforward. It enables installing the tool library on a local or virtual machine easily without any complicated steps. K6 has packages for Linux, Mac, and Windows. The installation commands are different for each OS. To install it on Mac, which is the OS used for local testing, navigate to the Mac terminal, useing the Homebrew software package management system, write the command ‘brew install k6’, and wait until the installation is completed. Figure 13 Highlights the K6 Mac terminal installation success output (K6, n.d).

Figure 7 K6 Mac terminal installation success



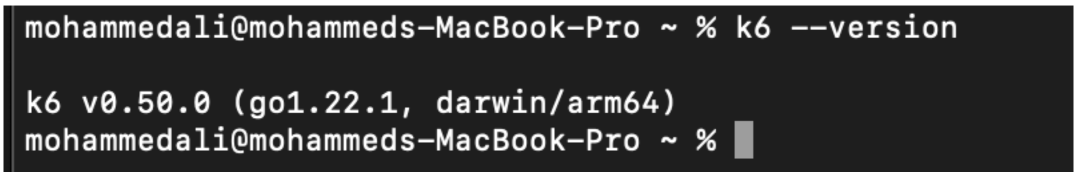
```
==> Auto-updating Homebrew...
Adjust how often this is run with HOMEBREW_AUTO_UPDATE_SECS or disable with
HOMEBREW_NO_AUTO_UPDATE. Hide these hints with HOMEBREW_NO_ENV_HINTS (see `man brew`).
==> Auto-updated Homebrew!
Updated 4 taps (homebrew/cask-versions, mongodb/brew, homebrew/core and homebrew/cask).
==> New Formulae
ffmpeg@6     lexido       liblc3       logdy        oj           parsedmarc   rustcat      sysaidmin    uni-algo
==> New Casks
boltai                ente-auth             hhkb-studio           oracle-jdk21          toneprint             yandex-music

You have 18 outdated formulae and 3 outdated casks installed.

Warning: k6 0.50.0 is already installed and up-to-date.
To reinstall 0.50.0, run:
  brew reinstall k6
```

To check if K6 is installed on the MacOS system, navigate to the Terminal, and run the command ‘k6 --version’. Figure 14 highlights the K6 installed version (K6, n.d).

Figure 8 K6 installed version

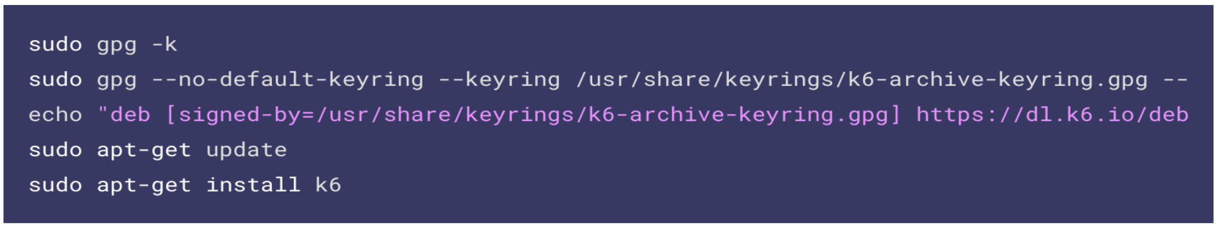


```
mohammedali@mohammeds-MacBook-Pro ~ % k6 --version

k6 v0.50.0 (go1.22.1, darwin/arm64)
mohammedali@mohammeds-MacBook-Pro ~ %
```

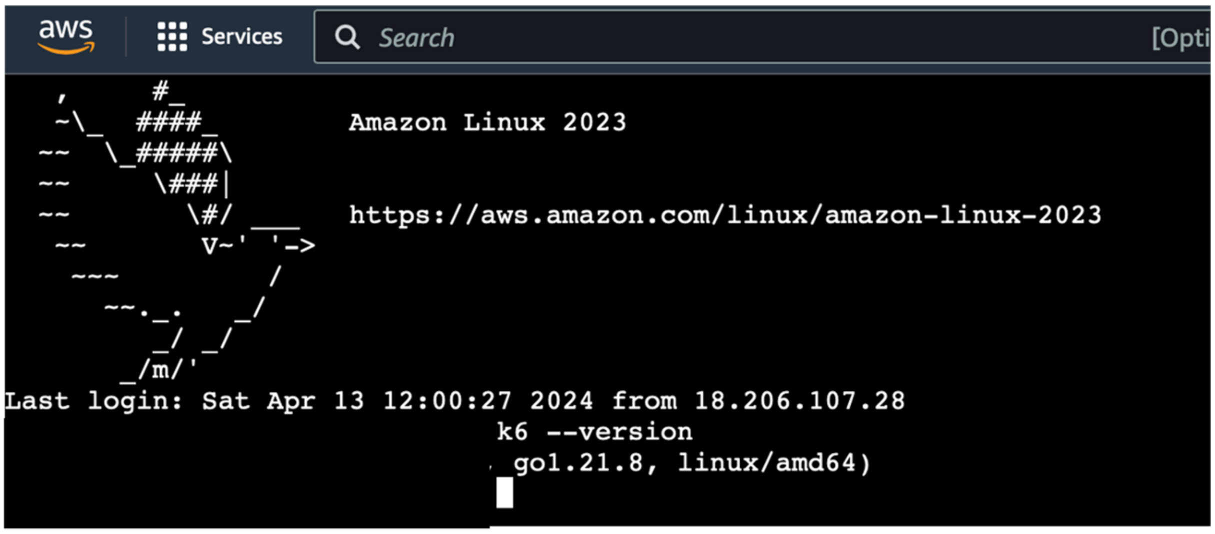
Kalibro.io's backend development team has deployed the ready services to the AWS cloud, this chapter will also cover installing K6 in the AWS EC2 instance. Install K6 in Amazon Web Services using the EC2 service, which provides a virtual machine instance that can be used for many purposes. First, create the VM instance with the necessary requirements, such as the OS, CPUs, and other available options. In this case, the OS used is Linux, and to install K6, there are a few steps. Navigate to the VM Web console as shown in Figure 16 and write the flowing commands shown in Figure 15 or follow the K6 official documentation. To check the version simply write ‘k6 --version’ (K6, n.d).

Figure 9 Commands to install K6 on Linux



```
sudo gpg -k
sudo gpg --no-default-keyring --keyring /usr/share/keyrings/k6-archive-keyring.gpg --
echo "deb [signed-by=/usr/share/keyrings/k6-archive-keyring.gpg] https://dl.k6.io/deb
sudo apt-get update
sudo apt-get install k6
```

Figure 10 AWS web console



```
  ,     #_
  ~\_  ####_        Amazon Linux 2023
 ~~  \_#####\
 ~~     \###|
 ~~       \#/ ___   https://aws.amazon.com/linux/amazon-linux-2023
  ~~       V~' '->
   ~~~         /
     ~~._.   _/
        _/ _/
      _/m/'
Last login: Sat Apr 13 12:00:27 2024 from 18.206.107.28
[redacted] k6 --version
            go1.21.8, linux/amd64)
```

Figure 15 shows the AWS web console, where all test cases can be pushed from local to cloud.

6 Performance Testing Implementation

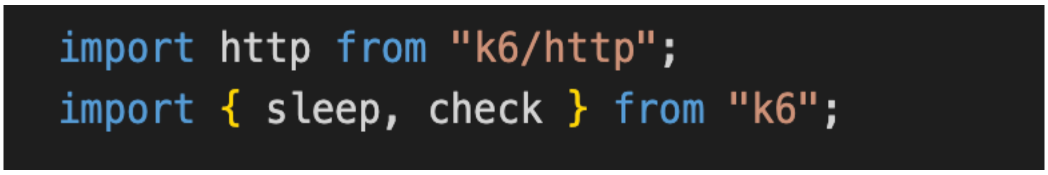
The first step in performing the testing was to determine whether it was in a local or cloud environment. Running it in a local environment is typically sufficient for development and small-scale testing, but cloud environments might be necessary for larger-scale tests. Since Kalibro.io was still in development and had no potential customers, it was considered a small-scale project, and tests were conducted in the local environment. This meant that the installation of K6 and the services were both hosted on a local machine.

Performance testing included four major types: load, stress, spike, and soak. All the test cases designed for these types were applied to kalibro.io services, so all services were tested using the same test cases.

After the installation and setting up of the environment were completed, the test phase started. Before designing and implementing the test cases, first, understand the context of a K6 performance-testing script. The body of the test script contains an object (options), a default function (default function), and an importing section. Each one serves a different purpose (K6, n.d).

The importing section of a K6 script includes many different modules that extend the script's functionality. These modules can be part of the standard K6 library or external JavaScript libraries that K6 supports. For instance, the HTTP Module, the most commonly used module in K6 scripts for making HTTP and HTTPS requests to your target website or API, the Checks Module for defining the success criteria of a test run, and more as shown in code snippet 1 (K6, n.d).

Code snippet 1 K6 importing section

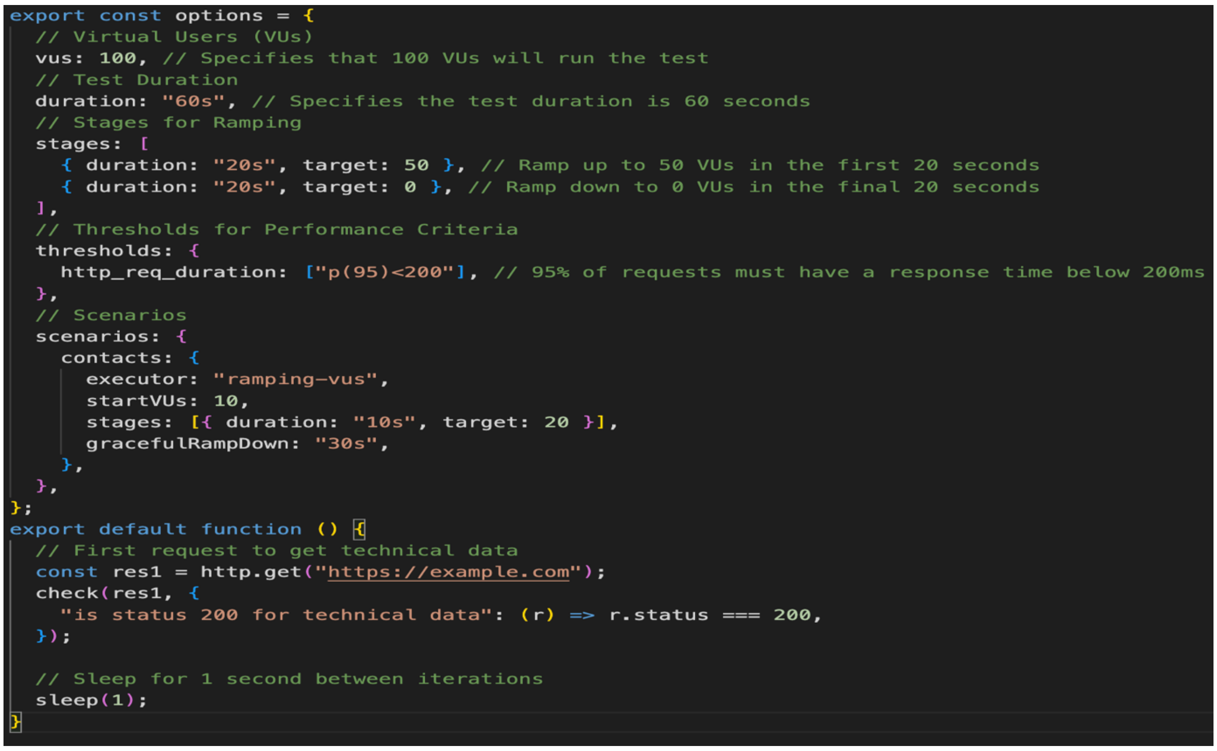


```
import http from "k6/http";
import { sleep, check } from "k6";
```

The option is an object where all the configurations of the test parameters are defined, such as virtual users, the duration of the test, and the ramp-up and ramp-down stages. It is possible to define multiple scenarios for the test. Thresholds are used to determine whether the test passes or fails based on criteria like response time and request errors (K6, n.d), as shown in code snippet 2.

The default function is defined as the actual performance-testing activities. Each VU will execute this function repeatedly according to the test configuration (options). Typical activities within this function include Making HTTP Requests and validating Responses (K6, n.d), as shown in code snippet 2.

Code snippet 2 K6 test case script body



```javascript
export const options = {
  // Virtual Users (VUs)
  vus: 100, // Specifies that 100 VUs will run the test
  // Test Duration
  duration: "60s", // Specifies the test duration is 60 seconds
  // Stages for Ramping
  stages: [
    { duration: "20s", target: 50 }, // Ramp up to 50 VUs in the first 20 seconds
    { duration: "20s", target: 0 }, // Ramp down to 0 VUs in the final 20 seconds
  ],
  // Thresholds for Performance Criteria
  thresholds: {
    http_req_duration: ["p(95)<200"], // 95% of requests must have a response time below 200ms
  },
  // Scenarios
  scenarios: {
    contacts: {
      executor: "ramping-vus",
      startVUs: 10,
      stages: [{ duration: "10s", target: 20 }],
      gracefulRampDown: "30s",
    },
  },
};
export default function () {
  // First request to get technical data
  const res1 = http.get("https://example.com");
  check(res1, {
    "is status 200 for technical data": (r) => r.status === 200,
  });

  // Sleep for 1 second between iterations
  sleep(1);
}
```

6.1 Load test

A load test was conducted using the K6 testing tool to evaluate the system's performance under varying user loads. The load test duration should not be long since it measures the system's normal-to-peak load under normal usage. The test progressively increased the number of virtual users from 100 to 200 and then decreased back to zero. The test duration time was 35 minutes. Let’s break the test case into smaller parts.

In the option object, the test began with a gentle increase of 100 VUs over 5 minutes, hammering the backend with requests. Then, there was an increase to 125 VUs over 10 minutes, then to 150 and 175 VUs over 10 minutes, testing the application's performance as the load increased, then maintaining the 200 VUs for another 15 minutes to assess the system’s behavior under sustained load. Then, it reached the last stage, where the load was reduced to zero over the last 5 minutes to assess the system’s recovery.

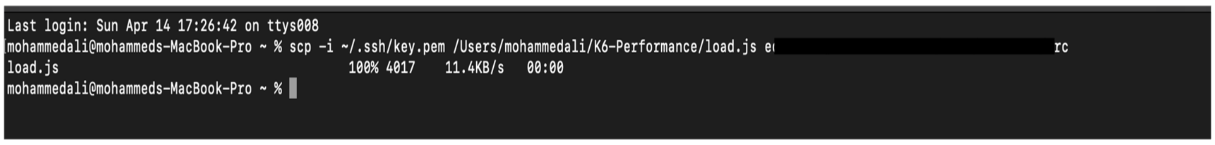
Thresholds were used to ensure performance standards. http_req_duration: Defined a threshold for the duration of HTTP requests. It ensured that 95 % of requests were completed within 1.35 seconds. http_req_failed: Specified a threshold for the error rate of HTTP requests. It ensured that the error rate remained below 3 %. http_req_connecting: Set a threshold for the time taken to establish connections for HTTP requests. It aimed for 95 % of connections to be established within 200 milliseconds. http_req_receiving: Defined a threshold for the time taken to receive responses for HTTP requests. It ensured that 95 % of responses were received within 300 milliseconds.

The default function started with an HTTP request to fetch/post the data at a predetermined API endpoint in kalibro.io services. Upon receiving the response, it checked to ensure the HTTP status was 200, confirming successful data retrieval. After executing the request, the function included a random math time pause (sleep), which was used to simulate a realistic delay between user actions, mimicking how a user might interact with the application under normal conditions. In the default function, there can be multiple HTTP/HTTPS requests, which means the ability to conduct the testing on multiple endpoints and saves time and effort.

Two HTTP methods were used in the K6 default function script: POST and GET. GET fetches API data from the kalibro.io database, while the POST method posts login and signup data to the backend. The payload consisted of JSON strings containing user data for login and signup requests. The params object specified request headers, so the Content-Type was set for application/JSON to inform the server that the body of the request is in JSON format.

Running the script locally, open the terminal, navigate to the test case directory, and run the command ‘k6 run file-name.js’ (K6, n.d). Running the test script on the AWS EC2 VM instance, we first need to transfer the code to the cloud. To do that, navigate to the terminal and use the command ‘scp -i ~/.ssh/key.pem path-to-test-case.js your-EC2-instance’. This uploads the script from the local to the cloud VM as shown in Figure 17. Running the test scripts in the cloud: Open the cloud console instance and run the command ‘k6 run file-name.js’ (K6, n.d). The test results are presented in the next chapter.

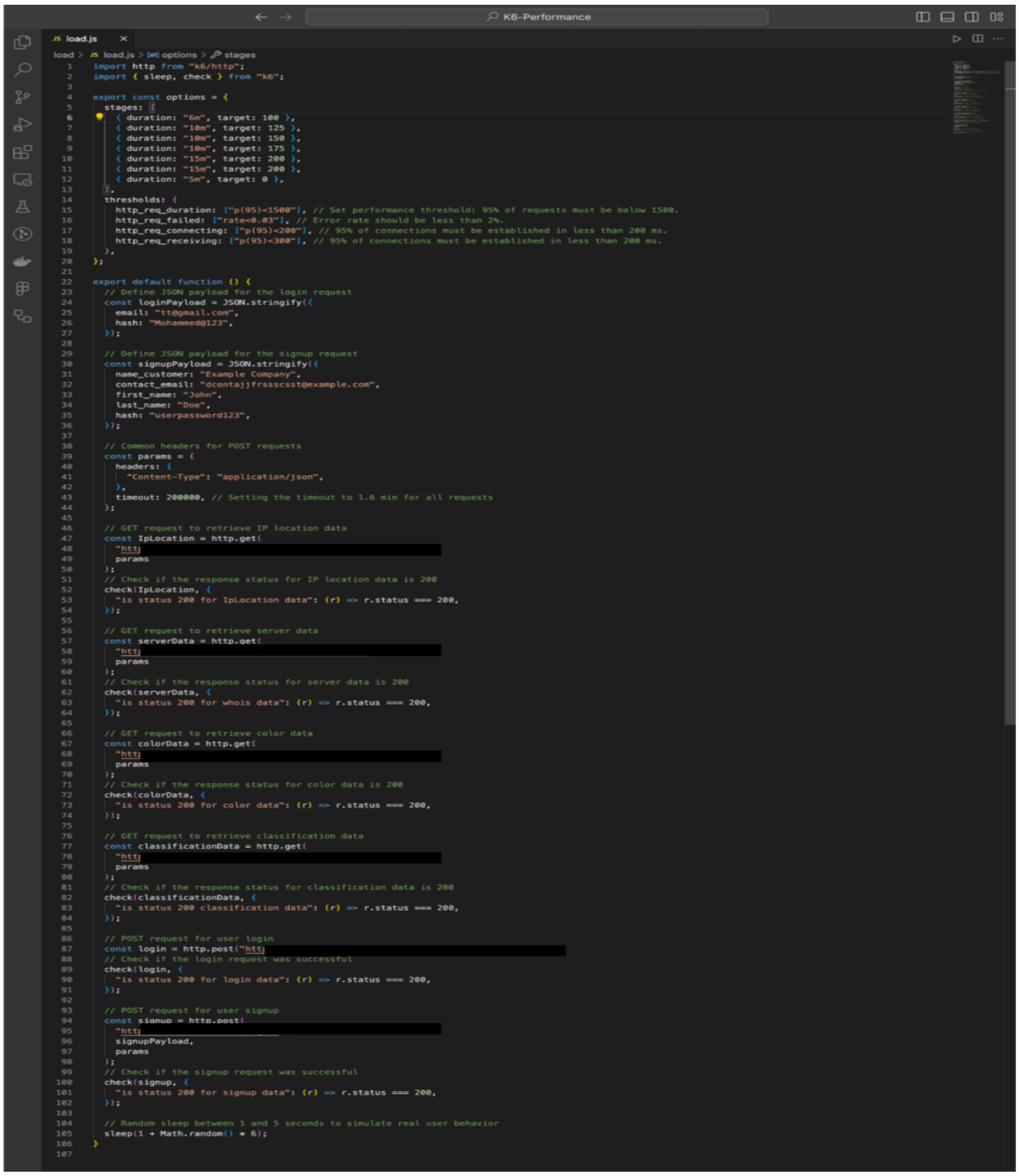
Figure 11 Transfer test case to EC2 VM



```
Last login: Sun Apr 14 17:26:42 on ttys008
mohammedali@mohammeds-MacBook-Pro ~ % scp -i ~/.ssh/key.pem /Users/mohammedali/K6-Performance/load.js e[redacted]rc
load.js                                       100% 4017    11.4KB/s   00:00
mohammedali@mohammeds-MacBook-Pro ~ %
```

Figure 16 shows that the K6 script was transferred successfully from local to AWS.

Code snippet 3 K6 load test case script



```
import http from "k6/http";
import { sleep, check } from "k6";

export const options = {
  stages: [
    { duration: "6m", target: 100 },
    { duration: "10m", target: 125 },
    { duration: "10m", target: 150 },
    { duration: "10m", target: 175 },
    { duration: "15m", target: 200 },
    { duration: "15m", target: 200 },
    { duration: "5m", target: 0 },
  ],
  thresholds: {
    http_req_duration: ["p(95)<1500"], // Set performance threshold: 95% of requests must be below 1500.
    http_req_failed: ["rate<0.03"], // Error rate should be less than 2%.
    http_req_connecting: ["p(95)<200"], // 95% of connections must be established in less than 200 ms.
    http_req_receiving: ["p(95)<300"], // 95% of connections must be established in less than 200 ms.
  },
};

export default function () {
  // Define JSON payload for the login request
  const loginPayload = JSON.stringify({
    email: "tt@gmail.com",
    hash: "Mohammed@123",
  });

  // Define JSON payload for the signup request
  const signupPayload = JSON.stringify({
    name_customer: "Example Company",
    contact_email: "dcontajjfrssscsst@example.com",
    first_name: "John",
    last_name: "Doe",
    hash: "userpassword123",
  });

  // Common headers for POST requests
  const params = {
    headers: {
      "Content-Type": "application/json",
    },
    timeout: 200000, // Setting the timeout to 1.6 min for all requests
  };

  // GET request to retrieve IP location data
  const IpLocation = http.get(
    "http[redacted]
    params
  );
  // Check if the response status for IP location data is 200
  check(IpLocation, {
    "is status 200 for IpLocation data": (r) => r.status === 200,
  });

  // GET request to retrieve server data
  const serverData = http.get(
    "http[redacted]
    params
  );
  // Check if the response status for server data is 200
  check(serverData, {
    "is status 200 for whois data": (r) => r.status === 200,
  });

  // GET request to retrieve color data
  const colorData = http.get(
    "http[redacted]
    params
  );
  // Check if the response status for color data is 200
  check(colorData, {
    "is status 200 for color data": (r) => r.status === 200,
  });

  // GET request to retrieve classification data
  const classificationData = http.get(
    "http[redacted]
    params
  );
  // Check if the response status for classification data is 200
  check(classificationData, {
    "is status 200 classification data": (r) => r.status === 200,
  });

  // POST request for user login
  const login = http.post("http[redacted]
  // Check if the login request was successful
  check(login, {
    "is status 200 for login data": (r) => r.status === 200,
  });

  // POST request for user signup
  const signup = http.post(
    "http[redacted]
    signupPayload,
    params
  );
  // Check if the signup request was successful
  check(signup, {
    "is status 200 for signup data": (r) => r.status === 200,
  });

  // Random sleep between 1 and 5 seconds to simulate real user behavior
  sleep(1 + Math.random() * 6);
}
```

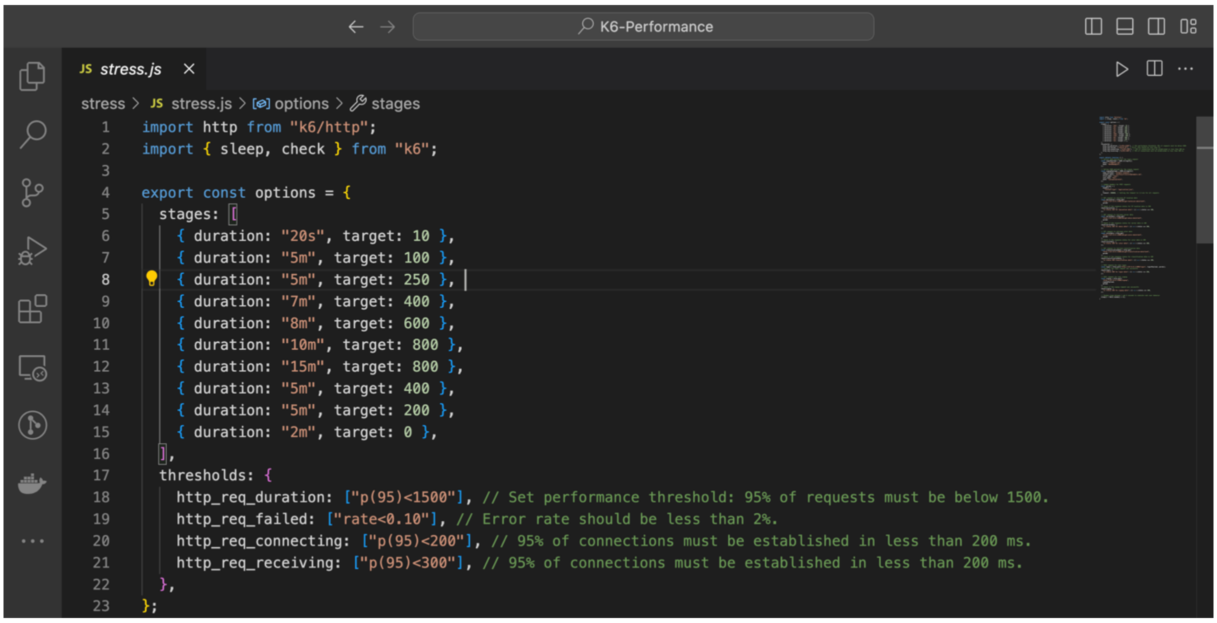
Code snippet 3 explains the script of the load test case. Since the test is conducted on the same endpoints, the default function in the load test will be used in all other test cases. Only options objects will be changed.

6.2 Stress test

The stress test was designed to challenge the system's performance by progressively increasing the load. It began with a quick ramp-up to 10 virtual users in 20 seconds to test initial capacity, followed by steady increments to 100 VUs over 5 minutes, and then to 250 VUs to load higher stress. The load was further increased to 400 VUs and held at 600 VUs to test limits and stability under extreme conditions. The peak stress load of 800 VUs was maintained to assess maximum stress performance, followed by a reduction phase to 400 and then 200 VUs to observe recovery dynamics. The test concluded by winding down to zero VUs over 2 minutes, allowing observation of the system’s final recovery behaviors. This approach provided a comprehensive assessment of the system’s performance across different stress conditions.

The load test already explained the body of the default function script and running it. In this test, only the stages in the option object were changed. The default function remained the same since the test was performed on the same endpoints.

Code snippet 4 K6 Stress test option object script



```javascript
import http from "k6/http";
import { sleep, check } from "k6";

export const options = {
  stages: [
    { duration: "20s", target: 10 },
    { duration: "5m", target: 100 },
    { duration: "5m", target: 250 },
    { duration: "7m", target: 400 },
    { duration: "8m", target: 600 },
    { duration: "10m", target: 800 },
    { duration: "15m", target: 800 },
    { duration: "5m", target: 400 },
    { duration: "5m", target: 200 },
    { duration: "2m", target: 0 },
  ],
  thresholds: {
    http_req_duration: ["p(95)<1500"], // Set performance threshold: 95% of requests must be below 1500.
    http_req_failed: ["rate<0.10"], // Error rate should be less than 2%.
    http_req_connecting: ["p(95)<200"], // 95% of connections must be established in less than 200 ms.
    http_req_receiving: ["p(95)<300"], // 95% of connections must be established in less than 200 ms.
  },
};
```

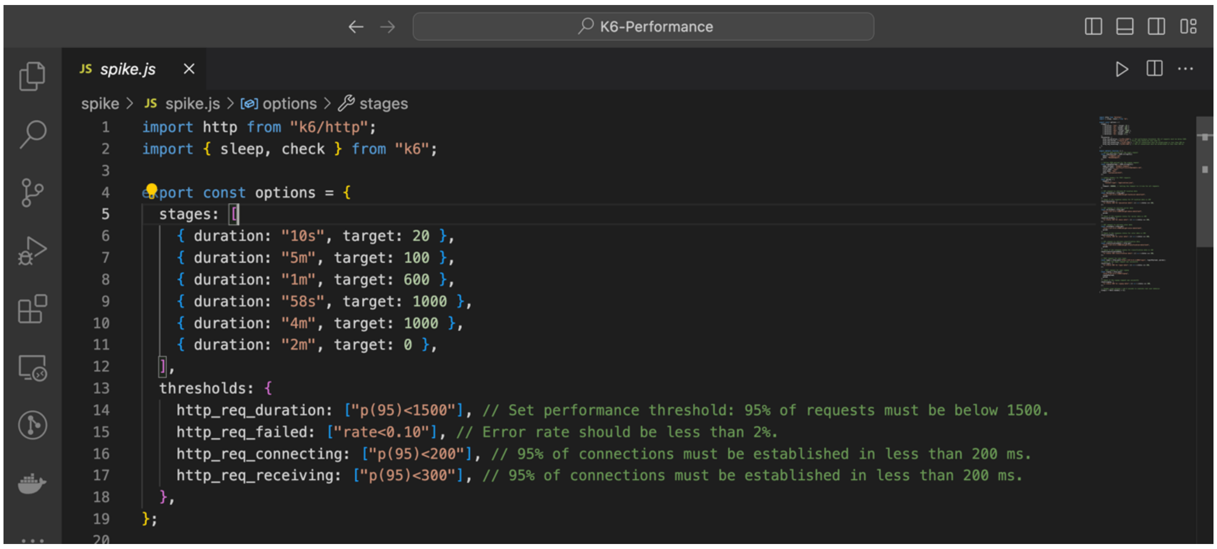
Code snippet 4 explains the configuration of the spike test case in the option object. The deflate function used in this test case is the same one in the load test since the test is conducted on the same endpoints.

6.3 Spike test

The spike test was designed to evaluate the system under a sudden load of users. The test started with progressively increased system load, starting with a target of 20 virtual users (VUs) over 10 seconds and increasing to 100 VUs for 5 minutes. Then, it quickly increased to 600 VUs for 1 minute first spike, followed by a second spike to 1000 VUs for the next 58 seconds. This peak load is sustained for 4 minutes, testing the system’s peak capacity before winding down to zero VUs over the final 2 minutes, evaluating system recovery.

The load test already explained the body of the default function script and running it. In this test, only the stages in the option object were changed. The default function remained the same since the test was performed on the same endpoints.

Code snippet 5 K6 spike test option object script



```
import http from "k6/http";
import { sleep, check } from "k6";

export const options = {
  stages: [
    { duration: "10s", target: 20 },
    { duration: "5m", target: 100 },
    { duration: "1m", target: 600 },
    { duration: "58s", target: 1000 },
    { duration: "4m", target: 1000 },
    { duration: "2m", target: 0 },
  ],
  thresholds: {
    http_req_duration: ["p(95)<1500"], // Set performance threshold: 95% of requests must be below 1500.
    http_req_failed: ["rate<0.10"], // Error rate should be less than 2%.
    http_req_connecting: ["p(95)<200"], // 95% of connections must be established in less than 200 ms.
    http_req_receiving: ["p(95)<300"], // 95% of connections must be established in less than 200 ms.
  },
};
```

Code snippet 5 explains the configuration of the spike test case in the option object. The deflate function used in this test case is the same one in the load test since the test is conducted on the same endpoints.

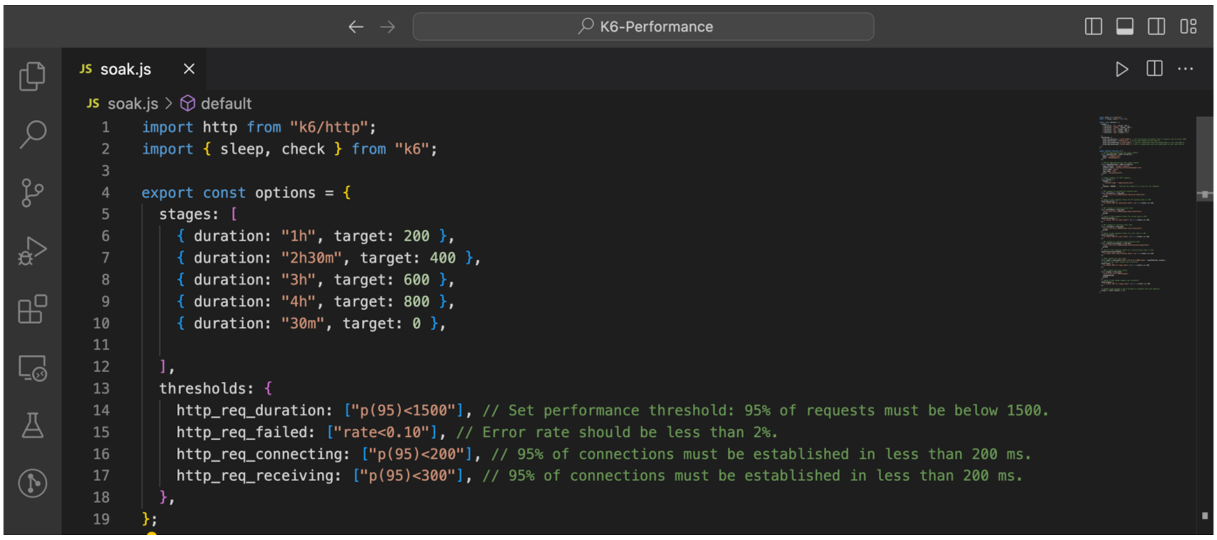
6.4 Soak test

A soak test, also known as an endurance test, is conducted to evaluate the system's performance over an extended period. The test duration time should be long enough to identify any memory leak that can't be detected in a short-time test or database connection exhaustion, and the server can handle an extended load without crashing or needing a restart. The load on the system is gradually increased to simulate different load levels:

starting with a one-hour warm-up at 200 virtual users (VUs), followed by a small load for two and a half hours at 400 VUs, a larger load for three hours at 600 VUs, and a peak load for four hours at 800 VUs, then ramp down to zero VUs in the last 30 minutes to evaluate the system recovery. This test helps pinpoint any performance issues and ensures the system's stability and efficiency under extended load.

The load test already explained the body of the default function script and running it. In this test, only the stages in the option object were changed. The default function remained the same since the test was performed on the same endpoints.

Code snippet 6 K6 soak test option object script



```javascript
import http from "k6/http";
import { sleep, check } from "k6";

export const options = {
  stages: [
    { duration: "1h", target: 200 },
    { duration: "2h30m", target: 400 },
    { duration: "3h", target: 600 },
    { duration: "4h", target: 800 },
    { duration: "30m", target: 0 },

  ],
  thresholds: {
    http_req_duration: ["p(95)<1500"], // Set performance threshold: 95% of requests must be below 1500.
    http_req_failed: ["rate<0.10"], // Error rate should be less than 2%.
    http_req_connecting: ["p(95)<200"], // 95% of connections must be established in less than 200 ms.
    http_req_receiving: ["p(95)<300"], // 95% of connections must be established in less than 200 ms.
  },
};
```

Code snippet 6 explains how to configure the soak test case in the option object. Since the test is conducted on the same endpoints, the deflate function used in this test case is the same one used in the load test.

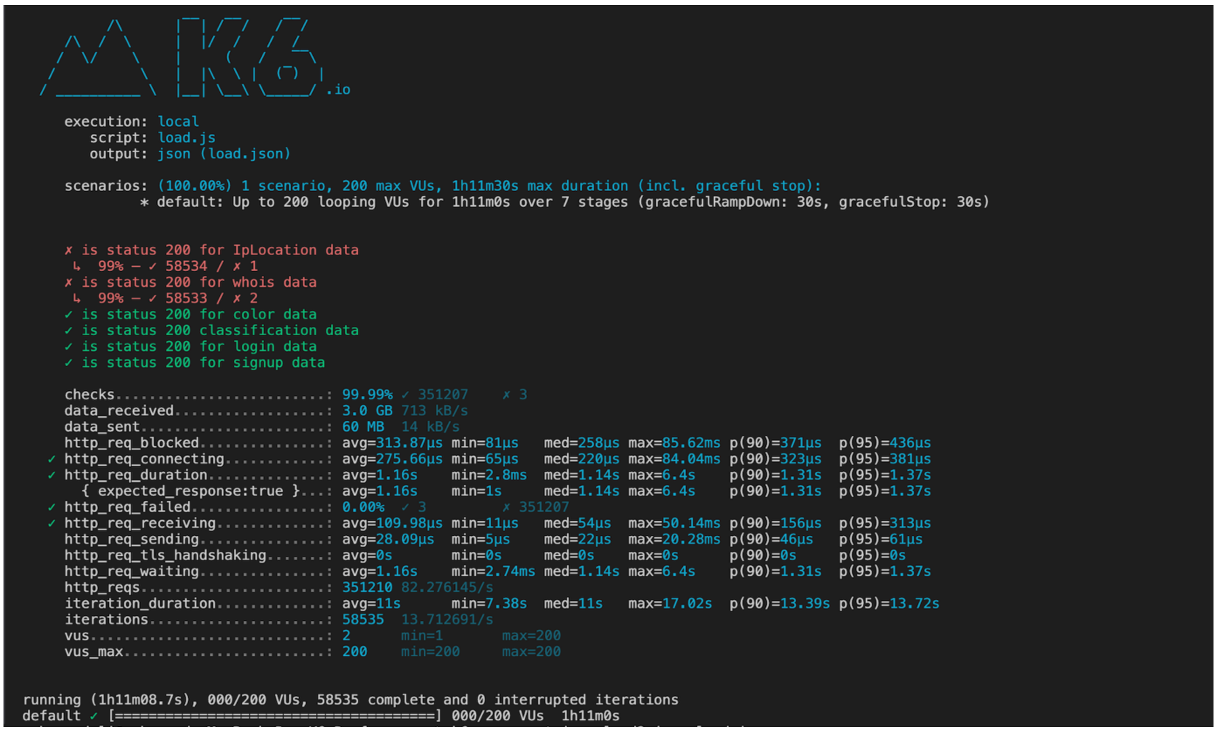
7 Test Results and Conclusion

This chapter reviews the performance testing results from previous chapters, highlights all the issues that occurred in the test cases, and analyzes the output results. In addition, the conclusion of the thesis.

7.1 Load Test Results

The load test results confirmed the system's robust performance, with 99.99% of all HTTP requests returning a 200 status. The average HTTP request duration was 1.16 seconds, with a peak at 6.4 seconds, identifying potential delay points during high load scenarios. The system efficiently handled data transactions, with HTTP request receiving times averaging 109.98 microseconds, but some outliers extending up to 50.14 milliseconds. Remarkably, there were only three failed HTTP requests, underscoring the reliability of the system under load. Over the test period, the system managed 351,210 HTTP requests and completed 58,535 iterations, reflecting its capacity to sustain high levels of traffic. Despite these positive outcomes, the iteration durations, averaging 11 seconds and reaching up to 17.02 seconds, suggest areas for further optimization to enhance the overall user experience during peak usage.

Figure 12 K6 load test results output



```
          /\      |‾‾| /‾‾/   /‾‾/
     /\  /  \     |  |/  /   /  /
    /  \/    \    |     (   /   ‾‾\
   /          \   |  |\  \ |  (‾)  |
  / __________ \  |__| \__\ \_____/ .io

     execution: local
        script: load.js
        output: json (load.json)

     scenarios: (100.00%) 1 scenario, 200 max VUs, 1h11m30s max duration (incl. graceful stop):
              * default: Up to 200 looping VUs for 1h11m0s over 7 stages (gracefulRampDown: 30s, gracefulStop: 30s)


     ✗ is status 200 for IpLocation data
      ↳  99% — ✓ 58534 / ✗ 1
     ✗ is status 200 for whois data
      ↳  99% — ✓ 58533 / ✗ 2
     ✓ is status 200 for color data
     ✓ is status 200 classification data
     ✓ is status 200 for login data
     ✓ is status 200 for signup data

     checks.........................: 99.99% ✓ 351207     ✗ 3
     data_received..................: 3.0 GB 713 kB/s
     data_sent......................: 60 MB  14 kB/s
     http_req_blocked...............: avg=313.87µs min=81µs   med=258µs max=85.62ms p(90)=371µs  p(95)=436µs
   ✓ http_req_connecting............: avg=275.66µs min=65µs   med=220µs max=84.04ms p(90)=323µs  p(95)=381µs
   ✓ http_req_duration..............: avg=1.16s    min=2.8ms  med=1.14s max=6.4s    p(90)=1.31s  p(95)=1.37s
       { expected_response:true }...: avg=1.16s    min=1s     med=1.14s max=6.4s    p(90)=1.31s  p(95)=1.37s
   ✓ http_req_failed................: 0.00%  ✓ 3          ✗ 351207
   ✓ http_req_receiving.............: avg=109.98µs min=11µs   med=54µs  max=50.14ms p(90)=156µs  p(95)=313µs
     http_req_sending...............: avg=28.09µs  min=5µs    med=22µs  max=20.28ms p(90)=46µs   p(95)=61µs
     http_req_tls_handshaking.......: avg=0s       min=0s     med=0s    max=0s      p(90)=0s     p(95)=0s
     http_req_waiting...............: avg=1.16s    min=2.74ms med=1.14s max=6.4s    p(90)=1.31s  p(95)=1.37s
     http_reqs......................: 351210 82.276145/s
     iteration_duration.............: avg=11s      min=7.38s  med=11s   max=17.02s  p(90)=13.39s p(95)=13.72s
     iterations.....................: 58535  13.712691/s
     vus............................: 2      min=1       max=200
     vus_max........................: 200    min=200     max=200


running (1h11m08.7s), 000/200 VUs, 58535 complete and 0 interrupted iterations
default ✓ [======================================] 000/200 VUs  1h11m0s
```

Figure 17 highlights the output of the K6 load test results that are explained at the beginning of this chapter.

Figure 13 Performance metrics correlation during load testing

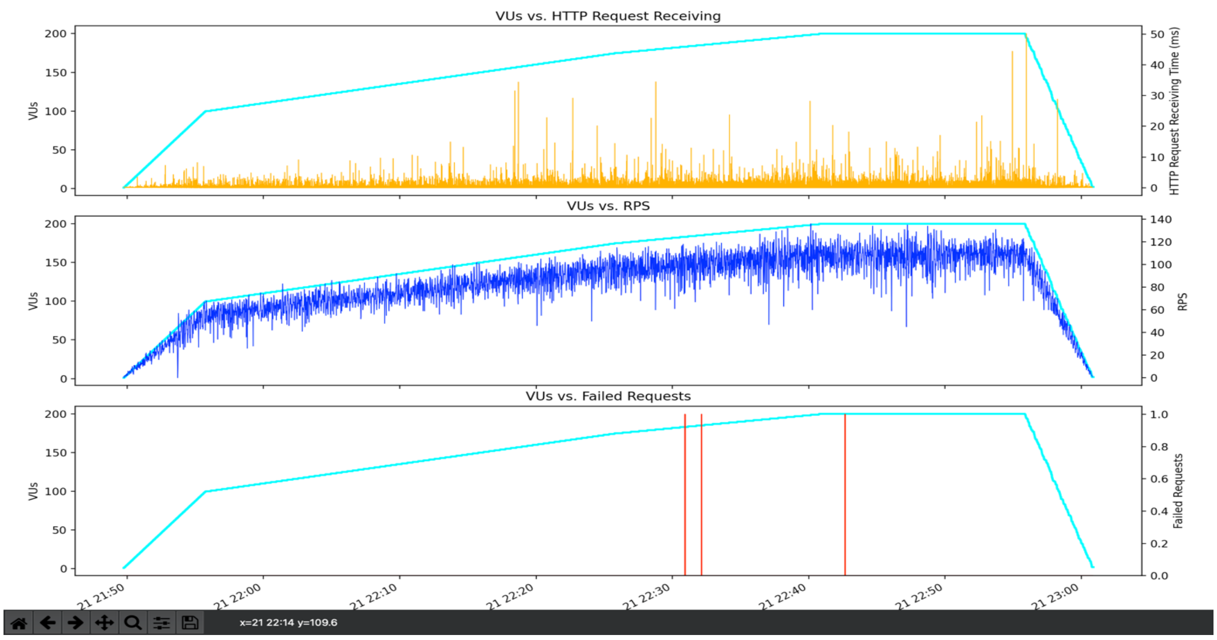
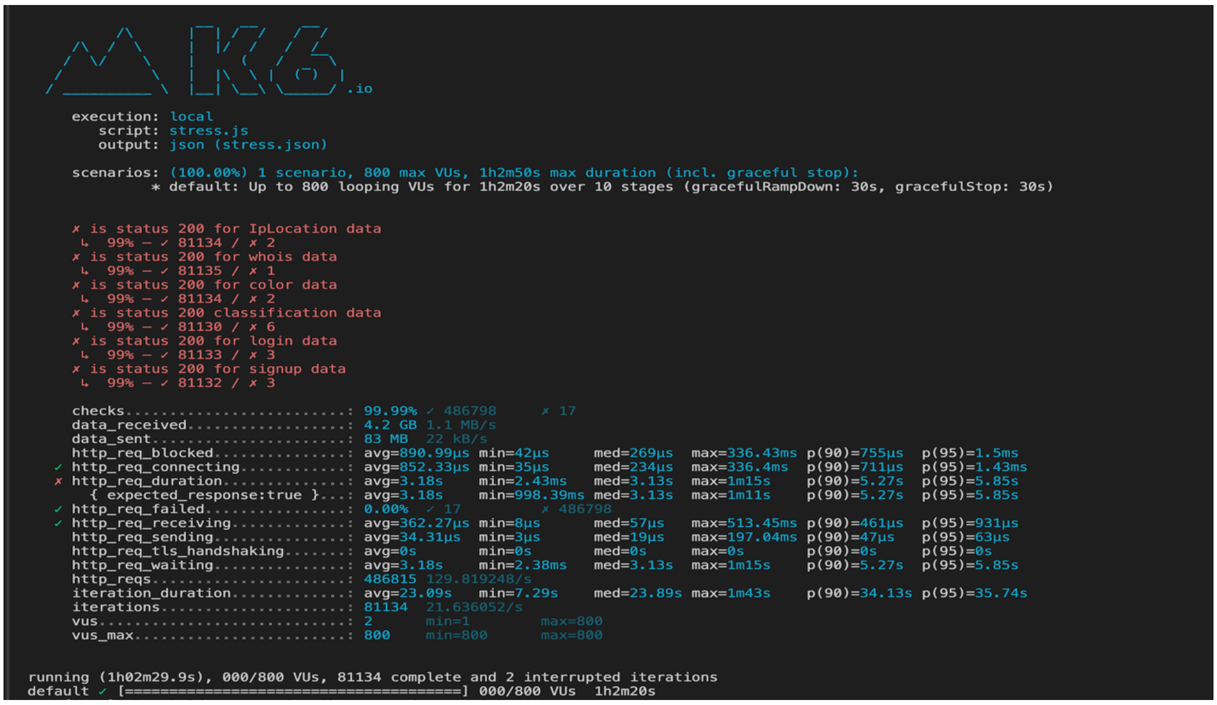


Figure 18 shows the relationship between the number of virtual users (VUs) and three important performance Metrics. The first graph shows the HTTP request receiving times as the VU load increases. The second graph presents the requests per second (RPS), which appears to maintain relative consistency despite the growing number of VUs. The final graph highlights the presence of minimal failed requests, which only become noticeable at peak VU levels.

7.2 Stress Test Results

The stress test results confirmed the system's robust performance, with 99.99% of all HTTP requests returning a 200 status. with an average HTTP request duration of 3.18 seconds and peak performance under stress reaching 1m15s, suggesting room for optimization at high loads. The system efficiently managed data, as shown by the average HTTP request receiving time of 362.27 microseconds, but with some slower outliers. The system proved its robustness with a total of 486,815 requests processed at a rate of 129 per second and only 17 failed requests. Iterations took an average of 23.09 seconds, with longer durations during peak activity, highlighting potential areas for enhancement. Overall, the system is capable of handling all the users and requests even when it’s at its peak stress load, but there is still room for improvement to make the HTTP request duration less to improve user experience under stress.

Figure 14 K6 stress test results



```
        execution: local
           script: stress.js
           output: json (stress.json)

        scenarios: (100.00%) 1 scenario, 800 max VUs, 1h2m50s max duration (incl. graceful stop):
                 * default: Up to 800 looping VUs for 1h2m20s over 10 stages (gracefulRampDown: 30s, gracefulStop: 30s)


        ✗ is status 200 for IpLocation data
         ↳  99% — ✓ 81134 / ✗ 2
        ✗ is status 200 for whois data
         ↳  99% — ✓ 81135 / ✗ 1
        ✗ is status 200 for color data
         ↳  99% — ✓ 81134 / ✗ 2
        ✗ is status 200 classification data
         ↳  99% — ✓ 81130 / ✗ 6
        ✗ is status 200 for login data
         ↳  99% — ✓ 81133 / ✗ 3
        ✗ is status 200 for signup data
         ↳  99% — ✓ 81132 / ✗ 3

        checks.........................: 99.99% ✓ 486798      ✗ 17
        data_received..................: 4.2 GB 1.1 MB/s
        data_sent......................: 83 MB   22 kB/s
        http_req_blocked...............: avg=890.99µs min=42µs      med=269µs   max=336.43ms p(90)=755µs   p(95)=1.5ms
      ✓ http_req_connecting............: avg=852.33µs min=35µs      med=234µs   max=336.4ms  p(90)=711µs   p(95)=1.43ms
      ✗ http_req_duration..............: avg=3.18s    min=2.43ms    med=3.13s   max=1m15s    p(90)=5.27s   p(95)=5.85s
          { expected_response:true }...: avg=3.18s    min=998.39ms med=3.13s   max=1m11s    p(90)=5.27s   p(95)=5.85s
      ✓ http_req_failed................: 0.00%   ✓ 17          ✗ 486798
      ✓ http_req_receiving.............: avg=362.27µs min=8µs       med=57µs    max=513.45ms p(90)=461µs   p(95)=931µs
        http_req_sending...............: avg=34.31µs  min=3µs       med=19µs    max=197.04ms p(90)=47µs    p(95)=63µs
        http_req_tls_handshaking.......: avg=0s       min=0s        med=0s      max=0s       p(90)=0s      p(95)=0s
        http_req_waiting...............: avg=3.18s    min=2.38ms    med=3.13s   max=1m15s    p(90)=5.27s   p(95)=5.85s
        http_reqs......................: 486815  129.819248/s
        iteration_duration.............: avg=23.09s   min=7.29s     med=23.89s max=1m43s    p(90)=34.13s p(95)=35.74s
        iterations.....................: 81134   21.636052/s
        vus............................: 2       min=1         max=800
        vus_max........................: 800     min=800       max=800


     running (1h02m29.9s), 000/800 VUs, 81134 complete and 2 interrupted iterations
     default ✓ [======================================] 000/800 VUs  1h2m20s
```

Figure 19 highlights the results output of the K6 stress test that is explained at the beginning of this chapter.

Figure 15 Performance metrics correlation during stress testing

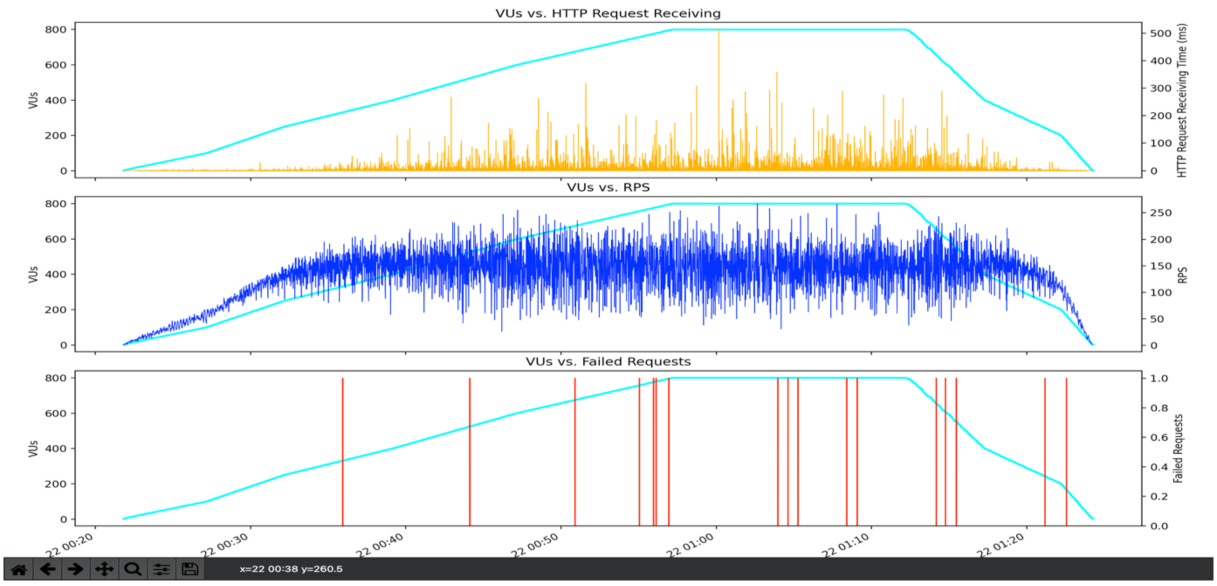
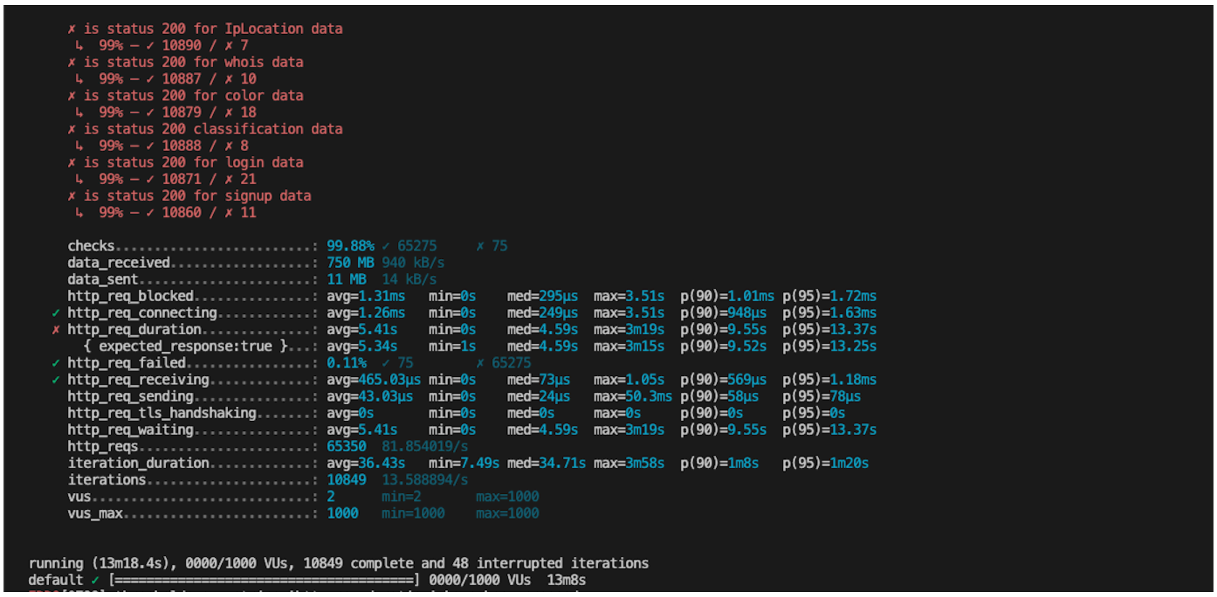


Figure 20 presents the system's behavior under a stress test. The top graph tracks the HTTP request receiving time against the number of virtual users, revealing heightened response times correlating with the higher VU counts. The middle graph shows the number of requests per second with VUs; this indicates that the response time gets higher when there are more users. The bottom graph highlights failed requests, which notably increase as the system approaches and sustains its peak load, providing insight into the system's limits and areas for optimization under stress conditions.

7.3 Spike Test Results

The spike test results were good but not good enough. The system was able to perform well under stress, with 99.88% of HTTP requests successfully processed. The average request durations were 5.41s, yet the maximum for certain tasks reached up to 3 minutes and 19 seconds, suggesting specific scenarios where performance could be optimized. With over 65,000 requests processed and only a 0.11% failure rate, the system proved its robustness. However, the longer iteration times, averaging 36.43 seconds, and in peak 3m58s pointed to potential areas for improvement to enhance the system’s efficiency and user experience during peak loads.

Figure 16 K6 spike test results output



```
     ✗ is status 200 for IpLocation data
      ↳  99% — ✓ 10890 / ✗ 7
     ✗ is status 200 for whois data
      ↳  99% — ✓ 10887 / ✗ 10
     ✗ is status 200 for color data
      ↳  99% — ✓ 10879 / ✗ 18
     ✗ is status 200 classification data
      ↳  99% — ✓ 10888 / ✗ 8
     ✗ is status 200 for login data
      ↳  99% — ✓ 10871 / ✗ 21
     ✗ is status 200 for signup data
      ↳  99% — ✓ 10860 / ✗ 11

     checks.........................: 99.88% ✓ 65275      ✗ 75
     data_received..................: 750 MB 940 kB/s
     data_sent......................: 11 MB  14 kB/s
     http_req_blocked...............: avg=1.31ms   min=0s     med=295µs   max=3.51s  p(90)=1.01ms p(95)=1.72ms
   ✓ http_req_connecting............: avg=1.26ms   min=0s     med=249µs   max=3.51s  p(90)=948µs  p(95)=1.63ms
   ✗ http_req_duration..............: avg=5.41s    min=0s     med=4.59s   max=3m19s  p(90)=9.55s  p(95)=13.37s
       { expected_response:true }...: avg=5.34s    min=1s     med=4.59s   max=3m15s  p(90)=9.52s  p(95)=13.25s
   ✓ http_req_failed................: 0.11%  ✓ 75         ✗ 65275
   ✓ http_req_receiving.............: avg=465.03µs min=0s     med=73µs    max=1.05s  p(90)=569µs  p(95)=1.18ms
     http_req_sending...............: avg=43.03µs  min=0s     med=24µs    max=50.3ms p(90)=58µs   p(95)=78µs
     http_req_tls_handshaking.......: avg=0s       min=0s     med=0s      max=0s     p(90)=0s     p(95)=0s
     http_req_waiting...............: avg=5.41s    min=0s     med=4.59s   max=3m19s  p(90)=9.55s  p(95)=13.37s
     http_reqs......................: 65350  81.854019/s
     iteration_duration.............: avg=36.43s   min=7.49s med=34.71s max=3m58s  p(90)=1m8s   p(95)=1m20s
     iterations.....................: 10849  13.588894/s
     vus............................: 2      min=2        max=1000
     vus_max........................: 1000   min=1000     max=1000


running (13m18.4s), 0000/1000 VUs, 10849 complete and 48 interrupted iterations
default ✓ [======================================] 0000/1000 VUs  13m8s
```

Figure 21 highlights the K6 spike test results, which are explained at the beginning of this chapter.

Figure 17 Performance metrics correlation during spike testing

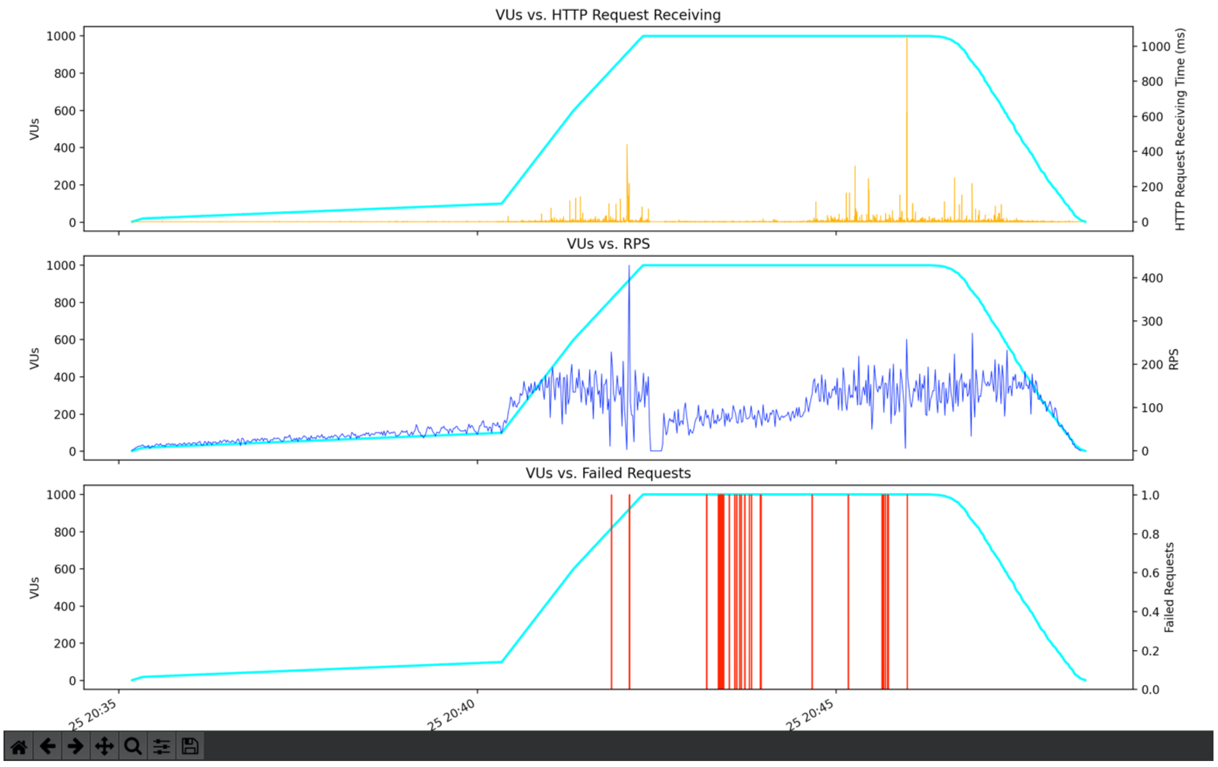
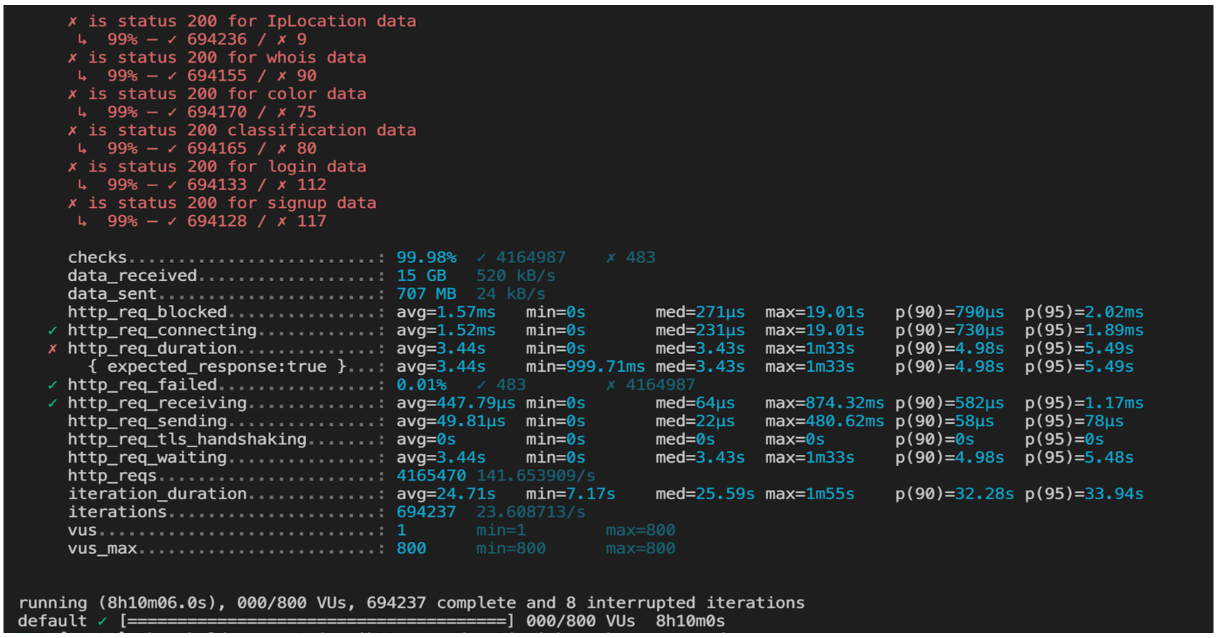


Figure 22 presents the system's behavior under a spike test. The top graph shows HTTP receiving time and the time it takes for the system to start receiving a response after a request is made, which could mean users experience delays when the system is busiest. The middle graph shows the number of requests the system can handle each second; it's pretty up and down, which suggests that the system's ability to handle actions varies under pressure. Lastly, the bottom graph indicates that as more users hit the system, there are moments when requests fail to go through, particularly when the system is at its busiest. These red lines show exactly when and how often requests aren't successful, giving a clear picture of when the system struggles the most.

7.4 Soak Test Results

The soak test results indicated that the system is stable, successfully handling 99.98% of HTTP requests. The HTTP request durations showed variation, with the minimum being 0 seconds and the maximum being 1m33s, pointing to potential delays under high-load conditions. The average HTTP request receiving time was efficient at 447 microseconds, but with some responses taking as long as 874 milliseconds, indicating occasional performance lags. Iteration durations also varied, averaging 23.608713s; with some other requests, it took longer during peak load and reached a maximum time of 1m55s. indicated that the system still needs to be improved. Overall, the system seems to handle the load well for a long period of time, but there could be potential optimizations to improve response times and reduce the range of request durations for a more consistent user experience.

Figure 18 K6 soak test results output



```
     ✗ is status 200 for IpLocation data
      ↳  99% — ✓ 694236 / ✗ 9
     ✗ is status 200 for whois data
      ↳  99% — ✓ 694155 / ✗ 90
     ✗ is status 200 for color data
      ↳  99% — ✓ 694170 / ✗ 75
     ✗ is status 200 classification data
      ↳  99% — ✓ 694165 / ✗ 80
     ✗ is status 200 for login data
      ↳  99% — ✓ 694133 / ✗ 112
     ✗ is status 200 for signup data
      ↳  99% — ✓ 694128 / ✗ 117

     checks.........................: 99.98% ✓ 4164987     ✗ 483
     data_received..................: 15 GB  520 kB/s
     data_sent......................: 707 MB 24 kB/s
     http_req_blocked...............: avg=1.57ms   min=0s       med=271µs  max=19.01s   p(90)=790µs  p(95)=2.02ms
   ✓ http_req_connecting............: avg=1.52ms   min=0s       med=231µs  max=19.01s   p(90)=730µs  p(95)=1.89ms
   ✗ http_req_duration..............: avg=3.44s    min=0s       med=3.43s  max=1m33s    p(90)=4.98s  p(95)=5.49s
       { expected_response:true }...: avg=3.44s    min=999.71ms med=3.43s  max=1m33s    p(90)=4.98s  p(95)=5.49s
   ✓ http_req_failed................: 0.01%  ✓ 483         ✗ 4164987
   ✓ http_req_receiving.............: avg=447.79µs min=0s       med=64µs   max=874.32ms p(90)=582µs  p(95)=1.17ms
     http_req_sending...............: avg=49.81µs  min=0s       med=22µs   max=480.62ms p(90)=58µs   p(95)=78µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s     max=0s       p(90)=0s     p(95)=0s
     http_req_waiting...............: avg=3.44s    min=0s       med=3.43s  max=1m33s    p(90)=4.98s  p(95)=5.48s
     http_reqs......................: 4165470 141.653909/s
     iteration_duration.............: avg=24.71s   min=7.17s    med=25.59s max=1m55s    p(90)=32.28s p(95)=33.94s
     iterations.....................: 694237 23.608713/s
     vus............................: 1      min=1         max=800
     vus_max........................: 800    min=800       max=800


running (8h10m06.0s), 000/800 VUs, 694237 complete and 8 interrupted iterations
default ✓ [======================================] 000/800 VUs  8h10m0s
```

Figure 23 highlights the results output of the K6 soak test that is explained at the beginning of this chapter.

Figure 19 Performance metrics correlation during soak testing

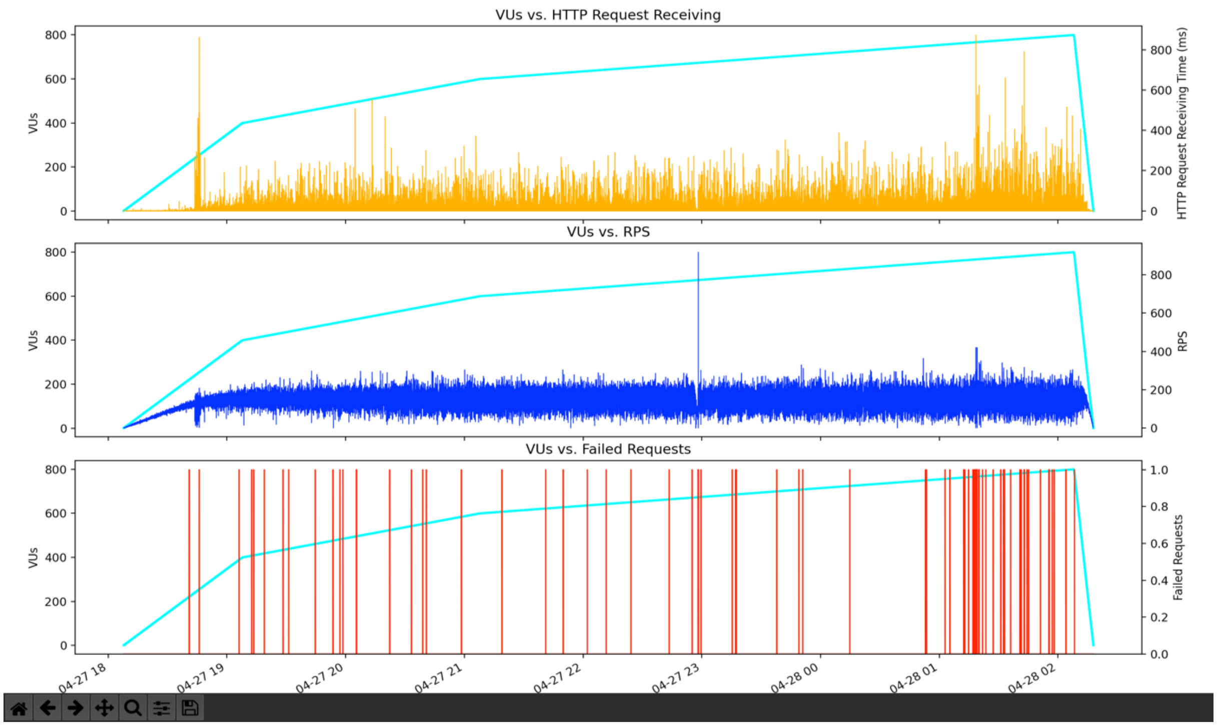


Figure 24 presents the system's behavior under a soak test. The top graph shows HTTP receiving time and the time it takes for the system to start receiving a response after a request is made. When more people use the system for a long time, it sometimes takes longer to respond. The middle graph shows the number of requests the system can handle each second; as shown in the figure, the request per second is mostly stable even when there are more VUs, indicating that the system is able to handle many RPS. the bottom graph indicates that as more users hit the system, there are moments when requests fail to go through, particularly when the system is at its busiest. These red lines show exactly when and how often requests aren't successful, giving a clear picture of when the system struggles the most. But even with many users and requests per second, the system keeps working without needing to restart the server.

7.5 Overall Results

Kalibro.io services performed well in all tests; the HTTP failed request ratio was nearly 0 in all cases, which indicates the system is stable. The system was able to handle thousands of requests and hundreds of users and didn’t crash or need to restart the server. However, the response time and iteration duration took a bit longer time when there were many users and requests, indicating that the system backend still needs some improvement, for instance, making the database connection handle more users if needed when there is a load on the system by creating a pool connection that has ready connections to be used when needed.

All the tests were conducted in a local environment; the results might have been different if the tests had been conducted in a cloud environment since the cloud has more resources, like CPUs, how many instances you are running, and more. This could make the system handle more requests with less response time. For that reason, the tests were conducted in a local environment to identify any service and code structure issues.

7.6 Conclusion

Performance testing significantly enhances the development outcomes of software by ensuring that applications perform effectively under various conditions. This type of testing is important for identifying performance-related issues in software design and architecture, ultimately contributing to more stable and robust software solutions.

One of performance testing's core benefits is its ability to improve user engagement and satisfaction. By ensuring that software applications are responsive and perform efficiently, performance testing helps maintain a seamless user experience, which is crucial for retaining users.

This thesis aimed to evaluate Kalibro.io's services by conducting performance testing. The K6 tool had a big impact on evaluating the system's performance and user capacity. The tool is designed to perform four major types of performance testing: load, spike, stress, and soak. Those four types have already been explained in the previous chapters. The tool was beneficial in simulating different scenarios of a load of users that hammered the services with requests. Using the tool, the author was able to determine the system's normal load of users, system performance when there is an increase in users, a sudden big load of users, and the system performance under a different load over a long period of time. The tool was easy to understand and use since the K6 documentation is clear and straightforward. K6 was a sufficient tool for conducting performance testing for many reasons. For instance, JavaScript makes it easier to find a lot of resources, and the tool is flexible, easy to modify, and easy to simulate thousands of users.

The primary goal of the thesis has been achieved by conducting performance testing on all Kalibro.io ready services. The system performance under different load conditions, the system user load capacity, and the response time for each HTTP request were evaluated. The test results have been captured, documented, and sent to the backend team to fix the issues that occurred in the tests.

8 Future Recommendations

Kalibro.io services performed well in all four major types of performance testing. However, there is room for improvement in service performance, such as reducing response time under extreme user load, improving the HTTP failed ratio, and more. The Kalibro.io backend team could follow the following steps to improve the services:

- Ensuring the code is optimized for performance. This can include optimizing queries, reducing complexity, and avoiding unnecessary processing.

- Using a load balancer to distribute incoming traffic evenly across multiple servers. This not only helps in handling more requests but also provides high availability.

- Asynchronous processing can be used for tasks that do not need immediate processing, freeing up resources to handle more incoming requests.

- Tunning the database settings for handling connections, such as adjusting the maximum number of concurrent connections.

- Regularly monitor system performance and resource utilization to proactively address bottlenecks and optimize resources.

In addition to the above recommendations, another performance test should be conducted when the tool is more stable and ready for production. The upcoming test should be conducted in a cloud environment Since it has more resources to determine its real load capacity and ensure its reliability and scalability.

9 Summary

I answered the research questions by implementing performance testing using K6, which allowed me to directly assess how well our systems and processes worked under load. This method was crucial in providing clear answers to the questions posed by our research. The use of K6, a tool new to me, introduced me to advanced testing methodologies that were not covered in my academic training. Learning to use K6 was both challenging and enriching, enhancing my understanding of practical testing environments and the specific demands they entail.

Through this experience, I learned a lot about the challenges of testing applications in the real world. Using K6 showed me how important it is to keep systems running smoothly to make sure they meet users' needs. This work has helped me understand how crucial performance optimization is, and I saw firsthand how it affects both the functionality of the system and the satisfaction of the user. This project not only improved my technical skills but also gave me a deeper insight into everyday operations that keep IT systems effective. I will use what I've learned here in my future projects, especially when it comes to performance testing.

References

Alsmadi, I. (2012). *Advanced Automated Software Testing* (p. 1). IGI Global. https://www.google.fi/books/edition/Advanced_Automated_Software_Testing_Fram/0G0mt4qUaGoC?hl=fi&gbpv=1&dq=automated+test+book&printsec=frontcover

Amazon. (n.d.). *What is Debugging?* Amazon.com. https://aws.amazon.com/what-is/debugging/. Accessed 3 May 2024.

Bierig, R., Brown, S., Galván, E., & Timoney, J. (2022). Essentials of Software Testing (1st ed., p. 3). TJ Books. https://www.google.fi/books/edition/Essentials_of_Software_Testing/zog7EAAAQBAJ?hl=fi&gbpv=1&dq=software+testing+google+book&printsec=frontcover

Boydaş, S. (Sep 19, 2023). *Fundamentals of Software Testing*. Medium. https://medium.com/huawei-developers/fundamentals-of-software-testing-e1841db5f43b

Chandrasekara, C., & Herath, P. (2019). *Software testing types* (p. 2). https://www.google.fi/books/edition/Hands_On_Functional_Test_Automation/5weZDwAAQBAJ?hl=fi&gbpv=1&dq=Functional+Testing+books&printsec=frontcover

Geeksforgeeks. (2022, November 17). *History of Software Testing*. Geeksforgeeks. https://www.geeksforgeeks.org/history-of-software-testing/

Geeksforgeeks (2023, May 24). *What are Performance Testing Tools?* https://www.geeksforgeeks.org/software-testing-performace-testing-tools/

Geeksforgeeks (26 Mar 2024). *Automated testing*. Geeksforgeeks. https://www.geeksforgeeks.org/automation-testing-software-testing/

Grafana (n.d.). *Grafana k6 documentation*. Grafana. https://grafana.com/docs/k6/latest/. Accessed 5 May 2024.

Gregorio, G. (2023, July 27). *A Deep into Performance Testing Strategies with K6*. Beon. https://blog.beon.tech/a-deep-into-performance-testing-strategies-with-k6/

Javatpoint (n.d.). *Manual testing*. Javatpoint. https://www.javatpoint.com/manual-testing. Accessed 1 May 2024.

K6 (2023). *The best developer experience for load testing*. K6. https://k6.io. Accessed 10 May 2024.

K6 (n.d.). *Installation*. K6. https://k6.io/docs/get-started/installation/. Accessed 10 May 2024.

Lewis, W. E., Dobbs, D., & Veerapillai, G. (2009). *Software Testing and Continuous Quality Improvement* (3rd ed., p. 3). Taylor && Francis Group. https://www.google.fi/books/edition/Software_Testing_and_Continuous_Quality/fgaBDd0TfT8C?hl=fi&gbpv=1&dq=software%2Btesting%2Bhistory&printsec=frontcover

Microsoft (n.d.). *Documentation for Visual Studio Code*. Visualstudio. https://code.visualstudio.com/docs. Accessed 1 May 2024.

Nayyar, D. A. *Instant Approach to Software Testing*. https://www.google.fi/books/edition/Instant_Approach_to_Software_Testing/TCy4DwAAQBAJ?hl=fi&gbpv=1&dq=software+testing+types&printsec=frontcover

P, R. (2023). *K6 Performance Testing Beginner Advanced*. https://www.amazon.com/K6-Performance-Testing-Beginner-Advanced-ebook/dp/B0CJ3VW9L3?asin=B0CJ3VW9L3&revisionId=6b89fb83&format=1&depth=1

Prasad, D. K. (2004). *Software_Testing_Tools* (p. 98). Dream Tech. https://www.google.fi/books/edition/Software_Testing_Tools_Covering_WinRunne/DuinhInx0moC?hl=fi&gbpv=1&dq=Software+Testing+Tools&printsec=frontcover

Rahman, K. (2019). *Introduction to automated test*. Google Books. https://www.google.fi/books/edition/Science_of_Selenium/-ofCDwAAQBAJ?hl=fi&gbpv=1&dq=automated+test+selenium+book&printsec=frontcover

Software Testing Types | Test Automation Resources. https://testautomationresources.com/software-testing-basics/software-testing-types/

T., Staff (2024, March 21). *What is performance testing?* Tricentis. https://www.tricentis.com/learn/performance-testing. Accessed 9 May 2024.

T., Staff (06 Aug, 2021). *Types of software testing tools.*Tricentis. https://www.tricentis.com/learn/software-testing-tools. Accessed 8 May 2024.

Tran, T. (2022, September 6). *Manual Software Testing: A Comprehensive Guide for Beginners*. Orientsoftware. https://www.orientsoftware.com/blog/manual-software-testing/ .

Appendix 1: Material Management Plan

All test cases created for this thesis are initially developed and tested locally on the author's machine. Upon completion and local validation, these test cases are stored on Kalibro.io GitHub, in a dedicated directory named "Performance-testing." This approach ensures centralized and secure storage of all digital test materials. Each test case is crafted following specific guidelines that adhere to the company’s standards for performance testing, ensuring consistency and reliability. Results from each test case are thoroughly documented and subsequently shared on the company’s Agile platform, Trello, facilitating immediate feedback and collaborative review.

Access to the GitHub repository and the "Performance-testing" directory is strictly controlled, with permissions granted only to authorized personnel. Regular backups of the GitHub repository are scheduled to prevent data loss, including all test cases and their associated results. Test cases are periodically reviewed and updated to reflect new insights, technologies, and methodologies, maintaining their effectiveness. To minimize risk and maintain data integrity, no test materials or results are stored locally on the author's machine beyond the initial development phase; once test cases are validated locally, they are uploaded to GitHub, and all local copies are removed.