Capturing test flaking

"How do we manage flaking tests to reduce cost and preserve the mental well-being of the developers?"

Thesis - Sanne Visser

Student	Sanne Visser
Date of Internship	7 February – 5 June 2022
Education	Computer Science – Software Engineering University of Applied Sciences Leiden
Graduation organization	Velory
Version	3.0.0 – Final – 14 June 22

Preface

Two years ago my partner asked me how cool it would be to live abroad for a period. His best friend had completed two semesters in Boston not long before that and he was really excited about his experience.

Hearing him talk about the different culture, all the friends he made and the experiences he had ignited a little flame inside me. What if I could do that?

Truth be told, I was shy and unsure back then. The thought alone was already very thrilling. Except this didn't keep me from exploring the idea. As I would do with anything that intrigues me, I started reading about it.

Reading all the positive experiences and encouragement of all the students that have lived abroad, I was ready to take the next step. I started looking into my options.

Feeling overwhelmed by the information, the corona pandemic and an operation I had to go under, I took a step back. I finished my third-year internship and went into the second half of my third year.

My partner brought it up again and let me know that if this is something I wanted I should go for it and act on it now. This is the only moment I have left to experience living abroad without all the commitments of a full-time job.

After his encouragement I reached out to the teacher that was involved with internationalization. Having conversations with her solidified the idea of doing my graduation internship, here, in Sweden.

Sanne Visser

June 1, 2022 - Gothenburg

Acknowledgment

I'd like to express my deepest gratitude to my partner, Lars. Without his support and encouragement throughout this period I never would've gone onto to this journey and I never would've stayed sane without our weekly video breakfasts.

I would also like to thank Bart, the best friend of Lars. Without him this idea would have never occurred to me and I wouldn't have had this amazing experience.

I'm extremely grateful for my family. Thank you so much for being a support system, for being there for me and for expressing interest in what I was doing.

I must also thank Carla, the teacher that is involved in internationalization. Thank you for helping me find a great company and checking in with me during my internship.

I would also like to thank Michiel, my education supervisor but also my study career counsellor. Thank you so much for not only being there during my internship but also before that.

I must also thank Thomas, my fellow student who was an intern at Velory before me. Thank you for putting in a good word for me and thank you for the all the fun conversations we've had and feedback we've exchanged.

This project would not have been possible without Jack, my company supervisor. Without him I wouldn't have had this fun project to work on, along side amazing colleagues. Thank you for helping me through this period and sticking by me even when you left Velory.

I'd also like to express my gratitude to CJ and Mickan, two of my colleagues that worked with me in the Gothenburg office. Thank you for welcoming me with open arms and for having fun lunches together.

I'd also like to acknowledge the help of Bruno, Adam and Martin, my colleagues with whom I sparred and talked about my research. Thank you for being the fun, kind and smart colleagues that you guys are.

Finally, I would like to thank Velory and the University of Applied Sciences Leiden for this opportunity. I have learned a lot in this process and I'm very grateful.

Contents

1 Summary

The question we are going to answer is “How do we manage flaking tests to reduce cost and preserve the mental well-being of the developers?”.

A flaking test is a test that can both succeed and fail without changing relevant code. There is little to no awareness about this and some developers know of the existence but don't realise it is flaking instead of failing.

These tests are detected by analysing the test result from both local and online test runs. With the use of the Jest Watch plugin we make sure we only consider the test results of tests that are testing unchanged code. The data is then being send to the backend where the other analytics are done.

The problem of invisible flaking tests is solved by a tool that highlights the frequency and the severity. It visualises the problem. This project is tightly coupled to the JavaScript testing framework used by Velory: Jest.

By visualizing the flaking tests in different types of dashboards we not only provide useful information for the developer via the ‘run’ and ‘test’ dashboard. We also provide useful information for a tech lead or CTO by providing a project overview, signalling any recurring problems or technical debt the team should focus on next.

The project dashboard gives insight into the severity and the overall frequency of the issue. The other two dashboards give more specific insight to help solve the flaking tests. There is also a dashboard dedicated to all the flaking tests to help decide what test should be solved first.

The project is open source, so it is free to use and other developers can add to the product. The backend is written in NestJS, the frontend is written in ReactJS and useful data is captured with a plugin build for the Jest testing framework. The stack is based on JavaScript since it’s the most used language according to StackOverflow. This way we hope to help as many people possible and find many contributors for our open-source project.

Future modifications to the resulting project include monetization. Since it’s a open source project there is currently no funding available. There are plans to use the Freemium model where users can either host the software themselves or use our (yet-to-be-build) premium hosting service optimized for our tool.

2 Introduction

Before you, you find the thesis Capturing Test Flaking written by me, Sanne Visser. This thesis has been completed within and with guidance by the company Velory in Sweden.

Within this thesis you will read about the phenomenon of test flaking and about the tool that has been made to capture this.

In short, a test can be marked as flaking when a test can both fail or succeed without relevant changes in the code. Velory had a lot of flaking tests that disrupted the workflow.

Before I came to Velory, it took them great time and effort to fix some of the flaking test they had identified. In the end, two big fixes were implemented: fixing known issues with the mocking server and intercepting error sensitive network requests.

Even with these issues resolved, a lot of flaking tests were still left. The biggest issues were not knowing which test to mark as flaking and how to resolve this.

The initial solution came from the company supervisor but, as will become clear throughout the thesis, has been changed to suit the needs of the company better.

The subject of this thesis is a big and relatively untouched subject. This means that it was already clear from the beginning that there was a possibility that the project wouldn't be completed in its entirety.

All the figures that will be used within this thesis can be found at readable scale in the appendix. Any research I've done beside the main thesis subject are referenced as a source in this document and can be found as external attachments.

All the research that has been done can be found as an external attachment (Visser, Research, 2022). The functional and technical design can also be found as an external attachment (Visser, Functional and technical design, n.d.).

3 The Graduation Organization

3.1 The Organization

Velory is an Employee Experience Platform that provides companies and their employees with Productivity-as-a-Service (PaaS), and offers a cushy process when purchasing hardware, software and services. You could say Velory acts as a bridge between companies and suppliers.

Velory also supports companies in becoming more sustainable in their operations. They do this by managing the lifecycle of all the hardware, software and services and deliver an overview of all the IT expenditures. To make the process even more green they work with the One Device - One Tree solution. For every device that gets replaced they plant a tree.

Velory consists of different departments to make the best product and experience they can:

- Brand & Comms
- Operations & Analytics
- People & Culture
- Product
- Finance & Legal
- Sales & Partners

I was part of the product department, as was Jack-Edward Oliver who was my company supervisor.

The product team, as the name suggest, is the development department that is responsible for creating and maintaining the product.

The product consists of two platforms, Velory and Resellers. The platform for companies and their employees and the platform for the "resellers" to manage all the available products.

Companies can manage all the assets they have and view which employee has what device. An employee can pick the devices they want based on a budget per category or when they exceed the budget get a deduction in salary.

3.1.1 What did the graduate do?

Coming into Velory I started working within the so called “portal” team, which is the team working on the migration of the Ruby on Rails apps to React apps. When I came in there was already a lot of progress with this migration.

The first weeks were spent on getting to know the team, the code and the way of working. I worked on migrating components and functions to shared libraries and creating primitive components.

During these weeks I sparred a lot with colleagues about different approaches and I've done multiple pair-programming sessions with both a senior developer and another frontend intern.

About a month into the internship, the portal team started with weekly sprints instead of having sprints every other week. On Monday the different tasks are distributed and on Friday everyone makes sure everything was finished and merged. We also had a retrospective to check in with everyone. What were the good things and bad things?

Beside this weekly retrospective I also had meetings every other week with my company supervisor and sparring partner Jack.

My focus was on my thesis, but I've done a few work items on the side because it was fun to work on and it helped with staying connected to the team.

3.2 Stakeholders

The stakeholders for this thesis stretch beyond Velory. The stakeholders are split into two categories: direct and indirect stakeholders.

3.2.1 Direct stakeholders

The direct stakeholders are the stakeholders who have a direct relationship with the project. They are mostly the people in Velory and the people who benefit directly form the results of this research.

These stakeholders are Jack, the direct colleagues within the portal team, the colleagues within the other teams, the CEO and the CTO.

Jack has been the most involved, both as a supervisor and a developer. Besides being a stakeholder he's been responsible for quality control and proof reading, biweekly strategy and progress meetings and helping with product design.

The colleagues within the portal team are directly benefiting from the research and the product. They also pitch ideas and act as a "rubber duck" meaning you can talk to them about the problems you face and they help you get through them.

The CEO and CTO of Velory are mainly interested the project benefits for the company, but they are not directly involved in the process. However, they can give input and directions to the project where they see fit.

Being the Universities graduation supervisor, Michiel Boere has also been a direct stakeholder. He is both interested and has influence over the scope and direction of the project. His main interests lay with keeping the project realistic, manageable and help bringing the graduation to a successful conclusion.

3.2.2 Indirect stakeholders

Indirect stakeholders are stakeholders that have no direct involvement with the project but must be taken into account when deciding on project requirements. For this project they are mainly “developers around the world”. These developers can be a part of a company or work on their own projects. If Jest is used in the product, they are someone we need to take into account.

Since the tool that is made is open source, all developers can benefit from it and thus save time, energy and headaches.

Below you can see each stakeholder in a diagram. Each category has their own colour.

Velory	●
Education	◍
Indirect stakeholders	○

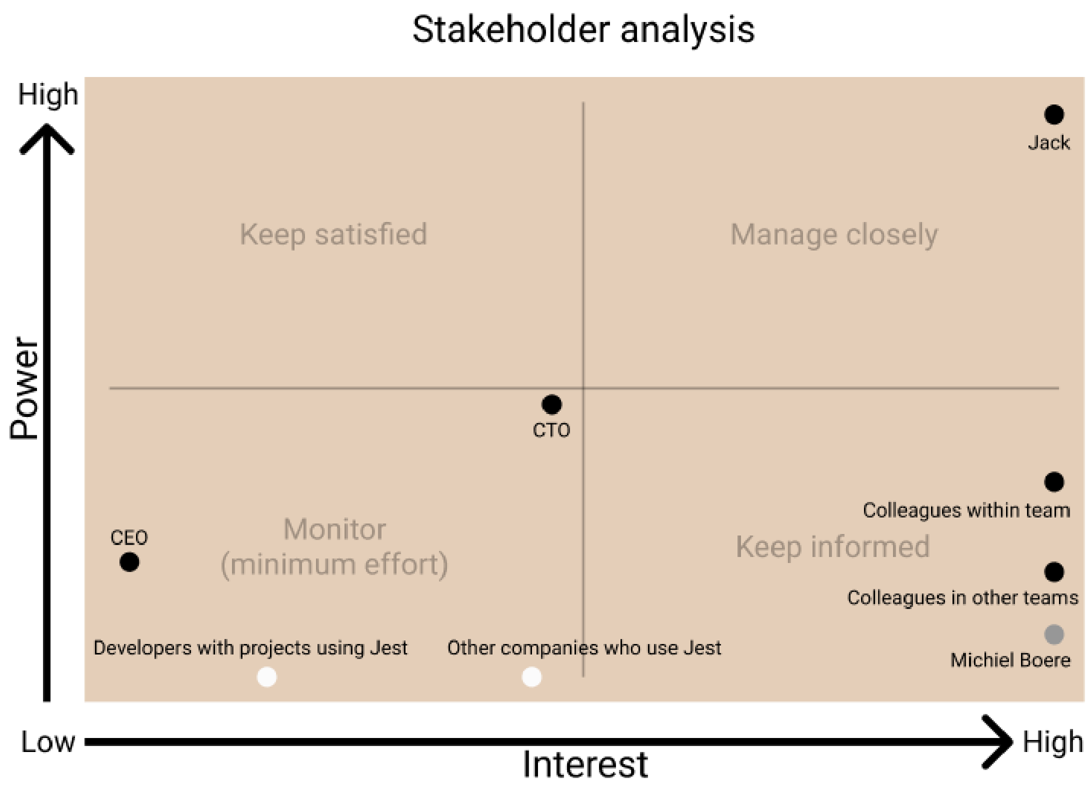


Figure 1: Stakeholder analysis

4 The Problem and Opportunity

Within this thesis we don't just solve a problem, but we also makes use of an opportunity that hasn't been taken yet.

4.1 Problem

Many products and projects have automated tests. These tests usually succeed until you change the code in a way so that the result is no longer the same. In some cases, a test can succeed and fail without relevant code being changed. This phenomenon is called test flaking.

First and foremost, this issues impacts the business side of a software project. For example, when a build pipeline fails because of a flaking test you would have to start it over and over again until the tests succeeds. This costs money for the resources used to run the tests and time that is spend until the pipelines succeeds and the software can be deployed.

Secondly, the issue impacts the developers. A developer often doesn't know if a test is flaking or if the code they wrote changed something that effects the test. It's often assumed that the tests fail for valid reasons after which the developer needs to investigate which changes caused the failure. This takes a lot of unnecessary time and energy impacting the confidence in the code and test suites.

Tests without confidence result in delayed deployments and longer development cycles. When a developer can't trust in the test suite they'll often take more time developing and testing to make sure their changes aren't breaking. This kind of defeats the purpose of a testing suite all together.

This problem has been seen up close at Velory, resulting in two full weeks of wasted time because a flaking test was failing without valid reason.

Due to the flaking tests the pipeline had to be run until all tests ran successful. When a pipeline finally succeeded, the changes would be merged into the master branch. Since tests are also required to run for the master branch we would start all over again with flaking tests failing over and over again.

These issues would often be brought up during retrospectives. All the developers had flaking tests problems but there wasn't enough time to actually fix the flaking tests. Deleting the tests wasn't an option as well because it tested some core functionality.

4.2 Opportunity

Since test flaking is a pretty unknown area there are no tools for JavaScript products yet. Creating an open-source tool for the world to use is a big opportunity to add useful knowledge and tools to test JavaScript software.

The tool developed during this research will serve the purpose of MVP (Minimal Viable Product). A fully fledged tool is out of scope since development is not the only thing done. Research into test flaking and how to best identify flaking tests is the main goal, the tool developed will demonstrate the results of this research.

Open-source means the tool is not only free to use, but other developers are free to submit changes where they see fit. In the long run the hope is that this tool will be a publicly developed and maintained tool.

Little research has been done in regards to test flaking. Besides resulting in a useful tool with practical applications, the research done can help others address flaking tests in their environment or it can serve as a base for future research or solutions.

5 The Intended Result

Test flaking is a little known phenomenon for developers. With this research and the accompanying tool we hope to create more awareness and help people recognize flaking tests.

In the ideal situation, after reading this research, developers should be more aware of the causes of failing tests and reduce the unnecessary amount of time they spend in rerunning their test pipelines. The tool developed as result of the research will help identify when a test might be flaking.

By creating a tool to discover the flaking tests and visualising this, it can help streamline the company better.

The focus of this tool can be summarized in 3 words

1. Visibility
2. Frequency
3. Severity

The biggest goal is to gain insight in what tests can flake, and when they usually do this. This could be done by analysing the current test data and finding correlations between different flaking reasons.

The second goal is to learn how often a test flakes, the frequency. Does it flake every other run? Or is it once every 10 runs. This could give insight into what flaking tests have priority to be fixed.

The third goal is severity, this means what is the impact of flaking tests? This could be translated to money and or time. Usually, every test run on a CI (Continuous Integration) pipeline costs money. In the case of Velory they run their pipelines with CircleCI, which means they pay for every second of processing time.

6 Plan of action

Given the scope of the project, a tight set of deadlines were required.

The plan of action was set up quite specific. The process was already determined and the deliverables were already set. The plan can be found as external attachment (Visser, Capturing Test Flaking - Graduation Plan, 2022).

In short summary, the planning was set up as followed:

Training: Working with the Frontend team	Week 1 & 2
Research: Causes of test flaking	Week 3
Research: How does Jest run its tests?	Week 4
Research: How to test each flaking cause efficiently?	Week 5, 6 & 7
Research: What programming languages to use?	Week 8
Developing: The server that processes all the data	Week 9, 10 & 11
Developing: The Jest plugin to process the data via the server	Week 12 & 13
Developing: The logging component to retrace the flaking	Week 14 & 15
Developing: The server saves the flaking	Week 16

However, during the research into the testing framework Jest it became clear that a broader picture was missing. The research up to that point was already focused on practical hands-on problems closely related to the situation of Velory.

The research focused too much on one given solution without taking other opportunities or problems into consideration. After some much needed brainstorming and reflection the focus was readjusted to research more general questions like “What exactly is test flaking”, “What are practical ways to identify test flaking” and “How do other companies solve similar problems”.

In the end, Jest was still taken as part of the solution for the test flaking experienced by Velory. But starting with a specific framework is never a good start for research into a deep and complicated subject like test flaking.

After reflecting on the direction of the research and desired solution the focus and main question became: “How do we manage flaking tests to reduce cost and preserve the mental well-being of the developers?”.

7 What are flaking tests?

The definition of a flaking test is a software test that can both pass and fail without changing relevant parts of the code. Having a test that fails to produce a consistent result means it is unreliable, you cannot predict the outcome (Palmer, Test Flakiness – Methods for identifying and dealing with flaky tests , 2019) (Palmer, Assert(js) 2019: Jason Palmer (Spotify) - Test flakiness; methods for dealing with flaky tests , 2019) (Avidon, 2019).

Flaking tests can occur in every kind of language and in all types of tests. The reason behind this is that code can be very dynamic.

The type of test in which test flaking is the most common is integration testing. With integration testing the individual software modules are combined and tested as a whole (Integration testing , 2021). Within a frontend application this could be rendering a page or the act of clicking through a page.

Common causes of test flaking are often related, but not limited, to:

- The (computer) environment in which a test is run
- The order in which tests are run
- The order of asserts
- The use of state
- The use of mocks
- Being too early with an assert.

7.1 What is Jest and why is it important?

Jest is the testing framework that Velory uses. This means that all the results that are gathered from the test suite are in the format that Jest came up with. It can be used in any JavaScript project and it has been created to make and run tests as easy as possible.

Jest has many ways of outputting data. This is often done by utilizing different test reporters depending on the desired file type and output format. However, Jest itself supports only two ways to output data out of the box. It can both output directly to the console (stdout) or to a text file with the JSON format. The easiest way to analyse the data is the JSON format, since it's both human-readable and easy to script.

The required data was extracted from Jest by utilizing an extension called TestScheduler. TestScheduler is a feature provided by a core library used by jest named aptly 'jest-core'. How this was done can be read in more detail in the research 'How does Jest work?' (Visser, How does Jest work?, 2022).

7.2What type of tests are we going to focus on?

Test flaking exists in all kinds of tests. With the limited time available it's important to only research the most common cases of test flaking. To achieve this, all tests needed to be analysed to determine which (kinds of) tests are flaking most often.

Given the thousands of tests Velory has, manually analysing them is out of the question. To this extend a script has been created to automatically run all tests for N amount of times and collect the results. This script was executed on a single computer instead of in the default pipelines to account for any possible instabilities. The script can be found in Appendix A: Get Test Results (Visser, Get Test Results, n.d.).

Now there were two ways to aggregate the data. By hand, which would be time consuming and prone to errors. Or the other option, that would be to create a script. This was an easy decision.

7.2.1 The script

To aggregate the data you need to have an idea what you want to aggregate it to. It starts off with looking at what data you have.

When retrieving the data from Jest, you get the test data either in JSON format or text format. The interface, provided by Jest, of the JSON format looks like Figure 2. The interfaces of some of the fields, like testResults, assertionResults and snapshot are found in Figure 3, Figure 4 and Figure 5.

```
interface TestResult {
  numFailedTestSuites: number,
  numFailedTests: number,
  numPassedTestSuites: number,
  numPassedTests: number,
  numPendingTestSuites: number,
  numPendingTests: number,
  numRuntimeErrorTestSuites: number,
  numTodoTests: number,
  numTotalTestSuites: number,
  numTotalTests: number,
  openHandles: any[],
  snapshot: SnapShot,
  startTime: number,
  success: boolean,
  testResults: SingleTestResult[],
  wasInterrupted: boolean,
}
```

Figure 2: Interface of test result from Jest

```
interface SingleTestResult {
  assertionResults: AssertionResult[],
  endTime: number,
  message: string,
  name: string,
  startTime: number,
  status: string,
  summary: string,
}
```

Figure 3: Interface of SingleTestResult

Having these interfaces makes it easier to spot some useful data or correlations.

Some basic calculations that can be made based on this information are how many and the percentage of tests failed, the average of tests failing per test run and per failing test the failure message.

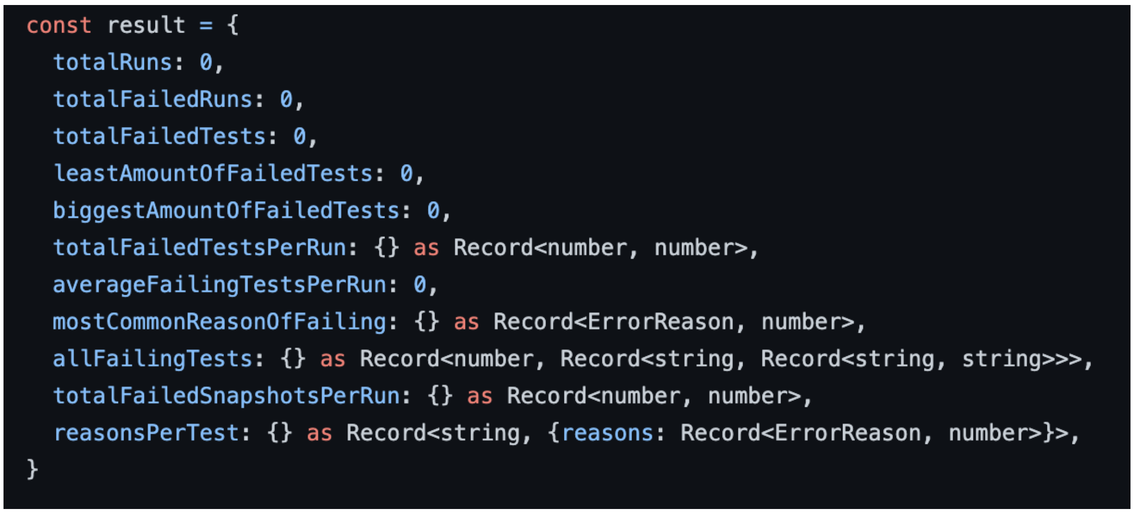
```
interface AssertionResult {
  ancestorTitles: string[],
  failureMessages: string[],
  fullName: string,
  location: any,
  status: string,
  title: string,
}
```

Figure 4: Interface of AssertionResult

```
interface SnapShot {
  added: number,
  didUpdate: boolean,
  failure: boolean,
  filesAdded: number,
  filesRemoved: number,
  filesRemovedList: any[],
  filesUnmatched: number,
  filesUpdated: number,
  matched: number,
  total: number,
  unchecked: number,
  uncheckedKeysByFile: any[],
  unmatched: number,
  updated: number,
}
```

Figure 5: Interface of SnapShot

Given the collected data and different scenarios, the following interface was the best way to aggregate results. See: Figure 6.

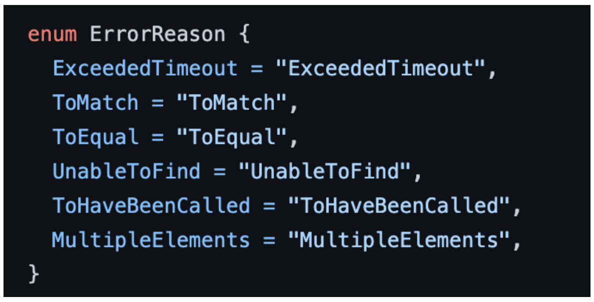


```
const result = {
  totalRuns: 0,
  totalFailedRuns: 0,
  totalFailedTests: 0,
  leastAmountOfFailedTests: 0,
  biggestAmountOfFailedTests: 0,
  totalFailedTestsPerRun: {} as Record<number, number>,
  averageFailingTestsPerRun: 0,
  mostCommonReasonOfFailing: {} as Record<ErrorReason, number>,
  allFailingTests: {} as Record<number, Record<string, Record<string, string>>>,
  totalFailedSnapshotsPerRun: {} as Record<number, number>,
  reasonsPerTest: {} as Record<string, {reasons: Record<ErrorReason, number>}>,
}
```

Figure 6: Interface analysis result

As seen in Figure 6 there are fields like most common reason of failing and reasons per test. For these fields the assertion results have been analysed and categories have been setup.

These categories have been made into an enum (Enumeration) to make it easier to work with and to have it more readable and dynamic in the interface. This enum is visible in Figure 7.



```
enum ErrorReason {
  ExceededTimeout = "ExceededTimeout",
  ToMatch = "ToMatch",
  ToEqual = "ToEqual",
  UnableToFind = "UnableToFind",
  ToHaveBeenCalled = "ToHaveBeenCalled",
  MultipleElements = "MultipleElements",
}
```

Figure 7: Enum for error reasons

When figuring out the error reason, the assertion results are being checked to contain a specific string. This is done in order of how specific the string is.

When an error reason is found it is both noted as the specific reason for that one test as it is being tracked as a general reason of error failing.

With these results available per run there are a lot of different diagrams to make both for an individual run as for a general dashboard. This means that this is a good start for the tool.

The whole script can be found in Appendix B: Test Analysis ScriptAppendix (Visser, Test Analysis Script, n.d.).

7.2.2 Results

As seen in the results, Appendix C: Result from script, we can conclude that integration tests are by far the most sensitive to flaking.

The main reasons for the flaking of the integration tests were:

- Certain texts could not be found,
- Components on the page could not be found,
- Functions were not called while this was expected.

The second thing that stood out was that there were a lot of timeout errors. All these errors mainly occurred in integration tests.

Within Jest there are so called snapshot tests. Snapshot tests are tests in which an entire page or just a component is being compared to a previously made snapshot of the page or component. These snapshots are saved in a separate file (Goss, 2022).

This snapshot testing is also called Visual Regression Testing. Here the test is being rendered and the result is being compared to a premade "snapshot" of the component (Bose, 2021). When snapshots don't match up, the test fails.

The integration tests at Velory are Visual Regression Tests. Since these integration tests are most common to flake, these are the tests that will be focussed on.

8 How can you find flaking tests?

There are a few different ways of finding flakes. Some are easy but take a lot of time and aren't fully reliable. Others are complex and take time setting it up but are thorough.

8.1 Keep running the test suite

The most obvious way is to run the test suite a few times and look at the results. Whenever a test both fails and succeeds in different runs it can be marked as a flaking test: you haven't changed the code and it succeeds and fails sometimes.

There are a few big disadvantages to this technique. The first one is that it can take a lot of time to run the test suite. This dependents on the size of the test suite.

The second disadvantages is that there is a big change you won't find all the existing flaking tests. It could be that a test flakes once every 100 runs. If you don't run the suite 100 times you wouldn't know about this test.

The third disadvantages is that tests are also dependent on the specific machine it is run on. The result can be different on an older machine then on a brand new one.

A big advantage is how easy and accessible it is to get results this way without over-engineering a solution.

An example of a company implementing this technique is Microsoft. See chapter "9.1 How do other companies solve this problem?" for more on their approach to test flaking.

8.2 Get the results from all the different machines and merge the result

A good effective way to get results from different environments is to hook into the actual testing process. When hooking into the actual testing process we can also take the covered code into account.

The covered code is the production code that is covered by the current running test. If this code has actually changed compared to the previous version there might be a good reason for this test failing.

When taking the covered code into account we can exclude changed tests from the flaking suspects since the change in the code might be the failing cause.

The data from the local machines plus the CI pipelines could be analysed to find the flaking tests.

A disadvantage to this technique is that it could take quite some time to get actual results.

8.3 Static analysis

With static analysis we can detect possible errors in the code before it is run. In languages like Java or C# we call this kind of analysis "on compile time". Which means that the code is checked on errors before it is compiled to machine code.

In the JavaScript eco-system we have tools like ESLint that achieve something similar. This option is quite inventive but unfortunately very complex. There are an undetermined amount of reasons why tests might flake. It's already tough to fix these mistakes when a test has been identified as flaking, let alone detecting this automatically.

However, there are a lot of principles and ideas about why a test can flake. All these principles and ideas can be converted to a program that can detect the possibility of flaking.

This principle is also used when detecting code smells. Certain patterns of code or usage of libraries can be recognized as fault sensitive. Tools like SonarCloud use this approach to warn developers of possible mistakes or security issues in their code (SonarCloud, n.d.).

Since a flaking test is often just a test that's written unsafely or instable, static analysis could be used to recognise flaking tests before they flake. This is, however, only in theory.

9 Designing a solution

Given the research thus far, we can conclude that we have to solve at least two main problems.

The first problem is the problem of visibility. It's hard to distinguish normal failing tests from flaking tests. In turn, this makes it difficult to estimate time spend on these tests and monetary cost made in unnecessarily running these tests.

The second problem is that the developers lose confidence in the test suite and get demotivated when they must retrigger the pipeline when they think a test is flaking.

By making the flaking tests visible it can solve a part of the insecurity of the developers. They can act on it to prevent it from flaking again.

9.1 How do other companies solve this problem?

There are a few companies that have said or wrote something down about test flaking. Some of these are made for internal use only but there are a few commercial options too. Here are a few examples of some bigger companies that have done so.

9.1.1 Spotify

Spotify has written an internal tool for detecting flaking tests called Odeneye. This tool is visualised as a big schema. Every individual run and test is visible in this schema. A developer can look at it and draw conclusions. For example when there is a vertical stripe of yellow dots there was a network failure (Palmer, Test Flakiness – Methods for identifying and dealing with flaky tests , 2019).

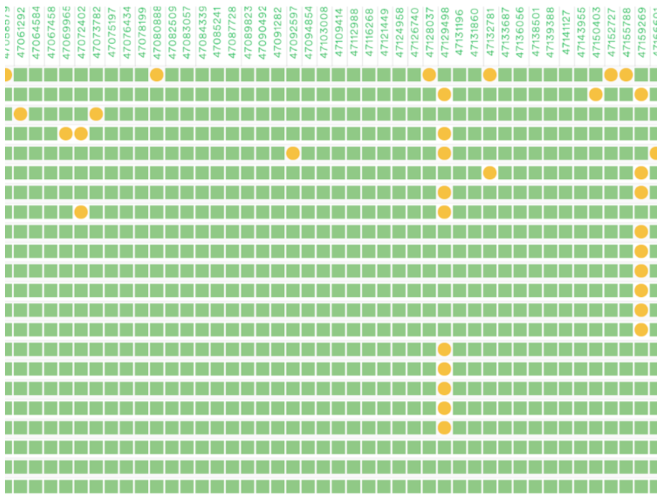


Figure 8: Visualisation of test flaking at Spotify

9.1.2 Microsoft

Microsoft also deals with flaking tests. In a talk they mentioned that for each green CI build, meaning every test succeeded in that build, they would run it another 500 times. If any of the tests fails it would be marked as flaking. In such a case a bug would be filed and the team would have to fix it (Microsoft Visual Studio, 2017).

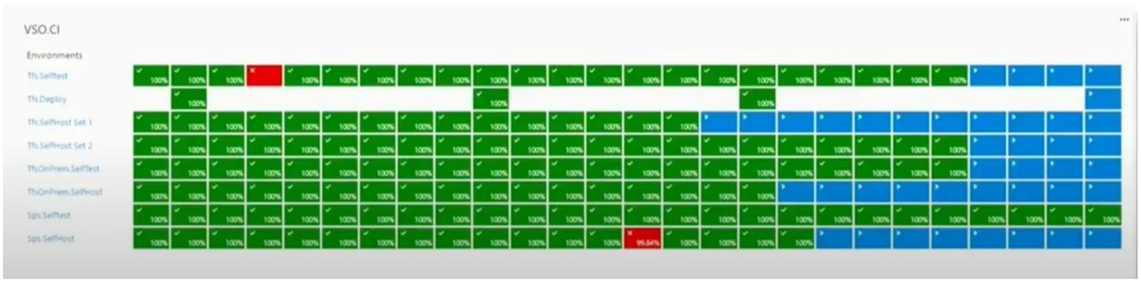


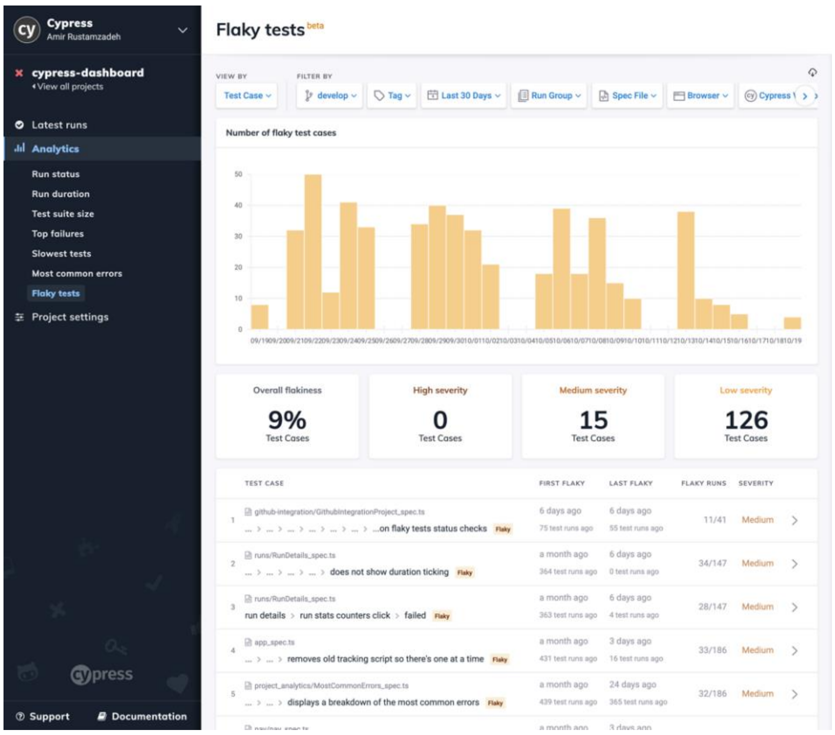
Figure 9: Dashboard of all the test runs from Microsoft Visual Studio

9.1.3 Cypress

Cypress also has the option to gain insights into flaking tests. They even have a specific dashboard for this.

However, this tool only works when using the Cypress testing framework and it's a paid feature.

In this dashboard they keep track of the past results, they retry the failing tests, they show different graphs to gain insights and they have the option to alert you about a flaking test on GitHub or Slack (Cypress, n.d.).



Overall flakiness	High severity	Medium severity	Low severity
9% Test Cases	0 Test Cases	15 Test Cases	126 Test Cases

	TEST CASE	FIRST FLAKY	LAST FLAKY	FLAKY RUNS	SEVERITY
1	github-integration/GithubIntegrationProject_spec.ts ... > ... > ... > ... > ... > ... > ...on flaky tests status checks Flaky	6 days ago 75 test runs ago	6 days ago 55 test runs ago	11/41	Medium
2	runs/RunDetails_spec.ts ... > ... > ... > ... > does not show duration ticking Flaky	a month ago 364 test runs ago	6 days ago 0 test runs ago	34/147	Medium
3	runs/RunDetails_spec.ts run details > run stats counters click > failed Flaky	a month ago 363 test runs ago	6 days ago 4 test runs ago	28/147	Medium
4	app_spec.ts ... > ... > removes old tracking script so there's one at a time Flaky	a month ago 431 test runs ago	3 days ago 16 test runs ago	33/186	Medium
5	project_analytics/MostCommonErrors_spec.ts ... > ... > displays a breakdown of the most common errors Flaky	a month ago 439 test runs ago	24 days ago 365 test runs ago	32/186	Medium

Figure 10: Cypress Flaking Test Management

This means they use a similar technique as Microsoft. They rerun the tests when it has already failed. Compared to Microsoft they do offer more information to solve the flaking tests.

9.1.4 TestProject

TestProject is a specific tool for Selenium, another testing framework. This used to be a paid service but during the research and developing of this project it became a free and opensource project (TestProject, n.d.). They took creating tests to the next level by implementing AI services to prevent mistakes.

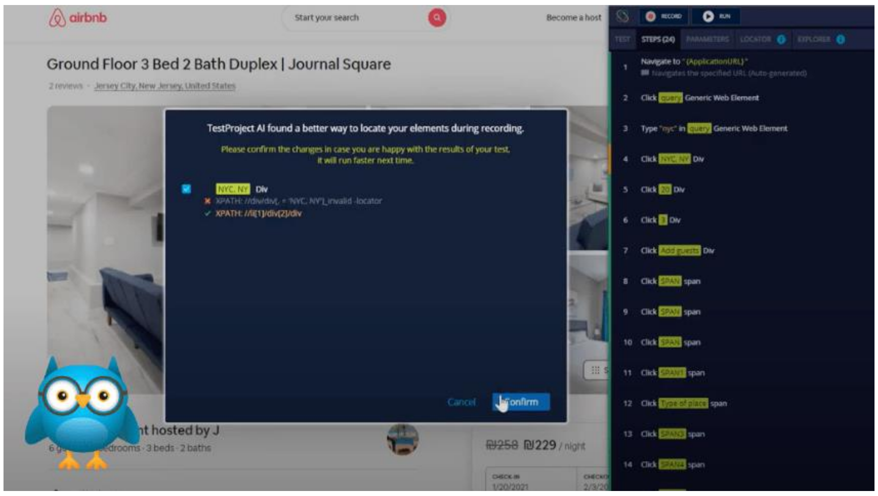
TestProject uses AI to optimize locator strategies, preventing UI dependencies and broken tests. They call this "Self Healing" because the tool does it for you.

The second thing they use AI for is to prevent failing tests due to variant loading times and async event. They call this "Adaptive Wait". They adjust the waiting time based on each individual test so that the developers doesn't have to build in the waiting time themselves.

The third thing TestProject uses AI for is to avoid flaking tests. They do this by analysing each step of a test to detect is a test reaches their target goal. If they find an inconsistency they automatically fix in during the run (TestProject, n.d.).

TestProject has a progressive way of detecting flaking tests. They take good care on the tests that flake because of UI shifts or the tests that would usually fail because of run time related issues.

There are, however, many more causes of flaking tests.


TestProject AI found a better way to locate your elements during recording.
Please confirm the changes in case you are happy with the results of your test.
It will run faster next time.

NYC, NY Div
✗ XPATH: //div/div[. = 'NYC, NY']_invalid -locator
✓ XPATH: //li[1]/div[2]/div

1 Navigate to "{ApplicationURL}"
2 Click query Generic Web Element
3 Type "nyc" in query Generic Web Element
4 Click NYC, NY Div
5 Click 20 Div
6 Click 3 Div
7 Click Add guests Div
8 Click SPAN span
9 Click SPAN span
10 Click SPAN span
11 Click SPAN1 span
12 Click Type of place span
13 Click SPAN3 span
14 Click SPAN4 span

Figure 11: Example TestProject AI

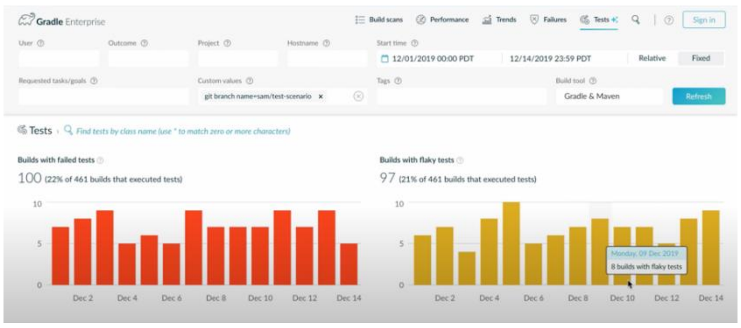
9.1.5 Gradle

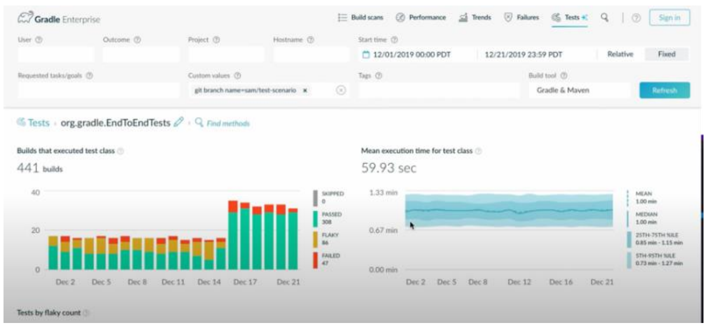
Gradle, a build tool for Java, C++ and Python projects, also has an option to detect flaking tests. This is an enterprise option, meaning you have to pay for it.

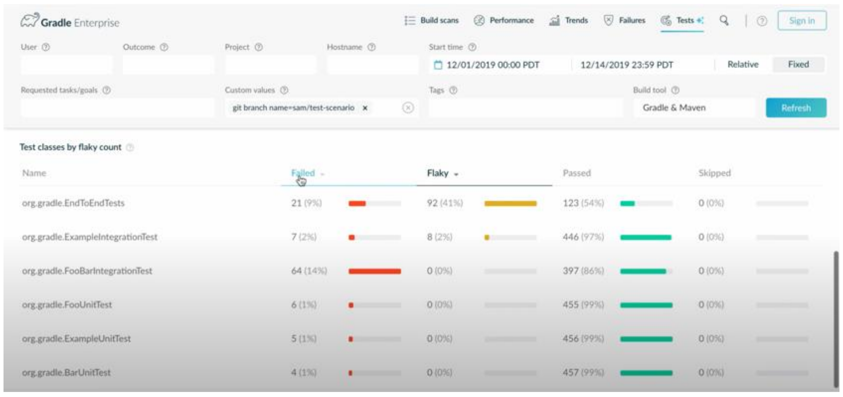
Gradle re-runs failed tests as part of the same build execution. When the tests succeed after re-trying, the test is marked as flaky.

The interesting part about how Gradle analyses it is that they keep track of both local and online test results. Giving them even more insights.

All this means they are quite similar to the other tools out there except they have a more elaborate dashboard making it easier for developers to solve flaking tests (Gradle, n.d.) (Gradle, 2020).







Test classes by flaky count

Name	Failed	Flaky	Passed	Skipped
org.gradle.EndToEndTests	21 (9%)	92 (41%)	123 (54%)	0 (0%)
org.gradle.ExampleIntegrationTest	7 (2%)	8 (2%)	446 (97%)	0 (0%)
org.gradle.FooBarIntegrationTest	64 (14%)	0 (0%)	397 (86%)	0 (0%)
org.gradle.FooUnitTest	6 (1%)	0 (0%)	455 (99%)	0 (0%)
org.gradle.ExampleUnitTest	5 (1%)	0 (0%)	456 (99%)	0 (0%)
org.gradle.BarUnitTest	4 (1%)	0 (0%)	457 (99%)	0 (0%)

Figure 12: Gradle dashboard

9.2 How are we going to solve it?

Looking at the dashboards of the other companies there are a lot of diagrams instead of just numbers. In combination with the colours you can glance at the screen and understand what is going. Unlike most of these companies there are no resources to run the tests on a loop. So we're going for another approach.

Wanting to finish a MVP, there is unfortunately also isn't time for one of the more advanced techniques to find flaking tests. The most effective and straight forward approach is to Get the results from all the different machines and merge the result.

This in combination with more advanced analytics and extensive dashboards make for a perfect MVP.

The dashboard are partially inspired by Spotify, Cypress and Gradle. There are more aspects to it which will be discussed in the following chapter.

9.3 How do we visualise the problem?

One of the three purposes of the tool is to visualise the problem. Since a flaking test can impact an entire project and an entire run it is important to show the impact on both.

Beside that it is important to give the developer all the relevant information to determine an approach to solve the flaking test.

This resulted in the idea of having three different dashboards:

- A general dashboard for the entire project,
- A general dashboard for a specific run and
- A dashboard for a specific test.

9.3.1 Project dashboard

The project dashboard has all the basic information about the project and a general overview of the results. This overview is all about giving a good analysis of where the project is at. You'll see the frequency and severity.

This dashboard can be used by a tech lead or a CTO to get a direct sense of how severe the test flaking issue is. This is the first screen an user seen when opening a project and developers can click on runs or tests to gain more insights on the tests.

To highlight a few items within this dashboard.

- You have diagrams that tell you how many builds have failed or are flaking,
- You have an overview of the latest runs,
- You have the top three categories of why the tests are flaking,
- You can see the total of the severity of flaking tests per severity level

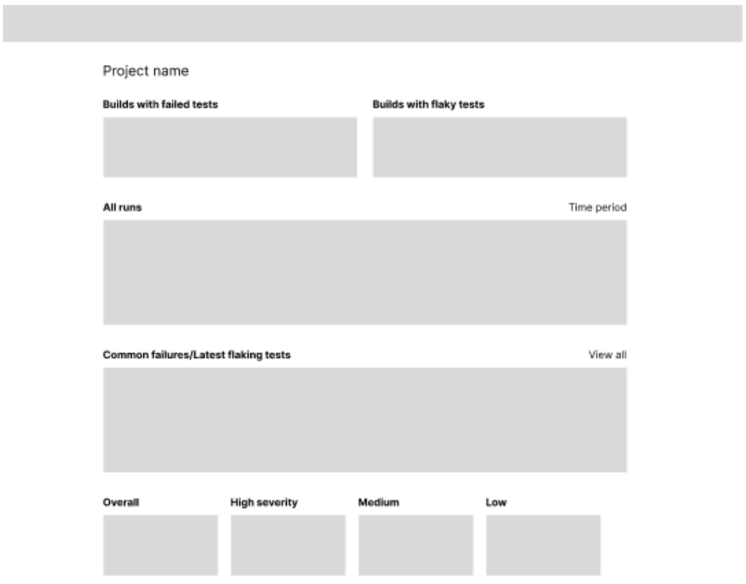
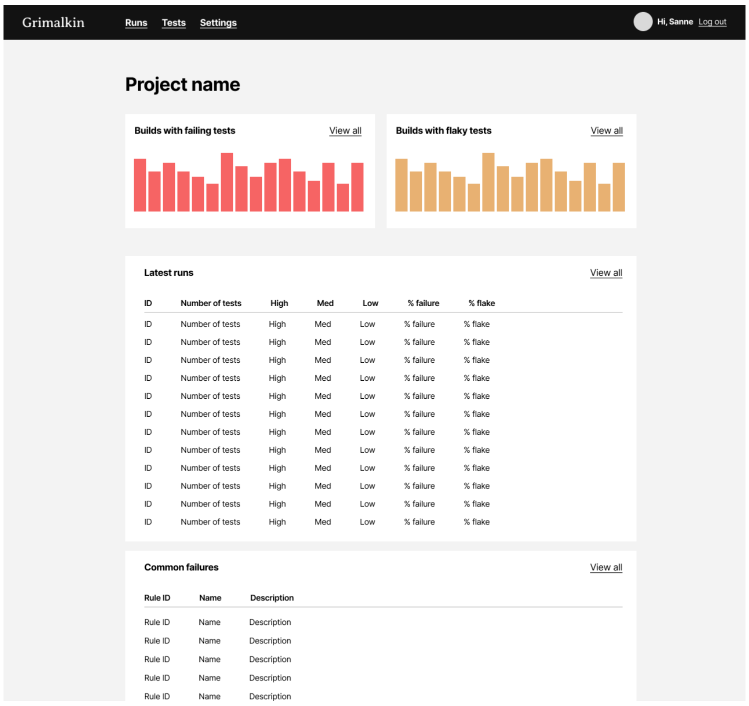


Figure 13: Wireframe project dashboard

To help developers determine what flaking tests to solve first and to help them solve them, they can navigate to a special flaking test dashboard.

This dashboard will only contain all the flaking tests and details about these tests. It is shown in order of severity.



Project name

Builds with failing tests View all

Builds with flaky tests View all

Latest runs View all

ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake

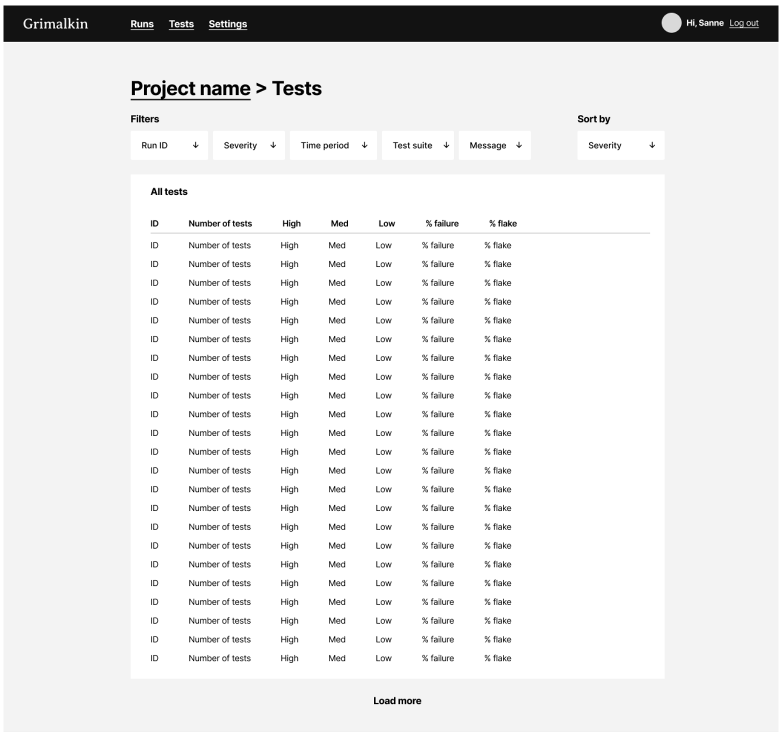
Common failures View all

Rule ID	Name	Description
Rule ID	Name	Description
Rule ID	Name	Description
Rule ID	Name	Description
Rule ID	Name	Description
Rule ID	Name	Description

Figure 14: Design project dashboard

Things the flaking test dashboard includes for all the tests marked as flaking within the project:

- The latest test results,
- What was the reason of the flake?,
- How often has it flaked in total,
- What was the last date time of the flake,
- During what runs did it flake?,
- What test suite is the test a part from.



Project name > Tests

All tests

ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake

Figure 15: Design flaking test overview

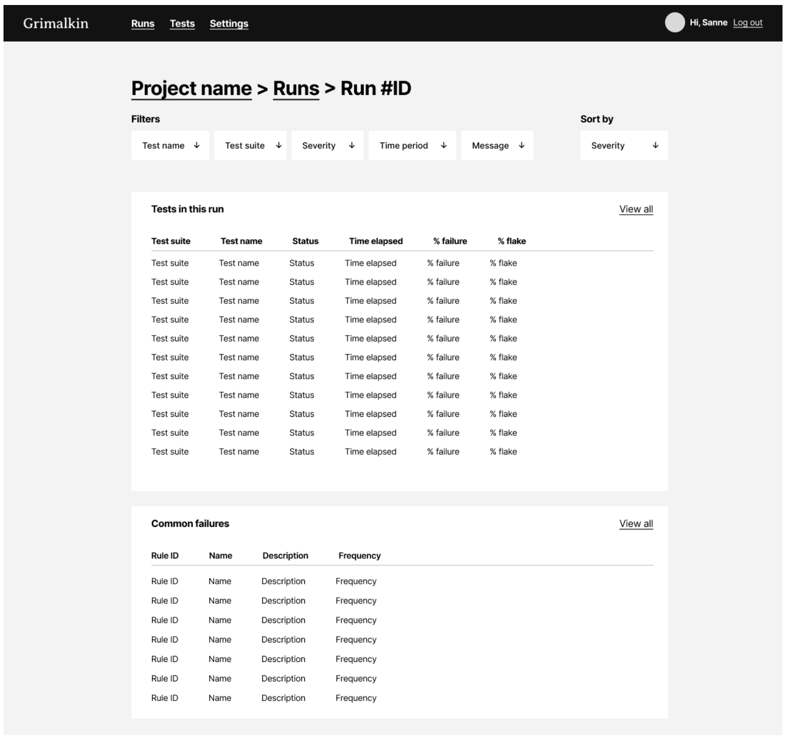
9.3.2 Run dashboard

The run dashboard is all about the in and outs of a specific run. What were the conditions of a run, how long did it take to complete (or fail)?

With this dashboard a developer can answer the question if the run failed because of network reasons or another collective issue. This is also a good view to get a sense of what tests are in a project.

To highlight a few items in this dashboard.

- The results of every test suite,
- Why did these tests fail,
- If there are any first time flakes.



Project name > Runs > Run #ID

Tests in this run

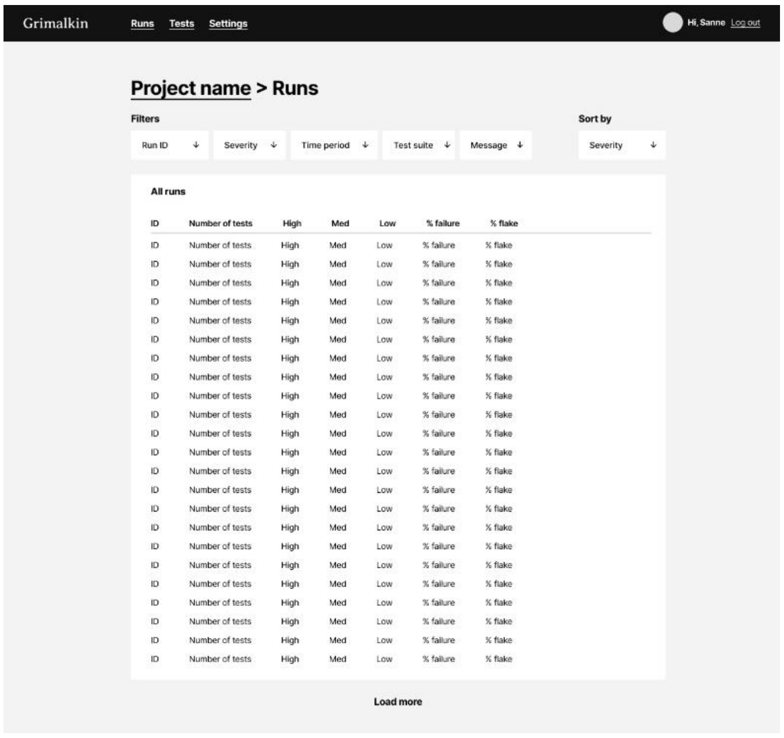
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake

Common failures

Rule ID	Name	Description	Frequency
Rule ID	Name	Description	Frequency
Rule ID	Name	Description	Frequency
Rule ID	Name	Description	Frequency
Rule ID	Name	Description	Frequency
Rule ID	Name	Description	Frequency
Rule ID	Name	Description	Frequency
Rule ID	Name	Description	Frequency

Figure 16: Design run dashboard

There is also a general overview of the runs with the flaking details as well. This could give insights in the reoccurrence of the amount of flaking tests.



Project name > Runs

All runs

ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake

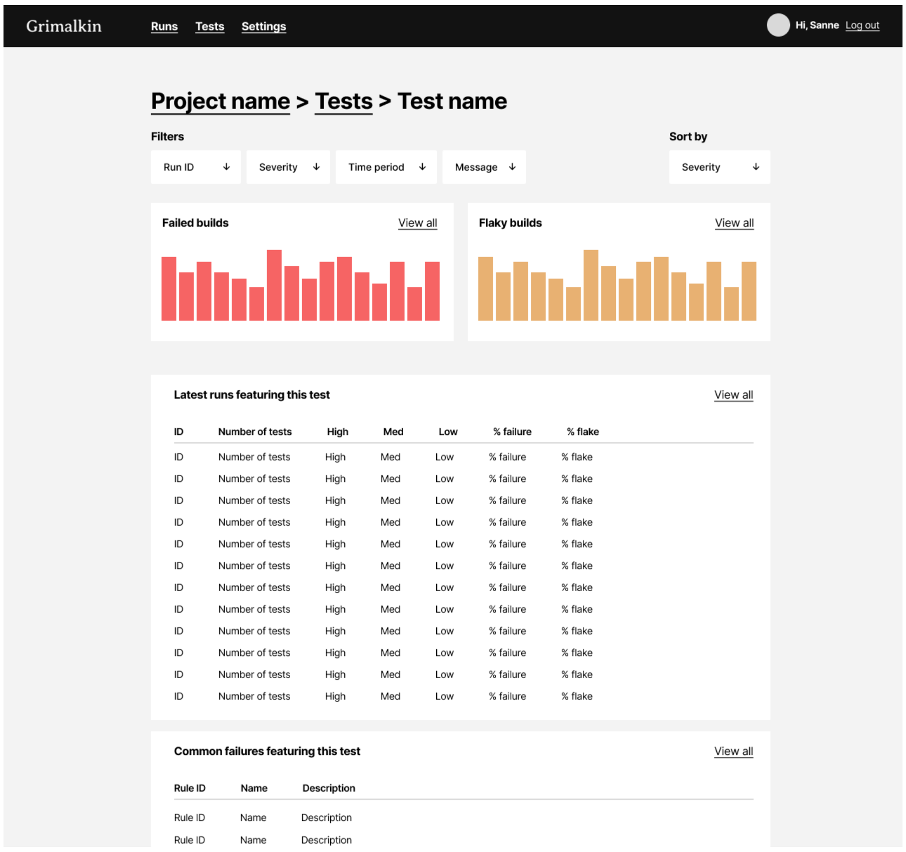
Figure 17: Design overview of runs

9.3.3 Test dashboard

The test dashboard includes all the specific information about a single test. It contains the following items:

- First + last time it flaked,
- Latest error message,
- Does it always have a similar error message?,
- Overview of all the times it flakes
    - Reason of the different flakes

This dashboard is designed to help solve the flake(s) within this specific test. By giving insights into the different reasons of the flakes a developer can determine if there is one or maybe more than one flake in the test. They also have a direction to search into.



Project name > Tests > Test name

Latest runs featuring this test

View all

ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake

Common failures featuring this test

View all

Rule ID	Name	Description
Rule ID	Name	Description
Rule ID	Name	Description

Figure 18: Design test dashboard

10 Creating the solution

Now with a concrete idea and the visualisation done it was time to get started. A lot of time has passed by at this point. It was important to get started right away to be able to deliver a product.

10.1 Data modelling

With the visualisation done it was time to create a data model.

Looking at the different dashboards you could get a sense of the general data that needs to be available. At first glance you would need the information about a company, about a project, the different test suites within a project, the different tests and the test results and general data per run.

Since a lot of this data is static, meaning it wouldn't change too often, the first idea was to use a NoSQL database, like MongoDB. This would be super easy to set up, implement and it would have a large free tier. Within a NoSQL database the data could look like Figure 19.

While modelling it became clear there were a lot of relations. Meaning that there would have to be a lot of get-requests or that a lot of unnecessary data would have to be retrieved each time.

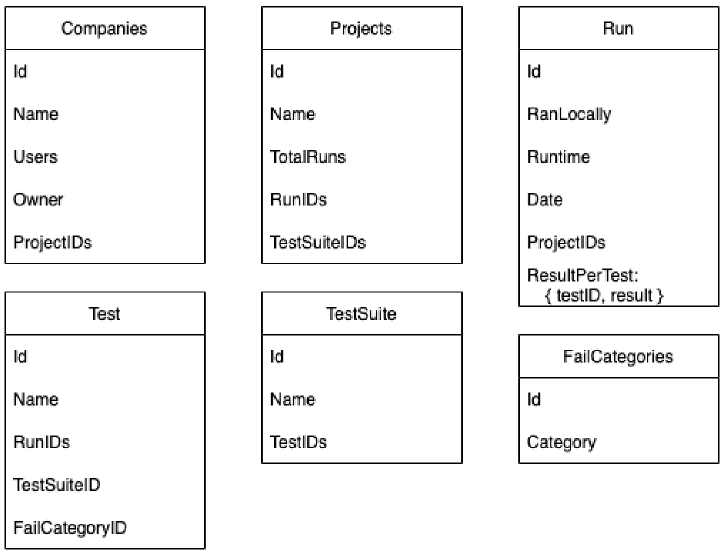


Figure 19: NoSQL Data Model

To explorer the option of having an SQL database there was another data model made. That one is seen in Figure 20.

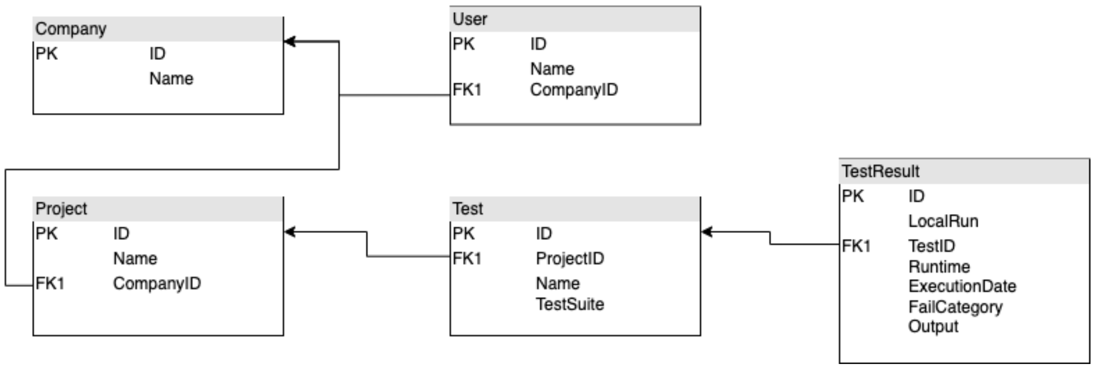


Figure 20: SQL Data Model

The SQL option would be faster and more dynamic in the long run. If there would be performance issues a View could be added. If it turns out the analyses take a long time or require a lot of power an additional aggregate table with the results could easily be added.

10.2 Set-up of the project

When you start setting up projects there are a few questions you need to ask yourself. What kind of project is it? What programming languages fit best with the criteria for the project? How am I going to deploy the project? And what is the best programming environment for the project?

On the first question we already have an answer: it's a tool to capture flaking tests. It's going to be open source and it should work well with the existing Jest Testing Framework.

10.2.1 What programming languages?

The question with the biggest impact on the project is what programming languages and, in extend to that, what frameworks are going to be used? The complete research can be found in the external attachment (Visser, What programming language is going to be used?, 2022).

10.2.1.1 Programming languages

While discussing this question there were a few things that need to be considered. The first is that it must work alongside the Jest Framework. The second thing is that it must be accessible for other developers to contribute to.

A lot of different languages like JavaScript, Rust and Ruby were looked at. But even though Rust and Ruby could be very useful we decided on JavaScript.

Jest is written in JavaScript, it's used in a lot of open-source projects, it's an accessible language and when used with Typescript it is a typed language which will make the error margin smaller. Besides that, JavaScript is the most used programming language (Stack Overflow, 2021).

10.2.1.2 Frameworks

Now the second biggest question on this subject, what frameworks will be used?

There are many more frameworks than there are programming languages. Since the language has been selected it can already be narrowed down.

We must decide on a framework for both the frontend and the backend.

The decision was made to have a separate REST API backend because it would make the boundaries between frontend and backend more clear. It would force the developer to think about the different API routes and which ones to expose to the outside world.

It would also separate the business logic better, having all the analysis on the backend.

10.2.1.2.1 Frontend framework

For the frontend the following frameworks were considered. These frameworks were chosen based on previous experience with the framework and the learning curve of the framework.

- ReactJS
- NextJS
- Remix

All three frameworks are React based. React has several advantages over other know frameworks like Vue or Angular, namely:

- React is developed and used by Facebook which means it will be maintained in the near future and it is build with scale in mind,
- React itself is more of a view library which makes it an all-round flexible tool,
- It is one of the most used and most popular front-end frameworks.

Weighing out the pros and cons of each framework ReactJS was chosen. This gives a lot of freedom in the chosen packages but there are no unnecessary packages meaning it will be a more lightweight tool.

10.2.1.2.2 *Backend framework*

For the backend another JavaScript framework was chosen with similar reasoning. It also had to be connected to a SQL database so it should be able to communicate with that.

The following options were considered.

1. Express
2. NestJS

Express would be the most basic option. It is a fast, unopinionated, minimalist web framework. NestJS is built on Express, meaning it has the same features and more.

Since NestJS already has Typescript build in and has an out-of-the-box application architecture, this framework is the best option.

The stack that was decided on is ReactJS, NestJS and Postgress.

10.2.2What environment?

The environment of the project is about the programming environment. Will the project be a monorepository or two single repositories? And where will the code be stored?

10.2.2.1 Code hosting platform

The easiest question in this category is where the code will be stored. The two most mainstream options are GitHub or Gitlab.

The biggest differences between GitHub and Gitlab are the configurations. Gitlab had a build-in CI/CD workflow and offers a complete software development solution. GitHub has less build-in services but offers a lot of integrations with many different programs and services. Besides that Visual Studio Code, the overall preferred code editor (Stackshare, 2022), has easier integrations for GitHub.

These are the reasons why GitHub was chosen to be the code hosting platform.

10.2.2.2 Monorepository or single repositories

The more elaborate question: will the project be in a monorepository or in two single repositories?

First, what are the differences and what are the effects of choosing either one.

A monorepository, also known as a monorepo, is a single repository with multiple projects. This could be a single frontend + backend but it could also consist of more than two projects. In fact, most of the Google products exist in a monorepo (Levenberg & Potvin, 2016).

An advantage of a monorepo is that there is more consistency within a monorepo. You have possibilities to make shared libraries which could include types, models or even components. But there could also be more consistency while deploying. When adding a feature, a single pull request can be made for both the frontend and the backend meaning there is no need to navigate between the two, or possibly more, projects.

Some disadvantages are that it is harder to separate the responsibilities of each project and it can be more tough to deploy. You must find a platform that supports the deploying of a monorepo (Powell, 2022) (Nx, n.d.).

We decided to keep all the code for this project in a monorepo based on the advantages above.

Nx was chosen as the tool to maintain the code repo because it simplifies and automates a lot of work. It has a predefined structure, and you can easily generate a project or library and all required configuration will be added automatically

Nx makes scaling different applications easier by doing code change analysis to see what apps or libraries are affected by a particular pull request

Nx also has a plugin for Visual Studio Code called Nx Console. It does what the command-line version of Nx does - it analyses the same meta information to create the needed UI. This means that anything you can do with Nx, you can do with Nx Console. This makes developing and testing even easier (Nx, n.d.) (Nx, n.d.).

10.2.3How to deploy?

There are different ways to deploy an NX app. You can deploy it as a whole, having it update in its entirety every time an update is pushed. Or you can deploy the apps separately, which would mean there is more networking going on if it's a backend and frontend.

The three options that directly came to mind were AWS (Amazon Web Services), DigitalOcean and Heroku. AWS, DigitalOcean and Heroku are cloud infrastructure providers. These providers have are in different price ranges, they are mainstream and they provide what would be necessary to deploy this project.

You could say AWS is the most flexible, you can do anything and make all different types of configurations. This is also the tricky part.

DigitalOcean is still flexible but you have a lot of preconfigured options and guides on how to use the different services.

Heroku is the most rigid of the three. However, they do offer full stack deploying of NX apps which the other services don't.

Pricewise AWS would be the cheapest option, especially when the product is being used more. Heroku is the most expensive when the product is being used a lot but starts with a free tier. DigitalOcean starts at $5 but the price doesn't increase as fast.

For these reasons the project is being set up with Heroku.

10.3 How do you make a Jest plugin?

Having the frontend and backend set up it is now time to get the data of each test run into the database. The most efficient way of doing this is to hook into Jest itself.

Jest has the option to add a watch plugin to a project. This means you can add your own code and make use of the hooks Jest exposes.

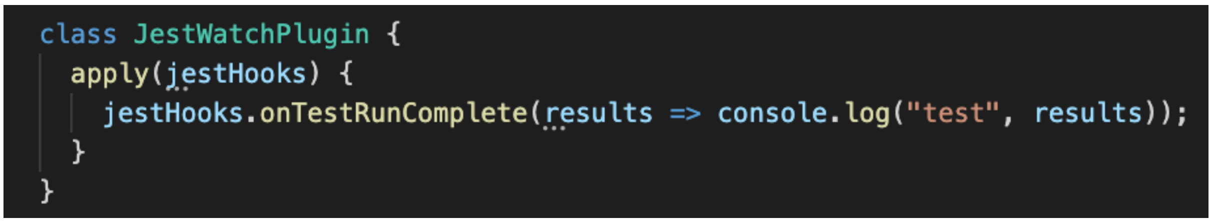
Hooks that Jest exposes are shouldRunTestSuite, onTestRunComplete, onFileChange. The names do speak for themselves.

The hook that will be used to retrieve the data and to send it to the API is the onTestRunComplete hook. The hook gets the results in the format seen in Figure 22 (JestJS, n.d.).

To create a watch plugin for Jest all you have to do is generate a project with Node and import this in the project you would like to hook up.

Creating such a project is done by picking or creating a folder, opening a terminal, switching to the right directory and writing "npm init" in the terminal. This generates a package.json file.

In this folder you will create a file index.js (or ts if it's going to be Typescript). In this file you will create a JavaScript class which could look like the class in Figure 21.



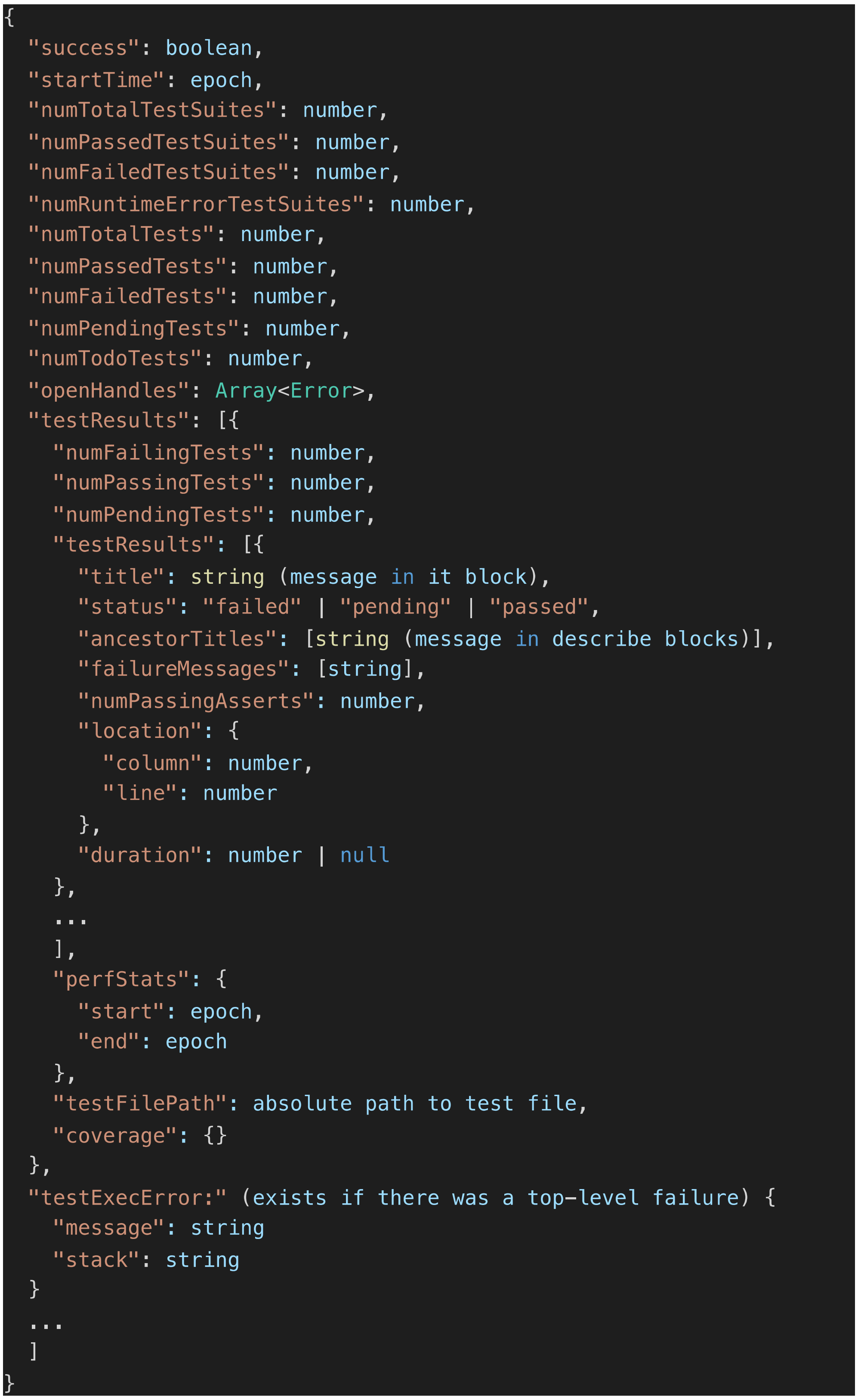
```
class JestWatchPlugin {
  apply(jestHooks) {
    jestHooks.onTestRunComplete(results => console.log("test", results));
  }
}
```

Figure 21: Example of a Jest Watch plugin

Once this is created it can be pointed towards from within the jest.config.ts file of the project.

Now the only issue is that this cannot be reached from outside the computer it has been created on. To solve this a NPM package can be made, now it can be imported from within any Node project.

This is done by writing "npm publish [<folder>]" in the terminal.



```
{
  "success": boolean,
  "startTime": epoch,
  "numTotalTestSuites": number,
  "numPassedTestSuites": number,
  "numFailedTestSuites": number,
  "numRuntimeErrorTestSuites": number,
  "numTotalTests": number,
  "numPassedTests": number,
  "numFailedTests": number,
  "numPendingTests": number,
  "numTodoTests": number,
  "openHandles": Array<Error>,
  "testResults": [{
    "numFailingTests": number,
    "numPassingTests": number,
    "numPendingTests": number,
    "testResults": [{
      "title": string (message in it block),
      "status": "failed" | "pending" | "passed",
      "ancestorTitles": [string (message in describe blocks)],
      "failureMessages": [string],
      "numPassingAsserts": number,
      "location": {
        "column": number,
        "line": number
      },
      "duration": number | null
    },
    ...
    ],
    "perfStats": {
      "start": epoch,
      "end": epoch
    },
    "testFilePath": absolute path to test file,
    "coverage": {}
  },
  "testExecError:" (exists if there was a top-level failure) {
    "message": string
    "stack": string
  }
  ...
  ]
}
```

Figure 22: Test result type from Jest

10.4 When do we mark a test as flaking?

One of the most important things is to differentiate between a test that is flaking and a test that is failing.

The criteria of a failing test is that the code has been changed. One of the Jest hooks that is available is the onFileChange hook. As the name suggests it retrieves all the test paths that Jest is watching.

The files retrieved from this hook in combination with the test results can conclude if the test has been changed, and thus if the test is failing or flaking.

The only issue we have is that Jest Watch is disabled in CI pipelines. This means there will have to be another solution to retrieving the flaking tests in online environment. Since there is no time to follow up on this this would be a great next step in research.

10.4.1 When did it stop flaking?

As easy as it is to mark a test as flaking, it is a little harder to decide when it stopped flaking.

As we know a flaking test is not predictable. There is a possibility a test flakes once every 100 runs. This means that you can't unmark a flaking test if it has passed 10 times.

You need to find a balance. You could say that if there has been a change to the test you remove the marker and add it again if it flakes again. Or you could remove the marker if it has been changed and wait a few runs to remove the marker.

In this case it makes more sense to remove the marker after a change in the test. If the test flakes again after the marker will be added again. Taking this approach a developer can feel better about his work because their effort is rewarded.

11 How could the project be sustainable?

The project will be open source, meaning there is no direct payment for the usage of the tool. However there are costs for the tool. The tool needs to be hosted and the data needs to be stored.

If the tool won't be used, it won't cost much because there are a lot of free tiers to host both the apps as the database. But when it will be it would already have been taken into account.

A structure that comes along often in the open-source community is pay for what you use, also called freemium. If you can host the apps yourself you can use everything for free. If you want the convenience of having it hosted, you can pay for that service.

There will also be a free tier which is designed to get a sense of how the tool works or that can be used for small projects.

11.1 When do you need pay?

As said before, if you want the service to be hosted for you, you will have to pay. The question is, when do you need to pay?

Within this project you cannot focus on how many projects or runs a person or company has tracked because the size of the project depends on the total of runs and the run depends on how many tests there are within a project.

This means that if a company can track one project for free and they'll track it for a year it will cost a lot of money.

Looking at the data model, it is quite easy to see what datatype is being used the most. In this case it is the TestResult table. For every test there are multiple test results, one for each run.

Putting a cap on the maximal test results mean a person or company can track multiple projects, and thus get familiar with the tool, but cannot drain all the resources.

12 Result and Discussion

12.1 Result

In this chapter we quickly summarize the results produced by this research. Next to the written research we also developed a proof of concept to prototype the findings from the research.

12.1.1 Research

The definition of a flaking test is a test that can either fail or succeed without having changed any relevant code. The category of tests that flake the most within the product of Velory are the Visual Regression Tests/integration tests.

To gain insights in the bigger picture and in the specifics of why a test can possibly flake an analysis script had been made. These results are being used to visualize the problem.

With inspiration from similar products and the results from the analysis different dashboards have been composed. There are three general dashboards and some additional dashboards with more specific information.

The project dashboard has general information about the project and some insights on how severe the flaking problem is. The test dashboard in which you find the specifics of that test and the history of flaking of that test. This gives the information that could help solve the flake. The run dashboard has more specific information about the run and the statistics of test flaking within that run, which could also help solve the flaking tests.

The flaking test dashboard shows the current flaking tests within the project in order of severity (how often it has flaked). This can be used to determine what test has a priority to fix.

The project is realised with ReactJS, NestJS and Postgress. The code is stored on GitHub and will be deployed on Heroku. It will be sustainable by making use of the freemium model.

Beside all the research that has already been carried out and the questions that are already answered there are still a few unanswered questions.

Are the results from the analysis similar to other companies? This use case has been specifically for Velory so the biggest problem of Velory had to be solved, but would the results of other companies also point out that integration tests are the most prone to flaking?

Could this tool be more than a test flaking capture tool specifically for Jest? The backend and frontend are not Jest specific, in theory this would mean that any testing framework could sent data to the backend and have it analysed.

Are there more possible insights about test flaking based on the current data? Having more eyes and more minds could result in gaining more insights in the test results we currently have. This is an exiting thought because it could mean that solving flaking tests can be made easier.

What is the best way to test a Jest plugin? During the development process this felt like it was impossible. Having more time to research this could make the process of developing this way easier.

How do we get test results from the CI pipeline? At first it seemed possible to get the test results from a CI pipeline via a Jest Watch plugin. After reading an article it became clear it wasn't because the Jest Watch option is not available in CI pipelines because usually it uses unnecessary computing power. There has to be an alternative way of achieving this result.

12.1.2 Product

The product will be an open-source test flake capture tool for software projects that use Jest. It captures the flaking tests both from CI pipelines and from local test runs. Developers can use the different dashboards to help solve flaking tests and a tech lead or CTO can gain useful information about the current status of test flaking. The product can be used by anyone following the freemium model.

Currently, the complete foundation of the product has been made. The database is set up, the backend has all the API routes, the first dashboard is completed, and a start has been made on the Jest plugin.

The MVP planned for this project was a working backend and frontend including all the base features like collection and aggregating test results and displaying them in a useful manner. Unfortunately, not all dashboards have been finished in time. Due to a delay in design and wireframing only the general overview is finished. This, however, still demonstrates the intent and usefulness of the project (Visser, Grimalkin, n.d.).

From here on the other dashboards can be developed, since the designs are ready to be implemented. Furthermore, to complete the original vision a dedicated server and payment system can be implemented to support the freemium model.

12.2 Discussion

In this chapter I discuss any flaws, open ends or continuations for this research and accompanying project.

As told before there are a few unanswered questions that need some attention. There also are a few subjects that would be amazing if continued upon.

12.2.1 Restrictions

Most of the research is done within the boundaries set by Velory. The project has mainly been for their benefit. Since it’s an open source project it could be useful to research different projects and include the findings in further product design. By taking open source projects as use case we can analyse a broad set of different types of projects. Running their test suites N amount of times and injecting the data into the tool could result in a more complete picture and a more general useful product for other use cases.

This project has been restricted to the use of Jest only. However, there are a lot of other testing frameworks. The current backend and frontend are test-framework agnostic. This means, that when other frameworks allow to extract the testing data on runtime, they could be used as input for our tool as well.

12.2.2 Open ends

What is the best way to test a Jest plugin? For this question there has to be desk research. In the short time I've researched this nothing came up. If there is no proper answer then it could be asked on forums like StackOverflow or there are some experiments that could be done.

How do we get test results from the CI pipeline? This is a big question and it feels terrible that I haven't been able to answer this yet! The approach that can be taken with this is to look at how Gradle has done this. Do they run their pipelines themselves? If they do, is there another way to hook into Jest to retrieve the test data? Maybe in the form of a reporter?

12.2.3 Improvements and opportunities

An improvement that can be made to matching the error categories. A division can be made for the "toMatch" category. This error often contains more information, like what it couldn't match to. Having this split up would give more insights and it would help developers fix flakes faster.

An opportunity that would be fun to follow up on are the different ways to find the flaking tests. Writing a script that puts the numbers of the coverage of the code to the amount of flaking tests is a very cool concept!

Another opportunity is to pick up another framework for the frontend. RemixJS sounds like it would suit the project very well. If I ever get the chance to rewrite it, I would.

A big and innovative opportunity is to dive deeper into solving the flakes. An idea I have to continue upon what TestProject has made is to run every test in a single environment. This could show tests that flake because of shared data, other environment issues or mocking issues.

13 Reflection

Having worked at a really fun company with amazing colleagues in a wonderful city has giving me a lot to reflect on as well.

As elaborated in the preface I have been doing this internship abroad. In Sweden to be specific. I've been living on my own for the past 6 months and I have learned a lot, both about myself as in the field of work.

Living abroad has played a big part during this internship.

13.1 Process

When I started at Velory I was warmly welcomed by the colleagues in the Gothenburg office. Bruno, one of my direct colleagues, was assigned to get me started.

He helped me setup a few things and showed me around in both the physical office as in the product they've been working on. The first few days I was to shadow Bruno.

He picked up an important item and he walked me through his process of getting this into the product. I quickly caught on and realized he was working on something that would consume a lot of time that he would have to delete later on. I brought it to his attention and we discussed it.

Having this as my first day felt like a big win and I was ready for more!

13.1.1 After those first few days

I started on getting to know more about flaking tests. I had done a bit of research before to see if the assignment was achievable but I didn't really get the chance to dive deeper.

I watched a lot of video's, read a lot of articles and tried to find out as much as I could.

After a week of watching Bruno work and the pipeline failing at his pull-requests I quickly realized how big the issue was. I logged some of the issues but there were too many. I noticed some of the issues were reoccurring which was inline with what I had learned about test flaking.

When I started working on according to my planning, meaning I started my Jest research, I realized that this wasn't a priority.

The reason why I put Jest as a starter was to gain more insight into why tests would flake but I learned more from reading about it else where then looking into Jest. Jest was just the test runner, all I could get from that were the test results.

I chose to review all of my choices in the planning and in the deliverables. I sat down with a lot of post its and I started brainstorming.

13.1.2 Brainstorming session

I didn't want to do this project with the wrong end in mind. So doing a brainstorming session meant having a clean slate and get ideas from the "why's" and "what's".

I started with what is the problem? And why is it a problem? I tried to find the minimal solution to fix this problem.

A few of the issues Velory had was that a flaking test costs a lot of time to fix, that it costs money, that the developers lost faith in the testing suite and that the developers felt really bad every time a test flaked.

With this in mind I could setup the right question to answer in this thesis: “How do we manage flaking tests to reduce cost and preserve the mental well-being of the developers?”.

Now, beside the reasons behind why this was an issue I had to find a solution that I would be able to finish in set time and that meant specifying what different solutions there could be.

At first, how can we detect flaking tests? There are four ways to detect flaking test, two of which are more advanced.

The first two proven ways of detecting flaking tests are to get the test results from each run or to brute force the test suite, meaning running it over and over again. These ways are similar but the difference is that the brute force way takes up a lot of resources and that the gathering of the results take time.

The two innovative ways of detecting flaking tests are to look at the test results in combination with code coverage or to analyse the code for code smells.

To incorporate code coverage means that if the code coverage is different for each run while the code hasn't been changed there is something wrong with the tests.

To incorporate code smells is really advanced. This technique would look at the different angles of the code to see if there are code smells. There is a theory that if you have clean code, meaning no code smells, there will be way less flaking tests.

Taking the time that I had left into consideration I thought it'd be best to start off with the first way. With this solution I had the biggest shot at finishing a product.

13.2 Product

Going into this thesis I expected myself to deliver a completely functional product, a MVP. Having not completely delivered that I am a little disappointed.

I knew this journey was going to be tough but it still wasn't what I expected. Having more struggles in my personal life and within the company, I'll come back to this later, I get why this is the way it went down.

At the time of writing this reflection I have the absolute basics setup for the product, there is a functional frontend and backend and the wireframes are being turned into actual pages. The database is setup and the basics of the Jest plugin stand.

Jack and I even thought of a name! Grimalkin. Which refers to a witch that can transform into a grey cat. The cat in this name is a wink to Schrodinger's cat.

This product is far from done and I will continue working on this after this thesis.

13.3 Competences

During this thesis there were a few competences that needed to be proven. Those were categorised in two categories. The A-competences, which are about the general attitude and skills of a student, and B-competences, which are about the field of expertise, so in this case Software Engineering.

13.3.1 A-competences

The A-competences are Research, Learn to Learn, Work professionally and Innovation.

Research is a big part of a thesis. As seen in the Jest research, the document has a clear and function format. The results are ordered by question and can be referred back to when necessary.

The Learn to Learn competence is already being fulfilled by moving to Sweden. The learning curve of finding a company to work at, finding a place to live at, learning about the culture is already big. Let alone doing all of it, like moving and having to live there on your own, it is massive.

Work professionally, meaning being in a work place and communicating and planning accordingly went well. I've talked a lot with my company supervisor and I had a lot of communicating to do when the company went through a reorganisation.

Innovation wasn't hard with this subject. Besides having the brainstorm session to figure out the problem even better. There had to be a lot of out-of-the-box thinking to figure out how to deal with certain issues like how to deal with the reorganisation and how to work from home.

13.3.2B-competences

The B-competences were harder to keep an eye on. In the graduation plan these competences were chosen and it was explained how these would be proven.

There has been a big change in schedule and deliverables which made it harder to track. Besides of having examples in the content of this thesis there is a chapter for each competence to show how I complied with them.

13.3.2.1 Analysing software

This competence was focussed on the research within the Software Engineering field of expertise.

There were a lot of researches that took place. Some were thorough and got a dedicated document to it. Others were less formal and noted were taken in Notion, the program I used to keep track of all my sources, summaries and to-do items.

Some of the more official researches that took place were " What are flaking tests?", "How does Jest work?", "How can you find flaking tests?", "What programming languages are we going to use?" and "How do you make a Jest plugin?".

Some of the more unofficial but important researches were "How do I aggregate test data to a useful format?", "How do other companies solve this problem?", "When do we mark a test as flaking? And when did it stop?" and "How could the project be sustainable?".

13.3.2.2 Software engineering

This competence is shown by the research, "What programming languages are we going to use?".

By researching what programming languages could be used, and with that what frameworks, I have shown I can weigh out the pros and cons to come to the best decision. I took into account that the project will be open-source and that the options I chose would be around for a while.

Beside the programming languages I had to look into where to deploy the apps and what the ideal environment (monorepo vs single repos) would work best for this application.

All these decisions were made after a discussion about my findings with colleagues.

13.3.2.3 Developing software

This competence is shown in the script to analyse the test data, the Jest plugin and in the source code for the project, Grimalkin. All the code can be found on my GitHub page as public projects.

The source code for Grimalkin exist of both a frontend and backend. The components within this project have been carefully chosen to be easy to learn and be readable.

To keep the standard of the project high all the pull-request will be reviewed. This means that all the code that is submitted to the main branch, meaning the code that will be deployed and continued to be worked on, will be checked and tested by another developer.

13.4 Experience

As talked about before, I have been living in Sweden on my own during my entire thesis.

To me this meant being away for half a year. Being away from family (including a 1 year old nephew!), friends, my cat and, the biggest one of all, my partner who I had been living with for the past few years.

This was a challenge on it's own and video calling with them has helped a bunch.

A big, tough lesson I learned during my time at Velory was that you never know what a company is going through.

Everything was well and things seemed to go amazing, until one Monday when a reorganisation was announced. This meant that half of the company was being laid off.

This in combination with Jack leaving to take on a new challenge meant a lot more people were leaving too. Jack was close friend with Bruno and Adam and they both took on new challenges too. This left the Gothenburg office with only one employee, me.

They closed the office and this meant I had to work from home the last week. This was a hard pill to swallow but a really valuable lesson!

14 Afterword

I would like to thank you for reading this thesis. It means a lot to me and I'm very grateful for this opportunity and the time you have given me to share my research and my experiences.

I hope you have learned something and maybe I even inspirited you to get the absolute most out of the situation you're in!

I would love to get your feedback, questions or ideas and I'm always open for a discussion! Reach out to me on LinkedIn if you are up for any of those things.

https://www.linkedin.com/in/sanne-visser-dev/

I can't express how grateful I am to the people who have sticked by me till the end.

Thank you Jack, for helping me even when you've left the company and being there for me when I needed it.

Thank you Lars, for being my biggest supporter even with this big distance between us.

And thank you reader, for your time and energy. I appreciate it!

Cheers,
Sanne

15 Bibliography

Avidon, E. (2019, May). *flaky test*. Retrieved from TechTarget: https://whatis.techtarget.com/definition/flaky-test

Bose, S. (2021, January 18). *Introduction to Visual Regression Testing*. Retrieved from BrowserStack: https://www.browserstack.com/guide/visual-regression-testing

Craske, A. (2021, June 18). *How Google Does Monorepo*. Retrieved from QE Unit: https://qeunit.com/blog/how-google-does-monorepo/

Cypress. (n.d.). *Flaky Test Management* . Retrieved from Cypress: https://docs.cypress.io/guides/dashboard/flaky-test-management

Goss, D. (2022, February 23). *Snapshot Testing*. Retrieved from JestJS: https://jestjs.io/docs/snapshot-testing

Gradle. (2020, June 5). *Best Tool for Managing Flaky Tests*. Retrieved from Youtube: https://www.youtube.com/watch?v=C5_jgsRANuI&ab_channel=Gradle

Gradle. (n.d.). *Failure Analytics*. Retrieved from Gradle: https://gradle.com/gradle-enterprise-solutions/failure-analytics/

Integration testing . (2021, October 11). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Integration_testing

JestJS. (n.d.). *Watch Plugins*. Retrieved from JestJS: https://jestjs.io/docs/watch-plugins

Levenberg, J., & Potvin, R. (2016, July). *Why Google Stores Billions of Lines of Code in a Single Repository*. Retrieved from Google Research: https://research.google/pubs/pub45424/

Microsoft Visual Studio. (2017, October 25). *Eliminating Flaky Tests*. Retrieved from Youtube: https://www.youtube.com/watch?v=Q4c5cvt1b3k&ab_channel=MicrosoftVisualStudio

Nx. (n.d.). *Monorepos*. Retrieved from Nx: https://nx.dev/guides/why-monorepos

Nx. (n.d.). *Nx Editor Plugins*. Retrieved from Nx: https://nx.dev/using-nx/console#nx-console-for-vscode

Palmer, J. (2019, October 8). *Assert(js) 2019: Jason Palmer (Spotify) - Test flakiness; methods for dealing with flaky tests* . Retrieved from Youtube: https://www.youtube.com/watch?v=38pW08_nY_k

Palmer, J. (2019, November 18). *Test Flakiness – Methods for identifying and dealing with flaky tests* . Retrieved from At Spotify: https://engineering.atspotify.com/2019/11/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/

Palmer, J. (2019, November 18). *Test Flakiness – Methods for identifying and dealing with flaky tests* . Retrieved from atspotify: https://engineering.atspotify.com/2019/11/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/

Powell, R. (2022, May 13). *Benefits and challenges of monorepo development practices*. Retrieved from Circle CI: https://circleci.com/blog/monorepo-dev-practices/

SonarCloud. (n.d.). *Save time and effort with higher Code Quality*. Retrieved from sonarcloud: https://sonarcloud.io/code-quality

Stack Overflow. (2021, June 15). *2021 Developer Survey*. Retrieved from Stack Overflow: https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language-prof

Stackshare. (2022, May 26). *IntelliJ IDEA vs Visual Studio Code*. Retrieved from Stackshare: https://stackshare.io/stackups/intellij-idea-vs-visual-studio-code

TestProject. (n.d.). *AI That Works For You* . Retrieved from testproject: https://testproject.io/ai-tools/

TestProject. (n.d.). *TestProject*. Retrieved from Github: https://github.com/testproject-io

Visser, S. (2022, February 17). *Capturing Test Flaking - Graduation Plan*. Retrieved from Google Drive: https://drive.google.com/file/d/118yctL8c4ck5e9yaur7sam1kxq41H9jT/view?usp=sharing

Visser, S. (2022, March 4). *How does Jest work?* Retrieved from Google Drive: https://drive.google.com/file/d/12M1W1Bqqj1xUBdn9Q59KAhElqxOAnrcS/view?usp=sharing

Visser, S. (2022, June 11). *Research*. Retrieved from Google Drive: https://drive.google.com/file/d/1H7tsD6SE7kx28_1Yv853t9uPcHdNQ8Ke/view?usp=sharing

Visser, S. (2022, June 11). *What programming language is going to be used?* Retrieved from Google Drive: https://drive.google.com/file/d/1GuoSWZvvqanllJFBXFYDUVwlN2WWNoUk/view?usp=sharing

Visser, S. (n.d.). *Functional and technical design*. Retrieved from Google Drive: https://drive.google.com/file/d/1Jh-ovnYveyfxYTWbOM3QZnHQ1BVpr8fO/view?usp=sharing

Visser, S. (n.d.). *Get Test Results*. Retrieved from Github: https://github.com/VisserSanne/AnalyseJestTests/blob/main/getTestResults.bash

Visser, S. (n.d.). *Grimalkin*. Retrieved from GitHub: https://github.com/VisserSanne/grimalkin

Visser, S. (n.d.). *Test Analysis Script*. Retrieved from Github: https://github.com/VisserSanne/AnalyseJestTests/blob/main/analyzeTests.ts

16 Appendix: Figures

Figure 1: Stakeholder analysis

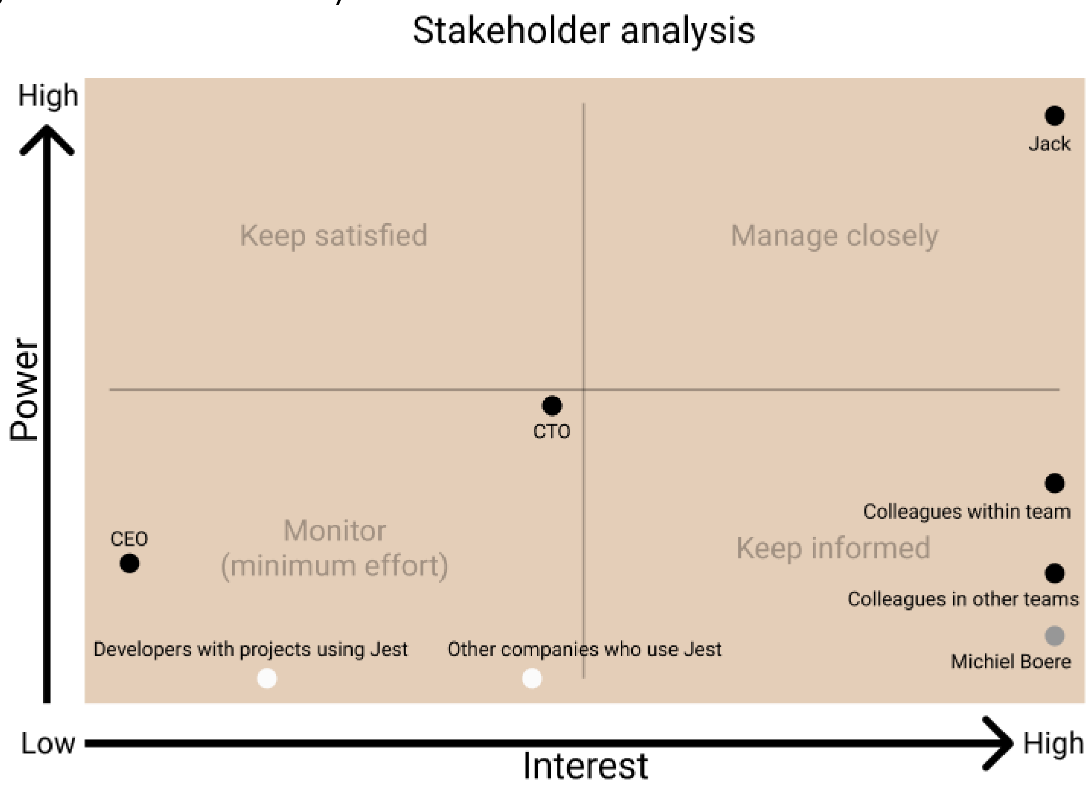
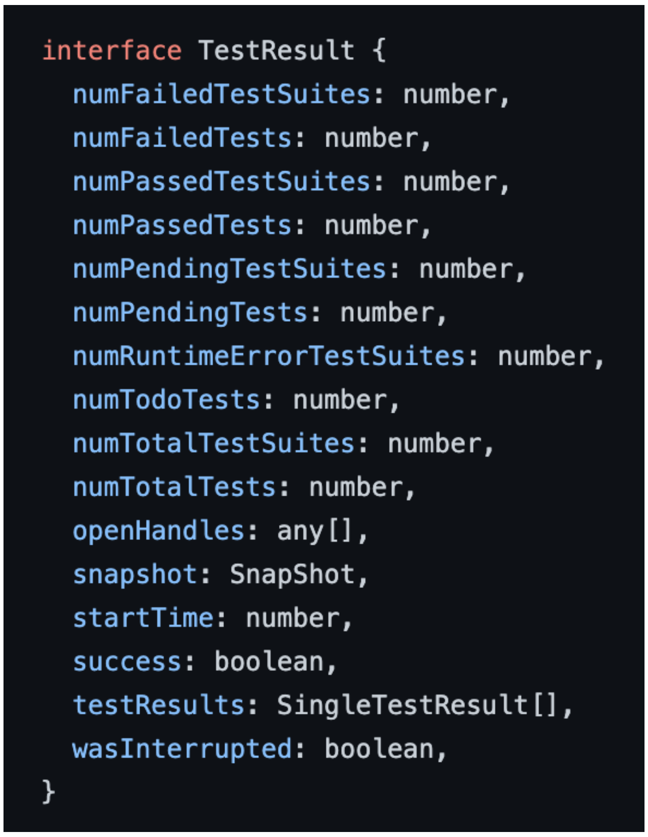
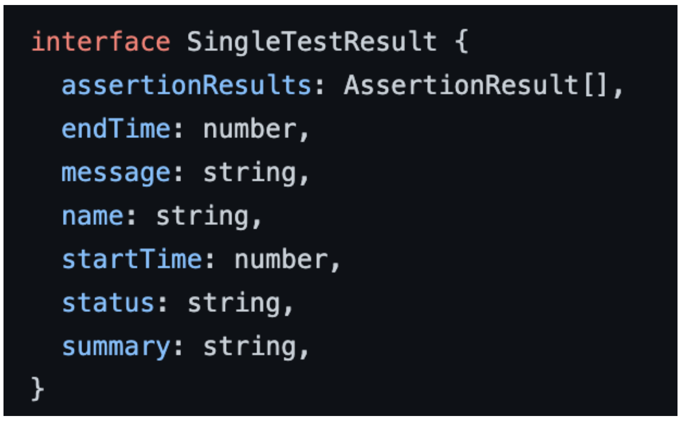


Figure 2: Interface of test result from Jest



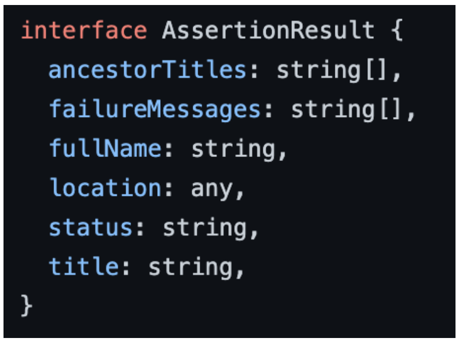
```
interface TestResult {
  numFailedTestSuites: number,
  numFailedTests: number,
  numPassedTestSuites: number,
  numPassedTests: number,
  numPendingTestSuites: number,
  numPendingTests: number,
  numRuntimeErrorTestSuites: number,
  numTodoTests: number,
  numTotalTestSuites: number,
  numTotalTests: number,
  openHandles: any[],
  snapshot: SnapShot,
  startTime: number,
  success: boolean,
  testResults: SingleTestResult[],
  wasInterrupted: boolean,
}
```

Figure 3: Interface of SingleTestResult



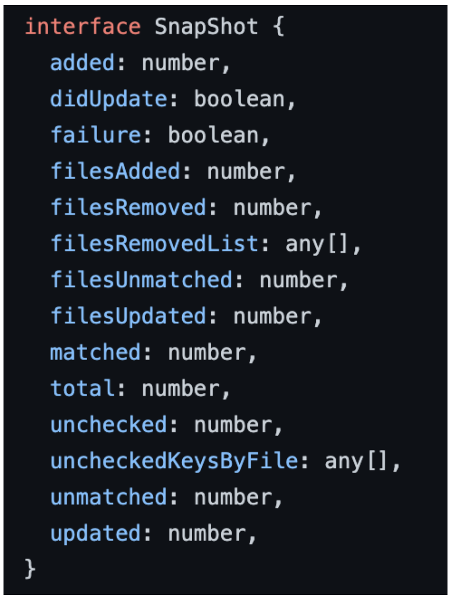
```
interface SingleTestResult {
  assertionResults: AssertionResult[],
  endTime: number,
  message: string,
  name: string,
  startTime: number,
  status: string,
  summary: string,
}
```

Figure 4: Interface of AssertionResult



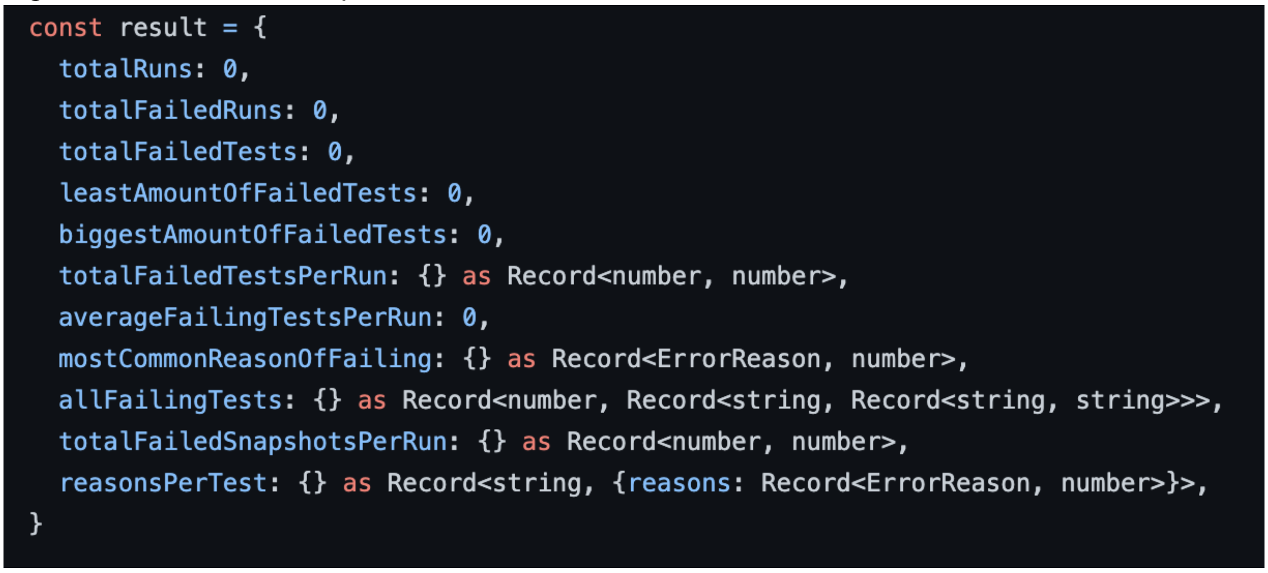
```
interface AssertionResult {
  ancestorTitles: string[],
  failureMessages: string[],
  fullName: string,
  location: any,
  status: string,
  title: string,
}
```

Figure 5: Interface of SnapShot



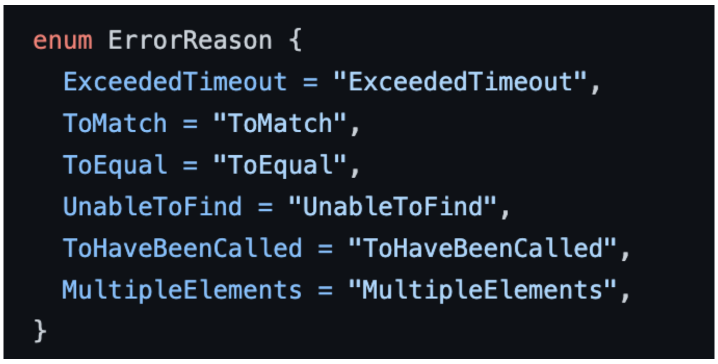
```
interface SnapShot {
  added: number,
  didUpdate: boolean,
  failure: boolean,
  filesAdded: number,
  filesRemoved: number,
  filesRemovedList: any[],
  filesUnmatched: number,
  filesUpdated: number,
  matched: number,
  total: number,
  unchecked: number,
  uncheckedKeysByFile: any[],
  unmatched: number,
  updated: number,
}
```

Figure 6: Interface analysis result



```
const result = {
  totalRuns: 0,
  totalFailedRuns: 0,
  totalFailedTests: 0,
  leastAmountOfFailedTests: 0,
  biggestAmountOfFailedTests: 0,
  totalFailedTestsPerRun: {} as Record<number, number>,
  averageFailingTestsPerRun: 0,
  mostCommonReasonOfFailing: {} as Record<ErrorReason, number>,
  allFailingTests: {} as Record<number, Record<string, Record<string, string>>>,
  totalFailedSnapshotsPerRun: {} as Record<number, number>,
  reasonsPerTest: {} as Record<string, {reasons: Record<ErrorReason, number>}>,
}
```

Figure 7: Enum for error reasons



```
enum ErrorReason {
  ExceededTimeout = "ExceededTimeout",
  ToMatch = "ToMatch",
  ToEqual = "ToEqual",
  UnableToFind = "UnableToFind",
  ToHaveBeenCalled = "ToHaveBeenCalled",
  MultipleElements = "MultipleElements",
}
```

Figure 8: Visualisation of test flaking at Spotify

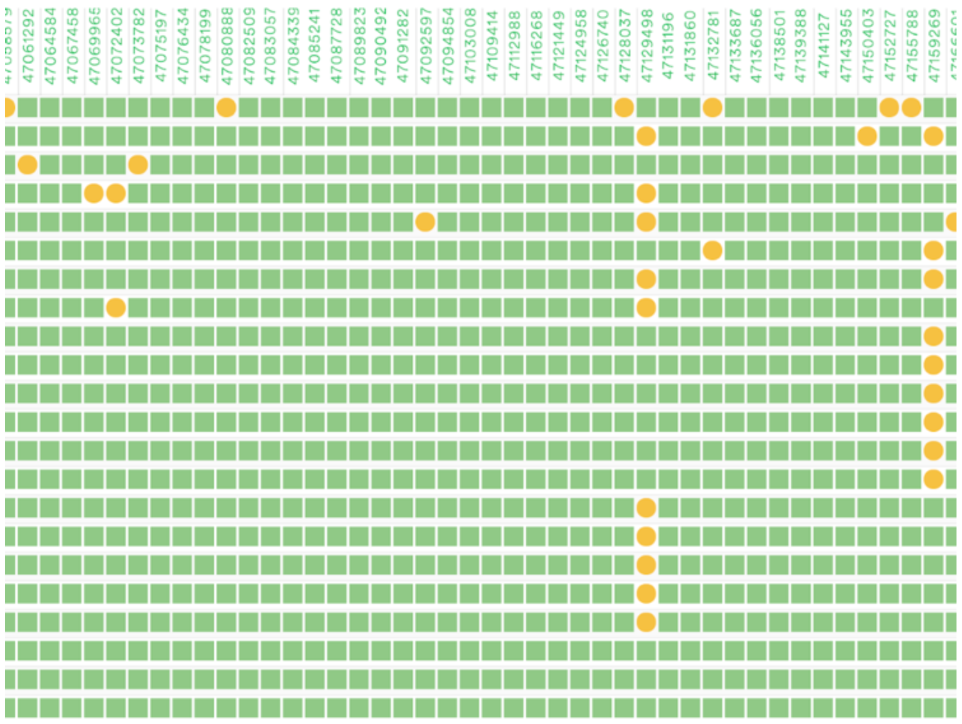


Figure 9: Dashboard of all the test runs from Microsoft Visual Studio

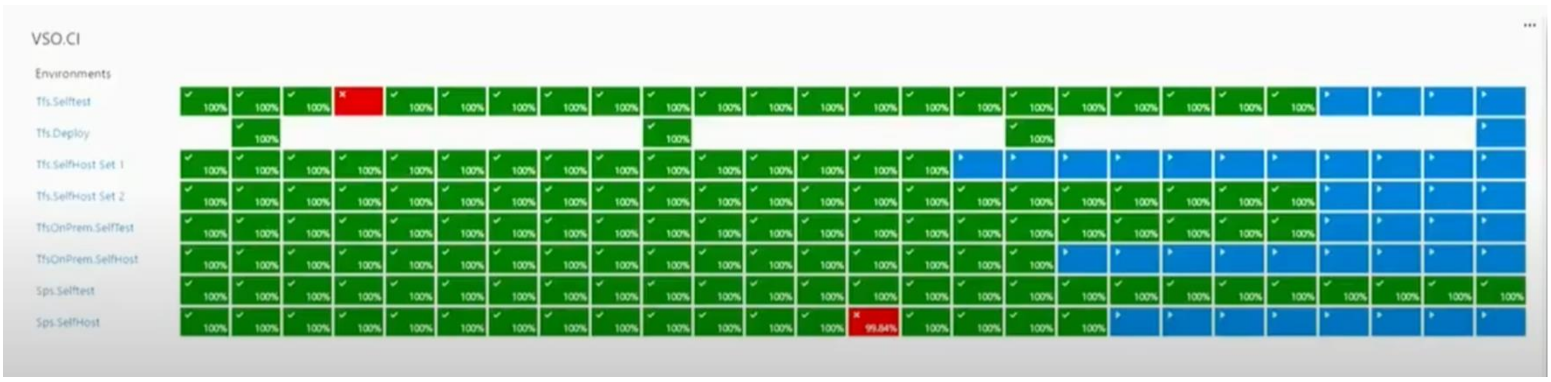
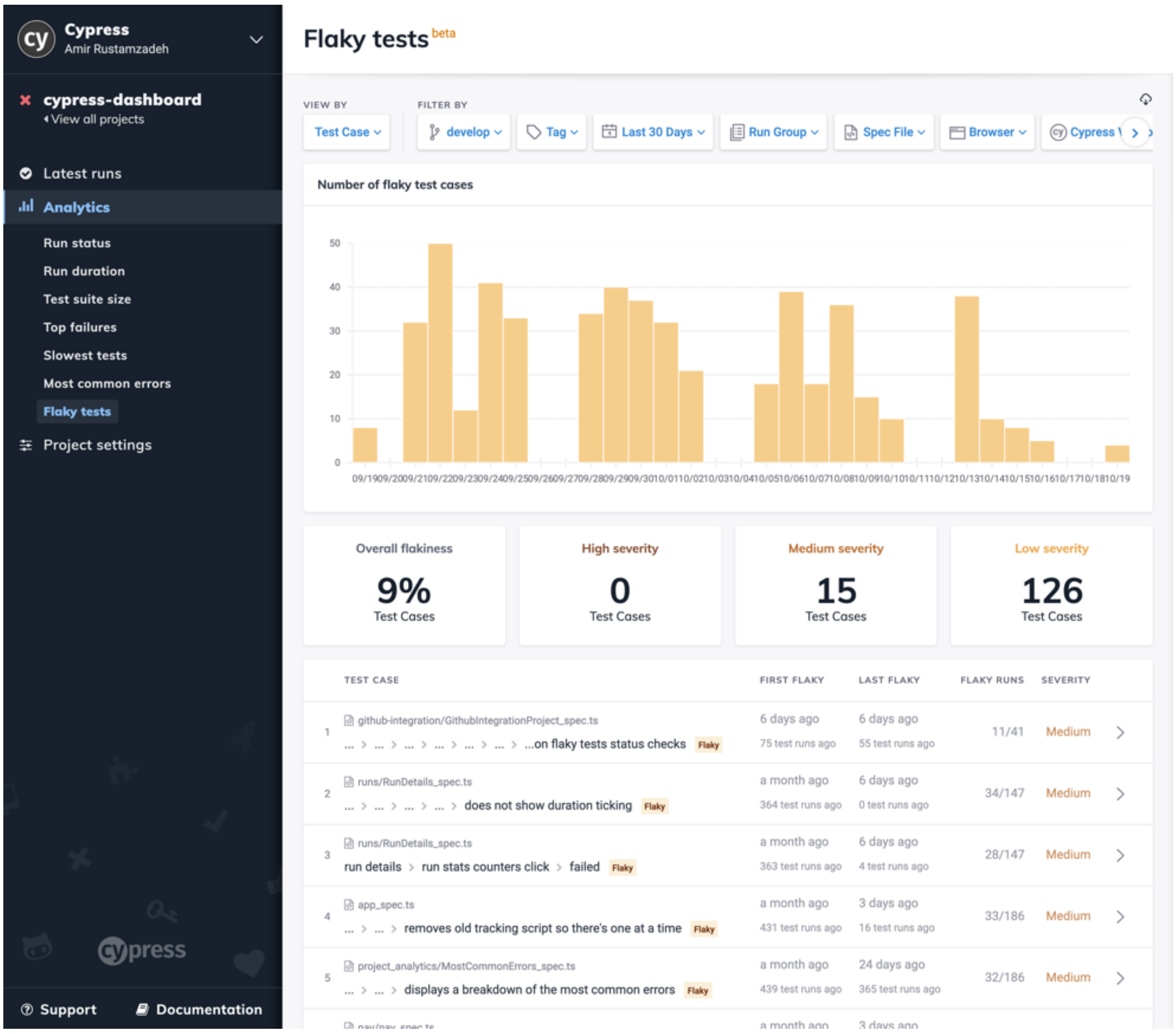


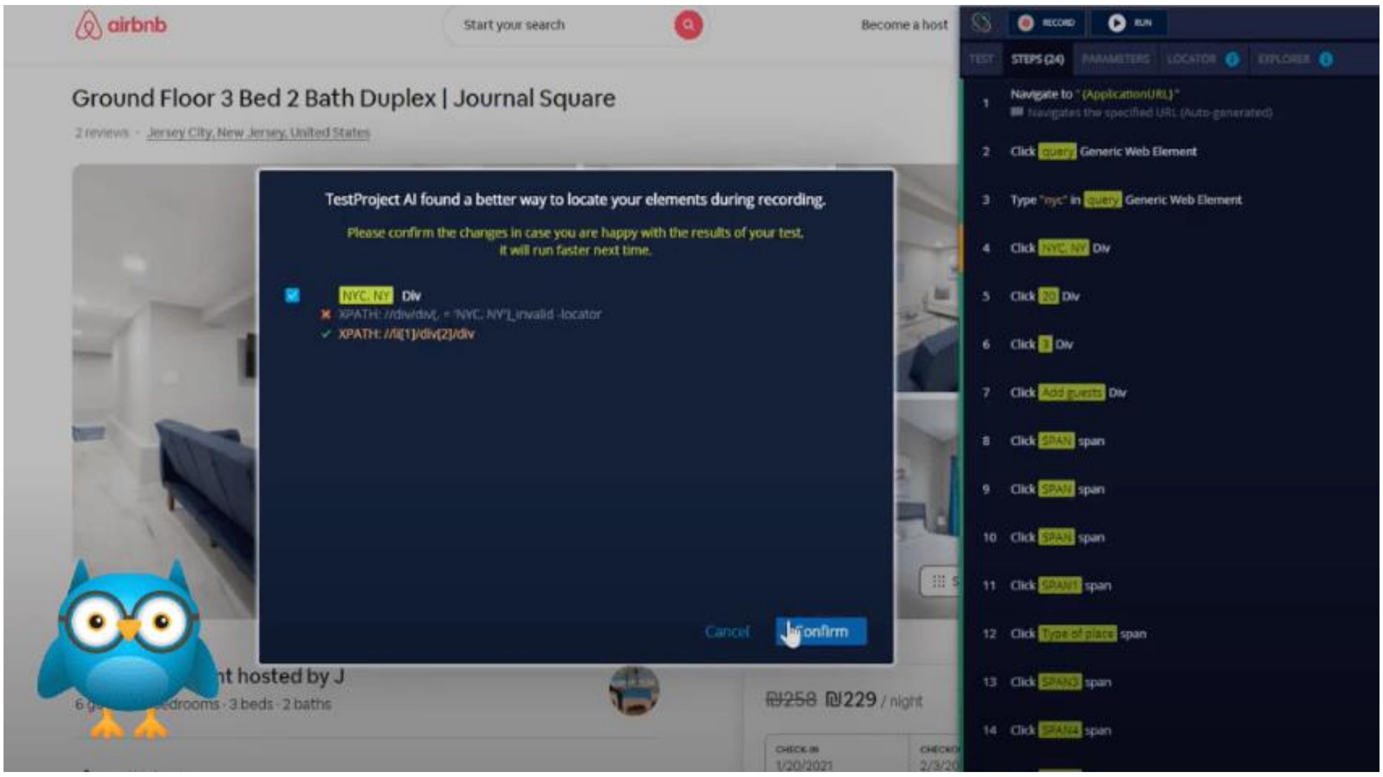
Figure 10: Cypress Flaking Test Management



Overall flakiness	High severity	Medium severity	Low severity
9% Test Cases	0 Test Cases	15 Test Cases	126 Test Cases

	TEST CASE	FIRST FLAKY	LAST FLAKY	FLAKY RUNS	SEVERITY
1	github-integration/GithubIntegrationProject_spec.ts ... > ... > ... > ... > ... > ... > ...on flaky tests status checks Flaky	6 days ago 75 test runs ago	6 days ago 55 test runs ago	11/41	Medium
2	runs/RunDetails_spec.ts ... > ... > ... > ... > does not show duration ticking Flaky	a month ago 364 test runs ago	6 days ago 0 test runs ago	34/147	Medium
3	runs/RunDetails_spec.ts run details > run stats counters click > failed Flaky	a month ago 363 test runs ago	6 days ago 4 test runs ago	28/147	Medium
4	app_spec.ts ... > ... > removes old tracking script so there's one at a time Flaky	a month ago 431 test runs ago	3 days ago 16 test runs ago	33/186	Medium
5	project_analytics/MostCommonErrors_spec.ts ... > ... > displays a breakdown of the most common errors Flaky	a month ago 439 test runs ago	24 days ago 365 test runs ago	32/186	Medium
	nav/nav_spec.ts	a month ago	3 days ago		

Figure 11: Example TestProject AI



Ground Floor 3 Bed 2 Bath Duplex | Journal Square

2 reviews · Jersey City, New Jersey, United States

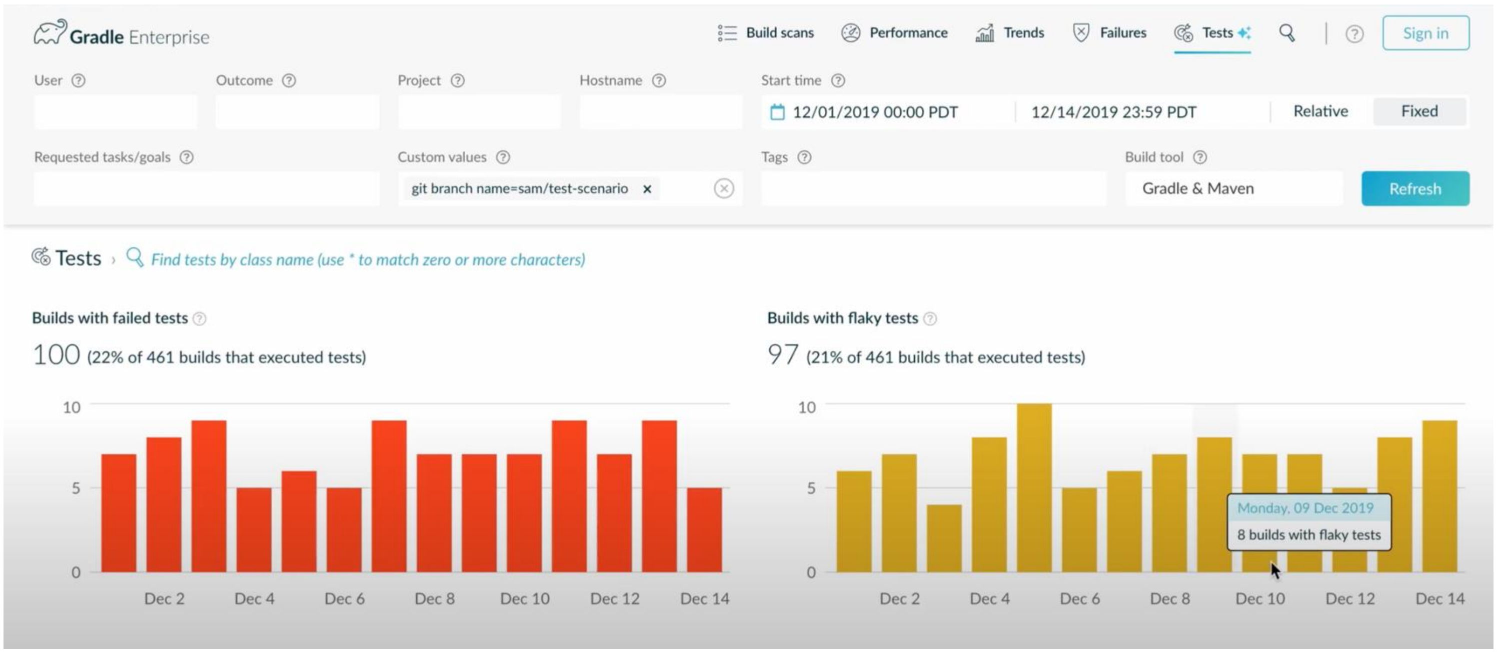
TestProject AI found a better way to locate your elements during recording.

Please confirm the changes in case you are happy with the results of your test.
It will run faster next time.

NYC, NY Div
XPATH: //div/div[, = 'NYC, NY'] invalid -locator
XPATH: //li[1]/div[2]/div

```
1   Navigate to "{ApplicationURL}"
    Navigates the specified URL (Auto-generated)
2   Click query Generic Web Element
3   Type "nyc" in query Generic Web Element
4   Click NYC, NY Div
5   Click 20 Div
6   Click 3 Div
7   Click Add guests Div
8   Click SPAN span
9   Click SPAN span
10  Click SPAN span
11  Click SPAN1 span
12  Click Type of place span
13  Click SPAN3 span
14  Click SPAN4 span
```

Figure 12: Gradle dashboard



Start time: 12/01/2019 00:00 PDT – 12/14/2019 23:59 PDT

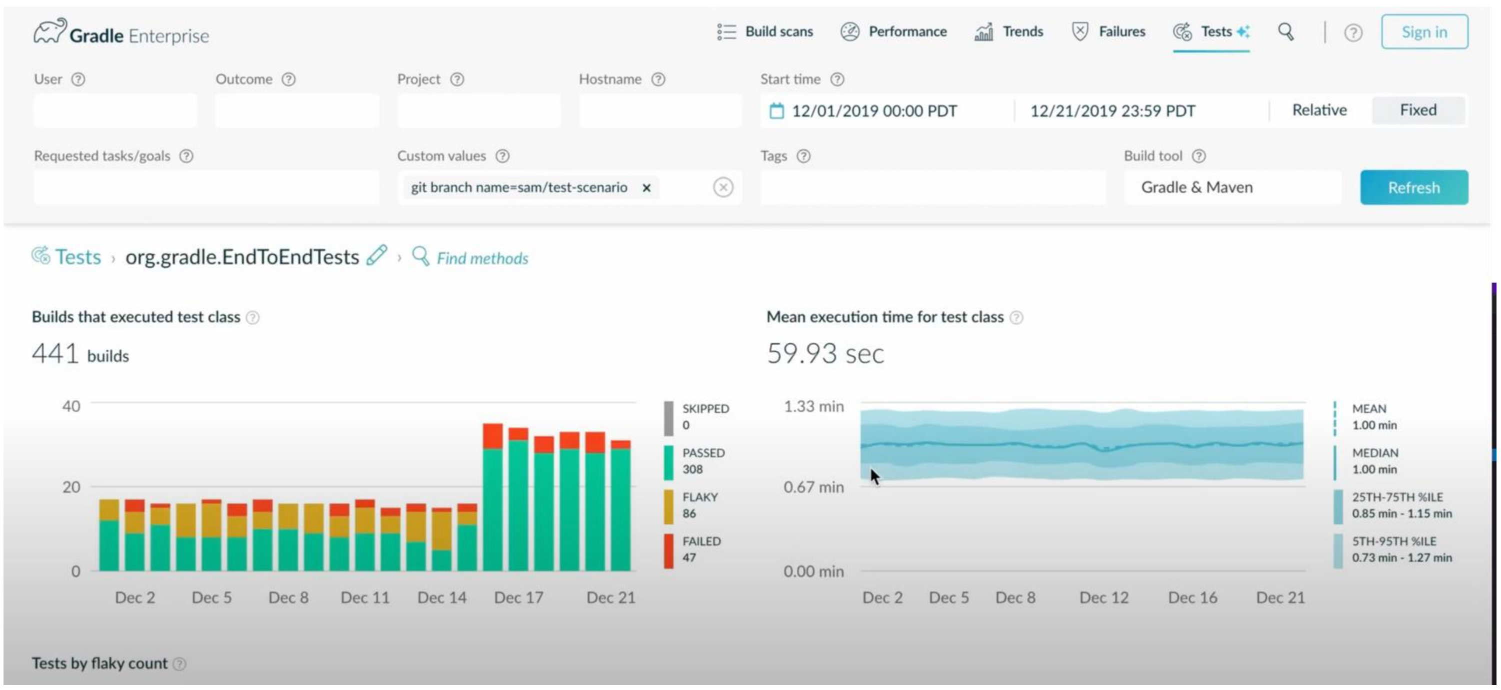
Custom values: git branch name=sam/test-scenario

Build tool: Gradle & Maven

Builds with failed tests
100 (22% of 461 builds that executed tests)

Builds with flaky tests
97 (21% of 461 builds that executed tests)

Monday, 09 Dec 2019
8 builds with flaky tests



Gradle Enterprise — Build scans · Performance · Trends · Failures · Tests · Sign in

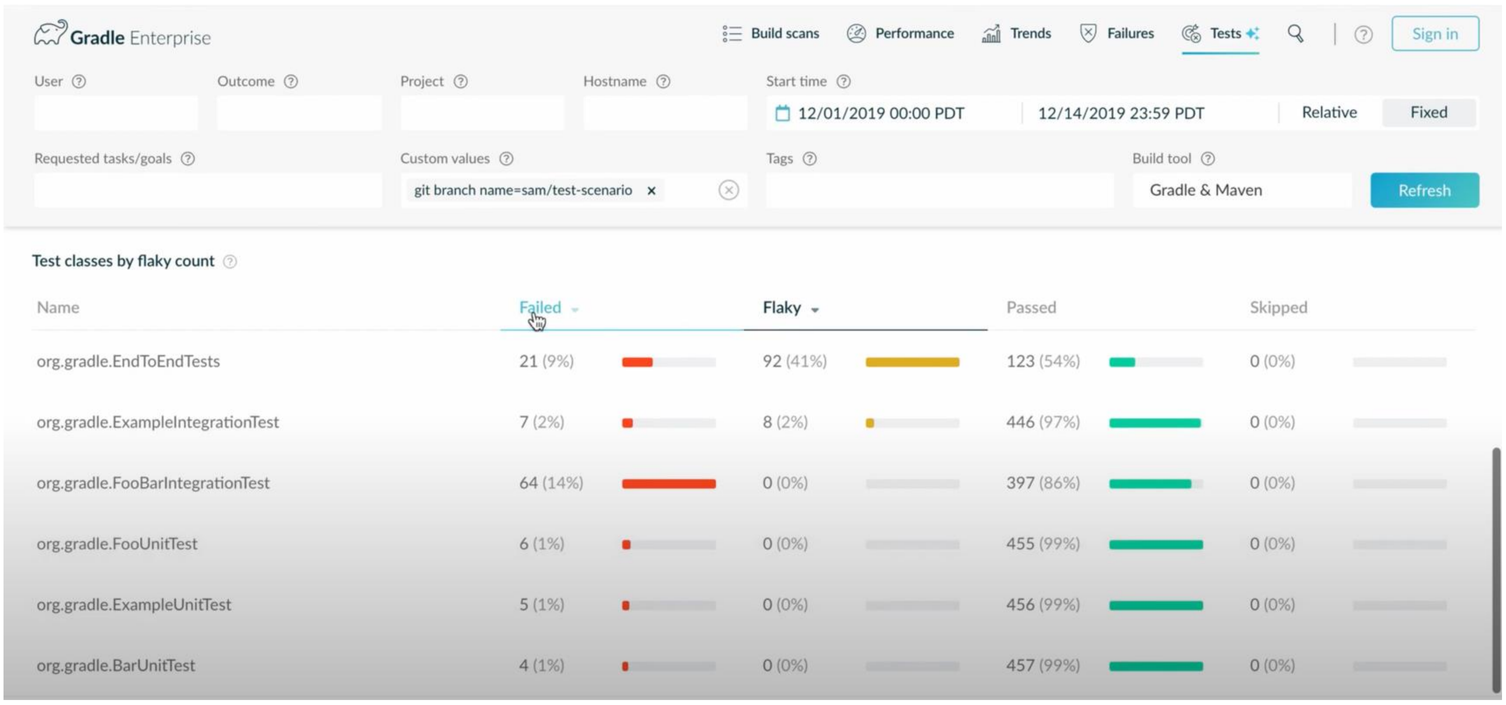
User	
Outcome	
Project	
Hostname	
Start time	12/01/2019 00:00 PDT – 12/21/2019 23:59 PDT (Relative / Fixed)
Requested tasks/goals	
Custom values	git branch name=sam/test-scenario
Tags	
Build tool	Gradle & Maven

Tests › org.gradle.EndToEndTests › *Find methods*

Builds that executed test class
441 builds

Mean execution time for test class
59.93 sec

Tests by flaky count



Gradle Enterprise — Build scans · Performance · Trends · Failures · Tests · Sign in

User	
Outcome	
Project	
Hostname	
Start time	12/01/2019 00:00 PDT – 12/14/2019 23:59 PDT (Relative / Fixed)
Requested tasks/goals	
Custom values	git branch name=sam/test-scenario
Tags	
Build tool	Gradle & Maven

Test classes by flaky count

Name	Failed	Flaky	Passed	Skipped
org.gradle.EndToEndTests	21 (9%)	92 (41%)	123 (54%)	0 (0%)
org.gradle.ExampleIntegrationTest	7 (2%)	8 (2%)	446 (97%)	0 (0%)
org.gradle.FooBarIntegrationTest	64 (14%)	0 (0%)	397 (86%)	0 (0%)
org.gradle.FooUnitTest	6 (1%)	0 (0%)	455 (99%)	0 (0%)
org.gradle.ExampleUnitTest	5 (1%)	0 (0%)	456 (99%)	0 (0%)
org.gradle.BarUnitTest	4 (1%)	0 (0%)	457 (99%)	0 (0%)

Figure 13: Wireframe project dashboard

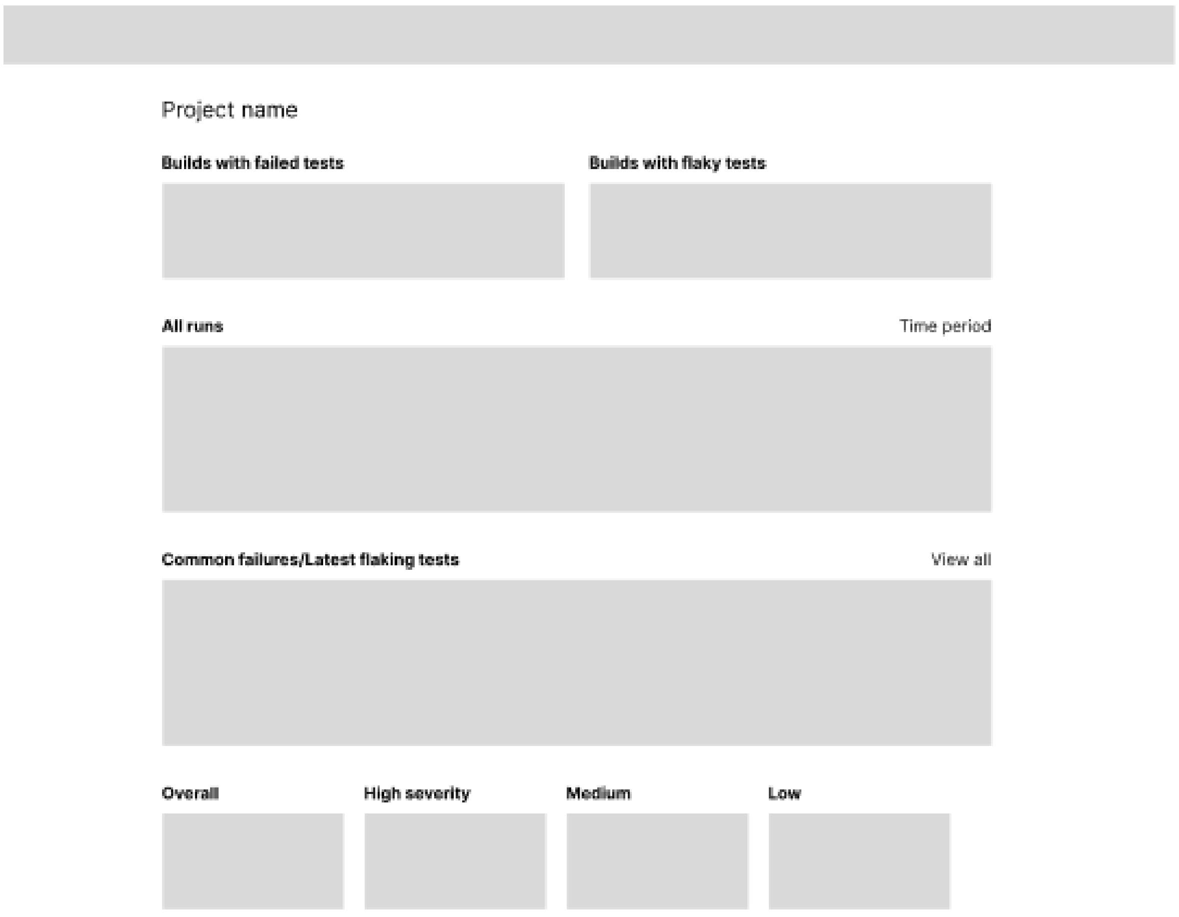
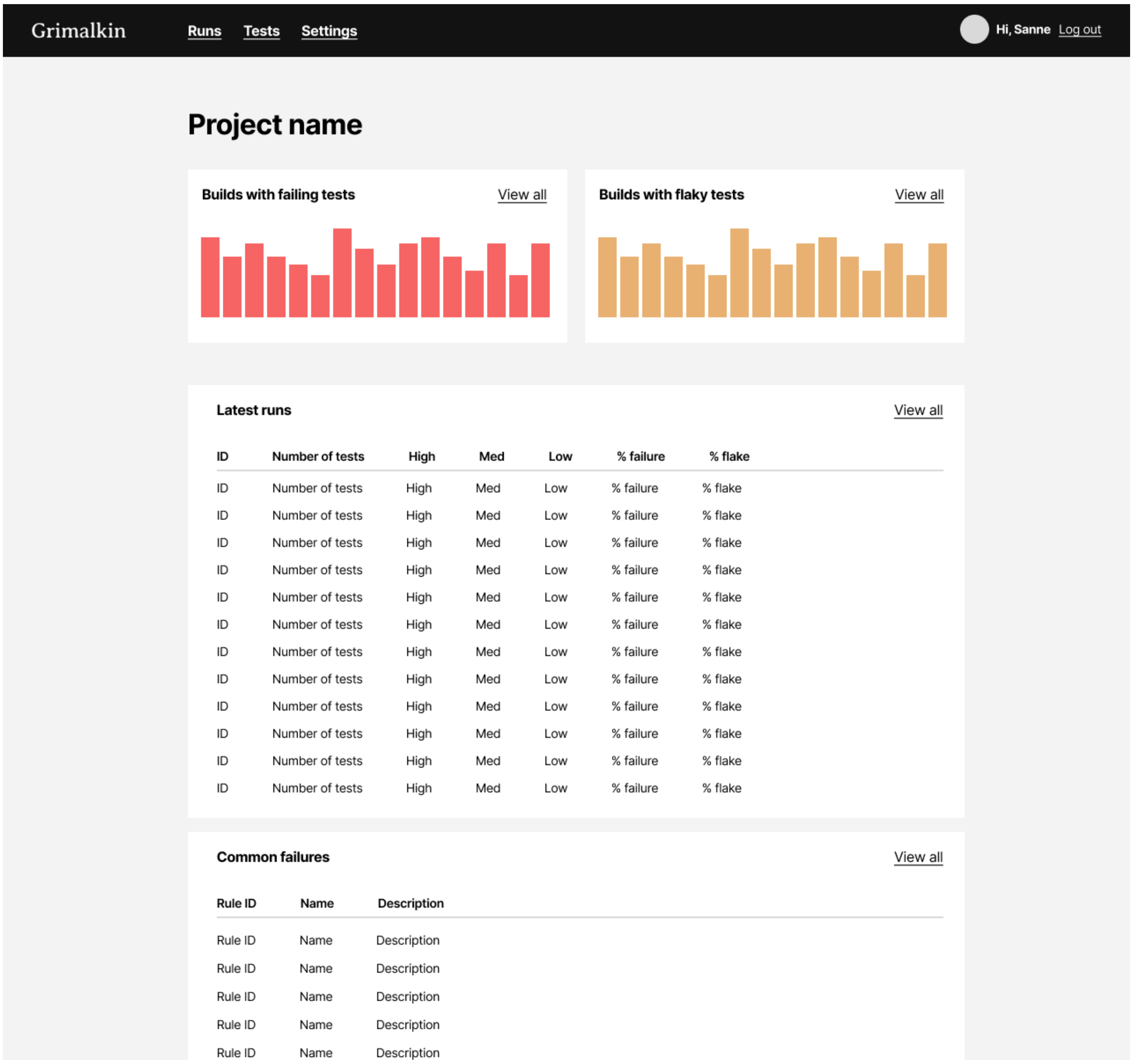


Figure 14: Design project dashboard



Project name

Builds with failing tests — View all

Builds with flaky tests — View all

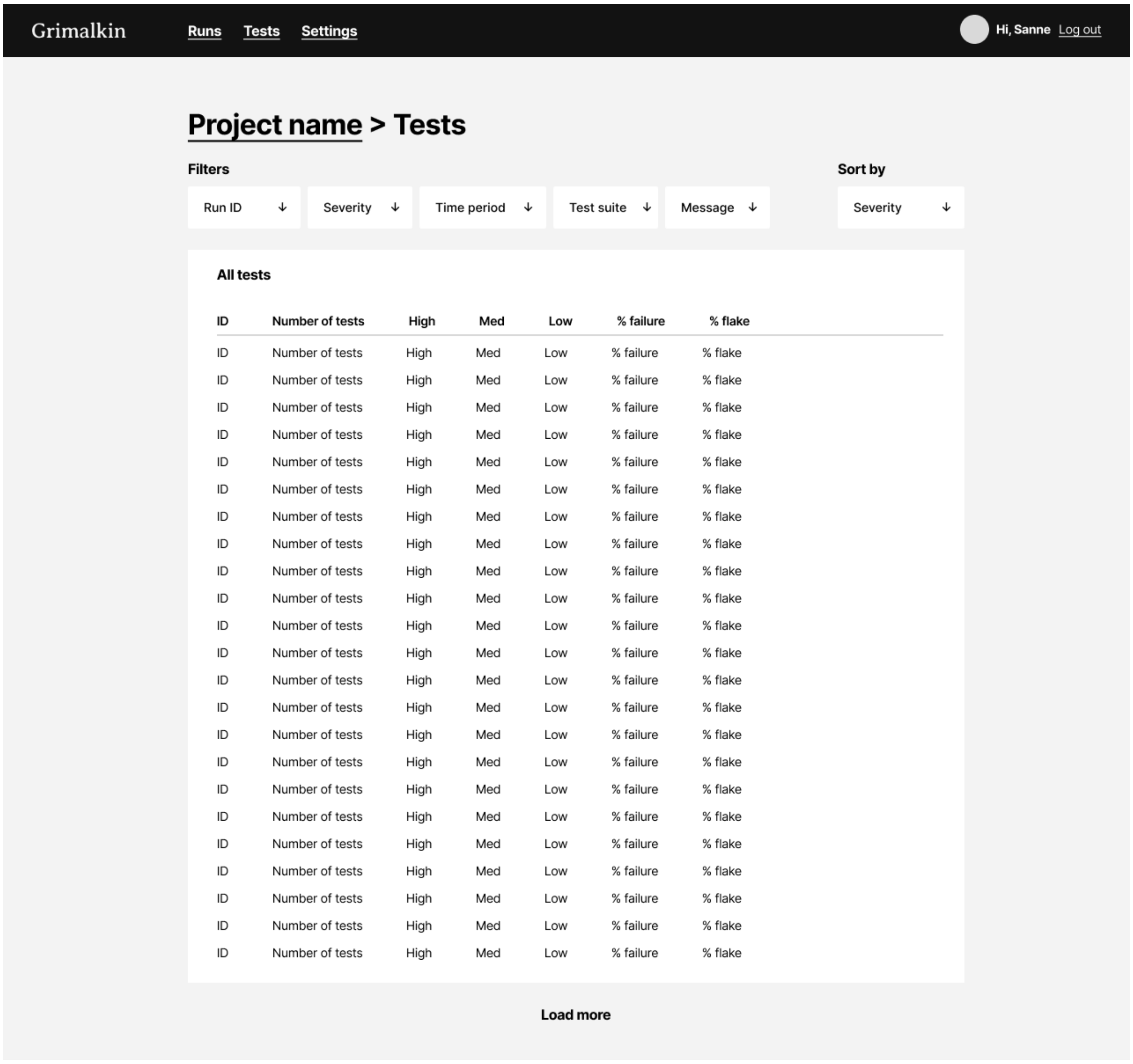
Latest runs — View all

ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake

Common failures — View all

Rule ID	Name	Description
Rule ID	Name	Description
Rule ID	Name	Description
Rule ID	Name	Description
Rule ID	Name	Description
Rule ID	Name	Description

Figure 15: Design flaking test overview



Grimalkin Runs Tests Settings Hi, Sanne Log out

Project name > Tests

Filters: Run ID ↓ | Severity ↓ | Time period ↓ | Test suite ↓ | Message ↓

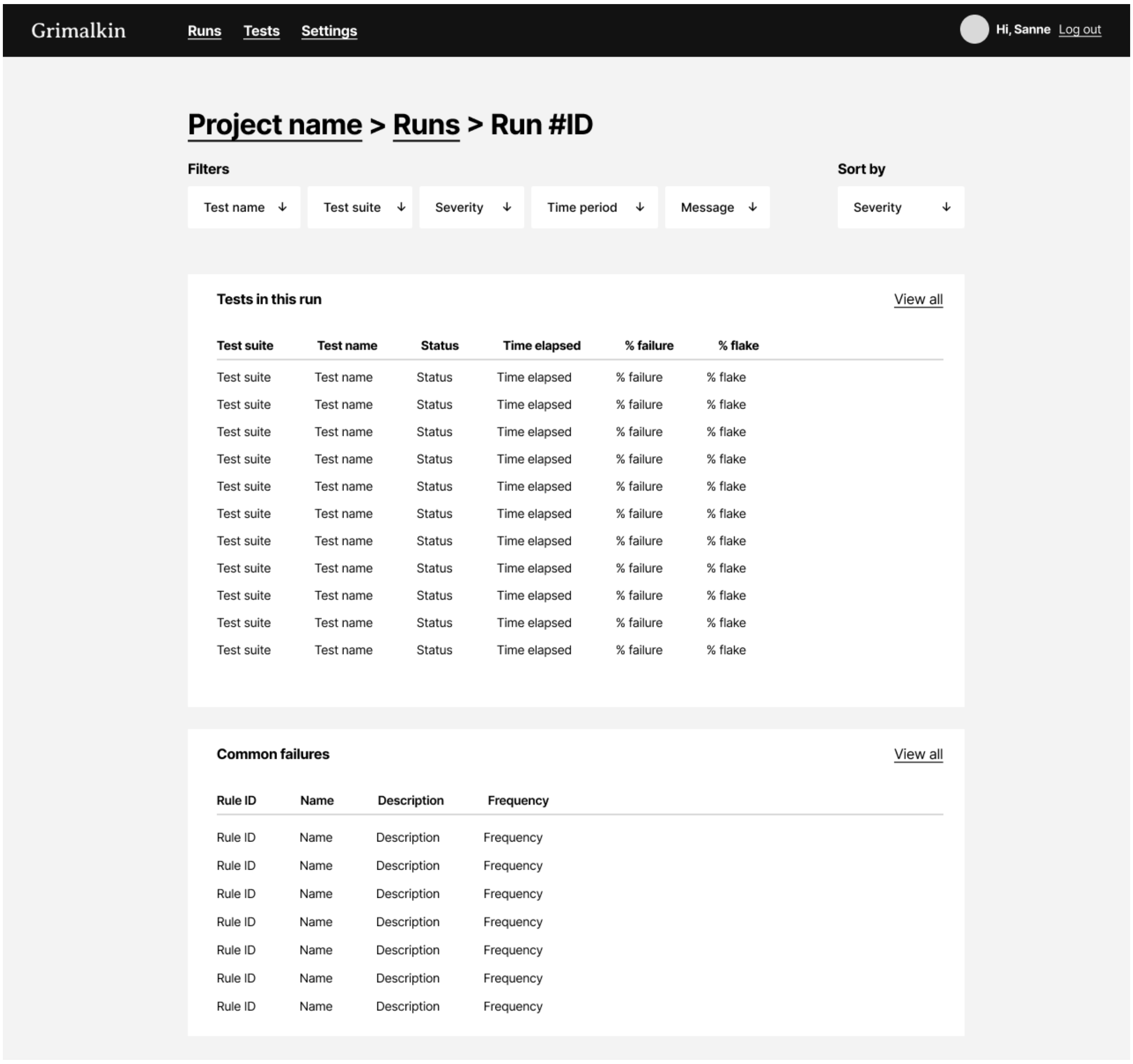
Sort by: Severity ↓

All tests

ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake

Load more

Figure 16: Design run dashboard



Project name > Runs > Run #ID

Filters: Test name ↓ | Test suite ↓ | Severity ↓ | Time period ↓ | Message ↓

Sort by: Severity ↓

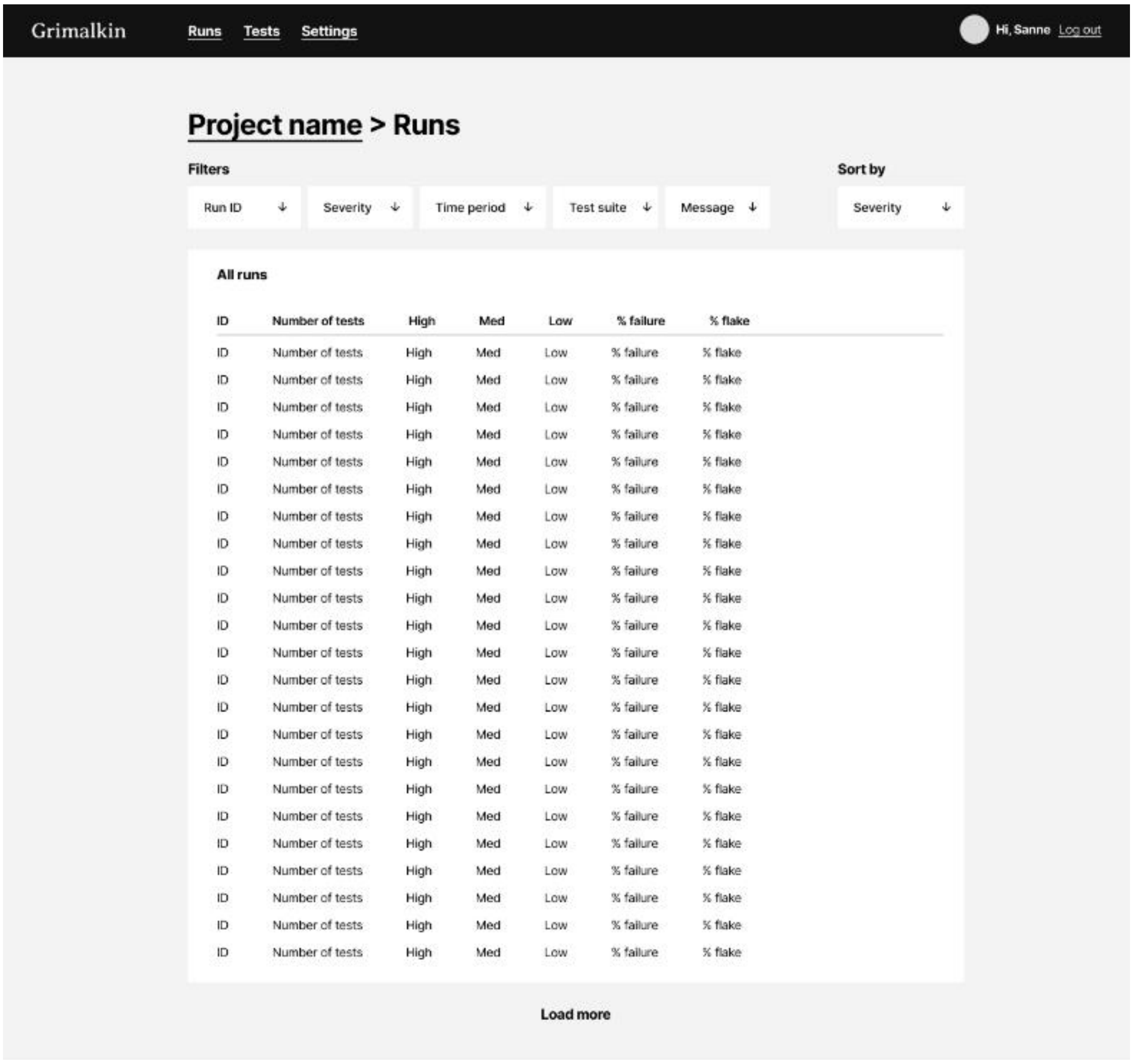
Tests in this run View all

Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake
Test suite	Test name	Status	Time elapsed	% failure	% flake

Common failures View all

Rule ID	Name	Description	Frequency
Rule ID	Name	Description	Frequency
Rule ID	Name	Description	Frequency
Rule ID	Name	Description	Frequency
Rule ID	Name	Description	Frequency
Rule ID	Name	Description	Frequency
Rule ID	Name	Description	Frequency
Rule ID	Name	Description	Frequency

Figure 17: Design overview of runs

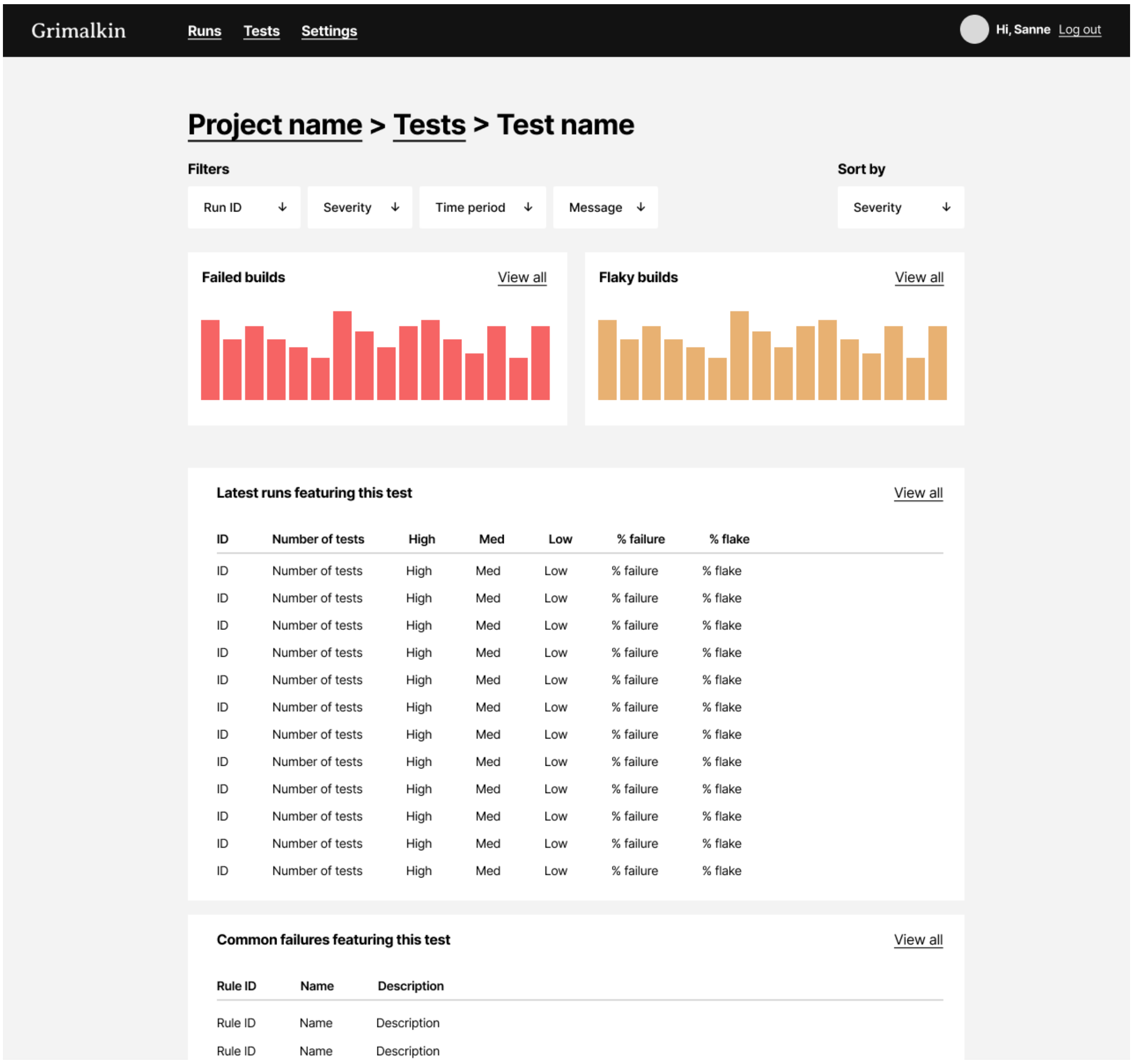


Project name > Runs

All runs

ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake

Figure 18: Design test dashboard



Project name > Tests > Test name

Filters

Run ID ↓ | Severity ↓ | Time period ↓ | Message ↓

Sort by

Severity ↓

Latest runs featuring this test View all

ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake
ID	Number of tests	High	Med	Low	% failure	% flake

Common failures featuring this test View all

Rule ID	Name	Description
Rule ID	Name	Description
Rule ID	Name	Description

Figure 19: NoSQL Data Model

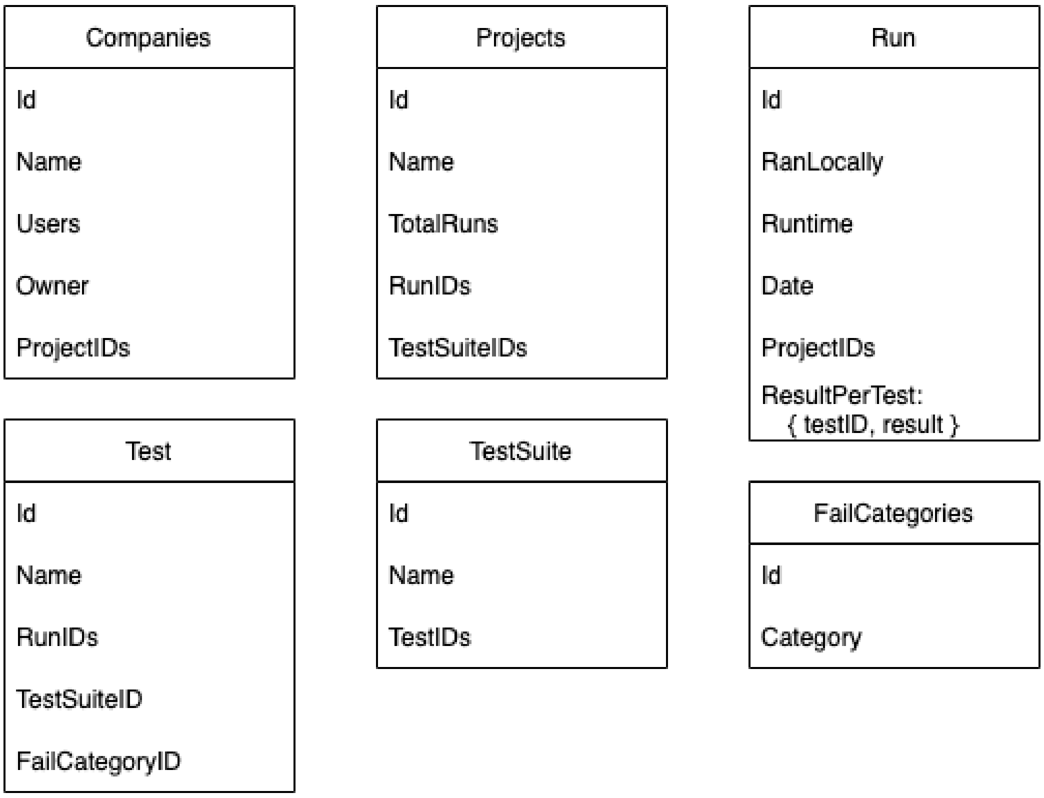


Figure 20: SQL Data Model

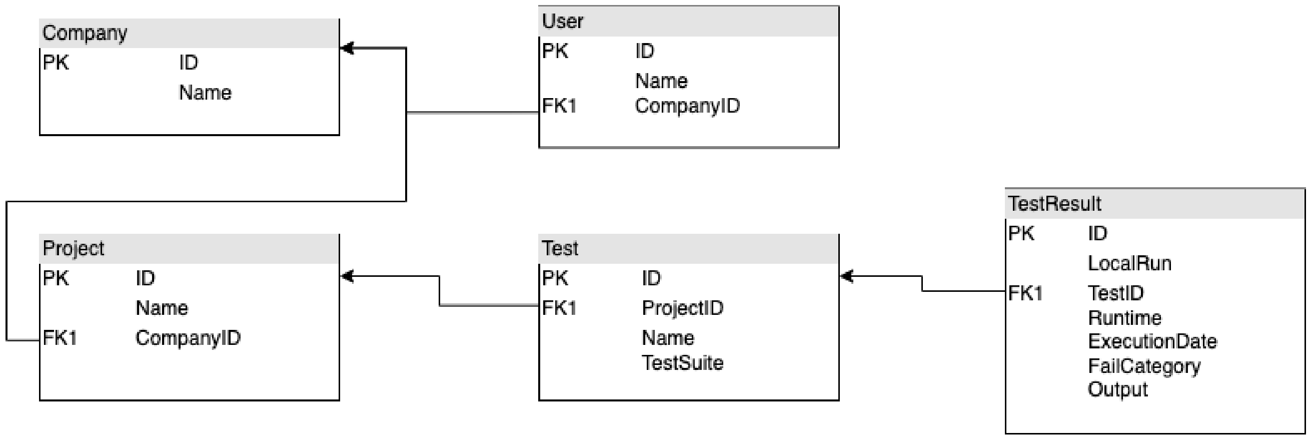


Figure 21: Example of a Jest Watch plugin

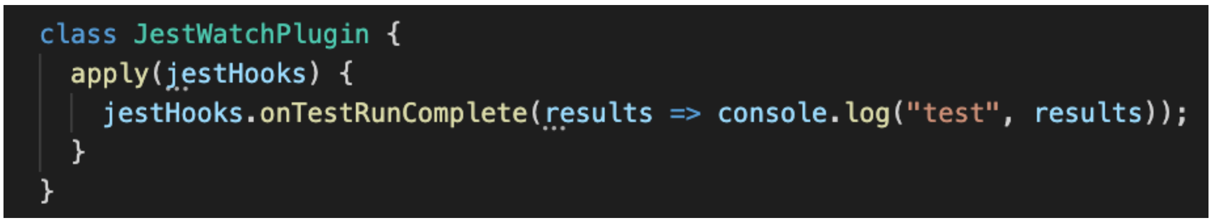
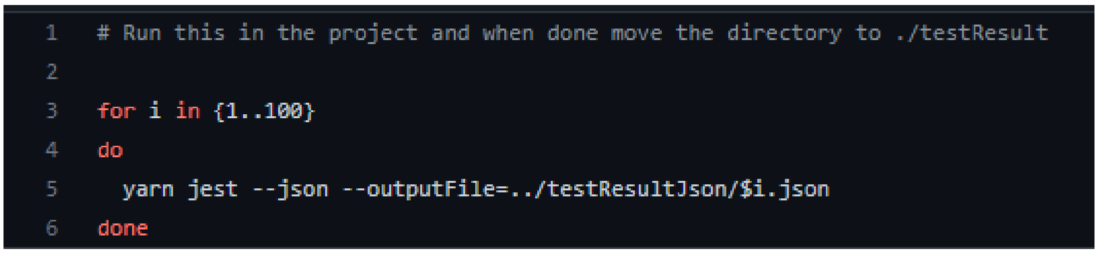

```
class JestWatchPlugin {
  apply(jestHooks) {
    jestHooks.onTestRunComplete(results => console.log("test", results));
  }
}
```

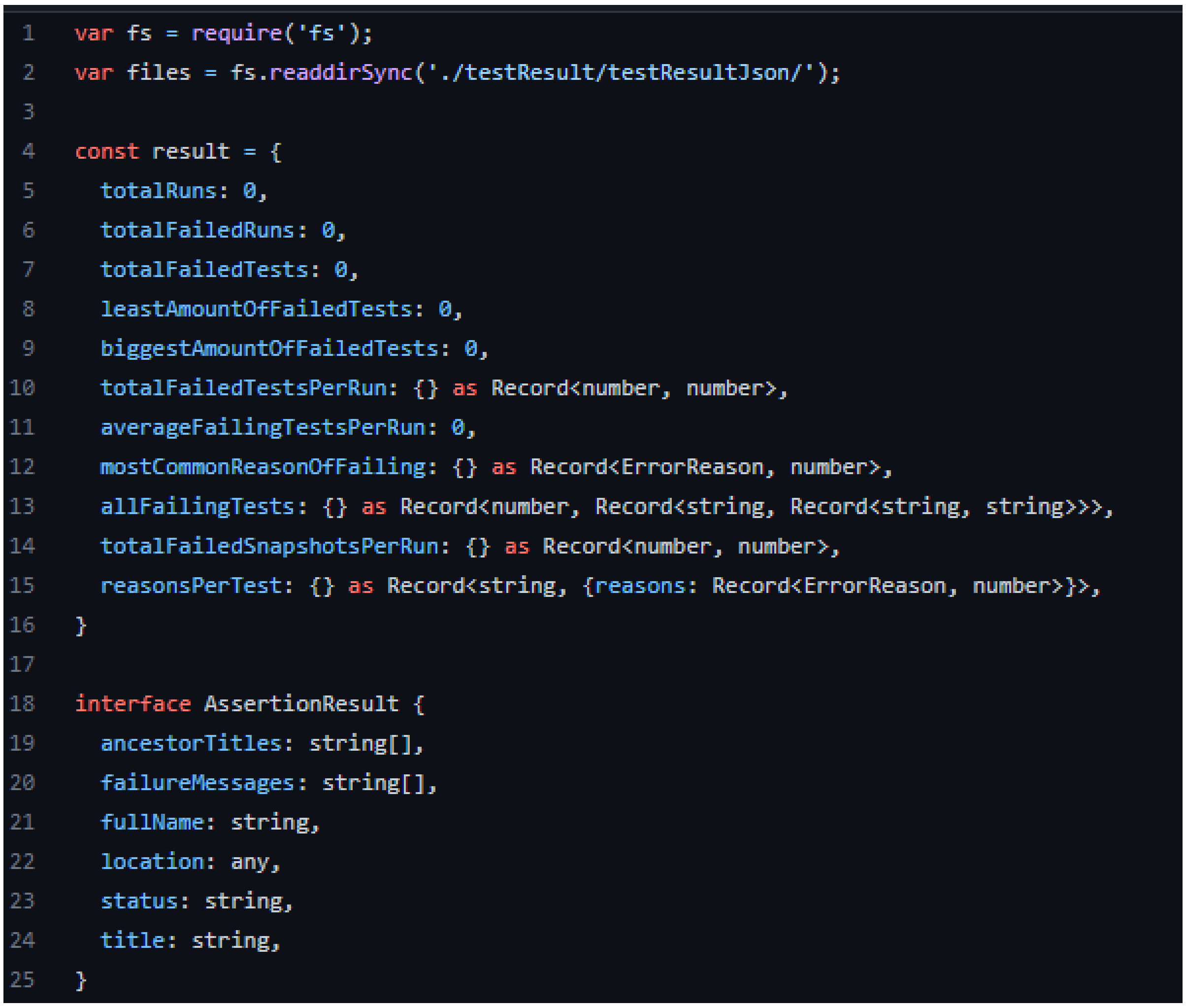
Figure 22: Test result type from Jest

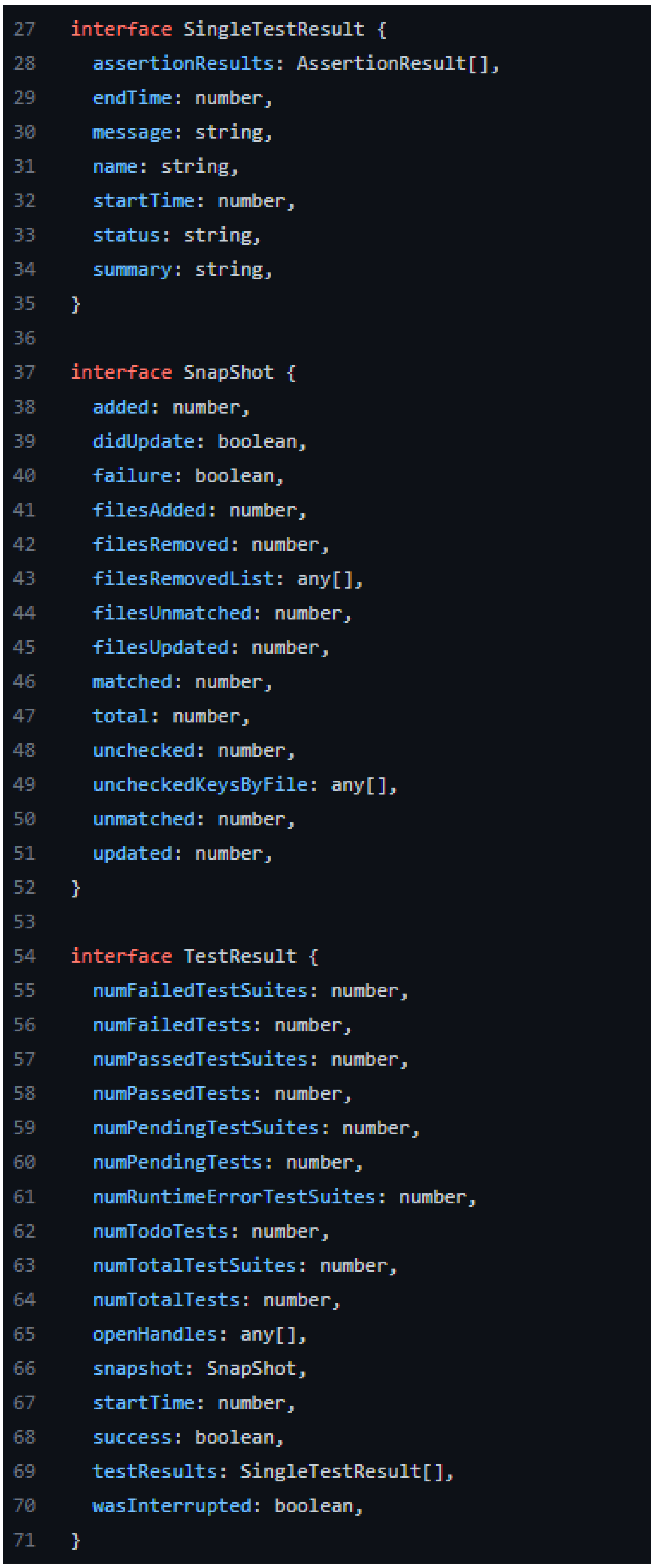
```
{
  "success": boolean,
  "startTime": epoch,
  "numTotalTestSuites": number,
  "numPassedTestSuites": number,
  "numFailedTestSuites": number,
  "numRuntimeErrorTestSuites": number,
  "numTotalTests": number,
  "numPassedTests": number,
  "numFailedTests": number,
  "numPendingTests": number,
  "numTodoTests": number,
  "openHandles": Array<Error>,
  "testResults": [{
    "numFailingTests": number,
    "numPassingTests": number,
    "numPendingTests": number,
    "testResults": [{
      "title": string (message in it block),
      "status": "failed" | "pending" | "passed",
      "ancestorTitles": [string (message in describe blocks)],
      "failureMessages": [string],
      "numPassingAsserts": number,
      "location": {
        "column": number,
        "line": number
      },
      "duration": number | null
    },
    ...
    ],
    "perfStats": {
      "start": epoch,
      "end": epoch
    },
    "testFilePath": absolute path to test file,
    "coverage": {}
  },
  "testExecError:" (exists if there was a top-level failure) {
    "message": string
    "stack": string
  }
  ...
  ]
}
```

17 Appendix A: Get Test Results

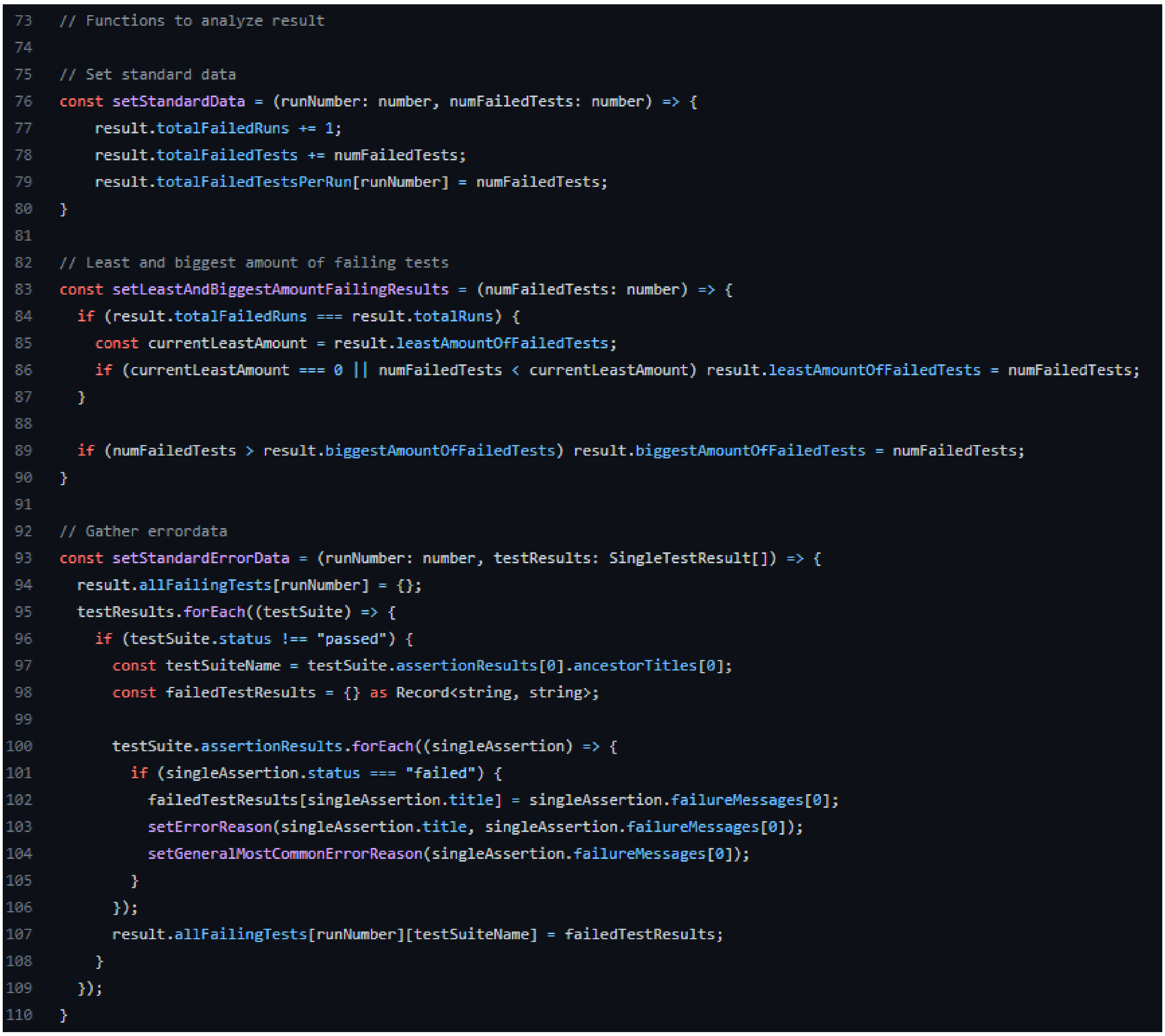

```
# Run this in the project and when done move the directory to ./testResult

for i in {1..100}
do
  yarn jest --json --outputFile=../testResultJson/$i.json
done
```

18 Appendix B: Test Analysis Script


```
var fs = require('fs');
var files = fs.readdirSync('./testResult/testResultJson/');

const result = {
  totalRuns: 0,
  totalFailedRuns: 0,
  totalFailedTests: 0,
  leastAmountOfFailedTests: 0,
  biggestAmountOfFailedTests: 0,
  totalFailedTestsPerRun: {} as Record<number, number>,
  averageFailingTestsPerRun: 0,
  mostCommonReasonOfFailing: {} as Record<ErrorReason, number>,
  allFailingTests: {} as Record<number, Record<string, Record<string, string>>>,
  totalFailedSnapshotsPerRun: {} as Record<number, number>,
  reasonsPerTest: {} as Record<string, {reasons: Record<ErrorReason, number>}>,
}

interface AssertionResult {
  ancestorTitles: string[],
  failureMessages: string[],
  fullName: string,
  location: any,
  status: string,
  title: string,
}
```

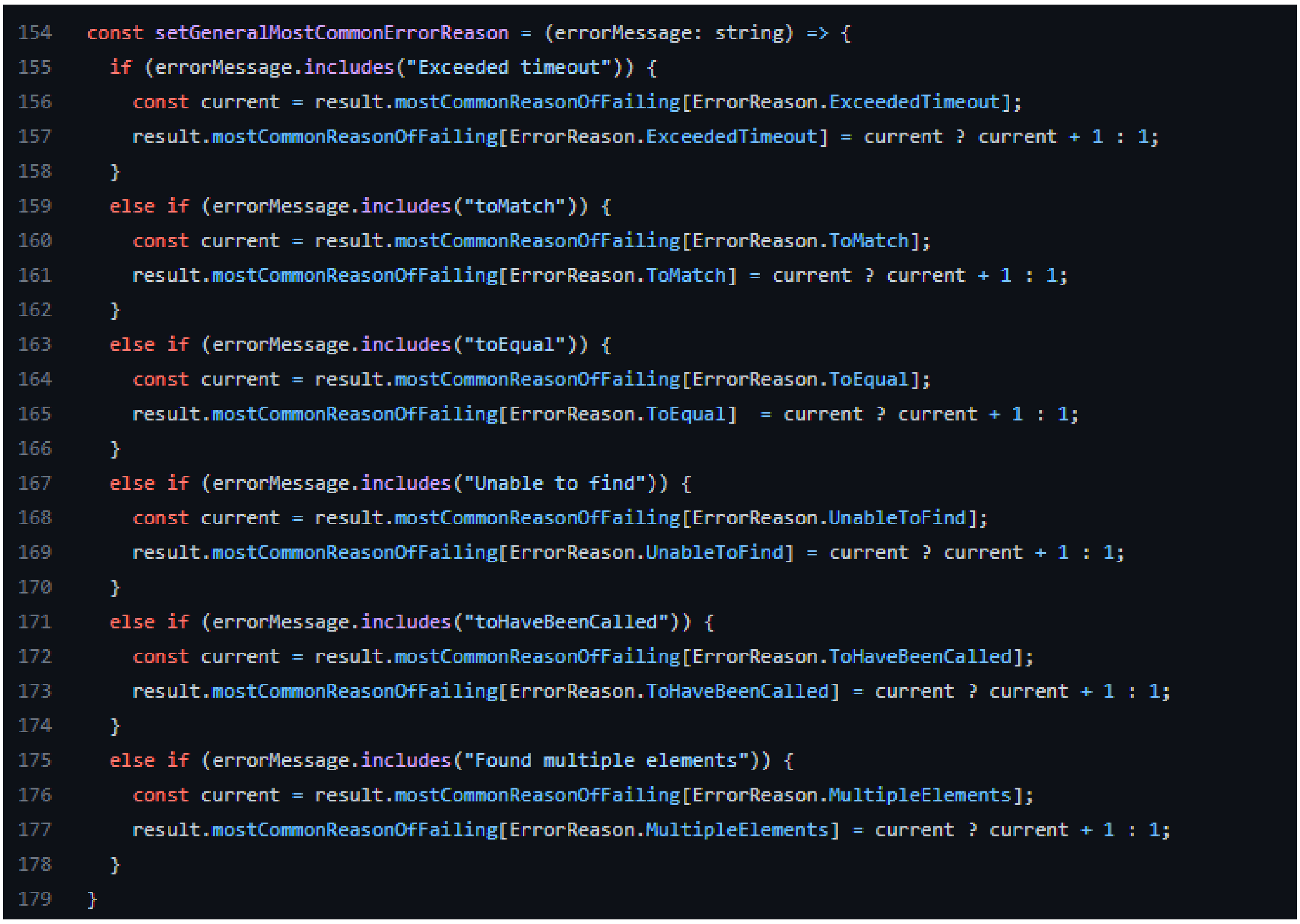


```
interface SingleTestResult {
  assertionResults: AssertionResult[],
  endTime: number,
  message: string,
  name: string,
  startTime: number,
  status: string,
  summary: string,
}

interface SnapShot {
  added: number,
  didUpdate: boolean,
  failure: boolean,
  filesAdded: number,
  filesRemoved: number,
  filesRemovedList: any[],
  filesUnmatched: number,
  filesUpdated: number,
  matched: number,
  total: number,
  unchecked: number,
  uncheckedKeysByFile: any[],
  unmatched: number,
  updated: number,
}

interface TestResult {
  numFailedTestSuites: number,
  numFailedTests: number,
  numPassedTestSuites: number,
  numPassedTests: number,
  numPendingTestSuites: number,
  numPendingTests: number,
  numRuntimeErrorTestSuites: number,
  numTodoTests: number,
  numTotalTestSuites: number,
  numTotalTests: number,
  openHandles: any[],
  snapshot: SnapShot,
  startTime: number,
  success: boolean,
  testResults: SingleTestResult[],
  wasInterrupted: boolean,
}
```

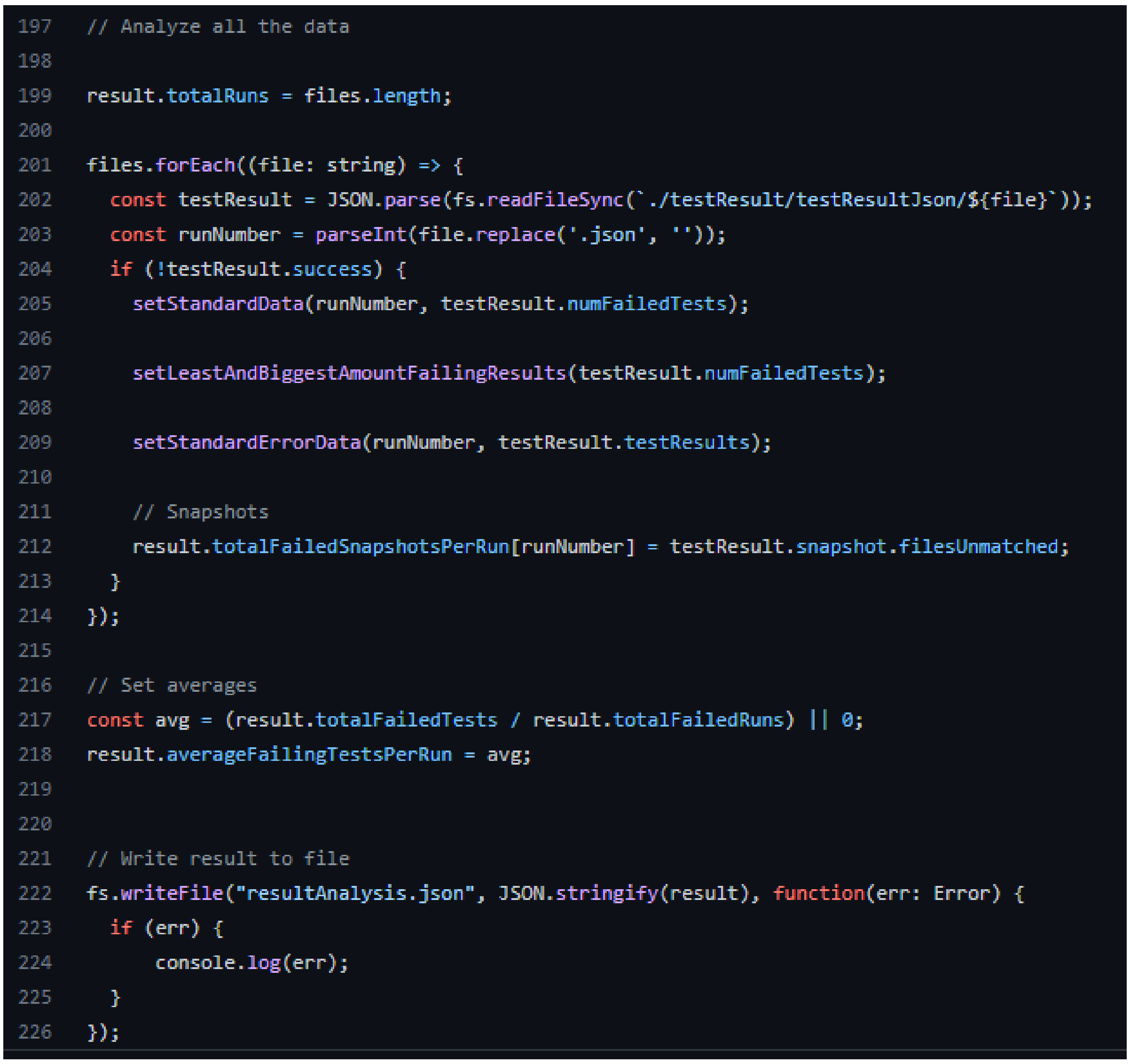


```
// Functions to analyze result

// Set standard data
const setStandardData = (runNumber: number, numFailedTests: number) => {
    result.totalFailedRuns += 1;
    result.totalFailedTests += numFailedTests;
    result.totalFailedTestsPerRun[runNumber] = numFailedTests;
}

// Least and biggest amount of failing tests
const setLeastAndBiggestAmountFailingResults = (numFailedTests: number) => {
  if (result.totalFailedRuns === result.totalRuns) {
    const currentLeastAmount = result.leastAmountOfFailedTests;
    if (currentLeastAmount === 0 || numFailedTests < currentLeastAmount) result.leastAmountOfFailedTests = numFailedTests;
  }

  if (numFailedTests > result.biggestAmountOfFailedTests) result.biggestAmountOfFailedTests = numFailedTests;
}

// Gather errordata
const setStandardErrorData = (runNumber: number, testResults: SingleTestResult[]) => {
  result.allFailingTests[runNumber] = {};
  testResults.forEach((testSuite) => {
    if (testSuite.status !== "passed") {
      const testSuiteName = testSuite.assertionResults[0].ancestorTitles[0];
      const failedTestResults = {} as Record<string, string>;

      testSuite.assertionResults.forEach((singleAssertion) => {
        if (singleAssertion.status === "failed") {
          failedTestResults[singleAssertion.title] = singleAssertion.failureMessages[0];
          setErrorReason(singleAssertion.title, singleAssertion.failureMessages[0]);
          setGeneralMostCommonErrorReason(singleAssertion.failureMessages[0]);
        }
      });
      result.allFailingTests[runNumber][testSuiteName] = failedTestResults;
    }
  });
}
```

```
enum ErrorReason {
  ExceededTimeout = "ExceededTimeout",
  ToMatch = "ToMatch",
  ToEqual = "ToEqual",
  UnableToFind = "UnableToFind",
  ToHaveBeenCalled = "ToHaveBeenCalled",
  MultipleElements = "MultipleElements",
}


const setErrorReason = (title: string, errorMessage: string) => {
  if (!result.reasonsPerTest[title]) {
    result.reasonsPerTest[title] = {} as {reasons: Record<ErrorReason, number>};
    result.reasonsPerTest[title].reasons = {} as Record<ErrorReason, number>;
  }

  if (errorMessage.includes("Exceeded timeout")) {
    const current = result.reasonsPerTest[title].reasons[ErrorReason.ExceededTimeout];
    result.reasonsPerTest[title].reasons[ErrorReason.ExceededTimeout] = current ? current + 1 : 1;
  }
  else if (errorMessage.includes("toMatch")) {
    const current = result.reasonsPerTest[title].reasons[ErrorReason.ToMatch];
    result.reasonsPerTest[title].reasons[ErrorReason.ToMatch] = current ? current + 1 : 1;
  }
  else if (errorMessage.includes("toEqual")) {
    const current = result.reasonsPerTest[title].reasons[ErrorReason.ToEqual];
    result.reasonsPerTest[title].reasons[ErrorReason.ToEqual]  = current ? current + 1 : 1;
  }
  else if (errorMessage.includes("Unable to find")) {
    const current = result.reasonsPerTest[title].reasons[ErrorReason.UnableToFind];
    result.reasonsPerTest[title].reasons[ErrorReason.UnableToFind] = current ? current + 1 : 1;
  }
  else if (errorMessage.includes("toHaveBeenCalled")) {
    const current = result.reasonsPerTest[title].reasons[ErrorReason.ToHaveBeenCalled];
    result.reasonsPerTest[title].reasons[ErrorReason.ToHaveBeenCalled] = current ? current + 1 : 1;
  }
  else if (errorMessage.includes("Found multiple elements")) {
    const current = result.reasonsPerTest[title].reasons[ErrorReason.MultipleElements];
    result.reasonsPerTest[title].reasons[ErrorReason.MultipleElements] = current ? current + 1 : 1;
  }
}
```

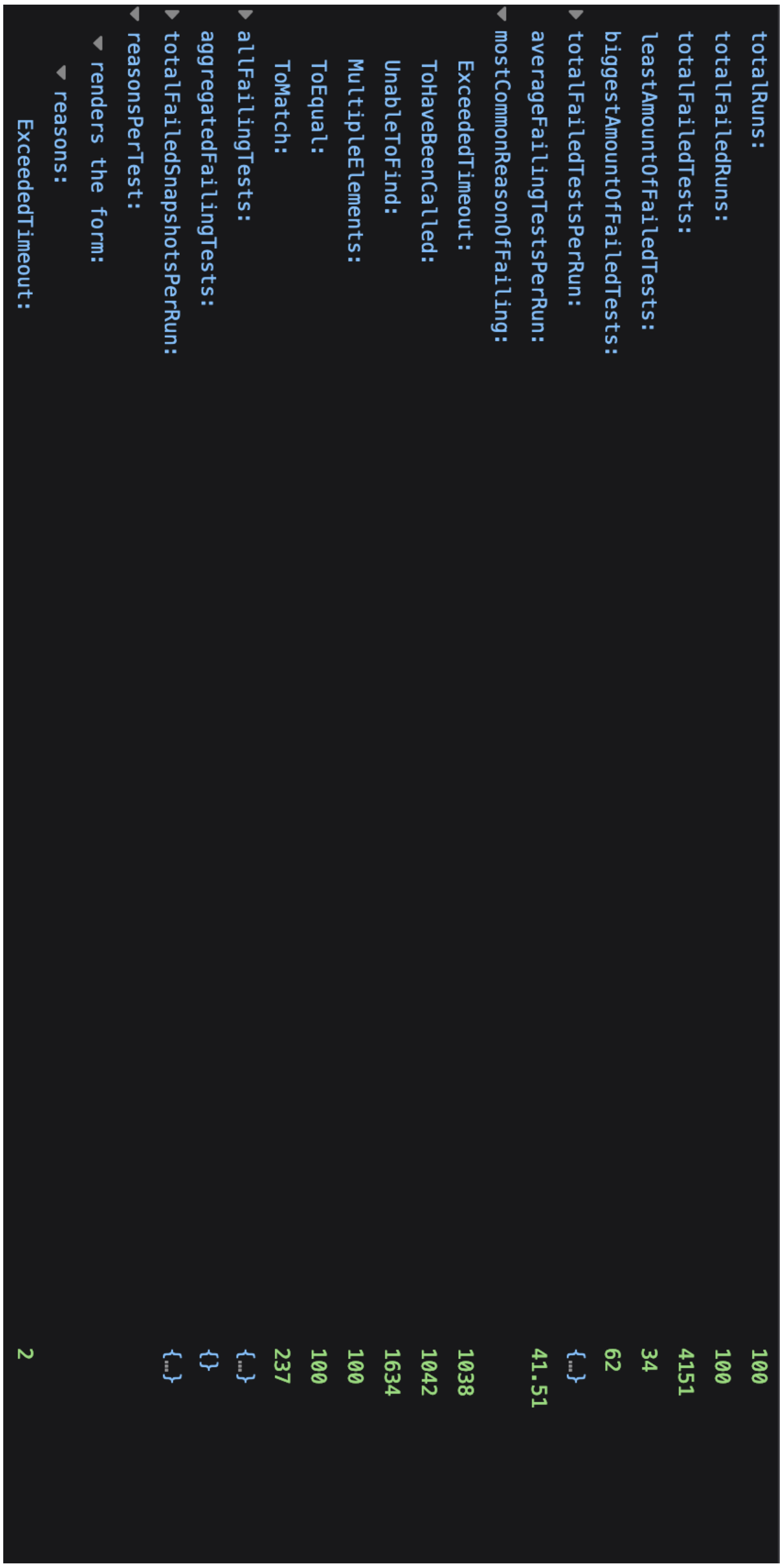


```
const setGeneralMostCommonErrorReason = (errorMessage: string) => {
  if (errorMessage.includes("Exceeded timeout")) {
    const current = result.mostCommonReasonOfFailing[ErrorReason.ExceededTimeout];
    result.mostCommonReasonOfFailing[ErrorReason.ExceededTimeout] = current ? current + 1 : 1;
  }
  else if (errorMessage.includes("toMatch")) {
    const current = result.mostCommonReasonOfFailing[ErrorReason.ToMatch];
    result.mostCommonReasonOfFailing[ErrorReason.ToMatch] = current ? current + 1 : 1;
  }
  else if (errorMessage.includes("toEqual")) {
    const current = result.mostCommonReasonOfFailing[ErrorReason.ToEqual];
    result.mostCommonReasonOfFailing[ErrorReason.ToEqual]  = current ? current + 1 : 1;
  }
  else if (errorMessage.includes("Unable to find")) {
    const current = result.mostCommonReasonOfFailing[ErrorReason.UnableToFind];
    result.mostCommonReasonOfFailing[ErrorReason.UnableToFind] = current ? current + 1 : 1;
  }
  else if (errorMessage.includes("toHaveBeenCalled")) {
    const current = result.mostCommonReasonOfFailing[ErrorReason.ToHaveBeenCalled];
    result.mostCommonReasonOfFailing[ErrorReason.ToHaveBeenCalled] = current ? current + 1 : 1;
  }
  else if (errorMessage.includes("Found multiple elements")) {
    const current = result.mostCommonReasonOfFailing[ErrorReason.MultipleElements];
    result.mostCommonReasonOfFailing[ErrorReason.MultipleElements] = current ? current + 1 : 1;
  }
}
```



```
// Analyze all the data

result.totalRuns = files.length;

files.forEach((file: string) => {
  const testResult = JSON.parse(fs.readFileSync(`./testResult/testResultJson/${file}`));
  const runNumber = parseInt(file.replace('.json', ''));
  if (!testResult.success) {
    setStandardData(runNumber, testResult.numFailedTests);

    setLeastAndBiggestAmountFailingResults(testResult.numFailedTests);

    setStandardErrorData(runNumber, testResult.testResults);

    // Snapshots
    result.totalFailedSnapshotsPerRun[runNumber] = testResult.snapshot.filesUnmatched;
  }
});

// Set averages
const avg = (result.totalFailedTests / result.totalFailedRuns) || 0;
result.averageFailingTestsPerRun = avg;


// Write result to file
fs.writeFile("resultAnalysis.json", JSON.stringify(result), function(err: Error) {
  if (err) {
      console.log(err);
  }
});
```

19 Appendix C: Result from script



```
totalRuns:                              100
totalFailedRuns:                        100
totalFailedTests:                       4151
leastAmountOfFailedTests:               34
biggestAmountOfFailedTests:             62
▶ totalFailedTestsPerRun:               {…}
averageFailingTestsPerRun:              41.51
▼ mostCommonReasonOfFailing:
    ExceededTimeout:                    1038
    ToHaveBeenCalled:                   1042
    UnableToFind:                       1634
    MultipleElements:                   100
    ToEqual:                            100
    ToMatch:                            237
▶ allFailingTests:                      {…}
aggregatedFailingTests:                 {}
▶ totalFailedSnapshotsPerRun:           {…}
▼ reasonsPerTest:
  ▼ renders the form:
    ▼ reasons:
        ExceededTimeout:                2
```