Bachelor's thesis

Information and Communications Technology

2022

Giorgio Diprima

Comparison of tools and metrics for non-functional testing of web pages

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2022 | Number of pages: 58

Giorgio Diprima

Comparison of tools and metrics for non-functional testing of web pages

This thesis aims at helping every person interested in selecting the proper non-functional testing tool, especially Wapice, the company where the author´s internship was carried out. The purpose was to compare different commercial solutions for non-functional testing. In order to understand the results offered by the pieces of software under test (Google Lighthouse, K6, and LoadRunner), the concept and metrics of non-functional testing are first defined. The three tools were tested on a web app created with the same technologies used during the internship. The final purpose of this study was to be able to apply the conclusion of the thesis to the project developed at Wapice during the internship.

The results of the tests show how Google Lighthouse can be a valuable tool when integrated into a pipeline to avoid regression; the potentialities of K6, a modern tool focused on performance, that allows executing a complete performance test for free; but also, the complexity of setting up and getting started with LoadRunner.

Keywords:

Testing, Non-Functional testing, Testing tools, ISO/IEC 25010, K6, LoadRunner, Google Lighthouse, Google PageSpeed, Performance testing

Contents

Equations

Figures

Pictures

Tables

List of abbreviations

CD	Continuous Development
CI	Continuous Integration
DOM	Document Object Model
ENISA	European Network and Information Security Agency
OS	Operating System
PR	Pull Request
QA	Quality Assurance
QE	Quality Engineer
SAAS	Software as a Service
SDLC	Software Development Life Cycle
UI	User Interface

1 Introduction

Nowadays, software is part of our everyday life, most of the services of the third sector are offered online through a web app. Sometimes there are bugs in the software that we use, and even more often, some functionalities are not working properly according to the users' expectations. Bugs in software can be quite pretty dangerous and can cause severe consequences, even death [1]. Poor performances are less impactful but can still cause an important substantial loss in the revenue of a company's revenue: a report from Akamai Technologies, Inc. shows that a 100 ms delay can hurt conversion rates by 7% in e-commerce. This happens because when a delay of 100 ms is added to an e-commerce, 7% of the visitors leave the website before finalising a purchase [2]. The same effect can be observed positively: when a website is well built and works as expected from the end user side, the number of visitors can increase, leading to a boost in revenue. The case of Swappie highlights this after improving the performance of the mobile version of the app, they saw the mobile revenue upsurging by 42% [3].

Testing is the key to reducing the number of bugs and understanding if there is a need to increase the software's performance. This can be achieved through two different types of testing [4]:

- Functional testing
- Non-functional testing

The main difference between these two kinds of tests is what they test.

Functional testing screens the functionalities of an app. Non-functional testing tests the performances of these functions and ensures that the application is usable and reliable.

In different Software Development Life Cycle (SDLC), distinct testing techniques are used. Continuous Integration / Continuous Development (CI / CD) is a software development practice that gained a lot of popularity in the last few

years. The key to its success is the use of automation in different phases, including testing.

So, when working with CI, selecting the proper automated testing tool is crucial since the correct selection of automation testing tools can help increase accuracy and save resources compared to manual testing.

Although Quality Engineer (QE) teams often claim that the team needs more people, most of the time, teams need the ability to do more and not necessarily have more team members. However, teams need better skills and tools to improve efficiency and productivity [5].

Choosing the most suitable testing tools is a defining factor in having successful and efficient software testing [6].

According to the data collected in the last world quality report, it can be seen that companies are still struggling to move to automated tests. Even if the benefits of testing automation are clear to 50% of the people interviewed, just 20% of the tests have been automated [5].

This thesis hypothesis that choosing the best tool can be a tedious task and, thus, an obstacle to start using automated testing. During his internship, the author observed that automation in functional testing is already in use and integrated into the SDLC. This happens because the goal of functional testing is clearer to understand, providing inputs and checking that the output matches the testers' expectations. On the contrary, non-functional testing requires an understanding of the metrics used to perform the evaluation, which is more complex to implement and less used.

This thesis aims to compare the features and use cases of three different non-functional testing tools. Furthermore, it studies the developer's experience using them by analysing the learning curve. Those findings/results will help developers and project managers at the moment they need to decide which type of test method they want to use, as well as which software best matches their requirements.

This thesis is structured as follows: Chapter 1 introduces the background to this thesis, and the objectives of the work carried out.

In Chapter 2, the reader can find a theoretical introduction to testing. Different types of tests are categorised into two categories: functional and non-functional tests. For each type of test, the objectives are going to be introduced. Chapter 3 provides an overview of some of the most important metrics to consider when performing non-functional tests. Chapter 4 introduces how the testing infrastructure has been realised and how the tools are chosen. Chapter 5 analyses the architecture of five testing tools; based on the findings, three tools will be tested. In Chapter 7, the results of the testing are collected. The last chapter, Chapter 8, summarises the work carried out in this thesis and reflects on how this work can be useful for companies to choose a non-functional testing tool.

2 Testing concepts

Testing can be defined as a set of activities to identify software failures and evaluate a system's quality. We have a failure in our system when we have defects. Since defects are introduced when a programmer writes the code, we can be sure that faults are present in every software.

Testing enables the identification of defects but does not show that the flaws are not present. Demonstrating the absence of error in complex systems is impossible because it is impossible to recreate all the possible inputs in all the possible contexts (hardware and software).

Testing enables the reduction of risk in the tested functionality instead. But since exhaustive testing in a real scenario is also complex and requires many resources, developers must be able to choose:

- Which test to design and execute,
- In which way the tests need to be performed.

There are two ways of testing: manual testing and automatic testing. The decision of which way the test needs to be performed also depends on the chosen development model. We can say that the decision of the development model is tied to the decision of the testing method. However, testing is always present in the SDLC, whichever development model is chosen.

To understand the importance of automatic software testing, it helps to consider the process that is adopted before publishing changes when manual testing is in use. The QE team is responsible for manually executing a list of tests provided by the developer that assert a specific feature works as expected; then, a report is returned to the developer to solve possible problems. The process is slow, expensive and error-prone. Instead, when automatic software for testing is used, it is the developer's responsibility to execute specific tests.

The developer is also responsible for choosing which tests to execute. This decision should be based on which software characteristic needs to be tested.

International standard ISO/IEC 25010 proposes a model to determine which quality characteristics can be considered when evaluating a software system's properties [7].

The model is organised into eight major characteristics, as shown in Figure 1, each has some sub-characteristics. Usually, those characteristics are divided into:

- Functional characteristics describe the functionalities provided by the system: what the system is supposed to **do**.
- Non-functional characteristics define how a system is supposed to **be**.

Figure 1. Software product quality, ISO/IEC 25010.

2.1 Functional testing

Functional tests focus on the services provided by the software and the requirements covered. Functional testing is a way of verifying that software has all the desired functionality defined in the requirements. Different types of tests can and should be performed. Usually, unit tests, integration tests and system tests are performed. This is also the order in which they are typically performed during the SDLC.

2.1.1 Unit testing

The system is divided into units (functions, procedures, methods) when unit tests are performed. The code is checked to verify that a unit works as expected

or that a particular defect is not present. The developer writes this type of test at the same time the software is developed, and for this reason, it allows the identification of problems early in the SDLC. Developers can write tests on the executable language, and the test case can be executed automatically.

Unit test tools are the most used tools to automate tests [8].

White box testing

Unit testing is considered white box testing because to write this type of test, the developer needs to be aware of the internal structure of the tested application.

We can talk about white box testing when the software is considered a set of lines of code, instructions, and components. The developer assumes that the software should work correctly once all its structural elements have been verified and validated.

2.1.2 Integration testing

Once the unit test results satisfy the expectation, the next step is combing the tested modules and testing the interface between the single units. This process is called integration testing.

Those tests are typically written as soon as the modules are tested and available to ensure that the integrated components work properly and detect possible errors related to the interface between modules/units.

Performing this type of test is more complex than performing unit tests because we need to ensure that the environment is working correctly. For example, a connection to the database or different platforms or environments may be necessary.

2.1.3 System testing

System testing is performed after Integration tests. Once all the modules are developed and tested, and the interfaces between them are also tested, the last step is to test the system as a whole. The goal is to evaluate end-to-end system specifications. This type of test includes verifying how the software is integrated with external peripherals (software and hardware).

There are different techniques to perform this kind of test. Here, we will overview two: black box and smoke testing.

Black box testing

This type of test focuses on the requirements and specifications of the software. This test aims to test all the requirements and specifications; once everything has been verified, the software is considered correct. This type of test is called “black box” because the software is regarded as a black box, and the focus is on the interfaces as described in the documentation, without taking into consideration the internal structure (code architecture)

Smoke test

We can consider smoke testing as a first system test generally performed by the developer and not by the QA team. It is used to check if the build is stable or not. It consists of a minimal series of tests to verify that the most critical features are working and there are no showstoppers. This type of test aims to confirm that there are no significant issues and test if the system is stable.

2.2 Non Functional testing

Contrary to functional testing, which focuses on the services provided by the software, non-functional testing focuses on how the services are provided. Non-functional aspects are checked by producing and collecting measurements and metrics. Those can be used for internal research to help developers find when and where it is necessary to increase the system's performance, usability, and reliability.

Taking ISO/IEC 25010 as a reference, we can say that the non-functional properties that need to be taken into account when testing are the following [9]:

- Reliability: how the product performs under certain conditions for a specific period. When the reliability needs to be evaluated, several properties must be considered, in Table 1, some popular ones [9].

  Table 1. Subcategories of reliability according to ISO/IEC 25010.

Category	Description
Availability	Measures the rate at which components are accessible when required.
Recoverability	Measures the rate at which a product can recover data in case of failure.
Maturity	Ability to avoid errors in case of an exception is thrown.

- Performance efficiently: in this case, we are interested in the relationship between the resources used and the application's performance. To understand our application's performance, we need to consider the response time and try to correlate it with the load generated by request. Table 2 shows the subcategories that can be evaluated to understand the application's performance.

Table 2. Subcategories of performance according to ISO/IEC 25010.

Category	Description
Time-behavior	Measures the rate at which the response time meets the requirements.
Resource utilisation	Measures the rate at which the amount of resources used meets the requirements.
Capacity	Measures the rate at which the maximum limit of the product meets the requirements.

- Usability describes how a product can be used by a specific user in a particular environment to reach specific goals.

Table 3. Subcategories of usability according to ISO/IEC 25010.

Category	Description
User interface aesthetics	Measures if the user interface is satisfying to interact with from the user's point of view
Accessibility	Measures the rate at which different users with different capabilities, can achieve a specific goal.
Learnability	Measures the rate at which a product enables users how to learn the system.

- Security is the degree to which a system can protect data so that users can access the information they have access.

Table 4. Subcategories of security according to ISO/IEC 25010.

Category	Description
Confidentiality	Measures if the data are accessible just from the authorised users.
Integrity	Measures if the data are modifiable just from authorised users
Non-repudiation	Measures if the events are logged in such a way that it is possible to prove that they happened, and they cannot be repudiated later.

2.2.1 Performance testing

Performance testing is the process of verifying how the stability and response time of the application change when the workload changes. This type of test aims to identify and remove possible performance bottlenecks in the application. The qualities that are evaluated are:

- Response time: determines whether the application responds quickly.
- Scalability; determines the maximum user load that the software application can handle.
- Stability; determines the strength of the application under varying loads.

It is essential to perform performance tests because before an application is released to the public, it is important to be aware of the following:

- If the software is working as expected, also when several users are using the app simultaneously;
- if the performance is consistent across different OSs.

A Dun & Bradstreet study found that 90% of the organisation have unexpectedly lost access to critical systems, and nearly a third deal with downtime issues

every month. Nearly 60% of Fortune 500 companies experience a minimum of 1.6 hours of downtime every week. And they also estimate their organisation's financial cost of downtime, which is between $250,000 and $1 million per hour [10].

Performance tests should be executed regularly, at each major deployment, possibly as part of the CI deployment process [11]. If we identify possible problems at the earlier stage, we can reduce the failure cost. By contrast, after-deployment failure can incur greater charges, as discussed in the previous example.

There are two main techniques for performing performance testing load and stress tests [12]. Both methods try to verify the performance of the application, but the first one tries to identify bottlenecks; instead, the second one wants to take the system to the breaking point and see how it reacts

Load testing

Checks the application’s ability to perform under anticipated user loads. The objective is to identify performance bottlenecks before the software application is deployed.
This type of test helps to answer these questions:

- Is the current infrastructure sufficient to run the application? Do we have enough resources to handle a peak of users?
- How many people can visit my website at once?
- Is the server fast enough to answer in such a way that provides a pleasant user experience?

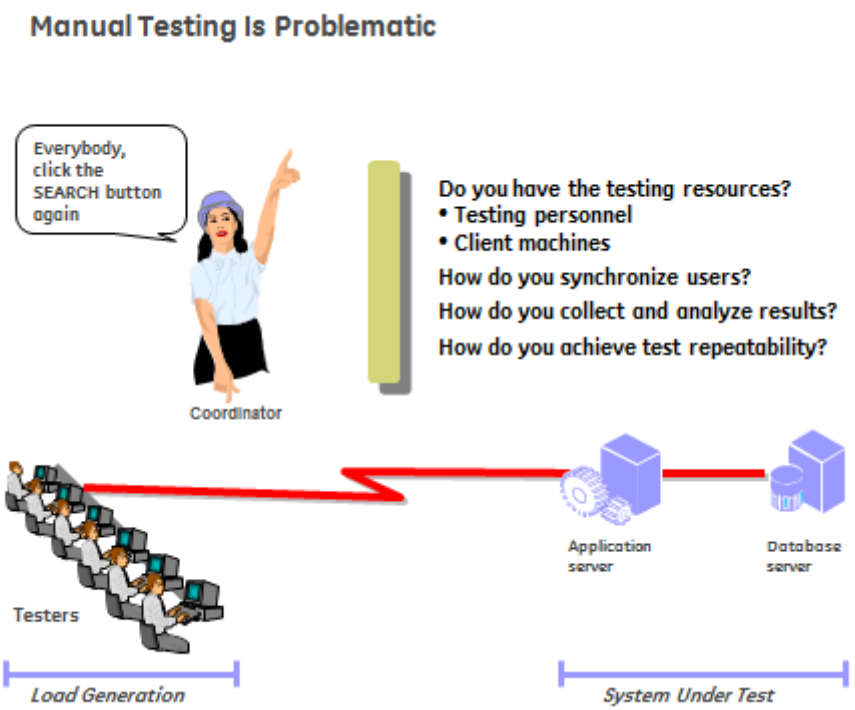
It is crucial to perform load testing because the system can perform completely differently for one user (functional test) than many (load test).
History is full of examples of popular platforms that suffered from downtimes because of higher traffic volumes than usual. One example is the toy store Toysrus.com, which decided to invest heavily in advertisement campaigns but

not in load tests. The platform could not handle the higher traffic, which generated a loss from the potential sales of toys and a loss because of the investment in the marketing campaign, without any positive effect [13]. In this case, the load test would have helped reduce system downtime risk and improve customer satisfaction.

When performing a successful load testing, the key is to replicate users' behaviour and environment (OS, resources, configurations) as best as possible and choose the best testing method and tool.

Even if theoretically possible, manual load testing is challenging to execute because of the need to simulate thousands of concurrent accesses. This condition requires different testers that are executing different workflows and the ability to coordinate the different testers [14].

Picture 1 [15] clearly describes some of the problems of manual testing; by contrast, an automated testing tool can simulate traffic via HTTP protocols rather than emulating the interaction with the end-user interface. To simulate the traffic is possible to record some action with the testing tool and generate a script to mimic the same activity. In this way, it is easier to perform **horizontal scaling** of the tests when an additional workload is needed. Another advantage of automated testing tools is the possibility of having repeatable results, unlike manual testing, which depends on the operator performing them. Additionally, for manual testing, it is difficult to provide a measurable level of stress on the application because of the impossibility of coordinating the work of different testers.



Picture 1. Perils of using manual testers for Performance testing.

Stress testing

Stress testing involves testing an application under extreme workloads to see how it handles high traffic or data processing. The objective is to identify the breaking point of an application in order to determine the **stability** and **reliability** of the software; the goals are:

- To identify the maximum workload that doesn't damage the application if applied for a short period.
- Ensure that the user sees a proper error message in case of an error.
- To be aware of what happens in case of extreme workload, then it is possible to try to minimise the effect of the possible system failure.
- To make sure that the system recovers after the failure: this property is called **recoverability** [16] [17]

Different types of tests can be used:

- Distributed stress testing involves running the test from different computers simultaneously. Cloud-based solutions are often used because they provide the possibility of generating traffic from different locations, generating a lot of traffic, to recreate the behavior of physical users and have a realistic scenario.
- Exploratory stress testing involves testing the system under abnormal conditions which will not likely come up in a real scenario. Some examples are adding a massive quantity of data to the database and many people trying to log in to the application.
- Systematic stress testing involves testing many services that run on the same server; it helps understand if one service stops working, the others are also influenced.

One of the most common problems that stress tests allows the identification of is **memory leaks**, this type of error happens when the memory is not correctly managed, e.g., the memory is no longer required but is not released. It is essential to select a tool that monitors the allocation and deallocation of resources. This type of test is performed by triggering several activities, with the intent of creating and removing a large amount of data from memory and taking the system to the **point of failure** to make sure the system can recover smoothly [17].

2.2.2 Usability testing

Usability testing is a testing method to measure how easy it is to use an application. It usually involves asking some user to perform a specific action and observing how the user tries to complete the task. This test helps identify User Interface (UI) problems and improves the software according to user preferences.
One approach to performing this type of test involves a facilitator and a participant [15]. The first one asks the second one to complete a task and observes and listens to the user's feedback. The facilitator must give instructions and ask questions to the participant without influencing their

behaviour. The facilitator's role can also be performed automatically by specific software.
Two types of data are collected: **qualitative data** on possible problems in the user experience, data collected during the interview, and **quantitative data on** how long it takes to complete a specific task.
Even if this kind of test looks challenging to perform because of the need for final users, some researchers show that five users are sufficient to find 75-99% of the usability problems [16]. Not all the researchers agree on the exact number; some more modern research shows that 16 users have the best benefit-cost ratio [17]. But the founding of the different studies agrees that a small testing group is sufficient to find most of the usability problems.

Another approach uses automatic software and takes into consideration just quantitative data. With this test, we can check if the user interface is stable, if the components have a standard behaviour and if the images are correctly described (alt attribute). These two approaches are not alternatives; they can be performed simultaneously because they will collect different information.

2.2.3 Security testing

Security testing is used to check if the software is vulnerable to cyber-attacks. This type of test aims to **identify** as many **vulnerabilities** in the software as possible. Each exposure must be **evaluated** by considering the **risk** the specific threat can cause to the business assets, and the likelihood of exploitation [21].

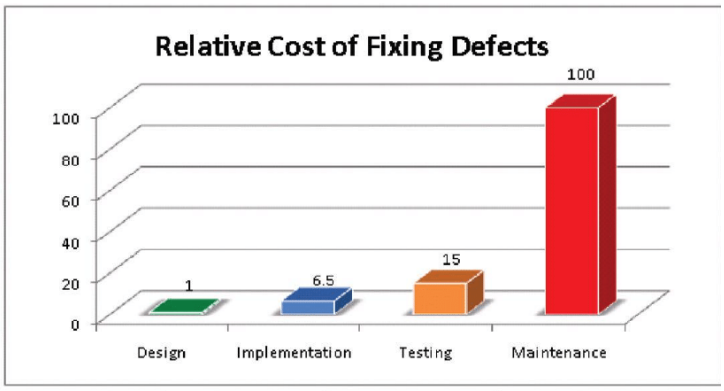


Figure 2. IBM System Science Institute Relative Cost of Fixing Defects.

Integrating security testing in the SDLC is very important because the concept is “earlier is better and cheaper”. Postponing testing security after the software implementation phase or even after deployments will be less cost-effective [20]. as shown in Figure 2.

The most common security testing technique is penetration testing.

Penetration Testing

To perform penetration testing, the developer must simulate a hacker attack to find security vulnerabilities and understand the risk of a hacker spoiling them.

As discussed in the introduction, nowadays, a lot of the services of the third sector are moving online, and web applications are always handling more sensitive data. The last report from ENISA shows that 20% of the companies are reporting DDoS attacks daily, with 52% of attacks more to web apps in 2019 compared to 2018, and most of those attacks, 84%, are due to security misconfigurations [18].

These data help us understand that attacks online are widespread, and testing is the only way to find out if there are vulnerabilities/ misconfigurations in the application.

The testing process is made up of three parts: first, the tester needs to find possible entry points; then, he needs to crack the testing environment; finally, the last step is filling a report to explain the process followed to breach the system and possibly give some instructions about how the risk can be mitigated. This type of test is usually unstructured, and different attack

techniques can be used. They can be categorised into three main categories, based on the knowledge that the tester has of the environment: black box when the auditor does not have any knowledge of the internal processes/functions used to transform the input into output; white box, in this type of test there is no separation between the tester and the code creator, and the tests are written based on the software code; grey box, this is a hybrid modality where the tester has a partial view of the internal structure. Since the nature of this test is unstructured, it is often more efficient to use manual testing techniques, but when the resources are limited, some automated tools can also be used. In this case, we can refer to the test as **vulnerability scanning**.

3 Testing metrics

The growth of the popularity of testing has required new models and metrics to evaluate projects. Metrics help organisations to gain information about the quality of the software and enable stakeholders to define quality goals.

We can define a metric as a unit of measurement that quantifies results, and in particular, we are interested in software metrics. Those are the metrics applied to services and products. Engineers use them to gain information about the processes and understand if it is necessary to improve the quality and productivity of the services offered by the software. Clear objectives and attributes need to be identified to perform an effective measurement, focusing on qualities that can be measured with a meaningful scale. The scale should then be refined and improved based on the data analysis [19] [20].

As technology evolves, metrics also need to keep up, some metrics used in the past are nowadays deprecated, and some new ones have been introduced. This is because not all the criteria that can be quantitatively measured are useful.

3.1 Response metrics

Those metrics give us information about the request sent from the user. They can measure the time from when the user sends the request to the server to when the process is over, typically measured in (KB/sec). Or information about the request's status, how many are successful, and how many caused errors [21].

3.1.1 Average load times:

This metric evaluates the average time taken to download and display the entire webpage. It is one of the most critical metrics to ensure product quality. When the average load time increases, the bounce rate also increases. As proof, we

can take a look at the study from Pingdom, which shows that when a website takes longer than five seconds to load, 38% of the site visitors will leave ("bounce") the website before having the requested service [22]. In internet marketing, this percentage is defined as the bounce rate. Table 5 shows how the load time affects the bounce rate

Table 5. How the load time influences the bounce rate.

Page Load Time (second)	**Bounce Rate (%)**
1	7
2	6
3	11
4	24
5	38
6	46
7	53
8	59
9	61
10	65
11	62
12	67
13	69
14	66
15	69
16	73

3.1.2 Response Time

The response time measures the time it takes for a user query to receive a response from the server. We can calculate the average or the peak response time. The average response time can give a general overview of the

application's performance. Instead, the peak response time helps individuate which components are not working correctly. The specific cause of the hung-up can be found. For example, if the peak response time is on a specific page or in a particular database call, the developer knows where to look for the performance problem.

Different testing software uses different metrics to measure the response time, often the Time To First Byte (TTFB). This metric measures the time that it takes for the browser to receive the first byte of data. But it is also possible to consider the time to receive the header or load the HTML [23].

3.1.3 Error Rate

It is a metric that tracks the percentage of requests that generate an error concerning the total number of requests. It is an excellent practice to keep this parameter under control because a peak in the error rate can lead to a central failure point in the near future.

3.2 User-centric performance metrics

Some quantitative characteristics influence the usability of the application. They try to measure. How long it takes to load the content of the page (perceived **load speed**); once the content is loaded, they measure how quickly it reacts to the user input. But also check the visual stability, if there are any unexpected page content shifts, or if the animations are fluid. Google identify several metrics to evaluate some user-centric characteristics. They also identify three of them as Core Web Vitals. Those are metrics that Google considers essential to examine to deliver a great user experience [26].

3.2.1 First Contentful Paint - FCP

This metric refers to the time it takes to render the first pixel after the user accesses the website. It is a metric that measures the perceived load speed and gives information about “how quickly a page can load and render all of its visual elements to the screen.” [24]. It is helpful to have a lower value as possible, between 0 – 1.8 seconds) to reassure the user that the page is loading [25].

3.2.2 Largest Contentful Paint - LCP

LCP is one of the three Core Web Vitals. This metric refers to the time it takes to render the largest image or text block in the viewport. It is used to check when the page's main content is loaded. Other metrics are also used to measure the same event, e.g., Speed Index (SI) and First Meaningful Paint (FMP), but W3C, after some research, have found out that those metrics are uselessly complex and not accurate. For this reason, they advise using LCP instead [27]. Google says that to provide the best user experience, this metric should have a value between 0 to 2.5 seconds.

3.2.3 Time to Interactive - TTI

TTI measures the time between FCP and when all the resources on the page are loaded. After this amount of time, the response time is more forecastable.

It is important to minimise the time between when the elements are visible and when they are interactive. This is because, in this time span, the user can think that the page is broken and leaves the web app prematurely. This is a performance problem that leads to problems in the user experience. For this reason, it is essential to have a minimal difference in the value of FCP and TTI.

3.2.4 First Input Delay - FID

FID is the second of three of the Core Web Vitals [26]. It is a load responsiveness metric. It gives an idea of how fast the application responds to the user inputs (click of a link, button). It does not measure the time necessary to process the event

In Figure 3, a timeline of a typical web page loading [28]:

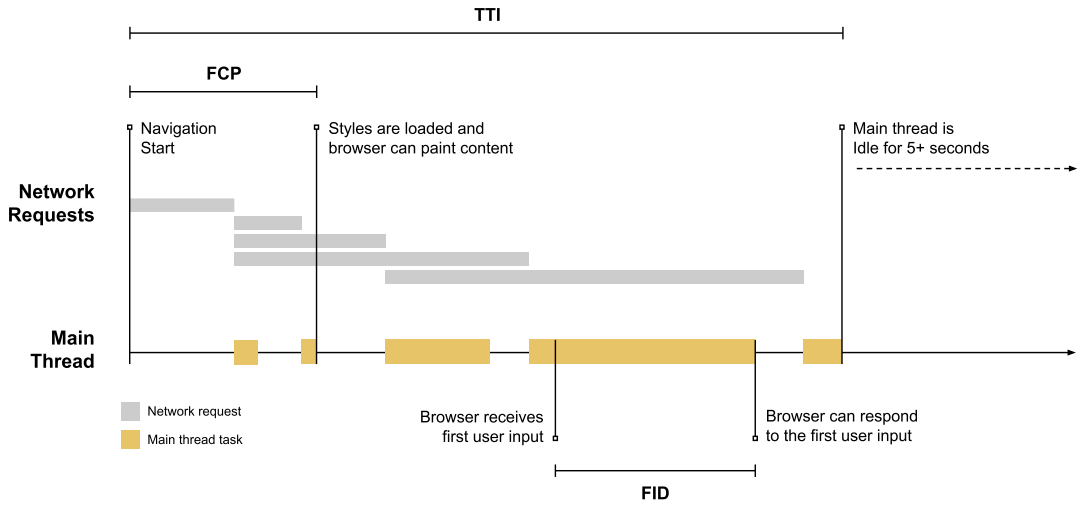


Figure 3. Timeline illustrating web page loading.

In the y-axis, we can find all the network requests and the status of the main thread. The network requests are used to retrieve CSS and JS files. Once the request is completed, the resource is retrieved and available, then the main thread can handle the file. FID can be noticed after FCP and before TTI is completed. In this scenario, the user can see and press the content display on the viewport, but the page is still loading some content. In Figure 3, the user sends an input during the execution of the longest job, for this reason, the time between the first user input and the time in which the browser is able to handle the request is quite long, and the user will experience slowness.

Figure 3 also explains why it is enough to consider the first input. The most significant delay is during the page loading. After this phase, the page is

responsive, has a minimal delay, and is no longer considered a user experience problem.

3.2.5 Cumulative Layout Shift - CLS

This is the last Core Web Vitals [26]. This metric helps evaluate visual stability. The user experiences a layout shift when there is an unexpected movement of the page, in other words, when a visible element changes its position from one render to the next. CLS usually happens because the Document Object Model (DOM) is built asynchronously, and components are added to the page above the visible content.

This metric intends to measure how often a CLS happens to real users [29].

The value for this metric can be calculated with Equation 1

$$layout\ shift\ score = impact\ fraction\ \times disatnce\ fraction$$

Equation 1. Layout shift score.

Where impact fraction is “The union of the visible areas of all unstable elements for the previous frame and the current frame “, and the distance fraction “is the greatest distance any unstable element has moved in the frame divided by the viewport's largest dimension” [29]. When this value is between 0 to 0.1, we are sure that the CLS does not influence the user experience.

4 Requirements

Two different types of requirements are going to be analysed. First, the requirements for the tools under test. In other words, what kind of features/services are needed for the testing tools to be taken into use. Then the requirements for the web app on which those testing tools will be tested.

4.1 Tools requirements

The initial idea was to compare a couple of products to Google Lighthouse since this was the only software used for testing non-functional properties in the project, I took part in during my internship. To choose which software testing tool to test, all the most popular commercial solutions for performance, load, and usability testing have been selected. Some of them are open source, e.g. JMeter, k6, but most are proprietary software, e.g. Tricentis NeoLoad, LoadRunner etc.

The key characteristics that will be analysed before using them are the following: popularity, the possibility of having a free trial, learning curve, quality, and amount of documentation available.

After checking several standing [30]–[32] to have an idea of the most popular software, the popularity was checked with google trends. Still, it was challenging to compare the results, so another tool similarweb.com has been used instead. Similarweb simplifies the comparison of different products by watching the traffic and engagement of the website.

Then when the choice was restricted to 5 tools, it became easier to analyse the characteristics of those and select 3 of them. The analysis follows in Architecture.

4.2 Testing website requirements

The testing website has been created with the same architecture and with the same technologies used for the website used during the internship. But it does not include the same authentication mechanism. In the testing website, the authentication has been developed from scratch; on the contrary, the original project uses Microsoft's Open Authentication. The architecture used is “Clean Architecture” [33]. Adopting this architecture allows a clear separation between the enterprise logic and types, domain level, business logic, and application level. The technologies used are ASP.NET core for the back end, React and MobX for the front end, and SignalR for real-time communication. The infrastructure where the web app is hosted was impossible to emulate because the hosting used in the platform developed during my internship is done in-house. On the contrary, the testing platform for the testing website will use Heroku.

5 Architecture

In this section, the characteristics of the five selected tools will be analysed, with the procedure described in the requirements.

5.1 Google Lighthouse

Google Lighthouse is a free tool that helps find a web app's slowness and usability. After running the tests, a number between 0 to 100 is returned. Values over 90 are considered good values. The performance's "grade" is given by taking into consideration: FCP, SI, LCP, TTI, TBT, and CLS [34]. Those metrics are not going to test the performance with a load or stress testing technique because just one user session is used

There are different ways to run Lighthouse:

- If Google Chrome is in use, then in the **DevTools,** there is a Lighthouse panel form where it is possible to run the tests. The advantages of this running mode are that it is not necessary to train the developers to use this software. It is possible to run a single test with a click, as everything is automatic, and the report is elementary to read
- Through the website **PageSpeed**, a tool built on top of Lighthouse, that can analyse even more data. Lighthouse uses just lab data. Instead, page speed considers both lab data and field data. The difference is that lab data are collected in a controlled environment with predefined settings and devices; instead, field data are collected from real users visiting the website.
- Node CLI, it is possible to install Lighthouse with npm and then run it from the **command line**.
- Node module. It is possible to write programs that use Lighthouse Thanks to this option, many projects took life. We will take a look at the only one officially maintained by Google: **Lighthouse CI**

Figure 4 [35] helps us understand how Google Lighthouse works. The operation of this software is based on the Driver modules, which include a set of APIs to interact with Chrome (Chrome DevTool Protocol). The Gatherers collect data from the page using the Drivers. That data is then processed by the Audits, which are tests for a specific feature/ metric. In the end, a report based on the results of the audits is generated.

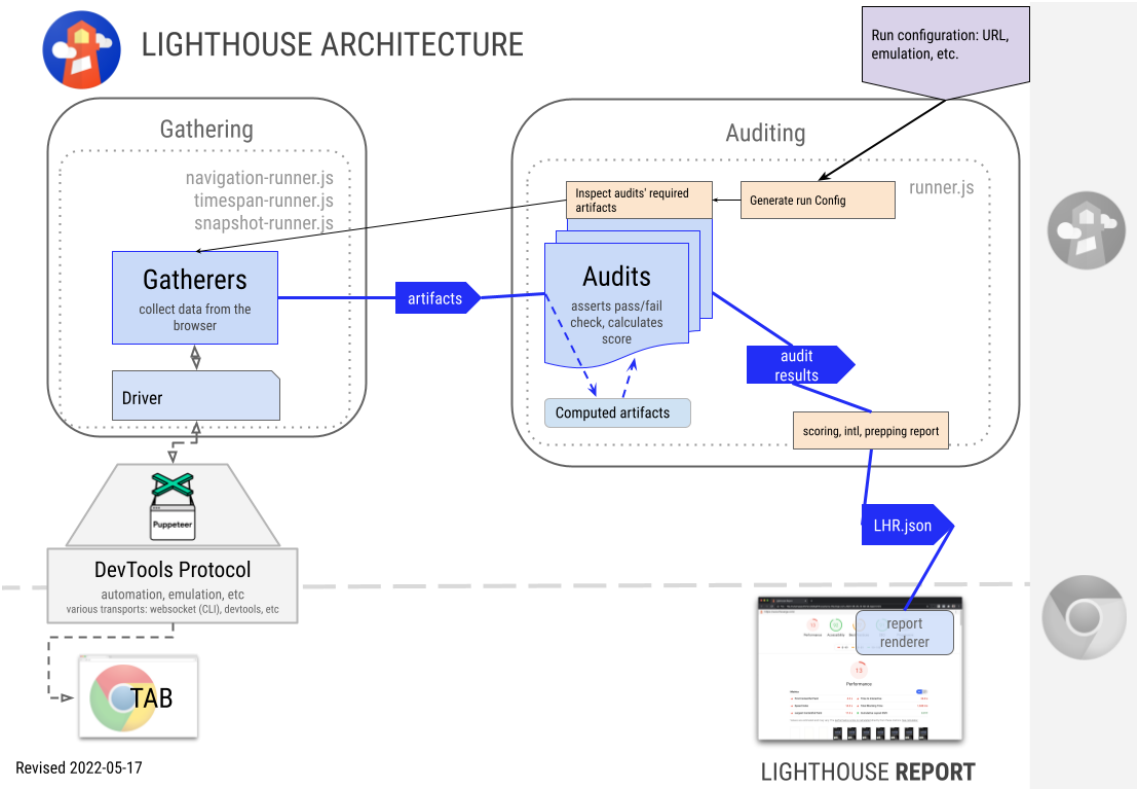


Figure 4. Google Lighthouse architecture.

5.2 LoadRunner

This tool is a pioneer, launched in 1999, and is also the market leader in performance testing tools [36].

We can look at the architecture in Figure 5 to understand what LoadRunner does. To perform a test, we need to start by recording the communication from the client to the server, and we can do it from the Virtual User Generator (VUGen). The

actions will be translated into a script, and the testing engineer needs to decide how many users will perform those tasks (more complex simulations are also possible); at this point, we are using the controller component. Then the script is executed by the agent machine, and the controller sets the number of devices required. This additional layer allows a more accurate simulation compared to only one machine performing all the tasks.

Once the task is completed, all the simulation error results and statistics are accessible thanks to the Analyze component.

LoadRunner is one of the most complete tools in the market, and often the results of the test are used to compare other tools

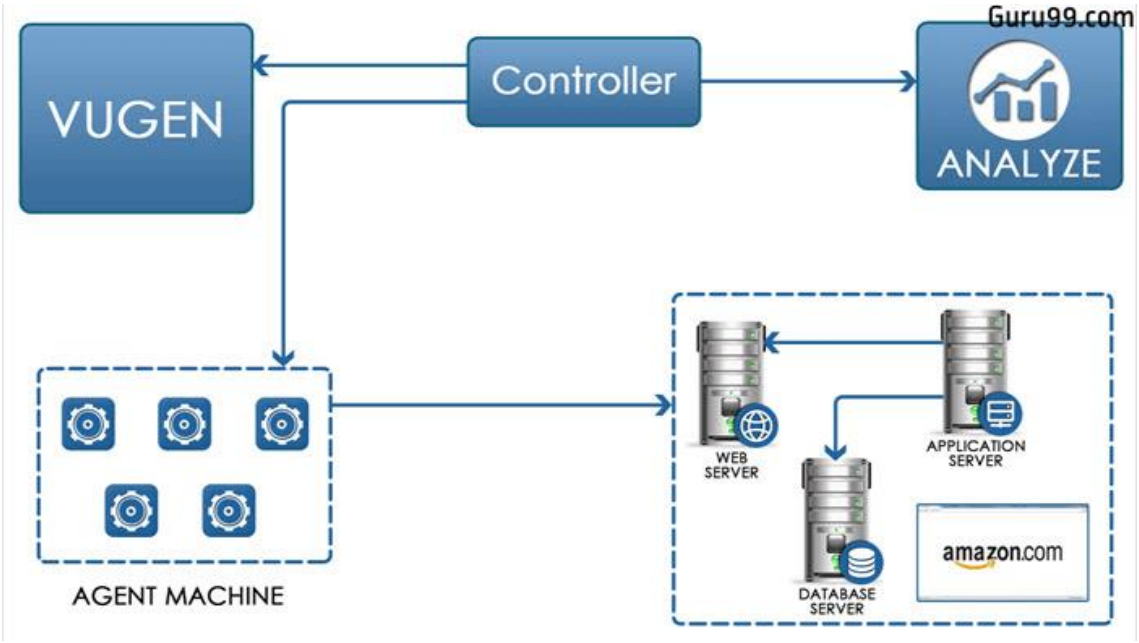


Figure 5. LoadRunner architecture.

5.3 Apache JMeter

JMeter is an open-source tool developed in Java by Apache Foundation. It was first realised in 1998 and became very popular because it provided the same services as other proprietary load-testing tools. It is a load-testing **protocol based**. It simulates the traffic at API or protocol level. This tool also provides a GUI that makes it easier to use for the user. It generates requests to a target

server, and once all the responses are collected, statistics are generated and returned to the user in different formats. The advantage of protocol-based testing is that tests are faster and require fewer resources. Still, since they don't execute JavaScript code, it can be challenging to test Single Page Applications where JS has an important role. Another advantage of this software is its extendibility. JMeter also supports plugins, and because of its longevity on the market, many plugins are available. The architecture of this tool follows the pattern of master and slave. The system that is running JMeter GUI is the master and gives orders to the slaves that are the servers running the JMeter server. Each slave is going to receive a control instruction from the master. Where the master tells the slave, which request needs to be sent, to which target and how many times it needs to be repeated. Figure 6 depicts the functional concept of JMeter [37].

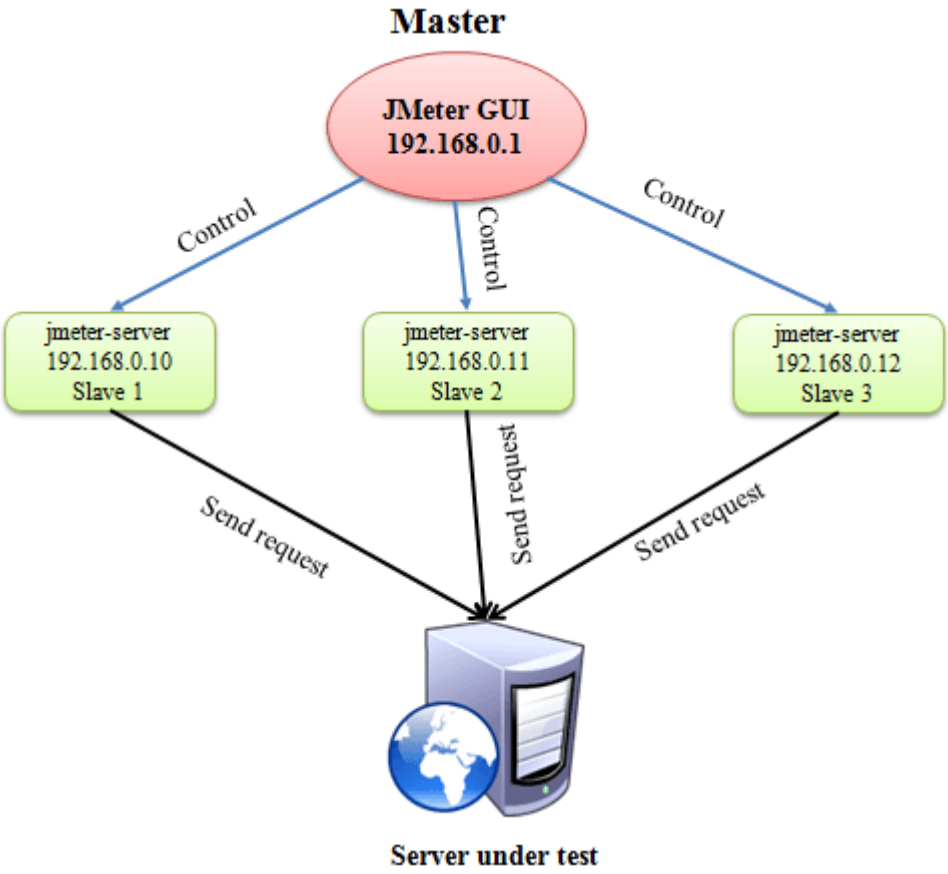


Figure 6. JMeter architecture.

5.4 LoadNinja

It is a cloud-based performance testing platform developed by Smart Bear, powered by an open-source framework (selenium). Load Ninja is a **browser-based** load-testing platform, which means that it is going to simulate real user workflow. The advantage of those solutions is that they can provide a picture of the end-user experience. However, running this operation requires more processing power and time, so running this test is more expensive.

5.5 K6

K6 is an open-source tool developed in 2017 by Grafana Lab. It is written in Go and does not have a GUI built in, but it is possible to use GUI with the k6 cloud. Like JMeter, it allows us to extend the functionalities with plugins, but fewer are available since it is a newer product. Also, the setup is more complex compared to JMeter. The advantage is related to **performance**, because Go, the language chosen, is a compiled language, contrary to Java, the language used to write JMeter, which is an interpreted language and needs an interpreter. K6 can be an interesting solution when there is the need to simulate many virtual users. In this case, the test can be cheaper to run with K6. One load generator in JMeter can simulate a few thousand virtual users; instead, with one load generator in K6, we can simulate tens of thousands of virtual users.

6 Implementation

6.1 Software under test

The first software tested was **Google Lighthouse** because there are no downsides to taking it into use. It is easy to use, and it provides valuable data. It is essential to consider because it is widely accessible, familiar and can be used as an evaluation tool.

In particular, Lighthouse was chosen over PageSpeed because of the fact that real users do not use the website under test, and there are no field data to analyse.

The intention is to use at least one **open-source** software and one **market leader**. The market leaders are HP LoadRunner and Apache JMeter. The first one is proprietary, and the second one is open source. HP LoadRunner was chosen because of its position in the market and because it is often used to compare the result of other testing software. It will hopefully help when comparing the results collected from the other software.

The open-source software tool taken into use is not Apache JMeter, even if it is the most popular one; instead, K6 was chosen because, according to several articles ([37]–[39]), it appears to be a more modern alternative focused on performance.

7 Test results

7.1 Google Lighthouse

Google Lighthouse CI was the first software under test. The intention was to understand if the complexity of the setup is paid back by the functionality unlocked by this product. Running Lighthouse CI gives access to all the data produced by running Lighthouse from the DevTool; we can also generate a report alongside every PR (Pull Request). With the adoption of this solution, it is possible to avoid regression by merging only PR that does not degrade the website's performance.

The first test to run, with as little configuration as possible, was evaluating the home page's performance three times. An average value has been returned in the report that has been published online. The values returned are very similar to the result of the test that is possible to run from the DevTool, with the difference that testing from a browser does not calculate an average score. Those scores are helpful in giving a general idea, but different websites have different needs. If a platform has a professional working from an office location as the target user, there is no need to test the website from a mobile device with low connectivity. Besides, it can be less valuable the effort in trying to reduce the bouncing rate, and maybe it can be more meaningful to see how the website reacts to user input FID instead of trying to optimise LCP as much as possible. Lighthouse CI allows the definition of custom targets regarding the amount of resources transferred or even an acceptable lighthouse score per category. These limits are called **budgets**.
Defining those values can be challenging in the first instance, but tools, usually called budget calculators [40], help make the first decision. It is a good idea to get some value from the calculator and then apply changes according to the needs.
The following test uses standard values [41] because defining those limits requires to discuss with the client or end-user.

At first, the test involved just the page where no authentication was required. But since the user needs to log in to access the most important website pages, the results collected do not help evaluate the web page's performance.

There were several options [42] for implementing authentication, but once again, the most flexible option has been chosen instead of the easiest. Puppeteer [43] is a library that provides API to control Chrome. This library allows writing an algorithm to interact with the element on the page. A script to simulate the login has been written [44], and it operates as follows:

- It waits until the login button appears on the page, then click it.
- It waits for an email and password input to appear, then fills them out and completes the login.

The setup has been challenging because several flags are necessary to launch Puppeteer correctly and I was struggling to launch it in desktop mode, the default was mobile. But since our website is not built for mobile, the scores were lower than expected.

In Figure 7, the execution of Lighthouse CI. The program recognises that configuration files are in use and refers to the budgets file, the pages on which the performance test is carried out (the home page and activities page). The homepage is accessible when the user is not logged in, and the page “activities” is accessible just after the login.

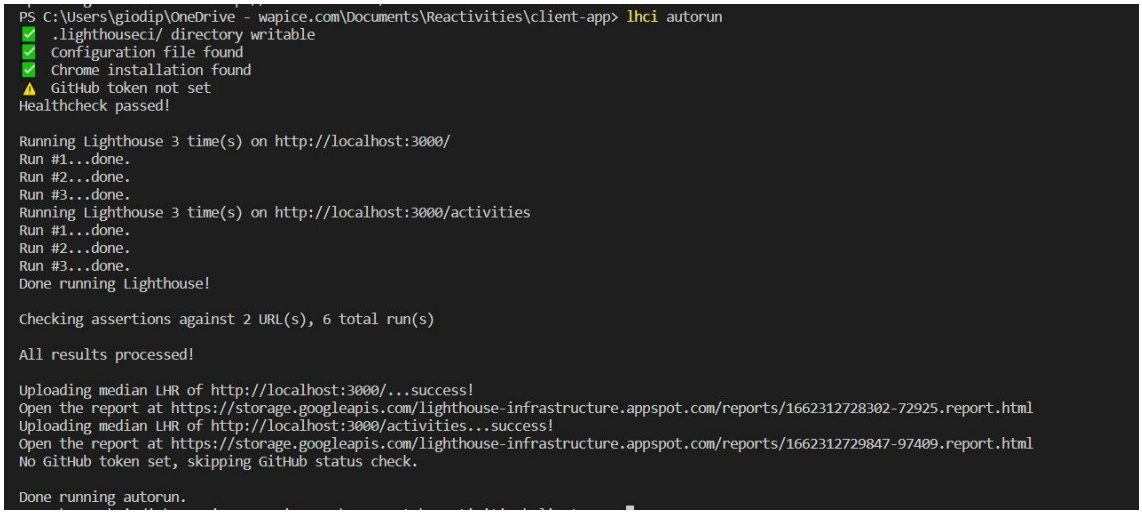


Figure 7. Lighthouse execution.

The test results are accessible through a public address, and the results will be available there for the next seven days. The results of this specific test have been saved on the GitHub repository of the test website [45].

The following image, Figure 8, is also an example of the results. A score per each category is returned, but also some specific diagnostic, where some of the problems are listed. For instance, one of the raised problems was that the image's width and height had not been explicitly defined. For this reason, the page is suffering from CLS.

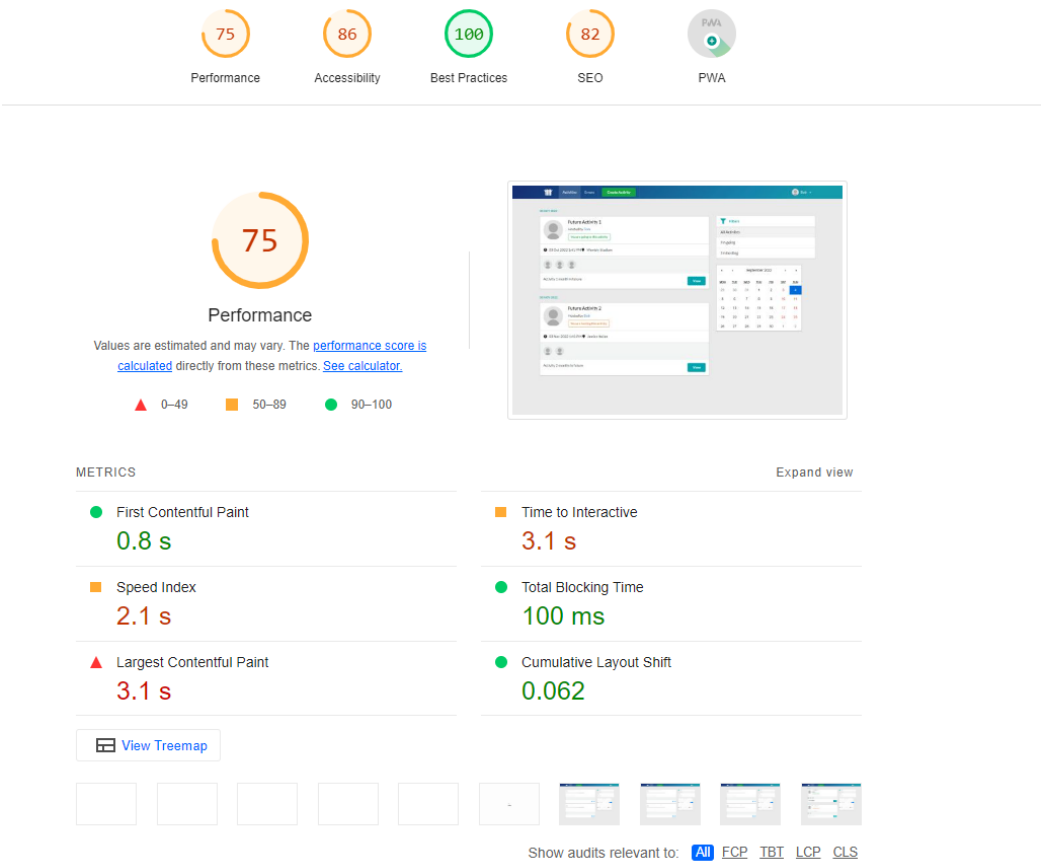


Figure 8. Lighthouse test results.

7.2 LoadRunner

Before starting testing with LoadRunner, it is necessary to choose which scripting tools to use. To take this decision, it is possible to have a lot of information from the documentation [46].
At first, VuGen was taken into use because it is one of the software that supports more protocols and allows the simulation of the interaction with the browser; this gives the possibility to create a script more efficiently compared to the interaction with the API.

Once the script is created, it can be executed locally or in the cloud. The cloud SaaS (Software as a Service) version has been used, with the 1-month trial version of the software. It is possible to execute the script from computers in

different parts of the world. It allows a more realistic simulation compared to running multiple users from the same machine.

A couple of tests will be executed, trying to replicate a real-use scenario of our website.

The first test is going to simulate the process of registering a user. The script is structured in three steps, each of which is described in Table 6. This test can be interesting to start gaining confidence with the tools in use, but it is also needed for the following tests because multiple user logins are required.

Table 6. LoadRunner registration steps.

Steps	Actions
Visit the home page	visit the page https://tutorial-reactivities.herokuapp.com/
Press the button "Register"	A new pop-up window is going to be displayed where the user's information needs to be inputted
Enter user detail and register	If the information is correct (password with number, no email already registered etc.), there will be a redirect to a page where all the activities are shown

This script requires us to use type parameters because if we want to simulate multiple users, we need to provide all the usernames to use.
It was challenging to use the type parameters with VuGen, so instead, another scripting software has been taken into use, TruClient. It is a simplified version of the first one that supports the protocols needed for my tests HTTP/HTML. This software made using type parameters to generate different usernames and emails easier an example of the graphical interface in Figure 9.

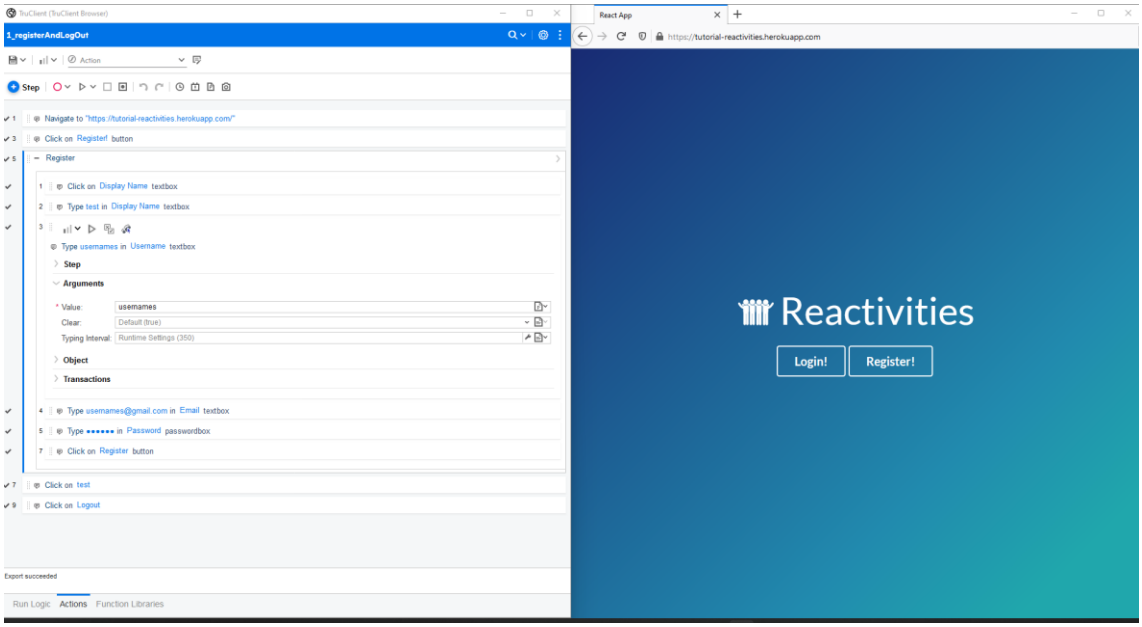


Figure 9. VuGen graphical interface.

The script [47] has been imported into the LoadRunner cloud and sets the number of virtual users to use. For this test, the number of users will start from 0 and increase to 20. To simulate a realistic scenario, users will be generated in regular intervals, not all at once. It is also possible to select from which location the users will interact. Once the test is completed, the results are accessible from a dashboard Figure 10.

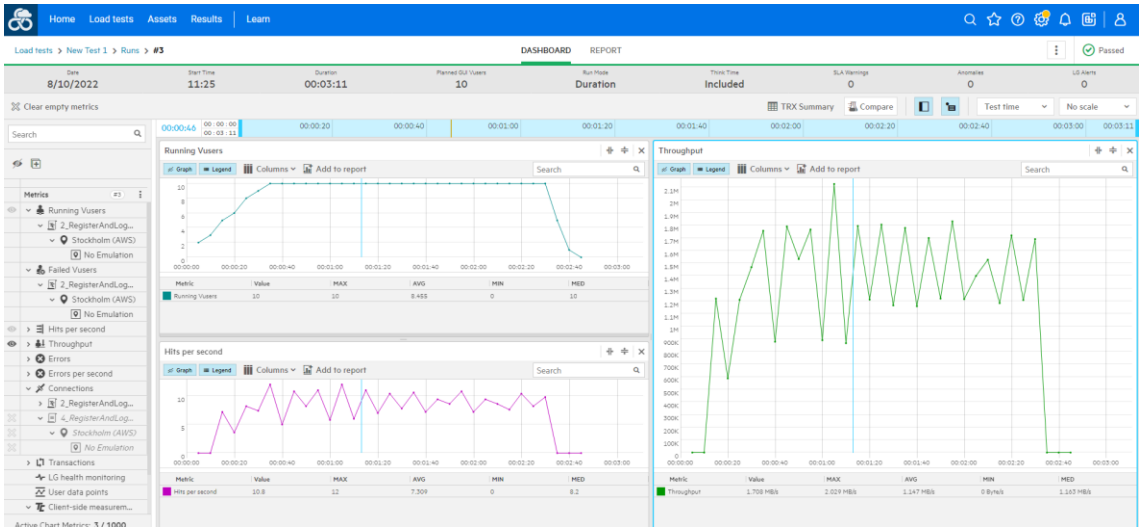


Figure 10. LoadRunner cloud dashboard.

The second script to be executed is going to test a bit more the interactions with the page. The idea of the second script is to open one activity and write a comment in the comment section.

Table 7. LoadRunner steps to write a comment.

Steps	Actions
Login	Visit the home page, press the login button, and insert the credentials
Open an activity	After being redirected to the page showing all the activities, open the first one.
Write a comment and press enter	After being redirected to the activity page, it is possible to write a comment and send it.

Some problems have been faced with the API interaction after the login, even after saving the token for authentication and adding it to the authentication header. Moving back another time to VuGen has been tried out as a potential solution because it allows more configuration, but this interaction was too complicated to test. Even after days of tutorials and debugging. It has been then decided to move on to the next test and consider this aspect for the conclusion.

Here is the script where problems were faced [48].

7.3 K6

K6 support three execution modes:

- Local: the test execution happens on a single machine
- Distributed: the test execution happens across a Kubernetes cluster
- Cloud: the execution happens on k6 Cloud or private cloud infrastructure, such as AWS; at the moment, Azure is not supported.

The following tests are going to be executed locally and in the cloud. At first, the scripts will be tested in a local environment and once the scripts are working, the tests will be replicated in a more realistic scenario. That means making use of the K6 cloud services.

When the test is running in locally, the number of virtual users and the number of completed interactions will be printed in standard output.

Once the test is completed, a summary is returned. It gives a primary overview with some statistics about the performance of the test run. But it is also possible to stream the test results to an external output. Some platforms allow a graphical visualisation of the data, such as Amazon CloudWatch, Datadog, and Influx DB, or it is possible to export the data in CSV or JSON format.

The documentation is easy to follow, and the guides are handy for beginners. There are also many step-by-step tutorials that show how to integrate K6 with other external services.

The first test executed [49] consists of getting the home page, checking the HTTP status code returned, and the duration of the request. The number of virtual users executing the script was ramping up from 0 to 9 and then again down to 0, as shown in Figure 11.

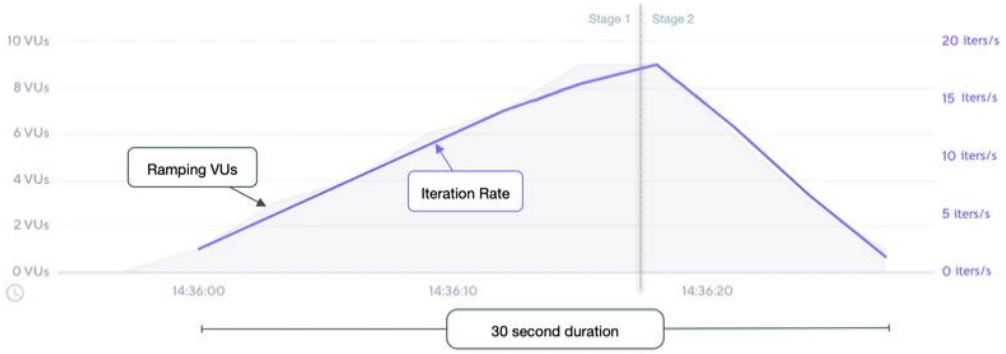


Figure 11. K6 curve of active users.

The most interesting projects/tools, articles, and videos concerning K6 are collected in an open-source GitHub repository [50]. The following tests are based on the Multi scenario template project, found in the related project section of this GitHub repository. The template has been modified to test our API. The full testing code is public on GitHub [51]. The main features are now going to be discussed.

The test is made up of different parts:

- The setup where the user authenticates and the authentication token for the following tests are saved.
- The back end is tested with a read operation. A get request is sent to fetch a specific activity
- The back end is tested with an end-to-end workflow. A post request creates an activity, and the same activity is modified with a put request and removed with a delete request.
- The UI is tested, the home page is fetched with a get request, and the title and language are checked.

Different execution modes have been defined based on the type of test that we want to perform. The different execution modes are different to each other because of how they ramp up and down the users.

- A smoke test is a preliminary test where the number of virtual users is kept very low, and the execution time is kept as low as possible. A few executions of the tests will be performed, intending to find severe failure.
- Load test, in this case, we need to simulate a more realistic scenario. We are not trying to break the system but to see how it will perform once real users use it.
  In this case, the execution time and the number of users need to be increased. The number of virtual users to use can be defined based on the expected traffic.
- In this case, the stress test aims to see the website performs when there is a peak of users. For this test, the number of users is going to rump up every interval (e.g., five minutes). The goal is to understand the application's breaking point.

In the following image, Figure 12, we can see the metrics we can analyse. The response metrics include the average response time per each scenario API

request, the error rate, the different scenarios defined, and whether it meets the requirements. In this case, all the scenarios meet the requirements.

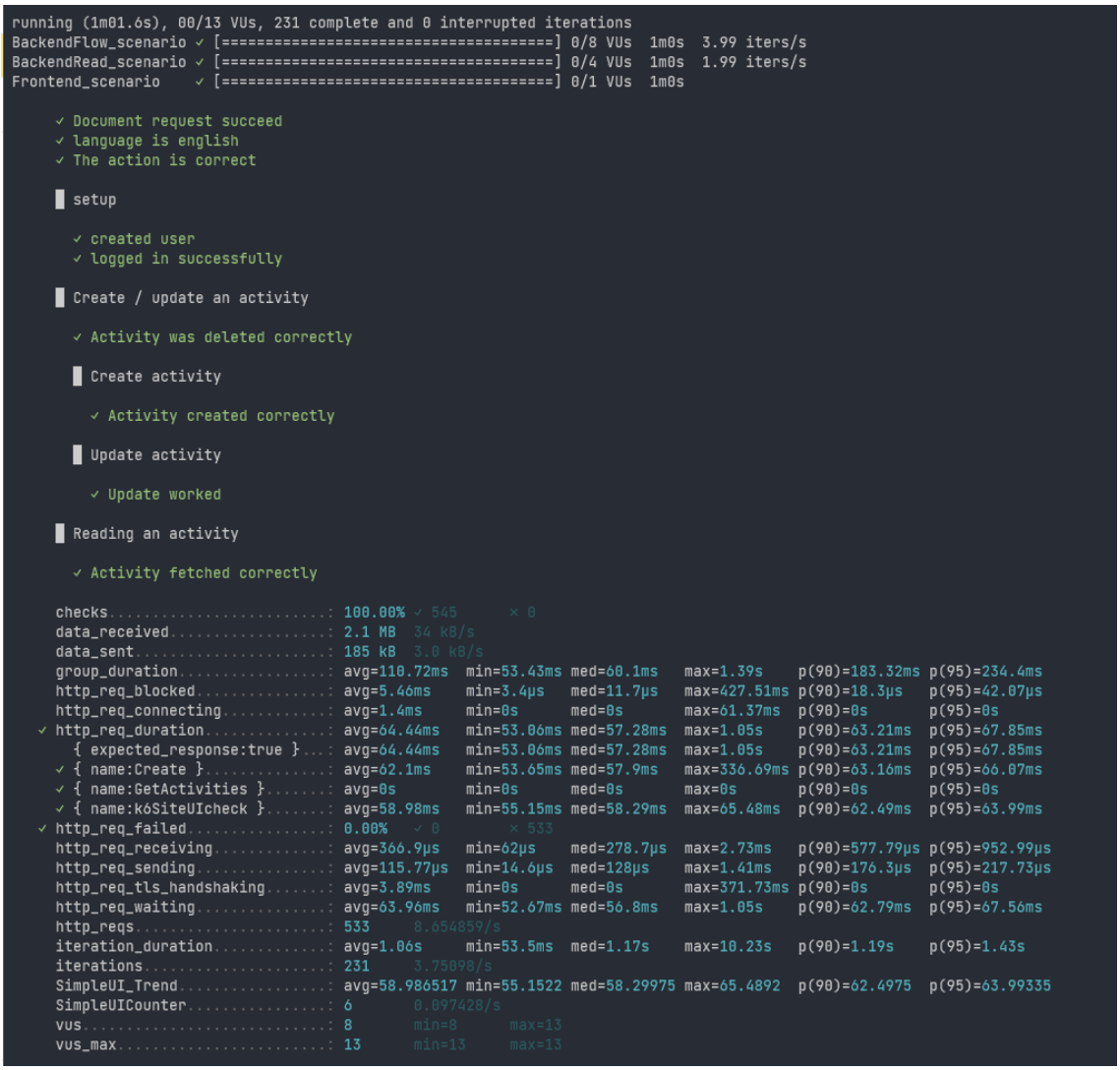


Figure 12. K6 execution.

The next step is replicating the same load test in the K6 cloud infrastructure, but since the free trial is quite limited, it was necessary to scale down the number of users to 50 and the execution time to 12 minutes. Here can be found a comparison of the results of the test run in local env compared to the one run in cloud.

Table 8. Comparison execution mode K6.

Scenario	Metric	Cloud execution	Local execution
Backend flow			
	Response time [p(95)]	86ms	74ms
Backend read			
	Response time [p(95)]	87ms	73ms
Frontend			
	Response time [p(95)]	90ms	76ms

When the test runs on the cloud, some additional tests are performed, and some of them verify that the test script is “well written”. For example, they can check if there are too many metrics defined.

Since the cloud plan was quite limited, all the following tests have been executed in the local environment. In the following table, Table 9, the results of 3 tests can be found, 2 of which are stress tests and one load test. During the first stress test's execution, many requests failed when the users exceeded 300 units. That is why in the second test, the limit of virtual users was set to around 300.

Table 9. Comparison of test results K6.

Scenario	Metric	Stress	Stress	Load test
Backend flow				
	Response time [p(95)]	589ms	85ms	74ms
	Error rate	22%	0%	0%
Backend read				
	Response time [p(95)]	557ms	84ms	73ms
	Error rate	6%	0%	0%
Frontend				
	Response time [p(95)]	169ms	79ms	76ms
HTTP req failed		9,8%	0%	0%
Virtual User				
	min	551	1	29
	max	1101	321	50

During the executions of the first stress test, an error “Too many files open” was printed. More investigation is necessary, and this message should be modified with a more explicit error.

8 Conclusion and future work

This study aimed to compare three different non-functional testing tools. After comparing the feature of the five most popular ones, three of them have been chosen to conduct the testing: Google Lighthouse, K6, and LoadRunner. To make a conscious choice on which tool to select, it has been necessary to introduce different testing techniques for both functional and non-functional testing; and the metrics used in the result returned from the non-functional testing tool. After this essential step, it was easier to compare different testing tools and reading the test results was more accessible. Moreover, it has become more straightforward to designate which testing tool is suitable for specific scenarios

After having tested Google Lighthouse, LoadRunner and K6, it was possible to understand the different use cases of each software. The following paragraphs aim to sum up the author’s experience as a developer in taking the tool into use.

Google Lighthouse is the most accessible tool because it allows users to run a simple test from the DevTool in Chrome. The great advantage of this tool is the simplicity of taking into use and interpreting the output. However, those results can be misleading because they do not consider the platform's target user but a general audience. With some minimal configuration, it is possible to define custom targets for each performance metric. In this manner, the results are going to be more accurate. The most interesting feature of Google Lighthouse is the CI/CD pipeline integration to avoid regression. In this case, it was necessary to have some configurations with Puppeteer to log in to the user and test the page where authentication was required. Once the code is integrated into the pipeline, Google Lighthouse CI is going to run a test before merging any changes, and just in case the test is passed, the changes are accepted. Thus, the performance score of the web apps is never going to downgrade under the target limit. It is essential to mention that Google Lighthouse cannot perform stress or load tests because just one user session is used.

More studies are needed to understand how to use Puppeteer with OAuth 2.0 and how it is possible to test in case of two steps authentication.

Contrary to Google Lighthouse, LoadRunner is a tool which is very difficult to get started, with a complex learning curve. The results were unsatisfying even after spending more time becoming acquainted with this tool compared to the others. The documentation and resources online were not as clear and complete as the other pieces of software. More time needs to be spent on this tool because the result can be more interesting since it is the only one that is going to test not just the API but also the JS code of the front end.

Getting started with K6 was not as easy as with Google Lighthouse. Still, it has been a pleasant experience because the documentation was complete, and many open-source resources are also available. K6, rather than Google Lighthouse, enables the execution of load and stress tests. Slight differences in the results can be noticed in running the test in a local environment compared to the cloud. The discrepancy in the response times is small because the free cloud trial allows a small number of users to be allocated. More tests need to be performed, allocating a more significant number of users to understand the inconsistency of the results. The author believes that a local execution is still a valid choice if the budget for non-functional testing is limited and does not allow buying a cloud license. Further research on how to integrate the tests into a CI/CD pipeline can be conducted.

References

[1] S. Garfinkel, “History ’ s Worst Software Bugs,” *Wired*, 2005.

[2] Inc. Akamai Technologies, “Akamai Online Retail Performance Report: Milliseconds Are Critical | Akamai Technologies Inc.,” 2017. https://www.ir.akamai.com/news-releases/news-release-details/akamai-online-retail-performance-report-milliseconds-are (accessed Apr. 25, 2022).

[3] L. Hansson, “How Swappie increased mobile revenue by 42% by focusing on Core Web Vitals,” Sep. 15, 2021. https://web.dev/swappie/ (accessed Apr. 23, 2022).

[4] E. Kinsbruner, “What Is Non-functional Testing? | Non-functional Testing Types | Perfecto,” Apr. 16, 2020. https://www.perfecto.io/blog/what-is-non-functional-testing#authorerankinsbruner (accessed Apr. 23, 2022).

[5] World quality report, “World quality report 2021-22,” 2022. Accessed: Apr. 25, 2022. [Online]. Available: https://www.microfocus.com/media/report/world-quality-report-2021-22.pdf

[6] M. Albarka Umar and C. Zhanfang, “A Study of Automated Software Testing: Automation Tools and Frameworks,” 2019.

[7] ISO, “Iso/Iec 25010:2011,” *Software Process: Improvement and Practice*, vol. 2, no. Resolution 937, pp. 1–44, 2014, [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35733

[8] M. Polo, P. Reales, M. Piattini, and C. Ebert, “Test automation,” *IEEE Softw*, vol. 30, no. 1, pp. 84–89, 2013, doi: 10.1109/MS.2013.15.

[9] Codacy, “ISO/IEC 25010 Software Quality Model - Codacy | Blog Developer.” https://blog.codacy.com/iso-25010-software-quality-model/ (accessed Jun. 14, 2022).

[10] “SYSTEM DOWNTIME AFFECTS 3 WAY S COMPANIES AND FOUR METHODS TO MINIMIZE IT 3 Ways System Downtime Affects You & 4 Methods to Minimize It 2”.

[11] “What Types of Performance Tests Should You Use? | LoadNinja Blog.” https://loadninja.com/articles/performance-test-types/ (accessed Jun. 05, 2022).

[12] T. Hamilton, “Performance Testing Tutorial: What is, Types, Metrics & Example.” https://www.guru99.com/performance-testing.html (accessed May 31, 2022).

[13] “Preparing for peak traffic”.

[14] “How to Choose A Performance Testing Tool · SPE BoK.” https://tangowhisky37.github.io/PracticalPerformanceAnalyst/pages/spe_howtos/howto_choose_a_performance_testing_tool/ (accessed Jun. 09, 2022).

[15] Moran K, “Usability Testing 101,” 2019. https://www.nngroup.com/articles/usability-testing-101/ (accessed Jun. 09, 2022).

[16] J. Nielsen, J. Lewis, and C. Turner, “Determining Usability Test Sample Size,” *International Encyclopedia of Ergonomics and Human Factors, Second Edition - 3 Volume Set*, Mar. 2006, doi: 10.1201/9780849375477.CH597.

[17] J. Nielsen and T. K. Landauer, “Mathematical model of the finding of usability problems,” *Conference on Human Factors in Computing Systems - Proceedings*, pp. 206–213, 1993, doi: 10.1145/169059.169166.

[18] ENISA, “ENISA Threat Landscape 2020 Web Application Attacks,” 2020.

[19] J. W. Palmer, “Web site usability, design, and performance metrics,” *Information Systems Research*, vol. 13, no. 2, 2002, doi: 10.1287/isre.13.2.151.88.

[20] P. B. Nirpal and K. v. Kale, “A Brief Overview Of Software Testing Metrics,” *International Journal on Computer Science and Engineering*, vol. 3, no. 1, 2011.

[21] “Key Performance Metrics for Load Testing | SoapUI.” https://www.soapui.org/learn/load-testing/key-performance-indicators-for-load-testing/ (accessed Jul. 17, 2022).

[22] “Does Page Load Time Really Affect Bounce Rate? - Pingdom.” https://www.pingdom.com/blog/page-load-time-really-affect-bounce-rate/ (accessed Jun. 28, 2022).

[23] “Page Load Time vs. Response Time - What Is the Difference? - Pingdom.” https://www.pingdom.com/blog/page-load-time-vs-response-time-what-is-the-difference/ (accessed Jul. 17, 2022).

[24] P. Walton, “User-centric performance metrics.” https://web.dev/user-centric-performance-metrics/#types-of-metrics (accessed Jul. 17, 2022).

[25] “First Contentful Paint.” https://web.dev/first-contentful-paint/ (accessed Jul. 17, 2022).

[26] P. Walton, “Web Vitals.” https://web.dev/vitals/ (accessed Jul. 17, 2022).

[27] “WebPerf WG @ TPAC 2021.” https://w3c.github.io/web-performance/meetings/2021/2021-10-TPAC/index.html (accessed Jun. 30, 2022).

[28] P. Walton, “First Input Delay (FID).” https://web.dev/fid/ (accessed Jun. 30, 2022).

[29] P. Walton and M. Mihajlija, “Cumulative Layout Shift (CLS).” https://web.dev/cls/ (accessed Jul. 17, 2022).

[30] Brian, “Best Automation Testing Tools for 2022 (Top 15 reviews) | by Brian | Medium.” https://briananderson2209.medium.com/best-automation-testing-tools-for-2018-top-10-reviews-8a4a19f664d2 (accessed Jul. 12, 2022).

[31] “Top 10 Best Load Testing Tools 2022 - Performance Lab.” https://performancelabus.com/best-load-testing-tools/ (accessed Jul. 12, 2022).

[32] T. Hamilton, “9 BEST Performance Testing Tools (Load Testing Tool) in 2022.” https://www.guru99.com/performance-testing-tools.html (accessed Jul. 12, 2022).

[33] “jasontaylordev/CleanArchitecture: Clean Architecture Solution Template for .NET 6.” https://github.com/jasontaylordev/CleanArchitecture (accessed Aug. 11, 2022).

[34] “Lighthouse performance scoring.” https://web.dev/performance-scoring/ (accessed Jul. 12, 2022).

[35] “lighthouse/architecture.md at master · GoogleChrome/lighthouse.” https://github.com/GoogleChrome/lighthouse/blob/master/docs/architecture.md (accessed Sep. 26, 2022).

[36] T. Hamilton, “What is HP LoadRunner Testing Tool? Components, Architecture.” https://www.guru99.com/introduction-to-hp-loadrunner-and-its-archtecture.html (accessed Jul. 12, 2022).

[37] M. Yaseen, “Comparison: Jmeter vs K6. Just think about checking your other… | by Mohamed Yaseen | Nerd For Tech | Medium.” https://medium.com/nerd-for-tech/comparison-jmeter-vs-k6-32c19164c7c4 (accessed Aug. 08, 2022).

[38] “(6) K6 vs. JMeter - Head2Head | LinkedIn.” https://www.linkedin.com/pulse/k6-vs-jmeter-head2head-twan-koot/ (accessed Aug. 08, 2022).

[39] A. Ghahrai, “k6 - The Best Developer Experience for Load Testing.” https://devqa.io/k6-load-testing/ (accessed Aug. 08, 2022).

[40] “Performance Budget Calculator.” https://perf-budget-calculator.firebaseapp.com/ (accessed Sep. 26, 2022).

[41] “Reactivities/budget.json at feature/last-chages · diprimagiorgio/Reactivities.” https://github.com/diprimagiorgio/Reactivities/blob/feature/last-chages/test_googleLighthouse/budget.json (accessed Sep. 26, 2022).

[42] “lighthouse/authenticated-pages.md at master · GoogleChrome/lighthouse · GitHub.” https://github.com/GoogleChrome/lighthouse/blob/master/docs/authenticated-pages.md (accessed Sep. 04, 2022).

[43] “puppeteer/puppeteer: Headless Chrome Node.js API.” https://github.com/puppeteer/puppeteer (accessed Sep. 26, 2022).

[44] “Reactivities/lh-auth.js at feature/last-chages · diprimagiorgio/Reactivities.” https://github.com/diprimagiorgio/Reactivities/blob/feature/last-chages/test_googleLighthouse/lh-auth.js (accessed Sep. 26, 2022).

[45] “Reactivities/localhost_2022-09-04_20-31-43.html at feature/last-chages · diprimagiorgio/Reactivities.” https://github.com/diprimagiorgio/Reactivities/blob/feature/last-chages/test_googleLighthouse/localhost_2022-09-04_20-31-43.html (accessed Oct. 03, 2022).

[46] “Scripting tools.” https://admhelp.microfocus.com/lrc/en/2022.06/Content/Storm/lp_scripting_tools.htm?Highlight=create%20script (accessed Sep. 24, 2022).

[47] “Reactivities/1_registerAndLogOut.zip at feature/last-chages · diprimagiorgio/Reactivities.”

https://github.com/diprimagiorgio/Reactivities/blob/feature/last-chages/test_loadrunner/1_registerAndLogOut.zip (accessed Sep. 25, 2022).

[48] “Reactivities/Action.c at feature/last-chages · diprimagiorgio/Reactivities.” https://github.com/diprimagiorgio/Reactivities/blob/feature/last-chages/test_loadrunner/Action.c (accessed Sep. 25, 2022).

[49] “HomePage test k6.” Accessed: Sep. 23, 2022. [Online]. Available: https://github.com/diprimagiorgio/Reactivities/blob/feature/last-chages/test_k6/testDurationHomePage.js

[50] “grafana/awesome-k6: A curated list of resources on automated load- and performance testing using k6 .” https://github.com/grafana/awesome-k6 (accessed Sep. 23, 2022).

[51] “Reactivities/k6_multi_scenario_template.js at feature/last-chages · diprimagiorgio/Reactivities.” https://github.com/diprimagiorgio/Reactivities/blob/feature/last-chages/test_k6/k6_multi_scenario_template.js (accessed Sep. 23, 2022).