

Bachelor Thesis

June 14, 2021

Flaky Tests Detection in a Continuous Integration Pipeline

Implementation of a Proof-of-Concept System

Dylan Puser

of Domat/Ems, Switzerland (14-924-054)

supervised by

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



University of
Zurich^{UZH}



Bachelor Thesis

Flaky Tests Detection in a Continuous Integration Pipeline

Implementation of a Proof-of-Concept System

Dylan Puser



University of
Zurich^{UZH}



Bachelor Thesis

Author: Dylan Puser, dylan.puser@uzh.ch

Project period: 14.12.2020 - 14.06.2021

Software Evolution & Architecture Lab

Department of Informatics, University of Zurich

Acknowledgements

First and foremost, I would like to thank Prof. Dr. Harald C. Gall and Dr. Pasquale Salza of the software evolution & architecture lab (s.e.a.l.) for allowing and supervising this thesis.

I would further like to thank Dr. Pasquale Salza and Dr. Valerio Terragni for their invaluable support and inputs throughout the entire process of this thesis. They made it possible and were always available for questions and feedback.

Lastly, I would like to thank Fabio Greter for the great collaboration we had during the writing of our theses.

Abstract

Tests in software engineering are used to control the validity of code, to make sure newly written or modified code does not have unintended consequences and to create a more maintainable project. Sometimes however, a test can be flaky. A flaky test will fail occasionally, even though neither the test nor the code under test were modified. Such tests erode the trust in the tests, are difficult and costly to identify and rectify, and can have a considerable negative impact on companies and developers. Furthermore, they can be an indication of a deeper fault in the system itself. Based on a proposal of identifying the root cause of flaky tests using a container-based fuzzy-driven approach and an implementation of such a system, we discuss how to best make it available to a typical user. We then present an implementation of such a system and evaluate shortly its value.

Zusammenfassung

Tests in Softwareentwicklung werden verwendet um die Validität von Code zu kontrollieren, um neu geschriebenen Code oder kürzlich modifizierten Code auf unbeabsichtigte Konsequenzen zu prüfen und um ein besser wartbares System zu schaffen. Manchmal können diese Tests aber "flaky" sein. Ein Test der flaky ist kann sporadisch und ohne dass der Test oder der Code der getestet wird modifiziert wurden fehlschlagen. Solche Tests können die Zuversicht in sie zerstören, sind schwierig zu finden und beheben, und können einen beachtlichen negativen Effekt auf Firmen und Entwickler haben. Zudem können sie eine Indikation eines tiefgründigeren Problems im Code sein. Basierend auf einem Vorschlag um den unterliegenden Grund eines flaky Tests mittels einer Container-basierten "fuzzy"-getriebenen Infrastruktur zu finden und basierend auf einer solchen Implementierung, besprechen wir, wie wir ein solches System einem typischen Nutzer zur Verfügung stellen können. Danach präsentieren wir eine solche Implementation und evaluieren kurz ihren Wert.

Contents

1	Introduction	1
2	Background	3
2.1	Flaky Tests	3
2.2	Continuous Integration	3
3	Related Work	5
3.1	Finding Flaky Tests	5
3.2	Identifying the Root Cause of a Flaky Test	6
4	Approach	7
4.1	Requirements	7
4.1.1	CI Pipeline	7
4.1.2	Frontend Application	8
4.2	Implementation	9
4.2.1	CI Pipeline	9
4.2.2	Frontend	14
5	Evaluation	17
6	Conclusion	21

List of Figures

4.1	A mockup of the test result page. The doughnut charts break down the results per execution scenario and below is the breakdown of each test case.	8
4.2	An overview of the abstract architecture of the system. The main components are the frontend (blue), CI pipeline (red) and the interface to <i>flakiness inducer infrastructure</i> (green).	9
4.3	Relationship between components of the pipeline-part of the system. In red are components provided by our system.	10
4.4	Sequence diagram of creating a test run from a GitHub workflow using the GitHub action. Note: the interaction between the pipeline-service and <i>flakiness inducer infrastructure</i> before creating the test run is intentionally largely omitted here.	13
4.5	The test run overview page. It shows a list of test runs, as well as some further details.	15
4.6	The test results page. It shows a breakdown of each execution scenario as well as a breakdown of each test case.	16
5.1	The breakdown of one of the flaky tests identified in the Servicetalk repository. . .	18
5.2	A breakdown of the execution scenarios of one of the spring boot test runs. . . .	18

List of Tables

5.1	A list of tests from the Servicetalk repository that were identified by other developers and successfully reproduced by our system.	17
-----	---	----

List of Listings

4.1	Exemplary .flaky.yaml configuration file content. This configuration runs the tests FlowableMergeTest and FlowableRefCountTest 50 times each on every execution scenario listed at the end of the file.	11
4.2	An example of a GitHub workflow definition. This workflow runs daily, on any push or pull request to the develop branch, and can be triggered manually from GitHub. It includes our action with the line <i>uses: flaky-infrastructure/flaky-infrastructure-action/@v1</i> and passes in some inputs with the <i>with:</i> keyword.	13

Introduction

Flaky tests are tests that, without any change to the code being tested or to the test code itself, can fail intermittently. In 2017, Google reported that roughly 1.5% of their tests were flaky once a week [1]. That may not sound like a lot, but considering they had around 4.2 million tests at the time, that amounts to roughly 63'000 flaky tests [1]. Google further reported that 84% of their test transitions from a passing to a failing state were due to flakiness, rather than due to a breakage [2]. Flaky tests have been found to be a considerable burden on developers. Lam, Godefroid, Nath, Santhiar, and Thummalapenta [3] conducted a survey on 58 developers working for Microsoft and found flaky tests to be the second-most burdensome problem while using continuous integration (CI) pipelines. A survey by Eck, Castelluccio, Palomba and Bacchelli [4], conducted on 121 professional and academic developers, found that 90% of them encountered the problem a couple of times a year and 58% of them every month. Furthermore, 79% of the developers considered flaky tests to be a moderate to serious problem [4].

Such flaky tests can have a variety of adverse effects on developers and companies. They can erode the confidence of developers in the test, leading to them possibly disregarding the test results [4]. They can also require developers to rerun tests multiple times, to see whether a test result can be trusted. Google reported in 2017 that they used 2-16% of their compute resources to rerun flaky tests [1]. Beyond "just" being a nuisance, flaky tests can be an indication of problematic underlying issues in the code and can not always be ignored [5].

However, there are not a lot of tools to easily and accurately detect flaky tests. There have been some proposals both for identifying flaky tests and for finding the root cause of a flaky test, some of which will be discussed in later chapters. This thesis builds upon one such proposal by Terragni, Pasquale, and Ferrucci, who propose a fuzzy-driven approach to finding the root cause using a container-based infrastructure [6]. It tries to identify the root cause of a flaky test by running it repeatedly in isolated containers with varying resources and fuzziness, depending on what potential root cause is being explored.

Such an infrastructure is being built by Fabio Greter, and this thesis was developed in conjunction with it. Thus it assumes such an infrastructure to exist. The goal of the thesis is to design and implement a system that would make such an infrastructure available to an end-user. Users should be able to use it in CI pipelines, as well as have access to a user interface (UI) for easy monitoring and evaluation. This implementation should serve as a proof-of-concept that shows the validity and value of such a tool for detecting flaky tests and their root cause in an easy-to-use and unobtrusive fashion.

The thesis is structured as follows. First, a few background concepts relating to this thesis will be explained in Chapter 2. Then, in Chapter 3 some academic work related to this topic will be presented. Next, the approach will be detailed in Chapter 4, first by discussing the requirements

we had for our system in Section 4.1 before presenting the resulting implementation including all the components in Section 4.2. In the proceeding chapter, Chapter 5, we will discuss a short evaluation of our system, followed by the conclusion and the future work in Chapter 6.

Background

2.1 Flaky Tests

A flaky test is a test with a non-deterministic outcome. That is, it can fail intermittently without any change in its code or the code being tested [5]. Such tests come with a variety of problems. They erode the trust a developer might have in a test and can lead to the test being entirely ignored [4]. Flaky tests are also difficult to reproduce and fix, requiring considerable time and resources that could better be used elsewhere [2,4].

As such, considerable effort has been made to try and understand what the root cause of a flaky test might be. We will henceforth refer to this as the *root cause analysis*. Zolfaghari, Parizi, Srivastava, and Hailemariam in a literature review list a summary of identified possible root causes. This includes tests where two or more tasks are run in parallel (concurrency), the order in which tests are run, and input/output related problems, to name a few [7].

2.2 Continuous Integration

Continuous integration is the concept of automating the build and testing process whenever a change is made to a code repository [8]. Developers often keep a large set of unit tests, tests that usually only test a singular feature, function or procedure. This test set then runs every time a developer pushes changes committed to the repository. In doing so, they verify that the new changes do not introduce any unintended faults and that everything still functions. This is often referred to as regression testing.

Lam et al. have shown that it is an important problem with regard to flaky tests for surveyed Microsoft employees [3]. As such, it is an important use case to consider when developing a system to identify flaky tests.

Related Work

The research field of flaky tests is relatively young, though the amount of research has been increasing in the past years [7]. As such, several methods of identifying and mitigating the effects of flaky tests have been published. In the following sections some of the proposed methods for finding flaky tests as well as some proposals on identifying the root cause of a flaky test will be discussed. Lastly, the proposed method by Terragni et al. [6] upon which this thesis builds will be discussed in more detail.

3.1 Finding Flaky Tests

Arguably the simplest method of finding flaky tests is to rerun a test multiple times. If a specific test does not pass (or fail) on every run, it is flaky. This method is often already supported by a lot of test runners and frameworks. Still, it is usually a time and resource consuming activity, and it can be difficult to gain valuable information from it.

A somewhat related concept offered by many frameworks and tools is that of rerunning only failed tests. If a test does not pass on the first run it gets executed again until it either passes or reaches the retry limit. In the former case the test is flaky, in the latter it is most likely not flaky, but entirely broken and can not be ignored. However, a test might be flaky and yet still pass in the first run. Such false negatives are not picked up by this method and the goal here is less to identify flaky tests and more to safely ignore them. Some example tools/frameworks that implement this are Cypress [9] and Maven Surefire [10].

Shaker is a method proposed by Silva, Teixeira, and d'Amorim, that introduces noise to the execution environment through stressor tasks [11]. The stressors e.g. compete for central processing unit (CPU) resources, to manifest flakiness more prominently. This system focuses primarily on concurrency as a root cause [11], and as such is quite limited.

Another method of detecting flaky tests is through code coverage. For example, *DeFlaker* by Bell et al. [12] analyses which tests cover a new code change and mark any test that fails without covering those changes as flaky. An advantage of this differential code coverage approach is that it has a low run-time overhead, as tests do not have to be executed multiple times [12]. One obvious limitation of such a system is that tests which cover newly written or recently modified code cannot be checked for flakiness. Also entirely newly written tests can not be tested and according to an empirical study by Luo, Hariri, Eloussi, and Marinov [5] 78% of flaky tests are flaky when they are first written, so this limitation encompasses a rather large use case.

Further research efforts, which for the sake of brevity will not be elaborated upon further, attempt to use machine learning to predict flaky tests [13] or try to establish a causal link between test smells and flakiness [14].

3.2 Identifying the Root Cause of a Flaky Test

Regarding finding not only flaky tests, but their root cause as well, there is considerably less research. Lam et al. [3] propose a system that first identifies flaky tests by the method described above, where a failing test gets retried up to a certain limit and is deemed flaky if it passes. To identify the root cause, the binaries and dependency binaries of these flaky tests are collected and instrumented. The instrumented tests are then run locally many times in an attempt to collect informative logs for both passing and failing runs of a test. The logs are then further processed line-by-line, where at each line a set of predicates are evaluated. In a last step, the predicates are examined to identify ones where they are opposite for passing versus failing tests, e.g. if all runs of a test that pass exhibit a certain negative predicate, while for the failing test runs said predicate is positive, that predicate can give an indication of what the root cause might be. Instrumentation can however interfere with the test execution, Lam et al. [3] themselves observed some tests that were only flaky with instrumentation and some that were only flaky without instrumentation. Thus, a higher degree of isolation is needed.

A Container-Based Infrastructure for Fuzzy-Driven Root Causing of Flaky Tests. Terragni et al. [6] propose to find the root cause by fuzzing the execution environment in a container-based infrastructure. Thereby a test suite is run multiple times under different execution clusters (hereafter renamed to as *execution scenarios* for clarity). Each scenario represents a possible root cause, such as concurrency, and contains multiple containers, with each optionally having a fuzzy loader. The containers can be configured with different resources and the fuzzy loaders can occupy and free resources with dummy operations. Thus, each container and accompanying fuzzy loader can help explore the test runs under different conditions of its execution scenario. [6]

In addition to the different execution scenarios one wants to test for, a baseline scenario is run as well. It serves as a control and represents the tests being run under normal conditions. After all execution scenarios have finished, the root cause analysis can be conducted as follows: A test is flaky in an execution scenario if it both fails and passes at least once. If this test is not flaky in the baseline scenario, but is flaky in one or more execution scenarios, the execution scenario with the highest fail-rate for that test is most likely the root cause for the flakiness. If however, the baseline scenario is flaky as well, the execution cluster which deviates most in its flakiness compared to the baseline scenario (both positively or negatively) is most likely the root cause. [6]

Approach

In the following sections, an overview of the requirements of the infrastructure being built will be presented. Then, the actual implementation will be shown and discussed.

4.1 Requirements

The requirements for the system being built were, given a hypothetical container-based fuzzy-driven infrastructure for finding the root cause of flaky tests (hereafter referred to simply as *flakiness inducer infrastructure*) as described by Terragni et al. [6]:

1. to define an interface for other applications and services to make use of the infrastructure;
2. to make the infrastructure available to a CI system;
3. to create a frontend application for users to easily consume and gather insights from the information of the infrastructure;

In the following subsections, the requirements for 2. and 3. will be detailed further.

4.1.1 CI Pipeline

Regarding CI and DevOps in general, we wanted to first and foremost have the system be configurable by the user to make it as open and flexible as possible. A user should be able to configure three things:

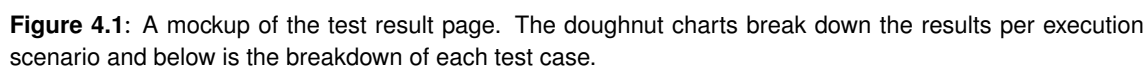
1. DevOps related options: Configuring when to run *flakiness inducer infrastructure*, e.g. on a certain version control system (VCS) branch push or on a schedule;
2. options concerning the integration of *flakiness inducer infrastructure* into a CI pipeline, e.g. whether or not to abort a pipeline as soon as a flaky test is discovered;
3. options regarding *flakiness inducer infrastructure* itself, e.g. the amount of times a test is run or which execution scenarios (possible root causes) they are run in;

With that in mind, following are some exemplary use cases that the system should cover.

Use in a CD Pipeline. Often, tests are run before deploying a new code version, e.g. before releasing a new build to the public. In such circumstances tests may act as the last barrier to catch any faults in the code before a release. This represents another use case for the system of this thesis. If added to such a CD pipeline, it could prevent deployments that include flakiness.

The frontend application should serve as the primary interface for users to interact with *flakiness inducer infrastructure*. Its main use case should be to provide the user with informative and digestible information from *flakiness inducer infrastructure* test runs. Further, it should allow for users to manage all their applications, test runs and more. Following are some more detailed requirements.

Test Run Results. The main requirement. The application should take the result data collected by *flakiness inducer infrastructure* and present it to the user in a way that is easily understandable and informative. If possible, the application should deliver real-time updates as the *flakiness inducer infrastructure* test run is being conducted. Furthermore, the application should provide the user with a breakdown of the results by execution scenario, as well as with a breakdown of each individual test case. Once the test run is finished, it should further provide the user with the root cause analysis for each test case. The final mockup of this part can be seen in Figure 4.1



Application and Test Run Overview. The application should provide an overview of the users' applications (repositories) and test runs, and give them options to interact with and manage them. Possible interactions include deleting, creating new applications, starting new test runs etc.

4.2 Implementation

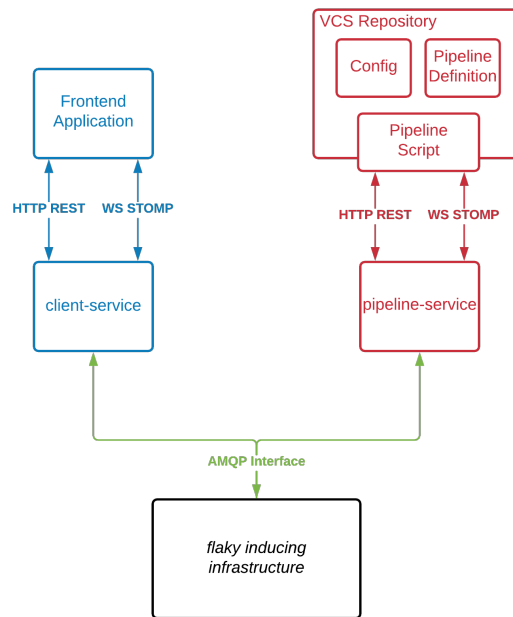


Figure 4.2: An overview of the abstract architecture of the system. The main components are the frontend (blue), CI pipeline (red) and the interface to *flakiness inducer infrastructure* (green).

Figure 4.2 shows an overview of the abstract architecture. A micro-service architecture was chosen for its loose coupling and high flexibility. The architecture includes the AMQP interface that handles communication between *flakiness inducer infrastructure* and the system this thesis implements, a micro-service for client/frontend applications, the actual frontend application, a micro-service for CI pipelines, as well as components to facilitate the usage of this system from CI systems. Every major component was created using clean architecture [15] to enhance the maintainability and longevity of the system.

In the following subsections each component will be further elaborated upon, as well as the concrete implementation shown.

4.2.1 CI Pipeline

The concept of CI is tightly coupled with that of VCS systems, such as Git and Mercurial. CI systems offer the ability to perform jobs, such as building an app, running tests, to be performed when e.g. changes are pushed to the development branch, when a new version is released or when a user in an open-source project proposes changes. For this project, we chose to support

Git, as it controls the vast majority of the market share [16, 17].

Furthermore, most developers use distributed / hosted version control systems, such as GitHub, GitLab and BitBucket, which serve as remote repositories but include a plethora of other features as well. One such feature that all three offer are their own DevOps systems, GitHub Actions, GitLab CI/CD and BitBucket Pipelines respectively. Each allows users to configure their own CI/CD pipelines. It was decided to use these systems for this thesis' CI system, as it a) allows for users to use a system they are already familiar with and b) can be used in conjunction with other services and systems (e.g. using *flakiness inducer infrastructure* followed by a deploy to Amazon Web Services (AWS)).

GitHub and BitBucket additionally offer ways of encapsulating and sharing functionality, called GitHub Actions¹ and BitBucket Pipes respectively. These can be used to facilitate the integration of a service into a CI/CD pipeline. The proof-of-concept version of the system, the version this thesis implements, supports GitHub by providing such a GitHub action. This action handles the proper communication with the pipeline-service. The pipeline-service is however designed to be agnostic to which provider is used, meaning one can easily create and provide a BitBucket pipe, or generally a script for any similar system, to use *flakiness inducer infrastructure*.

Thus to use *flakiness inducer infrastructure* all a user needs to do is create a GitHub workflow file,

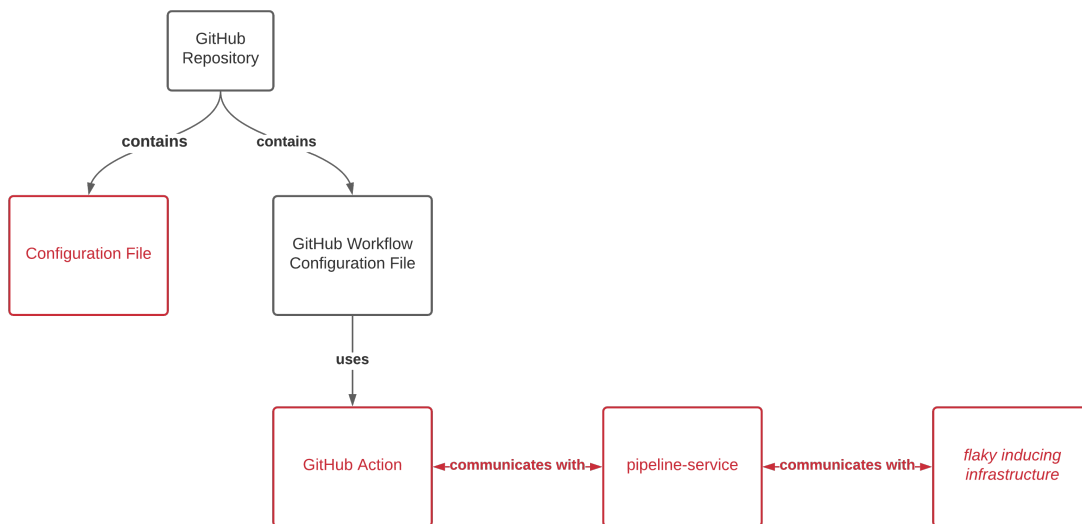


Figure 4.3: Relationship between components of the pipeline-part of the system. In red are components provided by our system.

which specifies when the pipeline is run, and to include the provided action which additionally allows for some inputs to configure itself. Lastly, a configuration file is used to configure some *flakiness inducer infrastructure* specific properties. The action then handles the communication between the repository and the pipeline-service, which in turn acts as a gateway to *flakiness inducer infrastructure*. Figure 4.3 shows this part of the system in more detail. In the following subsections, each of these components will be described more in depth.

¹GitHub Actions can refer to both to their DevOps system, as well as to the method of encapsulating and sharing functionality via scripts being discussed here. To avoid confusion, hereafter action generally refers to the script and the term workflow shall be used to denote a DevOps pipeline.

Configuration File and Configuration Schema

As stated in 4.1.1, one requirement (3.) was that users could configure options regarding the running of *flakiness inducer infrastructure*. This configuration should be independent of how a *flakiness inducer infrastructure* test run is initiated, be it from a pipeline, from the frontend application or other. It also includes some configurations necessary for our system. In short, the options a user **must** configure are:

- the main language of the repository under test²;
- the framework that is used to run the tests³;
- the number of times each test case should be run under each execution scenario;
- a build-timeout in seconds;
- a timeout in seconds for the test run;

Furthermore, a user **can** specify:

- a white-list of execution scenarios to run the tests in, by default all scenarios are used;
- a white-list of test class names to run, by default all test cases are run;
- additional build dependencies needed

language: `java`

framework: `gradle`

buildDependencies:

- `openjdk11`

executionConfiguration:

numberOfRuns: `50`

buildTimeoutInSeconds: `1800`

testRunTimeoutInSeconds: `7200`

testConfiguration:

runOnly:

testSpecification: [

`'io.reactivex.rxjava3.flowable.FlowableMergeTest',`

`'io.reactivex.rxjava3.flowable.FlowableRefCountTest'`

]

executionScenarioConfiguration:

runOnly: [`'base', 'cpu-load', 'concurrency', 'concurrency-cpu-load'`]

Listing 4.1: Exemplary `.flaky.yaml` configuration file content. This configuration runs the tests `FlowableMergeTest` and `FlowableRefCountTest` 50 times each on every execution scenario listed at the end of the file.

²In this first version of the system only Java is supported.

³Here too currently only one framework is supported, Gradle.

These options are configured in a file written in **YAML Ain't Markup Language**⁴ (YAML), a serialisation language. When using *flakiness inducer infrastructure* from a CI pipeline, this file is to be named *.flaky.yaml* and to be placed in the project root, though both of these can be customised. Listing 4.1 shows an example configuration.

To facilitate users implementing the configuration file, a schema was developed. Many integrated development environments (IDEs) support the association of certain files with a schema, after which users receive auto-complete suggestions and warnings when e.g. a required field is not set. Eventually, we would like to host the schema on a language server, so users can include it at the top of the file. The schema itself is also written in YAML, and enforces the rules and options described above.

Pipeline Script - A GitHub Action

We provide a GitHub action written in JavaScript that handles the work needed to be done during a GitHub DevOps workflow (pipeline). It checks for and prepares all the necessary files, extracts the relevant GitHub user details and handles the communication with the hypertext transport protocol (HTTP) RESTful (REST) application programming interface (API) as well as the web-socket (WS) simple text oriented messaging protocol (STOMP) API. More precisely, it:

- checks for user credentials⁵;
- checks for the existence of the configuration file;
- gzips and tars the repository;
- extracts information from GitHub, such as the username, commit id and more;
- makes a multipart/form-data POST request to the pipeline-service including the repository tar, the configuration file and the user data;
- optionally subscribes to the test run status over a WS;

GitHub actions naturally allow for customisation through inputs, and this allows us to configure some properties of how the action is to be consumed (requirement 2. of Section 4.1.1). The inputs allow for users to use custom filenames and file-paths for their configuration file, as well as determine the behaviour of the action after the test run is created. By default, once the test run has been created, the action exits successfully, allowing the GitHub workflow to finish or continue with the next steps. A user can however set the action to subscribe to the test run results and only exit once it has finished. Furthermore the user can specify if the action should fail if the test run fails or results in flaky tests, or if the action should exit normally regardless. The idea here being that in a critical workflow, e.g. using *flakiness inducer infrastructure* as part of a release CD pipeline, a user might want to ensure that no flaky tests are detected and abort otherwise, while in many other cases it might suffice to simply initiate the test run and return. Figure 4.4 shows the sequence of events when creating a test run from a workflow. An example GitHub workflow definition using our action can be seen in Listing 4.2.

⁴YAML's acronym is recursive, hence it appears in its own definition.

⁵This is mocked in the proof-of-concept version.

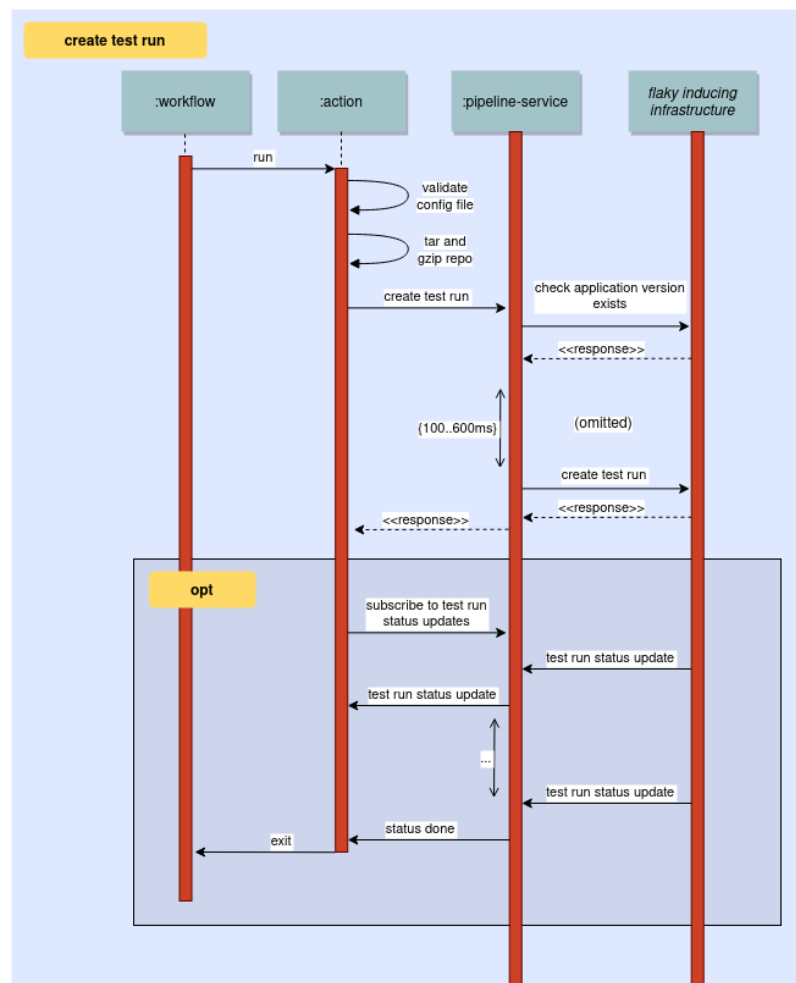


Figure 4.4: Sequence diagram of creating a test run from a GitHub workflow using the GitHub action. Note: the interaction between the pipeline-service and *flakiness inducer infrastructure* before creating the test run is intentionally largely omitted here.

name: flaky infrastructure CI

on:

push:

branches:

- develop

pull_request:

branches:

- develop

schedule:

- cron: '* * * 1 * *'

workflow_dispatch: # manual dispatch

jobs:

```

run-tests:
  name: Run flaky-infrastructure tests
  runs-on: ubuntu-latest
  steps:
    - name: run tests
      uses: flaky-infrastructure/flaky-infrastructure-action/@v1
      with:
        credentials: {{ secrets.FLAKY_INFRASTRUCTURE_TOKEN }}
        configFile: flaky-infrastructure.config.yaml
        waitForFinish: false

```

Listing 4.2: An example of a GitHub workflow definition. This workflow runs daily, on any push or pull request to the develop branch, and can be triggered manually from GitHub. It includes our action with the line `uses: flaky-infrastructure/flaky-infrastructure-action/@v1` and passes in some inputs with the `with:` keyword.

Pipeline-Service

The pipeline-service handles the initiation of test runs via CI/CD pipelines. It is a micro-service that uses three transport protocols to communicate with other components. The communication with *flakiness inducer infrastructure* is done using RabbitMQ, which implements the advanced message queuing protocol (AMQP). It sends requests to and receives responses from *flakiness inducer infrastructure* over dedicated exchanges for different resources, e.g. applications, users, test-runs etc. Furthermore, it listens to a queue that sends test run status updates. Test runs are initiated from clients with a HTTP POST request. Clients may subscribe to test run status updates using a WS STOMP subscription, after which they receive WS STOMP messages including the status updates. The service was written using Spring Boot, a Java framework to build applications and services.

4.2.2 Frontend

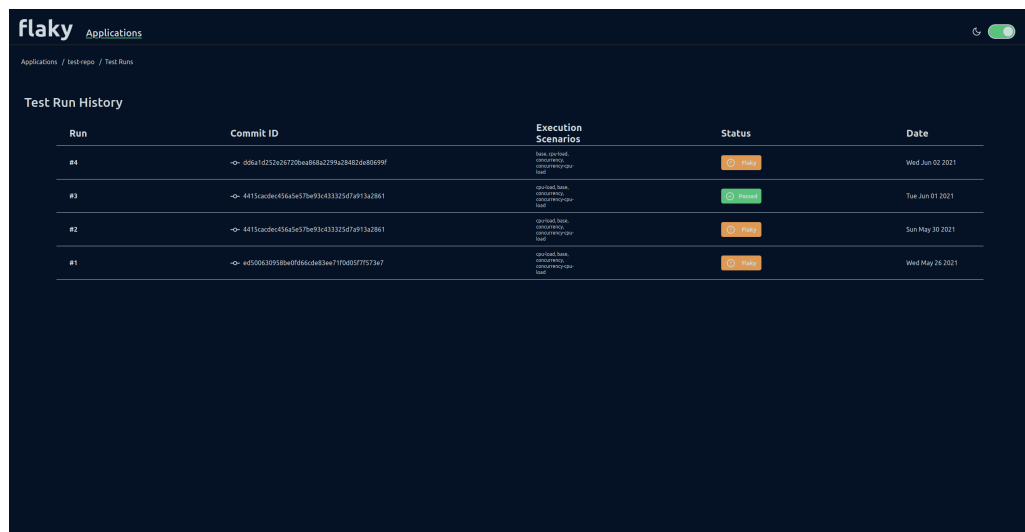
As previously mentioned, the frontend should serve as the main interaction point for users using *flakiness inducer infrastructure*. Its main goal should be to present users with the information gained from *flakiness inducer infrastructure* in an informative and intuitive manner. It consists of a micro-service, which serves as a gateway to *flakiness inducer infrastructure*, and a web-application. Both components will be detailed more in the following subsections.

Frontend Application

The frontend application was built as a web application written using TypeScript and Svelte, and is comprised of 3 parts. The application overview page currently serves as the landing page. On it, users see a list of all of their applications, which is an abstraction that generally means a Git repository. It also shows the amount of versions the application has, which again is an abstraction that usually means a Git commit.

The test run overview page (Figure 4.5) shows a list of the test runs for an application, newest first. It includes details such as which execution scenarios are being tested for and the Git commit id. The status badge in colour shows the status of the test run, i.e. if a test run is still running, if it failed, if it detected flaky tests and so on. It updates in real-time over a WS connection.

The main part of the application is the test results page, which can be seen in Figure 4.6. It



Run	Commit ID	Execution Scenarios	Status	Date
#4	55a1d123a26720ba868a2299a284e2da89699f	test case 1 test case 2 test case 3	Pass	Wed Jun 02 2021
#3	4415cadec45645657be93a43325d7a913a2861	test case 1 test case 2 test case 3	Fail	Tue Jun 01 2021
#2	4415cadec45645657be93a43325d7a913a2861	test case 1 test case 2 test case 3	Fail	Sun May 30 2021
#1	ed500630958ba0f66dcad83ae719f0d5f77573a7	test case 1 test case 2 test case 3	Fail	Wed May 26 2021

Figure 4.5: The test run overview page. It shows a list of test runs, as well as some further details.

is updated in real-time using WS, so a user does not have to refresh the page when monitoring a running test run. At the top there is a loading bar, that shows the progress of the test run. Below it, is the breakdown of the execution scenarios. Each doughnut chart shows one execution scenario, depending on which ones were selected. It shows how many test cases are passing, flaky and failing per execution scenario. Passing and failing in this case means the test case passes respectively fails every time in this execution scenario. It can provide the user with an indication of which execution scenario is causing the most flakiness.

Next, there is a breakdown of each test case. For each, the total fail rate, i.e. the fail rate over all execution scenarios combined, is shown. A fail rate of 0 means the test has passed every time and thus is not flaky. A fail rate of 1 means the test fails every time and is broken. A fail rate between 0 and 1 then means the test case is flaky. The row can additionally be expanded, at which point it shows the fail rate for each execution scenario. Once the test run is done, the root cause analysis for each flaky test is shown.

Client-Service

The client-service is built analogously to the pipeline-service. It is a micro-service and it acts as a gateway to the frontend application, as well as to other possible clients. It contains a REST and STOMP API, and communicates with *flakiness inducer infrastructure* over the RabbitMQ interface. The REST API is used for most of the communication between the frontend application and the client-service. Using it, the application can get the applications, test runs and test run results from *flakiness inducer infrastructure*. For the real-time monitoring, two STOMP subscription endpoints are used. The application can subscribe to the status update of a test run, as well as to results from newly run test cases. The client-service too, was written using Spring Boot and Java.

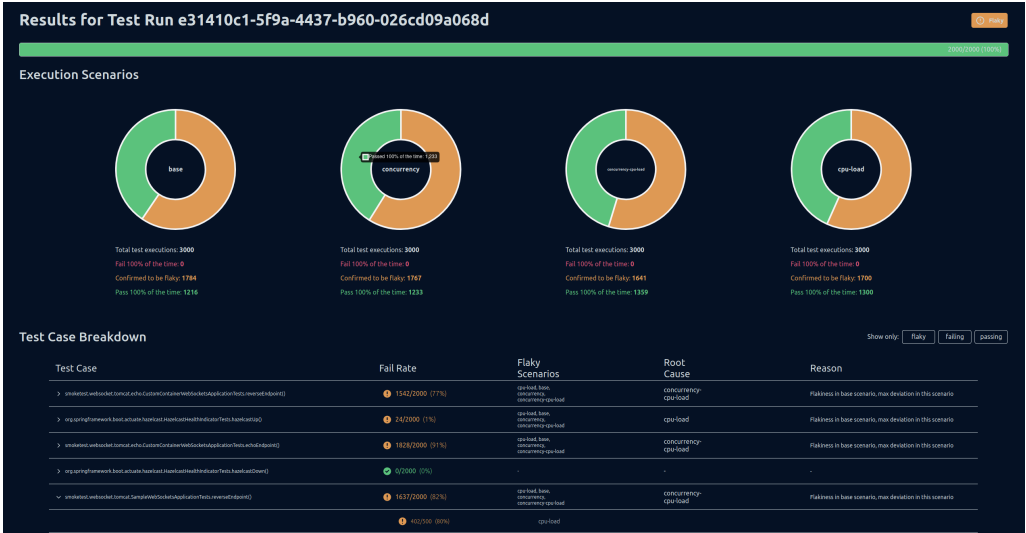


Figure 4.6: The test results page. It shows a breakdown of each execution scenario as well as a breakdown of each test case.

Evaluation

We tested our system on several popular open-source Java Gradle repositories. We wanted to test two things, a) that we could use the system to find flaky tests in existing projects and b) that we were able to reproduce flaky tests identified by other developers. For the latter we searched through issues and pull requests of these repositories looking for ones mentioning and discussing flaky tests. When promising ones were identified, the repository was forked and a commit near the date of the issue was checked out. Then, our system was used on that repository. Below are some of our findings.

Servicetalk. Servicetalk¹ is a network application framework developed by Apple. Several suitable flaky test candidates were identified² by developers. Our system was then used on these candidate tests.

The configuration was set to run every test 12'000 times on four execution scenarios, the baseline, concurrency, concurrency-cpu-load and cpu-load scenarios. The baseline scenario includes no fuzziness or limitations on resources and acts as the control. The concurrency scenario varies the amount of CPU cores available, the cpu-load scenario generates CPU load in the containers and the concurrency-cpu-load scenario is a combination of these two.

We were indeed able to reproduce the flakiness of these candidate tests, as can be seen in Table 5.1. Furthermore, we were able to identify four more flaky tests, not originally identified by the de-

Repository	Test case	Measured Fail Rate	Root Cause Analysis
servicetalk	FlushStrategyOnServerTest .twoStreamingResponsesFlushOnEach[2:strategy=OFFLOAD_ALL]	75/12000 (0.63%)	concurrency-cpu-load
servicetalk	FlushStrategyOnServerTest .twoStreamingResponsesFlushOnEach[1:strategy=DEFAULT]	46/12000 (0.38%)	concurrency-cpu-load
servicetalk	FlushStrategyOnServerTest .aggregatedAndThenStreamingResponse[2:strategy=OFFLOAD_ALL]	46/12000 (0.38%)	concurrency-cpu-load

Table 5.1: A list of tests from the Servicetalk repository that were identified by other developers and successfully reproduced by our system.

¹<https://github.com/apple/servicetalk/>

²<https://github.com/apple/servicetalk/issues/920>

developers. Figure 5.1 shows the test case breakdown of one of these tests. The top row shows a summary of the test case along all execution scenarios. More precisely it shows the summarised fail rate, which execution scenarios were run as well as the root cause analysis. In this case it is the `cpu-load` scenario because the baseline scenario was flaky and the `cpu-load` scenario deviated the most in its flakiness (in this case zero) from it. In the sub-rows, the fail rate for each individual execution scenario is shown. We can see that the test was flaky in three of the four scenarios.

io.servicetalk.http.netty.FlushStrategyOnServerTest.streamingAndThenAggregatedResponse[1; strategy = DEFAULT]	7/2000 (0.35%)	base, concurrency, concurrency-cpu-load	cpu-load	Flakiness in base scenario, max deviation in this scenario
	0/500 (0%)	cpu-load		
	3/500 (0.6%)	base		
	2/500 (0.4%)	concurrency		
	2/500 (0.4%)	concurrency-cpu-load		

Figure 5.1: The breakdown of one of the flaky tests identified in the Servicer talk repository.

RxJava. Another repository we used was RxJava³, a library that implements observables in Java. Two candidate flaky tests were identified⁴ by *Shaker* [11], a related tool we discussed briefly in Section 3.1. Here we were able to confirm the flakiness of one of the two candidate tests. When run 1600 times on the same four execution scenarios as before, `FlowableMergeTest.synchronizationOfMultipleSequencesLoop` exhibited a fail rate of 12/1600 (0.75%). The root cause was deemed to be `concurrency-cpu-load`.



Figure 5.2: A breakdown of the execution scenarios of one of the spring boot test runs.

Spring Boot. Our third repository used was Spring Boot⁵, a framework for building applications and services in Java. We found a discussion⁶ regarding flaky tests, but were ultimately not

³<https://github.com/ReactiveX/RxJava>

⁴<https://github.com/ReactiveX/RxJava/issues/7136>

⁵<https://github.com/spring-projects/spring-boot>

⁶<https://github.com/spring-projects/spring-boot/issues/25410>

able to reproduce any of them. This was however not entirely unexpected, as it seemed as the flakiness could likely be due to networking and timeouts, two execution scenarios not yet implemented in *flakiness inducer infrastructure*. Nevertheless, we were able to identify six other flaky tests.

Figure 5.2 shows the execution scenario breakdown of one of these test runs. With it we can see how many test executions are flaky, passing or failing in any given execution scenario. In this example, the baseline and cpu-load scenarios are roughly equal, while concurrency and concurrency-cpu-load are lower. Since the baseline scenario is flaky, this suggests the concurrency and concurrency-cpu-load scenarios as possible root causes, which is indeed the result the root cause analysis arrived at for four out of the 5 flaky tests.

Conclusion

We set out to show that we could create a system to identify flaky tests and their root cause. Furthermore, we wanted to provide this system in a way that would be intuitive, easy-to-use and useful to developers. As such, we provide a way for users to easily use this system as part of a CI pipeline. Additionally, we wanted to provide feedback and results from the system in real-time and in a way that makes the information easy to understand for a user. Thus, we provide a web application for this purpose.

With our proof-of-concept system we were successfully able to show the value of such a tool. By adding just a configuration file, users can easily integrate the system into their repositories. Using just one line, they can include the system in their CI pipelines. Then, they can view the status and the results from the test run in real-time. Users are presented with an indication of which execution scenarios seem to be causing most of the flakiness and they are shown a breakdown of each test case, how flaky it is and what is most likely the root cause.

The system we built acts as a proof-of-concept and as such a lot can be added in the future. We built the system to be flexible and open to extensions, and as such it can be used as a foundation to build upon. In particular, here are some things we envision could be implemented in the future. One of the most pressing additions would be that of authentication and authorisation. Currently, most of this is mocked or omitted. In conjunction to this, one could also add more user-related functionality, such as registering a user from the frontend.

Currently, test runs can only be initiated from pipelines. Eventually, it would enhance the usability of the system if it would allow users to initiate test runs and to configure their CI pipelines from the frontend as well. This would require to allow users to connect their account using GitHub (and eventually GitLab and BitBucket) OAuth. Then, using things such as GitHub hooks, one could react to e.g. branch pushes and initiate test runs.

As BitBucket and GitLab are widely used alternatives to GitHub, they should eventually be supported as well by creating a BitBucket pipe and possibly a general script that can also be used by GitLab CI/CD.

As mentioned previously, the system currently only supports Java and Gradle, of course eventually it should support all major languages and frameworks.

Bibliography

- [1] J. Listfield, "Where do our flaky tests come from?," 2017.
- [2] J. Micco, "The state of continuous integration testing @google," 2017.
- [3] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on software testing and analysis*, ISSTA 2019, pp. 101–111, ACM, 2019.
- [4] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "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*, ESEC/FSE 2019, pp. 830–840, ACM, 2019.
- [5] 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.
- [6] 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, ACM, 2020.
- [7] B. Zolfaghari, R. M. Parizi, G. Srivastava, and Y. Hailemariam, "Root causing, detecting, and fixing flaky tests: State of the art and future roadmap," *Software, practice & experience*, vol. 51, no. 5, pp. 851–867, 2021.
- [8] M. Azure, "What is continuous integration?," 2021.
- [9] Cypress, "Test retries," 2021.
- [10] A. M. Project, "Rerun failing tests," 2021.
- [11] 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, 2020.
- [12] 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.
- [13] T. M. King, D. Santiago, J. Phillips, and P. J. Clarke, "Towards a bayesian network model for predicting flaky automated tests," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 100–107, 2018.

- [14] F. Palomba and A. Zaidman, "Retraction note: Retraction note to: The smell of fear: on the relation between test smells and flaky tests," *Empirical software engineering : an international journal*, vol. 25, no. 4, pp. 3041–3041, 2020.
- [15] D. Deutsch, "A quick introduction to clean architecture," 2018.
- [16] Openhub, "Compare repositories," 2019.
- [17] C. Gehman, "Git vs. mercurial: How are they different?," 2019.