Bachelor

June 14, 2021

An isolated containerized infrastructure for flakiness discovery

Implementation of a Prototype

Fabio Greter

of Lucerne, Switzerland (13-760-913)

supervised by

Prof. Dr. Harald C. Gall
Dr. Pasquale Salza
Dr. Valerio Terragni

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor

An isolated containerized infrastructure for flakiness discovery

Implementation of a Prototype

Fabio Greter

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor

Author: Fabio Greter, fabio.greter@uzh.ch

URL: https://github.com/flaky-infrastructure

Project period: 14.12.2020 - 14.06.2021

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

This thesis was supported by the Seal institute at the University of Zurich. I would like to thank Prof. Dr. Harald C. Gall for his sponsorship and Dr. Pasquale Salza for his excellent supervision. Additionally, Dr. Valerio Terragni from the University of Auckland has also provided crucial support in this prototype implementation of their proposed architecture. Together, Dr. Salza and Dr. Terragni have helped me keep the implemented prototype aligned with what they proposed originally.

This thesis interfaces with another thesis by Dylan Puser, and I thank him for the excellent collaboration.

Finally, I would like to thank my family, my girlfriend, and my friends for their support while working on this project.

Abstract

As software projects increase in complexity, testing becomes increasingly important. It is essential that tests can be built to ensure that a test failure reliably indicates problems in production code, which can then easily be fixed. Because modern software systems have become inherently non-deterministic, intermittent test failures are also becoming more frequent, in what is known as flaky tests. Existing techniques to remedy this problem focus on efficient detection of flaky tests without identifying the root causes of their intermittent behaviour, or are specific to certain root causes. Other approaches rely on instrumentation of the production code, which may affect test outcomes.

In this thesis, we present a prototype implementation of an architecture to induce test flakiness by executing tests under various circumstances, called execution scenarios. Our prototype allows for flexible implementation of these scenarios and provides an API to be used in various environments, such as continuous integration pipelines. We find that it can reproduce known flaky test behaviour, and that in some cases, our prototype execution scenarios can exhibit different failure rates for certain test cases. We also propose future enhancements to continue development on the prototype.

Zusammenfassung

Da Softwareprojekte zunehmend komplexer werden, wird das Testen ebenfalls immer wichtiger. Es ist essentiell, dass Tests so konstruiert werden können, dass sie im Falle eines Versagens zuverlässig Probleme im Produktions-Code indizieren, die dann einfach behoben werden können. Da moderne Softwaresysteme aber zunehmend undeterministisches Verhalten zeigen, können Testfälle intermittierend fehlschlagen. Dieses Phänomen ist als "Flaky Test" bekannt. Aktuelle Techniken um dieses Problem zu beheben, konzentrieren sich vor allem auf effiziente Entdeckung der Flaky Tests, ohne ihre Ursachen zu identifizieren, oder können nur spezifische Ursachen entdecken. Weitere Ansätze verlassen sich auf Instrumentierung des Produktionscodes, was Testresultate beeinflussen kann.

In dieser Arbeit präsentieren wir eine Prototyp-Implementierung einer Architektur, die gezielt "Test Flakiness" hervorrufen soll, indem die Tests unter verschiedenen Bedingungen, genannt "Execution scenario", ausgeführt werden. Unser Prototyp erlaubt eine flexible Implementierung dieser Szenarios und stellt eine API zur Verfügung, die in verschiedenen Umgebungen benutzt werden kann, zum Beispiel in Continuous Integration Pipelines. Wir zeigen, dass unser Prototyp bekannte Flaky Tests reproduzieren kann, und dass unsere Execution scenarios in einigen Fällen unterschiedliches Verhalten zeigen. Desweiteren zeigen wir verschiedene zukünftige Erweiterungen auf, um unseren Prototyp weiterzuentwickeln.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Software systems are constantly evolving and growing in complexity. They are being moved to the cloud, expanded to make use of increasing CPU core counts, and distributed across many nodes in diverse geographical locations. Modern software behaviour has become increasingly non-deterministic as a result of this development. This non-deterministic behaviour can lead to intermittent test failures, in so-called "Flaky Tests". Flaky tests are test cases that neither fail nor pass all the time. They can cause failures in continuous integration pipelines which are hard to debug, since the developers may not be easily able to reproduce the behaviour or discern the reasons why their tests suddenly fail. If the intermittent test results are caused by issues in the production code, rather than the test implementation itself, they also highlight issues that may occur in production.

Flaky tests have been shown to be caused by specific types of weaknesses in either the test code or the production code itself. These types of weaknesses are called “root causes”, and specific typical root causes have already been identified [1].

This has been noted as an open challenge for the field of software testing [2, 3]. So far, approaches focused mainly on increasing efficiency in the detection of flaky tests, but there is a lack of methods to find the root causes of flaky tests. Identifying root causes quickly can speed up the developer’s response to sudden flakiness in their tests, preventing the erosion of trust that can occur when flaky tests show up [2, 4].

Terragni et al. [2] have proposed an infrastructure for targeted execution of flaky tests under different environments, called "execution scenario". These scenarios vary the environment the tests are executed within, with each corresponding to a common root cause of flaky tests. In order to determine the most likely root cause, each scenario is compared to a baseline. This information can then be used to quickly identify and fix weaknesses in tests.

According to their proposal, the tests are to be run inside software containers, which can be executed dynamically with different environment configurations, and in which so-called "fuzzy loaders" create different scenarios, such as high CPU load, high network usage, concurrent database operations, etc. A functioning implementation of this proposal allows for a detailed root cause analysis of flaky test behaviour.

The main challenges for this implementation are as follows:

- the environment for the containers to run in needs to be isolated as much as possible in order to exclude external influences on test results. Multiple test containers should be allowed to run in parallel to speed up data gathering of test behaviour;
- the containers need to be as platform-agnostic as possible. The software that needs to be tested might come in various programming languages and frameworks, and the same should be possible for the fuzzy loaders;

- the results need to be collected and sent out for analysis in a way that creates as little overhead as possible for running the tests;

Terragni et al. [2] envision this infrastructure to be used in two main use cases:

1. as part of a continuous integration pipeline, for detection and root-cause analysis [2];
2. after continuous integration, when flaky tests have been detected, for more targeted analysis [2];

Our contribution in this work is to lay out an ideal approach for this infrastructure, from now on referred to in this thesis as "flakiness inducer infrastructure", and to show that the approach is feasible to implement. To this end, we have created a prototype, which we will describe in further detail. Finally, we show a preliminary evaluation of the prototype and lay out future work on the project.

This thesis is structured as follows: In chapter 2, we show the current state of the art in research into flaky tests, and identify the gaps that our approach intends to fill. In chapter 3, we provide background on the concrete tools used to implement the prototype. The requirements for our infrastructure, an ideal approach and our concrete implementation are all described in detail in chapter 4. Chapter 5 details the approaches we used to validate our prototype implementation, with the results in chapter 6. We draw a conclusion of our contribution in chapter 7 and elaborate on future work in chapter 8.

Chapter 2

Related work

Research into intermittent test behaviour is still fairly young, but recently, interest in flaky tests has increased. Earlier research by Bell et al. [5] focused mainly on efficient identification of flaky tests, for example through monitoring code changes, and marking tests as flaky when they fail without changes in the executed production code.

For this work, research into root-cause analysis of flaky tests is of particular interest. Zheng et al. [6] have identified several current approaches:

2.1 Code instrumentation and log analysis

Some approaches to detecting flaky tests and finding their root causes include code instrumentation. Lam et al. [7] have put forward an end-to-end framework to detect flaky tests and find their root causes. They add log statements to test code and relevant passages in the production code, and then analyze these logs to identify root causes. A similar method recently proposed by Ziftci et al. [8] also instruments code, and then detects the points of divergence between passing and failing executions. The information thus obtained is then provided to developers, requiring manual root-cause analysis. Both of these approaches also face the challenge that flakiness may only occur on non-instrumented versions of the code.

2.2 Direct root-causing of flaky tests

Other approaches attempt to directly cause specific types of intermittent test behaviour. Since many common root causes of flaky tests are known [1], these approaches try to directly affect test outcomes by executing them in specific ways.

Test dependency analysis. There are several approaches to detecting state pollution and resulting test order dependencies, which is one common root cause of flaky tests [1]. Recent approaches include the PRADET tool, created by Gambi et al. [9], and the iDFlakies tool by Lam et al. [4], which both classify flaky tests that are caused by test execution order.

Concurrency-related analysis. Silva et al. [10] have proposed SHAKER, a tool to execute tests in parallel with stressor tasks that load CPU and memory. They have found promising results, detecting more flaky tests with higher failure rates than by simply re-running the tests in hopes of randomly causing test failures.

2.3 Containerized infrastructure for root-causing

Terragni et al. [2] have proposed a method to execute flaky tests under various execution scenarios, each corresponding to a specific root cause. They envision that by executing a flaky test under a scenario that corresponds to its specific root cause, they can impact its failure rate and detect a deviation from a base scenario, which represents a typical execution environment. The execution scenarios in their proposal can also contain so-called "fuzzy loaders", which dynamically load resources to affect test execution. This approach has some similarities to other approaches to root-cause flaky tests, but differs in its concrete implementation. Instead of focusing on specific root causes, the method put forward by Terragni et al. [2] focuses on providing a scalable infrastructure in which tests can be executed under many different scenarios, and they elaborate on specific ones for the most common root causes:

Multi-threaded execution scenario. This scenario "explores the non-deterministic interleaving space of concurrent executions" [2]. It root-causes test failures related to concurrency by executing tests with different amounts of cores, and with fuzzy loaders that load the CPU with dummy operations. The approach is similar to the one by Silva et al. [10], but also varies CPU core count in addition to stressor tasks.

Network execution scenario. This scenario "explores the non-deterministic space of the network latency and response time" [2]. It executes the tests together with fuzzy loaders that create network loads, finding test failures that occur due to slow or failed network connections.

I/O execution scenario. This scenario "explores the non-determinism of I/O operations" [2]. It varies disk space allocations for tests and loads the disk with fuzzy loaders that read and write files randomly to contest for file system resources.

Test order execution scenario. This scenario "explores the non-determinism caused by test dependencies" [2]. By altering the test execution order randomly, flaky tests that are caused by state corruption will be identified.

Platform scenario. This scenario "explores the non-determinism caused by different execution platforms" [2]. Some flaky tests exhibit failures primarily with specific versions of frameworks or libraries, or on different operating systems. By varying these parameters, those types of flaky tests can be identified.

Root-cause analysis. According to the proposal by Terragni et al. [2], for identification of flaky test root causes, the tests are executed on the base scenario as well as all the desired other execution scenarios. They define the failure rate of a test case t on a specific scenario ε_k as:

$$\lambda_t^k = \frac{\sum_{i=1}^{N} exec_i(\varepsilon_k, t)}{N} \in [0; 1] \tag{2.1}$$

$exec_i(\varepsilon_k, t)$ denotes the outcome of the i^{th} execution of the test case t under execution scenario ε_k, which is either zero, if the test passed, or one, if it failed. According to Terragni et al. [2], the most likely root cause is given by the scenario with the highest deviation in failure rate from the

base scenario. If the test does not fail on the base scenario, this is just $\max_k \lambda_t^k$. If the test does fail on the base scenario, it is $\max_k |\lambda_t^k - \lambda_t^0|$, where λ_t^0 is the failure rate on the base scenario.

In this thesis, we discuss a prototype implementation of the infrastructure envisioned by Terragni et al. [2]. This prototype provides an interface for various frontend applications to interact with, such as graphical user interfaces for data analysis, or applications that implement it in a continuous integration process. These frontends are expected to exist.

Chapter 3

Background

In our prototype implementation, we relied on many other technologies related to container orchestration. Here, we provide a brief overview of essential concepts.

3.1 Container orchestration

For container orchestration, we rely on Kubernetes[1]. Kubernetes is a widespread tool to facilitate deployment of containerized applications. Since our entire application stack is designed to run within a Kubernetes cluster, we have made use of Kubernetes services and deployments for the implementation.

For the containers in which the tests are executed, we make use of Kubernetes' well-documented API to create them as Pods.

3.1.1 Resource limits

For some of the execution scenarios proposed by Terragni et al. [2], we rely on Kubernetes' ability to manage resources for containers [11]. Since our system depends on minimal interference by factors we do not actively control as part of the scenarios themselves, we also use its static CPU assignment policy [12] to isolate our containers from each other, as well as any other processes running on the nodes which execute them. This policy assigns containers to specific CPU cores. Under the hood, these limits and assignment policies use the cgroups feature of the Linux kernel.

3.1.2 Container building within containers

In our implementaiton, we wrap the tested software together with our custom test executors and fuzzy loaders in a container image. We build these using Google's Kaniko [13], which allows us to use the Kubernetes cluster which our tools run in to also build these images. Kaniko does not rely on Docker to build images, which simplifies this process, since running Docker within Docker, the typical approach for building container images within containers, requires elevated access for that container [14].

[1] `https://kubernetes.io`

3.1.3 RabbitMQ

Since we implemented the prototype in a microservice architecture and aim to provide a flexible API, we are using a message queue for both the API provided by our infrastructure as well as internal communication between the services. This allows for easy scaling of the infrastructure, adding more pods to the deployments for heavily loaded services. In particular, we used RabbitMQ[2].

[2] https://www.rabbitmq.com

Chapter 4

Approach

In this chapter, we outline the requirements and specific challenges for our infrastructure. We continue by proposing a model of an ideal system, followed by a description of our concrete prototype implementation. We elaborate on the specific technologies and architectures we have used, and describe the implementation of the execution scenarios we have included in this prototype.

4.1 Requirements

4.1.1 General software requirements

For the flakiness inducer infrastructure, we have the following specific requirements:

1. container images need to be built on-demand for different projects with different programming languages and test frameworks. They need to include all the necessary components to execute software tests together with the required fuzzy loaders. They also need to be able to report the test results to the data management component;

2. containers for the various execution scenarios need to be defined, launched and tracked efficiently during their execution. During their execution, they must be isolated as much as possible from each other and any outside influence;

3. a flexible interface needs to be provided to allow for dynamic configuration of execution scenarios. This also allows adding more of them later without large changes to other components;

4. data about users, applications and test runs must be stored and made accessible;

5. an API needs to be provided for various frontend options to access and modify the data, as well as start and monitor test runs;

4.1.2 Isolation

As mentioned in the introduction, the isolation of the testing environment is one of the main challenges in this approach. When executing tests under different execution scenarios, the confidence in our root-cause analysis is dictated by how well we can control the parameters which we do not actively vary between them. Therefore, outside influence on test execution must be ruled out as far as possible. There are two main aspects to this:

1. the system(s) on which the test containers are executed must be isolated from the rest of the software stack as much as possible. Other software executed on these systems must be kept to a minimum, and must be kept separate from the container execution;
2. it is desirable to run multiple test containers in parallel in order to speed up test run execution. In that case, the containers must also be isolated from each other;

4.2 Model of ideal system

For an ideal implementation of the infrastructure, there are three important components:

- **Data management component**: This component deals with data storage and user interaction, as well as with configuration of the execution scenarios. Ideally, it provides a flexible API for an arbitrary amount of different frontends. Also, it should allow an administrator to configure execution scenarios and add more of them, without changes to the underlying code;
- **Isolated system**: This component symbolizes the isolated systems on which the tests are executed. Ideally, it should be easy to add more systems, they should be completely independent from both each other and the management component, and they should perfectly fulfil the isolation criteria listed under the requirements section;
- **Test container**: This component executes the tests on the software to be examined in a software container. Fuzzy loaders, as defined by Terragni et al. [2], can be run in the same containers to generate dynamic load scenarios. The test results are then reported to the management component in a way that creates as little overhead as possible, in order to avoid interference with other test containers running in parallel on the isolated system;

In figure 4.1, we have drawn up a schematic depiction of the ideal system. The execution containers are shown as colored boxes, with each color referring to an execution scenario. When a test run is started, each of the scenarios creates a specified number of containers to be run on the isolated system. These can be managed by a common container orchestration tool, such as Kubernetes, Docker Swarm, OpenShift, Nomad, etc. The containers, when finished, still have to report the results of the execution to the data management component, so steps must be taken to ensure that this interferes with test execution in the other containers as little as possible.

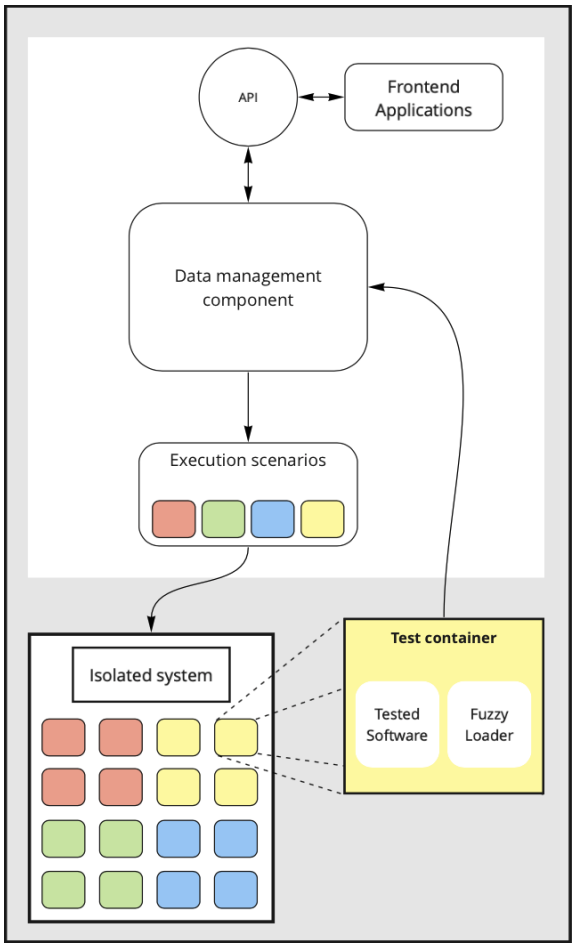

Figure 4.1: Schematic of the ideal architecture

4.3 Actual implementation

For our actual implementation, we have chosen Kubernetes as a container orchestration platform. Kubernetes is widely adopted and has excellent SDK support [15], which we require to interact directly with its API from within our tools. It also provides out-of-the-box features to restrict container access to CPU and memory resources [11]. For restricting access to other resources, plugins can be added, for example for the available bandwidth each container receives [16].

For the concrete installation, we have selected k3s [17], which is a Kubernetes distribution that can be installed as a single, light-weight binary on each of the nodes. This distribution is designed for minimal resource usage on weak systems, and should therefore have a minimal impact on our test flakiness evaluations.

4.3.1 Microservice architecture

The proposed ideal architecture should provide a lot of flexibility in the implementation - for example, it should be possible to add execution scenarios dynamically. Therefore, we have implemented the services as microservices. Microservices provide several key characteristics [18]. For our use case, modularity is particularly important. The execution scenarios can be implemented by the developers themselves or by a platform administrator (someone who provides the software platform for others) in any programming language they are familiar with. They only need to conform to a common API.

In addition to the execution scenarios, the management component can also be split up into smaller services, allowing each to deal with a specific aspect of the system, and keeping individual components simple and easy to understand.

4.3.2 Implementation overview

In figure 4.2, we provide an overview over our actual implementation. Everything runs within the same kubernetes cluster, in a single namespace.

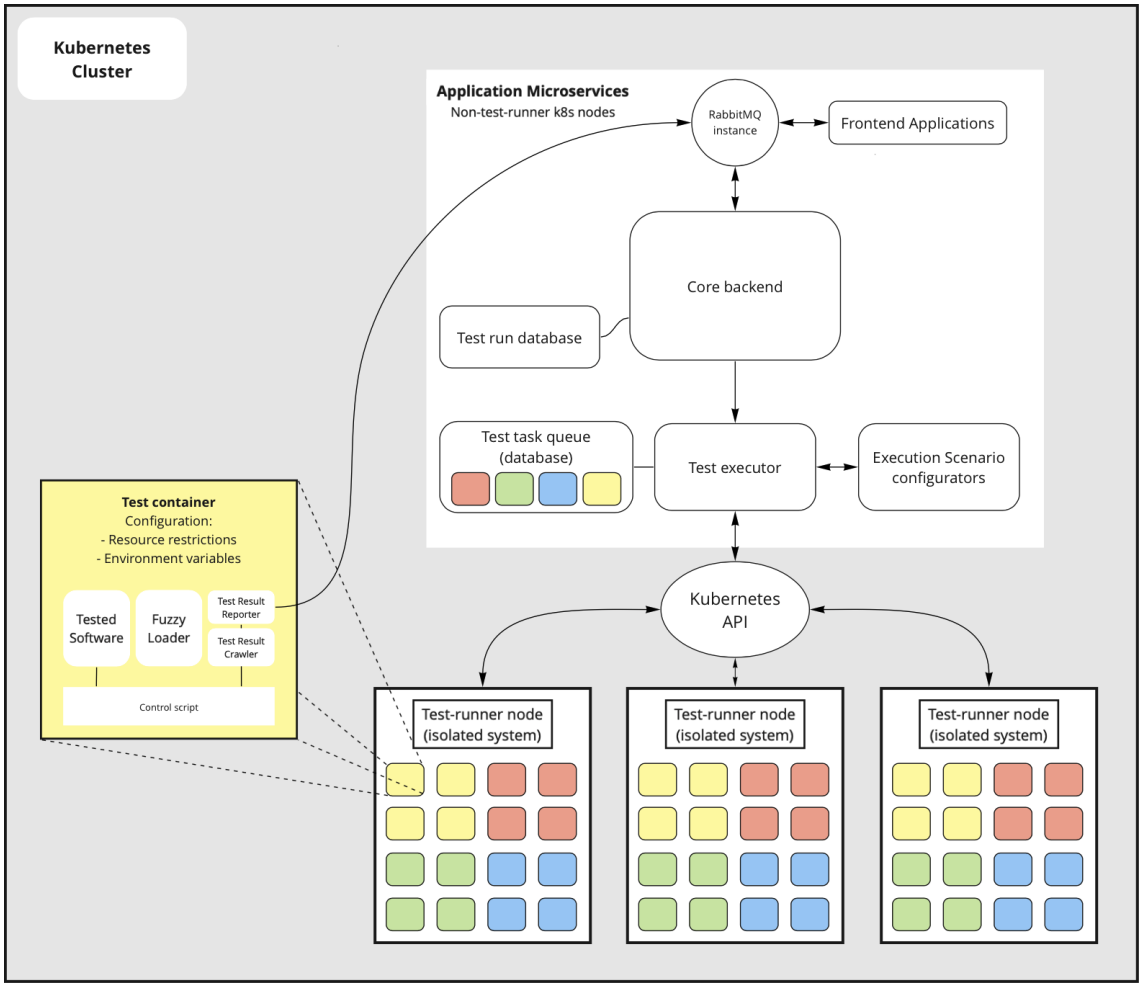

Figure 4.2: Schematic of our prototype implementation

- **Core backend**: The core backend deals with all user interactions and database storage. It fulfils the functionality of the management component defined in the ideal implementation. It also handles building of the **test containers**, using Google's Kaniko container build tool [13].
- **Test executor**: The test executor calls the execution scenario configurators to generate Kubernetes pod specifications, and executes those via the **test task queue**. The queue is setup so that not all pods are created in Kubernetes at once, reducing load on the Kubernetes API.
- **Execution scenario configurators**: These configurators provide an API for the test executor to call, and generate resource limit and environment variable distributions for the requested

amount of executions for their specific execution scenario. We have implemented an example configurator for the "concurrency" scenario proposed by Terragni et al. [2], which generates a distribution of CPU core limits around a value defined by the amount of cores available in the **test-runner nodes**.

- **Test-runner node**: The test runner node corresponds to the isolated system in the ideal architecture. It executes the test containers in an isolated environment, which we elaborate on in section 4.3.4.
- **Test container**: The test containers are implemented on a per-framework basis. In our prototype, we only support Java projects with the Gradle build agent. A Python control script manages the execution of both the tests and the fuzzy loaders. When the tests are complete, the fuzzy loaders are stopped, and the results are reported directly via the **RabbitMQ instance** to the **core backend**.
- **RabbitMQ instance**: The RabbitMQ instance provides APIs for frontend applications, as well as the test results. An arbitrary amount of different frontend applications can communicate with the **core backend**, and receive notifications on test run status from the core backend.

The core backend, test executor and existing example execution scenario configurator are implemented in Java, using the Spring Boot framework, and interact with the Kubernetes API via the Fabric8 Java client [19]. All the applications and databases within the "Application Microservices" section in figure 4.2 are run on arbitrary Kubernetes nodes within the same cluster, but not on the test-runner nodes. These applications are all designed to be stateless, and can thus be easily scaled according to the Kubernetes principles.

4.3.3 Test run flow through the application

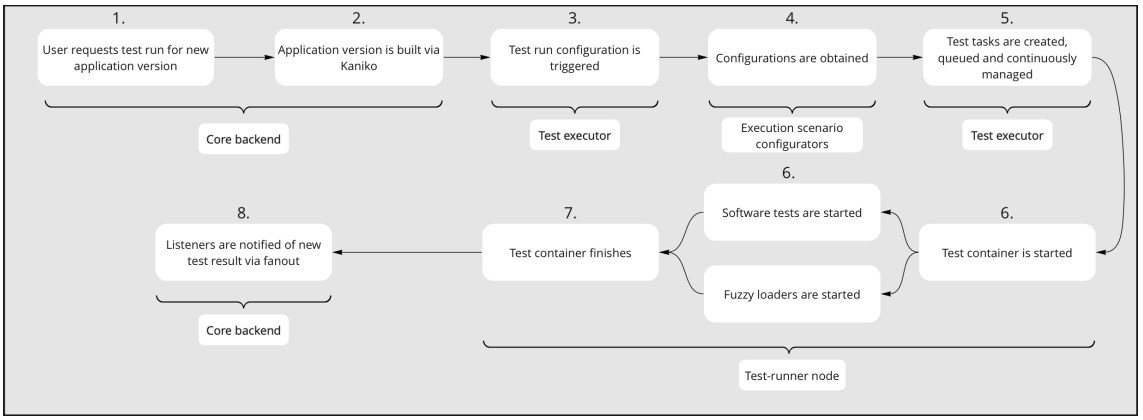

Figure 4.3: Schematic of test run flow through the application

In figure 4.3, we show what happens in order when a test run is started.

1. When a valid user request for a test run on a new application version comes in, the application source code is already expected to be stored on an NFS (network file system) server[1].

2. This directory is mounted, together with the scripts running in the test container, in a Kaniko build pod in order to build the test container. The container image is stored on a private docker registry.

3. When the build is finished, the test run definition is passed to the test executor, together with the image path on the docker registry.

4. For each execution scenario, the corresponding configurator is called to provide a distribution of resource requirements and environment variables. For each test run execution, there will be one dynamically determined set of both resource requirements and environment variables.

5. When the configurations are complete, the test tasks for each scenario are created, and a dynamically determined amount[2] of them is queued. Whenever a test task is finished, a new one is queued, in order of creation, until the queue is empty. Note that which containers are actually running at any given time is determined by Kubernetes itself, based on their resource requirements. The test executor merely provides a selection for Kubernetes to pick from.

6. In any given test run, there may be thousands of test containers that need to be executed. For each individual test container, when Kubernetes allocates it to a test-runner node, the control script starts both the fuzzy loaders[3] and the software tests. It executes each test case specification in order, or all tests, if no individual test cases were specified by the user.

[1]This is because in order to build the application in the Kaniko build container, its source code needs to be made available to that container. This can be done as file system mounts inside the Kaniko container

[2]based on the number of available CPU cores on all test-runner nodes

[3]configured via the environment variables provided by the execution scenario configurators

7. When the tests in this particular container are finished, the results [4] are collected by a "results crawler"[5], and reported back to the Core backend via RabbitMQ. Kubernetes detects that the container has finished and triggers an event, which the test executor listens to in order to queue another task.

8. The core backend sends a notification about the new test result to a fanout exchange on the RabbitMQ instance, which an arbitrary amount of frontends may be listening to. It is up to these frontends to distribute the notifications to the correct users.

9. Finally, when the test run is finished (not shown in figure 4.3), another status update is sent to the fanout, to notify the listeners that the test run is complete.

4.3.4 Isolation of the test-runner nodes and test containers

The test runner nodes need to conform to the ideal isolated system definition as well as possible. Therefore, they were implemented in our prototype as bare-metal systems, running a bare Ubuntu Server 21.04 installation with no additional services, apart from the Kubernetes agent.

Kubernetes allows for nodes to be labelled and tainted. This permits fine control over which pods get scheduled on which nodes. Therefore, the test-runner nodes can easily be configured to run only the pods that are essential to Kubernetes' inner workings. On top of that, Kubernetes provides several useful configuration options to achieve isolation of the pods from each other, as well as the other running system processes. Terragni et al. [2] have identified the number of cores, network bandwidth and disk size as critical resources that might affect overall test flakiness. Kubernetes provides some resource management out-of-the-box [11].

Static CPU assignment

In Kubernetes, the kubelet can be configured to use a static CPU assignment policy [12]. Under this policy, specific CPU cores can be reserved for Kubernetes tasks. Some Linux distributions, such as CentOS, also allow restriction of system tasks to specific CPU cores [20], but we have not implemented this feature in our prototype.

Kubernetes pods can have CPU and memory limits and requests, by default. If values for both CPU and memory are provided, and they are equal for limits and requests, the pods fall into the "guaranteed quality of service class" [12]. When using the static policy, these pods are assigned specific CPU cores for execution, provided that the CPU core request is an integer value. In theory, this should provide perfect computation isolation between the test containers. In practice, there will always be some degree of interference. CPU core speed is changed dynamically, influenced for example by current CPU power usage and thermals, which will affect test execution. Also, CPU cores are not always equally performant, so the specific cores allocated to a test container may also have an impact on the observed test flakiness. By keeping these interferences to a minimum, we can increase the confidence in our root-cause analysis.

Memory limits

The memory limits set for the pods are not implemented as hard limits. Instead, we have observed that a pod will consume physical system memory up to its limit, and then switch to system swap memory. This should replicate the same dynamic as if the tests were running on a bare metal system with total physical memory size equal to the memory limit the pod has specified. This

[4] test case name, passed/failed, and a stack trace in case of failure
[5] specific to the test framework that the software tests use

could also provide interesting avenues for further execution scenarios. We envision a scenario where containers are configured with different memory limits, which would effectively reduce memory speed past the limit due to switching to slower system swap memory. Fuzzy loaders which dynamically allocate system memory to contend with the tested software could be added to this scenario to increase the effect.

Other resources

Kubernetes also provides a plugin interface to restrict additional resources, such as network bandwidth limits [16]. For other resources, further investigation is necessary.

4.3.5 Implemented execution scenarios

As an example, we have implemented the following execution scenarios, according to the proposal by Terragni et al. [2]:

- **Base**: The base execution scenario is implemented as test containers with equal resource limits and environment variables, with no fuzzy loaders running. It provides a baseline to compare the other execution scenarios to, to allow for root-cause analysis as postulated by Terragni et al. [2].
- **CPU load**: This execution scenario adds a fuzzy loader to the base scenario, which dynamically loads the available CPU cores according to a base load percentage defined via an environment variable in the container. It changes the load at a specific interval to use different amounts of cores and load them to changing percentages.
- **Concurrency/CPU load**: This execution scenario expands on the pure CPU load variant by also changing the CPU core limit on the test containers dynamically. This corresponds directly to the "Multi-threaded execution cluster" defined by Terragni et al. [2].
- **Concurrency**: This is the same execution scenario as the concurrency/CPU load scenario, minus the CPU load. Since our architecture allows an administrator to configure each execution scenario with different fuzzy loaders, we have also designed this one to test with.

The fuzzy loaders are configured via the environment variables on the test containers. Each of them needs a name and a load percentage, around which the fuzzy loaders will load each resource. The current CPU load example is implemented as a Python script that is executed by the control script, and other loaders could be implemented the same way, with the interface that is in place to control them. However, the control script can also easily be modified to launch more processes to run in parallel, which can be implemented in any language. These scripts are all built into the test containers at build time, and will be executed only if specified via their corresponding environment variable.

Chapter 5

Evaluation

We will answer the following research questions:

- **RQ 1**: How well is our isolation concept implemented?
- **RQ 2**: How well does the implemented prototype show test flakiness, for both known test cases as well as in general?
- **RQ 3**: How do the implemented prototype execution scenarios affect test execution, i.e. do they affect overall failure rates?

5.1 Isolation between test containers

For the isolation between test containers, we are relying on the concept of cgroups in the Linux kernel. In order to validate this, we have run single tests with a benchmark script (listing 5.1) and checked their CPU assignment, as well as compared the execution of single test containers with multiple containers running in parallel.

```
def test_performance():
    results = []
    thread_start_time = time.time_ns()
    while len(results) < 2000000:
        random_characters = []
        for _ in range(random.randint(1, 60)):
            random_characters.append(random.choice(string.ascii_letters))
        random_string = ''.join(random_characters)
        results.append(random_string)
    thread_end_time = time.time_ns()
    print("Thread took", thread_end_time - thread_start_time, "ns")

if __name__ == '__main__':
    processes = []
    for i in range(8):
        processes.append(Process(target=test_performance))
    start_time = time.time_ns()
    for process in processes:
        process.start()
    for process in processes:
        process.join()
    end_time = time.time_ns()
    print("Total: took", end_time - start_time, "ns")
```

Listing 5.1: Python benchmark script

5.2 Test runs on projects with flaky tests

To answer our research questions, we have executed test runs on two projects with known flaky tests, as well as a project where we found additional flaky tests under specific circumstances:

- **Project 1**: RxJava, commit ID dde2c0e7b0435b4195d02699ef4c8f8d666480e2. In this project, flaky tests were identified by Silva et al. with the "Shaker" tool [10].
- **Project 2**: Apple Servicetalk, commit ID f9f6f76f25de59f667ae4ad36aef2d304c34f550. In this project, flaky tests were identified via GitHub issues [1]
- **Project 3**: Spring Boot, commit ID f81921c005fed703572faeb43f1a284ade2994f1. In this project, we found new flaky tests during our evaluations.

We have executed the test runs under all our prototype execution scenarios in order to determine whether they could be a suitable tool to explore the non-deterministic space and find the root cause of test flakiness. Additionally, we have varied other parameters of the execution, such as the OS on the test-runner node, and whether multiple containers are allowed to run in parallel. At this state, the goal of these evaluations is merely to validate our prototype and to show whether flakiness is affected by our proposed execution scenarios, not to determine actual root causes.

[1] https://github.com/apple/servicetalk/issues/920, https://github.com/apple/servicetalk/pull/1393

Chapter 6

Results

To answer our research questions, we used a system with the following specifications as the single test-runner node for evaluation, unless otherwise specified:

- OS: Ubuntu 21.04 with kernel version 5.11, no additional virtualization
- CPU: AMD Ryzen R9 5950X 16 Core/32 Thread processor
- Memory: 32 GB 3600 MT/s DDR4

6.1 Isolation testing results (RQ 1)

During the execution of single test containers, the following behaviour can be observed in the prototype test-runner node:

- The containers are assigned to specific CPU cores as configured via Kubernetes, as can be seen in figure 6.1
- When multiple containers use the CPU at the same time, the reserved CPU cores are not assigned: Figure 6.2. In this case, CPU core #3 was configured for system/Kubernetes usage.
- During a typical test run with only a few test cases, the overhead by the test result reporting is minimal, on the order of a few KBit/s. However, when large amounts of test cases are executed and stack traces of test failures are included, the reports can grow to quite a large size. In some cases, when many failures occurred (resulting in many long stack traces), they grew up to several hundred MB. This may, in the prototype's current state, impact other running pods, if the test cases rely on network connectivity. The impact of this needs to be reduced in order to improve test container isolation.

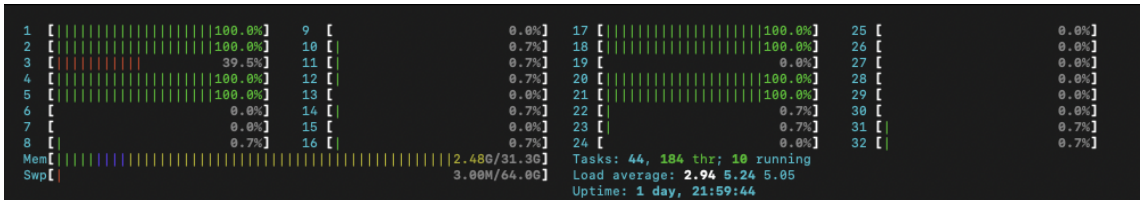

Figure 6.1: CPU & Memory usage when executing a single container

Test	Test run time [s]
1	56.8701
2	55.5201
3	55.9577
4	56.9129
5	56.4600
6	56.7677
7	57.2827
8	56.3920
9	56.3632
10	57.1106
Average	56.5637
Standard deviation	0.5390

Table 6.1: Execution times in seconds for single execution of one container, 10 runs

Test	Container 1 run time [s]	Container 2 run time [s]	Container 3 run time [s]
1	61.0826	71.1977	61.6736
2	61.9905	62.5855	61.3351
3	61.2887	72.5240	61.1602
4	61.7860	61.4694	61.6556
5	61.1651	71.7297	60.9568
6	61.4791	61.9351	61.8776
7	62.3008	67.2255	61.4669
8	61.5881	62.2681	62.3870
9	61.5575	73.7961	61.1472
10	62.0044	62.4657	62.0383
Average	61.6243	66.7197	61.5698
Standard deviation	0.3950	5.1060	0.4460

Table 6.2: Execution times in seconds for parallel execution of 3 containers, 10 runs

We have repeated the CPU core assignment test with the Python benchmark script (listing 5.1). This script generates two million random strings of random length and saves them in memory. Executing this function 8 times in parallel, it measures the time it took for each of the threads to finish, as well as the overall execution time.

Running the script in listing 5.1 with 8 available CPU threads (one per execution thread in the script), we have compared the results of a single container execution, seen in table 6.1 with the execution of three containers in parallel, seen in table 6.2. This preliminary evaluation shows the following interesting behaviour:

- When containers are run in parallel, they take longer to execute overall, probably because the CPU cores are likely not running as fast as when more of them are loaded at the same time.
- Container 1 and container 3 have very consistent execution times across test runs, comparable with single container execution.
- Container 2 usually behaves the same as container 1 and container 3, but sometimes experiences execution times increased by 10s (about 16% higher). In those cases, one or two of the threads in the script finished much faster than the others, and one or two took a lot longer.

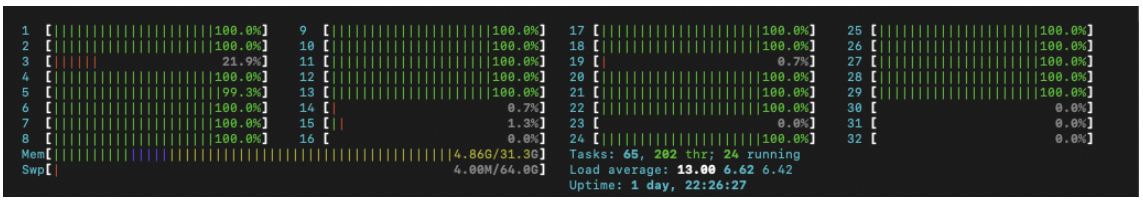

Figure 6.2: CPU & Memory usage when executing three containers concurrently

This could have many reasons. For example, the specific architecture of the processor used to run these evaluations has two core complexes of eight cores each. If the container CPU assignment done by Kubernetes is consistent, and container 2 always ended up on the same CPU cores, it could be that those were split between the two core complexes, resulting in higher latency and more cache misses. However, many other reasons are possible, for example, the specific behaviour of the pseudo-random generator in Python's random library or other implementation details of the benchmark script.

These investigations show that we have already achieved a certain degree of isolation between the test containers. With respect to CPU and memory usage, we have a clear picture of the resource isolation between containers, but we have found some issues with certain specific core assignments. Further experiments will need to be conducted in order to estimate the cause of this and the effect on test execution.

What cannot be estimated yet is the degree of isolation of our test system we have achieved. The most likely impact we can expect at the moment is the reporting of the test results, which will need further work to evaluate impact on test flakiness.

6.2 Test flakiness experiments (RQ 2 & 3)

6.2.1 RxJava

On the RxJava project, we evaluated flaky tests previously identified by Silva et al. [10]. We executed one test run with all four execution scenarios, with 400 evaluations per scenario.

Test Run - d5432d79-b49f-4d9d-8a86-1866c84c6922

Test case	scenario	Failure rate
synchronizationOfMultipleSequencesLoop	Total	12/1600 (0.75%)
	Base	3/400 (0.75%)
	CPU Load	2/400 (0.5%)
	Concurrency / CPU Load	4/400 (1%)
	Concurrency	3/400 (0.75%)
connectUnsubscribeRaceConditionLoop	Total	1/1600 (0.06%)
	Base	0/400 (0%)
	CPU Load	1/400 (0.25%)
	Concurrency / CPU Load	0/400 (0%)
	Concurrency	0/400 (0%)

Table 6.3: RxJava - Test Run results

In this case, as shown in table 6.3, our tool was able to reproduce the flakiness found by Silva et al. [10]. They found one failure each in 12 reruns, which is not enough for us to compare to. There are slight differences between the execution scenarios.

6.2.2 Servicetalk

On the Servicetalk project, we executed two test runs of interest. According to the original issue, we expected to find several flaky tests in the "io.servicetalk.http.netty.FlushStrategyOnServerTest" test package. These tests are parameterized, and so, we executed them with all parameters.

Test Run #1 - 84c9292b-3bd4-44be-a13b-23f086d0ef13 - Baseline

Test case	Scenario	Failure rate
streamingAndThenAggregatedResponse[1: strategy = DEFAULT]	Base	2/100 (2%)
aggregatedAndThenStreamingResponse[1: strategy = DEFAULT]	Base	1/100 (1%)
twoStreamingResponsesFlushOnEach[1: strategy = DEFAULT]	Base	1/100 (1%)
srvQueryResolutionResult	Base	1/100 (1%)
aggregatedAndThenStreamingResponse[2: strategy = OFFLOAD_ALL]	Base	1/100 (1%)
twoAggregatedResponsesFlushOnEnd[2: strategy = OFFLOAD_ALL]	Base	1/100 (1%)

Table 6.4: Servicetalk - Test Run #1 results

To establish a baseline, we executed all tests in the repository under the base execution scenario 100 times, for a total of 939'600 test executions. As shown in table 6.4, we found six flaky tests, including all the flaky tests identified in the original issue, however, not always with the same parameters. We also identified an additional flaky test, "srvQueryResolutionResult" in the "DnsServiceDiscovererObserverTest" class.

Test Run #2 - 18f905e2-7978-4599-8a32-768b3f8b0639 - all execution scenarios

Test case	Scenario	Failure Rate
twoStreamingResponsesFlushOnEach[2: strategy = OFFLOAD_ALL]	Total	75/12000 (0.63%)
	Base	25/3000 (0.83%)
	CPU Load	13/3000 (0.43%)
	Concurrency / CPU Load	11/3000 (0.37%)
	Concurrency	26/3000 (0.87%)
aggregatedAndThenStreamingResponse[1: strategy = DEFAULT]	Total	48/12000 (0.4%)
	Base	18/3000 (0.6%)
	CPU Load	7/3000 (0.23%)
	Concurrency / CPU Load	8/3000 (0.27%)
	Concurrency	15/3000 (0.5%)
aggregatedAndThenStreamingResponse[2: strategy = OFFLOAD_ALL]	Total	57/12000 (0.48%)
	Base	20/3000 (0.67%)
	CPU Load	11/3000 (0.37%)
	Concurrency / CPU Load	6/3000 (0.2%)
	Concurrency	20/3000 (0.67%)
streamingAndThenAggregatedResponse[2: strategy = OFFLOAD_ALL]	Total	56/12000 (0.47%)
	Base	27/3000 (0.9%)
	CPU Load	7/3000 (0.23%)
	Concurrency / CPU Load	12/3000 (0.4%)
	Concurrency	10/3000 (0.33%)

Table 6.5: Servicetalk - Test Run #2 results

For test run #2, we executed the test cases we found to be flaky under each execution scenario, 3000 times per scenario. In table 6.5, we show the results for a selection of the flaky tests. Although failure rates are very low overall, there are some promising differences between the execution scenarios in some cases.

6.2.3 Spring Boot

On the Spring Boot project, there are three interesting test runs:

1. Test-runner node on older OS (Ubuntu 20.04 with Linux kernel 5.4), parallel test execution
2. Test-runner node on newer OS (Ubuntu 21.04 as specified above), non-parallel test execution

3. Test-runner node on newer OS, parallel test execution

As can be seen, two parameters of the system were varied from test run #1 to #2. Test run #3 was executed in order to identify which of these parameters affected the test runs more. Test run #2 was executed with only one test container running at a time on the node, in order to check if parallel execution drastically changes the results, which could indicate a failure in the isolation between test containers.

Test Run #1 - e31410c1-5f9a-4437-b960-026cd09a068d - Ubuntu 20.04, parallel execution

In the results in table 6.6, we can see very high failure rates in four of the test cases. For the test case "smoketest.websocket.tomcat.echo.CustomContainerWebSocketsApplicationTests.reverseEndpoint()", there are also notable differences between the base execution scenario and the scenarios where the CPU fuzzy loader was running, with lower failure rates in the latter cases.

Test Run #2 - 3afe0405-f24b-4af1-a0b0-9dd1db9830e7 - Ubuntu 21.04, non-parallel execution

As we can see in table 6.7, no test failures occurred under the conditions specified above for test run #2 at all.

Test Run #3 - 8371be78-55d5-421b-805d-fb5285bdfe79 - Ubuntu 21.04, parallel execution

Test run #3 (results shown in figure 6.8) mostly repeats the results from run #2, apart from one test case, which exhibited flakiness with low failure rates (around 5%).

Discussion of Spring Boot results

Our test runs on the Spring Boot projects showed massive differences when varying execution parameters. Since the comparison between parallel and non-parallel execution on Ubuntu 21.04 showed minor differences, we can deduce that the isolation between test containers did not have a major effect on execution, although some impact can still be observed. This indicates that our isolation concept still needs improvement.

More interestingly, it appears that the switch from Ubuntu 20.04 to Ubuntu 21.04 caused all flaky behaviour to disappear, when failure rates were previously high (approximately 70-90%). This could have interesting implications for further execution scenarios, since one of the scenarios proposed by Terragni et al. [2] varies the OSes that tests are executed on. This scenario was not directly implemented as part of this prototype, but we envision that it could be implemented with either more bare-metal test-runner nodes, or perhaps also virtual machines, with different OSes.

6.2.4 Discussion

With regards to our research questions, our preliminary evaluations on projects with flaky tests suggest the following answers:

- **RQ 2**: How well does the implemented prototype show test flakiness, for both known test cases as well as in general?

Our prototype has shown test flakiness in projects where we evaluated previously known flaky test cases (RxJava and Servicetalk). In both cases, we could limit our test run execution to only execute these test cases, so that test run execution could be sped up when compared to executing the full suite of tests. We also successfully used the prototype to execute a test run for the entire test suite in the Servicetalk project, although with a low number of executions. In this case, most of the flaky tests with higher failure rates (as shown in the targeted executions of those cases) were also identified. In addition to that, we also found new flaky test cases in the Spring Boot project in the case where we executed the tests on an older OS version.

The prototype provides configurable options for test runs with regards to the executed test cases, the specific execution scenarios, and the total number of executions. Due to this, it has worked well both in a typical continuous integration environment, where entire projects can be evaluated for test flakiness, and more targeted execution of single test cases, which saves a lot of time and computational resources.

- **RQ 3**: How do the implemented prototype execution scenarios affect test execution, i.e., do they affect overall failure rates?

In our test runs, we have found the prototype scenarios to result in different failure rates. In particular, one of the tests evaluated in the Spring Boot project shows some difference between the base scenario and the concurrency scenarios. These preliminary evaluations are not statistically significant though, and much more stringent evaluations need to be made, with more scenarios and many more projects. Also, the concrete configuration and implementation of the individual scenarios must be further refined, in order to achieve the best possible results.

Since we do not know the root causes of the flaky tests in the projects we evaluated, and only a few execution scenarios were implemented, we also cannot judge whether the differences we observed between the scenarios are merely random, or actually indicate root causes. Therefore, we cannot answer this research question at the moment.

Test case	Scenario	Failure Rate
smoketest.websocket.tomcat. SampleWebSocketsApplicationTests.reverseEndpoint()	Total	1637/2000 (82%)
	Base	431/500 (86%)
	CPU Load	402/500 (80%)
	Concurrency / CPU Load	383/500 (77%)
	Concurrency	421/500 (84%)
smoketest.websocket.tomcat.echo. CustomContainerWebSocketsApplicationTests.reverseEndpoint()	Total	1542/2000 (77%)
	Base	419/500 (84%)
	CPU Load	365/500 (73%)
	Concurrency / CPU Load	345/500 (69%)
	Concurrency	413/500 (83%)
org.springframework.boot.actuate.hazelcast. HazelcastHealthIndicatorTests.hazelcastUp()	Total	24/2000 (1%)
	Base	3/500 (0.6%)
	CPU Load	10/500 (2%)
	Concurrency / CPU Load	6/500 (1%)
	Concurrency	5/500 (1%)
org.springframework.boot.actuate.hazelcast. HazelcastHealthIndicatorTests.hazelcastDown()	Total	0/2000 (0%)
	Base	0/500 (0%)
	CPU Load	0/500 (0%)
	Concurrency / CPU Load	0/500 (0%)
	Concurrency	0/500 (0%)
smoketest.websocket.tomcat.echo. CustomContainerWebSocketsApplicationTests.echoEndpoint()	Total	1828/2000 (91%)
	Base	465/500 (93%)
	CPU Load	460/500 (92%)
	Concurrency / CPU Load	444/500 (89%)
	Concurrency	459/500 (92%)
smoketest.websocket.tomcat. SampleWebSocketsApplicationTests.echoEndpoint()	Total	1861/2000 (93%)
	Base	466/500 (93%)
	CPU Load	463/500 (93%)
	Concurrency / CPU Load	463/500 (93%)
	Concurrency	469/500 (94%)

Table 6.6: Spring Boot - Test Run #1 results

Test case	Scenario	Failure Rate
smoketest.websocket.tomcat. SampleWebSocketsApplicationTests.reverseEndpoint()	Total	0/2000 (0%)
smoketest.websocket.tomcat.echo. CustomContainerWebSocketsApplicationTests.reverseEndpoint()	Total	0/2000 (0%)
org.springframework.boot.actuate.hazelcast. HazelcastHealthIndicatorTests.hazelcastUp()	Total	0/2000 (0%)
org.springframework.boot.actuate.hazelcast. HazelcastHealthIndicatorTests.hazelcastDown()	Total	0/2000 (0%)
smoketest.websocket.tomcat.echo. CustomContainerWebSocketsApplicationTests.echoEndpoint()	Total	0/2000 (0%)
smoketest.websocket.tomcat. SampleWebSocketsApplicationTests.echoEndpoint()	Total	0/2000 (0%)

Table 6.7: Spring Boot - Test Run #2 results

Test case	Scenario	Failure Rate
smoketest.websocket.tomcat. SampleWebSocketsApplicationTests.reverseEndpoint()	Total	0/2000 (0%)
smoketest.websocket.tomcat.echo. CustomContainerWebSocketsApplicationTests.reverseEndpoint()	Total	0/2000 (0%)
org.springframework.boot.actuate.hazelcast. HazelcastHealthIndicatorTests.hazelcastUp()	Total	93/2000 (5%)
	Base	24/500 (5%)
	CPU Load	25/500 (5%)
	Concurrency / CPU Load	27/500 (5%)
	Concurrency	17/500 (3%)
org.springframework.boot.actuate.hazelcast. HazelcastHealthIndicatorTests.hazelcastDown()	Total	0/2000 (0%)
smoketest.websocket.tomcat.echo. CustomContainerWebSocketsApplicationTests.echoEndpoint()	Total	0/2000 (0%)
smoketest.websocket.tomcat. SampleWebSocketsApplicationTests.echoEndpoint()	Total	0/2000 (0%)

Table 6.8: Spring Boot - Test Run #3 results

Chapter 7

Conclusion

In this project, we implemented a prototype for an architecture for inducing flaky test behaviour proposed by Terragni et al. [2]. Overall, we have shown that we were able to identify both previously known and new flaky tests with our prototype. Applications written in Java with the Gradle build tool can be built into our custom test containers, and test runs are automatically executed, with a simple API provided to frontend applications. At its core, the prototype system simply provides an API to build software containers and execute them repeatedly under different conditions.

In the introduction to this thesis, we mention several key challenges with regards to test system and container isolation, platform-independence for the tested software, and collection of test results. Our implementation has made use of several methods for isolation of both the isolated systems and the individual containers, although our preliminary evaluation shows that more work is to be done there to determine the impact on test flakiness that imperfect isolation could have. Thanks to its flexible architecture, our system can easily be extended to support more programming languages and testing frameworks, but we have not identified a completely platform-agnostic way to execute software tests. It is likely that such a way simply does not exist. Concerning the collection of test results, different methods to ensure minimal overhead for the running tests will have to be evaluated.

With regards to the system's effectiveness in showing root causes of flaky tests, we cannot yet make a conclusion. However, the system is highly configurable, and the prototype can be used to further evaluate this approach.

Chapter 8

Future work

For further development of the flakiness inducer infrastructure on the basis of the implemented project, there are two main categories of enhancements. On the one hand, general enhancements are required to transform this prototype into a stable platform. On the other hand, our core approached will have to be improved with further investigation into the isolation concept and the execution scenarios.

8.1 General enhancements

In the implementation of our prototype, we have adhered to general software development practices in order to provide a stable platform for enhancement. Nevertheless, the platform as a whole will still require a lot of work to eventually provide a stable solution.

8.1.1 Security

User management

Currently, users can register and login via API endpoints. Spring Boot security is used to safely store their passwords in conformance with current industry standards. However, they are currently neither authenticated nor authorized within the system, only identified via their unique ID. A service needs to be implemented that authenticates users via tokens that also provide authorization information to allow or deny access to specific resources. In the same vein, organizations could also be implemented, with several users who have different levels of access to the resources.

The authentication system should also support various platforms for continuous integration, for example via GitHub OAuth tokens.

Test container security

In our system, we execute arbitrary user-supplied software and user-defined test cases, and by default, processes in containers run with root privileges. Therefore, the containers that they run in need to be secured in a way that does not interfere with typical test cases, but also prevents malicious attacks on the infrastructure or other user's information. This should be possible, since Kubernetes provides various ways to sandbox pods, such as namespaces and security policies.

8.1.2 Administrator interface

Since we use a microservice-based approach in our prototype, and we envision further execution scenarios and fuzzy loaders to be supplied dynamically, a management system could be implemented, for administrator users to upload new versions of scenario configurators or fuzzy loaders, or register entirely new ones. This would also allow organizations to customize the infrastructure to fit their needs, on top of a standard configuration that would be supplied.

8.1.3 Test software support

Currently, the prototype only supports Java / Gradle-based projects. An enhancement to support Maven-based Java projects should be simple, but there is no general interface with which more programming languages and testing frameworks could be supported. Instead, supporting more software for flaky test evaluation will therefore likely be an ongoing task in the further development of the system.

8.1.4 Execution speed

While the ability of our prototype to show test flakiness looks promising, the test runs can take a long time and consume a lot of computational resources. Test execution will have to be optimized in order to incur a lower overhead. This can be done on a framework-specific basis, as we have done with Gradle testing, where we identify the specific Gradle subtasks which include the desired test cases in a multi-project software design. Container startup should also be sped up, since it can in some situations incur a large overhead, particularly when only few test cases are executed.

Other existing approaches to speed up flaky test re-running, such as the one used in the DEFLAKER tool put forward by Bell et al. [5], could also be included, to reduce the amount of tests that need to be examined in the first place.

8.1.5 Scalability

The architecture we have used to implement the major components of the system allows us to scale these as we wish. During further evaluation, the scalability of the other components will have to be investigated, and upgraded where necessary. For example, the NFS-based approach to the software builds is not as scalable, and there are better options (such as GlusterFS) which will provide scalability in that dimension.

8.2 Core approach enhancements

8.2.1 Test system isolation

We have already identified several methods to isolate the testing systems from the rest of the environment and the testing containers from each other. More investigation is necessary into the effects of allocation to specific CPU cores, perhaps also experimenting with different CPU models and disabling Hyperthreading, for example. To improve isolation between containers, further resource restrictions, for example network speed, will also have to be implemented. This may solve another problem, where the gathering and reporting of test results can have an impact on other tests. Other approaches to this particular challenge may include saving the results to the disks

for later gathering, or synchronizing containers to suspend result reports until all concurrently running containers have finished execution. No matter which approach is chosen, there will of course always be some overhead. The goal of future work will be to evaluate which approach, or combination of appraoches, yields the best results.

8.2.2 Execution scenarios

With regards to the core approach of different execution scenarios for root-cause analysis of flaky tests, the prototype provides a configurable platform for their implementation. In the future, more execution scenarios should be implemented, in order to better evaluate them for their usefulness in root-causing flaky tests. The prototype's interface currently supports varying resource restrictions, which are applied to the Kubernetes pods at execution time, as well as environment variables, which can be used by additional scripts (fuzzy loaders) running inside the test-container. This already allows for scenarios that explore CPU and memory restrictions, and any scenarios that can be purely implemented as fuzzy loaders. For example, a scenario to explore the impact of memory speeds could be implemented via varying memory restrictions and a fuzzy loader dynamically allocating and freeing memory, forcing the software under inspection to use slower swap memory. A fuzzy loader that writes random data to the disk could also be implemented, in order to evaluate root causes related to I/O speeds.

The prototype can also be extended to support other forms of variations. For example, pods could be restricted to run on specific test-runner nodes, which run different OSes, to explore the OS-related non-deterministic dimension.

In order to vary test run order execution, another scenario proposed by Terragni et al. [2], more investigation needs to be done into the test framework specifics. For Gradle, the build tool we support in our prototype, there are existing plugins[1] which allow control over test case order. For other frameworks, more individual solutions like that will likely be necessary.

Together with custom plugins to enhance Kubernetes' native resource limits, it should be possible to implement all of the scenarios proposed by Terragni et al. [2].

[1] https://github.com/gradle/gradle/issues/8520#issuecomment-493790187

Bibliography

[1] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on foundations of software engineering*, FSE 2014, pp. 643–653, ACM, 2014.

[2] V. Terragni, P. Salza, and F. Ferrucci, "A container-based infrastructure for fuzzy-driven root causing of flaky tests," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 69–72, 2020.

[3] N. Alshawan, A. Ciancone, M. Harman, Y. Jia, K. Mao, A. Marginean, A. Mols, H. Peleg, F. Sarro, and I. Zorin, "Some challenges for software testing research," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 1–3, 2019.

[4] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "idflakies: A framework for detecting and partially classifying flaky tests," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pp. 312–322, 2019.

[5] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 433–444, ACM, 2018.

[6] W. Zheng, G. Liu, M. Zhang, X. Chen, and W. Zhao, "Research progress of flaky tests," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 639–646, IEEE, 2021.

[7] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thumalapenta, "Root causing flaky tests in a large-scale industrial setting," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 101–111, 2019.

[8] C. Ziftci and D. Cavalcanti, "De-flake your tests : Automatically locating root causes of flaky tests in code at google," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 736–745, 2020.

[9] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–11, IEEE, 2018.

[10] D. Silva, L. Teixeira, and M. d'Amorim, "Shake it! detecting flaky tests caused by concurrency with shaker," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 301–311, IEEE, 2020.

[11] "Managing Resources for Containers | Kubernetes." Accessed: June 10, 2021. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/.

[12] "Control CPU Management Policies on the Node | Kubernetes." Accessed: June 10, 2021. [Online]. Available: https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/.

[13] "GoogleContainerTools/kaniko." Accessed: June 10, 2021. [Online]. Available: https://github.com/GoogleContainerTools/kaniko.

[14] N. Meisenzahl, "Enhance your Docker image build pipeline with Kaniko," 2018. Accessed: June 12, 2021. [Online]. Available: https://medium.com/01001101/enhance-your-docker-image-build-pipeline-with-kaniko-567afb6cf97c.

[15] "Client Libraries | Kubernetes." Accessed: June 10, 2021. [Online]. Available: https://kubernetes.io/docs/reference/using-api/client-libraries/.

[16] "CNI Bandwidth Plugin." Accessed: June 10, 2021. [Online]. Available: https://www.cni.dev/plugins/current/meta/bandwidth/.

[17] "K3s: Lightweight Kubernetes." Accessed: June 10, 2021. [Online]. Available: https://k3s.io.

[18] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.

[19] "Fabric8 Java Kubernetes client." Accessed: June 10, 2021. [Online]. Available: https://github.com/fabric8io/kubernetes-client.

[20] "Reserve Compute Resources for System Daemons." Accessed: June 10, 2021. [Online]. Available: https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/.