Comparative Performance Analysis between Spring Boot and Quarkus: An Empirical Study

Gabriel Ferreira da Rosa, Kleinner Farias, Carlos Fernando Santos Xavier
PPGCA, University of Vale do Rio dos Sinos, São Leopoldo, RS, Brazil
gabrielfr97@gmail.com, kleinnerfarias@unisinos.br, xavier@edu.unisinos.br

Abstract: Performance plays a key role in web application designs, and the topic is widely researched in the literature. Over the past few years, several Java frameworks, such as Spring Boot and Quarkus, have sought to improve their performance to remain relevant in the market. However, the current literature lacks comparative analyzes that help developers to choose one of the frameworks. Despite being widely used, few studies seek to provide comparative analysis, which leads many developers to rely on their experiences and not on empirical knowledge. This study, therefore, reports a comparative performance analysis between Spring Boot and Quarkus. For this, a case study was carried out in the context of communication scenarios via messages and their persistence in a database. And two applications were developed using their respective frameworks. Data from the Central Processing Unit (CPU), Random Access Memory (RAM) consumption and message processing time were used to measure the performance of each target application. The indicators obtained showed, statistically, that Quarkus presents a slightly superior performance in most of the analyzed scenarios. However, this is an initial study that seeks to explore the topic and pave the way for future research with other scenarios and elements.

Keywords: Spring Boot; Quarkus; RabbitMQ; native; performance.

1 INTRODUCTION

Performance, a non-functional requirement, is of paramount importance, being more important even than individual functional requirements. A poor performance can mean the disposal of an entire service or even a set of services, consequently resulting in financial losses (SOMMERVILLE, 2011). Based on the premise, *several Java frameworks* sought to improve their performance, either *through internal refactoring* or the implementation of new technologies. Technologies such as Ahead-of-time (AOT), java compilation for native code, Hot Reload of Live Code, container affinity, among others, lead to *modifications of existing frameworks* and the creation of new ones. However, the literature in these cases does not follow at the same speed of the changes made, thus creating questions such as: "Which framework should I use?".

It may seem like a simple question. However, as Sommerville (2011, p. 302) points out: "They are inherently complex and can take months for someone to learn how to use them. It can be difficult and expensive to *evaluate frameworks* available for choosing the *most* appropriate framework."

Choosing the tool that best meets customer needs, in this *case a framework*, brings a set of benefits such as: reducing infrastructure costs, reducing response time, and a better user experience (LARSSON, 2020). Therefore, analysis studies between Spring Boot and Quarkus help in this evaluation process, as studies on the most diverse scenarios are elaborated.

Some studies on the subject seek to perform this analysis between Spring Boot and Quarkus. The study by Almeida (2020) analyzes applications compiled in native code and code to be executed by the Java Virtual Machine (JVM) through HTTP (Hypertext Transfer Protocol) requests. However, analysis studies between Spring Boot and Quarkus, both native, using RabbitMQ as a *message broker*, were not found. Therefore, this gap motivated the development of this work, based on the cited work.

Therefore, this article reports a comparative performance analysis between Java frameworks: Spring Boot and Quarkus. For this, a case study was carried out to evaluate Spring Boot and Quarkus in the context of an asynchronous messaging application, in terms of CPU consumption, memory consumption and processing time. A messaging application was developed with Spring Boot and Quarkus. We try to keep the code as close as possible between applications, avoiding specific features of each framework. Such applications were submitted to 3 message loading scenarios, 150,000, 250,000 and 500,000, each one being executed 10 times. The results obtained show that Quarkus has an advantage over Spring Boot in most variables analyzed. The results obtained can benefit future developers in choosing the framework that will bring the best result in an asynchronous message context.

The study is structured as follows: Section 2 contains the theoretical foundation, which brings concepts that are used in research; Section 3 deals with related work, bringing a brief summary; Section 4 describes the methodology used, elucidating the objectives and hypotheses of the research as well as its variables and data capture and analysis process; Section 5 presents the results obtained through tables and graphs; and finally, Section 6 addresses conclusions and future work.

2 THEORETICAL FOUNDATION

This section discusses the theoretical concepts used during the construction and development of the study.

2.1 Native code and GrailVM

Java is known to be an interpreted language, in which Bytecode runs on the JVM. In older versions, they performed much lower than languages compiled directly to native code. However, versions were released that brought techniques to mitigate this problem, such as JIT-compilers. However, new technologies such as GraalVM seek to make it even faster by precompiling Java into native code, thus allowing you to get closer to languages that compile to native code such as C (LARSON, 2020).

GraalVM came up with the proposal to be the next generation of *virtual machines* of very high performance. For this purpose, it has brought together a set of features *such as Grail Compiler*, *GraalVM Native Image Mechanism,* T*ruffle Language Implementation Framework,* and *Sulong*, which are essential to ensure polyglot capability (Šipek; Muharemagić; Mihaljević; Radovan, 2020).

2.2 RabbitMQ

RabbitMQ is a message broker developed by Rabbit Technologies, which aims to manage messages on distributed systems. Therefore, it can integrate different systems with different languages, providing load, fault, and messaging management (IONESCU, 2015). RabbitMQ uses the *Advanced Message Queuing Protocol* (AMQP), which enables asynchronous communication between the entities that interact with it. In addition, entities do not need to be working at the same time to communicate, RabbitMQ can save messages until a consumer is available and fits into the processing rules (SHARVARI; NAG, 2019).

Messages are received and sent according to rules that are defined at the time of configuration. When a message is published, it must have a *routing key* that will address the correct queue. When there is processing, or consumption as it is also known, the entity that processes this message receives the message, performs all the processing on it, and at the end sends RabbitMQ an acknowledgment (ACK), which indicates that the message has been processed without the occurrence of problems (SHARVARI; NAG, 2019).

2.3 Frameworks

Spring Boot is a framework that came from a simplification of the Spring Framework. The objective was to make the setup time as small as possible, that is, from the moment the project was created in a few steps the application is ready to move up to the production environment (GUTIERREZ, 2019). Another major advantage is its integration with third-party market applications, where Spring Boot offers a fast integration process through its annotations and configuration management (MOHAN, 2022).

Quarkus is a versatile Java framework, ideal for *serverless*, microservices, and containers. To do so it offers a low boot time, low RSS (*Resident Set Size*) memory consumption, and good scalability (PLESSIS; Mendes, MENDES, CORREIA, 2021). Quarkus has a different treatment when it comes to Java Reflection, because this mode is the opposite of the premise of the native mode. In native mode, you want to know all the information of the classes at the time of compilation, while Java Reflection gets the information when the application is running. To get the best of both, Quarkus offers configuration files and annotations to make use of Java Reflection (KOLEOSO, 2020).

2.4 Case study

Case study is a form of empirical investigation that seeks to investigate a situation within a given context, using quantitative and qualitative methods. It is a comprehensive research strategy, which can be carried out in several ways, depending on the procedures chosen by the researcher according to the situation. However, procedures should be followed to ensure the quality of research (YIN, 2001). According to Wohlin (2012), it is several sources of evidence in order to investigate phenomena in a given context, especially when there is no clarity between phenomenon and context.

Due to the characteristic of both the case study and the frameworks mentioned in section 2.3, they are aligned with specific contexts and also because a clear separation between the phenomena and the context of the object under study cannot be achieved. The case study was chosen because it consists of an investigation to evaluate how the two frameworks behave in a given context.

3 RELATED WORK

The search for the related works was carried out in the digital *repository Google Scholar*. The main term used to perform the selection of works was "spring boot quarkus performance".

3.1 Analysis of related works

(ALMEIDA, 2020). The work presents a comparison between native Spring Boot, Spring Boot JVM, Native Quarkus, and Quarkus JVM. Through these forms of compilation, execution and Java frameworks, tests were performed to compare the performance of applications. The number of *HTTP (Hyper Text Transfer Protocol*) requests and the response time obtained for each application were taken into account. The author presents the difficulties encountered inherent to the use of new technologies that are being matured. In this study, there was a superiority of native Quarkus in almost all executed scenarios. Finally, it brings an important observation noted both by the author and in Larsson's study (2020, p. 1).

> "[...] we compared the performance of the community edition and enterprise edition versions of GraalVM to OpenJDK and OracleJDK, using Java 8 and Java 11 [...]. We found that the performance of the different JDKs vary significantly depending on the test, which makes it difficult to make any definitive conclusion." Larsson, R. [Our translation].

(BACK, 2016). It presents a comparative analysis between the methods of integration of REST (*Representational State Transfer*) and AMQP (*Advanced Message Queuing Protocol) services*. Several tests were performed in the local environment and Heroku where the latency and flow of the two approaches were measured. Also reporting the complexity observed in each method.

(BUONO; PETROVIC, 2016). In this study, the authors propose a new form of communication between microservices, replacing text-based communication with binary-based communication through Protocol Buffers. To do so, they use a modern architecture with Quarkus and GraalVM. It also highlights its integration with Kubernetes that allows during a moment of heavy data traffic and the need for more instances, Quarkus provides the rapid rise of a new instance. In this sense, the authors conclude that the use of binary communication leads to a considerable reduction in response time.

(RITZAL, 2020). It introduces the technologies that GraalVM uses and how they become faster than other *Virtual Machines* for Java. In addition, there is a brief introduction to major Java *frameworks* such as Spring, Micronaut, and Quarkus. With the use of Grailvm it is possible to optimize these *frameworks*. The paper presents a comparison between the JVM and the GraalVM regarding application startup time, memory usage, and runtime.

(SOUZA, 2020). Provides an overview of RabbitMQ and Apache Kafka technologies. It makes an analysis both in qualitative terms such as time decoupling and delivery assurance, as well as in quantitative terms through latency and tests of controlled environment. It also provides an analysis of which scenarios each technology performs best.

(FONG; RAED, 2021). The paper presents a Java Development Kits (JDKs), GraalVM Enterprise Edition (EE) and Community Edition (CE) performance test against Oracle JDK and OpenJDK for Java 8 and 11. For this, a collection of test cases was used where each JDK was tested. The results obtained show that GraalVM EE 11 obtained a better performance in most tests, however it was also verified that the hardware plays a fundamental role in its performance.

3.2 Comparative analysis of related works

Comparison Criteria. The Comparison Criteria (CC) are used to perform the comparison between the proposed work and the selected related papers.

- **Empirical Study (CC1):** the conclusions are based on concrete empirical evidence and can be validated through data obtained in experiments and tests.

- **Memory consumption analysis (CC2):** the ability to check the memory consumption of the tested application.
- **CPU consumption analysis (CC3): the** ability to check the processing consumption of the tested application.
- **Message processing time analysis (CC4):** the ability to check message processing time.

The result of comparing the related works and the proposed work is presented in Table 1. From the analysis, the following points were observed: only 3 studies analyze resource consumption; only 1 job checks the processing time of the messages; no paper presents an analysis of message processing time combined with resource consumption.

Table 1 - Comparative Analysis of Selected Related Papers

Related Work	**Comparison Criterion**			
	CC1	**CC2**	**CC3**	**CC4**
Proposed Work	X	X	X	X
(ALMEIDA, 2020)	X	X	X	0
(BACK, 2016)	X	0	0	X
(BUONO; PETROVIC, 2016)	X	X	0	0
(RITZAL, 2020)	X	X	0	0
(SOUZA, 2020)	X	0	0	X
(FONG; RAED, 2021)	X	0	0	0

Research opportunities. Based on the points highlighted in Table 1, the following research opportunity was identified: execution of a empirical study case on the performance of Spring Boot and Quarkus, native, in a context of asynchronous message, using the criteria explained above. The research opportunity is explored in the sections below.

4 METHODOLOGY

The section reports the methodology used in the execution of empirical research. Section 4.1 presents the objective and question of the proposed research. Section 4.2 presents the hypotheses. Section 4.3 presents the variables and metrics. Section 4.4 the artifacts and tools that were used to carry out the study. Section 4.5 displays the applications that will be evaluated. Section 4.6 describes the experimental process. Section 4.7 describes the analysis process. The methodology is based on previous studies: (LAZZARI, 2021), Evaluating the effort of composing design models: a controlled experiment (FARIAS; Garcia, GARCIA, 2010 WHITTLE; CHAVEZ; LUCENA, 2013) and Wohlin (2012).

4.1 Research Objective and Question

The **primary** objective of this research is to make a comparative analysis of performance between the Spring Boot and Quarkus frameworks, in the context of asynchronous messaging applications. This objective seeks to understand the effects of frameworks on three different variables: CPU consumption, RAM consumption and message processing time. Such frameworks are analyzed through synthetic message processing scenarios (Section 4.6), using RabbitMQ as a messaging platform. Next, the objective of this research is formulated following in the MQM model (BASILI, 1992):

Analyze Java *frameworks*
for the purpose of investigating its effects
in relation to performance
through the perspective of developers
in the context of asynchronous messaging applications.

The article aims to produce empirical evidence on the performance of Spring Boot and Quarkus, both using GraalVM, in message processing scenarios. In this sense, the following research question (QP) was formulated:

QP: Is Quarkus performance superior to Spring Boot in the context of asynchronous messaging applications and compiled in native code with GraalVM?

4.2 Hypotheses

Hypothesis one. It is conjecture that Quarkus can present a superior performance to Spring Boot, since the research conducted by Almeida (2020) indicates this advantage in a context of HTTP requests. Quarkus seeks other mechanisms for optimization, such as the disincentive of using the reflection API *that* has high cost, unlike spring boot that makes extensive use (KOLEOSO, 2020). Another point to be observed is that the native Spring Boot is still in the experimental phase, unlike Quarkus which has a final version. Despite this, Spring Boot has integration libraries with RabbitMQ and MongoDB developed specifically for it, which can translate into better performance.

In this way, the performance will be a resume of the use of hardware resources and processing time of messages, therefore declaring the null and alternative hypotheses as follows:

Null Hypothesis, Null H: Spring Boot uses less (or equal) hardware resources and has less (or equal) message processing time than Quarkus.

$H1_{null}$ *: Performance(Quarkus) ≤ Performance(Spring Boot)*

Alternative Hypothesis, H_{alt}: Quarkus uses fewer hardware resources and has shorter message processing time than Spring Boot.

$H1_{alt}$ *: Performance(Quarkus) > Performance(Spring Boot)*

By testing this hypothesis, it will allow you to check *the use of hardware* resources and time of consumption of messages. Thus comparing through these variables the two target applications. The data collected served as an insum for the decision-making of future users of these *frameworks*.

4.3 Variables and Metrics

Independent variable. The independent variable in this hypothesis is *that of frameworks* and message-oriented architecture.

Dependent variable. The dependent variable in this hypothesis is that of performance metrics defined in Table 1. The knowledge of this variable allows the performance of analyses to understand the impact of each *framework on* the tests. The variable is divided into three facets: CPU consumption, RAM consumption, and message processing time.

CPU consumption. Data represents the percentage of cpu usage of the system, which is captured every five seconds by Prometheus and displayed in Grafana graphics. The data is obtained by an average of all values captured during the test period.

RAM consumption. Data represents in MebiByte the use of RAM, which is captured every five seconds by Prometheus and displayed in Grafana graphics. The data is obtained by an average of all values captured during the test period.

Message Processing Time. Obtained by the time difference between the target application start command and the last message written to the MongoDB database. This information is contained in the *date field*, in the body of the message. When the target application under test processes the message, this field receives the current date and time and writes the message to the database.

The performance metric is represented in Table 2.

Table 2 - Performance metrics

Name	**Description**	**Tool**
Memory consumption	Memory resource consumption during load testing in MebiByte.	Grafana
CPU consumption	CPU resource consumption during load tests in percentage.	Grafana
Message processing time	Time elapsed between the target application start command and the recording of the last message in the database.	MongoDB

4.4 Artifacts and tools used in this research

The study requires tools and artifacts to be done. Therefore, Table 3 presents the list with all the elements used and their respective versions.

Table 3 - Artifacts and tools used in the research

Artifact	Version
Maven	3.6.3
Spring Boot	2.6.4
Spring Boot Actuator	2.6.4
Spring Boot Web	2.6.4
Spring Boot AMQP	2.6.4
Spring Boot Mongodb Data	2.6.4
Spring Boot Native (experimental)	0.11.3
GraalVM	22.0.0.2
MongoDB	5.0.6
Rabbitmq	3.8.4
Grafana	8.5.0
Prometheus	2.35.0
Prometheus Registry Micrometer	1.8.3
Quarkus	2.7.4
Quarkus Camel Good	2.7.4
RabbitMQ Client Quarkus	0.5.0
Resteasy Quarkus	2.7.4
Mongo Client Quarkus	2.7.4
Jackson Databind	2.12.4

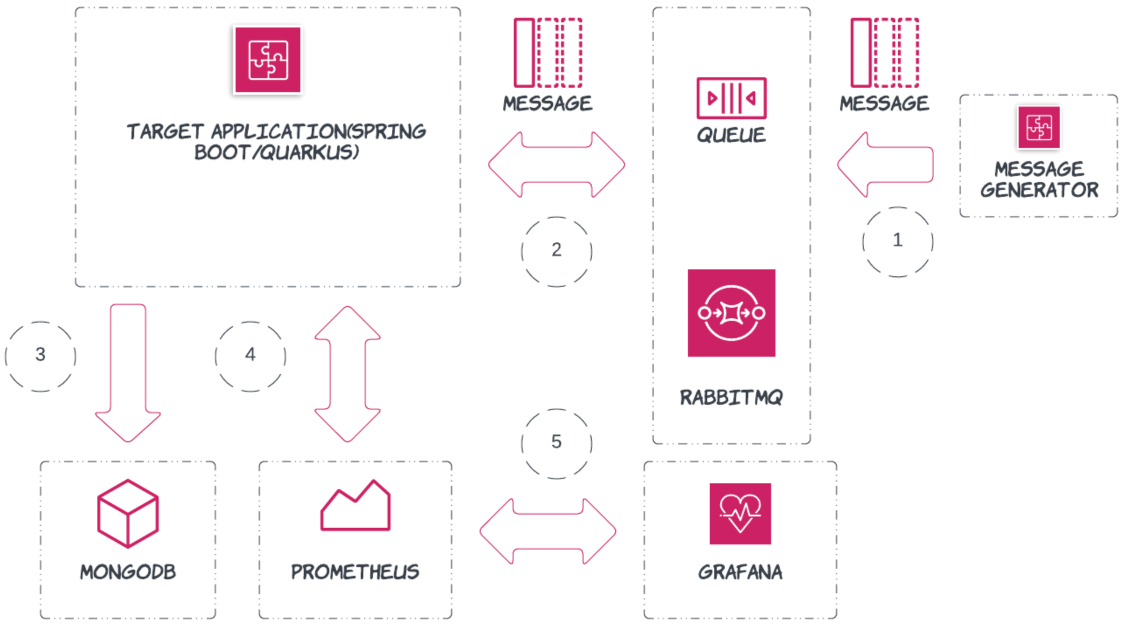
The machine used to perform the tests was a Lenovo notebook with 16GB of DDR4 RAM, 256GB Solid State Drive (SSD), Intel Core i7 10610U 1.8GHz processor and Ubuntu operating system.

4.5 Target applications

Two applications have been developed for research. One being developed using Spring Boot and the other using Quarkus, both use the Java language and GraalVM to perform the build for native code. The applications are small, perform only the processing of messages consisting of receiving the message, setting the *date field*, saving to the bank and returning the ACK to RabbitMQ. Therefore, it is possible to avoid possible interference during the processing of a message and the data obtained are actually related to the *framework* under test.

The applications connect with three tools: message broker RabbitMQ, MongoDB database and with prometheus monitoring tool. Figure 1 has a squeematic illustration of the data flow of target applications. In step 1 a message-generating application is triggered; in step 2 the target application, under test, connects *to the* RabbitMQ message broker to receive the messages and send the reading confirmation; in step 3 saves the messages in the MongoDB database; in step 4 there is sending information containing resource *consumption* to Prometheus and finally in step 5 there is communication between Prometheus and Grafana so that the graphics are generated in Grafana. It is important to note that step 2 starts only when step 1 finishes, that is, when you finish sending the defined batch of messages. Step 4 data flow takes place independently of steps 2 and 3.

Figure 1 - Illustration of the data flow



4.5.1 Load tests

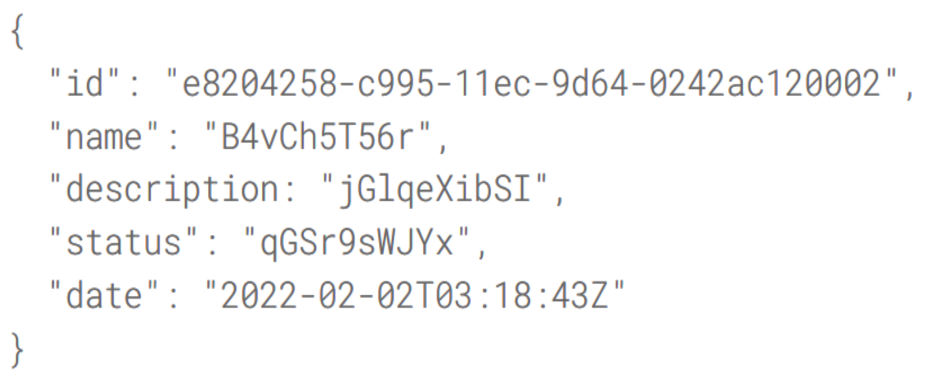
Load tests start when the message generator fires, as a parameter, receiving as a parameter the number of messages that must be generated. Figure 2 shows the structure of the message. As soon as all messages are sent, the target application starts up and consumes the messages. When initializing, communication with Prometheus is initialized and soon after the graphics begin to be formed in Grafana, thus allowing to follow the test and start capturing the metrics.

Some important points:

- The startup period of the application, where there is no message consumption yet, is taken into account in metrics.
- Each test all data from the previous test is removed from the database.

Figure 2 shows the message structure used in the tests. The message generator populates the fields: *id*, *name*, *description*, and status with *random data*. The date *field* is populated in the target application, under test, before the message persists in the database.

Figure 2 - Message structure



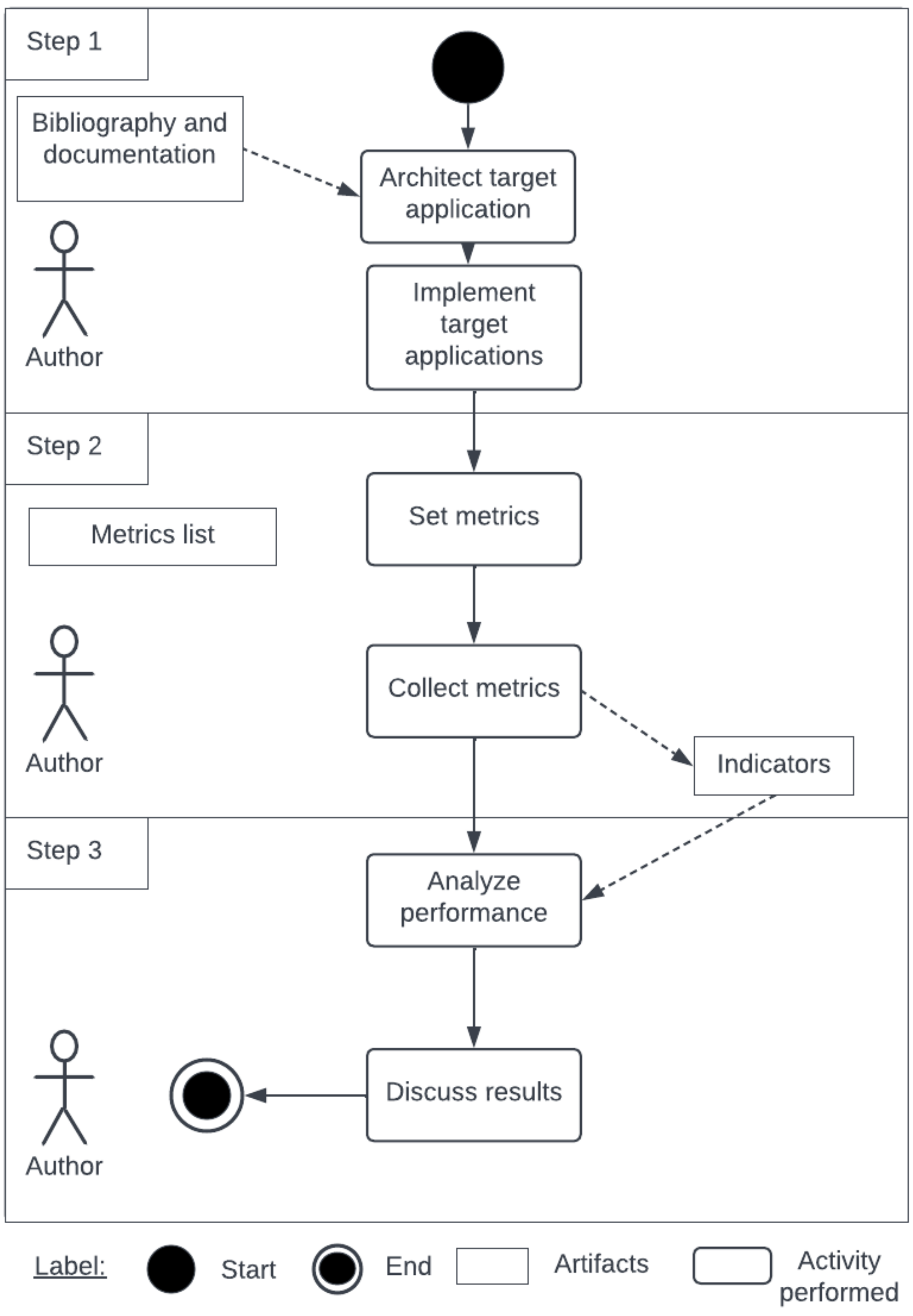
```
{
  "id": "e8204258-c995-11ec-9d64-0242ac120002",
  "name": "B4vCh5T56r",
  "description: "jGlqeXibSI",
  "status": "qGSr9sWJYx",
  "date": "2022-02-02T03:18:43Z"
}
```

The strategy adopted for this research was to produce a certain number of messages, send them to RabbitMQ and then consume them. The reason is that there is no interference of the producer agent on the metrics collected from the target application, that is, if the producer has a lower production capacity than the consumer's consumption capacity, it will not impact the result.

4.6 Experimental process

Figure 3 shows the experimental process adopted, which consists of three steps: developing the applications (1), defining and collecting metrics (2) and analyzing the collected data (3). Each step is elucidated below.

Figure 3 - Experimental process



Step 1: Develop target applications. In this stage the research focuses on the search for studies and documentation *of the frameworks* that serve as an input for the development of target applications. The result was the implementation of two applications that are aligned with the latest studies and technologies used by the community. As well as a code that uses good practices and follows the documentation made available.

Step 2: Define and collect metrics. Metrics are defined in this step according to the list of resource consumption metrics made available by Java and the message consumption time metric. From the definition, the load tests defined in Section 4.5.1 start, at the same time the metrics are collected and grouped. The result are indicators that served as an input for Step 3.

Step 3: Analyze the collected data. The study now searches through the indicators collected in Step 2 and application of the Wilcoxon test, analyze and confirm or refute the hypotheses proposed in Section 4.2. To this end, the indicators are displayed in tables and graphs that allow the visual identification of the results of the study.

4.7 Analysis process

Quantitative analysis. Statistical inference is used to perform the test of the proposed hypothesis. Wilcoxon's nonparametric test is applied, as it does not require two separate sets of identically distributed samples. A significant difference is found when *p-value* $\leqslant 0.05$.

Qualitative analysis. Qualitative data are collected from what is observed during the development of the research. Thus, non-tangible evidence, only by quantitative data, can be presented and enrich the research.

5 RESULTS

This section aims to present the results regarding the research questions formulated in Section 2. Section 5.1 presents descriptive analysis of the collected data. Section 5.2 presents a discussion of the hypothesis elaborated in section 4. Section 5.3 presents a discussion about the results of the variables analyzed. Section 5.4 presents the limitations of the study. Finally, section 5.5 reports some observations made during the study.

5.1 Descriptive statistic

The section discusses the aspects of the collected data regarding the performance of the *frameworks* studied. For this, trends such as means and medians and dispersions are used by means of the standard deviation to perform an analysis on the distribution of data of the observed variables.

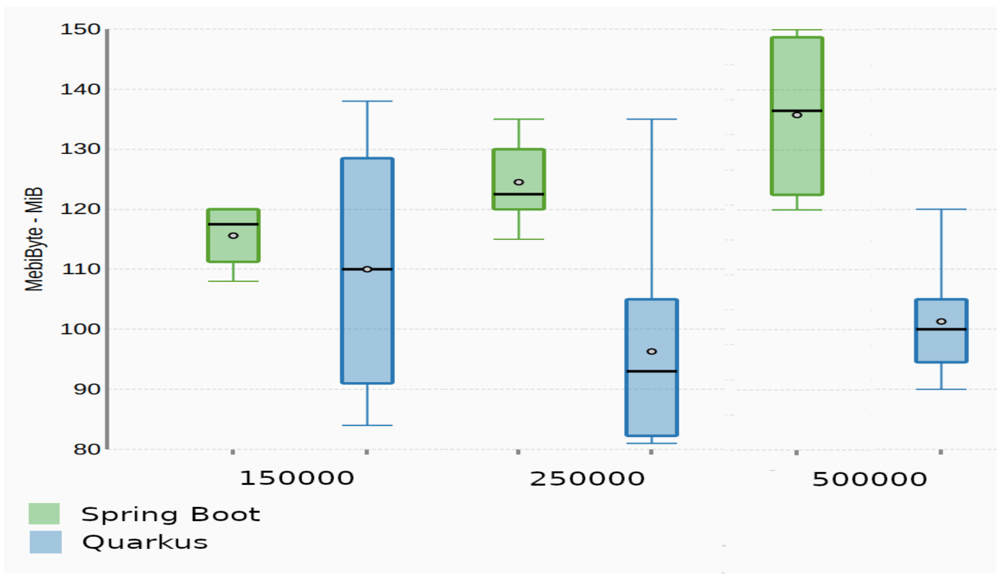
5.1.1 Memory Consumption

Descriptive statistic. The average ram usage for each scenario with Spring Boot were 115.6MiB, 124.5MiB and 135.8MiB, while for the scenarios with Quarkus were 110MiB, 96.3MiB and 101.3MiB. Therefore, it was found that on average Quarkus uses less RAM, it was also found that Quarkus presents a higher standard deviation, especially in the scenarios of 150,000 and 250,000 messages. Table 4 better evidences these behaviors.

Table 4 - Memory Consumption Result (measured in MebiByte - MiB)

Target application	Number of messages	N	Minor	Q1	Median	Q3	Bigger	Average	Standard deviation
Spring Boot	150000	10	108	111,25	117,5	120	120	115,6	5,21
Quarkus	150000	10	84	91	110	128,5	138	110	21,5
Spring Boot	250000	10	115	120	122,5	130	135	124,5	7,61
Quarkus	250000	10	81	82,25	93	105	135	96,3	17,04
Spring Boot	500000	10	120	122,5	136,5	148,75	150	135,8	12,81
Quarkus	500000	10	90	94,5	100	105	120	101,3	9,03

Figure 4 - Box-plot diagram of memory consumption.



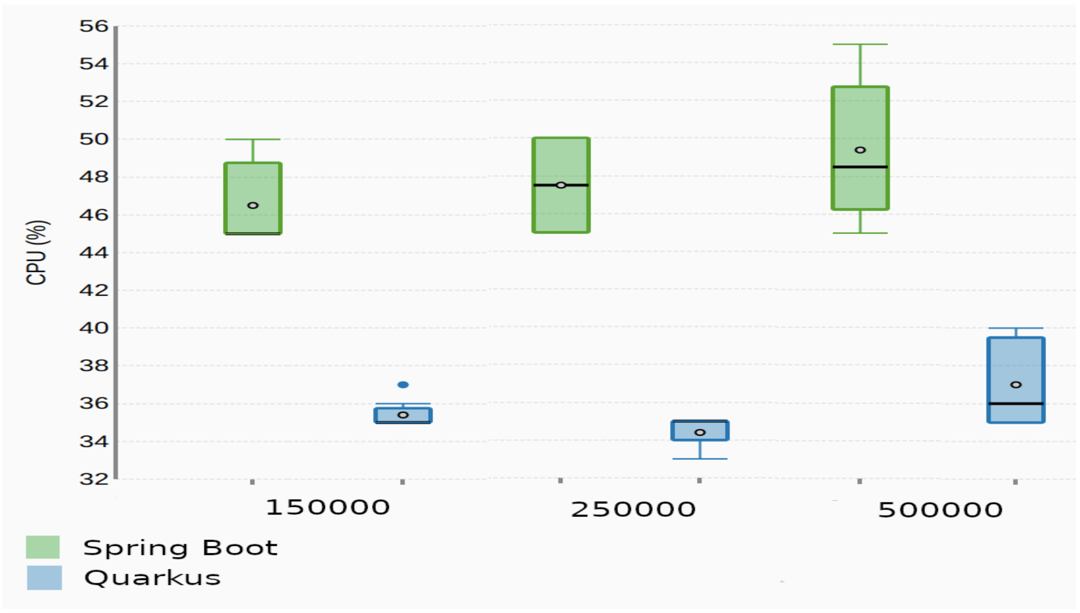
5.1.2 CPU consumption

Descriptive statistic. The average CPU usage for scenarios with Spring Boot were 46.5%, 47.5% and 49.4%, while for the scenarios with Quarkus were 35.4%, 34.4% and 37%. Soon, it was found that Quarkus consumes less CPU resources than Spring Boot. In addition, Spring Boot presented a higher standard deviation than Quarkus in all scenarios, noting that in scenarios where there are fewer messages (150,000 and 250,000), this difference is greater.

Table 5 - CPU Consumption Results

Target application	Number of messages	N	Minor	Q1	Median	Q3	Bigger	Average	Standard deviation
Spring Boot	150000	10	45	45	45	48,75	50	46,5	2,41
Quarkus	150000	10	35	35	35	35,75	37	35,4	0,69
Spring Boot	250000	10	45	45	47,5	50	50	47,5	2,63
Quarkus	250000	10	33	34	35	35	35	34,4	0,84
Spring Boot	500000	10	45	46,25	48,5	52,75	55	49,4	3,8
Quarkus	500000	10	35	35	36	39,5	40	37	2,3

Figure 5 - CPU consumption box-plot diagram.



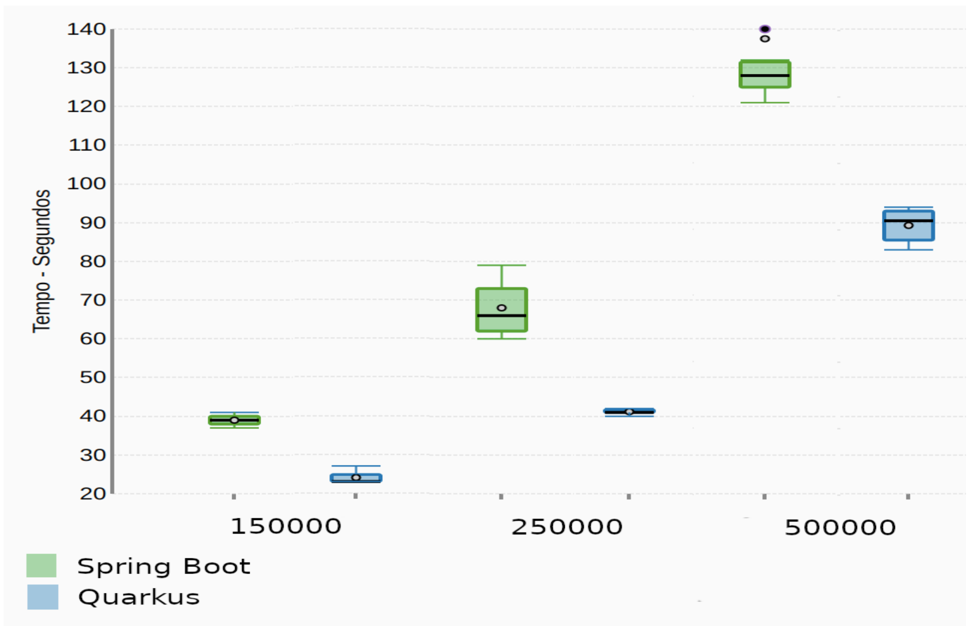
5.1.3 Runtime

Descriptive statistic. The average processing time for the spring boot scenario was 39s, 68s and 137.5, while for the scenarios with Quarkus were 24s, 41.2s and 89.3s. Through these values it was possible to notice that Quarkus can process messages faster in all scenarios. With the increase in the number of messages the standard deviation had a large increase in scenarios with Spring Boot, jumping from 1.49s with 150,000 messages to 28.2s with 500,000 messages, as shown in table 6.

Table 6 - Message consumption time results (in seconds)

Target application	Number of messages	N	Minor	Q1	Median	Q3	Bigger	Average	Standard deviation
Spring Boot	150000	10	37	38	39	40	41	39	1,49
Quarkus	150000	10	23	23	23	24,75	27	24	1,49
Spring Boot	250000	10	60	62	66	73	79	68	6,99
Quarkus	250000	10	40	41	41	41,75	42	41,2	0,63
Spring Boot	500000	10	121	125	128	131,5	214	137,5	28,2
Quarkus	500000	10	83	85,5	90,5	93	94	89,3	4,19

Figure 6 - Box-plot diagram of message processing time.



5.2 Hypothesis Test

Statistical tests were performed in order to verify that there is a difference in performance between the *frameworks*. Performance is given through three variables: CPU consumption, RAM consumption and message processing time. The conjugated analysis of the three indicates whether the hypothesis that Quarkus has a superior performance is true. The Wilcoxon test was applied with significance in 0.05 (*p-value* $\leqslant$ 0.05).

It is verified that **the null H** can be discarded since, according to the data presented in section 5.1, Quarkus presents better results. Therefore, it can be affirmed that, considering all the scenarios executed and analyzed, $\mathbf{H_{alt\ is}}$ true to the extent that only the scenario of 150,000 messages in the RAM consumption variable, when statistically analyzing Quarkus, is not superior to Spring Boot, according to Table 7.

Table 7 - Wilcoxon statistical test

Scenario	Statistics	CPU	RAM	Time of Processing
150000	P-value	0,005	0,374	0,005
250000	P-value	0,005	0,009	0,006
500000	P-value	0,006	0,006	0,006

RAM consumption. It has been conjectured that Quarkus has a lower consumption, which means higher performance than the Spring Boot framework. According to Table 7, **the alt H** can be confirmed for the scenarios of 250,000 and 500,000, however, for the scenario of 150,000 this hypothesis, statistically, cannot be confirmed where *the p-value* is 0.374.

CPU consumption. It has been conjectured that Quarkus has a lower CPU consumption. According to Table 7, **null H** can be discarded since in all scenarios, statistically, Quarkus consumes less, so it gets better performance.

Processing time. It has been conjectured that Quarkus has a shorter processing time than Spring Boot. According to Table 7, **null H** can be discarded as in all scenarios, statistically, Quarkus processes faster than Spring Boot, so it gets better performance.

5.3 Discussion

Memory consumption analysis. When analyzing the memory consumption of both applications, it was possible to notice that Quarkus has a significantly lower consumption compared to Spring Boot. However, Quarkus also has a significantly higher consumption variation than Spring Boot, but memory consumption has not increased, even though it has increased the number of messages. It is speculated in the face of the data that Spring Boot presents better resource management, so although it consumes more, there are no major variations as in Quarkus.

CPU consumption analysis. When analyzing the CPU consumption of both applications, it is noted that the two maintained linear consumption, regardless of the size of the batch of messages. As well as it was evidenced that Quarkus had a slightly lower consumption than spring boot. It is speculated that this consumption is best due to the techniques that Quarkus uses to obtain better resource consumption, such as restricted use of Java Reflection.

Analysis of processing time. By analyzing the CPU consumption of both applications, it is noted that Quarkus was able to process messages faster than Spring Boot. A highlight is the high variation observed in a sample with 500,000 messages in Spring Boot, however this variation can be explained by possible parallel, unintentional processing being performed on the machine on which the test was performed.

5.4 Study limitations

It is initial research that seeks to bring a light on the theme, explores new technologies that are still being developed and matured. Therefore, the research has some limitations that should be taken into account. The data collected refers to the scenario and specific settings, any change, even if applied to both, can lead to completely different data.

5.5 Observations

Changing the interaction settings between MongoDB and Quarkus can increase or decrease message consumption by more than 120 times per second. In this sense, there is a lack of documentation on the configurations referring to the connection between the two (especially the Quarkus). The shortage of documentation makes development time consuming and application susceptible to errors.

The compilation of Spring Boot in native mode proved to be extremely time-consuming compared to the normal build, as well as consuming all the machine resources and often causing it to stop due to a lack of RAM.

Because Quarkus is a relatively new framework, the amount of documentation, articles, academic papers, and discussions on the Internet is smaller compared to Spring Boot. This translates into a more time-consuming development and prone to less performative code, since they have not been widely tested.

6 CONCLUSION AND FUTURE WORK

This work carried out an empirical study to analyze the Spring Boot and Quarkus performance in an asynchronous messaging environment. The systems were submitted to a set of tests in order to collect previously selected data. Data such as CPU consumption, RAM consumption, and message processing time were observed and analyzed. The results show that in most tests Quarkus showed better results than Spring Boot.

The results obtained in this study point to the consumption of smaller hardware resources with Quarkus compared to Spring Boot. Thus, the processing time was shorter with Quarkus. It is possible to make decisions based on empirical knowledge when choosing a Java *framework*. It is important to highlight that the conclusions of this study are linked to the context where the tests were performed, and a generalization is not possible.

As future work, we seek to perform: (1) other test scenarios, with other communication protocols; (2) conducting intermittent load tests; (3) considering more parameters for evaluation. This work seeks to increase the existing work in the area of performance analysis between frameworks, so it brought a new scenario with other technologies involved with RabbitMQ. It also serves as an initial study involving Java, GraalVM, Spring Boot, Quarkus, RabbitMQ and MongoDB technologies, providing space for related or deeper studies.

REFERENCES

ALMEIDA, Matheus Santos De. **A Comparative Performance Analysis Between Different Server-Side Web Application Execution Technologies.** Monograph**.** Federal University of São Carlos. (Computer Engineering). San Carlos 2020. Available in: <https://bit.ly/3N8IztG>. Accessed: 30 May 2022.

BACK, Renato Pereira. **Comparative Analysis of Microservices Integration Techniques.** Federal University of Santa Catarina. Florianópolis, 2016. Available in: <https://bit.ly/3N3qxIW>. Accessed: 30 May 2022.

BASILI, V. R. **Software modeling and measurement**: the Goal/Question/Metric paradigm. [S.l.], 1992.

BUONO, Vincenzo; PETROVIC Petar. **Enhance Inter-service Comminication In Supersonic K-Native REST-based Java Microservice Architectures.** Kristianstad

University Sweden, 2021. Available in: <https://bit.ly/3wWshhJ>. Accessed: 30 May 2022.

FARIAS, Kleinner; GARCIA, Alessandro, WHITTLE, Jon; CHAVEZ, Christina von Flach Garcia; LUCENA, Carlos. Evaluating the effort of composing design models: a controlled experiment. **Model Driven Engineering Languages and Systems.** Berlin, pp. 676–691, vol. 7590, 2013. Available in: <https://bit.ly/3PQJNv9>. Accessed: 30 May 2022.

FARIAS, Kleinner; GARCIA, Alessandro, LUCENA, Carlos. Effects of Stability on Model Composition Effort: An Exploratory Study. **Journal on Software and Systems Modeling**, vol. 13, Issue 4, pp. 1473–1494, 2014.

FONG, Fredric; RAED, Mustafa. **Performance comparison of GraalVM, Oracle JDK and OpenJDK for optimization of test suite execution time.** 2021. Available in: <https://bit.ly/3wYwW2t>. Accessed: 30 May 2022.

GUTIERREZ, Felipe. **Pro Spring Boot 2 An Authoritative Guide to Building Microservices, Web and Enterprise Applications, and Best Practices**. Library of Congress Centrol. 2019.

IONESCU, Manuel Valeriu. **The Analysis of the Performance of RabbitMQ and ActiveMQ.** IEEE 2015. Available in: <https://bit.ly/38wo8aX>. Accessed: 30 May 2022.

KOLEOSO, Tayo. **Beginning Quarkus Framework**: Build Cloud-Native Enterprise Java Applications and Microservices. 2020.

LARSSON, Robin. **Evaluation of GraalVM Performance for Java Programs.** 2020. Available in: <https://bit.ly/3LUtcnb>. Accessed: 25 May 2022.

LAZZARI, Luan; FARIAS, Kleinner. **The effects of event-driven architecture on software modularity: An exploratory study.** 2021. Available in: <https://bit.ly/3t7w8Gl>. Accessed: 15 May 2022.

Mohan, Anand. "**Kafka Streaming Application using Java Spring Boot**." (2022). Available in: <https://bit.ly/3lUqDXy>. Accessed: 15 May 2022.

PLESSIS, Shani du; MENDES, Bruno; CORREIA, Noélia. **A Comparative Study of Microservices Frameworks in IoT Deployments**. 2021 International Young Engineers Forum (YEF-ECE), pp. 86-91, 2021. Available in: <https://bit.ly/3GwOXbq>. Accessed: 15 May 2022.

RITZAL, Roman. **Optimizing Java For Serverless Applications.** (Master's thesis) University of Applied Sciences, FH Campus Wien 2020. Available in: <https://bit.ly/3wVRXK4>. Accessed: 30 May 2022.

SHARVARI, T.; Nag, K. Sowmya. **A study on Modern Messaging Systems- Kafka, RabbitMQ and NATS Streaming.** Cornell University, 2019. Available in: <https://arxiv.org/ ABS/1912.03715>. Accessed: 30 May 2022.

ŠIPEK M.; MUHAREMAGIĆ, D.; MIHALJEVIĆ, B.; RADOVAN, A. **Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus**. 43rd International Convention on Information, Communication and Electronic Technology (MIPRO), 2020, pp. 1746-1751, 2020. Available in: <https://bit.ly/3azWLxr>. Accessed: 30 May 2022.

SOMMERVILLE, Ian. **Software Engineering.** 9.ed. Translation Ivan Bosnic and Kalinka G O Gonçalves. São Paulo: Pearson Prentice \hall, 2011.

SOUZA, Ronan de Araújo. **Perfomance Analysis Between Apache Kafka And RabbitMQ**. (Course Completion Work) Federal University of Campina Grande 2020. Available in: <https://bit.ly/3t4y4zA>. Accessed: 30 May 2022.

WOHLIN, Claes, et al. **Experimentation in software engineering**. Springer Science & Business Media, 2012.

YIN, Robert. **Case Study Planning** and Methods 2. ed. Porto Alegre: Bookman, 2001.