Imperial College London

MEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

Performance-Test-Driven Development for WebApps

Author:
Zhai Zirun

Supervisor:
Dr Robert Chatley

June 22, 2023

Submitted in partial fulfillment of the requirements for the MEng Computing degree of Imperial College London

Abstract

Performance optimization is critical for websites as it directly impacts user experience, revenue, and search engine rankings. Synchronous and asynchronous Javascript affect page loading time significantly. However, existing performance testing tools rely on measuring execution times of deployed systems, and are hence unsuitable for early-stage unit testing. Furthermore, these tools have high turnaround times, hindering continuous performance testing even after deployment. No previous tool exists for early-stage performance testing in front-end web-app development, resulting in technical debt and suboptimal final products. The development of a pre-deployment performance testing tool with fast turnaround times is therefore crucial.

This project proposes QuiP, a novel tool for performance testing Javascript front-end web-app code before deployment. QuiP extends the Jest testing framework to enable quick performance testing. It utilizes performance models for efficient runtime estimation in virtual time. To enable QuiP to analyze asynchronous code using the Async hooks API, this project also proposes a novel systematic search that identifies deviations between asynchronous dependencies and execution context relationships in Javascript.

QuiP is thus unique in that it can automatically analyze both synchronous and asynchronous code, which is novel in the field of performance testing before deployment. We demonstrate these capabilities by applying QuiP to the development of the front-end of an example web-app, showcasing its ability to accurately process complex combinations of asynchronous dependencies. Although only based on simple performance models, QuiP can predict percentage runtime changes within 20% before deployment and within 5% after refining with empirical data. QuiP outperforms state-of-the-art performance measurement tools with a 765% faster turnaround time while minimally impacting Jest unit test turnaround. These features demonstrate QuiP to be a promising candidate for integration into continuous unit testing practices for the front-end.

Acknowledgments

I would like to express my gratitude to my supervisor, Dr Robert Chatley, for your invaluable guidance, constant enthusiasm, and insightful ideas throughout this project. Thank you for always having patience for my questions.

I would also like to thank my parents for their support all these years. I would not be here without you (literally). Thank you for always helping me do what I want to do, and for inspiring me to be the best version of myself.

Last but definitely not least, thank you to my friends for helping me laugh through the pain. Special mention goes to Ethan (my boyfriend) for coming up with the name QuiP.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objectives . . . 2
1.3 Contributions . . . 2

2 Background **3**
2.1 Front-end Performance Testing . . . 3
2.1.1 Performance metrics . . . 3
2.1.2 Significance of Javascript in front-end performance . . . 4
2.1.3 Timing of performance testing in development . . . 4
2.2 Existing Tools to Measure Front-end Performance . . . 5
2.2.1 Javascript Libraries . . . 5
2.2.2 Google Lighthouse and Google PageSpeed Insights . . . 5
2.2.3 DrAsync . . . 6
2.3 Current Performance Testing in Test-Driven-Development . . . 7
2.3.1 PerfMock . . . 8
2.3.2 Estimating performance with performance models . . . 9
2.4 Unit Testing Frameworks in the Front-end . . . 10
2.4.1 Using Jest . . . 10
2.4.2 Extending `expect` . . . 11
2.4.3 Extending mocks . . . 12

3 Design and Implementation **13**
3.1 Project Design: Extending over Modifying Jest . . . 14
3.2 Architecture . . . 14
3.3 Runtime Estimation . . . 15
3.3.1 Investigating when mocks are called . . . 15
3.3.2 Estimating mock response time with performance models . . . 16
3.3.3 Attaching models to mocks . . . 16
3.3.4 Adding up runtimes . . . 17
3.3.5 Visualizing timelines . . . 18
3.4 Asynchronous Execution Analysis . . . 20
3.4.1 Async mocks with different types of execution . . . 21
3.4.2 Async and non-async mocks with different types of execution . . . 24
3.4.3 Nested function calls with different types of execution . . . 27

4 Usage **31**
4.1 Checkout architecture . . . 31
4.2 Writing performance tests with QuiP . . . 32
4.2.1 Setting up test context and mocks . . . 32
4.2.2 Writing tests for the checkout view . . . 33
4.3 Fixing failing performance tests . . . 35
4.4 Refining tests with empirical data . . . 36

5 Evaluation **39**
5.1 Evaluating asynchronous dependency parsing by QuiP (RQ1) 40
5.1.1 Serial and concurrent executions 40
5.1.2 Async and non-async mocks 41
5.1.3 Nested and non-nested function calls 41
5.1.4 Summary 43
5.2 Evaluating runtime predictions (RQ2) 43
5.2.1 Runtime measurement results 43
5.2.2 Factors leading to inaccuracy in runtime estimation 45
5.2.3 Summary 45
5.3 Evaluating predictions of the effect of code changes (RQ3) 45
5.3.1 Runtime measurement results 46
5.3.2 Accuracy of percentage change estimation despite inaccuracy of runtime estimation 46
5.3.3 Summary 47
5.4 Evaluating QuiP's effect on performance test turnaround time (RQ4) 47
5.4.1 Comparison with current performance testing methods 48
5.4.2 Comparison with current unit testing methods 49
5.4.3 Summary 49

6 Conclusion and Future Work **50**
6.1 Future Work 50
6.2 Ethical Considerations 51

Chapter 1

Introduction

1.1 Motivation

Performance optimization is crucial for websites. Faster web pages enhance the satisfaction of users, which leads to higher traffic and repeat visits. Ultimately, this raises page revenue and search engine rankings [1]. Every 100ms of latency in Amazon's site costs it 1% of sales [2]. Computational processes contribute up to 35% of page loading time, with a large proportion being synchronous and asynchronous Javascript code [3]. However, performance testing is usually not performed in the early stages of Javascript development [4]. This incurs technical debt and affects the performance of the end product [5]. Therefore, it would be ideal to conduct performance testing earlier, throughout development.

The reason why this is not standard practice for the front-end is because performance testing tools for the front-end, such as Google Lighthouse and DrAsync, currently rely on measurements of the execution times of deployed systems [6]. However, unit testing should be performed from the start of development with testing frameworks such as Jest, before the software is deployed [7]. Therefore, it is currently impossible for early-stage unit tests for the front-end to involve performance testing tools. Moreover, due to these tools' reliance on real-time measurements, they require a high turnaround time [8]. This obstructs continuous performance testing even after deployment.

Motivated by the benefit earlier testing can have on performance, performance testing before deployment is an active research area. Of great interest is PerfMock, an extension upon the jMock2 mock object framework for Java that performs runtime estimations based on performance models. This enables performance unit testing in the back-end with fast turnaround times [9].

In order to process both synchronous and asynchronous Javascript for runtime estimation in the front-end, the asynchronous dependencies in code need to be detected. Notably, DrAsync adopts the node.js Async hooks API, which uses the execution context feature of Javascript to dynamically analyze the dependencies between asynchronous deployed code [10]. However, Async hooks could be leveraged to analyze code before deployment as well.

Therefore, no previous tool exists for testing front-end performance before deployment, especially with fast turnaround times. To do so involves estimating runtime for both synchronous and asynchronous code without relying on real-time measurements. Existing research can only estimate the runtime of synchronous code for the back-end (in Java). Hence, novel techniques are expected to be developed to handle asynchronous dependencies in JavaScript for the front-end, even though the required libraries (such as the Async hooks API) rely on poorly documented features of JavaScript. [11]

1.2 Objectives

Therefore, this project aims to overcome the current limitation in front-end performance testing, and develop a novel tool that enables performance testing in front-end Javascript prior to deployment. It should provide fast turnaround times, which makes it suitable for integration into continuous unit testing. The more specific objectives of this work are as follows.

- Propose a novel tool for performance testing front-end web-application code before deployment. It can estimate the effect of synchronous and asynchronous Javascript on runtime, with short turnaround times.
- Demonstrate how the proposed tool can be used to test the performance of synchronous and asynchronous Javascript before deployment, by applying it to the development of the front-end of an example web-app.
- Evaluate the value that the proposed tool adds to current coding practices for the front-end, by testing its ability to estimate the effect of code on runtime before and after deployment and its test turnaround time.

1.3 Contributions

The contributions of this project are as follows:

- We develop a novel tool, QuiP, for performance testing JavaScript front-end web-app code before deployment. QuiP utilizes performance models to estimate runtime, with a focus on performance optimization rather than estimation accuracy. The design and implementation of QuiP as an extension to the Jest testing framework are presented in Chapter 3.
- To enable QuiP to analyze asynchronous code, we perform a novel systematic search that identifies deviations between asynchronous dependencies and execution context relationships in Javascript. This process, explained in Section 3.4 of Chapter 3, allows us to effectively use the Async hooks API in QuiP's implementation.
- QuiP is thus unique in that it can automatically analyze both synchronous and asynchronous code, which is novel in the field of performance testing before deployment. We demonstrate this capability by integrating it into the development of the front-end of an example web-app in Chapter 4.
- We evaluate the accuracy and efficiency of QuiP in Chapter 5 by using it to analyze code from our example web-app. These tests show that QuiP is capable of:
  - Accurately processing complex combinations of asynchronous dependencies in JavaScript, covering five out of the six considered cases of asynchronous code. It may have slight limitations in handling specific edge cases within the last case, concurrent executions.
  - Predicting the percentage change in runtime after a code change within 20% of the actual change before deployment, and within 5% after refining performance models with empirical data from deployment. Further accuracy can be achieved with more sophisticated performance models.
  - Outperforming other performance measurement tools (such as Google Lighthouse) with a 765% faster turnaround time while increasing Jest unit test turnaround time by only 11.3%.
- The project is fully open-source and available to install as a npm package at `https://www.npmjs.com/package/jest-performance-monitor`. The project code is available at `https://github.com/zzirun/jest-performance-monitor`, which also includes the code for the examples generated in this work.

Chapter 2

Background

This chapter provides an overview of the current techniques in performance testing for the front-end, and links it to unit testing. Section 2.1 explains the performance of front-end web-app code, and the significance of testing and optimizing it earlier in development. Section 2.2 introduces existing popular tools to measure performance. These tools have limited functionality before the system can be deployed, due to their reliance on concrete measurements. Section 2.3 thus presents the state-of-the-art technique of integrating performance testing into Test-Driven-Development with the PerfMock study (for Java development in the back-end). It bypasses the need for exact measurements by estimating performance using performance models, but is incapable of analyzing asynchronous code. Finally, Section 2.4 introduces Jest, a unit-testing framework for Javascript. This can provide a basis for our proposed technique of extending Javascript unit testing for performance optimization.

2.1 Front-end Performance Testing

Performance is an integral consideration in software engineering. The performance of front-end web-app code is often a determining factor in its success [12]. Faster web-apps improve the experience of its users, raising user satisfaction. This impacts its search engine rankings and user traffic, ultimately increasing its online revenue [1]. For instance, the BBC observed that they lost 10% of their users with every second their Page Load Time (PLT) increased by [13]. Developers of web-apps are therefore motivated to measure performance as part of testing [14].

2.1.1 Performance metrics

While measuring the performance of the back-end may involve factors such as resource utilization, memory leak detection, and response times, the performance of the front-end can be measured by its functionality and speed [15]. In particular, we focus on how long users have to spend waiting for the page to load.

Overall, this is indicated by the page's PLT. The PLT is defined as the time between when the page is first requested, and when the `DOMLoad` event is fired (i.e. when all embedded objects in the page are fetched and added to its `DOM`) [3].

More nuanced metrics may also be adopted for further insight on user experience. For example, First Contentful Paint (FCP) is the time taken for the first element or piece of content on the page to be rendered. The Total Blocking Time is how long the page is non-responsive for, while it is loading. Time to Interactive (TTI) is the time it takes for users to be able to interact with the application, and First Input Delay (FID) measures the response time of the application when the user makes an input [6].

2.1.2 Significance of Javascript in front-end performance

According to stack overflow, Javascript has held the title of most popular programming language for 10 years running, especially for professional software engineers [16]. This makes it ubiquitous for front-end development, and it plays a critical role in website performance.

The loading time of web pages is determined by a multitude of factors including its Javascript, due to the complexity of the loading process. This is illustrated by Figure 2.1 (from [3]). The process begins when a user creates a request. The Object Loader then downloads the page's root HTML, which is then sent to the HTML Parser. Once the HTML Parser receives its first chunk, it begins parsing the page and downloading its embedded objects. These objects have varying types, but two of them in particular (Javascript and CSS) require further Evaluation. Thereafter, the Rendering Engine renders the page on the web browser. The Object Loader is a network process, while the HTML Parser, Evaluator and the Rendering Engine are computation processes [3].

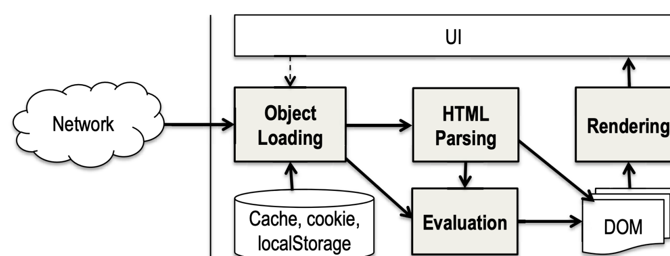


Figure 2.1: The workflow of a loading page (from [3])

WProf, a profiling tool that investigates the dependencies of a page's load time, was applied to the top 200 most visited web pages and 200 random pages from the top million most visited web pages (according to Alexa). It found that computation processes took a significant amount of time, 35% of the total PLT. Among these processes, Javascript evaluation took up a high proportion of time. The work also found that Javascript has been increasingly embedded into web pages [3]. It can be seen that a web page's Javascript has great bearing on its performance, and thus performance should be carefully considered during development.

On the other hand, this may not be universal, as there are optimization techniques that can help to reduce the impact of Javascript on performance. Firstly, caching reduces computation time, as browsers tend to cache the intermediate computation steps of web pages [3]. However, this will not have an effect on the first time users load a page, which could heavily determine if they continue using it. Google found that 53% of users give up on a mobile site if it takes more than 3 seconds to load [17]. Therefore it is insufficient to solely rely on caching to improve PLT.

Secondly, Javascript evaluation takes up a significantly reduced fraction of the critical path of page loading when asynchronous Javascript is used. This is because synchronous Javascript blocks the HTML parser from parsing further till it is fully evaluated. Asynchronous Javascript removes this blocker, which helps parsing to take less time [3]. Therefore, developers would be highly motivated to utilize asynchronous Javascript to optimize performance. Performance testing hence needs to be accessible for both synchronous and asynchronous Javascript.

2.1.3 Timing of performance testing in development

Performance testing is located in the fourth and last of the Agile Testing Quadrants. This is attributed to Agile development's focus on functionality prioritized by (often non-technical) customers. These customers, on the other hand, often overlook performance as a consideration that developers take care of independently [18]. Moreover, performance testing tools can only reliably measure the performance of deployed systems, and doing so involves a high turnaround time. Thus, developers usually

need to wait till the full software system is deployed to conduct performance testing [4].

However, conducting performance testing this late in development is not ideal. It means that inadequate performance is usually only detected when a high amount of technical debt has been accumulated. It may be expensive to fix this problem as the software system has already been fully implemented. Any optimizations may require large changes to the design of the system, which is highly inefficient [5]. It would be problematic if the optimizations were too expensive to perform, given how instrumental performance is to a web-app. Early performance testing before the software is deployable can prevent the above problems.

2.2 Existing Tools to Measure Front-end Performance

In general, front-end performance is currently monitored in local “lab” environments, and in the “real world” based on the usage environments of real users [6]. Within the local environment, Javascript has some functions that can measure execution timings using the system clock.

More insights can be obtained using browser-side tools that collect metrics by observing the deployment of the web-app. These tools also support performance optimization by inspecting elements of the page for sources of poor performance. Popular examples are Google Lighthouse and Google PageSpeed Insights. However, these tools cannot inspect code and thus have limitations for performance optimization.

On the other hand, DrAsync is an example of a code-side tool that can specifically inspect Javascript code. It helps to minimize misuse of asynchronous Javascript. Overall, however, the functionality of all of these tools are limited before the front-end can be deployed due to their reliance on concrete time measurements.

2.2.1 Javascript Libraries

There are several Javascript functions that report information on system time. They can be applied to measure PLT. For example, `console.time` and `console.timeEnd` can be used to record time elapsed in milliseconds. A usage example can be seen in Listing 2.1. These functions can be leveraged to measure the time taken to execute code. However, they cannot be used to measure the execution timings of functions that have not yet been implemented.

```
1  console.time("Test 1");
2     // functions here
3     console.time("Test 2");
4     // functions here
5     console.timeEnd("Test 2");
6  console.timeEnd("Test 1");
7
8  // this prints to output:
9  // Test 2: 0.575ms
10 // Test 1: 3.38293ms
```

Listing 2.1: Example usage of `console.time` and `console.timeEnd`

2.2.2 Google Lighthouse and Google PageSpeed Insights

Google lighthouse is one of Google Chrome’s developer tools (DevTools) that audits the quality of a website and suggests improvements based on failing audits. It can be run from the command line, in Chrome DevTools, or as a Node module. These audits fall under the categories of performance, accessibility, best practices, SEO (Search Engine Optimization), and PWA (Progressive Web-Apps) [19]. The performance audits measure a list of metrics such as FCP, TTI, and so on by reloading the site and recording when relevant milestones occur [14]. Lighthouse then inspects elements of the page

to suggest optimization opportunities such as sizing images properly and removing render-blocking resources [19].

There are some limitations to Lighthouse's functionality. As the tests are conducted locally with a single run per report, environmental fluctuations in client hardware and resource contention, as well as the network and web server, may result in inconsistencies in the measurements. Hence, users have to resort to aggregating the results of multiple runs [14]. Alternatively, Google PageSpeed Insights can perform the same analysis on individual web pages, but with real-user experience data and a more limited set of metrics [20].

Notably, both Lighthouse and PageSpeed Insights can also be integrated into local automatic testing. Lighthouse can be run programmatically as a Node module [21], and PageSpeed Insights has a callable API [22]. Both of these allow developers to test for performance automatically once deployable versions of the website are created.

The drawback of doing this is the large amount of time it would take, as the site would have to be reloaded with each relevant test. The HTTP Archive reports that even the fastest 1% of websites render TTI in approximately 2.2 seconds [8]. To test the performance of such websites, the developer would have to repeatedly load the site, taking 2.2 seconds each time. This leads to a high test turnaround time, which impedes the efficiency of testing.

Moreover, both Lighthouse and PageSpeed Insights can only be used to measure performance once the website is deployable to production (for PageSpeed Insights), or at least to localhost (for Lighthouse). Therefore, they are unable to address the problem of performance testing earlier in development.

2.2.3 DrAsync

A tool that can help to optimize front-end performance before it is fully deployed is DrAsync. DrAsync performs two types of analysis: static and dynamic, targeted at improving the usage of promises and `async/await` in implementing asynchronous Javascript [10].

DrAsync performs a static analysis for instances of previously defined antipatterns, which are pieces of code that result in sub-optimal performance. For example, the *asyncFunctionNoAwait* anti-pattern identifies `async` functions that do not contain `await` expressions. These are formatted as CodeQL queries that can be ran at any stage of development [10]. However, this static analysis is highly limited in that it is only able to check for previously known mistakes, with only 8 antipatterns being checked for in the study. The tool would not be able to help with blind spots that the developer is unaware of. It is also not very extensible, as additional antipatterns have to be hard-coded as queries.

Interestingly, DrAsync uses the Async hooks API to perform its dynamic analysis to visualize promise lifetimes. The Async hooks API facilitates this in two ways. Firstly, it enables customizable callbacks during the creation and resolution of resources. This allows DrAsync to log the system time these events occur at [10]. Secondly, Async hooks assigns `asyncIds`, as well as `triggerAsyncIds` to asynchronous resources. This was used to report the dependencies between asynchronous resources during deployment, which would be highly useful for debugging and optimizing performance [10]. However, similarly to Google Lighthouse, this feature can only be used at run-time once the system is deployable, and therefore has limited effect during the development process. Clearly, DrAsync has only explored `asyncId` in a limited way. This feature was applied to determine the dependencies between promises in deployable code, but it could be used to analyze code under test as well. Modifications could be made to the analysis procedure, such as using mocks to replace dependencies that have yet to be implemented.

Moreover, DrAsync makes the assumption that **unique** `asyncIds` are assigned to each **promise** [10]. This is not necessarily true - unique `asyncIds` are actually assigned to each **execution context**. Each execution context also has a `triggerAsyncId`, which is the `asyncId` of the execution context that

triggered it [23].

In Javascript, a promise is returned by every async function, and represents its completion. It acts as a placeholder for the eventual value returned by the function. Meanwhile, an execution context is the environment where code is executed. The global execution context in Javascript is its base execution context, where all code is run. This is assigned the `asyncId` 0. On the other hand, functional execution contexts are created upon every function invocation [24]. The Async hooks API monitors the lifetimes of execution contexts related to asynchronous operations. An execution context triggering another means that the new context was created due to the triggering context finishing its execution (such as sequential function calls), not that the new context was created within the triggering context (such as a function call within a function).

However, these contexts may not necessarily be unique to promises. For example, concurrent promises share the same execution context, and therefore share the same `asyncId`. Their individual relationships with subsequently triggered promises cannot be identified, as it is impossible to tell which specific promise, or all of them, triggered their child with just one shared `asyncId`.

Fortunately, this does not strongly affect DrAsync's functionality, as a large part of its visualization relies on timelines generated from promise runtimes in real system time [10]. Most ambiguities can be resolved with this additional information. However, before deployment is possible, real system time elapsed during Promises may be very small, making it more difficult to distinguish dependencies.

Finding ways to overcome this would be a necessary but non-trivial aspect of applying Async hooks to analyze code before deployment. This task is difficult due to the lack of exhaustive documentation on asynchronous execution contexts and their relationship to asynchronous Javascript [11].

2.3 Current Performance Testing in Test-Driven-Development

Agile testing and the continuous delivery of software have been increasingly adopted in software development [25]. These techniques involve incremental delivery of new functionality and testing changes during the development process [26], and can be applied to both the front-end and the back-end. By using mock objects to replace collaborators (external dependencies) of an object under test, components of software can undergo unit testing in isolation [27]. This is often employed in Test-Driven-Development (TDD). The 'red, green, refactor' cycle in TDD, as illustrated by Figure 2.2 (from [28]), has the benefit of creating fast feedback loops, which can reassure the developer that each change fits the current requirements and reduces the likelihood of large, expensive refactoring [7]. These tests are written and ran far before a deployable version of the software is completed. As the performance of a system cannot be reliably measured before it is deployed, it is difficult to consider performance in unit testing.

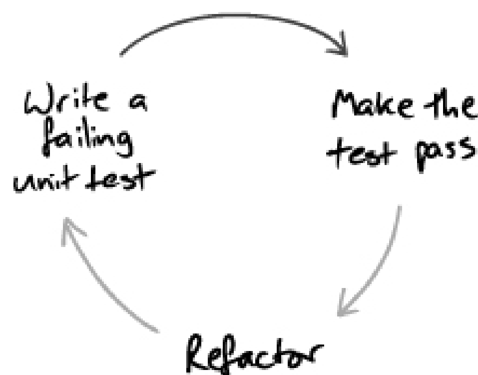


Figure 2.2: 'Red, green, refactor' cycle in Test-Driven Development (from [28])

On the other hand, PerfMock illustrates an approach to measuring performance in unit testing through estimating it with models. This has the additional benefit of ensuring low turnaround times.

2.3.1 PerfMock

PerfMock, an extension upon a mock object framework, was created to enable performance unit testing in the back-end for Java. Out of all the aspects of performance, response time is the most relevant to unit testing, since it can be directly measured by micro-benchmarking code [29]. However, it is difficult to determine response times when collaborators are replaced by mock objects, as they lack the internal logic of the collaborator being mocked [9].

PerfMock solves this problem by making it possible to assign performance models to mock objects. The performance models, ranging from (deterministic) average delays to probability distributions, are estimators for how long the collaborator will take to process messages it received. PerfMock performs these estimations in virtual time, which makes it possible to test performance in a continuous way before the full software can be deployed. This provides fast turnaround times, which is crucial to TDD. The models can also be iteratively refined as more real-time data is made available, such that predicted performance converges to actual performance [9].

```
1  public class CassandraUserServiceTest {
2      @Rule
3      public PerformanceMockery ctx =
4          new PerformanceMockery();
5      CassandraOperations db = context.mock(
6          CassandraOperations.class,
7          PerformanceModels.cassandraOpsModel());
8      @Test
9      public void getUserAliceFromCassandra() {
10         UserService users = new CassandraUserService(db);
11
12         ctx.repeat(2000, () -> {
13         ctx.checking(new Expectations() {{
14             exactly(1).of(db).selectOne(
15                 "SELECT * FROM users WHERE username='Alice';",
16                 User.class);
17             }});
18             users.getByUsername("Alice");
19         });
20
21         assertThat(ctx.runtimes(), matchMean(
22         PerformanceModels.userServiceModel()));
23     }
24 }
```

Listing 2.2: Example unit test written with PerfMock, using the matchMean function

An example unit test written with PerfMock is shown in Listing 2.2. A performance model, `cassandraOpsModel`, can be chosen according to the assumed performance characteristics of the external Cassandra service. The accuracy of the model can be assessed by checking whether it passes the `matchMean` check. Once the test passes, the model used can then be applied to estimate the performance impact of changes made to code calling on the Cassandra service.

```
1  public class TweeterControllerTest {
2      @Test
3      public void rendersUserTimelineWithReplies() {
4          TweeterController ctlr = new TweeterController(...);
5          User alice = new User("Alice");
6          List<Message> TEN_MSGS = ...
7
8          ctx.repeat(2000, () -> {
9              ctx.checking(new Expectations() {{
10                 exactly(1).of(users).getByUsername("Alice");
11                     will(returnValue(alice));
```

```
12                  exactly(1).of(msgs).getUserTimeline(alice);
13                      will(returnValue(TEN_MSGS));
14                  exactly(10).of(msgs).getReplies(
15                      with(any(Message.class)));
16              }});
17              ctlr.userTimeline("Alice", new ModelMap());
18          });
19
20          assertThat(ctx.runtimes(),
21              hasPercentile(80, lessThan(15.0)));
22      }
23  }
```

Listing 2.3: Example unit test written with PerfMock, using the hasPercentile function

Models can also be set as performance targets. Listing 2.3 illustrates an example where the test ensures that the runtime of the process tested is up to standard according to the model used. The test context's runtime `ctx.runtimes` is updated with the duration of execution of each process (either from real time, or virtual time as estimated by its model), and this runtime is expected to be within the 80th percentile of its performance model.

PerfMock does have limitations due to its reliance on estimations of performance before real data is made available. The performance of collaborators may be inaccurately estimated to a great extent, and there is a trade-off between accuracy and detail. Moreover, even when empirical data is available, models may still not be 100% accurate as device-side times are inconsistent and may cause actual performance to deviate [9]. This inaccuracy, however, is likely not to be higher than that of other performance measurement tools working locally, as they would have the same sources of error.

Moreover, PerfMock is still able to provide an estimate of whether code changes will cause performance to improve or worsen. This can provide valuable guidance for developers [9]. Fine tuning its accuracy with empirical data will make it relevant as an automatic performance testing tool that can be integrated into unit testing throughout the process of engineering a piece of software, before and after it is deployed for concrete performance measurements.

Therefore, the same idea of estimating performance with models during unit testing could be applied to performance testing in the front-end as well. However, PerfMock is concerned with serial executions, and is unable to estimate the runtimes of concurrent code. The Async hooks API, as used by DrAsync, could possibly be used to adapt this approach to analyze async Javascript.

2.3.2 Estimating performance with performance models

A performance model is any piece of code that returns a numerical estimation of the response time of the object represented [9]. Since exact measurements of time delay cannot be obtained till the system is deployed, estimations can be used to simulate how code changes can affect loading times. Even after the system is deployed, since it takes a long time to retry deployment, estimates in virtual time can offer a method of measuring performance with lower turnaround time.

Before deployment, performance models can be constructed based on theory. For example, a log-normal distribution can be used to represent the performance of network services, [9] since many network services have heavy tails [30]. Existing research on modelling the performance of big-data systems, such as Apache Hadoop [31], or No-SQL databases, such as Apache Cassandra [32], can be utilized to develop performance models.

After the system is deployed, the PLT can be measured. Performance models can now be constructed as aggregates of empirical data. For example, a large number of trials can be conducted to obtain a dataset of PLT, which can be approximated as a normal distribution. This distribution can then be implemented as a performance model to predict future PLT.

2.4 Unit Testing Frameworks in the Front-end

Given the significant amount that Javascript evaluation contributes to page loading time, as explained in Section 2.1, improving performance testing in Javascript development is highly important.

There are many popular Javascript frameworks used to create unit tests. Examples include Jest [33], Mocha [34], AVA [35], and Jasmine [36]. Jest was designed by a Meta team for the React library, as an extension to Jasmine [37]. In particular, it is known for its simplicity, large amount of features, and comprehensive documentation. It is also widely used, having a large presence on stack overflow [38]. These factors will all contribute to the likelihood of Javascript developers using Jest for unit testing.

Since using a popular testing framework increases the applicability of our work, we will focus on implementing performance testing by extending the Jest unit testing framework in Javascript. This section aims to introduce the relevant Jest boilerplate for writing unit tests, as well as frameworks for extending them. It is impossible exhaustively introduce its library functions - all functions used are defined and explained in Jest documentation.

2.4.1 Using Jest

The `describe` function is used to create a test suite/block. Test cases can be defined within it with the `it` function [39]. Matchers are used to assert the expectations of each test, with the `expect` object [40]. These tests can be grouped to properly represent the object they describe [41].

Additionally, the `beforeEach` function can be used to reduce duplication, as it is called before each test in the block. Similarly, the `afterEach` function is called after each test. The `beforeAll` and `afterAll` functions are called before and after all of the test cases [39].

Listing 2.4 (from [41]) shows an example of tests written for the implementation of a calculator. The `toEqual` matcher compares the `result` argument of `expect(result)` with its own argument, and asserts that they are equal. Test results will then be printed based on the result of the assertions.

```
1  describe('addition', () => {
2      let calc = null
3      beforeEach(() => {
4          const options = {
5              precision: 2
6          }
7          calc = new Calculator(options)
8      })
9      it('adds two positive numbers', () => {
10         const result = calc.add(1.333, 3.2)
11         expect(result).toEqual(4.53)
12     })
13     it('adds two negative numbers', () => {
14         const result = calc.add(-1.333, -3.2)
15         expect(result).toEqual(-4.53)
16     })
17 })
```

Listing 2.4: Example of Tests written in Jest (from [41])

Jest also supports creating mocks. Mocks lack the internal logic of the object they are mocking, and are used to avoid making a real call to the object. A mock function can be created with `jest.fn()` [42]. An example is given in Listing 2.5 (from [41]). The mock tracks its invocations, and can be audited by `expect`.

```
1      it('is callable', () => {
2          const mock = jest.fn()
```

```
3           mock('arg')
4           expect(mock).toHaveBeenCalled()
5           expect(mock).toHaveBeenCalledWith('arg')
6           expect(mock).toHaveBeenCalledTimes(1)
7       })
```

Listing 2.5: Example usage of jest.fn (from [41])

2.4.2 Extending `expect`

The `expect` object can be extended to define the developer's own matchers. This can be done with the `expect.extend` function, within which a custom matcher function can be defined. This can then be used in tests [40]. An example is Listing 2.6 (from [40]). The toBeWithinRange matcher is newly defined to return unique pass/fail messages and a boolean based on the result of matching.

```
1   expect.extend({
2       toBeWithinRange(received, floor, ceiling) {
3           const pass = actual >= floor && actual <= ceiling;
4           if (pass) {
5               return {
6                   message: () =>
7                       `expected ${received} not to be within range
8                           ${`${floor} - ${ceiling}`}`,
9                   pass: true,
10              };
11          } else {
12              return {
13                  message: () =>
14                      `expected ${received} to be within range
15                          ${`${floor} - ${ceiling}`}`,
16                  pass: false,
17              };
18          }
19      }
20  })
21
22  ...
23      it('identifies numbers within a range', () => {
24          expect(100).toBeWithinRange(90, 110)
25      })
26  ...
```

Listing 2.6: Example of defining a matcher in Jest (from [40])

`expect.extend` can also support `async` matchers, which will return a `Promise` with the results. Tests using the matcher have to be `async` as well, and `await` the settlement of the matcher's `Promise`. While a `Promise` awaited is still pending, execution of the test pauses, and control returns to the main event loop. When the `Promise` is settled with a value, or rejected with an error, the result of the expect is printed and execution of the test resumes, such as by executing the next await expression [40]. An example can be found in Listing 2.7 (from [40]).

```
1   expect.extend({
2       async toBeDivisibleByExternalValue(received) {
3           const externalValue =
4               await getExternalValueFromRemoteSource();
5           const pass = received % externalValue == 0;
6           if (pass) {
7               return {
8                   message: () =>
9                       `expected ${received} not to be divisible by
10                          ${externalValue}`,
11                  pass: true,
12              };
13          } else {
```

```
14              return {
15                  message: () =>
16                      `expected ${received} to be divisible by
17                          ${externalValue}`,
18                  pass: false,
19              };
20          }
21      },
22  })
23
24  ...
25      it('is divisible by external value', async () => {
26          await expect(100).toBeDivisibleByExternalValue()
27          await expect(101).not.toBeDivisibleByExternalValue()
28      })
29  ...
```

Listing 2.7: Example of defining an async matcher in Jest (from [40])

2.4.3 Extending mocks

Mocks created with `jest.fn()` can optionally take in a mock implementation to define its behaviour, in both synchronous and asynchronous contexts [42]. For example, in Listing 2.8 (from [41]), `fakeAdd` is a mock that takes in the implementation `(a, b) => 5`. Therefore, when it is called with `(1, 1)`, it will return 5, as expected by `toBe(5)`.

```
1      it('adds two positive numbers', () => {
2          const fakeAdd = jest.fn().mockImplementation((a, b) => 5)
3          expect(fakeAdd(1, 1)).toBe(5)
4          expect(fakeAdd).toHaveBeenCalledWith(1, 1)
5      })
```

Listing 2.8: Example of customizing the implementation of mocks in Jest (from [41])

Jest is also capable of automatically mocking collaborator modules with `jest.mock`. For example, `axios` is a module used to call APIs asynchronously. This should be mocked in unit testing to prevent real calls to the API. In Listing 2.9 (from [43]), the `async` test for `swapiGetter` aims to test its ability to call an api through `axios`. `axios` is imported as `mockAxios` to indicate that it is being mocked locally. It is mocked with `jest.mock` and a custom implementation is set for the mock using `mockResolvedValue`. This function is a refactor of `mockImplementation`, and causes `mockAxios` to return a Promise resolved with its input. Therefore, when `swapiGetter` is called, it calls upon `mockAxios` instead of `axios`, and we are able to monitor its actions towards axios through the mock.

```
1   import swapiGetter from "../swapiGetter";
2   import mockAxios from "axios";
3
4   jest.mock("axios");
5   mockAxios.get.mockResolvedValue({ data: { name: "Jimmy Jedi" } });
6   // is equivalent to
7   // mockAxios.get.mockImplementation(() =>
8   //     Promise.resolve({ data: { name: 'Jimmy Jedi' } })
9
10  describe("swapiGetter", () => {
11    afterEach(jest.clearAllMocks);
12
13    test("should return the first entry from the api", async () => {
14      const result = await swapiGetter(1);
15      expect(result).toBe("Jimmy Jedi");
16      expect(mockAxios.get).toHaveBeenCalledTimes(1);
17    });
18  });
```

Listing 2.9: Example of mocking the axios module and invoking the mock asynchronously (from [43])

Jest does provide other mechanisms for mocking, but for simplicity's sake we will focus on these.

Chapter 3

Design and Implementation

In this work, we propose QuiP, a novel method for performance testing Javascript front-end web-app code before deployment. QuiP extends the Jest testing framework to support performance optimization by estimating runtime using performance models, as shown in Fig 3.1. This chapter provides the details of the design and implementation of QuiP.

Section 3.1 explains why QuiP was designed as an extension, rather than a modification of Jest. Section 3.2 describes the architecture of QuiP, which uses an Observer pattern. Unit tests trigger the RuntimeContext subject within QuiP, which notifies the RuntimeMonitor observer about mock calls. The SerialRuntimeMonitor, an extension of the RuntimeMonitor class, processes non-async code to facilitate this estimation. The implementation details of this approach are illustrated in Section 3.3.

Building on this, QuiP analyzes asynchronous code automatically, which is novel in the area of performance testing before deployment. Section 3.4 explains how QuiP achieves this by implementing the AutoRuntimeMonitor, which leverages the Async hooks API. The API is used to detect relationships between the execution contexts of Javascript code under test, in order to form a timeline of asynchronous dependencies. This involves a systematic search for deviations in Javascript execution context relationships from asynchronous dependencies, which is a novel contribution of this project as well.

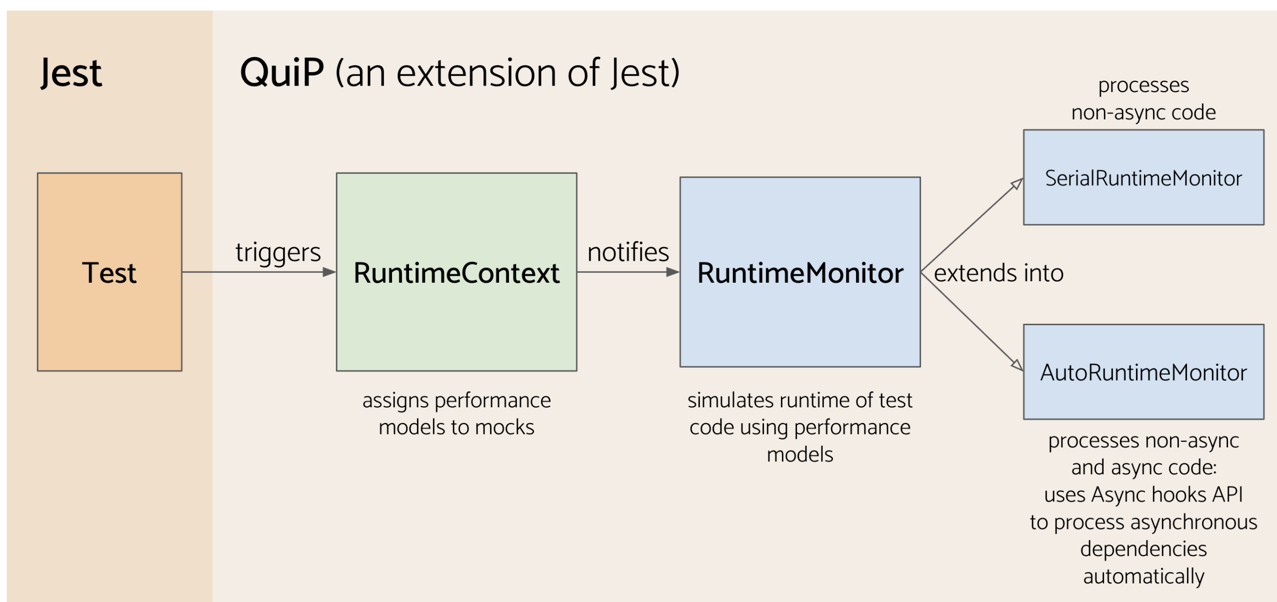


Figure 3.1: An overview of QuiP's implementation

3.1 Project Design: Extending over Modifying Jest

In order to incorporate performance testing into Jest unit tests, QuiP could be set up as a modified version of Jest. The advantage of this approach is that being able to modify Jest code provides more flexibility to QuiP's functionality. The internal structure of a mock can be changed to include information on its performance, such as performance models, to enable testing before deployment. Meanwhile, Jest testing implementations can be adjusted by adding virtual runtime estimations to the runtime of tests. These extensions have the additional benefit of ensuring that the existing Jest boilerplate only needs to be modified minimally for users to track performance in their unit tests.

However, the downside of this method is its maintainability. When the original Jest package is updated with new features or bug fixes, users may face a dilemma of choosing between using the updated version of Jest or sticking with QuiP. Given that QuiP's implementation differs from Jest, applying the updates to QuiP can be a complex and time-consuming task.

Therefore, an alternate approach considered is to set up QuiP as an extension of Jest. For example, a wrapper class or method can be created to store both the mock and its corresponding performance model, to add onto Jest's tests. This means that no matter what changes are made to Jest, only its API will affect the functionality of QuiP. This may be more challenging to implement, with no edit access to Jest's original code. Additionally, users may need to write more boilerplate, which reduces the convenience of the performance tests. However, workarounds can be devised to account for both functionality and user experience.

Hence, to avoid issues of maintainability, QuiP is set up as an extension of Jest. The rest of this chapter describes how its implementation enables performance testing before deployment is possible, even without edit access to the Jest repository.

3.2 Architecture

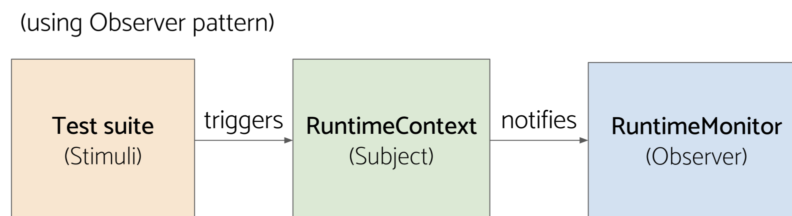


Figure 3.2: Architecture for QuiP

Before the system under test is fully deployable, its measured runtime in real time may not be an accurate indicator of its final performance. This is because for each piece of code tested, its collaborators may not be deployable, and running the code under test may take a shorter time as it skips over the runtime of its collaborators. Hence, the collaborators could be replaced with mocks containing information on their estimated performance. During each mock call, estimations of collaborator response time can be made, and added as virtual time to the real runtime of the code under test. The simulated runtime of the code under test now becomes the sum of the real time taken for the code to execute, and the total virtual runtime of mock calls.

Hence, the architecture for QuiP (as shown in Figure 3.2) is designed with the initial intuition of tracking mock calls. This requires QuiP to set up a listener for mock calls. Therefore, an observer pattern can be used for QuiP's implementation. Tests are considered stimuli. Within each test suite, a RuntimeContext object is created to act as the subject. Its role is to notify its RuntimeMonitor, the observer, about mock calls. The RuntimeMonitor ultimately uses this information to calculate the total estimated runtime for each piece of test code, which can be returned to the user.

3.3 Runtime Estimation

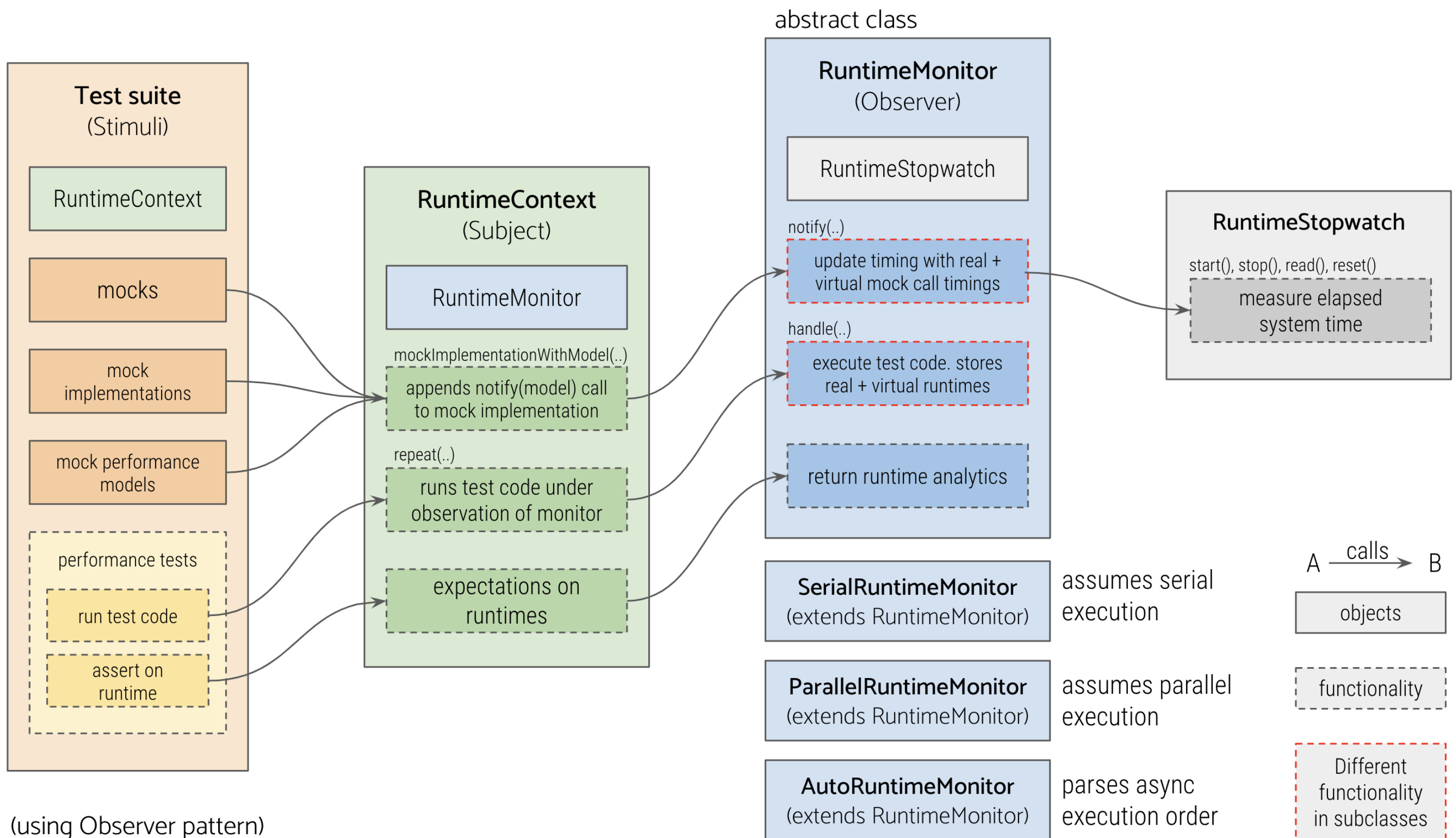


Figure 3.3: Flowchart of QuiP's algorithm

In order to calculate the total estimated runtime, both the real time and virtual time elapsed during code execution must be measured. The real time elapsed can be measured using system clock time, but virtual time measurement is far less straightforward. Since the virtual component of the test runtime is the product of the number of mock calls made and their respective virtual response times, the RuntimeMonitor must know when each mock is called in the test code, and what its estimated response time is.

Figure 3.3 illustrates QuiP's algorithm to estimate runtime, and will be explained throughout the following subsections.

3.3.1 Investigating when mocks are called

The need of the RuntimeMonitor to know when a mock is called during a test program gives rise to a difficult design choice, as QuiP cannot change Jest internally. Therefore, we are unable to change the behaviour of a Jest mock to notify the RuntimeMonitor in response to a new mock call. However, we can take advantage of the Jest API. For each mock created, calling `.mock` on it returns an object with three members: `calls`, `instances`, `and` `invocationCallOrder`. The number of times a mock is called during each test can be inferred from `calls`, and the order of mock calls can be obtained from `invocationCallOrder`. On the other hand, the specific timing of mock calls and how much real time is between them is still unknown. It will be possible to sum up the total virtual runtime, assuming a serial (non-async) execution, but constructing a detailed timeline is not possible.

Therefore, it will make more sense to automatically append a `notify(mock)` call to each mock implementation. To do so, a wrapper function is defined in the RuntimeContext. This function takes in the user implementation and appends the `notify(mock)` call to it, before defining the combined function as the actual mock implementation. This is shown in Listing 3.1. As a result, the RuntimeMonitor can be notified exactly when mock calls are being made, and the user does not need to write any additional input to achieve this (as shown in Listing 3.2).

```
1    mockImpWithModel(mock, imp) {
2        const implementationWithNotif = () => {
3            this.monitor.notify(mock);
4            return imp();
5        };
6        mock.mockImplementation(implementationWithNotif);
7    }
```

Listing 3.1: Abbreviated function definition of RuntimeContext.mockImpWithModel() leaving model out

```
1    ctx.mockImpWithModel(mockAdd, () => (5));
```

Listing 3.2: Example of user code assigning an implementation to a mock using QuiP

3.3.2 Estimating mock response time with performance models

Performance models are used in PerfMock to estimate response times of mocks. Each model returns a numerical estimation of the response time of its associated mock. As previously explained in Section 2.4 of the Background, these models can be applied to estimate mock response times during front-end unit testing before deployment.

Performance models can provide an estimate of the effect of code changes on runtime. This paper primarily focuses on using these models to estimate the effect of code changes, enabling performance optimization, rather than striving for maximum accuracy. Although their accuracy may be limited before deployment, these models can be enhanced with empirical data gathered post-deployment. Moreover, since the estimates are made in virtual rather than real time, performance can be measured with low turnaround time both before and after deployment. Therefore, developers can benefit from their estimations without needing highly precise models.

3.3.3 Attaching models to mocks

Hence, users should be allowed to assign performance models to mocks. Once again, this is difficult due to the fact that Jest mocks are not defined with a model field, and their implementation cannot be changed. The performance model of a mock cannot be directly stored within it.

Therefore, the most elegant way to give the RuntimeMonitor access to a mock's performance model is to directly pass the model to it during the mock call notification. This can be done by the RuntimeContext, when calling `notify(mock, model)` on the RuntimeMonitor in response to mock calls in the test code (illustrated in Listing 3.3). Users should pass the mock's performance model to the RuntimeContext when defining the mock implementation, as shown in Listing 3.4.

```
1    mockImpWithModel(mock, model, imp) {
2        const implementationWithNotif = () => {
3            this.monitor.notify(mock, model);
4            return imp();
5        };
6        mock.mockImplementation(implementationWithNotif);
7    }
```

Listing 3.3: Brief function definition of RuntimeContext.mockImpWithModel()

```
// returns random duration between MAX and MIN (excluded)
const randPerfModel = (run, args) =>
  (Math.floor(Math.random() * (MAX - MIN) ) + MIN + 1);

runtimeCtx.mockImpWithModel(mockAdd, randPerfModel, () => (5));
```

Listing 3.4: Example of user code assigning a performance model and an implementation to a mock

3.3.4 Adding up runtimes

At this point, the RuntimeMonitor needs to measure the real time elapsed, and combine that with the virtual time estimation. The real time elapsed can be measured by the difference in system time before and after the test code is run. In order to do so, the user could manually calculate the difference within their test code, as shown in Listing 3.5. However, this introduces code repetition and may not be accurate. Instead, the code under test could be passed to the RuntimeContext as a lambda, as shown in Listing 3.6.

```
test("should add up input", async () => {
    const runs = 10;
    const start = process.hrtime.bigint();
    for (let i = 0; i < runs; i++) {
        registrar.addUp(1, 2);
    }
    const end = process.hrtime.bigint();
    expect((end - start) / runs).toBeLessThan(10);
});
```

Listing 3.5: Example of manually measuring real time elapsed during code execution

```
test("should add up input", async () => {
    const runs = 10;
    await runtimeCtx.repeat(runs, () => {
        registrar.addUp(1, 2);
    });
    expect(runtimeCtx.runtimeMean()).toBeLessThan(10);
});
```

Listing 3.6: Example of passing code under test to RuntimeContext

RuntimeMonitor measures the duration of code execution with a RuntimeStopwatch, which encapsulates the reading of system time. Different time units can be supported by converting the output of `process.hrtime.bigint()`, which is in nanoseconds appropriately. The time units supported, seconds, milliseconds and nanoseconds, are represented by static TimeUnit objects. Users can be allowed to customize the time units of test output by passing the corresponding TimeUnit to the RuntimeContext constructor.

Within the `repeat()` call in the RuntimeContext, the code under test is passed to its RuntimeMonitor by calling `handle()` on it. `handle()` starts the RuntimeStopwatch, and runs the code. Because of `mockImpWithModel()`, `notify()` is added to the implementations of mocks. During code execution, Jest runs the implementations of mocks, instead of the implementation of the object mocked. This triggers the `notify()` calls embedded in their implementations, which notifies the RuntimeMonitor that the mock has been called.

Upon each `notify()` call, the RuntimeMonitor reads the real time elapsed between the previous mock call (or the start of the program) and the current mock call. It also passes the number of times the mock has been called so far and the arguments it was called with (both obtained from `mock.calls`) to the mock's performance model, which will then return the virtual response time of the mock. The real time elapsed between each mock call and the virtual response time of the mock calls form the full timeline of a serial execution. A visualized example of a timeline generated is shown in Figure 3.4. Finally, the RuntimeMonitor is able to return the total timing, as well as the timeline (as a JSON), after executing the code under test. Commonly used data, such as the mean runtime or nth percentile

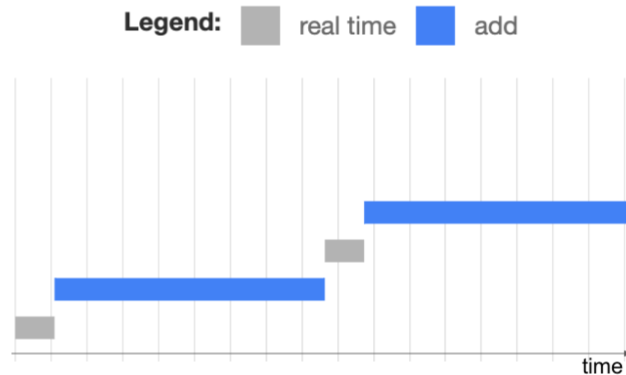


Figure 3.4: Example of a visualized serial code execution timeline

of runtime across multiple runs of the same code under test, can also be calculated by the RuntimeMonitor. It can be retrieved by the runtimeContext and returned to the user to be asserted upon, like in line 6 of Listing 3.6.

3.3.5 Visualizing timelines

It may be difficult for the user to view the timeline data in the form of a JSON, especially considering the number of test runs and complexity of test timelines. Hence, a timeline visualizer tool is implemented to display timelines in the form of graphs. This can help greatly to interpret the timelines and optimize the critical path of the timeline. The test suite automatically saves results to a `.txt` file, which can be uploaded to the visualizer.

A screenshot of the timeline visualizer is shown in Fig 3.5. Graph representations of the timelines are automatically created with random unique colours for each mock, with a legend displaying the colours for the mocks. These graphs are generated with the Chart.js API [44]. Hovering over the lifetimes of each mock call displays information on its mock call name, and estimated start and end time. In the event where a large number of test runs are made for the same code under test, it may not be practical for the viewer to look through every timeline generated. Instead, the timeline visualizer creates an overview of the runs by only printing the timelines for the runs with longest, shortest and median timing.

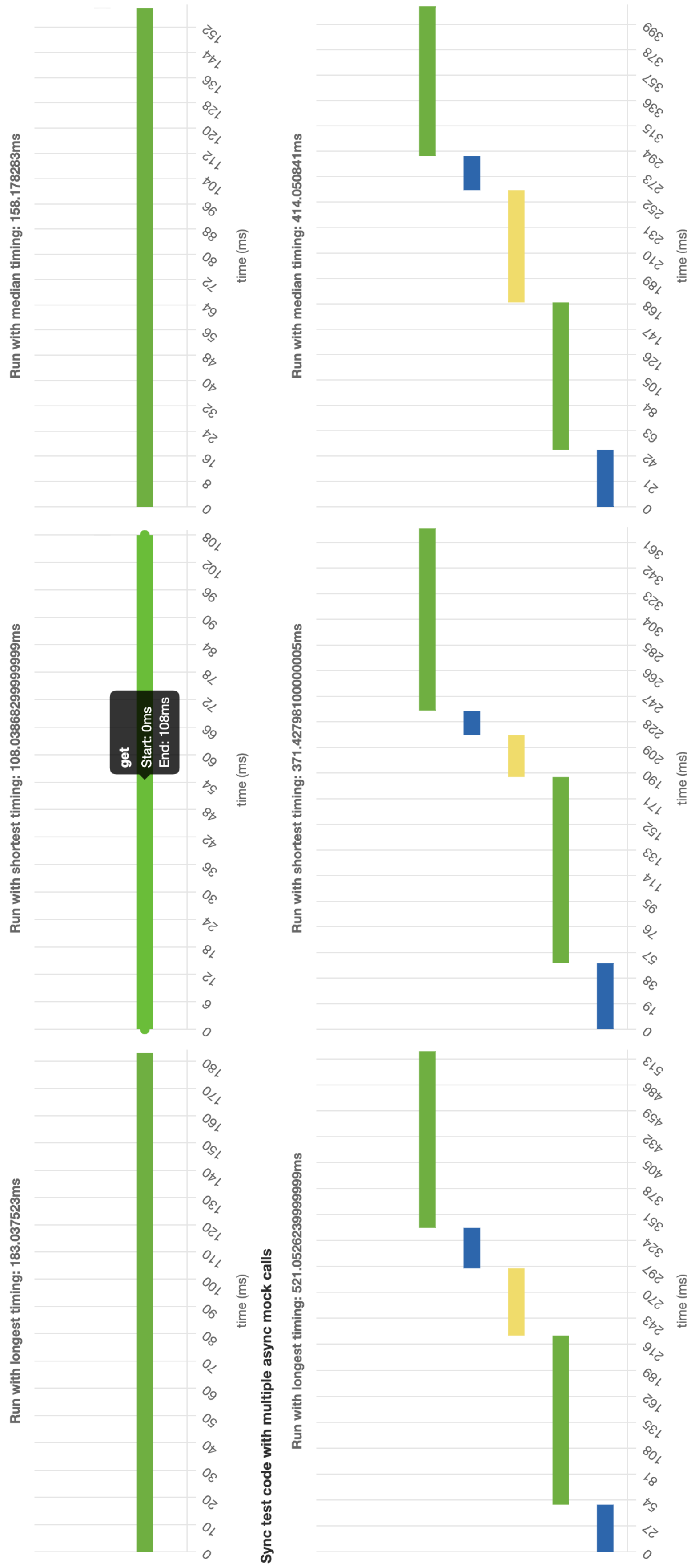


Figure 3.5: Screenshot of timeline visualizer tool displaying example test results

3.4 Asynchronous Execution Analysis

The previous RuntimeMonitor design can only work with non-async executions, since they arrange mock calls by their sequence in real time. It is referred to as the SerialRuntimeMonitor from now on. With async executions, even if two mock calls are triggered sequentially in real time, it is unclear if they are called serially or concurrently, as environmental factors can account for minute differences in their start time. For example, given a piece of code that executes two get() calls concurrently, the SerialRuntimeMonitor assumes that they are scheduled serially, because one call is made slightly after the other. This could lead to huge discrepancies in how the timings of the test program's mock calls are added up.

Using Async hooks for dependency analysis

In order to simulate the execution of test programs involving asynchronous mock calls, the dependencies between mock calls need to be analyzed. The AutoRuntimeMonitor is implemented to perform this analysis automatically, using the Async hooks API. The central idea of this novel implementation is to take advantage of the unique `asyncIds` that Async hooks assigns to each execution context related to asynchronous resources.

At any time, `executionAsyncId()` returns the `asyncId` assigned to the current execution context, while `triggerAsyncId()` returns the `asyncId` of the execution context that triggered the current one. This `asyncId` is the `triggerAsyncId` of the current execution context. The current context is created due to the triggering context finishing its execution (such as sequential function calls). Therefore, there is a high level of correlation between asyncIds, and the dependencies between asynchronous function calls.

Once mappings between mock calls and `asyncIds` are stored, it is possible to track the dependencies between mock calls. This can be used to determine if the execution of two mocks is concurrent.

Since Async hooks assigns `asyncIds` to execution contexts, there are different cases of async test program execution that cause deviations in `asyncId` assignments from actual async dependencies. Due to insufficient documentation of the relationship between execution contexts and asynchronous code, these cases are found through systematic testing of the Async hooks API. They are described in Table 3.1.

Table 3.1: Table showing various cases of asynchronous code that cause deviations in `asyncId` assignments

Characteristic	Categories	Description
Executions	Serial Concurrent	Whether mocks are called serially or concurrently (or a mix)
Mocks	Async Non-async	Whether mock implementations are async (i.e. return a promise)
Function calls	Non-nested Nested	Whether all mock calls are made within function under test, or made by other functions that are then called by the function under test

The way that the AutoRuntimeMonitor analyzes `asyncIds` needs to be modified for these cases. Section 3.4.1 describes how the base case, async mocks with different types of execution in a non-nested function call, is processed. Section 3.4.2 extends this by mixing async and non-async mocks, and Section 3.4.3 explains how nested function calls are analyzed. More information is required to analyze a few edge cases with greater accuracy, given that the code under test is not ready for

deployment. Additionally, it is possible that the edge cases listed are not exhaustive. However, most edge cases currently considered are successfully analyzed after modifications to the algorithm for the base case.

3.4.1 Async mocks with different types of execution

The base case assumes that all mocks have asynchronous implementations. Before AutoRuntimeMonitor.repeat() runs the code under test, an async hook is created, and starts listening for events. Each mock call causes a call to asyncNotify() on the AutoRuntimeMonitor. asyncNotify(), as shown in Fig 3.6, calculates the ending time of the mock call.

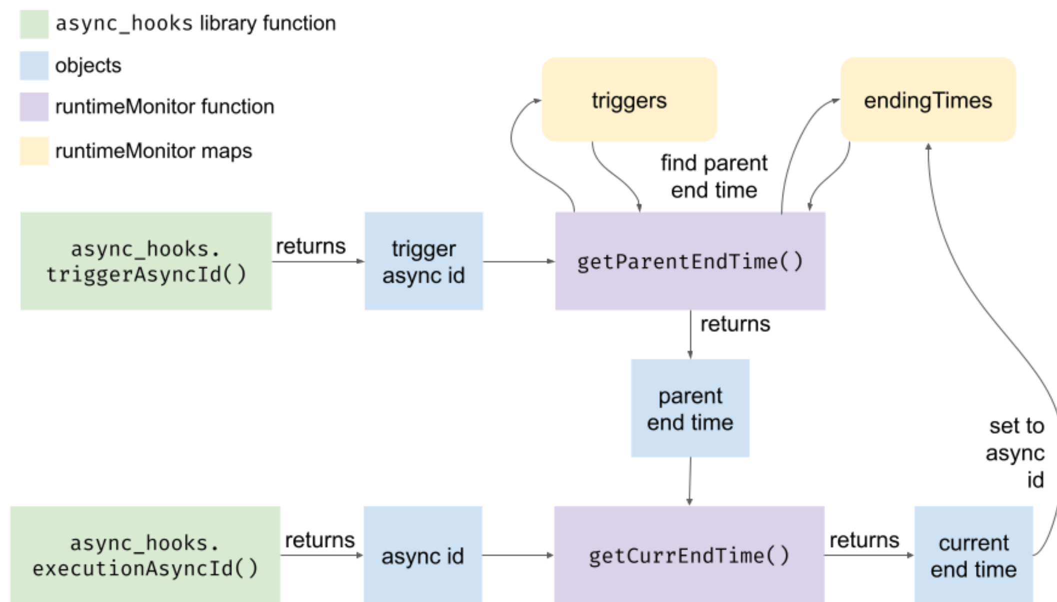


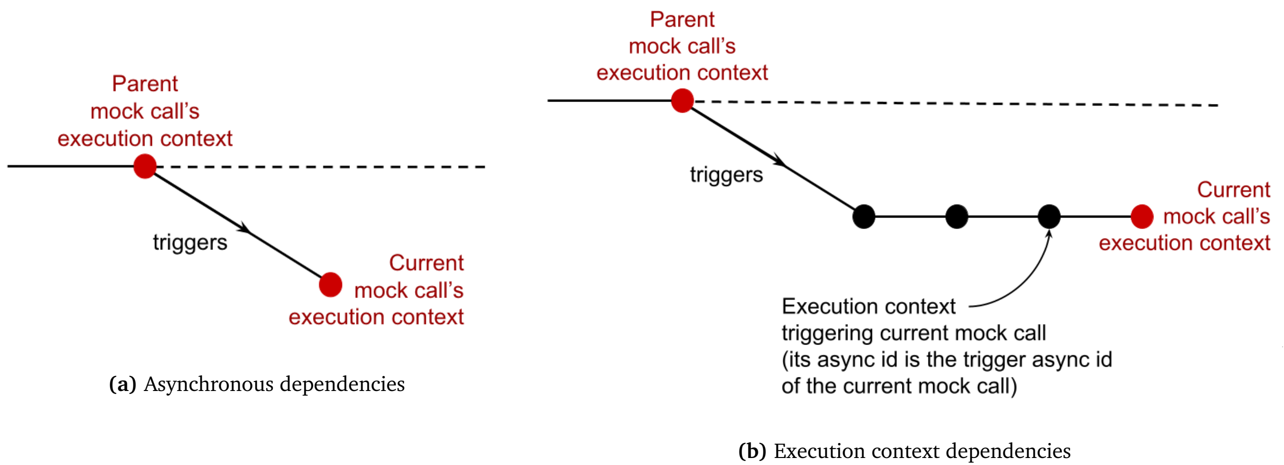
Figure 3.6: Diagram of flow within `asyncNotify()` function

Within the asyncNotify() function, the `asyncId` and `triggerAsyncId` of the current execution context (corresponding to the mock call) are retrieved with Async hooks. The ending time of the current mock call can be estimated using these ids.

First, the end time of the most recent mock call in the current mock call's dependency chain (i.e. the current mock's parent) needs to be found. This is handled by the getParentEndTime() function. The parent's end time is to be used as the start time of the current mock's execution. Therefore, a map, `endingTimes`, of `asyncId` to the ending time of its corresponding mock call should be stored. The id of the current mock's parent can be used to query the map for its ending time.

However, the parent mock call's `asyncId` may not necessarily be the `triggerAsyncId` of the current execution context, as shown in Fig 3.7. This is because there may be execution contexts between the parent and the current mock, and the `triggerAsyncId` of the current mock is the `asyncId` of the execution context triggering it. In order to find the id of the parent, the init function of the async hook is defined to store each execution context's `triggerAsyncId` in a local map, `triggers`. The `asyncId` of the parent can be found by tracing each id's trigger until an id is found having a corresponding ending time stored in endingTimes.

The ending time of the parent mock call can thus be retrieved from `endingTimes`. This is used to estimate the ending time of the current mock in getCurrEndTime(). The estimated response time of the mock is the sum of the real time elapsed since its parent mock call and the virtual time generated



(a) Asynchronous dependencies

(b) Execution context dependencies

Figure 3.7: The async dependencies of the parent and the current mock call, compared to their execution contexts

by its performance model. This should be added to the parent's ending time to obtain the ending time of the current mock call.

Thereafter, the end time of the mock call can be set to its `asyncId` in the endingTimes map. This process repeats upon each mock call in the code under test, until the execution timeline and total estimated runtime are obtained. Examples of executions that can be correctly analyzed by this AutoRuntimeMonitor are shown in Figure 3.8. Get and put are both async mocks. The performance model assigned to the mocks for the tests are random number generators (within a predefined range), which explains the inconsistencies in their estimated runtimes.

(a) Serial executions

(b) Concurrent executions

(c) A mock call triggering two concurrent mock calls, with a real time delay before one of the concurrent mock calls

Figure 3.8: Examples of various executions correctly analyzed by the base case of the AutoRuntimeMonitor

Concurrent mocks with children edge case

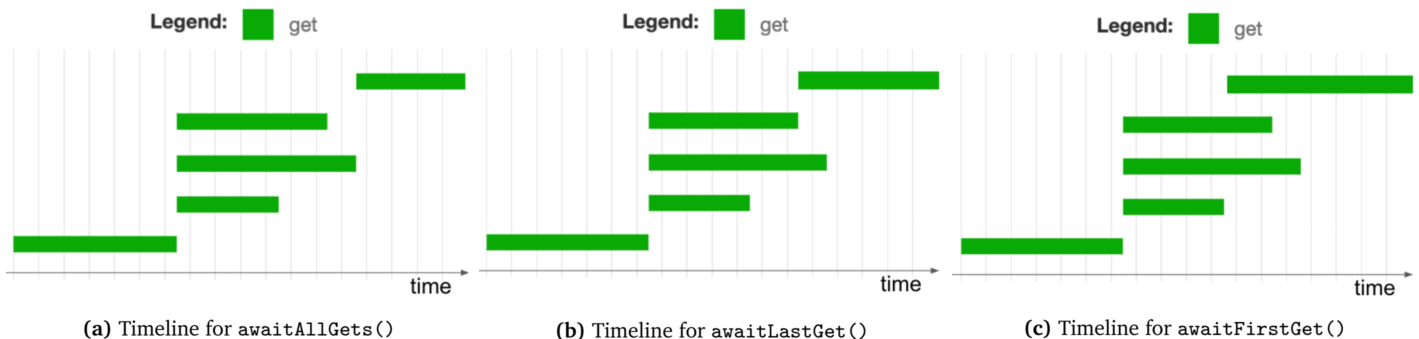
However, a problematic edge case occurs when mocks with children are called concurrently. Concurrent mock calls are made in the same execution context, and are assigned the same `asyncId`. Since mock call dependencies are tracked by `asyncId`, and mock call ending times are stored by `asyncId`, this can lead to conflicts when concurrent mock calls' children look up their parents' end times.

There are two ways to handle the storage of ending times under the same `asyncId`: storing the longest ending time so far (according to virtual time), or storing the most recently occurring ending

time (according to real time). Both of these strategies are suboptimal in different situations. This is illustrated through the example code in Listing 3.7. Timelines that are accurate to the actual asynchronous dependencies in the `awaitAllGets()`, `awaitLastGet()`, and `awaitFirstGet()` function calls are shown in Fig 3.9.

```
1  // concurrently calls get 3 times
2  async getIdTogether(id) {...}
3  // serially calls get 2 times
4  async doubleGet(id) {...}
5
6  // concurrently calls 3 gets. Last get called upon the completion of all 3 concurrent
      get calls
7  async awaitAllGets(id) {
8      await this.getIdTogether(id);
9      const get4 = await axios.get(`https://swapi.dev/api/people/${id}/`);
10 }
11 // concurrently calls 3 gets. Last get called upon the completion of the last get
     call
12 async awaitLastGet(id) {
13     const get1 = axios.get(`https://swapi.dev/api/people/${id}/`);
14     const get2 = axios.get(`https://swapi.dev/api/people/${id}/`);
15     const get3And4 = this.doubleGet(id);
16     return Promise.allSettled([get1, get2, get3AndPut]);
17 }
18 // concurrently calls 3 gets. Last get called upon the completion of the 1st get call
19 async awaitFirstGet(id) {
20     const get1And4 = this.doubleGet(id);
21     const get2 = axios.get(`https://swapi.dev/api/people/${id}/`);
22     const get3 = axios.get(`https://swapi.dev/api/people/${id}/`);
23     return Promise.allSettled([get1, get2andPut, get3]);
24 }
```

Listing 3.7: Example of code under test where the non-async mock has more than one child



(a) Timeline for `awaitAllGets()` **(b)** Timeline for `awaitLastGet()` **(c)** Timeline for `awaitFirstGet()`

Figure 3.9: Accurate timelines according to asynchronous dependencies in example functions from Listing 3.7

Table 3.2 shows the parent ending times of the final get, get4, when the two ending time storage strategies are used in various situations. Correct ending times (according to asynchronous dependencies) are highlighted in green. Both of these methods work when the ending time that they store coincides with the ending time of the child call's parent. However, storing the longest ending time is always correct for the general case where the child awaits all the concurrent get calls. Meanwhile, storing the most recent ending time is correct for the general case where the child is awaiting the most recent get call.

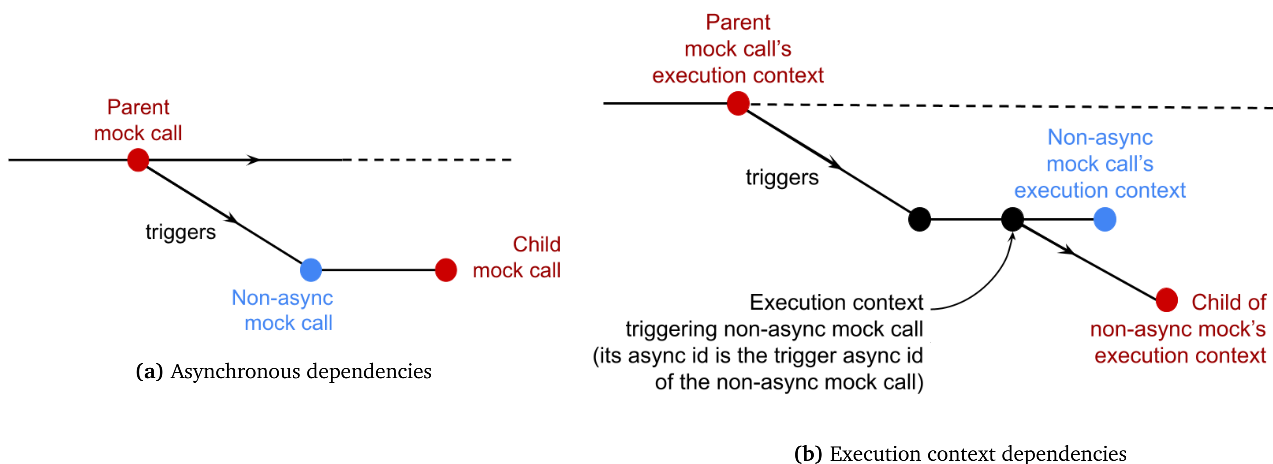
We chose to store the longest ending time, assuming that subsequent calls awaiting all previous concurrent calls might be more common than only the most recent concurrent call being awaited. However, this means that the analysis produced by AutoRuntimeMonitor may not be entirely accurate for the listed scenarios.

Table 3.2: Table showing accuracy of ending storage strategies in various situations

Situation		Parent ending time according to storage strategy		
Parent(s)	**Longest get call**	**Longest ending time**	**Most recent ending time**	**Actual parent ending time**
All get calls	get3	get3	get3	get3
	get2	get2	get3	get2
Most recent get call (get3)	get3	get3	get3	get3
	get2	get2	get3	get3
Not most recent get call (get1)	get3	get3	get3	get1
	get2	get2	get3	get1

3.4.2 Async and non-async mocks with different types of execution

The next edge case to consider is the use of both async and non-async mocks. The mock implementations of async mocks are async functions that return a Promise which may contain its output. Meanwhile, the mock implementations of non-async mocks do not return promises. Since the Async hooks API specifically tracks the lifetimes of execution contexts related to asynchronous resources, it is unable to track non-async resources accurately. During a non-async mock call, the `asyncIds` and `triggerAsyncId` returned would not follow the same conventions as those assigned to async mock calls.



(a) Asynchronous dependencies

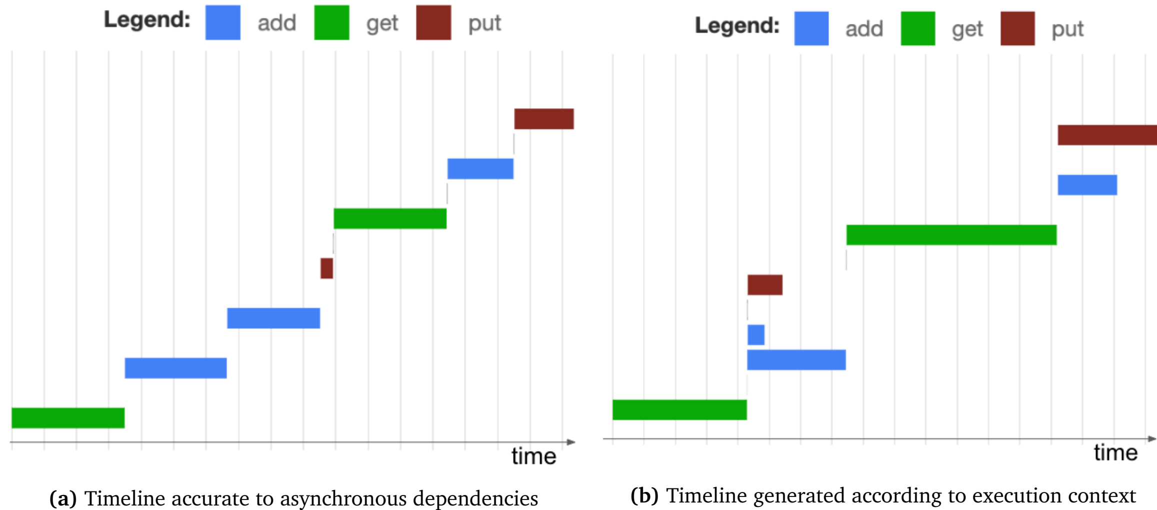
(b) Execution context dependencies

Figure 3.10: The async dependencies between the parent of the non-async mock call, the non-async mock call, and its child, compared to their execution contexts

Experimentation confirms that the `asyncId` assigned to the execution context for a non-async mock call is skipped over when identifying the `triggerAsyncId` of its child. This could be attributed to the fact that the non-async mock does not initialize an asynchronous execution context, and thus the execution contexts of its children are considered to be triggered by the execution context of the most recent asynchronous parent. This is visualized in Figure 3.10.

During the actual execution of the code under test, the child of the non-async mock call needs to wait for its completion. Therefore, it is dependent on the non-async mock call. However, the `triggerAsyncId` of the child is the `triggerAsyncId` of the non-async mock call, rather than its `asyncId`. This means that the trigger of the child's execution context is the execution context of the most recent async mock call. It can be seen that these mock calls' execution contexts do not directly reflect their dependencies.

Since the child of a non-async mock recognizes the `triggerAsyncId` of the non-async mock as its own `triggerAsyncId`, its corresponding `notify()` call does not have access to the `asyncId` of the non-async mock. When the child is retrieving its parent end time, it retrieves the ending time of the most recent async mock, instead of the ending time of the non-async mock (its actual parent). This means that the start time of the non-async mock's child is recognized to be the same as that of the non-async mock. The AutoRuntimeMonitor will identify them to be executing concurrently, when that is not possible with non-async code. This can be seen in Figure 3.11, where `add` is a non-async mock and `get` and `put` are async mocks.



(a) Timeline accurate to asynchronous dependencies

(b) Timeline generated according to execution context

Figure 3.11: Timeline of a serial execution of async and non-async mocks

Hence, the behaviour of the AutoRuntimeMonitor needs to be modified to analyze the asynchronous dependencies of non-async mocks. A new `notify()` function is set up to handle non-async mocks separately. In order for the child of the non-async mock to retrieve the ending time of the non-async mock as its parent ending time, the child must be either given access to the `asyncId` of the non-async mock, or the ending time of the non-async mock.

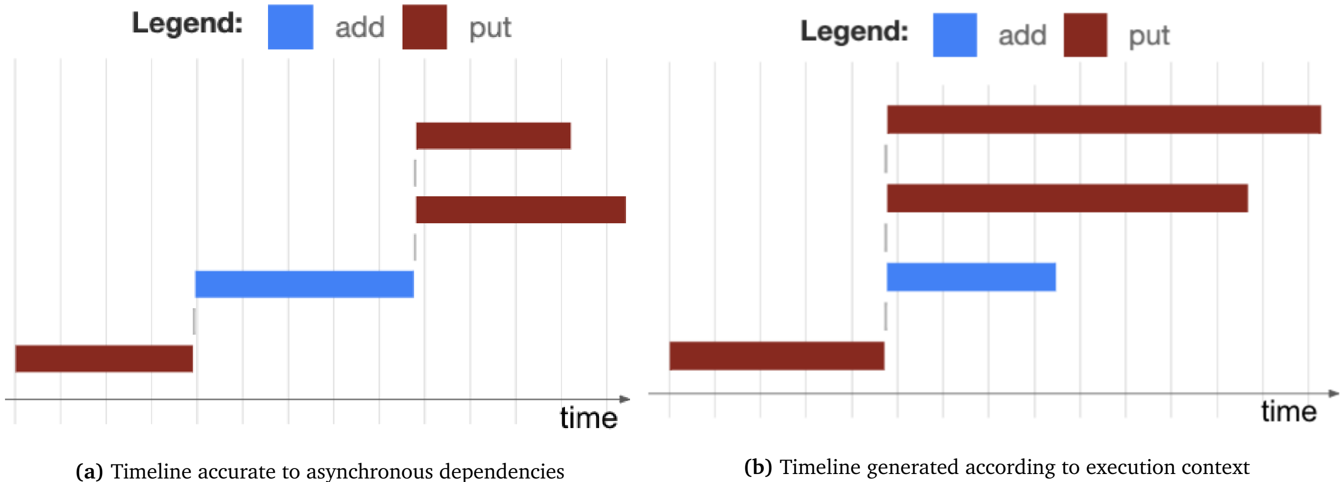
Giving the child of the non-async mock access to its id

The first approach considered is to give the child of the non-async mock access to the id of the non-async mock. This cannot be done during the `asyncNotify()` or `notify()` call of the child, as there are no ways to distinguish the child of a non-async mock call from the child of an async mock call. The only difference between these are their `triggerAsyncIds`, but there is no other information available to determine if a mock call's `triggerAsyncId` is actually reflective of its dependency. Therefore, the AutoRuntimeMonitor will not know when to modify the child's behaviour to account for a non-async parent.

Hence, the `notify()` function needs to be modified to leave subsequent mock calls access to its id. For example, its id could be stored in a local variable, to be read by its child. However, this approach is problematic, as it cannot be determined which of its subsequent mock calls (in real time) are actually its children and needs to read its id. Consider the code in Listing 3.8, where `axios.put` is an async mock, and `add` is a non-async mock. The AutoRuntimeMonitor's base case produces the timeline in Fig 3.12b when analyzing it, even though a more accurate timeline would be the one in Fig 3.12a. In modifyId, `add` has more than 1 child - `put1` and `put2`. It is erroneous to assume by default that only the first mock call succeeding the non-async mock call in real time is its child. The method is thus inoperable in this case.

```
1 async modifyIds(p) {
2     await axios.put(`https://swapi.dev/api/people/`, {id: modifiedId, name: "test"});
3     var modifiedId = add(p, 1);
4     let put1 = axios.put(`https://swapi.dev/api/people/`, {id: modifiedId, name: "
          test"});
5     let put2 = axios.put(`https://swapi.dev/api/people/`, {id: modifiedId, name: "
          test"});
6     await Promise.allSettled([put1, put2]);
7 }
```

Listing 3.8: Example of code under test where the non-async mock has more than one child



(a) Timeline accurate to asynchronous dependencies

(b) Timeline generated according to execution context

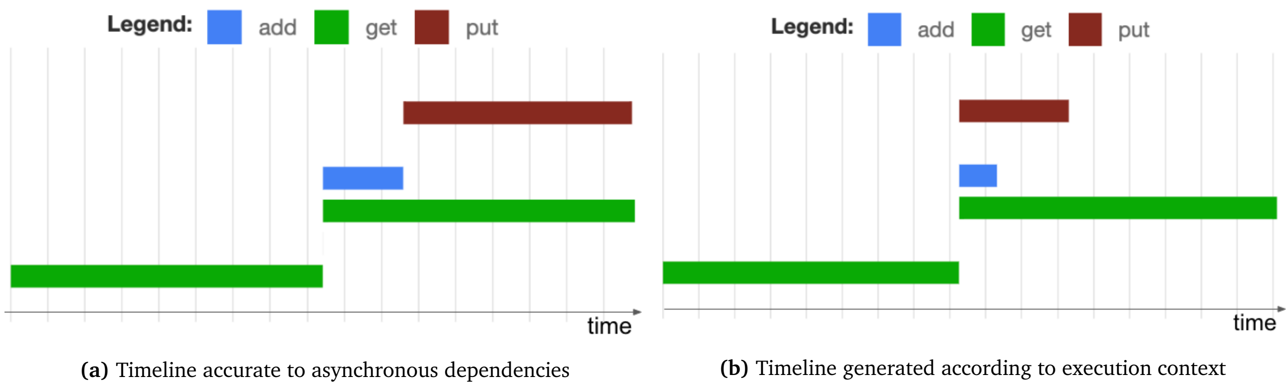
Figure 3.12: Timeline of execution of code under test from Listing 3.8

Giving the child of the non-async mock access to its end time

The other approach considered is to give the child access to the non-async mock's end time. For the same reasons as above, this cannot be done in the `asyncNotify()` or `notify()` call for the child. Storing the end time as a local variable would also fail in some cases, as shown above. Instead, the end time should be stored during the `notify()` call in a way that only the child of the non-async mock can access.

In order to do so, we consider that by default, the parent end time of each mock call is retrieved with its parent id. This parent id is found by tracing the trigger ids of the mock call's `triggerAsyncId` until an id is found with a corresponding ending time. We know that the non-async mock call and its child share a common `triggerAsyncId`. Therefore, during the `notify()` call for the non-async mock call, the estimated ending time of the non-async mock could be set for both its `asyncId` and its `triggerAsyncId`. In this way, when its child retrieves its parent end time using the non-async mock's `triggerAsyncId`, the correct end time is returned.

A concern with this method is whether the parent timings of other mock calls are affected, since the non-async mock call sets its ending time to an id that is not its own. Earlier mock calls (in real time) would not be affected, as their end times have already been saved to the timeline output. This can be seen in Fig 3.13, where `add` is a non-async mock and `get` and `put` are async mocks.
Later mock calls (in real time) with the same `triggerAsyncId` as the non-async mock call can be considered in two broad categories: children of the non-async mock call and children of the parent of the non-async mock call. Children of the non-async mock call are meant to take on the end time of the non-async mock call as their parent end time, leaving children of the parent of the non-async mock call as the problematic group.



(a) Timeline accurate to asynchronous dependencies

(b) Timeline generated according to execution context

Figure 3.13: Timeline of execution of code under test from Listing 3.8

However, if the children of the parent of the non-async mock call are called later in real time than the non-async mock call is, they should start after the non-async mock call ends. Non-async mock calls, by definition, cannot be executed concurrently with any other function calls. It is impossible for any other mock calls to share a parent end time with it, as this would mean that they start at the same time and are concurrent. Originally, these children would recognize their parent end time as the end time of the parent of the non-async mock call. Now, their parent end time would be set as the end time of the non-async mock call, which is accurate to their async dependencies.

Therefore, the final approach chosen to resolve this edge case is to give children of non-async mocks access to the non-async mock's end time, by setting this end time to its `triggerAsyncId` as well. This modified AutoRuntimeMonitor is used to generate the accurate timelines illustrated in Figures 3.11a, 3.12a and 3.13a.

3.4.3 Nested function calls with different types of execution

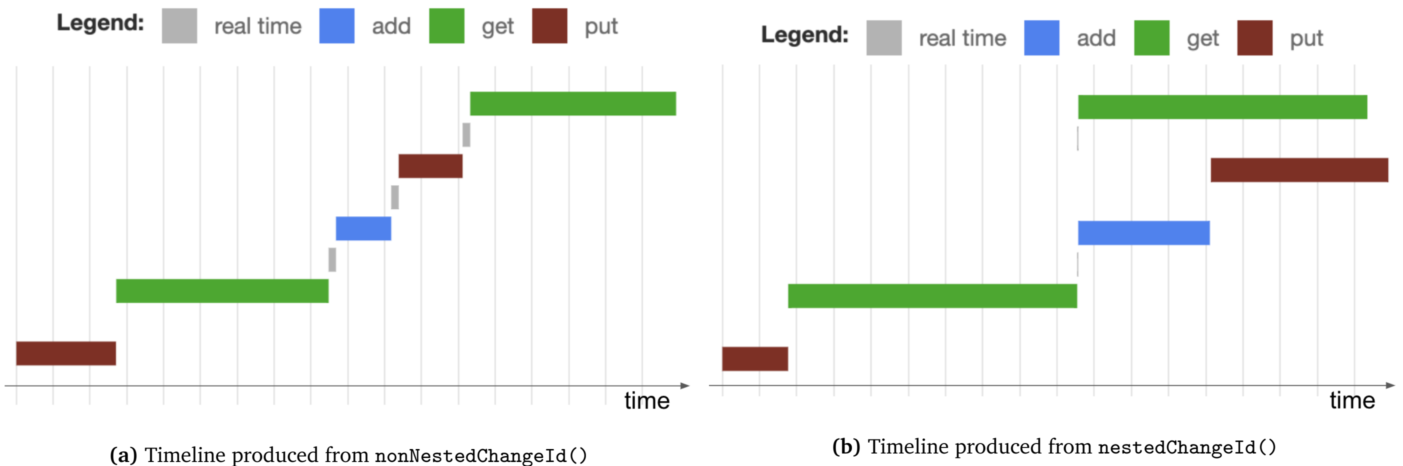
Finally, the edge case of nested function calls also causes conflict between its asynchronous dependencies and execution contexts. A nested function call refers to mock calls encapsulated within a separate function, which is called by the function under test. This is opposed to the function under test making all mock calls directly within itself. An example can be seen in Listing 3.9. Assuming the `axios` module is mocked in tests, `add`, `axios.get` and `axios.put` are mock calls. `nestedChangeId()` calls the nested function, `changeId()`, which performs a `get` mock call and a `put` mock call. Meanwhile, `nonNestedChangeId()` performs the `get` mock call and `put` mock call directly within itself, without calling the nested `changeId()` function.

```
1  async changeId(oldId, change) {
2      let get = await axios.get(`https://swapi.dev/api/people/${oldId}/`);
3      let newId = add(oldId, change);
4      await axios.put(`https://swapi.dev/api/people/`, {id: newId, name: get.data.name
           });
5  }
6  async nestedChangeId(id) {
7      await axios.put(`https://swapi.dev/api/people/`, {id: id, name: "test"});
8      await this.changeId(id, 1);
9      await axios.get(`https://swapi.dev/api/people/${1}/`);
10 }
11 async nonNestedChangeId(id) {
12     await axios.put(`https://swapi.dev/api/people/`, {id: id, name: "test"});
13
14     let get = await axios.get(`https://swapi.dev/api/people/${id}/`);
15     let newId = add(id, change);
16     await axios.put(`https://swapi.dev/api/people/`, {id: newId, name: get.data.name
           });
17
18     await axios.get(`https://swapi.dev/api/people/${1}/`);
```

```
19 }
```

Listing 3.9: Example of code under test where `nestedChangeId()` includes a nested function call, and `nonNestedChangeId()` performs the same mock calls without nested function calls

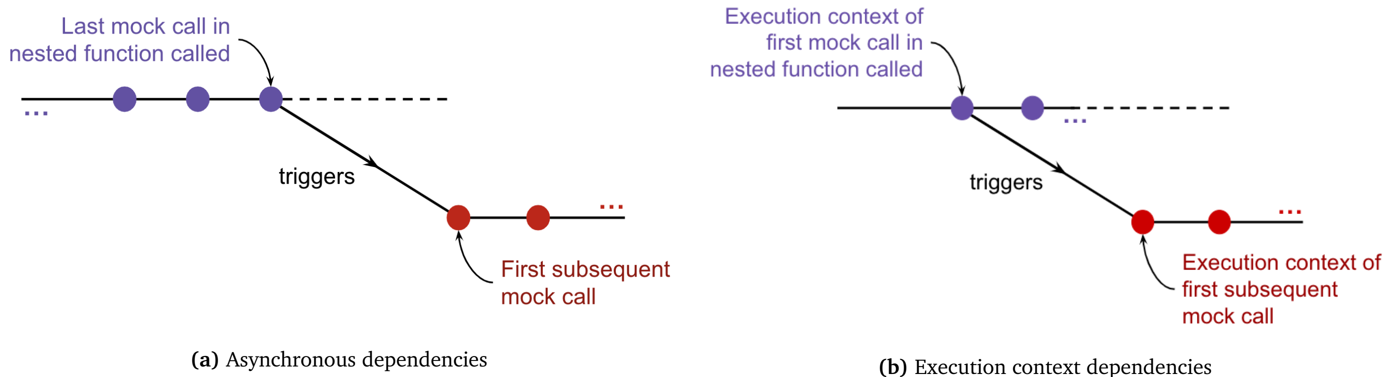
Practically speaking, both `nestedChangeId()` and `nonNestedChangeId()` perform the same mock calls, with the same asynchronous dependencies (both of these functions execute their mock calls serially). However, according to the AutoRuntimeMonitor, the relationships between their execution contexts are different. This results in different timelines being produced, as shown in Fig 3.14. Clearly, only the timeline produced from `nonNestedChangeId()` is accurate to their asynchronous dependencies.



(a) Timeline produced from `nonNestedChangeId()`

(b) Timeline produced from `nestedChangeId()`

Figure 3.14: Timelines of execution of code under test from Listing 3.9



(a) Asynchronous dependencies

(b) Execution context dependencies

Figure 3.15: The async dependencies between the a nested function call and subsequent mock calls, compared to their execution contexts

In the timeline produced from the nested function, shown in Fig 3.14b, the parent of the get mock call following changeId() is found to be the first mock call in changeId(). However, it should be the last mock call in changeId(). This deviation in its execution context dependencies from its async dependencies is shown in Fig 3.15. Further experimentation confirms the general trend where mock calls following a nested function call recognize the first mock call in the nested function as its parent, rather than the last. This is regardless of the number of mock calls in the nested function, and whether the subsequent mock call belonged in a nested function itself.

A possible explanation is that the function execution context of the nested function is the execution context of the first mock call in the nested function. The execution context of the mock call following

the nested function is created due to the nested function's execution context finishing its execution. Therefore, the execution context of the mock call recognizes the nested function's execution context as its trigger, and is thus triggered by the execution context of the first mock call in the nested function.

Therefore, the behaviour of the AutoRuntimeMonitor needs to be modified to account for the asynchronous dependencies of nested functions. In order to achieve this, two problems need to be solved. Firstly, the last mock call in the nested function and its end time needs to be retrievable by the succeeding mock call. This would be the actual parent of the succeeding mock call of the nested function. Secondly, the AutoRuntimeMonitor needs to be able to distinguish between mock calls succeeding entire nested function calls, and mock calls succeeding the first mock call in the nested function call. These two cases appear to have the same parent, but only the parent end time of mock calls succeeding the nested function call would need to be modified.

Retrieving the last mock call in the nested function and its end time

Since the mock call succeeds the nested function call, its associated `asyncNotify()` or `notify()` call always occurs later in real time than the `asyncNotify()` or `notify()` calls of the mock calls within the nested function. Therefore, the ending time of the succeeding mock call's "real" parent (according to asynchronous dependencies), the last mock call of the nested function, should have already been stored in the endingTimes map (as this is done during `asyncNotify()` or `notify()`). It would be possible for the succeeding mock call to retrieve this ending time and use that as its parent end time instead, as long as it has access to the associated `asyncId`.

Hence, the AutoRuntimeMonitor should first find the latest descendant of the original parent of the incoming mock call. If the incoming mock call was succeeding a nested function, this descendant would be the last mock call of the nested function, and the actual parent of the incoming mock call. To achieve this, the `asyncId` of each child should be set to the `asyncId` of each parent in a map named visitedParentIds. The latest descendant of a parent is the last value obtained when repeatedly querying the map with the parent's `asyncId` until no entries can be found.

At this point, given the `asyncId` of the last mock call of the nested function, its end time can be retrieved from the endingTimes map. However, this should not be indiscriminately set as the parent end time of the incoming mock call. For example, if there are two mock calls running concurrently in the nested function after the first mock call, the parent of the second concurrent mock call would be set to the first concurrent mock call. However, its parent is actually meant to be the first mock call of the nested function. Therefore, the AutoRuntimeMonitor needs to differentiate between a mock call succeeding a nested function, and a mock call succeeding the first mock call in the function, and only change the parent of the former.

Distinguishing between mock calls succeeding a nested function call and mock calls succeeding the first mock call in the nested function call

In order to adjust only the parent of the mock call succeeding the nested function call, instead of the mock call succeeding the first mock call in the nested function, the AutoRuntimeMonitor needs to be able to distinguish between these two cases. This can be done based on the real time elapsed between the first mock call in the nested function and the succeeding mock call. A mock call succeeding the nested function would be called later in real time than a mock call succeeding the first mock call in the nested function.

Therefore, the real time that each mock call is made at should be stored in a map named realTimes. The real times of the first mock call in the nested function, the last mock call in the nested function, and the incoming mock call (which may be succeeding the nested function, or succeeding the first mock call in the nested function) can then be retrieved. The real time elapsed from the first mock call in the nested function to the last mock call, and the real time elapsed from the first mock call in

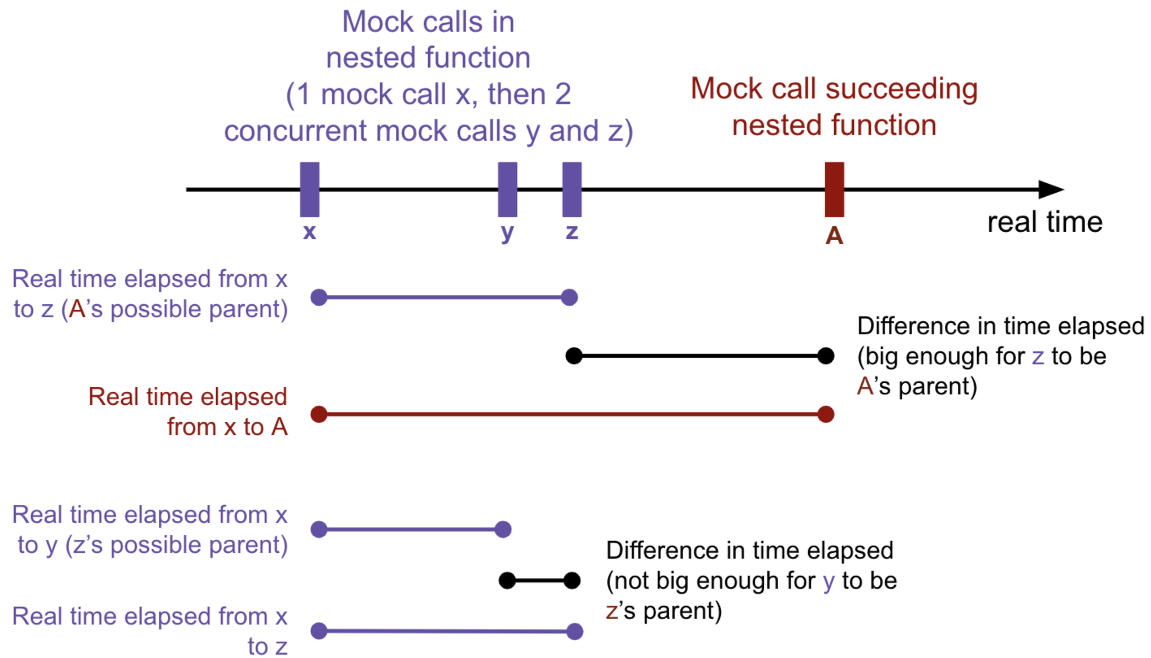


Figure 3.16: Diagram of real times that mock calls are made at for a nested function call with a succeeding mock call

the nested function to the incoming mock call can be calculated. The difference between the times elapsed can be simplified to the real time elapsed from the last mock call in the nested function to the incoming mock call. The bigger this difference, the more likely that the incoming mock call is called after the last mock call in the nested function, rather than after the first mock call in the nested function, concurrent with the second. These differences are illustrated by Fig 3.16. The first difference visualized corresponds to the case where the incoming mock call is actually succeeding the entire nested function, while the second difference visualized corresponds to the case where the incoming mock call is actually succeeding the first mock call in the nested function.

The AutoRuntimeMonitor needs to decide based on this difference in time elapsed if the incoming mock call is called after the first or last mock call in the nested function. However, this difference may vary greatly between devices, based on environmental factors such as hardware, or resource contention. It may be more user-friendly to allow users to input a threshold on the difference in time elapsed, according to their own needs, instead of stipulating a global threshold that is seldom accurate.

Therefore, the AutoRuntimeMonitor takes in `assumeSerialThreshold` in its constructor. If the real time elapsed from the last mock call in the nested function to the incoming mock call exceeds this threshold, the AutoRuntimeMonitor assumes that the incoming mock call actually succeeds the entire nested function, rather than the first mock call in the nested function, and changes its parent to the last mock call in the nested function. This may cause some non-determinism in the timelines generated across test runs, if the real time elapsed varies above and below the threshold. However, once a suitable `assumeSerialThreshold` is found through some experimentation, timelines that are more accurate to the asynchronous dependencies of the code under test can be produced. For example, the AutoRuntimeMonitor was able to produce the timeline in Fig 3.14a after this modification, while it produced Fig 3.14b before.

Chapter 4

Usage

This chapter demonstrates how QuiP can be used to test the performance of synchronous and asynchronous Javascript web-app code before deployment. This is done through applying QuiP to the development of the checkout page of a book-selling web-application, using a Model-View-Controller pattern. To simulate how a developer might use QuiP, standard test-driven development (TDD) practices are used. When it comes to the performance aspect of the tests, the testing procedure introduced by PerfMock is incorporated into the TDD development cycle.

Section 4.1 provides a brief overview of the architecture of the checkout web-application. Section 4.2 explains the process of writing performance tests with QuiP, by going through the process of writing two example tests for the checkout view. However, the performance assertions within the tests may not always pass. Section 4.3 demonstrates how these assertions can be made to pass, and how the performance of the code under test can be improved in the process. Finally, Section 4.4 shows how QuiP remains useful even after the web-application becomes deployable, since its performance models can be refined with empirical data.

4.1 Checkout architecture

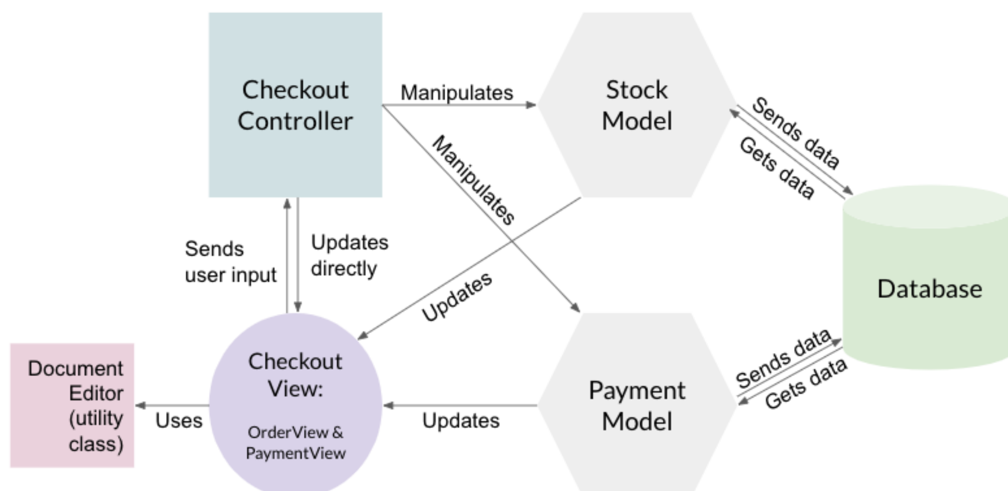


Figure 4.1: Architecture for the checkout page

Fig 4.1 shows an overview of the architecture of the checkout web-application. The view is exposed to the user, and forwards input to the controller. The controller processes this input by manipulating the stock and payment models, which can interact with the database. Following this, any relevant

updates are sent to the view to be displayed in response to the earlier user input. Since the purpose of QuiP is to conduct performance tests for front-end web-app code, we focus on tests written for the checkout view, the front-end of the checkout web-app, during this demonstration.

4.2 Writing performance tests with QuiP

In test-driven development, tests are written for desired behaviour before feature implementation is done to make the tests pass. Therefore, we begin by writing tests expecting the desired behaviour of the checkout view, before implementing the view and running the tests. The first step is to set up the test context, and to mock the collaborators of the view. This is described in Section 4.2.1. Then, the tests for the view are written, as explained in Section 4.2.2, before they can be run.

4.2.1 Setting up test context and mocks

The user must first initialize a RuntimeContext for the performance tests to be run inside of. This can be done with the code in Listing 4.1. The RuntimeContext constructor takes in 3 arguments: the `asyncMode`, the `timeUnit`, and (optionally) the `assumeSerialThreshold`. The `asyncMode` determines the type of RuntimeMonitor used - a SerialRuntimeMonitor, a ParallelRuntimeMonitor, or an AsyncRuntimeMonitor. The `timeUnit` determines the unit of time that test runtime is measured and presented in - nanosecond, millisecond, or second. The `assumeSerialThreshold` is used by the AsyncRuntimeMonitor to resolve ambiguities in execution contexts in the case of nested function calls in the test code.

```
1  const runtimeCtx = new RuntimeContext(AsyncMode.Auto, TimeUnit.nanosecond, 5000);
```

Listing 4.1: Code initializing a new RuntimeContext object to test the CheckoutView

Following this, the collaborators of the view need to be mocked, since the purpose of the tests is solely to test the functionality of the view. A snippet of the code for this is shown in Listing 4.2. Jest is used to create the mock object itself and the user can optionally define a custom implementation for the mock. For the RuntimeMonitor to include the estimated response time of the mock in the total runtime of tests, the user has to assign a performance model to the mock. To do so, the user includes the model as an argument when calling `mockWithModelAsync()` (for async mock implementations), or `mockWithModel()` (for non-async mock implementations) on the RuntimeContext.

In this example, a simple random number generator within a specified range is used as the performance model, instead of a more complex one. The reason for this choice is that the main focus of this demonstration, as well as the entire work, is to improve performance with runtime estimations rather than the accuracy of the estimations.

It is important that the user assigns the implementation to the mock by including the implementation as an argument in `mockWithModelAsync()` or `mockWithModel()`. If `mockImplementation()` from the Jest library is used instead to assign the implementation to the mock, `mockWithModelAsync()` and `mockWithModel()` will overwrite the mock's implementation with a `notify()` call to the RuntimeMonitor. However, if the user intends to exclude the mock from the calculation of virtual runtimes, it is safe to create the mock and assign an implementation to it with vanilla Jest.

```
1  const mockController = require("../../exampleCode/caseStudy/checkoutController.js");
2  jest.mock("../../exampleCode/caseStudy/checkoutController.js");
3
4  const getQtyImp = () =>
5      Promise.resolve(orderView.updateQuantities(QUANTITIES));
6  function produceRandPerfModel(max, min) {
7      const randPerfModel = (run, args) =>
8          (Math.floor(Math.random() * (max - min) ) + min + 1);
9      return randPerfModel;
10 }
11 runtimeCtx.mockWithModelAsync(mockController.getQuantities,
12                               "controller.getQuantities",
```

```
13                                      produceRandPerfModel(1500000, 1000000),
14                                      getQtyImp);
```

Listing 4.2: Code to mock the getQuantities function of the checkout controller

The rest of the collaborators of the checkout view are mocked in a similar way. These collaborators include the other functions of the `checkoutController` object, and the functions of a `documentEditor` object (a utility class that encapsulates any functions on the document object, for editing the html of the front-end). These can be seen in greater detail in the `__tests__\caseStudyTests` folder in the project git repository.

4.2.2 Writing tests for the checkout view

We begin the implementation of the checkout view by writing tests for desired behaviour. Listing 4.3 shows how the test suite is set up, using Jest boilerplate. `clearContext()` needs to be run on the RuntimeContext between tests to remove the timeline data of the previous test. `writeResultsToFile()` saves the timeline data from all tests in the test suite into a `.txt` file at the path given.

```
1  describe("order info view", () => {
2      afterEach(() => {
3          orderView = new OrderView();
4          QUANTITIES.set("1", {name: "A Tale of Two Cities", qty: "1"});
5          QUANTITIES.set("5", {name: "Pride and Prejudice", qty: "4"});
6          QUANTITIES.set("110", {name: "Maurice", qty: "10"});
7
8          jest.clearAllMocks();
9          runtimeCtx.clearContext();
10     });
11
12     afterAll(() => {
13         runtimeCtx.writeResultsToFile("../orderTimelineData.txt");
14     })
15 }
```

Listing 4.3: Code to set up the test suite for the order information part of the checkout view

The first behaviour to implement is order quantity rendering. Therefore, a unit test needs to be written for the `renderQuantities()` function of the OrderView class. An example is shown in Listing 4.4. The unit test is written within the framework of a regular Jest test, but with the code under test passed to the RuntimeContext to be run, and with performance assertions carried out by QuiP. Calling `repeat()` on the RuntimeContext runs the code under test under observation of the RuntimeMonitor, which estimates the total runtime of each test run. If a performance test name is specified in the third argument of the call to `repeat()`, the timeline data from the test runs is saved by the RuntimeContext to be written into a `.txt` file when `writeResultsToFile()` is called. Following this, the test can make assertions about the performance of the code under test based on the needs of the web-application.

```
1      it("should display books and quantities selected", async () => {
2          await runtimeCtx.repeat(runs, async () => {
3              await orderView.renderQuantities();
4          }, "renders book quantities within average of 15000000ns")
5
6          // should query controller
7          expect(mockController.getQuantities).toHaveBeenCalledTimes(runs);
8          // should receive and display update from stock model
9          expect(orderView.quantities).toBe(QUANTITIES);
10         expect(mockDocumentEditor.addQtyToOrderTable).toHaveBeenCalledTimes(
               QUANTITIES.size * runs);
11
12         // should run within an average of 15000000ns
13         expect(runtimeCtx.runtimeMean()).toBeLessThan(15000000);
14         expect(runtimeCtx.runtimePercentile(100)).toBeLessThan(20000000);
15     });
```

Listing 4.4: A unit test for the renderQuantities() function of the order view

After writing the unit test, we implement the `renderQuantities()` and `updateQuantities()` functions in the OrderView class. `renderQuantities()` makes a request to the controller for the book quantities in the order. The mock of the controller calls `updateQuantities()` with the dummy `QUANTITIES` map, which is displayed by the mockDocumentEditor. This allows the functionality assertions in the test to pass.

However, the performance assertions in the unit test fail. The test output can be seen in Fig 4.2. The average runtime across the test runs is within the expected 15000000ns, or 15 ms, but the maximum runtime across the test runs exceeds the expected 20000000ns, or 20 ms. The visualized test timelines confirm this, as can be seen in Fig 4.3.

```
● order info view › should display books and quantities selected

  expect(received).toBeLessThan(expected)

  Expected: < 20000000
  Received:   47316301

    104 |         expect(mockDocumentEditor.addQtyToOrderTable).toHaveBeenCalledTimes(QUANTITIES.size * runs);
    105 |         expect(runtimeCtx.runtimeMean()).toBeLessThan(15000000);
  > 106 |         expect(runtimeCtx.runtimePercentile(100)).toBeLessThan(20000000);
        |                                                   ^
    107 |     });
    108 |
    109 |     // Serial async mock calls

    at Object.toBeLessThan (__tests__/caseStudyTests/checkoutViewTest.spec.js:106:51)
```

Figure 4.2: Test output of unit test for renderQuantities() function of the order view

Figure 4.3: Test timelines of unit test for renderQuantities() function of the order view

The quantities of each book need to be rendered before their prices can. Therefore, the test for rendering the prices of each book needs to call `renderQuantities()` first, before their corresponding prices can be rendered with the `renderPrices()` function. The test code to do so is shown in Listing 4.5, with functionality assertions removed for brevity.

```
1   it("should display price of each unique book selected", async () => {
2       await runtimeCtx.repeat(runs, async () => {
3           await orderView.renderQuantities();
4           await orderView.renderPrices();
```

```
5         }, "renders book prices within average of ns")
6
7         // should run within an average of 30000000ns
8         expect(runtimeCtx.runtimeMean()).toBeLessThan(30000000);
9     });
```

Listing 4.5: A unit test for the renderQuantities() and renderPrices() functions of the order view

These two render operations are expected to run within an average of 30000000ns, or 30ms, twice of the time `renderQuantities()` is expected (and managed) to run for. However, the performance assertions fail, as shown in Fig 4.4 and Fig 4.5. This may be due to the additional real-time delay between function calls.

```
● order info view › should display price of each unique book selected

  expect(received).toBeLessThan(expected)

  Expected: < 30000000
  Received:   30277305.2

    119 |         expect(orderView.prices).toBe(PRICES);
    120 |         expect(mockDocumentEditor.addPriceToOrderTable).toHaveBeenCalledTimes(QUANTITIES.size * runs);
  > 121 |         expect(runtimeCtx.runtimeMean()).toBeLessThan(30000000);
        |                                          ^
    122 |     });
    123 |
    124 |     // Parallel async mock calls

    at Object.toBeLessThan (__tests__/caseStudyTests/checkoutViewTest.spec.js:121:42)
```

Figure 4.4: Test output of unit test for running renderQuantities(), then renderPrices()

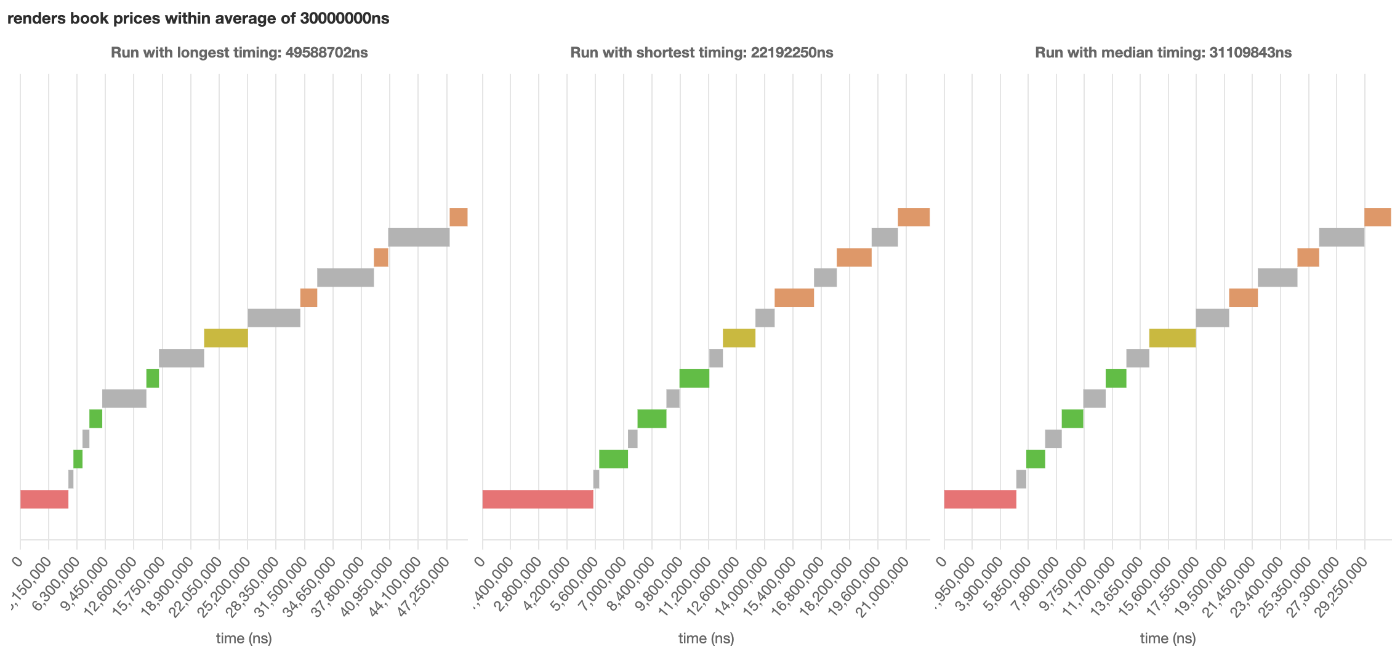


Figure 4.5: Test timelines of unit test for running renderQuantities(), then renderPrices(). Legend is the same as the one in Fig 4.3

4.3 Fixing failing performance tests

At this point, the performance unit test methodology introduced by PerfMock suggests several strategies to fix the failing performance tests: making the performance assertions more lenient, changing

the performance models of the mocks called, and editing the code under test [9]. Being able to visualize test timelines guides the developer in choosing an effective approach, as it is easy to determine which dependencies the longest sections of the critical path correspond to.

If the runtimes of the mock calls take up a majority of the test runtime, changing the performance models of the mocks may have the greatest impact in optimizing performance. If real time takes up the majority of the test runtime, making the mocks run faster would have a limited impact on the total runtime. Instead, the developer would have to either optimize the code under test, or increase the time limit in the performance assertions.

In the example in Listing 4.4, the performance assertions fail because the maximum runtime of the test runs is too long. It can be seen that in the run with the longest timing, shown in Fig 4.3, real time takes up the majority of the test runtime. Since the code under test already consists of just one function call and cannot be optimized further, it makes more sense to relax the time limit of the performance assertion for the test to pass.

Meanwhile, for the example in Listing 4.5, real time takes up a significant portion of the test runtime as well (as seen from the timelines in Fig 4.5. However, it is possible to optimize the average runtime of the test runs further by parallelizing the `renderQuantities()` and `renderPrices()` functions. This leads to the test results shown in Fig 4.6. The code under test is much faster and passes the performance assertion.

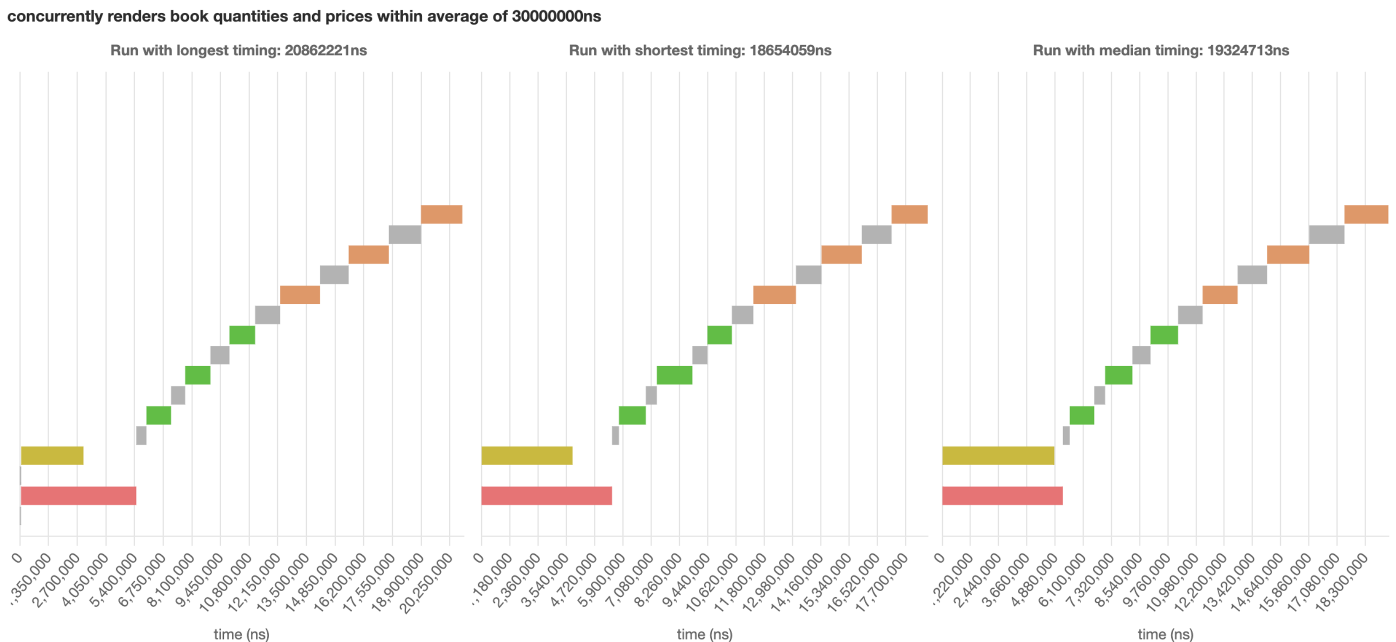


Figure 4.6: Test timelines of unit test for running `renderQuantities()` and `renderPrices()` concurrently. Legend is the same as the one in Fig 4.3

4.4 Refining tests with empirical data

Once the implementation of the checkout view reaches a deployable state, we are able to obtain empirical measurements of the response time of the checkout view's collaborators. This data can be used to refine the performance models assigned to the mocks of the collaborators.

In this case, we create a minimal viable product for the case study by completing the DocumentEditor class to populate the html of the checkout page, implementing basic functionality for the Checkout-

Controller, StockModel and PaymentModel classes, and linking the Model classes to a simple database hosted on Supabase [45]. Data for the response times of the DocumentEditor and CheckoutController is thus obtainable through logging while deploying the checkout page locally. The random number generator models are retained for simplicity, with the bounds of the numbers generated adjusted according to the runtime data of 10 deployments. In Table 4.1, the original assumed parameters of the mock are compared with parameters derived from empirical data.

Table 4.1: Comparison of original assumed model parameters with parameters derived from empirical data

Mocked contributors		**Model parameters (in ms, to 3s.f.)**			
Class	**Function**	**Assumed minimum**	**Assumed maximum**	**Actual minimum**	**Actual maximum**
CheckoutController	getQuantities	5	5.5	63.7	397
	getPrices	1.5	5	55.8	306
	changeQuantity	3	4	209	377
	verifyPaymentInfo	7	8	60.1	117
	chargePayment	6	7	0.1	1.4
	verifyPaymentWithBank	5	10	58.7	155
	getDeliveryDate	3	4	54.8	87.6
DocumentEditor	addQtyToOrderTable	1	1.5	0.1	0.3
	addPriceToOrderTable	1.5	2	0.1	0.2
	addPaymentStatus	0.6	0.7	0.1	0.2
	addDeliveryDate	0.6	0.7	0.1	0.2

After adjusting the mocks according to empirical data, different test timelines are returned from CheckoutView's unit tests. For example, Fig 4.7 shows the timelines produced for the unit test for running `renderQuantities()` and `renderPrices()` concurrently. The runtimes of the documentEditor mock calls are now too short relative to the checkoutController mock calls, and cannot be seen on the graphs unless hovered over. This is in contrast to the test timelines produced before the mocks were adjusted, as shown in Fig 4.6.

This test output remains highly useful for developers, even though other performance measurement tools based on real time measurements are now available after the web-app is deployed. It helps in visualizing the deployment timeline of the web-app and choosing an effective approach for performance optimization. For example, based on the output data, it is clear that the checkoutController dependency is the limiting factor of performance, rather than the documentEditor dependency. Moreover, using QuiP has the advantage of a low test turnaround time compared to deploying the actual web-app in real time, since mock runtimes are estimated in virtual time.

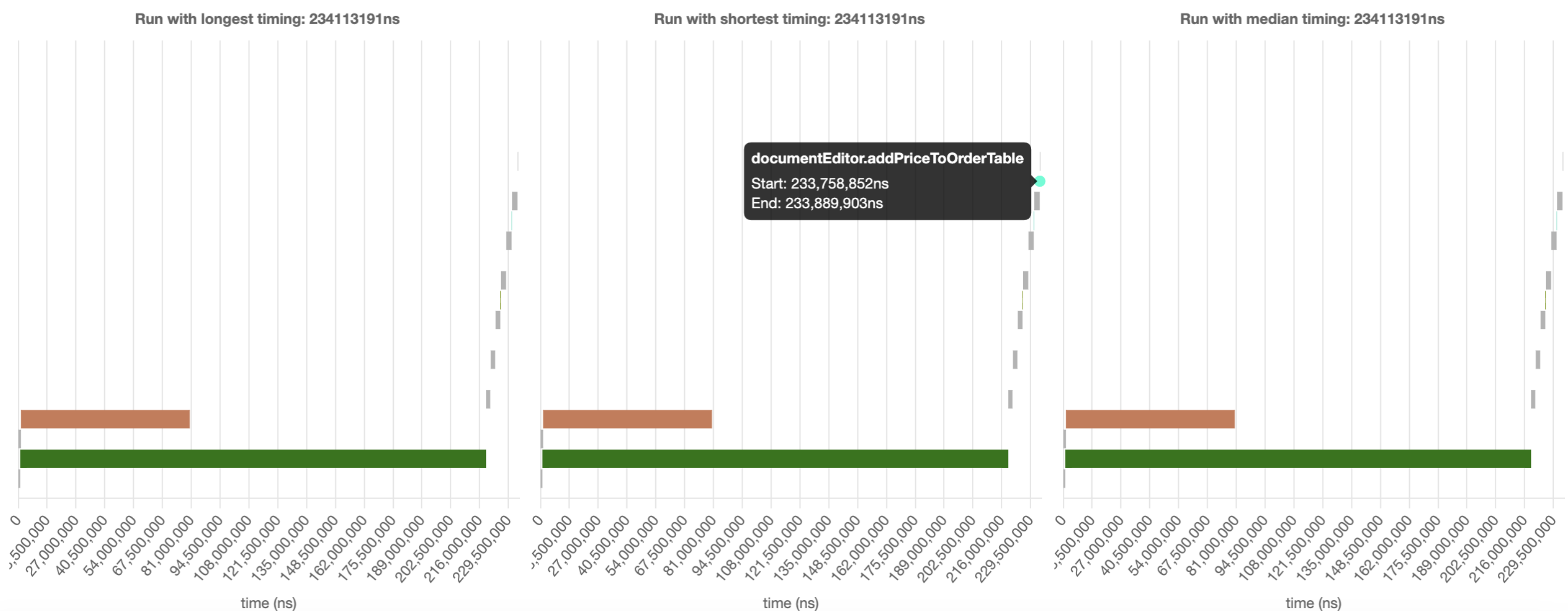


Figure 4.7: Test timelines of unit test for running `renderQuantities()` and `renderPrices()` concurrently, after mock models have been adjusted according to empirical data

Chapter 5

Evaluation

This section focuses on evaluating QuiP's accuracy and how effective it is in improving front-end performance. We consider its ability to estimate the effect of Javascript code on runtime before and after deployment, and its test turnaround time. The final product (integrating performance testing into unit testing, which can be carried out before deployment) should be compared to existing controls that represent current runtime simulation, performance testing, and unit testing methods.

1. DrAsync can be used to represent current runtime simulation tools, and it similarly relies on the Async hooks API like QuiP does.

2. Performance measurement tools that rely on real time measurements from deployed software can be used to represent current performance testing methods. Examples include logging the current system time, or Google Lighthouse. This is in contrast with QuiP, which estimates runtime with virtual time measurements from performance models.

3. Current unit testing methods do not conduct performance testing before the software system is deployable, as there are no obtainable real time measurements. This is in contrast with QuiP, which allows performance testing from the start of development.

In the process of comparing QuiP to the above controls, our evaluation aims to answer the following Research Questions(RQ):

RQ1	To what extent is QuiP capable of processing the dependencies in asynchronous test code?
RQ2	How accurate is QuiP in predicting the runtime of code under test?
RQ3	How accurate is QuiP in predicting the effect of changes in code under test on its runtime?
RQ4	How effective is QuiP in reducing the turnaround time of performance tests?

The following sections cover the evaluation and answers to the above questions. Section 5.1 verifies if QuiP can accurately process the various cases of asynchronous code present in the unit tests written for the checkout web-application's view, to answer **RQ1**. This is compared to the control of DrAsync, which represents current runtime simulation tools. Section 5.2 compares the predicted and measured runtimes of order display and payment making flows in the checkout web-application, to answer **RQ2**. The measured runtimes are obtained with real-time measurements using system time logging. Section 5.3 answers RQ3 by comparing the predicted and actual percentage change in runtime after a code change, where the actual runtime is measured by system time logging. Finally, Section 5.4 compares the test turnaround time of unit tests with performance testing in virtual time (using QuiP), performance tests in real time, and unit tests without performance testing. This is used to conclude how much time QuiP saves in performance testing, and how little additional time is required to incorporate performance testing in unit testing using QuiP, in answer to **RQ4**.

5.1 Evaluating asynchronous dependency parsing by QuiP (RQ1)

In order to evaluate QuiP's ability to analyze test code for asynchronous dependencies, we consider the various cases of asynchronous code, as listed in Table 3.1. In comparison, DrAsync, another tool for asynchronous dependency processing that makes use of the Async hooks API, does not account for these edge cases in its analysis.

We determine whether QuiP can effectively handle different combinations of these scenarios by simulating runtimes of unit tests written for the checkout view introduced in Section 4.1. We use QuiP's AutoRuntimeMonitor to do so, due to its ability to analyze asynchronous code.

5.1.1 Serial and concurrent executions

The first scenario analyzed involves different types of executions: serial, concurrent, serial followed by concurrent, and concurrent followed by serial executions. We aim to evaluate QuiP's ability to process each of these scenarios effectively. This can be done by examining how QuiP processes two test cases: the overall payment flow for a successful payment without bank verification, and the concurrent rendering of book quantities and price.

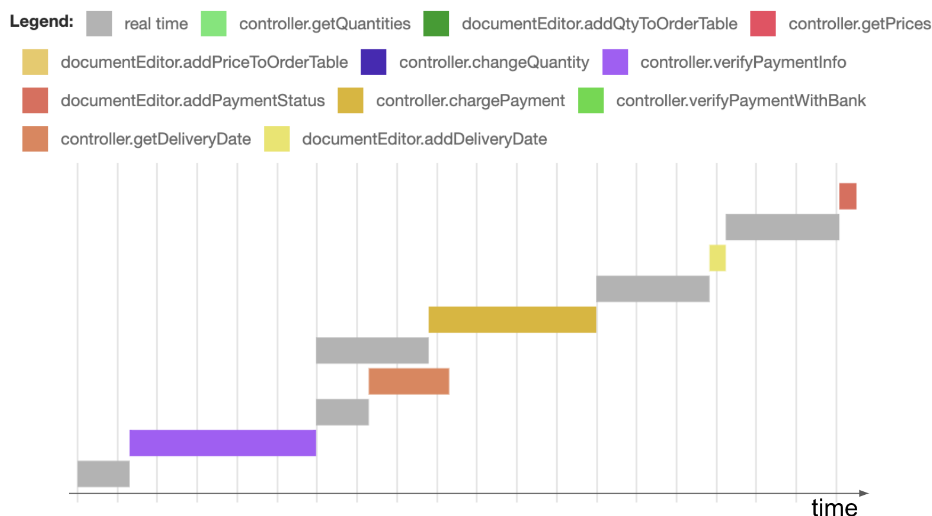


Figure 5.1: Timeline QuiP generates for the overall payment flow for a successful payment without bank verification

During the overall payment flow for a successful payment without bank verification, the various types of executions mentioned above are present. These are all accounted for by QuiP, as shown in the generated timeline in Fig 5.1. To invoke the payment flow, the test calls `paymentView.processPayment()`. The payment flow starts off as serial with a call to `controller.verifyPaymentInfo`, before calling `controller.getDeliveryDate` and `controller.chargePayment` in parallel. Notably, a delay in real time was introduced before the call to `chargePayment`, which is successfully captured by QuiP. After awaiting the settlement of both of these functions, two serial calls to `documentEditor.addDeliveryDate` and `documentEditor.addPaymentStatus` are made.

On the other hand, the test case for the concurrent rendering of book quantities and price shows that QuiP's analysis of concurrent followed by serial executions is not entirely correct. Within the test code, two calls to `orderView.renderQuantities()` and `orderView.renderPrices()` are made concurrently. A timeline that QuiP generates for this case is shown in Fig 5.2. It can be seen that two calls to `controller.getQuantities` and `controller.getPrices` are scheduled concurrently. Technically, the call to `controller.getQuantities` is settled earlier and it sets off calls to

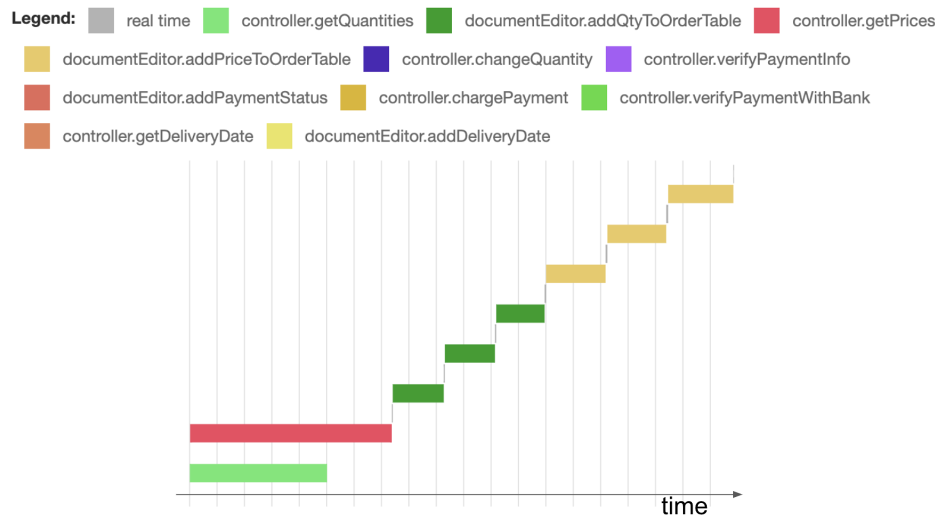


Figure 5.2: Timeline QuiP generates for the concurrent rendering of book quantities and price

`documentEditor.addQtyToOrderTable` before the call to `controller.getPrices` is settled. However, QuiP records the calls to `documentEditor.addQtyToOrderTable` as having started only after both `controller.getQuantities` and `controller.getPrices` are settled.

This behaviour, while not ideal, is expected and addressed in Section 3.4.1, which explains the different cases of concurrent followed by serial mock calls that QuiP can and cannot process.

5.1.2 Async and non-async mocks

Next, we aim to evaluate QuiP's ability to analyze asynchronous code involving both async and non-async mocks. In particular, it should analyze the asynchronous dependencies of mock calls succeeding non-asynchronous mock calls correctly, as implemented in Section 3.4.2. Within the test suite written for the checkout view, mocks of `controller` functions have asynchronous implementations, and mocks of `documentEditor` functions have non-asynchronous implementations. We can tell from the test case for the payment flow for a payment invalidated by the bank if QuiP is capable of processing code containing both async and non-async mocks.

In this test case, `paymentView.processPayment()` is called. Within this function, calls are made first to the async mocks `controller.verifyPaymentInfo` and `controller.verifyPaymentWithBank`. Following this, a call is made to the non-async mock `documentEditor.addPaymentStatus`, before another call is made to the async mock `controller.verifyPaymentWithBank`. Finally, this sets off a last call to the non-async mock `documentEditor.addPaymentStatus`. This is processed by QuiP as the timeline shown in Fig 5.3.

It can be seen that QuiP successfully processes the whole test case as a serial execution. It does not mistakenly recognize the second call to `controller.verifyPaymentWithBank` as a concurrent with the first call to `documentEditor.addPaymentStatus`. This behaviour has been explained and successfully implemented in Section 3.4.2, which makes it safe for users to invoke calls to both async and non-async mocks.

5.1.3 Nested and non-nested function calls

Lastly, we consider QuiP's ability to process nested function calls. Once an appropriate `assumeSerialThreshold` is set, it should be able to accurately process code where mock calls are made within a separate function, which is then called within the code under test. This is further explained in Section 3.4.3. Every

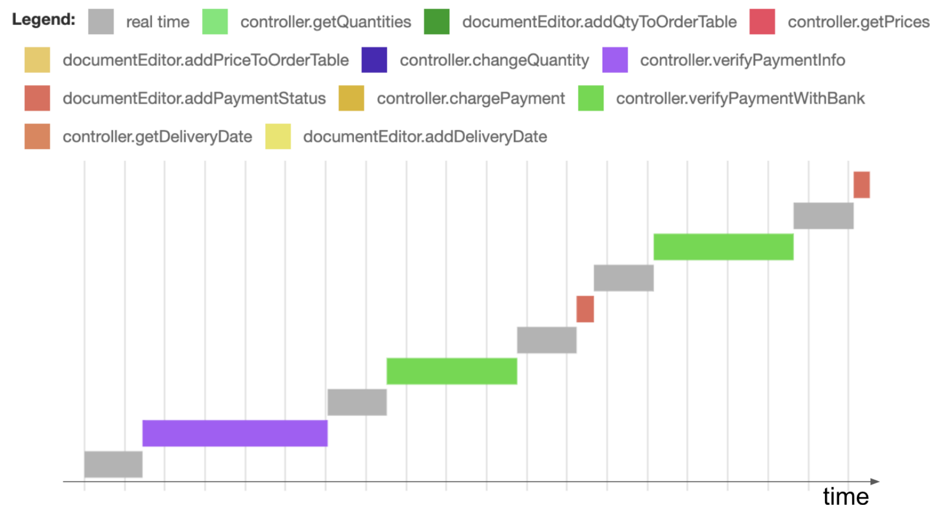


Figure 5.3: Timeline QuiP generates for the payment flow for a payment invalidated by the bank

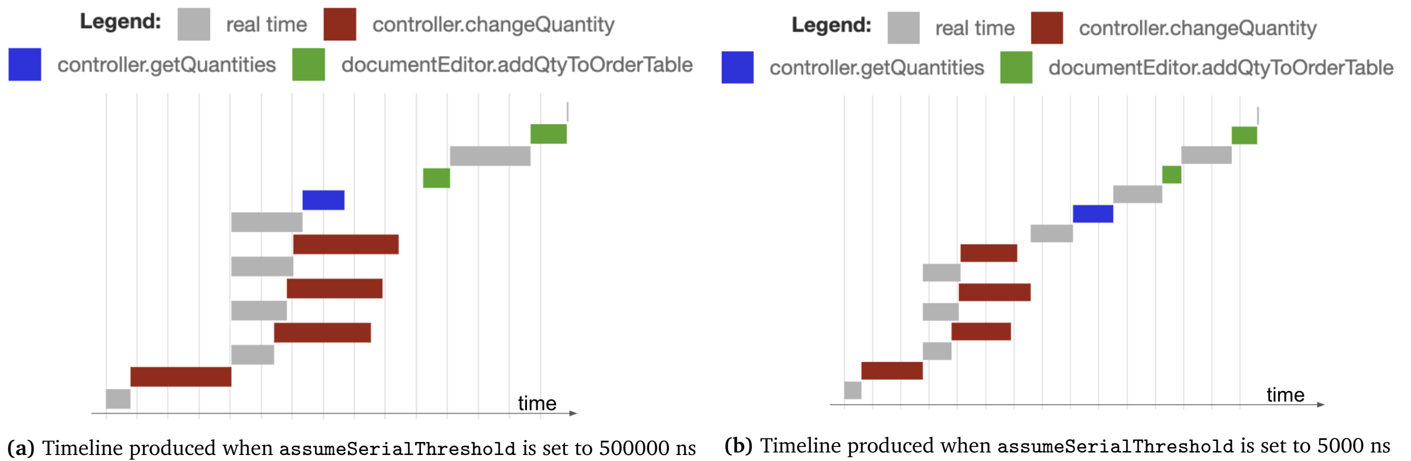
test case in the test suite for the checkout view involves nested function calls. To verify this, the correctness of any timeline generated can be checked. In particular, we focus on the test for performing quantity changes and re-rendering quantities.

```
// Function definition for changeQuantities:
async changeQuantities(changes) {
    await controller.changeQuantity("test", 0);
    let toAwait = []
    for (let i of changes) {
        toAwait.push(controller.changeQuantity(i.id, i.change));
    }
    await Promise.allSettled(toAwait);
}

// Test code:
it("should display new quantities after order changes", async () => {
        await runtimeCtx.repeat(runs, async () => {
            let change1 = {id: "1", change: 2};
            let change2 = {id: "5", change: -4};
            let change3 = {id: "10", change: 4};
            await orderView.changeQuantities([change1, change2, change3]);
            await orderView.renderQuantities();
        }, "performs quantity changes and re-renders quantities within average of ns"
            )

        ...
    });
```

Listing 5.1: Code under test for the test case of performing quantity changes and re-rendering quantities

The code under test is shown in Listing 5.1. `changeQuantities()` sets off a test call to the `controller.changeQuantity` mock, before calling it again for each change requested. After that, `renderQuantities()` calls the `controller.getQuantities` mock, which triggers calls to the `documentEditor.addQtyToOrderTable` mock.

Users need to choose an appropriate `assumeSerialThreshold` through some experimentation to enable QuiP to identify mock calls succeeding nested functions, as explained in Section 3.4.3. For example, when the `assumeSerialThreshold` is set to 500000 nanoseconds, which is too big, this leads to QuiP being unable to recognize `controller.getQuantities` as a mock call succeeding a nested function call. It fails to reassign the latest mock call in the nested function call as its parent, as shown in Fig 5.4a. However, when the `assumeSerialThreshold` is set to a more appropriate value of 5000 nanoseconds, it allows QuiP to correctly reassign its parent to be the last `controller.changeQuantity`, as shown in Fig 5.4b.



(a) Timeline produced when `assumeSerialThreshold` is set to 500000 ns **(b)** Timeline produced when `assumeSerialThreshold` is set to 5000 ns

Figure 5.4: Timelines of execution of code under test from Listing 3.9

5.1.4 Summary

In answer to RQ1, QuiP performs well in analyzing most of the different cases of asynchronous code considered in Table 3.1. However, it has difficulty with accurately handling scenarios where mock calls are scheduled serially after concurrent mock calls. It is also worth noting that the list of cases considered is not exhaustive. Through future research, there is potential to enhance QuiP's asynchronous dependency analysis even further, enabling it to achieve even greater accuracy.

Overall, QuiP is an excellent tool for automatically processing asynchronous code dependencies, especially compared to tools, such as DrAsync, that use the Async hooks API too but lack the fine-tuning for the above edge cases.

5.2 Evaluating runtime predictions (RQ2)

Another evaluation consideration is the accuracy of QuiP's runtime predictions. This was evaluated in the context of the book checkout web-application. In particular, the predicted times to execute the order display flow and successful payment making flow are compared to their actual measured times, when the web-application is deployed and observed in Google Chrome.

This is done by estimating the total runtime of the flows with QuiP during performance unit testing, before and after the performance models of the mocks are improved with empirical data from deployment. Following this, the time the deployed web-application takes to execute the same flows is measured by logging the difference between the timings returned by `window.performance.now()` before and after the flows are executed. The measurements are performed 10 times and averaged to reduce noise.

5.2.1 Runtime measurement results

The results of the runtime measurements are shown in Table 5.1 for the order display flow, and Table 5.2 for the successful payment making flow. It can be seen that the runtime estimations made by QuiP deviate significantly initially from the real time required to execute the flows. The average predicted order display flow time deviates by **92.3%** (3s.f.) from the real time required after deployment. The average predicted successful payment making flow time deviates by **79.8%** (3.s.f.) from the real time required after deployment. This is expected, as the initial runtime estimations are made based on the predictions of performance models that may diverge from the actual runtimes of the represented objects.

Table 5.1: Comparison of the predicted and actual timings for order display flow execution of the checkout web-application

Run	Time to execute order display flow (ms)		
	Predicted before model adjustments	**Predicted after model adjustments**	**Measured after deployment**
1	14.859844	257.13916	219.8
2	15.382149	374.276177	247.8
3	14.729036	274.88913	190.2
4	14.780004	179.758056	179.4
5	14.622264	186.96029	207.3
6	14.848939	393.884073	216.5
7	14.840807	381.493264	104.1
8	14.575085	134.569895	157.3
9	14.55059	178.325559	170.9
10	15.253981	248.978552	233.1
Average	14.8442699	261.0274156	192.64
Deviation% from real time	**92.3%**	**35.5%**	

Table 5.2: Comparison of the predicted and actual timings for successful payment making flow execution of the checkout web-application

Run	Time to execute successful payment making flow (ms)		
	Predicted before model adjustments	**Predicted after model adjustments**	**Measured after deployment**
1	47.85667	379.309608	265.2
2	48.926895	395.616684	275.7
3	53.202254	470.31507	280.8000002
4	55.103556	426.911988	267.9000001
5	59.475253	399.63587	273.5999999
6	56.763657	426.200398	258.0999999
7	55.232893	401.71575	274.5
8	55.715547	456.31055	242.2
9	54.093699	409.088178	293.7
10	52.382399	299.542757	241.0999999
Average	53.8752823	406.4646853	267.28
Deviation% from real time	**79.8%**	**52.1%**	

In contrast, the runtime estimations made by QuiP deviate less from the real time measurements after the performance models have been adjusted. The average predicted order display flow time now deviates by **35.5%** (3s.f.) from the real time required after deployment, **56.8%** less than before the performance models were adjusted. The average predicted successful payment making flow time deviates by **52.1%** (3.s.f.) from the real time required after deployment, **27.7%** less than before the performance models were adjusted. This increase in accuracy is likely attributed to the fact that the performance models were improved with empirical data. The response times predicted by the improved models are closer to the real-time response times of the collaborators mocked. However, the deviation in runtime estimations after the performance models were adjusted is still larger than expected.

5.2.2 Factors leading to inaccuracy in runtime estimation

Fundamentally, the performance models used to estimate the runtime of collaborators are extremely simple and inaccurate. Incorporating empirical data improves their accuracy, even with the same simple model. To achieve better accuracy, a more sophisticated model would be needed. However, as this work focuses on early performance testing and optimization rather than precise runtime modelling, uniform probability distributions are used as performance models. These form a limited view of the runtime of their mocked objects, which causes inaccuracy in the resulting runtime estimation.

Moreover, the runtime estimations and real-time measurements are performed in different environments. The virtual runtime estimations are conducted in an `node.js` environment during unit tests, with the use of the `process.hrtime` function which has the precision of nanoseconds. Meanwhile, the real time measurements used to fine-tune the performance models are obtained with logging on a browser with `window.performance.now()`. This causes several environmental inconsistencies that could explain the inaccuracy in the runtime estimation provided by QuiP.

Firstly, the `node.js` environment may execute the same piece of code more slowly than a browser environment, as it follows a single-threaded event loop, while the browser environment can make use of parallelism and multiple threads to speed up its Javascript execution. Additionally, as a security measure, browses introduce noise to the output of `window.performance.now()` to guard against timing-based attacks such as fingerprinting or Spectre [46]. This could be a factor leading to the large deviation of virtual runtime estimations from real runtime measurements.

Finally, the measurements used to fine-tune the performance models were obtained through logging on a browser as well. This introduces inconsistencies and noise into the performance models, for the same reasons as above. Since the performance models have a uniform probability distribution, they are sensitive to outliers in the timing data used to determine their parameters. For example, if there is one significant outlier when measuring the real runtimes of collaborators, it will become the new maximum/minimum of its mock's performance model. As a result, the estimated runtimes of the mock will vary over a larger range, leading to more noise in the final estimated runtime of the tested code.

5.2.3 Summary

In answer to RQ2, QuiP's predictions of the runtime of implementation code are limited in accuracy, especially before deployment. However, QuiP's predictions could be improved after gathering a larger amount of empirical data to reduce noise, and using a more sophisticated performance model. As the model is evolved according to real deployment data, the predicted and actual performance of code under test will converge to a greater extent.

Moreover, the accuracy of QuiP is not crucial to its effectiveness, since the focus of this work is on the use of early performance testing to optimize runtime, rather than runtime modelling itself. Despite the relatively suboptimal precision of QuiP's estimations, it remains highly valuable for evaluating the impact of code modifications on performance, as discussed in the next section.

5.3 Evaluating predictions of the effect of code changes (RQ3)

The next evaluation consideration is regarding how well QuiP can estimate the effect of code changes. To assess this, QuiP is used to evaluate the quantity and price display flow of the checkout view of the example web-app. QuiP's predicted improvement in code runtime is compared to the actual improvement observed when transitioning from a serial quantity and price display flow to a concurrent one. The code under test is shown in Listing 4.5 (for the serial flow).

This is done by estimating the total runtime of the flows with QuiP during performance unit testing, before and after the performance models of the mocks are improved with empirical data from de-

ployment. Based on the estimation results, the predicted percentage improvement in runtime from the code change can be calculated. Following this, the time the deployed web-application takes to execute the same flows is measured by logging the difference between the timings returned by `window.performance.now()` before and after the flows are executed. The actual percentage improvement in runtime can then be obtained as well. The measurements are performed 10 times and averaged to reduce noise.

5.3.1 Runtime measurement results

The results of the runtime measurements are shown in Table 5.3 for the serial display flow, and Table 5.4 for the concurrent display flow. Finally, the improvement in runtime as a result of the code change is calculated as a percentage of the original runtime of the serial display flow. This is calculated for the predicted runtime before model adjustments, the predicted runtime after model adjustments, and the real measured runtime. The calculated percentage change in runtime is illustrated in Table 5.5. It can be seen that the predicted change in runtime is in the same direction as the actual change. As the performance model is optimized, the percentage of predicted improvement converges towards the actual percentage of improvement.

Table 5.3: Comparison of the predicted and actual timings for serial quantity and price display flow of the checkout web-application

Run	Time to execute serial quantity and price display flow (ns)		
	Predicted before model adjustments	**Predicted after model adjustments**	**Measured after deployment**
1	18.825061	426.992806	232.0999999
2	17.360499	448.678034	193.8
3	17.63538	353.752624	205.7
4	19.200996	337.887007	217.5
5	18.172392	302.089264	253.2
6	18.663854	163.396468	296.5999999
7	19.176275	547.814803	210.6999998
8	18.066921	323.663921	201.6000001
9	19.479879	469.964016	205.8000002
10	17.432495	340.660675	236.8
Average	18.4013752	371.4899618	225.38
Deviation% from real time	**91.8%**	**64.8%**	

5.3.2 Accuracy of percentage change estimation despite inaccuracy of runtime estimation

Despite the predicted code runtimes still deviating significantly from the actual runtimes (by up to **91.8%** before model refinement and up to **79.1%** after model refinement), the predicted change in runtime is still much closer to the actual change. Before model refinement, the predicted percentage change was within **20%** of the actual change, and after model refinement, the predicted percentage change was within **5%** of the actual change.

This may be due to the fact that the percentage change in runtime was calculated with respect to runtimes predicted using the same performance models. Even if the predicted runtimes by the models are highly inaccurate when compared to the real runtimes of the collaborators, the predicted runtimes of the code before and after the code change are skewed in the same direction. The effect that the code change has on the order of execution of mocks is the same across both simulated and real code

Table 5.4: Table showing the predicted and actual timings for concurrent quantity and price display flow of the checkout web-application

Run	**Time to execute concurrent quantity and price display flow (ns)**		
	Predicted before model adjustments	**Predicted after model adjustments**	**Measured after deployment**
1	15.287281	135.520587	141.8
2	14.160915	244.582309	120.8
3	14.288696	259.686432	130.5
4	13.766373	309.330554	140.3
5	15.015881	227.206228	151.5999999
6	14.519327	321.102314	142.2
7	14.621973	386.091546	158.0999999
8	14.371388	224.264367	138.6999998
9	14.711513	194.260769	140.3000002
10	15.045628	150.238637	105.3
Average	14.5788975	245.2283743	136.96
Deviation% from real time	**89.4%**	**79.1%**	

Table 5.5: Comparison of the predicted and actual changes in runtime after code change

Performance improvement when switching from serial to concurrent display flow		
Predicted		Actual
Before model adjustments	After model adjustments	
20.8%	34.0%	39.2%

executions. Hence, the predicted percentage change in runtime is likely to be more similar to the real percentage change in runtime than the estimated runtimes are to the measured runtimes.

5.3.3 Summary

In answer to RQ3, QuiP's predictions of the effect of code change on runtime are increasingly accurate with model improvement. Even before deployment, the direction of change of performance predicted is accurate. As the model is evolved according to empirical data, the predicted and actual change in performance will converge to a large extent.

Although QuiP may not achieve high accuracy in predicting code runtime, as discussed in Section 5.2, its ability to accurately predict the impact of code changes has greater significance for its purpose of runtime optimization. This capability allows developers to make substantial performance improvements even before the performance models can be enhanced with empirical data. When used in conjunction with QuiP's timeline visualization tool, the high accuracy of QuiP's predictions of the effect of code changes can help to significantly enhance code performance at earlier stages of development.

5.4 Evaluating QuiP's effect on performance test turnaround time (RQ4)

Finally, QuiP's effect on test turnaround time is evaluated. Test turnaround time measures the (real) time it takes to run a test. In order to assess this, unit testing with QuiP is compared to current

performance testing methods and current unit testing methods, when testing the checkout view of the example web-app. For the former, the time that QuiP takes to generate a report on page loading time is compared to that of performance testing methods relying on real-time measurements, using Google Lighthouse as a representative. For the latter, the time that Jest takes to run a unit test suite with and without the use of QuiP for performance testing is compared.

5.4.1 Comparison with current performance testing methods

First, QuiP's test turnaround time is compared to that of current performance testing methods. In particular, Google Lighthouse is used as a typical tool for measuring the PLT of web-applications. Table 5.6 shows the real time taken to generate a report on the page loading time of the example web-app by QuiP and Google Lighthouse respectively. QuiP ran a single test of the page loading procedure of the checkout view for 10 runs, and returned the average of the runtimes. Meanwhile, Google Lighthouse generated a typical report on only page performance of the example web-app.

Table 5.6: Comparison of the test turnaround times of QuiP and Google Lighthouse when evaluating the example web-app

Run	**Time to generate a report on page loading time (s)**	
	Using unit testing with QuiP	**Using Google Lighthouse**
1	1.936	12.79
2	1.62	11.77
3	1.651	12.01
4	1.563	11.8
5	1.453	11.23
6	1.308	13.05
7	1.284	12.92
8	1.213	12.375
9	1.243	14.03
10	1.217	13.35
Average	1.4488	12.5325
% Lighthouse is slower by	**765%**	

It can be seen that Google Lighthouse is far slower, by a factor of 765%. This likely stems from the fact that QuiP measures runtime partially with virtual time. It employs performance models to estimate the response times of collaborators of the object under test, rather than relying on actual measured response times. This means that it does not have to wait for the actual page to load before it can measure the page loading time.

Meanwhile, the long time that Lighthouse takes to generate a report could also be justified by the larger volume of information scouted for in its report. Lighthouse includes additional statistics such as FCP and TTI, as well as tailored recommendations to improve the performance of the page. QuiP, on the other hand, is unable to generate such statistics automatically at the moment.

Hence, Lighthouse could be used for thorough inspections of performance statistics and suggested optimization opportunities after deployment. In contrast, QuiP offers the advantage of efficiently gathering large volumes of data, as well as analyzing runtime dependencies and estimating performance before deployment. Perhaps QuiP could be further extended as well, to offer automatic analysis of the same statistics that Lighthouse scans for.

5.4.2 Comparison with current unit testing methods

Next, QuiP's impact on test turnaround time is evaluated. We investigate how much longer the same unit tests take to run when their runtimes are monitored and asserted on by QuiP. This is done by running the test suite created for the checkout view, which monitors runtime with QuiP, and then recording the test turnaround time as reported by Jest. Then, QuiP is removed from the test suite, which is run with only unmodified Jest.

The results of running the same test suite with and without performance testing is shown in Table 5.7. The results indicate that including performance testing within unit tests leads to small increase in test turnaround time, with an average of 11.3%. However, when considering the exact increase in time, which amounts to 0.176 seconds, it appears insignificant compared to the range of test turnaround times, which spans 0.767 seconds for unit tests without performance testing.

Once again, the minuity of this increase may be due to the fact that QuiP measures runtimes partially in virtual time. The unit tests do not need to wait for collaborators to contribute their response times to the overall measured runtime, as performance models can be used to quickly estimate this. Therefore, any additional time incurred is only because of QuiP's runtime dependency analysis, and additional assertions on performance.

Table 5.7: Comparison of the test turnaround times of unit testing with and without QuiP

Run	**Time to run unit tests (s)**	
	Unit tests with performance testing	**Unit tests without performance testing**
1	2.292	1.711
2	2.158	2.059
3	2.011	1.658
4	1.338	1.325
5	1.295	1.368
6	1.342	1.322
7	1.55	1.604
8	1.721	1.354
9	1.88	1.292
10	1.784	1.918
Average	1.7371	1.5611
% Unit testing with QuiP is slower by	**11.3%**	

5.4.3 Summary

In answer to RQ4, QuiP offers a much faster alternative for measuring runtime, compared to tools that rely on real-time measurements. At the same time, the inclusion of QuiP in unit testing does not significantly slow test turnaround times. This indicates that QuiP is capable of quickly estimating runtime both before and after deployment, enabling its integration within a TDD cycle.

Chapter 6

Conclusion and Future Work

In this project, we have successfully developed QuiP, a novel tool for performance testing front-end web-application code before deployment. Through the development of the checkout view of a web-app, we have demonstrated QuiP's effectiveness in testing the performance of both synchronous and asynchronous JavaScript. Furthermore, we have evaluated QuiP's value in enhancing current coding practices by assessing its ability to estimate the runtime impact of code changes before and after deployment, as well as its test turnaround time.

QuiP extends the Jest unit testing framework and offers fast test turnaround times through virtual runtime estimation with performance models. Our innovative approach also includes a systematic search that identifies deviations between asynchronous dependencies and execution context relationships in JavaScript, despite limited documentation. This allows us to leverage the Async hooks API within QuiP. By automatically analyzing both synchronous and asynchronous code, QuiP achieves a substantial advancement in performance testing before deployment.

Based on the demonstration of QuiP's capabilities in developing the checkout view of a web-app and evaluating its performance estimation in unit testing, it is evident that QuiP excels in processing dependencies in asynchronous test code. This allows it to predict the runtime of code under test with increasing accuracy as its performance models are bolstered with empirical data from deployment. QuiP also demonstrates its ability to accurately estimate the percentage impact of code changes on runtime, achieving accuracy within 20% before deployment and within 5% after refinement. Notably, QuiP surpasses the turnaround time of real-time performance testing tools such as Google Lighthouse by 765%, while only introducing a marginal 11.3% increase in Jest unit test turnaround time.

In summary, these features demonstrate QuiP to be a promising candidate for integration into continuous unit testing practices for the front-end. Meanwhile, there are still exciting possibilities for further work, such as enhancing its handling of asynchronous dependencies, and expanding the range of performance insights it offers. By addressing these issues, QuiP's effectiveness and value to developers can be further enhanced.

6.1 Future Work

- **Enhancing QuiP's handling of asynchronous dependencies**

  A limitation of QuiP is in its ability to handle edge cases in asynchronous code, as discussed in Section 5.1 of Chapter 5. Specifically, QuiP's analysis of concurrent followed by serial executions is not entirely correct. Hence, there is a need to derive a more accurate analysis for the case of concurrent mock calls with children, especially for the wrongly processed scenarios in Table 3.2. Addressing this can enhance the accuracy of QuiP's performance estimations.

- **Constructing more accurate performance models**

  The accuracy of runtime estimation by QuiP is affected by the simple performance models used (i.e. uniform probability distributions), as described in Section 5.2 of Chapter 5. Although the emphasis of this work is on early performance testing and optimization rather than precise runtime modelling, there is potential for future work to enhance QuiP's performance models. This improvement can provide developers with a greater variety of information about runtime behavior, enabling them to fine-tune their code for further performance optimization.

- **Expanding range of performance insights**

  Currently, QuiP is only capable of reporting the runtime of test cases. It could be enhanced to report additional performance statistics, such as PLT and TTI, similar to Google Lighthouse. This would allow developers to measure their web-app's performance against industry benchmarks. Suggestions for optimization based on static code analysis could be added as well.

6.2 Ethical Considerations

QuiP does not have a specific target audience or exclusive focus, and can be utilized by both benign and malicious projects for performance testing. However, it is important to note that the tool itself was developed without any inherent malicious intent. While we cannot control how users choose to apply the tool, its purpose remains rooted in supporting software development practices.

Secondly, the project needs to be careful to respect copyright licensing of software it uses. The project aims to extend Jest, a Javascript testing framework, while making use of the Async hooks API. Both of these are licensed under the MIT license. This provides us with the right to use both of them, as long as their copyright notice and permissions notice are included in all copies, or substantial portions of the Software.

Bibliography

[1] Shivakumar SK. In: Getting Started with Web Performance Optimization. Berkeley, CA: Apress; 2020. p. 3-25. Available from: https://doi.org/10.1007/978-1-4842-6528-4_1.

[2] Amazon found every 100ms of latency cost them 1% in sales.; 2022. Available from: https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/.

[3] Wang XS, Balasubramanian A, Krishnamurthy A, Wetherall D. Demystifying Page Load Performance with {WProf}. In: 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13); 2013. p. 473-85.

[4] Chen L. Continuous Delivery: Huge Benefits, but Challenges Too. IEEE Software. 2015;32(2):50-4.

[5] Smith CU, Browne JC. Performance Engineering of Software Systems: A Case Study. In: Proceedings of the June 7-10, 1982, National Computer Conference. AFIPS ’82. New York, NY, USA: Association for Computing Machinery; 1982. p. 217–224. Available from: https://doi.org/10.1145/1500774.1500800.

[6] Measuring frontend performance (in modern browsers); 2022. Available from: https://www.skovy.dev/blog/measuring-frontend-performance-in-modern-browsers?seed=4hvumu.

[7] Beck. Test Driven Development: By Example. USA: Addison-Wesley Longman Publishing Co., Inc.; 2002.

[8] Loading speed. HTTP Archive; 2023. Available from: https://httparchive.org/reports/loading-speed#ttci.

[9] Chatley R, Field T, Wei D. Continuous Performance Testing in Virtual Time. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C); 2019. p. 109-15.

[10] Turcotte A, Shah MD, Aldrich MW, Tip F. DrAsync: Identifying and Visualizing Anti-Patterns in Asynchronous JavaScript. In: Proceedings of the 44th International Conference on Software Engineering. ICSE ’22. New York, NY, USA: Association for Computing Machinery; 2022. p. 774–785. Available from: https://doi.org/10.1145/3510003.3510097.

[11] Loring MC, Marron M, Leijen D. Semantics of Asynchronous JavaScript. SIGPLAN Not. 2017 oct;52(11):51–62. Available from: https://doi.org/10.1145/3170472.3133846.

[12] Riet Jv, Paganelli F, Malavolta I. From 6.2 to 0.15 seconds – an Industrial Case Study on Mobile Web Performance. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME); 2020. p. 746-55.

[13] Clark M. How the BBC builds websites that scale. net magazine; 2018. Available from: https://www.creativebloq.com/features/how-the-bbc-builds-websites-that-scale.

[14] Heričko T, Šumak B, Brdnik S. Towards Representative Web Performance Measurements with Google Lighthouse; 2021. p. 39-42.

[15] Doglio F. Top metrics you need to understand when measuring front-end performance. OpenReplay Blog; 2021. Available from: `https://blog.openreplay.com/top-metrics-you-need-to-understand-when-measuring-front-end-performance`.

[16] Stack overflow developer survey 2022; 2022. Available from: `https://survey.stackoverflow.co/2022/#most-popular-technologies-language`.

[17] Mobile site abandonment after delayed load time. Google;. Available from: `https://www.thinkwithgoogle.com/consumer-insights/consumer-trends/mobile-site-load-time-statistics/`.

[18] Crispin L, Gregory J. Agile Testing: A Practical Guide for Testers and Agile Teams. 1st ed. Addison-Wesley Professional; 2009.

[19] Lighthouse overview;. Available from: `https://developer.chrome.com/docs/lighthouse/overview/`.

[20] About pagespeed insights nbsp;—nbsp; google developers. Google;. Available from: `https://developers.google.com/speed/docs/insights/v5/about`.

[21] GoogleChrome. Lighthouse/readme.md at main · Googlechrome/Lighthouse; 2022. Available from: `https://github.com/GoogleChrome/lighthouse/blob/main/docs/readme.md#using-programmatically`.

[22] Get started with the pagespeed insights API nbsp;—nbsp; google developers. Google;. Available from: `https://developers.google.com/speed/docs/insights/v5/get-started`.

[23] ;. Available from: `https://nodejs.org/api/async_hooks.html`.

[24] Arora S. Understanding execution context and execution stack in Javascript. Medium; 2019. Available from: `https://blog.bitsrc.io/understanding-execution-context-and-execution-stack-in-javascript-1c9ea8642dd0`.

[25] Papatheocharous E, Andreou AS. Empirical evidence and state of practice of software agile teams. Journal of Software: Evolution and Process. 2014;26(9):855-66. Available from: `https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1664`.

[26] Farley D, Humble J. Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley signature series. Addison-Wesley Professional; 2010. Available from: `https://books.google.com.sg/books?id=OkeatwEACAAJ`.

[27] Mackinnon T, Freeman S, Craig P. Endo-Testing: Unit Testing with Mock Objects. Endo-Testing: Unit Testing with Mock Objects. 2001 12.

[28] Freeman S, Pryce N. Growing Object-Oriented Software, Guided by Tests. 1st ed. Addison-Wesley Professional; 2009.

[29] Horký V, Libič P, Steinhauser A, Tůma P. DOs and DON'Ts of Conducting Performance Measurements in Java. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering. ICPE '15. New York, NY, USA: Association for Computing Machinery; 2015. p. 337–340. Available from: `https://doi.org/10.1145/2668930.2688820`.

[30] Ersoz D, Yousif MS, Das CR. Characterizing Network Traffic in a Cluster-based, Multi-tier Data Center. In: 27th International Conference on Distributed Computing Systems (ICDCS '07); 2007. p. 59-9.

[31] Barbierato E, Gribaudo M, Iacono M. Performance evaluation of NoSQL big-data applications using multi-formalism models. Future Generation Computer Systems. 2014;37:345-53. Available from: `https://www.sciencedirect.com/science/article/pii/S0167739X14000028`.

[32] Dipietro S, Casale G, Serazzi G. A Queueing Network Model for Performance Prediction of Apache Cassandra. VALUETOOLS'16. Brussels, BEL: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering); 2017. p. 186–193. Available from: https://doi.org/10.4108/eai.25-10-2016.2266606.

[33] Facebook. Facebook/jest: Delightful JavaScript testing.;. Available from: https://github.com/facebook/jest.

[34] The fun, simple, flexible JavaScript test framework;. Available from: https://mochajs.org/.

[35] Avajs. Avajs/Ava: Node.js test runner that lets you develop with confidence;. Available from: https://github.com/avajs/ava.

[36] Frank DW. Jasmine 1.0 released; 2013. Available from: https://web.archive.org/web/20140222155946/http://pivotallabs.com/jasmine-1-0-released/.

[37] Nakazawa C. Jest 11.0 · jest; 2016. Available from: https://jestjs.io/blog/2016/04/12/jest-11.

[38] Aroush G. An Evaluation of Testing Frameworks for Beginners inJavaScript Programming: An evaluation of testing frameworks with beginners in mind; 2022.

[39] Globals · jest;. Available from: https://jestjs.io/docs/api.

[40] Expect · jest;. Available from: https://jestjs.io/docs/expect.

[41] Qiu J. In: Get Started with Jest. Berkeley, CA: Apress; 2021. p. 15-33. Available from: https://doi.org/10.1007/978-1-4842-6972-5_2.

[42] Mock Functions · jest;. Available from: https://jestjs.io/docs/mock-functions.

[43] Cleveland J. Mocking asynchronous functions with jest;. Available from: https://blog.jimmydc.com/mock-asynchronous-functions-with-jest/.

[44] Chartjs documentation;. Available from: https://www.chartjs.org/docs/latest/.

[45] Supabase. SUPABASE documentation; 2023. Available from: https://supabase.com/docs.

[46] MozDevNet. Performance: Now() method - web apis: MDN;. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now.