Universidade Federal de São Carlos– UFSCar
Centro de Ciências Exatas e de Tecnologia– CCET
Departamento de Computação– DC
Programa de Pós-Graduação em Ciência da Computação– PPGCC

João Victor Lima Lopes

Enhancing Rerun-driven Flaky Test Detection with Container Orchestration

São Carlos
2026

João Victor Lima Lopes

Enhancing Rerun-driven Flaky Test Detection with Container Orchestration

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências Exatas e de Tecnologia da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Engenharia de Software

Supervisor: Prof. Dr. André Takeshi Endo

São Carlos
2026

Lopes, João Victor Lima

Enhancing Rerun-driven Flaky Test Detection with Container Orchestration / João Victor Lima Lopes -- 2026.
81f.

Dissertação (Mestrado) - Universidade Federal de São Carlos, campus Sorocaba, Sorocaba
Orientador (a): André Takeshi Endo
Banca Examinadora: André Takeshi Endo, Auri Marcelo Rizzo Vincenzi, Marco Aurélio Graciotto Silva
Bibliografia

1. Testes Flaky. 2. Orquestração de Containers. 3. Automação de Testes com Kubernetes. I. Lopes, João Victor Lima. II. Título.

Ficha catalográfica desenvolvida pela Secretaria Geral de Informática (SIn)

DADOS FORNECIDOS PELO AUTOR

Bibliotecário responsável: Maria Aparecida de Lourdes Mariano - CRB/8 6979

Folha de Aprovação

Defesa de dissertação de mestrado do(a) candidato(a) João Victor Lima Lopes, realizada em 01/04/2026

Comissão Julgadora

Prof(a) Dr(a) André Takeshi Endo (UFSCar)

Prof(a) Dr(a) Auri Marcelo Rizzo Vincenzi (FEUP)

Prof(a) Dr(a) Marco Aurélio Graciotto Silva (UTFPR-CM)

O relatório de defesa assinado pelos membros da comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação

Acknowledgements

I would like to express my gratitude to everyone who supported me throughout this journey, making the completion of this thesis possible.

To God for the gift of life, for His love, for this moment, and for all the achievements and victories attained so far. There are so many, I can only give thanks.

I am deeply thankful to my family for their unwavering love, encouragement, and belief in me. To my wife, who has been by my side the whole time. To my friends, thank you for your understanding and the moments of joy that helped me throughout this process.

I am deeply grateful to my advisor for his guidance, patience, insights and friendship. I also extend my thanks to the examining committee for dedicating their time and expertise to evaluate this thesis.

Finally, I would like to acknowledge the Federal University of São Carlos (UFSCar) for providing an academic environment and the resources needed to conduct this research. To everyone who contributed, directly or indirectly, to this endeavor.

Resumo

Testes de software são uma atividade fundamental para alcançar altos padrões de qualidade, onde os desenvolvedores dependem de testes automatizados para identificar e corrigir erros de software e validar soluções. No entanto, esses testes às vezes não são confiáveis e apresentam resultados não determinísticos, como passar e falhar ao executar o mesmo código e mesmo teste várias vezes. Esse tipo de teste inconsistente é denominado teste *flaky*. Embora a literatura apresente diversas iniciativas para identificar testes *flaky*, a execução repetida dos testes continua sendo a principal abordagem para depuração e análise da *flakiness* dos testes. Esta dissertação apresenta uma abordagem que aprimora a detecção de testes *flaky* orientada por *rerun*, aplicando orquestração de contêineres, de modo que os testes *flaky* sejam detectados mais rapidamente, com maior taxa de reprodução e de forma mais eficiente. Para isso, utilizamos tecnologias de orquestração de contêineres, como Kubernetes, e outras ferramentas, como Grafana, Prometheus e Loki. A abordagem proposta é implementada em uma ferramenta chamada FlakyTestLab. Avaliamos o FlakyTestLab com vinte e oito projetos de código aberto, sendo quatorze em Java e quatorze em Python. Usando um computador de alto desempenho, o FlakyTestLab executa *reruns*, em média, 76,93% mais rápido do que a abordagem comum e é 300% mais rápido na detecção da primeira falha. O FlakyTestLab melhorou a taxa de *flakiness* em 143,40% e detectou mais testes *flaky* quando comparado com a abordagem comum. Por outro lado, observamos que a flakiness ratio dos testes flaky diminuiu quando o FlakyTestLab empregou mais paralelismo.

Palavras-chave: Automated tests. Flaky tests. Container orchestration. Kubernetes.

Abstract

Software testing is a primary activity to achieve good quality standards, where developers rely on automated tests to identify and fix software bugs and validate solutions. These tests are sometimes unreliable and present non-deterministic results, such as passing and failing when executing the same code and test several times. This type of inconsistent test is called flaky test. While the literature presents many initiatives to identify flaky tests, rerunning the tests several times remains the primary approach to debugging and analyzing test flakiness. This master's thesis presents an approach that enhances rerun-driven flaky test detection by applying container orchestration so that flaky tests are detected faster, and more flaky tests are uncovered. To do so, we employ container orchestration technologies like Kubernetes and other tools, like Grafana, Prometheus, and Loki. The proposed approach is instantiated in a tool called FlakyTestLab. We evaluated FlakyTestLab with twenty-eight open source projects, being fourteen in Java and fourteen in Python. Using a High-End computer, FlakyTestLab performs reruns on average 76.93% faster than a baseline, and is 300% faster than the baseline in finding the first failure. FlakyTestLab detected flaky tests up to 143.40% more than the baseline. On the other hand, we observe that the flakiness ratio decreased when FlakyTestLab employed more parallelism.

Keywords: Automated tests. Flaky tests. Container orchestration. Kubernetes.

List of illustrations

Figure 3.1 – Baseline of current projects. 35
Figure 3.2 – FlakyTestLab Architecture. 37
Figure 3.3 – FlakyTestLab Tool Grafana example. 41
Figure 3.4 – Local host resource consumption metrics. 42
Figure 3.5 – Project file structure. 45
Figure 3.6 – Project file structure after test execution. 46
Figure 4.1 – Average rerun runtime (RQ_1). 56
Figure 4.2 – Total runtime (RQ_1). 59
Figure 4.3 – SpeedUp from 100 runs (s). 60
Figure 4.4 – Time to first Failure from all tests (s). 61
Figure 4.5 – Time to first Failure from tests that failed in all configurations (s). . . . 61
Figure 4.6 – First failure speedup in relation to Naive (Only for tests that failed in all configurations). 62
Figure 4.7 – Quantity of Flaky tests found in each configuration. 63
Figure 4.8 – Flaky Ratio to all tests. 66
Figure 4.9 – Flaky Ratio to tests that failed in all configuration. 66

List of tables

Table 4.1 – Projects selected to validate the tool FlakyTestLab. 50
Table 4.2 – Average rerun runtime in seconds (RQ_1). 55
Table 4.3 – Total runtime (for 100 reruns), in seconds (RQ_1). 57
Table 4.4 – Comparison of total time spent to execute one hundred runs between Naive and P16. 58
Table 4.5 – Average time to first failure to all tests and to tests that failed in all configurations. 62
Table 4.6 – Percentage of projects that flaky tests were detected. 64
Table 4.7 – Number of flaky tests reported per project. 65

Summary

1	**INTRODUCTION**	**17**
2	**BACKGROUND**	**21**
2.1	**Software Testing**	**21**
2.2	**Flaky Tests**	**24**
2.3	**DevOps**	**25**
2.3.1	Agile Methodologies	25
2.3.2	DevOps Culture	25
2.3.3	Continuous Integration (CI) / Continuous Delivery (CD)	27
2.4	**Docker**	**27**
2.4.1	Container Orchestration	28
2.5	**Monitoring**	**29**
2.6	**Related Work**	**30**
2.7	**Concluding Remarks**	**34**
3	**TOOL AND APPROACH**	**35**
3.1	**FlakyTestLab**	**35**
3.1.1	Components	40
3.1.2	Monitoring Components	41
3.2	**Implementation and Usage**	**43**
3.3	**The Naive Approach**	**47**
3.4	**Concluding Remarks**	**47**
4	**EVALUATION**	**49**
4.1	**Experimental Settings**	**49**
4.1.1	Selected projects	50
4.1.2	Hardware used	50

4.1.3 Metrics collected for each RQ 51
4.1.4 Configuration definition for projects 51
4.1.5 How the metrics were collected 52
4.1.6 Replication package 52
4.2 Analysis of Results 54
4.2.1 RQ_1: To what extent does FlakyTestLab accelerate test reruns? 54
4.2.2 RQ_2: To what extent can FlakyTestLab detect the first failure of a flaky test? 60
4.2.3 RQ_3: To what extent does FlakyTestLab enhance the manifestation of flaky tests? 63
4.3 Discussion 67
4.4 Concluding Remarks 69

5 CONCLUSION 71
5.1 Future Work 71

REFERENCES 75

Chapter 1

Introduction

In recent decades, software systems have been used in a wide range of fields of knowledge, leading to an increasing demand for quality, productivity, and efficiency (BARBOSA et al., 2000; COELHO et al., 2025). To meet this demand, Software Engineering has been widely adopted. Pressman and Maxim (2021) define software engineering as the application of a systematic, disciplined, and quantifiable approach to the development, operation, and maintenance of software.

One of the purposes of software engineering is to build systems with quality and reliability. Software testing is considered an essential activity in software development. In general terms, testing corresponds to observing the execution of a software system in order to validate whether it behaves as intended and to identify potential bugs (BERTOLINO, 2007). Kaner, Falk and Nguyen (1999) argue that finding bugs is the core of the work of testers who seek to validate the quality of their systems.

Testers should seek to find as many bugs as possible, however, the goal should not be to find all of them, as doing so is costly, frustrating, and does not provide proportional value. Moreover, it is impossible to test a system completely for three reasons: the domain of possible inputs is too large to be exhaustively tested; there are too many possible execution paths in the program to be tested; and user interface issues (and, therefore, design problems) are too complex to be tested comprehensively (BERTOLINO, 2007).

Wong et al. (2016) argue that bug localization in software systems is one of the most - if not the most - costly, tedious, and time-consuming activities in software debugging. Bertolino (2007) state that testing costs may reach up to 50% of the total cost of a software system. This scenario creates a demand for techniques and approaches aimed at minimizing such efforts and costs. Automation is one of the approaches adopted, as manual test execution is a factor that increases both software development costs and delivery

time (GAROUSI; ELBERZHAGER, 2016; CRISPIN; GREGORY, 2009). Bernardo and Kon (2008) also argue that executing tests in an automated manner provides greater confidence when making any changes to the code and allows tests to be executed exhaustively and consistently, facilitating the automatic detection of faults, guiding developers in bug localization, and making the process more agile and less dependent on human intervention. The use of *Continuous Integration* (CI) tools, as well as other tools and practices from the DevOps culture, is also common (EBERT et al., 2016).

CI tools integrate developers' work more rapidly and more frequently. They also allow more costly stages, such as software testing, to be incorporated and executed automatically through an integration pipeline. In this way, the system is continuously tested, enabling software development with higher quality and lower cost (EBERT et al., 2016).

In addition to CI, a range of tools (such as containers) and practices are employed to enhance agility (e.g., agile methodologies), ensure quality (e.g., automated testing), and improve cost-efficiency in the development process. A container is a software unit that packages code and its dependencies, ensuring consistent execution across any environment (MERKEL, 2014). To manage this complexity at scale, container orchestration is used: a process that automates the deployment, networking, and scalability of these units (BURNS et al., 2016). Currently, Docker stands out in container creation, while Kubernetes is the leading technology for orchestration (BERNSTEIN, 2014), followed by solutions such as Docker Swarm and Amazon ECS. This set of practices and tools is grounded in the DevOps culture.

DevOps culture seeks to mitigate organizational issues through the integration of development and operations teams. This integration enables efficient deliveries, thereby facilitating a lean and fluid connection between these traditionally separated domains. It also enables the development of agile, flexible, secure, and high-quality software, as tests can be executed more easily, rapidly, and frequently. (KIM et al., 2021).

Parry et al. (2021) argue that developers rely on testing to identify and fix issues in systems, as well as to validate solutions. However, if the tests employed are unreliable and produce non-deterministic results, they can be considered inconsistent and are commonly referred to as *flaky tests*. Flaky tests typically manifest when, even without any changes to the system code or the test code, the outcome of a test execution varies - passing in some runs and failing in others. Lam et al. (2019) state that flaky tests are difficult to identify and fix precisely because they are inconsistent, non-deterministic, and infrequent (often a flaky test only becomes evident after executing the tests hundreds or thousands of times), thereby generating high cost and effort for identifying and correcting the bug causing the test failure.

Given the growing importance of the flaky test topic, several studies have been conducted to identify flaky tests (BELL; KAISER, 2014; BELL et al., 2018; LAM et al., 2019) and to understand their lifecycle and impact on the industry (ECK et al., 2019;

LAM et al., 2020). Other works have aimed to improve reproducibility and fault detection (RAHMAN et al., 2024; CORDEIRO et al., 2021; MORÁN et al., 2020), automate the execution and detection of flaky tests (GRUBER; FRASER, 2023b), as well as enhance the identification and resolution of flaky tests (CORDY et al., 2022; RAHMAN; SHI, 2024).

The most common approaches to accelerating flaky test identification revolve around re-running (rerun) tests until the failure reoccurs, or introducing some form of perturbation to expose the flaky behavior with fewer executions (LAM et al., 2020). Many reruns may be needed to obtain a witness run for a flaky test (ALSHAMMARI et al., 2021). Reruns may also be needed for debugging flaky tests and the effort of re-execution must be balanced, as the return on investment from generating hundreds of failures for diagnosis is marginal (GRUBER; FRASER, 2023a).

Many of these approaches (LAM et al., 2019; BELL; KAISER, 2014; BELL et al., 2018; CORDEIRO et al., 2021; CORDY et al., 2022) focus on specific programming languages, such as Java and Python, which limits their applicability to other languages. These approaches also tend not to leverage widely adopted industry tools, such as CI (which automates and manages testing and software delivery tasks), Containers (which provides system isolation and resource/access control, potentially reducing certain types of flaky tests or aiding in their identification), and Kubernetes (which allows orchestration of large groups of containers to organize tests and/or create diverse scenarios and environments for test execution).

Some initiatives explore the use of containers in the context of flaky tests: Tools designed to perform fuzzing in CI environments, such as Shaker (CORDEIRO et al., 2021), aim to improve efficiency in the identification of flaky tests by introducing disturbances that compete for CPU or memory resources; present ideas on how to leverage containers, proposing a container–based infrastructure capable of creating a set of clusters with distinct hardware configurations, however, no implementation is provided (TERRAGNI; SALZA; FERRUCCI, 2020); empirical study on the effects of computation resources in flaky tests, demonstrating that the availability of such resources can directly influence test behavior (SILVA et al., 2024).

Nevertheless, to our best knowledge, no work has explored the adoption of container orchestration to improve flaky test detection. This thesis introduces an approach that enhances rerun-based flaky test detection through the application of container orchestration. To implement this approach, FlakyTestLab tool is presented as an alternative for creating distinct environments that can be used to run tests in general in order to improve the detection of flaky tests. The tool facilitates running the same set of tests under different CPU and RAM configurations, and also enables test parallelism and isolation.

The focus of this work was limited to evaluating, within FlakyTestLab, the impacts of parallel test execution and the behavior of known flaky tests across 28 open source

projects, including 14 Java projects and 14 Python projects. For all these projects, we focused on three main research questions (RQs) to achieve our evaluation goals: RQ_1 - To what extent does FlakyTestLab accelerate test reruns?; RQ_2 - To what extent can FlakyTestLab detect the first failure of a flaky test?; and RQ_3 - To what extent does FlakyTestLab enhance the manifestation of flaky tests?.

A set of experiments was conducted using standard executions, that is, without the use of the tool, referred to as Naive, and the results were compared with those obtained from the experiments executed using the proposed tool.

In summary, the contributions of this thesis are as follows:

- **Approach:** A different approach to the execution and detection of flaky tests is proposed, in which container orchestration, together with supporting tools, is employed to create environments with distinct characteristics, thereby increasing the likelihood of flaky test manifestation;
- **Software:** The FlakyTestLab tool was developed and made available to the community, enabling the execution of various combinations of tests, across different programming languages and for different purposes;
- **Tool Evaluation:** A study was conducted to evaluate the developed tool, demonstrating that it enhanced the detection of flaky tests for almost all evaluated projects, with significant gains both in the number of flaky tests detected and in the time required for detection.

The remainder of this thesis is structured as follows. Chapter 2 provides the necessary background on Docker, Kubernetes, and container orchestration, discusses the importance of monitoring in cluster environments, and presents the related work, providing a comprehensive bibliographic review of studies on similar themes. To address the challenges of test instability, Chapter 3 details the design of FlakyTestLab, explaining how these background components are leveraged to create a tool capable of executing multiple containers under diverse configurations, this chapter also describes the core framework and its specific monitoring components, details the tool's development, and introduces the Naive aproach, which serves as a baseline for performance comparison. In Chapter 4, we present three RQs to evaluate the tool, along with the experimental settings, the replication package, a detailed analysis of the results, and an interpretation of the findings. Finally, Chapter 5 summarize our contributions and outline potential avenues for future work.

Chapter 2

Background

This chapter presents fundamental concepts necessary for understanding this work. Section 2.1 explains basic concepts of Software Testing. Section 2.2 addresses Flaky Tests, which are the central focus of this thesis. Section 2.3 discusses the DevOps culture. Section 2.4 explains basic concepts of container and Docker. Section 2.5 presents some characteristics of observability. Finally, Section 2.6 presents the related work.

2.1 Software Testing

Software Testing is a verification and validation process aimed at determining whether a software system meets business needs, in addition to verifying the technical requirements that guided its design and development (PRESSMAN; MAXIM, 2021). It also validates whether the behavior of the software aligns with expectations, identifies bugs, and categorizes them according to severity levels so that they can be addressed and corrected (MYERS et al., 2004).

Wong et al. (2016) suggest that, regardless of the type or criticality, when a bug is identified, it is also necessary to determine its location and the reason it occurs. According to the authors, bug localization can be a very costly process, requiring substantial time, numerous people, and multiple tools. Furthermore, the effective use of all these resources still depends on the developer's knowledge of the software, their capacity for logical reasoning, and their experience.

Even though the bug localization stage is very costly, it is not the only one. Practically all stages of testing are expensive, especially when aiming to achieve a software system with good test coverage (AMMANN; OFFUTT, 2017). This is mainly due to the repetitive and time-consuming work caused by frequent testing. When working with Agile

Development Methodologies, which guide developers to make small and more frequent deliveries, costs can increase due to the need to execute the test suite more often, as it is considered a best practice to test the software whenever a change occurs (CRISPIN; GREGORY, 2009).

To guide the Software Testing process, a set of Test Cases can be defined. This set is derived from project requirements, design artifacts, or source code (AMMANN; OFFUTT, 2017). Test cases serve as the basis for software evaluation. They consist of a set of test inputs, execution conditions, and expected outputs. Test cases are developed for a specific purpose, such as exercising a particular path in a program or verifying a specific requirement. In some systems, a test case may be a sequence of events or actions to be applied and/or observed (IEEE, 1990). A test case fails when one of the assertions evaluates to false. If all assertions are validated without failure, the test is considered to have passed.

Bernardo and Kon (2008) illustrate a very common scenario in which a developer needs to create a solution, typically by studying the problem, devising a solution, and then implementing it. After these steps, some manual tests are conducted to observe the behavior of the solution, during which it is common to detect bugs. Upon detecting these bugs, the developer must correct them and then repeat the test suite. Crispin and Gregory (2009) point out that executing a test manually after a system change becomes costly and consumes a significant amount of the developer's time, generating additional expenses for the company because it is a repetitive and time-consuming process.

Bernardo and Kon (2008) point out that performing tests in an automated manner provides greater confidence when making any changes to the code, as working with automated tests allows for the creation of more elaborate, complex, and comprehensive test cases that can be executed exhaustively and consistently. To execute automated tests, it is necessary to have a framework that enables the systematic execution of a set of tests. This allows the time invested in bug identification to be better utilized, as a test can be written once and executed countless times automatically, requiring only periodic maintenance. Writing tests, rather than merely executing them manually, also provides a historical record, where small details that might otherwise be forgotten are documented, preventing errors and delays in bug identification.

Ammann and Offutt (2017) state that, although most types of tests can only be executed after some portion of the software has been implemented, the planning and construction of tests should occur at different stages and in parallel with software development. Dias-Neto (2007) demonstrates that bugs can be introduced during the transformation of information between the different phases of the software development lifecycle. As these phases such as requirements elicitation, architecture definition, and system planning are undertaken, a series of transformations may introduce bugs into the system. Due to this sequence of transformations, it is necessary to perform testing at different levels, aiming

to evaluate the software from various perspectives at each stage of the cycle. Regression Testing is an important strategy for reducing “side effects”. With each new version of the software or each iteration, the entire test suite is reapplied, verifying not only the recently modified components but the system as a whole. This strategy can be applied at any testing level. According to the author, there are four main levels of software testing:

- Unit Testing: examines the smallest unit of the project, looking for bugs caused by logic or implementation defects;
- Integration Testing: identifies bugs associated with interfaces between modules when they are integrated to build the complete software structure;
- System Testing: evaluates the software for bugs through its actual use, as an end user would. Tests are conducted in the same environment, under the same conditions, and with the same input data that a user would utilize, verifying whether the product meets its requirements; and
- Acceptance Testing: performed by a restricted group of end users of the system. These users simulate routine system operations to verify whether its behavior aligns with the specified requirements.

A simple “test” can be written alongside the main code of a system to validate whether that main code, or a portion of it, behaves as expected. In Listing 2.1, an example of a test is shown, with the purpose of ensuring that a service and its replica return the same response to a specific message. On line 6, there is an assertion (Assert) that compares the response returned by the first service (line 4) with the response returned by the replica (line 5) (LAM et al., 2019).

Listing 2.1 – Example of an automated test.

```
1  [TestMethod]
2  public void TestReplicaService() {
3     // ...
4     byte[] response = Service.SendAndGetResponse(pload);
5     byte[] replicaResponse = ServiceReplica.SendAndGetResponse(pload);
6     Assert.AreEqual(response, replicaResponse);
7  }
8
9  public class Service : NetworkService {
10    public byte[] SendAndGetResponse(Request req) {
11       // ...
12       DateTime currentTime = DateTime.UtcNow;
13       Message message = new Message(req, currentTime);
14       return base.SendAndGet(message.Serialize());
15    }
16 }
```

The implementation and maintenance stages of Software Testing face several challenges, one of which is the occurrence of Flaky Tests. According to Parry et al. (2021), Flaky

Tests are tests that fail inconsistently and unpredictably, even when there are no changes to the system code or the tests themselves. Flaky tests are difficult to detect and do not provide a clear indication of their presence or location. Therefore, like any other bug, they limit the reliability of both the test suites and the system itself, as it is not possible to determine with certainty whether the failure is due to the test, the system, both, or neither.

2.2 Flaky Tests

Software development represents an ongoing challenge in which problems inherent to implementation objectives must be addressed while simultaneously ensuring that the proposed solutions do not introduce new issues. Parry et al. (2021) argue that software developers rely on testing to identify and fix problems in systems, as well as to assess whether their solution is sufficient to achieve a given objective. However, if the tests used to ensure the quality of the solution are unreliable and produce non-deterministic results - for example, passing in some runs and failing in others when executing the same code and the same test multiple times - these tests can be considered inconsistent and are commonly referred to as Flaky Tests.

According to Lam et al. (2019), flaky tests are difficult to identify and fix precisely because they are inconsistent, non-deterministic, and occur only occasionally. As a result, they are difficult to reproduce in a development environment, generating high costs for identifying and correcting the bug that causes the test failure. Many of the bugs responsible for these failures may arise from factors external to the code and tests, such as concurrency, parallelization, and network or database dependencies. Despite the high cost of identifying and fixing flaky tests, the authors state that ignoring these failures can be dangerous, as they may represent real defects, and allowing them to reach a production environment can be highly damaging to the business.

Luo et al. (2014) present an empirical study in which they categorize the most prominent root causes of flaky tests, identifying asynchronous waits, concurrency, and order dependencies as the three most common. Ways to amplify the manifestation of flaky tests have been demonstrated, such as modifying delay durations in asynchronous wait calls. The most common practices used by developers to address flaky tests have also been described, including performing a general environment cleanup before executing a test suite to mitigate issues caused by order-dependent calls. Eck et al. (2019) introduce four new categories of flaky tests, compare them with the work of Luo et al. (2014), and report, for the first time, that flaky tests can be caused by code already deployed in production.

Lam et al. (2019) identify several common root causes of flaky tests, including: improper use of time-handling APIs, which can cause failures in situations such as time zone changes; the use of random numbers in parts of the test, as the impact of each possible

number cannot be predicted; deadlocks and randomness in thread usage when testing a multithreaded system; and resource leaks, which occur when a system passes or fails due to inadequate resource management, such as excessive memory usage or file locks.

Returning to Listing 2.1, it can be observed that the assertion on line 6 (Assert) occasionally fails, making this a flaky test case. Failures may occur because the calls to Service and ServiceReplica (SendAndGetResponse) may or may not use the same timestamps. If the SendAndGetResponse calls on lines 4 and 5 are executed within a short time interval, the timestamps produced by DateTime.UtcNow on line 11 may be identical due to the limited granularity of the system timer, which is seconds by default. If the calls are executed without any interference, allowing the timestamps to match, the test passes; otherwise, if any interference causes a delay in the replica call, the test fails (LAM et al., 2019).

2.3 DevOps

2.3.1 Agile Methodologies

One of the main objectives of a software development team is to deliver a coherent, fast, and high-quality product. However, achieving this is always challenging, as it involves variables such as time and cost. In the early stages of software development organizations, traditional development methodologies derived from the manufacturing process were used, in which a cycle involved adding raw materials and a design, resulting in a fully finished product. A traditional methodology based on this idea is the Waterfall model (PRESSMAN; MAXIM, 2021).

In an effort to improve the development process and mitigate issues related to delays, management, and value, the Agile Manifesto was created (FOWLER; HIGHSMITH et al., 2001). In 2001, a group of developers launched this manifesto with the goal of establishing a methodology that would deliver value quickly and continuously to clients. This was achieved through small increments, which are planned periodically and developed according to current needs.

2.3.2 DevOps Culture

The idea of the DevOps culture originated at the Agile 2008 conference, an event on agile methodologies, where Patrick Debois and Andrew Schafer gave a presentation on Agile Infrastructure titled "same flour, same sack". The first mention of Dev and Ops occurred during the O'Reilly Velocity conference in 2009, where John Allspaw and Paul Hammond presented the seminar "10 Deployments per Day: Dev and Ops Cooperation at Flickr". Later in 2009, building on the idea from John and Paul's presentation, Debois

organized the first DevOpsDays in Ghent, Belgium, where the term DevOps was coined (KIM et al., 2021).

The idea and need to establish a new organizational culture for development and delivery arise from problems caused by the historical separation of Operations (Ops) and Development (Dev) teams. The Ops team, responsible for maintaining the fundamental infrastructure of applications, such as servers and networks, typically seeks to keep this infrastructure running continuously, minimizing changes that could cause instability. On the other hand, development teams aim to deliver the work completed in the latest development cycle, integrating changes into the infrastructure. This creates a conflict between the interests of the two teams. However, it is imperative that both teams work cohesively. After all, for development teams to be truly agile, the development flow must be fully agile, from project development to the final system delivery. Without this integration, agile development teams face a barrier in integrating their deliveries into the infrastructure, which negatively impacts team velocity and associated metrics (ZHU; BASS; CHAMPLIN-SCHARFF, 2016).

The DevOps culture aims to guide the joint work of Dev and Ops teams and reduce the distinctions and impacts of these roles, effectively making everyone both Developers (Dev) and Operators (Ops) of their systems. It enables teams to manage all stages, from project development to delivery, making them truly agile. The DevOps culture is characterized by five pillars, known as CALMS (LOUKIDES, 2012):

- **C**ulture: focuses on interpersonal relationships and team interactions. It is about people and their mindsets. The goal is to maintain a constant commitment to delivery and product quality, and to take responsibility for the production environment (the production environment refers to the system that serves the end customers, i.e., the most critical environment). Therefore, it relates to how professionals interact, think, behave, and perform their daily tasks. This unifies the organizational culture of development and operations, eliminating separation - everything becomes a single flow, from conception to deployment and subsequent maintenance.

- **A**utomation: DevOps is a culture, not a process, but processes are embedded within the culture, where the focus is on automating as many steps as possible, minimizing human errors and rework, generating documentation, and standardizing each step. At this stage, the emphasis is on pipelines, CI, and Continuous Delivery (CD), as well as other processes such as infrastructure-as-code generation and configurations that were previously done manually but are now managed through code-based tools.

- **Le**anIt (Lean methodology applied to IT): Lean thinking helps achieve more precise planning by using only the steps and processes that are truly necessary. This reduces the complexity of structures and systems, making it easier to identify problems,

bottlenecks, areas for improvement, costly steps, rework, or delays, thereby optimizing these flows and enabling faster delivery, greater efficiency, and higher quality. Continuous learning and improvement are integral to the process.

- **M**easurement: Monitoring the infrastructure and systems, generating fast and accurate feedback, is essential for understanding what is truly happening in the environments. In addition to creating knowledge, it provides predictability regarding potential incidents or disasters that may arise. With logs and dashboards, there is sufficient data to analyze failures and continuously implement improvements. Monitoring occurs at all levels, from the fundamental infrastructure - such as servers, networks, and databases - to higher layers, including user behavior and business rules; and

- **S**haring: Sharing information and responsibilities helps decentralize tasks, distributing them across teams and facilitating integration and collaboration, which is the main objective of the culture. This enables teams to operate as DevOps, prevents conflicts and dependencies, levels the teams, and allows for faster and more effective implementations and fixes, which benefits the organization as a whole.

2.3.3 Continuous Integration (CI) / Continuous Delivery (CD)

Continuous Integration (CI) enables the automation of the process of merging a change into an existing system. This merge essentially involves building a new system with the integrated components. This process allows for the early detection of errors, code conflicts, and also evaluates the overall quality of the change. For these reasons, CI is widely adopted in the industry and in open-source projects. Its evolution, Continuous Delivery (CD), goes beyond code integration by also facilitating the deployment of changes directly to environments, automating not only the construction of the new system but also its delivery to the end user (ZAMPETTI et al., 2021).

According to Zampetti et al. (2021), both processes are important, especially when applying agile and DevOps concepts, as development increasingly relies on shorter cycles, automated tests, validations, and automation for every system change. Therefore, having a process that allows for the organization and systematic application of continuous value delivery to the client is extremely useful, as it facilitates and enables the consistent application of these practices. Furthermore, the authors note that the use of CI/CD allows for early defect detection, increased developer productivity, and accelerated delivery cycles.

2.4 Docker

Docker is a container-based virtualization technology, which can be considered an evolution of virtual machines (VMs), yet lighter and more efficient. When a Docker container

is created, calls are made to the Linux Kernel to generate an isolated environment - a sort of “box” - with defined attributes, such as access to specific file systems, CPU, memory, and network. Additionally, containers operate within a process namespace, ensuring that execution is both isolated and controlled.

The main distinction of Docker compared to traditional virtualization lies in its resource efficiency. While VMs virtualize an entire system, including the operating system and physical hardware, Docker shares the host system's kernel, eliminating the need to duplicate operating system resources. This significantly reduces resource overhead, resulting in a leaner and faster solution.

This approach makes Docker a highly effective alternative in terms of performance, especially in scenarios where traditional virtualization may present limitations. As a result, Docker has become increasingly popular in development, automation, and infrastructure-as-code environments (ANDERSON, 2015).

2.4.1 Container Orchestration

Container orchestration refers to the process of automating and managing multiple containers at scale, whether Docker or not, ensuring they operate efficiently, resiliently, and at scale. Instead of managing containers manually, orchestration allows for the automation of tasks such as provisioning, deployment, scalability, monitoring, and failure management. Several popular orchestration tools offer advanced features, such as load balancing, auto-scaling, and fault recovery, while also facilitating continuous deployment and application integration in cloud or data center environments (TURNBULL, 2014).

Kubernetes is one such open-source container orchestration tool (Docker-based or otherwise), and it is the most popular and widely used. It manages systems running in containers distributed across multiple hosts. It enables the execution, administration, scaling, and monitoring of various containerized systems. Originally created by Google, it was donated to the Cloud Native Computing Foundation (CNCF) in March 2016. Kubernetes is an essential tool for managing production workloads (BAIER, 2017).

Kubernetes allows users to declare a desired state for a system using concepts such as deployments and services. For instance, a user can specify the need for a deployment with three instances of a web system. Kubernetes then initiates these three instances, which, in the Kubernetes context, are referred to as **pods** (each pod is considered the smallest unit of a system in Kubernetes, and can contain one or more containers). Kubernetes continuously monitors these pods, providing automatic restarts, rescheduling and replication to ensure that the system remains in the desired state (BAIER, 2017).

Kind[1] is a tool for running local Kubernetes clusters using a single “host” as the node for the pods. Kind was primarily designed for testing Kubernetes itself, but it can also be used for local development or continuous integration (Kubernetes SIGs, 2026).

[1] <https://kind.sigs.k8s.io/>

Helm[2] is a package management tool for Kubernetes, designed to simplify the installation, upgrading and maintenance of applications and services in Kubernetes clusters (Helm Project, 2023). Similar to Apt or Yum in Linux systems, Helm enables the distribution and management of packages (known as charts) that contain the configurations and resources necessary to deploy a project. It facilitates the deployment of complex applications, such as databases, web servers or monitoring tools, by allowing users to install, configure and manage these services in a standardized and efficient way, without the need to manually write and manage all the Kubernetes YAML manifests (Kubernetes Authors, 2023a).

2.5 Monitoring

Monitoring is the process of collecting, analyzing, and visualizing data about the health and performance of systems, applications, and IT infrastructure (BEYER et al., 2016). Tools like Prometheus[3] and Grafana[4] have become industry standards for monitoring modern environments, particularly in distributed systems, microservices and orchestrators like Kubernetes (BRAZIL, 2018).

Prometheus is an open-source monitoring and alerting tool that collects metrics from systems, services and applications. It uses a time-series data model, where data is stored with timestamps, allowing for tracking the behavior of systems over time. Prometheus is particularly popular in container and Kubernetes environments due to its easy integration with these ecosystems (BRAZIL, 2018).

Grafana, on the other hand, is a platform for visualizing and analyzing metrics. It is widely used alongside Prometheus to display the collected metrics in an intuitive manner, creating interactive dashboards and visually appealing alerts (BRAZIL, 2018). It allows the creation of interactive panels and customized dashboards that connect to various data sources, such as Prometheus. It is frequently used to display performance metrics, logs and other real-time information, offering a rich and easy-to-configure interface for visualizing complex data. It supports integration with a wide variety of data sources, creating interactive visualizations like graphs, tables, maps, and also supports alerts to notify users about significant events or state changes (Grafana Authors, 2023a).

Vector[5] is an open-source tool for log and metrics aggregation and routing. It is designed to be highly efficient, scalable, and easy to configure. The platform unifies the ingestion and management of logs, metrics, and traces, enabling data to be collected from multiple sources, processed, and forwarded to monitoring and analysis systems such as Datadog, Prometheus, Elasticsearch, Grafana Loki, or local text files. Vector supports

[2] <https://helm.sh/>
[3] <https://prometheus.io/>
[4] <https://grafana.com/>
[5] <https://vector.dev/>

real-time filtering, transformation, and routing of data, facilitating integration within modern microservices-based architectures (Vector, 2026).

Grafana Loki[6] is a log storage and visualization solution developed by the same team behind Grafana. Just as Prometheus is used for metrics, Loki is used for logs. It is designed to be highly scalable, user-friendly, and efficient in storing large volumes of logs, employing an architecture based on labels and log streams. The main advantage of Loki is its native integration with Grafana, providing a seamless experience for visualizing both metrics and logs within the same dashboard (Grafana Labs, 2026).

Grafana Loki is optimized to work in container and microservices environments, especially when integrated with Prometheus, to provide a complete observability solution. With Loki, it is possible to query logs through Grafana, using the same set of metrics and labels, which facilitates the correlation between logs and metrics within a single interface (Grafana Authors, 2023b).

Metrics Server is a key component within Kubernetes, responsible for collecting and providing resource utilization metrics, such as CPU and memory, to the cluster's management and scaling system. It collects these metrics from each pod and node in the cluster (in this project, the single host) and makes them available to other Kubernetes tools and functionalities, such as the Horizontal Pod Autoscaler (HPA), which automatically adjusts the number of pod replicas based on resource demand, and to Prometheus. The integration of the Metrics Server with Kubernetes facilitates efficient cluster resource management and enhances the performance of distributed applications (Kubernetes Authors, 2023b).

2.6 Related Work

Leinen, Perathoner and Pretschner (2024) investigate the hypothesis that computational resource scarcity, specifically CPU resource exhaustion, is a determining factor for the occurrence of flaky tests. Through the analysis of a dataset containing 232 user interface (UI) test cases for macOS across 20 open-source projects, the study reveals that, although flaky test executions spend significantly more time at the CPU limit than stable tests, surprisingly, failures in these tests occur less frequently under maximum CPU stress than their successful executions, suggesting that server resource management can mitigate, but not completely eliminate, flakiness.

Iqbal, Begum and Sakib (2025) present a Machine Learning model (Random Forest) to identify order-dependent (OD) flaky tests before exhaustive re-executions are performed. The study demonstrates that the technique is capable of reducing the detection effort by up to 82% in terms of the number of re-executions required, compared to traditional isolation approaches. Furthermore, the tool achieved an accuracy of 0.92 in identifying

[6] <https://grafana.com/oss/loki/>

OD tests, enabling critical intermittent failures to be prioritized and addressed earlier in the CI pipeline, thus saving computational time.

Cordeiro et al. (2021) developed the *Shaker* tool, which aims to provide greater efficiency in identifying flaky tests related to concurrency issues, such as async/await. *Shaker* introduces disturbances that compete for CPU or memory resources while the test suite is rerun. These disturbances increase the stress on these resources, which in turn increases the likelihood of revealing flaky tests caused by concurrency and parallelism problems. The idea behind adding these disturbances comes from the observation that simultaneity (or concurrency, when distinct processes attempt to use the same resource simultaneously) is a significant cause of instability in tests. By introducing stress into the test environment, the tool can cause unexpected interruptions and changes in the order of events, potentially influencing test results and revealing flaky tests that might otherwise remain hidden or require more reruns to be detected. Therefore, *Shaker* enables the identification of flaky tests with fewer reruns compared to the most common market approach, ReRun, which essentially involves rerunning tests numerous times to find inconsistencies. However, this tool is currently available only for Python and Java projects.

Terragni, Salza and Ferrucci (2020) propose a container-based infrastructure capable of creating a set of clusters with distinct hardware configurations (CPU, memory, and network), where tests could be executed automatically to discover the root causes of flaky tests. Their approach involves using a Fuzzy Orchestrator to perform controlled test executions along with noise injection based on fuzzy logic. The authors also present a potential method for identifying the root causes of flaky tests using the results obtained from these executions. However, the work was presented as a short paper/new idea, was not implemented, and does not include any experimental results.

Silva et al. (2024) present a study on how the variation of computational resources, such as CPU and RAM, influences the manifestation of flaky tests. A total of 52 projects with known flaky tests were analyzed under various resource-availability configuration. They found that resource reduction not only increases the failure rate but also detected some that were not previously reported in research literature. The study concludes that flakiness detection is highly sensitive to the environment, demonstrating that tests that appear stable on powerful machines may systematically fail in more modest CI environments. The authors further demonstrate that, by controlling the availability of computational resources, developers can reduce the likelihood of flaky failures or, alternatively, increase it when desired. The study was conducted using virtual machines, with one instance dedicated to each test execution. All machines were provisioned with the same amount of computational resources (4CPU and 16GiB of RAM). The test suites were executed in Docker containers, which were also used to control the amount of available memory and/or CPU. Additional tools were employed to regulate access to the network and disk. The study does not provide a specific tool; instead, it adopts a conventional

approach in which tests are re-executed (300 times) while varying the available resources. Several analyses are presented, and methods are suggested for analyzing and determining the appropriate amount of resources for each system.

Gruber and Fraser (2023a) investigate the application of SBFL to diagnose the root cause of flaky tests in industrial systems. The study focuses on multiple executions (reruns) to generate a reliable spectrum of failures, given that flaky tests fail intermittently. The authors conducted experiments with 46 flaky tests from the open-source OpenSearch project and found that effectiveness quickly plateaus: collecting more than 5 to 10 failure 'witnesses' does not yield significant gains in accuracy. The authors concluded that, in an industrial context, the effort of re-execution must be balanced, as the return on investment from generating hundreds of failures for diagnosis is marginal.

An empirical study investigated developers' perceptions regarding flaky tests: how they deal with the causes, their impact on workflows, how they fix them, and what they consider to be the biggest challenges. It was identified that flaky tests can cause adjacent effects that are not mentioned in other projects, such as organizational problems related to resource allocation, whether computational or human. Regarding the effort to resolve these failures, prepared environments, execution time in delivery queues, availability of environments for test execution (as there is significant repetition in test execution, which can interfere with other teams or tasks), and skilled personnel with more experience who typically have limited time—are required, thus resulting in high costs for companies (ECK et al., 2019).

Lam et al. (2020) present a characterization of the life cycle of flaky tests as well as their impacts. According to them, flaky tests impact both industry and research. In recent years, several impact reports have emerged, such as one from Google, where 1.5% of the 4.2 million individual tests fail, and of these, 16% can be classified as flaky. The authors also analyzed six major proprietary projects at Microsoft, which provided them with a dataset obtained through the CI/CD tool, CloudBuild. It was possible to identify that, even in projects with many flaky tests, their development workflows are not necessarily impacted. For example, a project with 1,400 flaky tests had only 0.4% of its builds fail due to flaky tests. On the other hand, a project with 176 flaky tests experienced more than 35% build failures. Another interesting point is that, just like in open-source projects, the main causes of flaky tests in proprietary projects are waits in asynchronous calls. This led to the development of the FaTB (Flakiness and Time Balancer) tool, which helps perform the same tests up to 78% faster. This is possible because the tool detects the minimum wait time without flaky tests occurring and configures test execution with this time. Typically, test slowness happens when flaky tests are caused by asynchronous calls; sometimes (31%), the developer simply increases the wait time for these calls or the method's timeout.

Bell and Kaiser (2014) present an approach to mitigate failures in unit tests due to order issues. The problem occurs when a test either fails or passes simply by altering the order in which it is executed relative to other tests. This typically happens due to some shared resource that is either affected or not by the previous test. The authors propose running test cases in isolation, each within its own process, thus preventing one test from inadvertently affecting another. To demonstrate this approach, the authors developed the VmVm tool, which reinitializes classes containing static attributes, as these can facilitate interference with other processes. Reinitializing the classes establishes a new runtime state for the test, thereby improving execution speed. The use of the tool reduced test time by up to 97% (on average, 62%) without impacting the tests or decreasing the fault detection capacity.

Rahman et al. (2024) present *FlakeRake*, an automated approach to reproduce flaky test failures due to order issues, which occur simply by altering the order in which processes in the test are executed or by introducing a longer wait than in other executions where the failure does not occur. First, they attempted to reproduce 1,167 flaky test failures in an existing dataset known to contain flaky tests. By rerunning the tests 10,000 times, they found that flaky tests were detected only 10% of the time, highlighting the difficulty in identifying such failures by merely rerunning the tests (ReRun). *FlakeRake* aims to increase the efficiency of reruns by injecting actions into specific positions in the test code that temporarily pause a thread or task, causing the processes to have their execution order altered more than usual and creating a longer wait time than expected. When applying this approach, the authors found they were able to reproduce more failures than other approaches, such as ReRun. They were also able to reproduce failures more reliably, with more than 50% of the times the same test set was executed, producing a consistent number of failures (107), while other approaches sometimes failed to reproduce any failures at all.

Morán et al. (2020) present the *FlakyLoc* technique, which focuses on automatically detecting the root cause of flaky tests. The technique involves characterizing factors that can reveal flaky tests in web systems. It executes the same test case multiple times in differently configured environments, altering characteristics such as CPU, web browser, screen resolution, and memory. *FlakyLoc* is applied in a real-world case study, where it was possible not only to automatically detect the root cause of the flaky test but also, through the characterization performed, provide relevant information for its correction.

Gruber and Fraser (2023b) present a flaky test mining tool for Python projects, called *FlaPy*. The tool is capable of executing tests from Python projects that are provided to it via file or can sample projects from the Python Package Index (PyPI). After executing the tests, the tool parses the results and presents statistics and information regarding the execution. The tests are isolated using Docker, providing greater reliability for the test results. It supports different methods of installing dependencies to ensure diversity

within the sample set of projects, parallelizes test execution, and analyzes and presents the results in a concise format. To validate the tool, they randomly selected over 26,000 Python projects hosted on Git repositories that contained tests. They executed each test at least 200 times - 100 times consecutively and 100 times randomly - totaling more than 20,000 hours of execution time. Of the 26,000 projects, approximately 10,900 had at least one test, and of these, around 400 had at least one flaky test.

2.7 Concluding Remarks

In this chapter, essential concepts related to software testing were presented, along with foundational aspects of flaky tests and DevOps culture, including automation, CI and CD, as well as Docker and container orchestration. Monitoring concepts were also discussed, and related work was reviewed to provide the theoretical foundation for this thesis. In the following chapter, the FlakyTestLab tool will be introduced, in which the concepts presented here were applied during its development, resulting in a tool that can be broadly used for test execution in diverse scenarios.

Chapter 3

Tool and Approach

This chapter presents the developed tool. Section 3.1 describes the adopted approach and the structure of FlakyTestLab. Its components and their respective purposes are also presented. Section 3.2 describes how the implementation was carried out and how the tool can be used. Section 3.3 introduces the Naive approach, which was used as the baseline for the studies presented herein.

3.1 FlakyTestLab

Flaky tests are a significant issue in the industry, often regarded as the second biggest offender in delivery times during certain cycles, causing delays and generating rework (LAM et al., 2019). Identifying flaky tests is costly, as it usually involves re-running tests repeatedly, which is not only expensive but also does not guarantee the failure can be reproduced (PARRY et al., 2021). Figure 3.1 shows a representation of the re-execution approach, one of the most common methods, also known as ReRun. This approach involves re-running the tests N times or until the failure occurs (CORDEIRO et al., 2021). It is commonly applied whenever a developer encounters an inconsistent failure in a test.

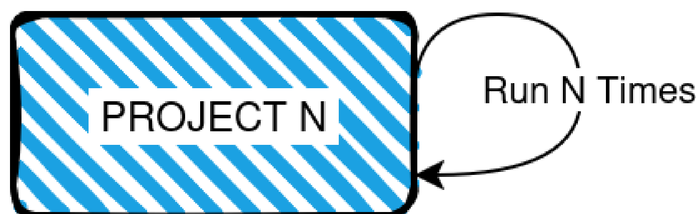

Figure 3.1 – Baseline of current projects.

Efforts have been made to provide more efficient and/or less costly alternatives, as well as to improve the reliability of flaky test detection (LAM et al., 2019; BELL; KAISER, 2014; BELL et al., 2018; RAHMAN et al., 2024; CORDEIRO et al., 2021; MORÁN et al., 2020). However, these are focused on more specific programming languages, leaving a large group of systems unsupported in terms of flaky test detection.

FlakyTestLab is presented (Figure 3.2), which, unlike these previous works, explores more global mechanisms used by virtually all systems, such as CPU and RAM. FlakyTestLab was developed with the aim of supporting the detection of flaky tests through reruns, as the tool provides an infrastructure in which multiple environments can be configured, allowing multiple tests to be executed in isolation. In addition, the tool also provides an observability layer that captures and stores logs and test reports (such as XML, TXT files, etc.) and offers tools, including visual ones, enabling the tester to monitor the reruns. Therefore, FlakyTestLab provides a mechanism for detecting flaky tests through reruns, allowing testers to execute their tests in isolation, with different configurations and multiple repetitions, and to monitor them through the observability layer. This is possible due to the characteristics of the approach adopted for the implementation of the tool:

- **Language-Agnostic:** virtually any type of project can be applied within the tool, as it is designed to execute containers in general, regardless of the programming language. However, exceptions may exist;

- **Container-based isolation:** tests can be executed in complete isolation, mitigating the possibility of one execution affecting another, except with respect to resources external to the pods, such as networks, databases, and related components. To manage these isolated executions, container orchestration was employed to schedule the executions in a specific order and with a defined configuration;

- **Variation of common configurations**: By allowing the tester to primarily vary common computational resources (CPU and RAM), the tool enables the execution of a broader range of projects, since the configuration of these resources is independent of the system type or programming language. In addition, other parameters, such as the Docker image, test command, and additional settings, can also be modified;

- **Variation of parallelism:** in addition to allowing variation in computational resources, the tool also enables the tester to work with parallel executions rather than being limited to sequential ones, which can optimize the time required to complete a task. This feature, as well as the number of parallel executions, depends on the size of the host on which the tool is deployed.

FlakyTestLab can be used across a wide range of systems for faster and more efficient detection of flaky tests, as it allows the execution of multiple systems in different confi-

gurations, in parallel, with control over the resources of each test environment, isolated, while also providing monitoring of the results.

In Figure 3.2, the architectural representation of the tool can be observed. The structure consists of the combination of several components that collectively enable the functionality proposed by FlakyTestLab. Two main blocks containing the core components of the tool can be identified: **Kubernetes Components**, which comprise a set of essential plugins required for the operation of most Kubernetes clusters, along with additional plugins that support the tool's purpose; and **Monitoring Components**, which consist of supplementary systems installed in the cluster primarily to provide the observability layer.

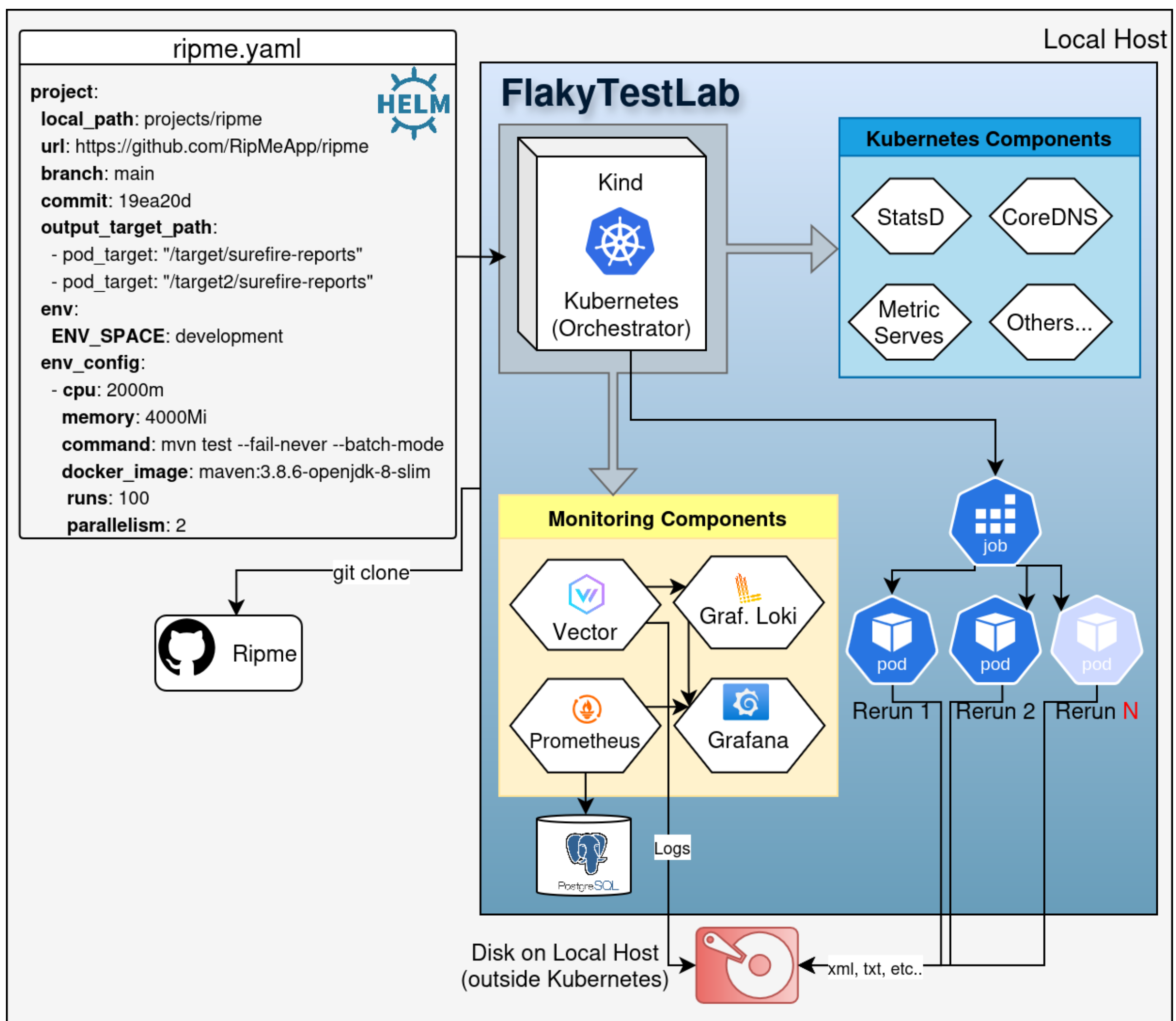

Figure 3.2 – FlakyTestLab Architecture.

Storage layers (**Disk on Local Host**) are also provided, through which any type of file can be exported from within the pods, provided that it is mapped in the input file (a *yaml* file). As illustrated, multiple small volumes were used to store test report files and logs on the host. Similar layers can also be employed to run persistent databases within the tool through the use of pods, such as in the case of PostgreSQL. Without these storage

layers, execution logs, report files (XML, TXT, etc.), and database data would be lost whenever a pod terminated its execution.

In Figure 3.2, a **Job** can also be observed. A Job is a Kubernetes component whose function is to manage the execution of pods according to a predefined specification. This specification is defined within FlakyTestLab using a Helm chart[1] as a template (an example is also shown in Listing 3.1). In FlakyTestLab, the Job manages **pod** executions according to this definition. In the example presented in the Figure 3.2, the configuration file specifies that the test should be executed one hundred times, with two pods running in parallel at a time.

The **Orchestrator** can also be observed, which consists of Kind running locally on the host. This enables a Kubernetes cluster to be executed independently on a single host.

An input file containing the project definition in **yaml**[2] format (ripme.yaml) is also provided. This file is written in a format that is subsequently interpreted by a script and by Helm. When this file is applied within FlakyTestLab, a tool script reads and interprets it, cloning the project into the environment and making the system's source code available for execution. Subsequently, the same script submits the yaml file to Helm, which deploys the cloned system into the Kubernetes cluster. This deployment creates several components, such as the previously described **Jobs** and **pods**. At this stage, test execution begins.

FlakyTestLab, therefore, provides an environment built on top of the Kubernetes orchestrator, enabling the creation of configurable environments with CPU, RAM, and parallelism as adjustable parameters. Through these configurations, computational resources can be limited and multiple replicas of the same environment can be executed, thereby making flaky test detection faster and more efficient. The innovation lies in the automation layer, which simplifies the deployment of the tool and its components, as well as in the architectural design, in which multiple distinct components were organized to operate cohesively as a unified solution capable of fulfilling the objectives of this work.

As an input for FlakyTestLab, Listing 3.1 shows the configuration of a project to be executed. In the project definition, the local directory path (*local_path)* where the project will be cloned (always projects/<project_name>), the project's *URL* on GitHub (or similar), the target *branch*, the specific *commit*, a list of directories (*output_target_path*) containing test output files (which are then mapped outside of the Kubernetes pods), and the definition of environment variables (*env*), required for certain projects, are specified. With these parameters, the tool itself handles cloning the defined project, providing the source code ready for execution.

[1] <https://helm.sh/docs/topics/charts/>
[2] <https://yaml.org/>

In Listing 3.1, there is also the *env_config* section, which includes a list defining five distinct environments. In this example, all parameters are maintained except for *parallelism*. That is, five environments are configured with the same CPU and RAM, but with different concurrent execution configurations: 1, 2, 4, 8, and 16 simultaneous pods; with each environment scheduled to run (*runs*) 100 times. There is also the definition of the *command*, which is essentially the command used to run the tests, and the *docker_image*, where the Docker image to be used for test execution can be specified. These parameters make the tool flexible, as it is possible, for example, to run a Python or JavaScript project simply by changing the Docker image and the test execution command.

Listing 3.1 – Example of configuring a Java project (ripme.yaml).

```
project:
  local_path: projects/ripme
  url: https://github.com/RipMeApp/ripme
  branch: main
  commit: 19ea20d
  output_target_path:
    - pod_target: "/target/surefire-reports"
  env:
    ENV_SPACE: development
env_config:
  - cpu: 2000m
    memory: 4000Mi
    command: mvn test --fail-never --batch-mode
    docker_image: maven:3.8.6-openjdk-8-slim
    runs: 100
    parallelism: 1
  - cpu: 2000m
    memory: 4000Mi
    command: mvn test --fail-never --batch-mode
    docker_image: maven:3.8.6-openjdk-8-slim
    runs: 100
    parallelism: 2
  - cpu: 2000m
    memory: 4000Mi
    command: mvn test --fail-never --batch-mode
    docker_image: maven:3.8.6-openjdk-8-slim
    runs: 100
    parallelism: 4
  - cpu: 2000m
    memory: 4000Mi
    command: mvn test --fail-never --batch-mode
    docker_image: maven:3.8.6-openjdk-8-slim
    runs: 100
    parallelism: 8
  - cpu: 2000m
    memory: 4000Mi
    command: mvn test --fail-never --batch-mode
    docker_image: maven:3.8.6-openjdk-8-slim
    runs: 100
    parallelism: 16
```

3.1.1 Components

The construction of FlakyTestLab used several components widely adopted in the industry by large websites and companies, which help manage workloads and distribute computational resources, enabling cost control and serving various other purposes.

FlakyTestLab is built on top of Kubernetes, but using Kind. It was used to build a tool that provides the definition and creation of various environments. Unlike a production environment, which is typically built on cloud services such as AWS or Google GCP and uses multiple hosts as a large cluster, Kind allows workloads to be run on a single host, while still providing all the functionalities of a distributed Kubernetes cluster (Kubernetes SIGs, 2026).

In this work, Helm was essential to achieve the flexibility and ease of use, as it made it possible to launch various types of applications with different configurations using a single base. In FlakyTestLab, all applications are executed in the form of Kubernetes jobs, where each job is a set of definitions with a specified number of runs to be executed. Each execution is performed within an isolated Kubernetes pod. Additionally, it enabled the creation of an interface that accommodates various application models, as seen in Listing 3.1. The data is simple, and there is no need for the user to have an extensive knowledge of Kubernetes. In Listing 3.2, the configuration of a Python project can be seen, using the exact same base, with only essential parameters modified, such as the Docker image to be used in the job and the command for executing the tests.

Listing 3.2 – Example of configuring a Python project (levitate.yaml).

```
1  project:
2    local_path: projects/levitate
3    url: https://github.com/AppliedAcousticsChalmers/levitate
4    branch: master
5    commit: f6b2b26
6    output_target_path:
7      - pod_target: "/output"
8    env:
9      ENV: development
10 env_config:
11   - cpu: 2000m
12     memory: 4000Mi
13     command: "pip install -e . && pip install -r requirements.txt && pytest --junitxml=
           output/levitate.xml || true"
14     docker_image: python:3.5-buster
15     runs: 100
16     parallelism: 1
```

A framework for database creation is provided within the tool so that applications or services requiring storage can also be supported, since many tests may need to validate interactions with databases. In the project, a PostgreSQL DBMS was added as a demonstration of this structure, however, any DBMS can be provisioned through a Docker pod and made available in the environment. It also follows the same resource management

characteristics, meaning that its resources can be limited or increased for experiments in which this may influence application behavior.

CoreDNS is a highly flexible and extensible DNS server solution (Domain Name System), specifically designed for Kubernetes. It operates as a plugin-based system, allowing for the dynamic configuration of DNS functions, such as name resolution, caching and support for specific protocols like DNS over TLS (DoT) and DNS over HTTPS (DoH). Its main advantage over other DNS solutions is its modularity and efficiency. It is designed to be lightweight and scalable, which is essential in large Kubernetes clusters. Additionally, it provides a simple configuration mechanism based on configuration files, facilitating customization and administration (CoreDNS Authors, 2023). In FlakyTestLab, CoreDNS was used to ensure internal communication between services and other components, such as databases and Prometheus, as well as to enable the future definition of new project models, such as providing testing environments for frontend systems for example.

3.1.2 Monitoring Components

The applications executed in FlakyTestLab often generate various output files containing test-related data, such as XML, TXT files, and logs. For a more thorough analysis of the test results, FlakyTestLab provides options to analyze and extract this data from the pods where the tests were executed, considering that these pods are temporary and may become unavailable after execution.

For simpler analyses, it is possible to use Prometheus, Grafana and Loki directly, as shown in Figure 3.3, where example graphs were created to observe certain error behaviors. However, these can be further developed to display various types of information and also create alerts.

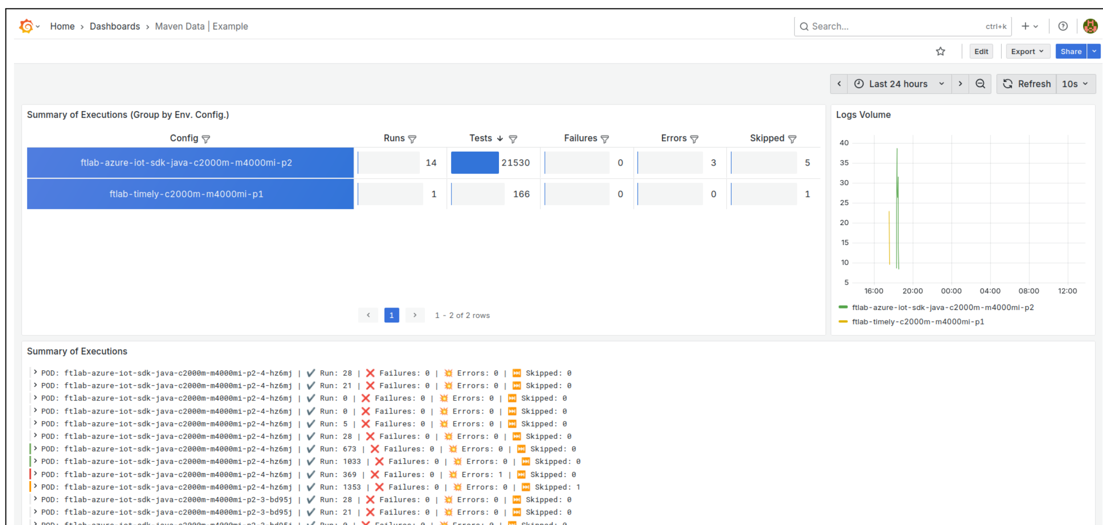

Figure 3.3 – FlakyTestLab Tool Grafana example.

Prometheus collecting metrics from different services and components is essential to ensure system observability and health. Prometheus stores metrics as time series, enabling flexible and efficient queries over time (BRAZIL, 2018). With its own query language called PromQL, Prometheus enables powerful and precise analysis of large volumes of data. It also offers advanced alerting features. It can be configured to trigger real-time notifications when specific conditions are met, such as a metric dropping, reaching a critical threshold, or when a flaky test is detected. When integrated with other visualization tools like Grafana, Prometheus forms a powerful solution for real-time system monitoring, being widely adopted by DevOps teams, operations, and software engineers (Prometheus Authors, 2023a).

Grafana is used in FlakyTestLab as a visual interface where certain logs and metrics can be accessed. It integrates with Prometheus databases (containing system metrics) as well as Loki (a log database), enabling testers to monitor system data and measure resource usage. It is fully open within the tool, allowing testers to create their own dashboards, set alerts for metrics or logs, and visually track the performance of running pods. Figure 3.3 shows the use of Grafana with the creation of an example dashboard for Java Maven, available in the tool. In addition to this example dashboard, several other dashboards that are natively useful for Kubernetes users (such as the Node Exporter / Nodes dashboard in Figure 3.4) are installed by default during the Prometheus installation and are maintained by the community.

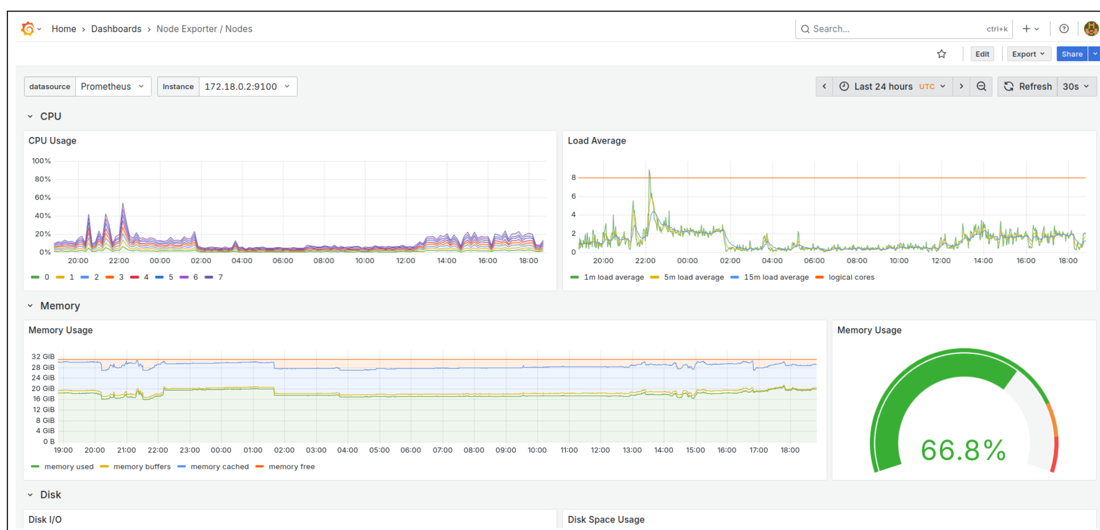

Figure 3.4 – Local host resource consumption metrics.

Grafana Loki is used in FlakyTestLab to centralize the execution logs of the Kubernetes pods into a single component. Complementing Prometheus, which stores time series and performance metrics, Loki is responsible for log storage, allowing the addition of labels for precise analysis. This enables detailed reading and analysis of service logs. Furthermore, Grafana provides the ability to analyze both time series data (such as CPU and RAM usage) and service logs within a single platform. This makes it possible to

perform searches, create dashboards and conduct various other essential analyses for the applications under test, all within Grafana.

Vector is used in FlakyTestLab to collect logs directly from the Kubernetes pods. It acts as an intermediary between the logs generated by the applications within the pods and Grafana Loki, which is used as the log storage. Vector was chosen for its high efficiency and flexibility, which also made it possible to configure multiple outputs (sinks) and export the logs from the pods to local files within the host (Figure 3.2). This is important for situations where logs will also be analyzed, including the use of scripts that can facilitate the analysis of this data, as was done in the experiments of this project, which will be presented in Chapter 4.

Prometheus StatsD Exporter is a tool that allows integration of metrics collected by StatsD with Prometheus. StatsD is a widely used metrics collection agent in software environments for monitoring the performance of systems and services. However, Prometheus does not natively support the collection of metrics in the StatsD format. The Prometheus StatsD Exporter acts as an intermediary between StatsD and Prometheus, translating the metrics sent by StatsD into a format that Prometheus can process. It also supports count metrics, latency, and other statistical metrics that are essential for monitoring the performance of systems and applications (Prometheus Authors, 2023b). StatsD enables the user of FlakyTestLab to customize the metrics they need, when the logs and test output files are not sufficient for their analysis, making the tool more flexible and comprehensive.

For this project, the Metrics Server is the tool that provides data on resource usage by the pods on the host. It is possible to track metrics for individual pods or also for the global host (Figure 3.4), where issues like CPU limitations for a service as well as OOMKilled problems can be detected. Metrics provided by this service can also help identify the minimum configuration required for a project to run its tests, in case the user does not have this information.

3.2 Implementation and Usage

FlakyTestLab was implemented using Kubernetes as its foundation and includes the components presented before. Kubernetes was chosen as the base because the tool's goal is to provide an environment with various configurations, which can be used to run tests in general and increase the occurrence of flaky tests.

Tools like Shaker (CORDEIRO et al., 2021) aim to enhance the detection of flaky tests by introducing disturbances into the test environment, such as processes that compete for CPU or allocate and deallocate RAM in a way that interferes with the testing process. This is a method that works, but in some cases, it can be wasteful, as such disturbances may require introducing noise that exceeds the resource usage of the system under test.

For example, Shaker was partially tested using GitHub Actions as the testing environment, which by default provides four virtual CPUs (4 vCPUs) and 16GB of RAM for public repositories (GITHUB, 2023). Some of the projects used in Shaker, such as Ripme, were executed in FlakyTestLab with up to four pods running in parallel, using the same hardware. Some data about this comparison will be presented in Section 4.3.

FlakyTestLab, therefore, aims to create an environment where tests in general can be executed and provides disturbance mechanisms with effects similar to Shaker (CORDEIRO et al., 2021). By "similar effects", we mean restricting computational resources to enhance the detection of flaky tests, but without necessarily generating unnecessary processing. It simply restricts resources using the mechanisms available in Kubernetes. Additionally, it makes use of resources that would otherwise be wasted to run more tests in parallel.

To make this possible, a resource management feature of Kubernetes was used, which involves configuring resources. This allows you to define values for CPU and RAM, ensuring that pods cannot use more resources than what has been allocated to them. This, in turn, limits the applications' access to the host's resources, even if they are still available. In Listing 3.3, an example of this configuration can be seen. Specifically for this project, the Guaranteed QoS (Quality of Service) approach (Kubernetes Contributors, 2023) was used, which ensures that resources are guaranteed both in terms of what is requested and the set limit. In other words, a pod will not use less than what was requested, nor will it exceed the defined limit. In this example, to define the Guaranteed QoS, both the request and limit need to be set to the same values.

Listing 3.3 – Example of configuring a resource limits to PODs (Guaranteed QoS).

```
1 resources:
2   requests:
3     memory: "512Mi"
4     cpu: "500m"
5   limits:
6     memory: "512Mi"
7     cpu: "500m"
```

In its current form, the tool requires the user to have a Linux operating system (tested on Ubuntu 20.04 and 22.04) with available CPU and RAM resources, superuser access and Internet connectivity. With this, it is possible to clone the project repository and run the commands *make setup* and *make cluster_up*, where all installation and configuration scripts will be executed, making the tool available for use.

After installing and configuring the tool, the user should be inside its directory, where the structure will be as shown in Figure 3.5. Here, the configuration file for the demo project (ripme.yaml), presented in Listing 3.1, can be found. In other words, to run a test using the tool, the user needs to execute the two commands for setup and cluster initialization, then add the file with the test parameters following the directory structure in Figure 3.5 and the model shown in Listing 3.1.

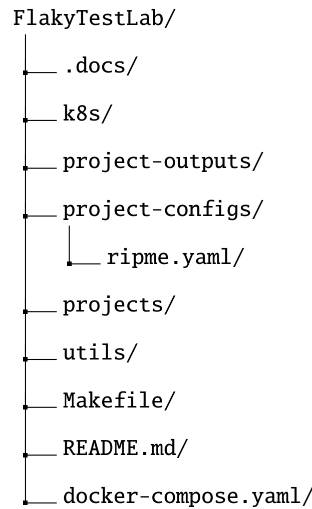

Figure 3.5 – Project file structure.

After defining the project, the tests should be executed and the generated logs and files analyzed. To run the test, the command *make up project=ripme* is used (*make down project=ripme* to uninstall the project). This command performs all the necessary steps to execute the project, from cloning the repository to running it in the cluster. It is important to note that for parallel executions or a large volume of runs, the host must be sized to meet these requirements. For example, it will not be possible to run parallel=4 for a project that requires 4GB of RAM on a host with only 8GB of RAM. In such a situation, as many pods as the host can accommodate will be executed, with the remaining ones in a waiting state.

During the execution of the project, it is possible to monitor its status by running commands such as *make pods* to see the list of running pods and their respective states, and *make jobs*, as shown in Listing 3.4, where you can track a summary of the state of each job (each job represents a configuration from the *env_config* list presented in Listing 3.1). In the example in question, five configurations were executed, as shown in Listing 3.1, with four completing and only the configuration with parallel=1 (ftlab-ripme-c2000m-m4000mi-p1 or just P1) still running.

Listing 3.4 – Example of getting jobs.

```
1 # make jobs
2 ftlab-ripme-c2000m-m4000mi-p1 Running 66/100 16h 16h
3 ftlab-ripme-c2000m-m4000mi-p16 Complete 100/100 4h31m 16h
4 ftlab-ripme-c2000m-m4000mi-p2 Complete 100/100 14h 16h
5 ftlab-ripme-c2000m-m4000mi-p4 Complete 100/100 9h 16h
6 ftlab-ripme-c2000m-m4000mi-p8 Complete 100/100 6h31m 16h
```

After the execution is completed, FlakyTestLab directory structure will have more directories, where the test logs and their respective generated files will be stored. In Figure 3.6, the structure with the generated outputs can be observed. The structure begins with the

project name, which allows for the execution of multiple projects. Following that, there are the "logs" and "outputs" directories, containing the logs and generated files, respectively, followed by ftlab-ripme-c2000m-m4000mi-p1, which represents the configuration data, i.e., the Ripme project with the definition of two CPUs (c2000m), four gigabytes of RAM (m4000mi), and running only one pod at a time in parallel (p1). This allows for the individual analysis of test data for each configuration generated by FlakyTestLab.

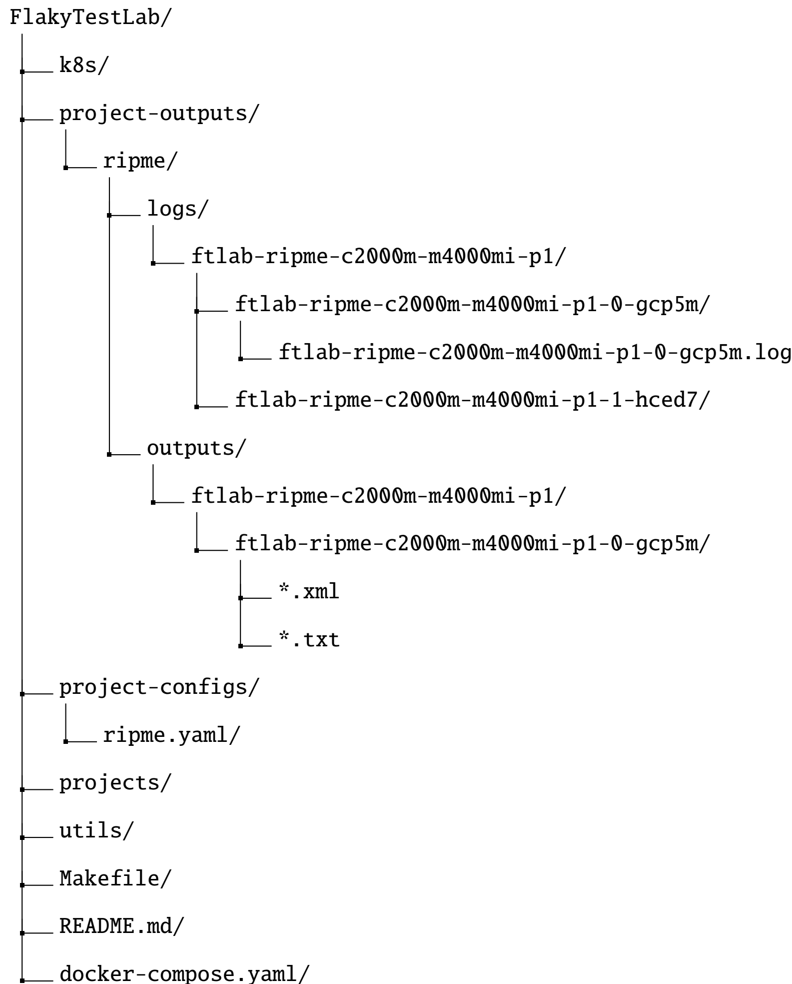

Figure 3.6 – Project file structure after test execution.

After the tests are completed, FlakyTestLab provides the test data to be analyzed according to the user's needs. The tool does not provide any type of analysis because, due to its goal of being as generic and broad as possible, defining analysis methods could make the tool more selective. Therefore, the development was focused on ensuring that all the necessary data for a thorough analysis was provided to the user. It is then up to each individual to define their objectives and, consequently, what and how to analyze the information provided. For this work, 28 projects were analyzed, 14 in Java and 14 in Python, and for each programming language, a different script had to be created so that

our evaluation could be conducted, following our criteria and answering the RQs defined. The data regarding the tests will be presented in Chapter 4.

3.3 The Naive Approach

In order to have a comparison baseline for FlakyTestLab, we implemented a Naive Approach based on practical assumptions. Therefore, we needed an environment without any interference from the tool, and that would be close to the typical setup of users. To do this, all projects were executed using only Docker, with runs using the same Docker image applied in FlakyTestLab. To simplify these executions, a small *bash* script was created to automate the project runs.

The same file defined in Listing 3.1 is leveraged so that exactly the same characteristics (Docker image, output file paths, commit, branch, and command) are utilized and ensured during execution. The script follows a simplified workflow: the project is cloned using the specified branch, a Docker container is executed with the image defined by the project, and the tests are run using the specified command. All logs and outputs are extracted from the container and stored on the host for subsequent analysis. All executions were performed sequentially, with one container executed at a time, as is commonly done by testers who adopt the rerun approach.

Therefore, Naive consists of a Bash script that reads exactly the same configuration source as the project applied in FlakyTestLab, however, its execution is performed without parallelism or resource restrictions, such that each container utilizes the full capacity of the host.

3.4 Concluding Remarks

In this chapter, the FlakyTestLab tool was presented, along with its components. The purpose of each component and how they contributed to the functionalities designed for the tool were also described. Additionally, the implementation process was explained, as well as how the tool was used to conduct the experiments. Finally, the Naive approach was introduced, serving as the baseline for all the studies in this work. In the following chapter, the experiments conducted will be described in detail. The projects used in the analyses, the hardware employed, the (RQs), and the results obtained will be presented.

Chapter 4

Evaluation

This chapter presents the details of the experiments and the analysis of the results. Section 4.1 describes the experimental settings, the selected projects, the hardware used, and the available replication package. Section 4.2 presents and analyzes the results. Finally, Section 4.3 offers reflections on the experiments and their outcomes, as well as a brief comparison with Shaker (CORDEIRO et al., 2021).

4.1 Experimental Settings

In order to evaluate FlakyTestLab, we define the following research questions (RQs):

- $\mathbf{RQ}_1$ To what extent does FlakyTestLab accelerate test reruns?
- $\mathbf{RQ}_2$ To what extent can FlakyTestLab detect the first failure of a flaky test?
- $\mathbf{RQ}_3$ To what extent does FlakyTestLab enhance the manifestation of flaky tests?

$\mathbf{RQ}_1$ intends to evaluate how FlakyTestLab performs, from a runtime perspective, a pre-determined number of test reruns. Rationale: the practitioner wants the results of reruns fast. Additionally, $\mathbf{RQ}_2$ targets the time taken to detect the first failure of a flaky test. Rationale: When debugging a flaky test, the reruns should be stopped when the first failure occurs; this should provide enough pieces of information for the developer.

In $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$, we compare FlakyTestLab with a Docker-based baseline implemented outside the container orchestration, we call it Naive (see Section 3.3). Rationale: rerunning is usually implemented by practitioners as trivial scripts or repetition-based procedures in testing frameworks.

Finally, $\mathbf{RQ}_3$ assesses whether FlakyTestLab can increase the manifestation ratio of flaky tests. Rationale: it is necessary to validate whether the FlakyTestLab tool can reduce computational effort and the time required to expose flaky tests. By measuring the "extent", we quantify the tool's effectiveness in transforming a rare failure into an observable and frequent event. For this RQ, FlakyTestLab is compared with Naive, the baseline, to determine whether any improvement is observed in the number of detected flaky tests.

4.1.1 Selected projects

The selection of projects used in the experiments was based on papers where flaky test cases had already been identified (LAM et al., 2020a; CORDEIRO et al., 2021) or on public databases containing some projects in different programming languages, which can be used for these purposes (TESTINGRESEARCHILLINOIS, 2025). These projects are listed in Table 4.1.

Table 4.1 – Projects selected to validate the tool FlakyTestLab.

Project Name	GitHub URL	Commit Hash	Programming Language	Test Cases	Base setup
ripme	https://github.com/RipMeApp/ripme	19ea20d	Java	227	2CPU 4GB-RAM
exhibitor	https://github.com/soabase/exhibitor	d345d2d	Java	53	2CPU 4GB-RAM
mockserver	https://github.com/mock-server/mockserver	b1093ef	Java	2870	2CPU 4GB-RAM
azure-iot-sdk-java	https://github.com/Azure/azure-iot-sdk-java	a9226a5	Java	4364	2CPU 4GB-RAM
timely	https://github.com/nationalsecurityagency/timely	3a8cbd3	Java	703	2CPU 4GB-RAM
java-websocket	https://github.com/TooTallNate/Java-WebSocket	fa3909c	Java	145	2CPU 4GB-RAM
rxjava2-extras	https://github.com/davidmoten/rxjava2-extras	d0315b6	Java	390	2CPU 4GB-RAM
shardingsphere-elasticjob	https://github.com/apache/shardingsphere-elasticjob	b022898	Java	560	2CPU 4GB-RAM
luwak	https://github.com/flaxsearch/luwak	c27ec08	Java	205	2CPU 4GB-RAM
delight-nashorn-sandbox	https://github.com/javadelight/delight-nashorn-sandbox	da35edc	Java	79	2CPU 4GB-RAM
db-scheduler	https://github.com/kagkarlsson/db-scheduler	4a8a28e	Java	51	2CPU 4GB-RAM
karate	https://github.com/intuit/karate	09bc49e	Java	605	2CPU 6GB-RAM
killbill	https://github.com/killbill/killbill	killbill-0.22.21	Java	35	2CPU 6GB-RAM
fastjson	https://github.com/alibaba/fastjson	e05e9c5	Java	4464	2CPU 6GB-RAM
levitate	https://github.com/AppliedAcousticsChalmers/levitate	f6b2b26	Python	768	2CPU 4GB-RAM
textdistance	https://github.com/life4/textdistance	bcd2174	Python	430	2CPU 4GB-RAM
kevinarpe-rambutan3	https://github.com/kevinarpe/kevinarpe-rambutan3	5d74699	Python	190	2CPU 4GB-RAM
pyquery	https://github.com/gawel/pyquery	f2533d1	Python	150	2CPU 4GB-RAM
kb-python	https://github.com/pachterlab/kb-python	3ce4d8e	Python	129	2CPU 4GB-RAM
piripherals	https://github.com/quantenschaum/piripherals	74e4807	Python	85	2CPU 4GB-RAM
r-map	https://github.com/mentaal/r-map	56cad81	Python	66	2CPU 4GB-RAM
fossor	https://github.com/linkedin/fossor	d8dbdc4	Python	64	2CPU 4GB-RAM
pysshutil	https://github.com/choppsv1/pysshutil	e3ab876	Python	57	2CPU 4GB-RAM
service-manager	https://github.com/hmrc/service-manager	e0a9f2a	Python	57	2CPU 4GB-RAM
packeteer	https://github.com/lungdart/packeteer	2c6e0e8	Python	50	2CPU 4GB-RAM
hotwing-core	https://github.com/jasonhamilton/hotwing-core	f4d864b	Python	49	2CPU 4GB-RAM
pymemorydb	https://github.com/davidbetz/pymemorydb	9c8e864	Python	47	2CPU 4GB-RAM
wiki-futures	https://github.com/AndrewRPorter/wiki-futures	ae6bec3	Python	6	2CPU 4GB-RAM

4.1.2 Hardware used

The development and initial testing of FlakyTestLab were conducted on conventional hardware: an Intel® Core™ i7-1165G78 with eight cores, 16 threads, and 32GB of RAM. After the initial evaluations, much more robust hardware was used to meet the high demands of the experiments: two Intel(R) Xeon(R) Gold 6122 CPUs @ 1.80GHz with 40 cores, 80 threads, and 384GB of RAM. With this hardware, it was possible to run up to sixteen pods of the same project in parallel without issues, even with projects requiring two virtual CPUs (2vCPU) and six gigabytes of RAM (6GB RAM) per pod (see last

column of Table 4.1). More pods could be run simultaneously if needed, but following the goal of doubling the number of parallelism pods (parallelism=1,2,4,8,16), it would not be feasible for this machine to double again with enough headroom (parallelism=32).

4.1.3 Metrics collected for each RQ

During the design of the experiments, the RQs were used as the basis to obtain the appropriate metrics for answering each RQ.

In rerun-based flaky test detection, the approaches need to perform several repetitions (i.e., reruns) in order to observe and identify flaky tests. In this study, we define 100 reruns as a reference to compute the metrics; similar values have been used in the literature (PINTO et al., 2020; HASHEMI et al., 2025; LAM et al., 2019). Each rerun is designed to be isolated and independent, so each time all steps are performed, from instantiating the container, installing dependencies, building to running the test suite.

For RQ_1, we collected the runtime (in seconds) needed to perform each rerun and calculate its mean value, we call it *Average Rerun Runtime*. We compute the total time needed to complete 100 reruns, we called it *Total Runtime* (TR). Also, we calculate the speedup of FlakyTestLab with respect to the Naive approach, i.e. $TR_{Naive}/TR_{FlakyTestLab}$.

Concerning RQ_2, we collected the runtime (in seconds) taken to detect the first failure of a flaky test.

In RQ_3, we computed the test flakiness ratio of test cases given by the number of failures divided by the total number of reruns, i.e., flaky ratio (HENDERSON et al., 2023). We also calculate number of flaky tests detected.

In RQ_1-RQ_3, we compare different configurations of FlakyTestLab with a baseline that implements a Docker-based approach that executes test suite runs sequentially. So, each test suite run is executed in a separate container, similarly to other studies (LAM et al., 2020b; PARRY et al., 2022). For the baseline, all resources of the machine are available, without other applications running in parallel, apart from the OS services.

4.1.4 Configuration definition for projects

To define the configurations of FlakyTestLab, we did a basic configuration without restricting the resources too much. After that, we run the project once and track the pod's resource consumption through Grafana (subsection 3.1.2). With the data collected from Grafana, we had a basic configuration to run the project. So we ran it five times in a row to ensure that there would be no issues with that configuration, especially with RAM (OOMKilled). This configuration is listed in the last column of Table 4.1, per project.

4.1.5 How the metrics were collected

All metrics were collected using FlakyTestLab structure. As presented in Chapter 3, FlakyTestLab provides mechanisms that externalize test results from within the pod to the local host. To achieve this, each project was also run manually to detect which project directories contained report files and then these directories were configured in configuration files like Listing 3.1.

For all the projects used in the experiments, the manifestation of flaky tests was monitored, with a test being considered flaky whenever its output changed, i.e., when it first failed (or encountered an error) and later pass, or vice versa. Analyzing the results, the tests considered flaky were categorized as follows:

- All projects were executed the same number of times, that is, one hundred times (100 reruns);
- A test was considered failed whenever its output was either a failure or an error;
- Those that failed or reported an error, meaning they did not pass at all during the 100 reruns, were discarded as they do not meet the criterion of variation in their output and therefore are not considered flaky;
- The first occurrence of a flaky test was considered the first instance of variation. If the first variation occurred in reference to the first run, it was considered the initial appearance of the flaky test.

4.1.6 Replication package

All tests were conducted on the same host (Section 4.1.2), always without competition from other projects. That is, for FlakyTestLab, all configurations defined in Listing 3.1 were executed simultaneously (parallelism 1,2,4,8, and 16), with continuous monitoring to ensure the host did not become overloaded (using the htop command, the load average (WALKER, 2006) was always monitored to ensure it remained below 80%).

However, there were some exceptions to these executions, as some projects took more than 40 hours to complete the 100 runs. It was decided to make better use of the host's resources, especially considering that, as Kubernetes isolates the requested resources (presented in Section 3.2), it would be possible to run other, less resource-intensive projects in parallel with projects that were still running with parallelism=1, which is the slowest case. Therefore, the exact criteria adopted to execute the tests were:

- Each project was selected according to Table 4.1, with all of them having been part of previous research, where flaky test cases were identified;

- Each project was configured as shown in Listing 3.1, following the exact same configuration for all runs, except for the parallelism parameter, which was varied between 1, 2, 4, 8, and 16 simultaneous runs;

- The FlakyTestLab tool was deployed on the host mentioned in Section 3.2;

- An arbitrary project was started to execute all its runs (totaling 500 runs, with 100 from each parallelism configuration);

- As expected, the parallel=16 configuration was always the first to finish due to running more pods simultaneously. It is also the one that consumes the most host resources. This was followed by parallelism=8, where the behavior was similar - that is, the tests completed before parallelism=4 but after parallelism=16;

- Once the parallelism=16, 8, 4, and 2 tests were completed, with only parallelism=1 remaining, another project would be started to optimize the use of the host's resources, as long as there was no overload - i.e., ensuring the load average stayed below its limit, always leaving at least 20% of capacity available;

- After the tests were completed, a same script was used for each language to extract the desired information. Several spreadsheets in *csv* format were created, from which the data was extracted and processed into another spreadsheet, summarizing the results.

For the Naive approach (Section 3.3), it was decided to run each experiment individually, with no concurrency between projects in order to avoid influencing other executions and to keep the experiment as close as possible to a typical execution. This approach was necessary because there was no control or restriction on CPU and RAM usage, except for the basic isolation provided by Docker itself. In other words, each Naive execution had full access to the resources of the host described in Section 3.2.

After the individual execution of each project, the same data collection process applied to FlakyTestLab was followed, using the same scripts to generate the spreadsheets in the same format. This ensured ease of comparison and helped avoid any unintended differences in the output.

The FlakyTestLab tool and Naive are available for use under the MIT License and can be accessed via GitHub at the following addresses:

- **Naive:** <`https://github.com/VictorLopess/flaky-naive`>

- **FlakyTestLab:** <`https://github.com/VictorLopess/FlakyTestLab`>

Threats to Validity

The results may vary if the same experiment is conducted on hosts with capabilities different from those used in this study. Furthermore, to enable a more faithful analysis of the results and achieve a higher degree of reliability, prior knowledge of the projects and their characteristics is an important factor, as it would make it possible to anticipate their behavior and better understand the factors underlying the observed results.

Depending on the characteristics of the evaluated projects, the observed behavior may also not be entirely realistic, since these projects evolve over time, as do their dependencies. For example, some projects evaluated in other studies, such as `Chronicle-Queue` and `Ozone`, could not be assessed in this study because they no longer functioned properly, either due to unavailable dependencies or due to timeouts in certain APIs.

There are also some projects that appear in the results without any flaky cases. This behavior may be due to the high amount of resources available to these projects, particularly Python projects, since the environment sizing was defined to accommodate most of the projects. As a result, the environments were relatively generous for the majority of Python projects. A future study evaluating project behavior under varying CPU and RAM configurations is strongly recommended and, based on some data from the current study, is expected to yield promising results. For this scenario, it is suggested that environment sizing be performed on an individualized basis.

4.2 Analysis of Results

The results for our three RQs are presented as follows. For ease of reading and understanding, FlakyTestLab-P1 (configuration that executes one pod in parallel) will be referred to as P1. The same applies to the others, meaning they will be referred to as P2, P4, P8, and P16, respectively.

4.2.1 RQ_1: To what extent does **FlakyTestLab** accelerate test reruns?

Table 4.2 shows the average rerun runtime in seconds, per project. Columns 2-7 bring the results of the approaches compared, while the last four rows present the mininum, maximum, median, and average values.

Notice that in most projects (23 out of 28 projects) the rerun runtime increases from Naive to P1, even though these two approaches are similar, i.e., they execute one container/pod at a time without parallelism. P1 is slower due to the resource usage constraints, while Naive has access to all host's resources. The trend of increasing runtime remains when the parallelism is increased in FlakyTestLab from P1 to P16, all of 28 projects have growing test suite runtime. This behavior is also expected, as more pods in parallel com-

pete for certain resources (like Internet connection, disk access), this impacts on individual runtime of the projects' reruns.

As for Naive, project **java-websocket** has the fastest average rerun runtime (5.22s), while **azure-iot-sdk-java** is the slowest (299.73s). When looking at FlakyTestLab with any level of parallelism, project **kevinarpe-rambutan3** has the fastest test suite runtime (4.96s), while **karate** is the slowest (2480.44s). Notice that the projects are different, this may occur since some projects are more performance-sensitive to resource restriction. On the other hand, projects like **kevinarpe-rambutan3**, **pyquery**, and **wiki-futures** are in some cases faster in FlakyTestLab because they demand fewer resources.

Table 4.2 – Average rerun runtime in seconds (RQ_1).

Project	Naive	P1	P2	P4	P8	P16
ripme	267.34	337.65	360.91	387.24	428.56	417.61
exhibitor	80.4141	121.18	143.6	186.86	245.86	309.21
mockserver	175.03	423.11	442.37	481.21	553.92	590.32
azure-iot-sdk-java	299.73	528.69	530.99	567.16	603.1	631.55
timely	77.21	234.73	276.75	359.64	473.3	597.84
java-websocket	5.22	35.98	36.81	39.87	42.29	45.93
rxjava2-extras	73.84	152.31	159.73	166.44	178.51	197.81
shardingsphere-elasticjob	219.02	305.22	300.68	310.49	324.54	336.23
luwak	30.29	91.36	94.75	101.25	111.46	127.09
delight-nashorn-sandbox	12.76	83.31	83.48	88.7	97.29	107.81
db-scheduler	14.56	72.17	86.9	114.15	150.3	196.13
karate	241.82	764.96	915.3	1191.87	1363.57	2480.44
killbill	294.32	1457.23	1521.66	1592.3	1617.84	1612.46
fastjson	74.64	227.64	240.36	270.33	304.41	358.85
levitate	94.31	76.94	86.87	105.09	136.82	180.82
textdistance	284.34	271.05	284.9	309.59	346.71	388.21
kevinarpe-rambutan3	5.9	4.96	5.29	5.7	6.39	7.02
pyquery	16.44	13.92	14.3	15.16	15.89	16.98
kb_python	54.17	89.94	113.77	162.37	239.15	313.82
piripherals	41.17	41.67	42.06	42.96	43.83	44.88
r_map	7.79	8.24	8.84	9.85	11.2	12.48
fossor	56.21	58.88	59.76	61.39	63.41	63.14
pysshutil	241.77	246.83	250.18	254.72	263.42	277.01
service-manager	20.91	23.7	27.2	32.9	39.2	47.37
packeteer	11.54	12.35	13.34	14.76	16.29	17.44
hotwing-core	21.71	41.94	62.48	103.43	184.96	334.28
pymemorydb	6.39	6.64	7.14	7.85	9.12	10.35
wiki-futures	13.19	12.83	13.04	13.5	14.12	14.68
Min	5.22	4.96	5.29	5.7	6.39	7.02
Max	299.73	1457.23	1521.66	1592.3	1617.84	2480.44
Median	55.19	86.625	90.825	109.62	164.405	196.97
Average	97.93	205.19	220.84	249.89	281.62	347.78

To complement Table 4.2, Figure 4.1 shows the boxplots of the average rerun runtime per approach; the triangle represents the average value. There are a lot of intersection between the interquartile ranges, with similar runtime. Nevertheless, it is noticeable a certain increase when the parallelism grows in FlakyTestLab. The behavior depends on the characteristics of each project, but certain dependencies, network, and disk usage may impact on the results.

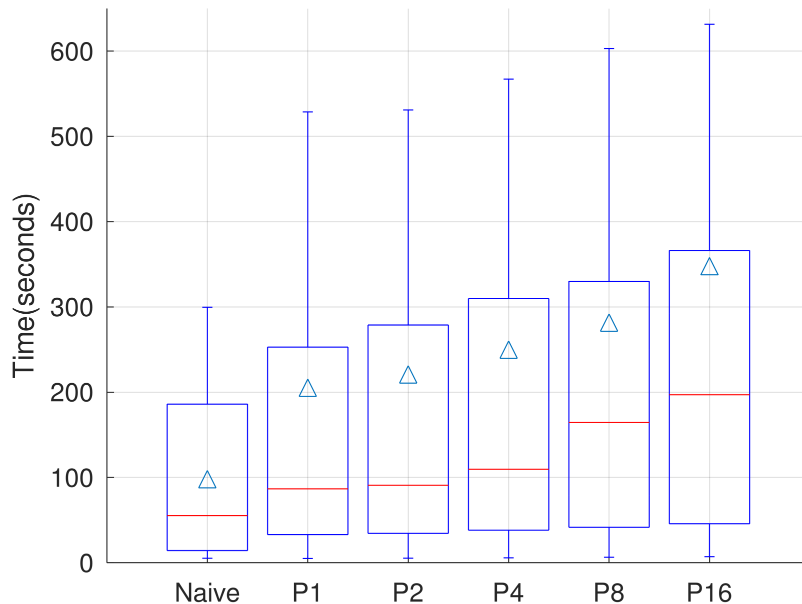

Figure 4.1 – Average rerun runtime (RQ_1).

Table 4.3 shows the total runtime for 100 reruns, per project. Columns 2-7 bring the results of the approaches compared, while the last four rows present the minimum, maximum, median, and average values.

Similarly to the rerun runtime, the total runtime on average doubles from Naive to P1. This increase occurs in 25 projects, the exceptions are **levitate, textdistance,** and **pyquery** in which P1 is slightly faster. As expected, FlakyTestLab starts to accelerate as the parallelism grows. With 2 pods in parallel, it almost matches Naive (faster in 17 projects), and P4 is faster in the 24 projects. The remaining approaches (P8 and P16) improve the performance, but the gains are smaller when taking the number of pods in parallel.

Project **kevinarpe-rambutan3** is the fastest for 100 reruns in all approaches, it goes from 877 seconds in Naive to as low as 83 seconds in P16. In Naive, project **azure-iot-sdk-java** is the slowest (30,302 seconds), while **killbill** is the project that takes more time to execute 100 reruns in all configurations of FlakyTestLab (158,450 seconds).

There are some projects that exhibited slightly different behavior than the majority. It can be seen in Table 4.3 that for the **hotwing-core** project, the total execution time was higher for P1, P2, and P4 configurations, and slightly lower for P8 and P16. This indicates that even with parallelism increased up to sixteen times, the total execution time remained almost the same when compared to Naive, suggesting that the project has a strong dependency on resources, such as CPU, RAM, and Network. This is because Naive does not impose any resource limitations on the service. This project showed a reduction of only 10.23% in execution time when comparing P16 configuration to Naive, making it the project with the smallest overall reduction.

Table 4.3 – Total runtime (for 100 reruns), in seconds (RQ_1).

Project	Naive	P1	P2	P4	P8	P16
ripme	28549	34405	18372	9869	5592	2853
exhibitor	8319	12767	7513	4862	3256	2059
mockserver	17838	42996	22645	12441	7636	3975
azure-iot-sdk-java	30302	53566	27047	14428	7902	4424
timely	8011	24140	14222	9232	6091	3949
java-websocket	1585	4285	2217	1215	643	383
rxjava2-extras	9742	15943	8359	4424	2376	1376
shardingsphere-elasticjob	21984	31240	15452	7994	4301	2373
luwak	3284	9851	5117	2717	1507	874
delight-nashorn-sandbox	3244	9045	4561	2417	1351	768
db-scheduler	3753	7905	4746	3053	2019	1292
karate	24540	77182	46197	30253	17492	16055
killbill	29923	158450	76600	40327	21126	11163
fastjson	7783	18534	12415	7046	3998	2353
levitate	9749	8197	4631	2777	1821	1247
textdistance	28620	27531	14498	7875	4542	2683
kevinarpe-rambutan3	877	993	523	280	152	83
pyquery	2081	2014	1035	557	304	175
kb_python	5784	9644	6021	4249	3150	2111
piripherals	4376	4679	2369	1211	648	359
r_map	1070	1384	736	403	227	129
fossor	5921	6560	3465	1804	1021	872
pysshutil	24432	25266	12825	6590	3480	1957
service-manager	2378	2944	1678	1005	597	367
packeteer	1408	1760	936	516	289	160
hotwing-core	2433	4731	3392	2729	2398	2184
pymemorydb	893	1202	631	343	198	114
wiki-futures	1603	1811	935	491	264	150
Min	877	993	523	280	152	83
Max	30302	158450	76600	40327	21126	16055
Median	5852.5	9344.5	4931.5	2915	2197.5	1334
Average	10374.36	21393.75	11397.79	6468.14	3727.89	2374.57

The `pysshutil` (Python) and `Ripme` (Java) projects were the ones that achieved the greatest reduction in total runtime for 100 reruns (again, comparing Naive with the `P16` configuration), with reductions of 91.99% and 90.07%, respectively. This demonstrates a significant reduction in the total runtime when using pod parallelism in the tool.

For all projects, the difference between the total runtime for 100 reruns using Naive and `P16` configuration (the baseline approach and the approach expected to yield the greatest total reduction) was also calculated, followed by the percentage reduction between these two configurations. These data are presented in Table 4.4.

Table 4.4 – Comparison of total time spent to execute one hundred runs between Naive and P16.

Project Name	Naive	P16	Diff.	Red. (%)
ripme	28549	2853	25696	90.07
exhibitor	8319	2059	6260	75.23
mockserver	17838	3975	13863	77.71
azure-iot-sdk-java	30302	4424	25878	85.39
timely	8011	3949	4062	50.72
java-websocket	1585	383	1202	75.83
rxjava2-extras	9742	1376	8366	85.86
shardingsphere-elasticjob	21984	2373	19611	89.21
luwak	3284	874	2410	73.40
delight-nashorn-sandbox	3244	768	2476	76.34
db-scheduler	3753	1292	2461	65.56
karate	24540	16055	8485	34.57
killbill	29923	11163	18760	62.67
fastjson	7783	2353	5430	69.77
levitate	9749	1247	8502	87.22
textdistance	28620	2683	25937	90.61
kevinarpe-rambutan3	877	83	794	90.53
pyquery	2081	175	1906	91.58
kb_python	5784	2111	3673	63.50
piripherals	4376	359	4017	91.80
r_map	1070	129	941	87.94
fossor	5921	872	5049	85.29
pysshutil	24432	1957	22475	91.99
service-manager	2378	367	2011	84.51
packeteer	1408	160	1248	88.65
hotwing-core	2433	2184	249	10.23
pymemorydb	893	114	779	87.22
wiki-futures	1603	150	1453	90.64
Min	877	83	249	10.23
Max	30302	16055	25937	91.99
Median	5852.5	1334	4039.5	85.34
Average	10374.36	2374.57	7999.79	76.93

Subsequently, the mean and median for the percentage of time reduction between the two configurations were calculated. The mean reduction in total execution time for the one hundred runs was 76.93%, while the median was 85.34% for the same values. In Figure 4.2, the graph illustrating the average grouped execution time across all projects is presented, highlighting the reduction in total execution time.

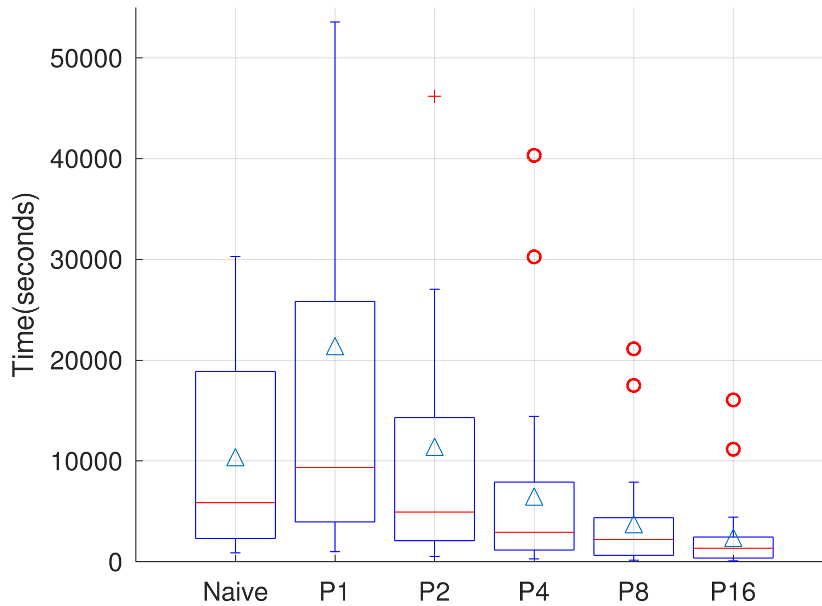

Figure 4.2 – Total runtime (RQ_1).

Figure 4.3 finally presents the graph showing the speed at which the projects are executed. This graph was constructed by normalizing the data through the calculation of the ratio between Naive and each of the parallel configurations. It is noteworthy that, as shown in Figure 4.2, where the total runtime was reduced by almost half as the parallelism increased, in Figure 4.3, the speed of these executions can be seen, following the same trend as the total runtime, i.e., nearly doubling the speed as the parallelism increased.

It is worth noting that, although execution speed increases with greater parallelism, Naive remains faster than P1 configuration, presenting an average and a median of 0.7. This is likely due to the resource constraints, as Naive allows each Docker container to run with all the available resources on the host. As a result, Naive is on average 30% faster than P1, as can also be observed in Figure 4.2.

Starting from P2, FlakyTestLab begins to demonstrate greater speedup relative to Naive, with P2, P4, and P8 presenting average speedups of 1.2, 2.2, and 3.9, and median speedups of 1.3, 2.3, and 4.0, respectively. The highest observed speedup occurred in P16 configuration, which presented an average of 6.6 and a median of 6.8, indicating that, under this configuration, FlakyTestLab is, on average, 6.6 times faster than Naive.

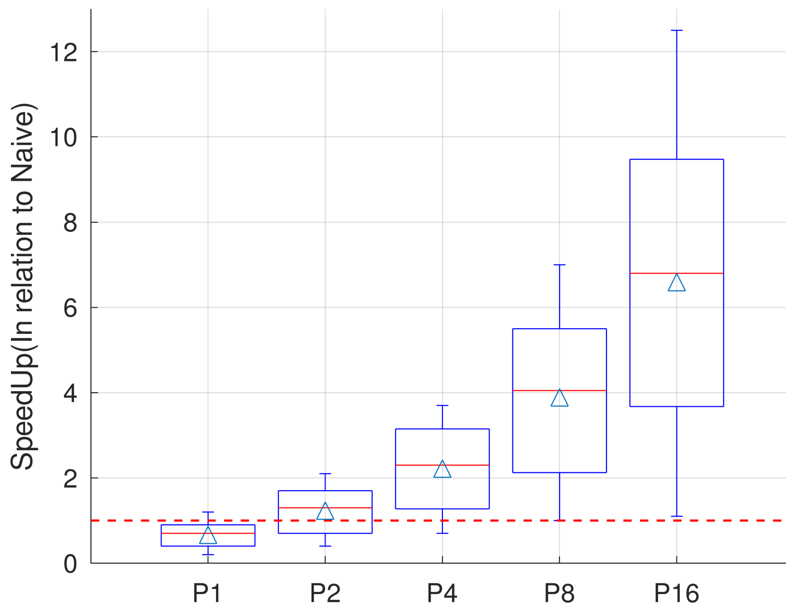

Figure 4.3 – SpeedUp from 100 runs (s).

> *Answer to RQ_1:* By utilizing the isolation provided by Kubernetes, as well as parallelism, the FlakyTestLab tool can accelerate the execution of multiple test reruns across various distinct projects. The execution of 100 rerun is, on average, 76.93% faster when comparing Naive and P16, with the reduction potentially reaching up to 91.99%, depending on the project. However, it may also be only 10.24% faster when the project has a strong dependency on disk, network, or other resources that limit the parallel execution of tests.

4.2.2 RQ_2: To what extent can FlakyTestLab detect the first failure of a flaky test?

In Figure 4.4, the distribution of the time to find the first failure can be observed. It is observed that as parallelism increases, the time to the appearance of the first failure decreases, on average. Figure 4.5 presents the same analysis, but considering only the flaky tests that occurred across all configurations, that is, in Naive and all other FlakyTestLab environments. Observe that for these tests, the average time to the manifestation of the first flaky test is approximately four times faster in P16 when compared to Naive.

Complementing the graphs, Table 4.5 compares the average times for the manifestation of the first flaky test between all tests and those that failed across all FlakyTestLab configurations (tests that failed in Naive, P1, P2, P4, P8, and P16). Notice that for these tests, starting from P2 configuration, the performance in detecting the first failure is alre-

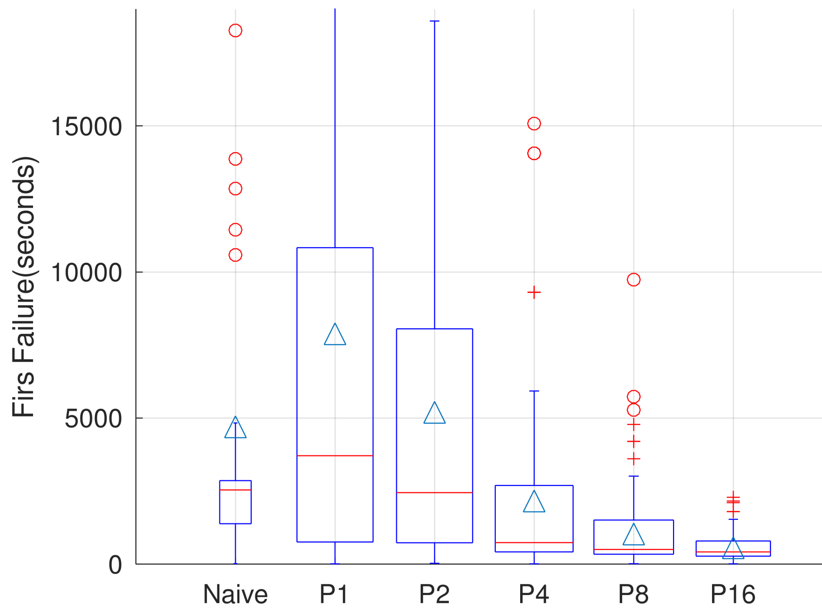

Figure 4.4 – Time to first Failure from all tests (s).

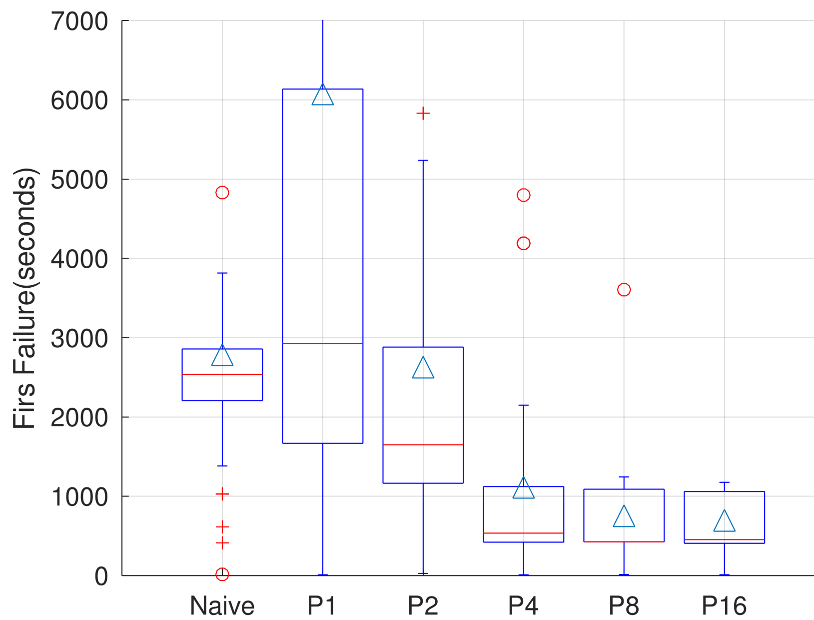

Figure 4.5 – Time to first Failure from tests that failed in all configurations (s).

ady better than Naive, while when considering all tests, this behavior is only true starting from P4.

Figure 4.5 also shows that all configurations have their quartiles located lower in time when compared to Figure 4.4, with the amplitudes also significantly reduced. This

indicates that for the tests that failed in all configurations, FlakyTestLab performed better, as the flaky tests manifested earlier than in Naive.

Table 4.5 – Average time to first failure to all tests and to tests that failed in all configurations.

	Naive	P1	P2	P4	P8	P16
All Tests	4698.24	7883.96	5195.75	2161.42	1031.08	558.68
Tests failed in all	2786.35	6075.075	2629.15	1113.75	754.65	696.95

The speedup in the appearance of the first failure can be seen in Figure 4.6. Note that by using FlakyTestLab, the manifestation of the first flaky test case occurs on average four times faster, with some tests being ten times faster, while others showed no significant improvement, having almost the same time to the manifestation of the first flaky test case as Naive.

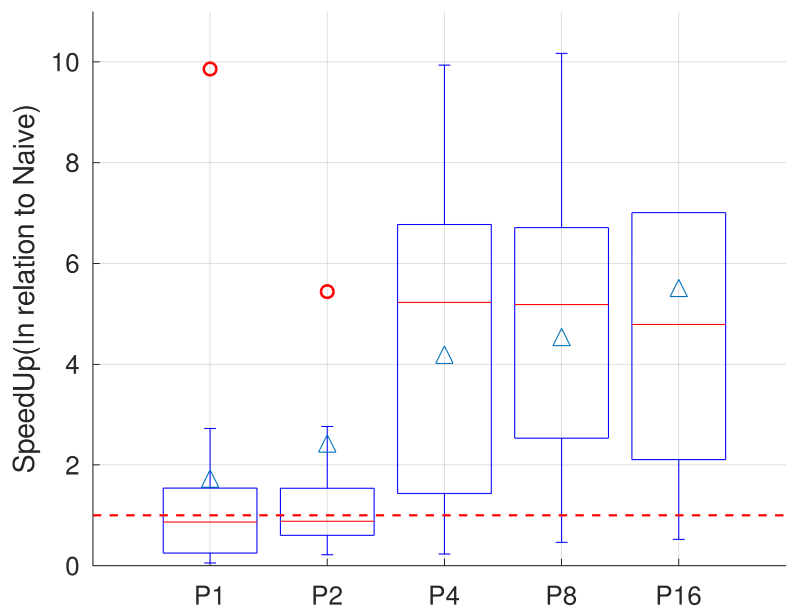

Figure 4.6 – First failure speedup in relation to Naive (Only for tests that failed in all configurations).

Answer to RQ_2: The use of FlakyTestLab may accelerate the first failure manifestation of a flaky test on average by up to four times, potentially reaching ten times, provided that the parallelism of the executions is increased to eight (P8) or to sixteen (P16), depending on the project. It is important to note that not all tests performed better in this regard, with some showing no significant improvement and manifesting almost at the same time as in Naive or even slower, regardless of the parallelism configuration used. The performance improvement in detecting the first failure only occurs after P4 when looking at all tests, and after P2 when focusing on the tests that failed across all configurations.

4.2.3 RQ_3: To what extent does **FlakyTestLab** enhance the manifestation of flaky tests?

Figure 4.7 shows that the number of flaky tests that manifested in FlakyTestLab environments more than doubled for all configurations when compared to Naive, bringing some evidence that the resource constraints on the computational resources used by the Kubernetes pods can enhance the manifestation of test flakiness.

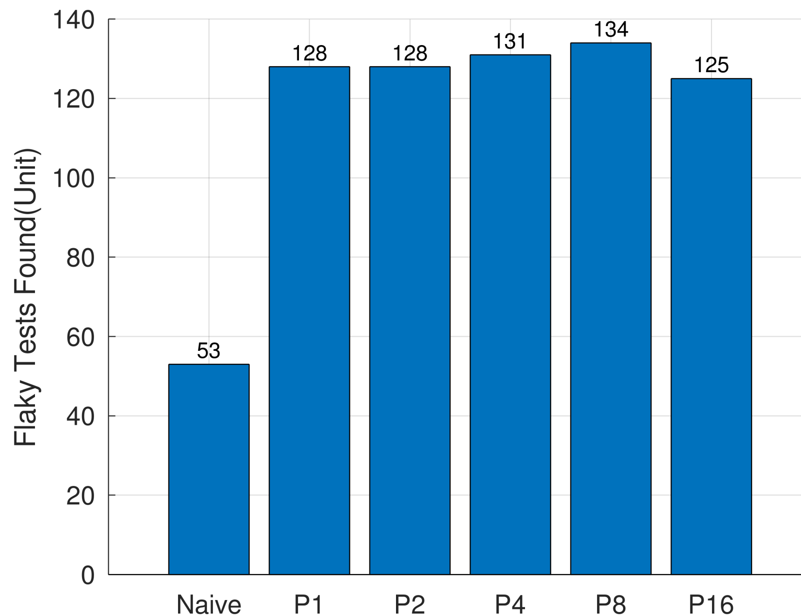

Figure 4.7 – Quantity of Flaky tests found in each configuration.

It can be observed in Table 4.7 and Table 4.6 that, by using FlakyTestLab, the average number of projects that manifested at least one flaky test case was 64.27% for all configurations (average of the percentages presented in Table 4.6), while in Naive, it was 35.71%. Therefore, the manifestation of flaky test cases using FlakyTestLab was, on average, 79.90% higher than Naive.

Table 4.6 – Percentage of projects that flaky tests were detected.

Configuration	Percentage (%)
Naive	35.71
P1	67.86
P2	71.43
P4	57.14
P8	67.78
P16	57.14

It can also be seen in Table 4.7 that out of the 28 projects, 6 did not present any flaky test, even though it is known that all the evaluated projects contain such tests. This indicates that, even with the use of FlakyTestLab, some flaky tests did not manifest. Among the factors that may have contributed to this outcome, the following can be considered: an insufficient number of reruns, since flaky tests are characterized by failing and passing in a non-deterministic manner, it is possible that the 600 reruns (100 in each FlakyTestLab configuration plus 100 in Naive configuration) were not sufficient for the failure to manifest; and the absence of environmental perturbation combined with improved isolation provided by Kubernetes, as ensuring that one execution does not directly affect another may reduce the incidence of certain types of failures.

Considering all projects, a total of 170 different flaky tests were found. Of these, 53 flaky tests were found in Naive, with only two tests (both in `azure-iot-sdk-java` project) being found exclusively in it. One hypothesis for this behavior is that the host size may have affected the tests, since in Naive no resource constraints were imposed; depending on how the tests were implemented, Maven may have used additional threads, thereby affecting components sensitive to concurrency. All the remaining 51 flaky tests were also found in at least one of FlakyTestLab configurations.

Table 4.7 also shows that the project `azure-iot-sdk-java` exhibited the most flaky test cases. For this project, 36 flaky tests were found in Naive, and an average of 65.6 flaky tests were found in FlakyTestLab configurations, indicating that the manifestation of flaky tests in FlakyTestLab was 82.22% higher than in Naive. Projects `rxjava2-extras` and `textdistance` presented the fewest flaky test cases, with only one each. Each of these appeared only once during the five hundred runs across the distinct environments in FlakyTestLab.

In Figure 4.8 and Figure 4.9, the flaky ratio of uncovered flaky tests is presented. In Figure 4.8, the flakiness ratio considering all tests and all projects can be observed. For Naive, the ratio is approximately 0.21, the lowest among all configurations, meaning that the tests identified as flaky failed on average twenty-one times over the one hundred runs.

In Figure 4.9, the data regarding the flake ratio is also presented, but only for the tests that were flaky in all configurations, that is, those that failed in Naive, as well as in P1, P2, P4, P8, and P16 configurations. It can be observed that as parallelism increased,

Table 4.7 – Number of flaky tests reported per project.

Project Name	Naive	P1	P2	P4	P8	P16
ripme	2	8	6	8	8	9
exhibitor	-	1	1	-	-	-
mockserver	-	5	5	5	6	6
azure-iot-sdk-java	36	68	68	67	66	59
timely	4	4	4	4	4	4
java-websocket	-	11	14	19	22	23
rxjava2-extras	-	-	-	-	1	-
shardingsphere-elasticjob	-	-	-	-	-	-
luwak	-	1	2	-	1	3
delight-nashorn-sandbox	-	1	1	1	1	-
db-scheduler	-	3	2	3	5	3
karate	-	4	5	3	3	-
killbill	-	-	-	-	-	-
fastjson	-	-	-	-	-	-
levitate	3	7	4	7	4	4
textdistance	-	-	1	-	-	-
kevinarpe-rambutan3	-	-	-	-	-	-
pyquery	-	-	-	-	-	-
kb_python	-	-	-	-	-	-
piripherals	-	1	1	-	1	-
r_map	1	-	1	1	1	1
fossor	1	1	1	1	1	1
pysshutil	-	2	2	2	1	1
service-manager	-	2	2	1	-	2
packeteer	2	1	-	2	2	1
hotwing-core	2	6	6	6	5	6
pymemorydb	1	1	1	1	1	1
wiki-futures	1	1	1	-	1	1
Total	53	128	128	131	134	125
Min	0	0	0	0	0	0
Max	36	68	68	67	66	59
Median	0	1	1	1	1	1
Average	1.89	4.57	4.57	4.68	4.79	4.46

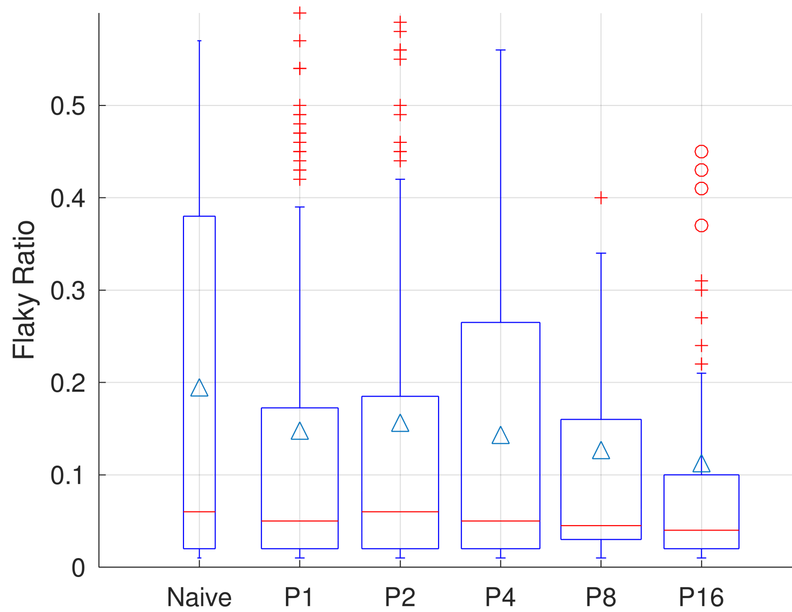

Figure 4.8 – Flaky Ratio to all tests.

the frequency with which the tests manifested repeatedly decreased, from an average of 0.3 in Naive to an average of 0.14 in P16. However, the flaky ratio was slightly higher in P1, P2, and P4.

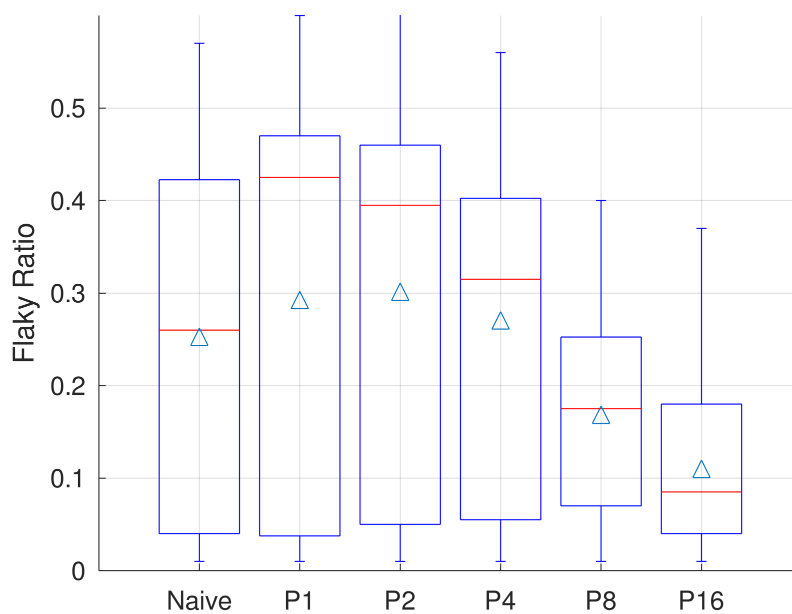

Figure 4.9 – Flaky Ratio to tests that failed in all configuration.

Answer to RQ_3: The use of FlakyTestLab can increase the detection of flaky tests by an average of 143.40% when compared to Naive, with an increase of up to 152.83%, and according to the experiments, at least 141.51%. The flaky ratio may decrease as parallelism increases. The flaky ratio was slightly higher in P1, P2, and P4 configurations, but was significantly reduced in P8 and P16 configurations.

4.3 Discussion

RQ_1 aimed to measure how much the tool could accelerate test reruns. On average, it gave evidence to be 76% faster than Naive, the approach used as a baseline for comparing FlakyTestLab's results. The acceleration in execution was primarily achieved through the parallelism applied to the environments, where up to sixteen pods were run simultaneously, a configuration that yielded the best result. Although most projects improved their execution times by an average of 76%, some performed below 50%, a result that is inferred to be due to limitations in critical resources for the project, such as network, disk, or even an external resource that may have some rate limits, preventing parallel executions from achieving optimal performance. Despite the parallelism, all the pods were executed from the same host and the same source IP.

RQ_2 aimed to determine whether FlakyTestLab could detect the first failure of a flaky test earlier than in Naive configuration, as this can be useful for those seeking to rerun tests until the failure occurs, allowing them to debug through data and logs. An improvement in the time to detect the first failure is an expected behavior, especially when considering RQ_1, where the total execution time is up to 91.99% faster than Naive. Therefore, as the total execution time reduces, the time for the first flaky test to manifest will also be shortened, since the total execution window is smaller. However, it was observed that for this purpose - speeding up the detection of the first failure - simply increasing the parallelism significantly does not guarantee earlier detection. As seen in the speedup of the metric, the time to detect the first failure stabilizes on average for parallelism configurations of four, eight, and sixteen. One hypothesis for achieving earlier detection is perhaps to modify the resources available to the test pods, as this limitation has the potential to accelerate the manifestation of the flaky test.

The goal of RQ_3 was to assess whether FlakyTestLab could increase the number of flaky tests detected during the experiments. A significant improvement was observed in all parallelism configurations, with an average of 143.40% more flaky tests detected compared to Naive. Based on our analyses, resource constraints were the main cause, as the average number of flaky tests manifested was consistent across all parallelism configurations, showing little variation between them, as presented in Table 4.7. Unlike RQ_1, where increasing the execution speed required increasing the parallelism, increasing the number of flaky tests does not necessarily require an increase in parallelism. This can

be seen in Figure 4.7 and Table 4.7, where some projects, such as `azure-iot-sdk-java`, `Ripme`, and `hotwing-core`, maintained a consistent average of flaky test manifestation across all configurations. Once again, it is hypothesized that, for the purpose of detecting more flaky test cases, it may be more efficient to better control the computational resources allocated to the pods, similarly to what was suggested for RQ_2 and investigated in Silva et al. (2024). It is important to note that, despite the increased detection of flaky tests in FlakyTestLab, the flaky ratio decreased on average as parallelism increased. This indicates that, to maximize the number of flaky tests manifested during testing, parallelism should not be increased excessively, as doing so may result in fewer detections than intended.

FlakyTestLab includes mechanisms to restrict computational resources, which, based on our analyses and as investigated in Silva et al. (2024), can improve the flaky ratio. The use of the tool demonstrated that simply running it without increasing parallelism, but instead restricting resources (P1), increased the quantity of detected tests. FlakyTestLab, however, is limited by the amount of resources available on the host, as ensuring the configured resources for the environments requires the host to have such resources.

Some testing tools, such as the Maven Surefire Plugin, allow tests to be executed in parallel; however, these approaches do not provide the same level of isolation as FlakyTestLab, which is achieved through Kubernetes. As a result, variables may be read and written concurrently, leading to concurrency and race condition issues, as well as other situations that may exhibit flaky behavior. However, such behavior may arise solely from the lack of isolation in the testing tool and therefore does not necessarily represent a genuinely flaky test. Furthermore, if desired by the developer, the same parallelism can be applied in conjunction with FlakyTestLab.

A brief comparison between FlakyTestLab and Shaker (CORDEIRO et al., 2021) shows that, while both tools share the similar goals, FlakyTestLab additionally aims to improve speedup and test isolation, as well as increase the percentage of detected flaky tests. Shaker also is focused on Java systems using Maven or on Python systems (at least at the time of writing, according to the authors). It is aimed at instabilities caused by concurrent behavior and was evaluated using a set of predefined perturbation configurations, which limits the range of projects it can affect. At the same time, the average test execution time may increase, since the tool introduces noise, thereby increasing system overhead. Given these observations, FlakyTestLab proposes a slightly more flexible model, with somewhat similar behavior (it restricts system resources but does so directly, without introducing additional overhead) and that, depending on the parallelism settings, allows for significantly faster test execution. It is worth noting that both tools can be used in combination, as the perturbations introduced by Shaker can be combined with the parallelism and computational resource constraints of FlakyTestLab, thereby increasing the potential for detecting flaky tests in experiments.

The monitoring components presented in Chapter 3 were extensively used during the experiments. Grafana, Prometheus, and Metrics Server were essential for determining the minimum environment size required to execute the tests. Prior to large-scale test execution, the projects were run several times, and their resource consumption was monitored using the aforementioned tools. Based on the data analysis, an appropriate environment size was defined to ensure that the tests could be executed without unintended interruptions, such as OOMKilled (Out of Memory Killed). These data are presented in Table 4.1.

Grafana, Loki, and Vector were essential for post hoc data analysis. Once the experiments had been defined and executed, these tools provided the necessary means to make logs available for analysis even after the executions had completed. Grafana enabled real-time visualization of pod resource consumption and their corresponding logs. Vector collected these logs and forwarded them to Grafana Loki and to volumes mounted on the host. This made it possible to develop scripts that traversed all logs and report files, generating several CSV tables. This entire infrastructure enabled the data to be analyzed and presented herein.

Due to its flexibility and ease of configuration, FlakyTestLab can be integrated into and used with CI/CD tools, such as Jenkins, and can also be employed in university or institutional laboratory environments as a framework for batch test execution, providing support for log storage, organization, and visualization. Furthermore, the entire infrastructure, including the monitoring stack, enables a wide range of experiments to be conducted. Analyses of application behavior, resource consumption profiling, and the correlation between the occurrence of flaky tests and resource usage at a given moment, among other experiments, can be carried out using the tool.

4.4 Concluding Remarks

This chapter presented all data related to the experiments and analyses conducted in this thesis. All components used to perform the experiments and to collect and analyze the data were described in detail. The repositories containing FlakyTestLab and Naive scripts were made available. All (RQs) were presented, including their objectives, characteristics, and justifications. Experimental data were summarized and presented in tables and graphs, and subsequently analyzed. Finally, a discussion was provided highlighting relevant observations and behaviors, along with a brief comparison with Shaker (CORDEIRO et al., 2021). In the following chapter, the conclusion of this thesis is presented. The chapter also presents proposals for future work, offering directions and new avenues of investigation for the continuation and further development of the tool, as well as for new experiments.

Chapter 5

Conclusion

This thesis presented the FlakyTestLab tool, which was developed with the aim of providing a flexible and robust environment to meet the demands related to software testing, specifically flaky detection via reruns. The tool is open to the community and can be installed on a standard host for specific tests or on a workstation, where it can handle various testing demands. This tool was built using a set of DevOps tools, which can be executed with just a few commands and provides an environment prepared to execute and manage multiple runs of different types of projects.

To evaluate the tool, 28 projects were used to understand whether it can be applied for the proposed purposes, primarily through the three RQs. The results were promising, showing substantial gains in some of the evaluated scenarios. The RQs mainly focused on execution parallelism, where environments with different parallelism configurations were created, as it was expected that parallelism would allow the same number of runs to be executed in less time. The experiments demonstrated a gain in total execution time, allowing the same number of tests to be run up to nearly ten times faster. It was also possible to detect a flaky test failing earlier than with the baseline approach and to more than double the number of flaky tests detected. There are indications that the tool's functionality, which allows for the management and restriction of resources for the pods, can help the user achieve objectives related to detecting a higher number of flaky tests, as well as detecting them faster compared to series reruns.

5.1 Future Work

We observed that the tool is flexible in terms of allowing a wide range of configurations, as it is possible to combine various parameters, such as CPU, RAM memory, Docker image

to be used in the pods, execution parallelism, and test parameters. This enables the tool to be used for various analyses and purposes. Therefore, the following work is suggested as future possibilities for this tool.

- Evaluate other configurations: As demonstrated, there is a hypothesis that varying the resources of the pods can significantly impact the test results, potentially creating environments more favorable to the manifestation of flaky test cases (TERRAGNI; SALZA; FERRUCCI, 2020; SILVA et al., 2024) . So far, only parallelism has been evaluated in more detail, but there are other characteristics that could alter the behavior of the results;

- Combining FlakyTestLab with other tools, such as Shaker (CORDEIRO et al., 2021): As presented in this thesis, FlakyTestLab is flexible and accepts various parameters. Among these, it is possible to change the Docker image used for the experiments. This allows FlakyTestLab to be combined with other tools, such as Shaker, since the latter can be executed using a custom Docker image. The same approach can be applied to other projects, enabling a wide range of diverse experiments to be conducted.

- FlakyTestLab in cluster of computers: FlakyTestLab was developed to facilitate the detection of flaky tests through reruns. One of its features is the ability to execute reruns in parallel. However, the degree of parallelism is constrained by the capacity of the host on which the tool is executed. It may be possible to distribute pods across multiple hosts, thereby increasing the available parallelism, including through cloud providers such as Amazon Web Services (AWS) or Google Cloud Platform (GCP). It is important to note that FlakyTestLab was implemented using Kind, which provides a way to run Kubernetes locally on a single node. Kind exists because the opposite - running Kubernetes in cluster mode - is the conventional way to use Kubernetes. Therefore, it is important to preserve certain features presented here, such as the observability layer, the collection and organization of logs, and the ease of access and execution of the tool.

- Evaluate FlakyTestLab with other programming languages: FlakyTestLab was designed to be as language-agnostic as possible; however, only Java and Python projects were analyzed. Several other programming languages could be evaluated.

- Use the tool for different purposes: As noted, the tool allows the creation of various environments with different configurations for executing and detecting flaky tests. However, the tool also offers various resources, such as Prometheus, Grafana, and others, which could enable it to be used for other purposes, such as controlled execution of workers, parallel task processing, running processes in environments

with controlled resources, and many other experimental possibilities that can be carried out with FlakyTestLab.

References

ALSHAMMARI, A. et al. Flakeflagger: Predicting flakiness without rerunning tests. In: IEEE. **2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)**. [S.l.], 2021. p. 1572–1584.

AMMANN, P.; OFFUTT, J. **Introduction to software testing**. [S.l.]: Cambridge University Press, 2017.

ANDERSON, C. Docker [software engineering]. **Ieee Software**, IEEE, v. 32, n. 3, p. 102–c3, 2015.

BAIER, J. **Getting started with kubernetes**. [S.l.]: Packt Publishing Ltd, 2017.

BARBOSA, E. F. et al. Introdução ao teste de software. **Minicurso apresentado no XIV Simpósio Brasileiro de Engenharia de Software (SBES 2000)**, 2000.

BELL, J.; KAISER, G. Unit test virtualization with vmvm. In: **Proceedings of the 36th International Conference on Software Engineering**. [S.l.: s.n.], 2014. p. 550–561.

BELL, J. et al. Deflaker: Automatically detecting flaky tests. In: **Proceedings of the 40th international conference on software engineering**. [S.l.: s.n.], 2018. p. 433–444.

BERNARDO, P. C.; KON, F. A importância dos testes automatizados. **Engenharia de Software Magazine**, v. 1, n. 3, p. 54–57, 2008.

BERNSTEIN, D. Containers and cloud: From lxc to docker to kubernetes. **IEEE Cloud Computing**, IEEE, v. 1, n. 3, p. 81–84, 2014.

BERTOLINO, A. Software testing research: Achievements, challenges, dreams. In: IEEE. **Future of Software Engineering (FOSE'07)**. [S.l.], 2007. p. 85–103.

BEYER, B. et al. **Site reliability engineering: how Google runs production systems**. [S.l.]: "O'Reilly Media, Inc.", 2016.

BRAZIL, B. **Prometheus: up & running: infrastructure and application performance monitoring**. [S.l.]: "O'Reilly Media, Inc.", 2018.

BURNS, B. et al. Borg, omega, and kubernetes. **Communications of the ACM**, ACM New York, NY, USA, v. 59, n. 5, p. 50–57, 2016.

COELHO, M. et al. Software developers' perceptions of productivity: An industry-focused study. In: SOCIEDADE BRASILEIRA DE COMPUTAçãO (SBC). **Proceedings of the XXIV Brazilian Symposium on Software Quality (SBQS 2025)**. São José dos Campos, Brazil, 2025. p. 12–22. Peer-reviewed conference paper.

CORDEIRO, M. et al. Shaker: a tool for detecting more flaky tests faster. In: IEEE. **2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.], 2021. p. 1281–1285.

CORDY, M. et al. Flakime: laboratory-controlled test flakiness impact assessment. In: **Proceedings of the 44th International Conference on Software Engineering**. [S.l.: s.n.], 2022. p. 982–994.

CoreDNS Authors. **CoreDNS Documentation**. 2023. Accessed on September 23, 2025. Available in: <`https://coredns.io/docs/`>.

CRISPIN, L.; GREGORY, J. **Agile testing: A practical guide for testers and agile teams**. [S.l.]: Pearson Education, 2009.

DIAS-NETO, A. C. Introdução a teste de software. **Engenharia de Software Magazine**, v. 1, p. 22, 2007.

EBERT, C. et al. Devops. **IEEE software**, Ieee, v. 33, n. 3, p. 94–100, 2016.

ECK, M. et al. Understanding flaky tests: The developer's perspective. In: **Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. [S.l.: s.n.], 2019. p. 830–840.

FOWLER, M.; HIGHSMITH, J. et al. The agile manifesto. **Software development**, [San Francisco, CA: Miller Freeman, Inc., 1993-, v. 9, n. 8, p. 28–35, 2001.

GAROUSI, V.; ELBERZHAGER, F. Test automation: A survey of the guidelines and frameworks. **IEEE Software**, v. 33, n. 6, p. 68–76, Nov 2016. ISSN 0740-7459.

GITHUB. **GitHub Actions: Virtual Machines**. 2023. Accessed on September 23, 2025. Available in: <`https://docs.github.com/fr/enterprise-cloud@latest/actions/reference/runners/github-hosted-runners#standard-github-hosted-runners-for-public-repositories`>.

Grafana Authors. **Grafana Documentation**. 2023. Accessed on August 24, 2025. Available in: <`https://grafana.com/docs/`>.

_____. **Grafana Loki Documentation**. 2023. Accessed on September 19, 2025. Available in: <`https://grafana.com/docs/loki/latest/`>.

Grafana Labs. **Grafana Loki - A Log Aggregation System**. 2026. Accessed: 2026-01-17. Available in: <`https://grafana.com/oss/loki/`>.

GRUBER, M.; FRASER, G. Debugging flaky tests using spectrum-based fault localization. In: **IEEE/ACM International Conference on Automation of Software Test, AST 2023, Melbourne, Australia, May 15-16, 2023**. IEEE, 2023. p. 128–139. Available in: <`https://doi.org/10.1109/AST58925.2023.00017`>.

______. Flapy: Mining flaky python tests at scale. In: IEEE. **2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)**. [S.l.], 2023. p. 127–131.

HASHEMI, N. et al. Detecting and evaluating order-dependent flaky tests in javascript. In: **Proceedings of the IEEE Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2025. Available at arXiv:2501.12680.

Helm Project. **Helm Documentation**. 2023. Accessed on: 20 fev. 2026. Available in: <https://helm.sh/docs/>.

HENDERSON, T. A. D. et al. Flake aware culprit finding. In: **Proceedings of the 16th IEEE International Conference on Software Testing, Verification and Validation (ICST 2023)**. [s.n.], 2023. Available in: <https://hackthology.com/flake-aware-culprit-finding.html>.

IEEE. **IEEE Std 610.12-1990: IEEE standard glossary of software engineering terminology**. [S.l.]: IEEE, 1990.

IQBAL, H.; BEGUM, Z.; SAKIB, K. Reduction of test re-runs by prioritizing potential order dependent flaky tests. In: **2025 IEEE/ACM International Flaky Tests Workshop (FTW)**. [S.l.: s.n.], 2025. p. 1–8.

KANER, C.; FALK, J.; NGUYEN, H. Q. **Testing computer software**. [S.l.]: John Wiley & Sons, 1999.

KIM, G. et al. **The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations**. [S.l.]: It Revolution, 2021.

Kubernetes Authors. **Kubernetes Blog**. 2023. Accessed on: 23 set. 2025. Available in: <https://kubernetes.io/blog/2016/10/helm-charts-making-it-simple-to-package-and-deploy-apps-on-kubernetes/>.

______. **Metrics Server**. 2023. Accessed: 2026-01-03. Available in: <https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/>.

Kubernetes Contributors. **Pod Quality of Service (QoS)**. 2023. Accessed on eptember 23, 2025. Available in: <https://kubernetes.io/docs/concepts/workloads/pods/pod-qos/>.

Kubernetes SIGs. **Kubernetes in Docker (KIND)**. 2026. Accessed on February 23. 2026. Available in: <https://kind.sigs.k8s.io/>.

LAM, W. et al. Root causing flaky tests in a large-scale industrial setting. In: **Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis**. [S.l.: s.n.], 2019. p. 101–111.

______. A study on the lifecycle of flaky tests. In: **Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering**. [S.l.: s.n.], 2020. p. 1471–1482.

______. idflakies: A framework for detecting and partially classifying flaky tests. In: IEEE. **2019 IEEE 12th International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.], 2019. p. 312–322.

______. Understanding reproducibility and characteristics of flaky tests through test reruns in java projects. In: IEEE. **2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)**. [S.l.], 2020. p. 403–413.

______. Understanding reproducibility and characteristics of flaky tests through test reruns in java projects. In: VIEIRA, M. et al. (Ed.). **31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020**. IEEE, 2020. p. 403–413. Available in: <https://doi.org/10.1109/ISSRE5003.2020.00045>.

LEINEN, F.; PERATHONER, A.; PRETSCHNER, A. On the impact of hitting system resource limits on test flakiness. In: **Proceedings of the 1st International Workshop on Flaky Tests, FTW 2024, Lisbon, Portugal, 14 April 2024**. ACM, 2024. p. 14–19. Available in: <https://doi.org/10.1145/3643656.3643898>.

LOUKIDES, M. **What is DevOps?** [S.l.]: "O'Reilly Media, Inc.", 2012.

LUO, Q. et al. An empirical analysis of flaky tests. In: **Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering**. [S.l.: s.n.], 2014. p. 643–653.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. **Linux journal**, v. 2014, n. 239, p. 2, 2014.

MORÁN, J. et al. Flakyloc: flakiness localization for reliable test suites in web applications. **Journal of Web Engineering**, River Publishers, v. 19, n. 2, p. 267–296, 2020.

MYERS, G. J. et al. **The art of software testing**. [S.l.]: Wiley Online Library, 2004. v. 2.

PARRY, O. et al. A survey of flaky tests. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM New York, NY, v. 31, n. 1, p. 1–74, 2021.

______. Evaluating features for machine learning detection of order- and non-order-dependent flaky tests. In: **15th IEEE Conference on Software Testing, Verification and Validation, ICST 2022, Valencia, Spain, April 4-14, 2022**. IEEE, 2022. p. 93–104. Available in: <https://doi.org/10.1109/ICST53961.2022.00021>.

PINTO, G. et al. What is the vocabulary of flaky tests? In: **Proceedings of the 17th International Conference on Mining Software Repositories**. [S.l.: s.n.], 2020. p. 492–502.

PRESSMAN, R. S.; MAXIM, B. R. **Engenharia de software-9**. [S.l.]: McGraw Hill Brasil, 2021.

Prometheus Authors. **Prometheus Documentation**. 2023. Accessed on September 01, 2025. Available in: <https://prometheus.io/docs/>.

_____. **Prometheus StatsD Exporter Documentation**. 2023. Accessed on September 23, 2025. Available in: <https://github.com/prometheus/statsd_exporter>.

RAHMAN, S. et al. Automatically reproducing timing-dependent flaky-test failures. In: **International Conference on Software Testing, Verification, and Validation**. [S.l.: s.n.], 2024.

RAHMAN, S.; SHI, A. Flakesync: Automatically repairing async flaky tests. In: **Proceedings of the IEEE/ACM 46th International Conference on Software Engineering**. [S.l.: s.n.], 2024. p. 1–12.

SILVA, D. et al. The effects of computational resources on flaky tests. **IEEE Trans. Software Eng.**, v. 50, n. 12, p. 3104–3121, 2024. Available in: <https://doi.org/10.1109/TSE.2024.3462251>.

TERRAGNI, V.; SALZA, P.; FERRUCCI, F. A container-based infrastructure for fuzzy-driven root causing of flaky tests. In: ROTHERMEL, G.; BAE, D. (Ed.). **ICSE-NIER 2020: 42nd International Conference on Software Engineering, New Ideas and Emerging Results, Seoul, South Korea, 27 June - 19 July, 2020**. ACM, 2020. p. 69–72. Available in: <https://doi.org/10.1145/3377816.3381742>.

TESTINGRESEARCHILLINOIS. **International Dataset of Flaky Tests (IDoFT)**. 2025. Accessed on September 24, 2025. Available in: <https://github.com/TestingResearchIllinois/idoft>.

TURNBULL, J. **The Docker Book: Containerization is the new virtualization**. [S.l.]: James Turnbull, 2014.

Vector. **Vector - A High-Performance Observability Data Pipeline**. 2026. Accessed: 2026-01-17. Available in: <https://vector.dev/>.

WALKER, R. Examining load average. **Linux Journal**, Belltown Media Houston, TX, v. 2006, n. 152, p. 5, 2006.

WONG, W. E. et al. A survey on software fault localization. **IEEE Transactions on Software Engineering**, IEEE, v. 42, n. 8, p. 707–740, 2016.

ZAMPETTI, F. et al. Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study. In: IEEE. **2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.], 2021. p. 471–482.

ZHU, L.; BASS, L.; CHAMPLIN-SCHARFF, G. Devops and its practices. **IEEE software**, IEEE, v. 33, n. 3, p. 32–34, 2016.