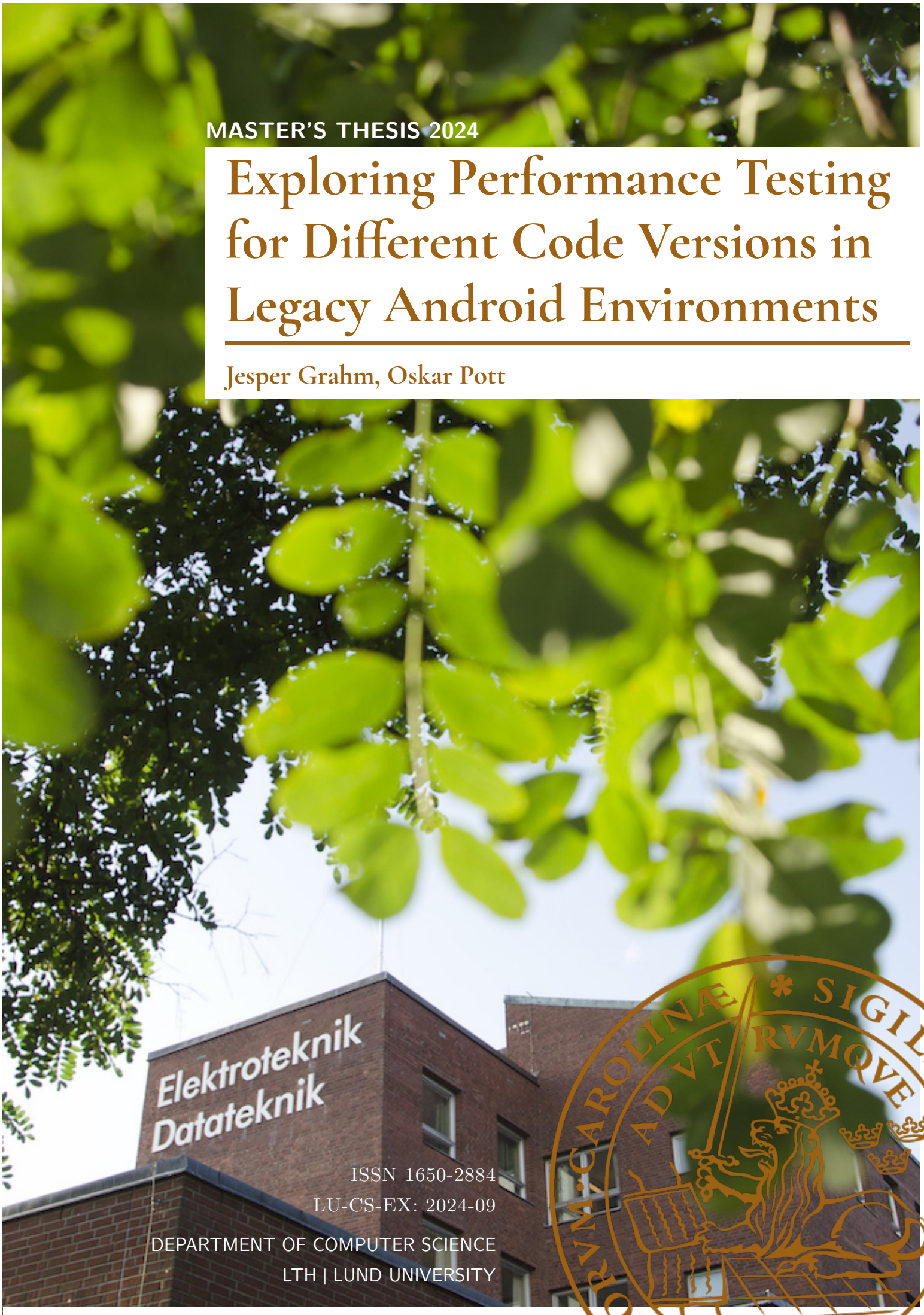



MASTER'S THESIS 2024

# Exploring Performance Testing for Different Code Versions in Legacy Android Environments

Jesper Grahm, Oskar Pott

ISSN 1650-2884
LU-CS-EX: 2024-09

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-09

# Exploring Performance Testing for Different Code Versions in Legacy Android Environments

## Prestandatestning av olika kodversioner i legacy Androidmiljöer

**Jesper Grahm, Oskar Pott**

# Exploring Performance Testing for Different Code Versions in Legacy Android Environments

Jesper Grahm
jesper.grahm@gmail.com

Oskar Pott
oskar.g.pott@gmail.com

February 7, 2024

Master's thesis work carried out at Tactel AB.

Supervisors: Niklas Hedström, niklas.hedstrom@tactel.se
Alexandru Dura, alexandru.dura@cs.lth.se

Examiner: Görel Hedin, gorel.hedin@cs.lth.se

## Abstract

As new technology develops the applications we use become either faster or more advanced, or both. With more and more applications being developed every day it is important to provide a good user experience if you want people to use your application over others in the same category. As many as 86% of users say that they have uninstalled or deleted a mobile app because of poor performance. This implies that performance is important for user retention.

We have looked at how the performance of an Android map application for airplane seatback screens has developed over time. To do this we designed a tool to automate performance testing of earlier code versions. We have also performed a case study based on the performance testing of different code versions with focus on devices running legacy Android versions. The base for the case study is our own work with developing the performance testing tool as well as a survey conducted on the development team of the map application.

Our results indicate that performance testing of different code versions of an Android app can be challenging but rewarding if certain criteria are fulfilled. Our findings also give a picture of how automated performance testing can help developers in their work. We also map how performance testing can be done for legacy Android versions and the availability of different tools depending on how old the running version is.



**Keywords**: performance testing, benchmarking, case study, Android, Android debug bridge

# Acknowledgements

First of all, we would like to thank Tactel for giving us the opportunity to do our thesis with them and making us feel welcome at the company. We would also like to direct a big thank you to the people of the Arc team, who have helped us with all our problems along the way. A special thanks goes out to Niklas Hedström, our supervisor at Tactel, who always pushed us to do our best and reminded us when it was time for lunch. We would also like to thank Alexandru Dura, our supervisor at LTH, for helping us along the way and Görel Hedin, our examiner, for stepping up as our examiner when we found ourselves without one halfway through the thesis. Lastly, a heartfelt thank you to all the friends at the D-guild who made these years a lot easier than they would have been otherwise.

# Contents

# Chapter 1
# Introduction

Android applications are run on millions and millions of devices each second all the time. There are over 2.5 billion Android users globally and in the landscape today where we use our phones more time each year it is imperative for developers to ensure that their app runs smoothly [1]. Ensuring the performance is important as suboptimal performance can lead to user dissatisfaction, decreased retention and ultimately, a decline in an app's popularity [2]. As many as 86% of users say that they have uninstalled or deleted a mobile app because of poor performance [3].

Even though app performance is important in all environments one could argue that it is more important when installed in an airplane. The applications that run on the screens in front of each seat in an airplane are often heavily integrated with the brand of the airline that owns the plane. The average user most likely sees the application as a part of the product that is their flight. Therefore, if the user experiences performance issues in the application, that will reflect badly on the airline.

In our thesis we explore the challenges in testing the performance of an Android application. The Android application is running on a touchscreen device used on airplanes. The airplane touchscreens are the ones that are located on the seat in front of you when flying, and they are used for both information and entertainment. The specific application that we wanted to test was the map application, in this application the earth is visible with a preview of where the airplane currently is on the globe. The touchscreens are running a legacy version of Android and the hardware is from the same era as the screens. In our thesis we have looked at two main aspects. First, we wanted to test these screens going back in time and testing multiple code versions to see how the performance has changed over time. To do this we developed an automated testing tool to test the performance of the application, where it interacted with the application to simulate a user and then saved the results to be viewed on a website that we also created. Secondly, we surveyed the developers' perception of the application's performance. We have structured our work in the style of an "exploratory case study" as described by Runeson and Höst [4].

## 1.1 Research Questions

This report aims to answer the following questions:

- What are the challenges in performance monitoring of software that run in legacy Android environments?
- What are the challenges in finding performance changes?
- Is performance monitoring of different code versions a viable way to find code changes that affect performance?

## 1.2 Limitations

One of the most challenging aspects of our thesis work has been the legacy Android versions that the screens are running. The hardware and Android version that the screens use are decided by the airlines and Tactel has to adapt to that. A single seatback screen can cost up to $10,000 according to The New York Times [5]. This makes it very expensive to replace the screens of an airplane, let alone an entire fleet. And since airplanes can be used for an average of 27 years [6] much of the seatback hardware of today's planes is quite old. Since many screens are based on old hardware, the Android versions are often also old. The Android version directly translates to an API-level [7] which dictates which features are available for the specific version. The oldest version that any device is running at Tactel is Android 4.3 which correlates to API-level 19 and was released in 2013 [8]. Thankfully, this version is only run by some handheld devices and not that relevant to performance test since the majority of devices have a higher API-level. We instead decided to run our tests on devices that used Android 5.1 (API-level 22), which was released in 2015. We did this because this was the lowest API-level for seatback, which is the most common type of screen. The functionality between API-levels can be quite different and because of that we decided to not test on any higher API-levels since we would have to rewrite parts of our code for each API-level. Also, since we are doing performance testing, the screens with the oldest hardware and operating system should in most cases have the worst performance when running the same application. Therefore we have worked under the assumption that newer screens will have similar or better performance compared to screens using API-level 22. Even though that might not be true for all performance issues it is beyond the scope of this thesis to test on different hardware since it will require more work and we believe that our results give a solid generalization of the performance of Arc.

Another limitation we had was a request from Tactel that the tool should be able to run completely externally. This meant that the tool should not be compiled with the target application. By making it external, the tool can be used to performance test different applications without having to affect the source code.

# Chapter 2
# Background

In this chapter we will mainly talk about performance testing, what it is, why it is important and how it can be done. We will also introduce the concept of monitoring performance over time which serves as the base for our thesis as well as include a brief introduction of the Arc application. Lastly we will present some papers that relate to our thesis in the work they have done.

## 2.1 Performance Testing

Performance testing is a crucial aspect of Android application development, playing a significant role in ensuring that applications meet the performance expectations of end-users. It encompasses a range of tests and evaluations designed to measure various performance metrics, such as responsiveness, resource utilization and load times. This section will explore the importance of performance testing in the Android development process and outline how it should be implemented effectively.

### 2.1.1 Importance of Performance Testing

As the Android ecosystem continues to expand, encompassing diverse devices, versions, and user behaviors, the need for robust performance becomes increasingly important. Performance stands as a pivotal factor when determining user satisfaction and application success, and since there are an abundance of apps for the user to choose from, the importance of user retention is vital [9].

#### User Retention

The importance of performance testing is related to the notion of application retention. Retention is measured by the percentage of users who continue to use an app after their

initial download or installation. According to Quettra the following table 2.1 shows the average retention values for Android applications [2]. This is very telling since it shows that the average application loses 76.58% of its DUAs (Daily User Acquisition, the amount of unique users who download the application) within the first 3 days [2]. Within the first 30 days the number is instead over 90%. This is however the average retention, if we look at the best performing applications instead in the tables 2.2, 2.3, 2.4 and 2.5 (according to the Google Play ranking) and their retention we can see that the retention is directly correlated to the ranking [2].

But how is user retention related to the application performance? According to Google, 20.3% of users will immediately use a different Android application with the same functionality if it takes too long to load [9]. Google also states that users have very low tolerance for performance issues and are very prone to change application if necessary. So performance is very important for user retention and as we can see from the tables 2.2, 2.3, 2.4 and 2.5 user retention is very closely related to success.

| Days Since Install | Users Still Active % |
|---|---|
| 1 | 29.17 |
| 3 | 23.42 |
| 7 | 17.28 |
| 14 | 13.11 |
| 30 | 9.55 |
| 60 | 6.82 |
| 90 | 3.97 |

**Table 2.1:** The average retention for Android applications.

The retention data for the best performing Android applications

| Days Since Install | Users Still Active % |
|---|---|
| 1 | 74.67 |
| 3 | 71.51 |
| 7 | 67.39 |
| 14 | 63.28 |
| 30 | 59.80 |
| 60 | 55.10 |
| 90 | 50.87 |

**Table 2.2:** Top 10

| Days Since Install | Users Still Active % |
|---|---|
| 1 | 64.85 |
| 3 | 60.31 |
| 7 | 54.13 |
| 14 | 49.48 |
| 30 | 44.81 |
| 60 | 39.60 |
| 90 | 34.50 |

**Table 2.3:** Top 50

## Application Stability

According to a report in application stability by bugsnag [10], the median stability score across all of the applications that were analyzed was 99.8%. This percentage is a score in stability where stability is defined as a percentage of app sessions that are crash-free. The

| Days Since Install | Users Still Active % |
|---|---|
| 1 | 48.72 |
| 3 | 42.96 |
| 7 | 35.93 |
| 14 | 30.79 |
| 30 | 25.45 |
| 60 | 21.25 |
| 90 | 18.98 |

**Table 2.4:** Top 100

| Days Since Install | Users Still Active % |
|---|---|
| 1 | 34.31 |
| 3 | 28.54 |
| 7 | 21.64 |
| 14 | 17.43 |
| 30 | 13.62 |
| 60 | 10.74 |
| 90 | 8.99 |

**Table 2.5:** Top 5000

report also showed that a 1% lower stability score can lead to a drop of almost 1 whole star in the app stores, out of five maximum. This means that stability is also a highly valued performance metric by users.

### 2.1.2 Defining Performance

The evaluation of a well performing application is subjective, often hinging on the end-user's perception rather than specific technical metrics. A proficient application allows seamless task completion without noticeable delays or frustration. Its success lies in enabling users to achieve objectives without distractions, maintaining smooth navigation for the users. Despite the seemingly straightforward criteria, defining optimal performance varies among individuals.

In the book The Art of Application Performance Testing, the concept of two different kind of performance indicators is proposed [11]. These indicators can imply if an application is performing desirably or undesirably. The two types are Service-oriented indicators and Efficiency-oriented indicators. Service-oriented indicators are availability and response time, they are defined as:

- **Availability**
  The amount of time an application is available to the end-user. The absence of availability carries considerable implications as even minor downtimes can result in substantial financial repercussions for numerous applications. From the perspective of performance testing, this signifies the inability for an end-user to utilize the application.

- **Response Time**
  The amount of time it takes for the application to respond to a user request. In performance testing, the standard practice involves assessing the system's response time, measuring the interval between a user initiating a request from the application and the arrival of a reply to the user.

Efficiency-oriented indicators are throughput and utilization and they are defined as follows:

- **Throughput**
  The rate at which the application can process a certain volume of work within a given time frame. It specifically measures the amount of data, tasks, or transactions that

an application can handle and complete successfully in a unit of time. Throughput is a critical performance metric because it reflects the efficiency and capacity of an application to execute operations.

- **Utilization**
  The degree to which the available resources (such as CPU, memory, disk, or network) are being used or occupied by the application at any given time. Often measured in a percentage.

These combined indicators can collectively provide an assessment of an application's performance. In this report we will focus on response time and utilization. Availability and throughput will not be in the scope of this report.

### 2.1.3 Performance Standards and Metrics

For Android applications there are no standards for performance metrics. However, there are guidelines for most of the performance metrics so that developers can relate to something [12].

#### Response Time

The metrics that correspond to the user making a request and the application responding are load times, response times and if the application suffers from inconsistent frame timing.

- **Load Times**
  Load time is defined as the time it takes for the application to open, not the time for the application to load once it has been opened. This can be divided further into three subcategories. The load times are according to the Android documentation [12].

  - **Cold Start**
    A cold start should never take longer than 5 seconds. A cold start initiates upon the creation of the application's process. It encompasses the users' experiences when installing the app and launching it for the first time, as well as reopening the app after device restarts or when the system entirely halts the app for various reasons.
  - **Warm Start**
    A warm start should never take longer than 2 seconds. A warm start happens when the application is started after the activity has been destroyed but not the application process. This scenario occurs when an app has been running in the background for a considerable period without the system terminating the entire process. Additionally, when an orientation change necessitates the destruction and subsequent recreation of the activity, it qualifies as a warm start.
  - **Hot Start**
    A hot start should never take longer than 1.5 seconds. In a hot start, an app, including an activity that was previously paused, becomes visible to the user and transitions into the started state. This can happen when you switch between applications without stopping them.

- **Response Time**
  Response time is the time it takes for the application to load new information on the screen and show it to the user. It might be the time it takes for a press of a button in the application to change content. According to Jakob Nielsen there are three different categories [13].

  – **0.1 Seconds**
    When a system responds within 0.1 second, users perceive it as instantaneous, requiring no specific feedback other than displaying the result. This is necessary when displaying characters in a document when typing, or playing a video game, where responsiveness is very important.
  – **1.0 Seconds**
    Up to 1.0 second is the threshold for maintaining uninterrupted flow of thought for users, though they will notice the delay. Typically, between 0.1 and 1.0 seconds, no explicit feedback is needed. But above this time users will be annoyed with the delay it takes to display new content.
  – **10 Seconds**
    If the delay is around 10 seconds the user will start to perform other tasks and won't be engaged directly. They will either wait for the process to finish or they will quit the application. This is highly dependent on what the user is waiting for.

  However, these numbers can be very individual. Some social media consumption have been found to affect the attention span of individuals. And users might be less likely to tolerate a longer response or load time [14].

- **Frame Timing**
  Frame timing is the time in between frames. The standard today is to always have a frame timing that corresponds to 60 frames per second. This equates to around 16ms frame time. Although it is the standard to have 16 ms frame timing it is more important that this number is consistent since inconsistencies is easily detected and will make the application feel less smooth [15].

## Utilization

For utilization metrics there are no standards either. This is even more dependent on the application. For example if you have a video editing software on your Android device you want that to be as fast as possible and you want it to use the full potential of the CPU. Yet, if you have an application that is doing something in the background it is not reasonable to use 100% of the CPU since that will limit what you can do in the foreground. There are many utilization metrics you can look at in an Android application. Here are some of the metrics that can be measured:

- **CPU**
  CPU utilization signifies the extent of processor usage during execution. Elevated CPU consumption often correlates with diminished performance, increased power consumption, and potential thermal concerns. Monitoring and managing CPU usage is important since prolonged high CPU usage can lead to thermal concerns.

- **GPU**
  The GPU utilization is similar to CPU utilization but is instead focused on the graphical and visual performance. Excessive GPU utilization may result in degraded visual fidelity, frame rate drops and thermal issues.
- **Thermal**
  This is closely related to the CPU and/or the GPU utilization, since the CPU and the GPU are the two components that will output heat into the system. A too high increase in the temperature of the device can lead to components breaking and failing. This is very serious and to mitigate this you can throttle the CPU or the GPU if they are too high in temperature. Throttling is when you lower the frequency and therefore limit the amount of power it can draw. This will also degrade the performance and is not ideal. This phenomena is called thermal throttling.
- **Memory**
  Memory utilization is related to the allocation and usage of the Random Access Memory (RAM). However, it does not function the same as the CPU utilization for instance. It can instead be broken down into two parts. Firstly, the amount of RAM that is occupied is important. A device has a set amount of RAM and if an application wants to allocate more, it will probably crash. Secondly, the speed of the RAM can differ, and slow RAM can lead to performance degradation if an application is doing a lot of memory operations such as allocation and reallocation.
- **Disk**
  Disk utilization can refer to the amount of data stored on the device. This is not the same as the RAM and is used for storing data that need to exist after a reboot of the device. It can also refer to the amount of operations that are being executed on the memory, a large amount of operations can lead to the disk being bottlenecked and subsequently the application might not work properly.
- **Network**
  Network utilization refers to the data consumption associated with network-related tasks such as retrieving content from remote servers. Excessive network usage can impede responsiveness, especially under low connectivity scenarios. So it is important to be efficient with the network usage since it can differ a lot between different devices.
- **Battery**
  Battery utilization quantifies an application's impact on device battery life. This is closely related to the CPU and GPU utilization since they can draw a lot of power. The screen of the device can also be a contributing factor on the battery drain. So managing background activity and optimizing resource intensive workloads can help improve battery performance. The battery is also affected by thermals and will function differently when hot or cold.

## 2.2 Performance Testing Strategy

According to the book The Art of Application Performance Testing, it is important to start with performance testing as early in the development life cycle as possible [11]. However,

it is necessary to address many factors before you can implement your performance testing strategy. The most important ones are the following:

- Choosing an appropriate performance testing tool
- Designing an appropriate performance test environment
- Setting realistic and appropriate performance targets
- Making sure your application is stable enough for performance testing
- Obtaining a code freeze
- Providing sufficient test data of high quality
- Ensuring accurate performance test design
- Identifying the key performance indicators (KPIs)
- Allocating enough time to performance test effectively

All of the mentioned factors might not be relevant for all projects. Nevertheless, they can be a good starting point. The factors will now be explained further.

### 2.2.1 Choosing an Appropriate Performance Testing Tool

Automated performance testing tools are almost always necessary to carry out performance testing tasks. There is usually not a practical way to provide reliable, repeatable performance tests without using some form of automation [11]. Automated performance testing tools aim to streamline the testing process by enabling the recording of end-user actions and transforming this information into some form of script. These scripts serve as the foundation for generating load testing scenarios that mirror typical end-user behaviors. These scenarios constitute the actual performance tests, and once established, they can be conveniently rerun whenever needed. This stands as a significant advantage over manual testing methods. There are however other ways of generating user actions, but these would not be as good as real end-user actions since they would not correctly simulate a real world scenario. Even though they are not as high quality they can be useful since they can be numerous times faster to implement. An example of this sort of tool is dumb monkey-testing where the actions are random. This is a common testing scenario for Android device [16].

A performance testing tool usually have the following components in some capacity:

- **Scripting Module**
  This module is the one responsible for recording user actions.
- **Test Management Module**
  This module is the one responsible for generating and creating scripts based on the recorded user actions.

- **Load Injectors**
  This is the module responsible for creating the load. Either a server if you want to simulate multiple users at the same time or just a workstation if you are only simulating one user.

- **Analysis Module**
  This module generates the results in some capacity, it might be a report, graph or something similar but it can also be just a number if the test is very lightweight.

So to choose a tool it is important to make sure that each component satisfies your requirements.

### 2.2.2 Designing an Appropriate Performance Test Environment

The test environment is of course highly dependent on the chosen tool. If you are carrying out performance tests that require a heavy workload it is important to take that into account in your environment.

### 2.2.3 Stakeholders Setting Performance Targets

Setting realistic and appropriate performance targets is a very important step in the testing strategy. There needs to be a consensus from all of the relevant stakeholders. An example of the stakeholders can be:

- **The Business Side**
  They are responsible for managing the cost of the project and there might be costs to consider when choosing the tool. For example, licensing and the procurement of workstations or servers used for the load injectors.

- **The Developers**
  They are responsible for creating the product and they need to be involved and make sure that the expectations for the targets are realistic and doable.

- **Testers**
  They can be the same as the developers in some cases but they are responsible for creating the tests and need to be aware of the targets when creating the tests.

- **The End-User**
  This is a very important stakeholder and if the end-user is not satisfied with the performance it can lead to them not using the product. Explained in the previous chapter called User Retention.

### 2.2.4 Making Sure Your Application Is Stable Enough for Performance Testing

Having a stable application has already been shown to be important in production environments in the section Application Stability. It is also an important part in performance testing because of the following reasons:

- **Reliability of Test Results**
  Unstable applications introduce inconsistencies in test results. When an app is prone to crashes or erratic behavior, testing outcomes become unreliable. Data collected under unstable conditions may not accurately represent the app's actual performance.

- **Inaccurate Performance Metrics**
  Stability issues skew the performance metrics obtained during testing. Metrics such as response times or resource utilization become unreliable when the app behaves unpredictably. This unreliable data can mislead developers to draw incorrect conclusions about the state of the application.

- **Difficulty in Root Cause Analysis**
  Instability makes it challenging to pinpoint the root causes of performance issues. When an app crashes sporadically during testing, isolating the exact cause becomes arduous. This hinders the resolution of underlying problems leading to instability.

- **Impact on Decision Making**
  Decision-making based on unstable test data can lead to flawed strategies. Businesses rely on accurate data to make informed decisions about releases, updates, or feature improvements. Unreliable data may result in poor decision-making and subsequent negative outcomes.

### 2.2.5 Obtaining a Code Freeze

This is probably one of the simpler factors, many companies use version control for the code and obtaining a code freeze would be trivial. However, there are situations where it can be more difficult even if you are using a version control tool. A situation where you need code from many different sources that depend on each other can prove to be more difficult than expected so it is important to have a strategy for obtaining a code freeze of the relevant code.

### 2.2.6 Providing Sufficient Test Data of High Quality

This is highly dependent on the type of tests that are being executed. If the type of tests are local hardware performance tests it is not relevant to think of test data since none are required. But, if the test is something similar to a load test where you want to simulate multiple users at the same time to see how the system handles the load. Then it is very important that you have correct and a sufficient amount of test data.

### 2.2.7 Ensuring Accurate Performance Test Design

This step is to make sure that you choose the correct sort of performance test. Some might be more relevant than others, it all depends on the sort of application you are testing.

- **Baseline Test**
  This is used to establish a start case of the how well the application performs. It should be very isolated, meaning that no other activities should be running on the system to make sure that the application is the only thing that is affecting the performance. Then, if and when things change in the application itself or other external factors, one can see if the applications performance has been degraded or if any optimizations of the performance has had an impact.

- **Load Test**
  This is a test where you want to test if the application can handle the load that it is designed to handle. The aim for such a test is to meet the specified requirements for availability, response time, throughput and utilization. This test should be designed to very closely resemble real world usage of the application at high load.

- **Stress Test**
  Similar to a load test it is designed to test many users at the same time. But, instead of limiting the amount to the requirements of the application, this is unlimited and it is designed to make the application fail in some way. Here it is interesting to look at which part is the one to fail first. Which performance metric is the first to miss its target requirement.

- **Soak or Stability Test**
  This is a test to see if there are issues in the long term of the application. A typical example can be a small memory leak that only affects the application after a longer time than a normal load test.

These are just examples of types of tests and there are a plethora of different ones to choose from.

### 2.2.8 Identifying the Key Performance Indicators (KPIs)

It is important to identify which key performance indicators that are crucial to your application and should be monitored. Identifying these can help later during root-cause analysis of any problem that can occur during the performance tests. And failure to identify the correct KPIs can lead to the performance test not working as intended and not correctly informing when the application is performing differently.

### 2.2.9 Allocating Enough Time to Performance Test Effectively

Allocating time is easy to plan for in theory but can be harder to do in practice and you need to allocate sufficient time to be able to performance test effectively.

## 2.3 Monitoring Performance Over Time

The reason we wanted to conduct this case study is that we felt there was valuable knowledge to be gained from comparing the performance of different code versions and especially earlier versions of the same application. By knowing how the application has performed in the past, the developers can get a good idea of where the performance should be and can be alerted if it deviates. If performance testing is introduced and run on each new code version it will always take time to build up a large enough data set of samples to get a good overview of the application's performance. If there is a need for a new metric to be introduced this cycle of building up a sample size restarts. Being able to performance test earlier code versions enables developers to change metrics and configurations as needed without having to sacrifice sample size. The pros and cons of this flexibility and when it should be used is something we wanted to explore further with this case study. In the section about Related Work we talk about some other papers that have researched similar things to us and upon which we draw inspiration for our case study.

## 2.4 Arc

Arc is an Android based map application used on the in-flight touch screens of airplanes. It consists of different views to let the user interact with the world map and get real time information about the flight they are currently on. A brief description of Arc's features is available on Tactel's website [17]. The application uses an engine, also developed at Tactel, to process the map, specifically the composition of the individual tiles that make up the surface of the map. It also uses input from the airplane itself to display information to the passengers. Tactel uses mocked environments for Arc to simulate the setup of an actual airplane. For example, Arc connects to a service called Atlas which feeds the application information from a simulated flight.

## 2.5 Related Work

The work of Liu et al. [18] relates to the work that we have been doing, but with some key differences. They found and characterized 70 performance bugs from eight popular smartphone applications. By identifying common patterns in the bugs they managed to chart common performance issues in these applications that could then be used with a tool to detect performance bugs in the source code. The found bugs were then communicated to the developers. The most common bug pattern that they found was "GUI-lagging" which constituted 75,7% of the performance bugs they found. We have also put focus into measuring issues with the UI being slow since this is an issue that is easily noticed by the end-user. Our way of doing this was mainly to measure FPS but also through memory and CPU usage which could slow down the application. The work of Liu et al. shows the importance of performance testing and the impact that finding bug patterns can have with real-world examples.

When Linares-Vásquez et al. [19] looked at 485 android projects they found that 73% of developers rely on manual testing to detect performance issues. By automating performance testing, the developers can continually get an updated frame of reference for the application's

performance put into numbers. This makes it much easier to notice when these metrics deviate and when performance issues arise. Therefore, with our case study, we also wanted to explore how efficient automation of performance testing different code versions could be done.

Reichelt et al. [20] have done some previous work in this area with favourable results but several differences from our work. They developed a tool that repeatedly measures performance on transformed unit tests of different code versions to detect changes in performance. A key difference compared to our work is that they measured performance on unit tests and assumed that the performance matches the actual program. Because they only looked at the code they used response time as their only metric for performance. By looking at the running application we could use more metrics that more concretely correlate to performance issues that users notice, for example, FPS and memory usage.

# Chapter 3

# Method

The base for our case study will be the research and work that we have put into building our performance monitoring tool. By researching ways to measure performance for Android applications we got a good understanding for which tools are available. Researching the available tools to test performance on Android we got broader knowledge of which tools that were commonly used depending on the Android version. Apart from our own experience gained from developing this tool we have also spoken a lot with the developers at Tactel and asked them about their views on performance testing and how they have done it in the past. Many of these conversations have guided our decisions in the development of our tool and contributed to our experience. To quantify their experience with Arc and performance testing as well as get a wider reach to all developers of Arc we also conducted a survey of which the results will provide further information to base our conclusions upon.

## 3.1 Performance Metrics

The metrics we chose are CPU usage, FPS (Frames Per Second) and RAM memory usage. The aspects we looked at were: what we could get data from, what we thought would provide best info, and what the developers thought would be useful.

When choosing which performance metrics we were going to evaluate we had to balance multiple different aspects. The first and arguably most important aspect was whether or not we could reliably measure data of the metric. Since we already were limited in our method because of the low API-level of our device we were essentially forced to rely on the information that a few specific commands could provide.

## 3.2 Android Debug Bridge

The Android Debug Bridge [21] or "adb" for short is a command-line tool that allows the user to connect to a device that is running Android. This connection can then be used to run commands through a Unix shell which can help in debugging an Android application. By executing commands on a device through adb it is possible to get valuable information about the applications that are running on that device. We mostly used the general command "adb shell dumpsys" followed by different options to get information about the Arc app. This command is very versatile but we were mostly interested in hardware statistics concerning among others CPU, memory and frames. The commands we used were:

- `adb shell dumpsys cpuinfo`

  Reads the current CPU usage of the device in %.

- `adb shell dumpsys meminfo`

  Reads the current RAM memory usage of the device in kB.

- `adb shell dumpsys SurfaceFlinger --latency SurfaceView`

  Outputs information about the last 128 frames that have been shown on the screen. The information that we looked at is the timestamp of when the frame was shown. By repeatedly clearing this cache we could count the number of frames shown since our last reading and by also measuring the time we could manually calculate the frames per second. For this to give accurate readings the commands had to be run with very short intervals. Maintaining this interval was trivial since the speed at which frames were rendered was severely limited by the old hardware of the device.

- `adb shell dumpsys meminfo <Arc process>`

  Reads the current RAM memory usage of a specific process. This command took several seconds to run and therefore we could not afford to run it with the other commands. Instead we ran it once at the end of the script to measure how much memory the process had allocated during the run. We chose to look at allocated memory for the native and dalvik heaps but for the scope of this thesis we do not need to fully understand how the native and dalvik heaps work. Briefly explained, the dalvik heap contains instances of Java objects while the native heap is allocated by the operating system and used by for example C code, which the Arc engine runs. Together, these heaps represent the total amount of allocated memory for the process.

## 3.3 Atlas

Atlas is the flight simulator for the application. The simulator runs on a local server and is by default configured to Atlas 1, this configuration is the same as described in Pipeline B step 2. There is also a web interface were it is possible to change a number of things such as what flight to simulate and the state of the flight, for example take-off or cruising. This is used by almost all of the developers every day, since a specific route often need to be reproduced to find where there was a bug. Or to look at a certain phase of the flight. In order to be

independent from the rest of what the company is doing we were given our own server called Atlas 4 so that we would not interfere with other developers and they would not interfere with our performance testing. We wanted to have the state of the flight to be the same when testing to minimize the number of factors that could impact our results.

This simulator is only available through a web interface which means that it is necessary to manually edit the parameters of the flight that is being simulated. It is not possible to automate this since there was no API available. This means that if we wanted to simulate a flight at the same point in time every run we would have to manually edit the flight simulator so the flight was at the correct time. This was not feasible since we wanted to do our testing over a period of many hours and not be dependent on any manual input. Our compromise was that we were just starting a long flight that would not end until all of our tests had been run. However, this also meant that the airplane was not always at the same place on the globe and this could have had an impact on our results. There is a way to stop the simulation in time and not have the airplane moving. This was discussed and it was deemed less appropriate since it would not reflect how the application works in normal usage.

Another issue we faced with Atlas was that we had to run two different versions depending on how long ago we wanted our code to be from. If we wanted to use a code version up to a few months back the regular version worked without issue. But further back and we had to use a legacy version. This was not a major issue but we had to manually change our pipeline depending on when the code was from.

## 3.4 UI Automator

UI Automator is a tool that simulates user input on an Android device. It is a UI testing framework made for conducting extensive functional UI testing across both the Android system itself and installed applications [22]. It provides the ability to interact with visible elements on a device. It also facilitates operations such as opening the settings app or changing the volume of the device. The UI Automator API support both Kotlin and Java natively, which means that test can be written in either language. The minimum required Android API level is 18 which corresponds to an Android version of 4.3 [22].

We could not use UI Automator directly, because it needed to be packaged with the rest of the application which is against our requirement of having external performance monitoring [23]. Instead we used a Python wrapper for the API and it does not need to be packaged with the application. It works by installing another application on the device called ATX which functions as a server that loads and sends commands to the device and the target application [24].

There are two ways of accessing and interacting with elements on the screen, the first way is to define what coordinates on the screen to interact with and then perform an interaction on that coordinate, one example would be to click on that location another would be to perform an interaction called a pinch out or pinch in. This interaction is most commonly associated with either zooming in or out. So all of the interactions are independent from what is being displayed on the screen. After clicking there would then be a delay for the next interaction so that the UI elements on the application would be able to update and display the correct information. This is done using the following code:

```
d = u2.connect()
```

```
d.app_start("com.android.settings")
d.app_wait("com.android.settings", timeout=30)
d.click(100, 200)
```

This is a fully functioning script using the UI Automator API. The first line is to connect the program to the device and storing that connected device in a variable called `d`. Then second and third line is used to start the settings application and waiting for the application to start. This is necessary because it takes time for UI elements to load and the following click action would be performed before the application have had time to open. The fourth line is a click interaction with the screen on the given x, y coordinates. Here x is 100 and y is 200.

The second way of interacting with the screen is dependent on what elements are actually displayed on the screen. This is done by using something called a selector [25]. A selector can be used with the following code:

```
d = u2.connect()
d.app_start("com.android.settings")
d.app_wait("com.android.settings", timeout=30)
battery_button = d(text='Battery', className='android.widget.TextView')
battery_button.click()
```

Here the first three lines are identical but the fourth is where the selector is defined. The selector selects the element on screen that was the properties that are given as parameters. Here the text must be "Battery" and the className of the object must be "android.widget.TextView". If there exists two or more elemets that fit this selector the first one in the hierarchy would be used. The element that matches these properties is the button for accessing the battery settings and that object would be stored in the `battery_button` variable. The fifth line is used to click the specified object.

In our project we decided to use the approach with selectors. And we use different selectors to define what we want to interact with inside the application.

### 3.4.1 Our Simulation

In the image 3.1 one can see the home page of the Arc application. This is also called the Flight Info. It is one of the views that can be switched between in the application. In our simulation we wanted to use as many of the views as feasible. We also chose a long flight so we could cue as many performance tests as possible. The flight we chose was a flight from Los Angeles to Tokyo. And as explained in the Atlas section we always use the same flight but it might not be in the exact same location every simulation.

In the bottom left of the image 3.1 there is an orange button that when pressed will bring up a list of the different views. It looks like image 3.2 when pressed.
In the image 3.2 we can see some of the available views. There are a total of seven choosable views plus the option to navigate the map. All of them are on the next page with a preview of what they look like 3.3.

In our simulation we start by pressing the orange button and then we change view for five seconds and then change to the next one.

It is important that when we start the simulation the airplane is in flight. Otherwise, some of the views are not available. For example, Window View is only available if the airplane is cruising and above a certain altitude.

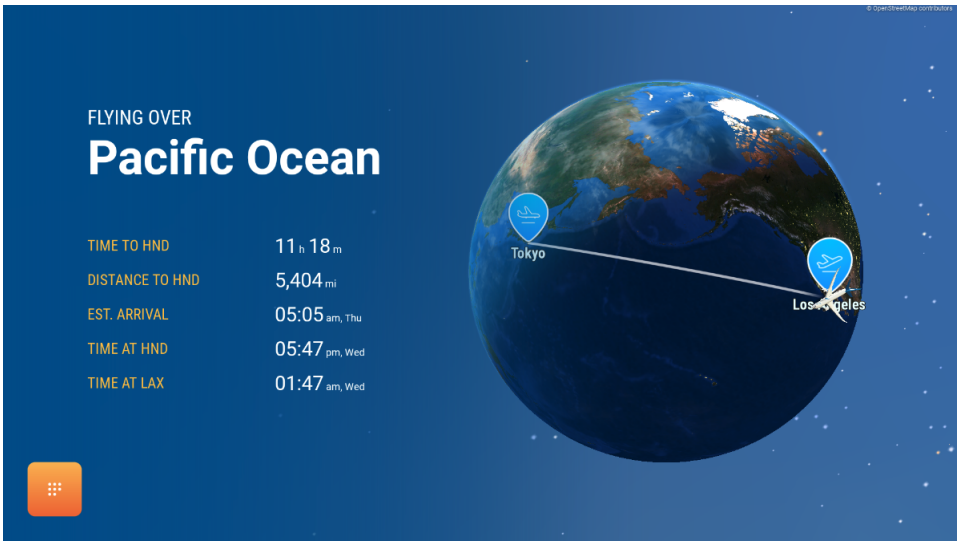



**Figure 3.1:** The default screen of the Arc application called Flight Info.

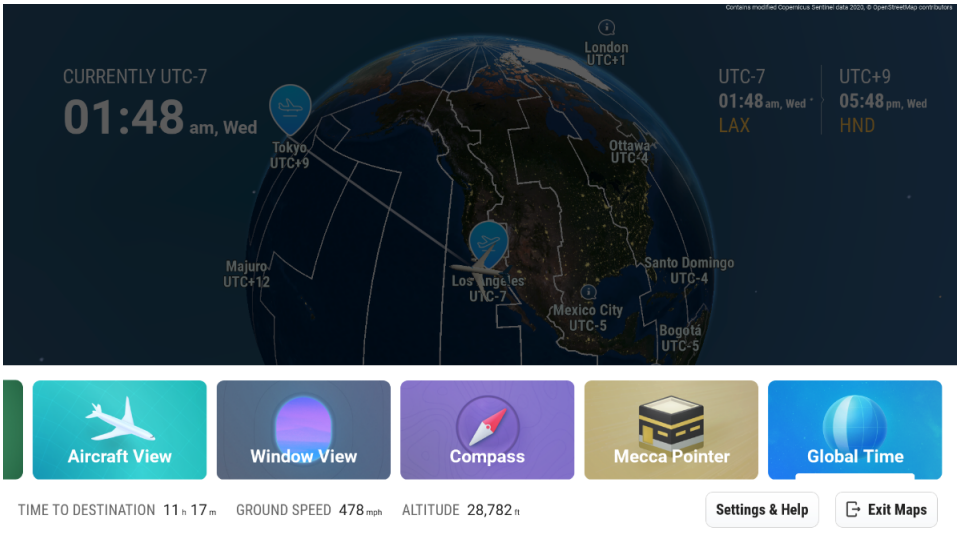
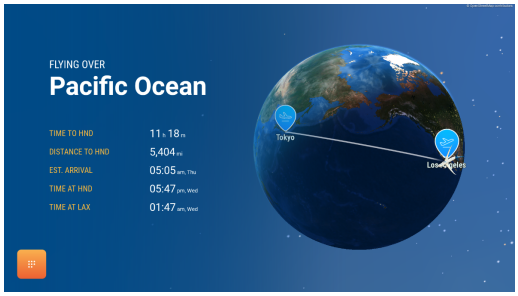
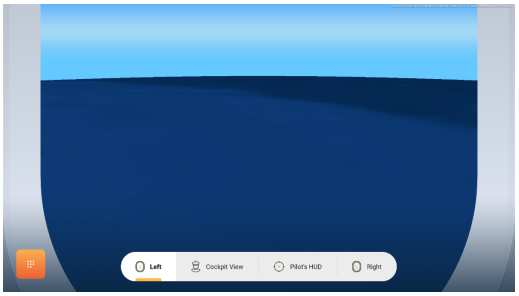



**Figure 3.2:** The list of different views

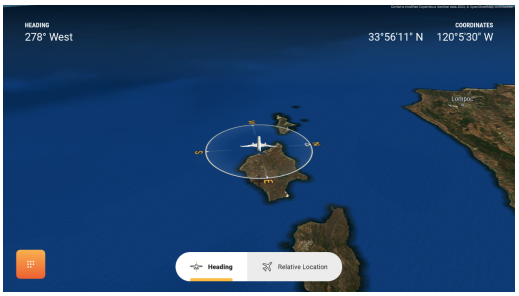



**(a)** Flight Info

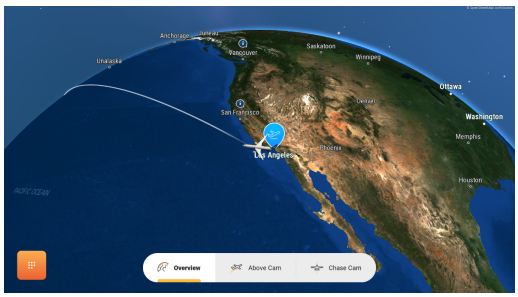



**(d)** Window View

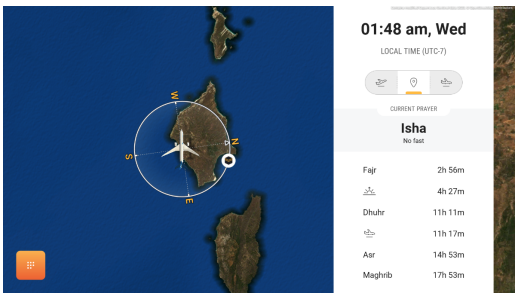

**(b)** Autoplay



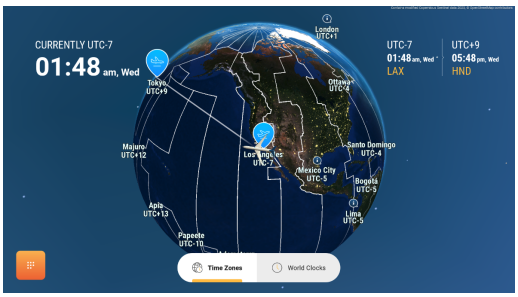

**(e)** Compass




**(c)** Aircraft View




**(f)** Mecca Pointer




**(g)** Global Time

**Figure 3.3:** All of the available views.

## 3.5 Pipeline

We built the pipeline using the automation tool Jenkins, inside of Jenkins we configure multiple other parameters needed for running our performance test automatically. Some of the parameters are: the seatback screen we want to connect to, Atlas configuration and code versions.

### 3.5.1 Arc

The Arc project consist of three different code components; the engine, the application code and the application configuration code. It is structured in a way where each of the three code components are in different repositories. To compile the program we checkout the application code and when doing that we get the latest version of the engine as well.

When the developers are updating the engine to a new version they are integrating that change into the application repository. This means that when we pull the latest from the application we receive the newest version of the engine. Conversely, if we pull an older version of the application we get an older version of the engine depending on when the latest engine update was. This is done by changing a version number in the application code so when building the application it uses the correct engine version.

Along with the application code we also need a configuration. The configuration consists of assets such as images and airline-specific configurations. The configuration code exists as a submodule inside of the application repository and when using the pipeline that is used by developers we automatically get a compatible configuration version to our application code.

### 3.5.2 Jenkins

Jenkins is an open-source automation server that allows for continuous integration and continuous delivery of software. We will use it to continuously run our performance testing code. A Jenkins pipeline is defined as a series of steps and these steps can be written in code. The code can be written in either declarative pipeline syntax or scripted pipeline syntax. Since Tactel already have a Jenkins setup and are already using scripted pipeline syntax, we choose to do the same. Since the code might be implemented in their production later it was important that our setup was familiar to theirs. The Jenkins pipelines can be configured using the Groovy language.

We are using two pipelines to run our performance tests, Pipeline A and Pipeline B. The first pipeline A is used to trigger the second pipeline B. Pipeline A consists of the following parameters:

- The IP address of the screen that is being deployed to
- A list of commits that will be scheduled to run in Pipeline B

The pipeline consists of a loop where it goes through every commit from the list and then trigger Pipeline B with that commit and the IP address of the screen. It then waits for the completion of Pipeline B until it continues the loop, and trigger Pipeline B again.

This is the pseudo code for Pipeline A:

```
PIPELINE A
    list_of_commits = input_parameter1
    screen_ip = input_parameter2

    for (commit in list_of_commits) {
        build_job("Pipeline B", commit, screen_ip)
        wait_for_completion("Pipeline B")
    }
```

Pipeline B consists of the following operations:

1. **Checkout a specified commit from the application repository**
   We specify our git repository, credentials and what branch we want to perform our action on. In our case it is the master branch. Next step is to check if our commit is empty or not. If the parameter is empty we checkout the latest master version, otherwise we checkout the specified version.

   We then run git submodule sync and git submodule update –init to sync and update our submodules. Our submodules is the configuration of the application.

2. **Checkout application configuration**
   Since we need a configuration along with the application we checkout a development version of the configuration since that usually is the most compatible. The configuration consists of different assets such as images for the buttons or the name of all the airports in all of the different supported languages.

3. **Compile the the application code**
   We compile the code using Gradle and the command:

   ```
   ./gradlew --no-daemon assembleMockedRelease
   ```

4. **Sign the apk**
   We use the Android Signing Plugin in Jenkins to sign the apk. Where we specify the following parameters: keyStoreId, keyAlias and apksToSign.

5. **Connect to the desired screen with adb**
   We first make sure that there are no previously connected devices with adb. Then we connect to the the screen with adb using the ip address:

   ```
   adb disconnect
   adb connect <ip>
   ```

6. **Install the application on the device**
   To install the application we find the mocked-release, and install it using adb. The variable APK_VARIANT is just a variant of the application that will never change and the variables default value will always be used. The command is the following:

   ```
   adb install ‘find . -type f -name ${APK_VARIANT}*mocked-release.apk‘
   ```

7. **Checkout our performance testing repository**
   We simply clone our repository each time since it is quite small in comparison to the rest of the pipeline and cost no extra time. To be able to clone the repository we need to remove the previous cloned repository. We use the following commands to first delete previous version of the repository and then clone a new one:

   ```
   rm -rf Benchmark/
   ```

   ```
   git clone <git_repo> Benchmark/
   ```

8. **Run the performance test**
   Our scripts run using python3 and we only have to run one of them since that program will start the other script. We start the performance testing script that will start the simulation script. The command to run is:

   ```
   python3 Benchmark/benchmark-exjobb/benchmarking.py
   ```

9. **Store the artifacts**
   We have a Jenkins post action that is set to always run and save all .json files. Since we create two json files both are saved. The command is:

   ```
   archiveArtifacts artifacts: '*.json'
   ```

## 3.6 Visualization of Data

To visualize the data retrieved from our script we built a simple website using Flask[26] to serve as the base. We used Dash[27] and Plotly Express[28] to make the content for the website as well as the graphs visualizing our data. A script reads the data from the two json files that each Jenkins run produces and combines them into a single json file holding all relevant data. The website can then read all this data from the master json file and visualize it in the browser. The website consists of a homepage showing mean values for all runs as well as a secondary page for each run showing all values measured during the run.

Figure 3.4 shows the home page of the website. As stated before, the home page shows mean values of each metric for each run in graphs. The graph for the last metric, mean memory usage, is further down on the page but not shown in the figure. If a data point is clicked it redirects to that runs specific page which is shown in figure 3.5. At the top of the page it shows the commit id, a link to the Gerrit page of that specific code version, and memory usage at the end of the run. Below that, the graphs for each metric are shown (the FPS graph is the only one being shown in the figure). In every graph on the page for each specific run there are timestamps that show which views were tested when those specific values were measured.

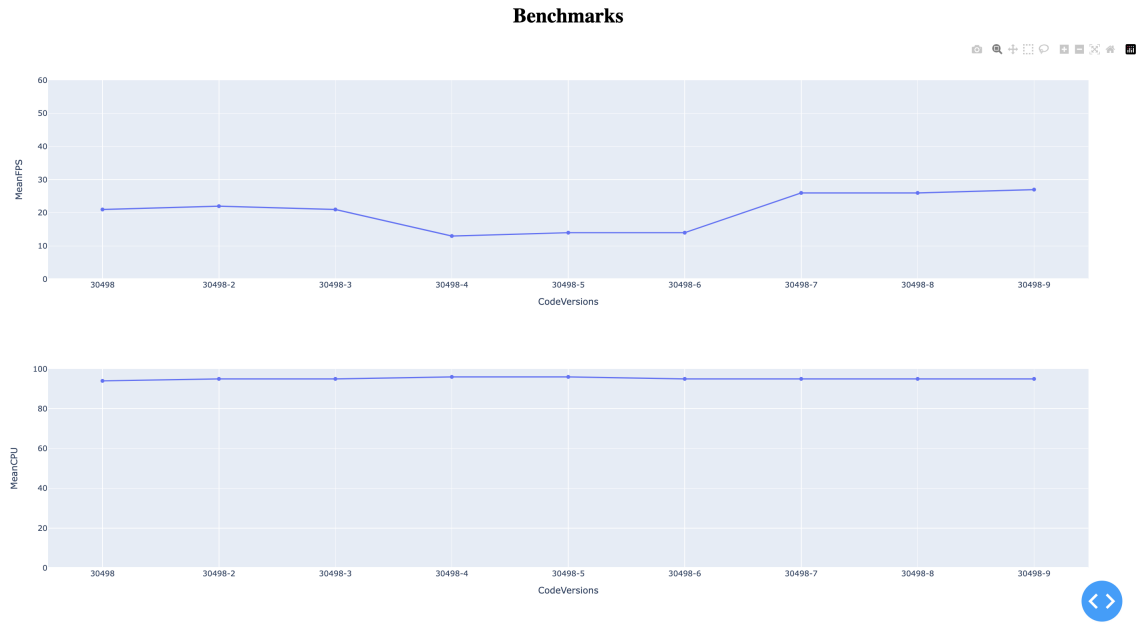



**Figure 3.4:** The home page of our website

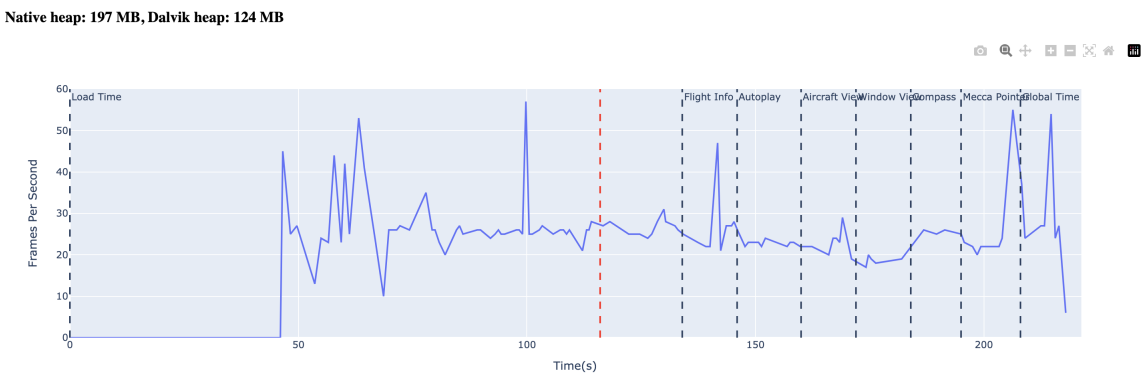



**Figure 3.5:** The page for a specific run on our website

## 3.7 Survey

This survey was done to learn more about the historical performance of Arc, assess how performance issues have been handled before, and see how the team that works with the product view its performance. The survey was composed of 13 questions of different characters. The answer for each question was in the form of grading on a scale, checkboxes or short text depending on the question. The survey was sent out to the members of the Arc development team and anyone that had any relevant experience with Arc's performance was asked to respond.

### 3.7.1 Survey Questions

- How long have you worked with Arc?
- What is your role? (For example tester, developer)
- What is you experience level in Arc?
- On what platform do you most often run Arc? (For example emulator, Astrova)

These questions above were asked because we wanted to get an understanding of the background and experience of the respondent. This information could be useful to find common patterns in the answers. We also asked about the platform on which they run Arc since performance changes significantly depending on the hardware running the application.

- How do you experience the overall performance of Arc? (on the platform you run most often)
- In your experience, how stable has Arc been over the last year? (on the platform you run most often)

The next two questions concerned the performance and stability of Arc. These questions give insight into how the respondents have experienced the performance of Arc. The answers were on a scale of 1 to 10 to give quantitative answers that are easy to analyse.

- How often do you encounter reported performance bugs? (bugs on master)
- How often do you encounter performance bugs in your own local environment?
- When was the last time you encountered a performance bug in Arc?

To get a better understanding for the personal experience of the respondent we asked about their encounters with bugs on different levels. Reported bugs are easier to spot since they show up in the tool for version control if labeled properly but information about performance bugs in the local environment is something we would not be able to get without the survey. These questions also help paint a picture of the stability of Arc and the work of the Arc team.

- Describe any performance bugs you have encountered in Arc.
- What were the symptoms of the performance bugs you encountered?
- What were the root causes of the performance bugs you encountered?
- What would help you discover performance bugs before pushing your local changes?

The last questions asked were designed with free text answers to get more specific information about the type of encountered bugs. These were qualitative questions that were designed to help us get a better view of the type of bugs that could appear in Arc, their symptoms, and root causes. The last questions was aimed to give us an idea of the kind of tools that would be useful to the people working with Arc on a daily basis.

# Chapter 4
# Results

In this chapter we will present or findings. First we will talk about the results from our performance testing runs on different code versions of Arc. Then we will prove that our tool works by introducing two simple bugs and show that our graphs capture performance degradation after the introduction of the bugs compared to a control version. Lastly, we discuss the results of our survey answered by the development team for Arc.

## 4.1 Performance Testing Results

When we were compiling the results of our performance testing tool we encountered a rather significant problem. We ran performance tests on about 50 different code versions spanning across the last year. The reason was because a single run took a lot of time, this is further discussed in the section Issues and Limitations When we plotted the graphs of the results we saw very little difference of any versions or any metric. Since the application was running on old hardware and an old Android versions we could not attribute the small changes we saw to the code versions and therefore we could only interpret them as noise in our monitoring. This can be explained by Arc being a product that has been in development for many years now and even though some minor features are still developed the status of the product is more akin to maintenance. Because of this, any performance bugs that might have appeared in Arc have since long been solved and in maintenance development there is low risk of new ones appearing. This implies that Arc is a very stable product that has held the same performance over the last year. Unfortunately for us it means that we could not get any clear results from our tool when it comes to detecting performance issues. Therefore, to prove that our tool worked as intended we had to introduce some performance degrading changes. Performance enhancing changes would not be feasible to try to introduce. To do this we asked the developers of Arc for help with coming up with realistic code changes that would lead to performance issues. Together with them we came up with two performance bugs, one that affected FPS and one that affected memory usage. For both of these bugs we began with

a control code version that we ran our performance testing tool on to get the baseline values for our metrics. We then cherry-picked the bug on top of the baseline version and ran the performance testing tool on the new version.

### 4.1.1 FPS Bug

To see if our tool would detect changes in fps we introduced a bug in the engine of Arc. The engine processes the tiles of the map that is always shown in the background of Arc. The bug consisted of making some changes to the configuration of the engine. This lead to the C++ compiler not optimizing correctly which in turn leads to performance degradation. In the app this is shown through a drop in FPS because the processing of the map takes longer than usual. This bug was a good example of a realistic performance bug because the developers informed us that this was something that had occurred in the development of the engine. A developer chose the incorrect configuration options and pushed the code. According to an engine developer this was caught before release but not before making it quite far in the pipeline. This is a situation where continually running our performance monitoring would have caught this bug early in the development process.

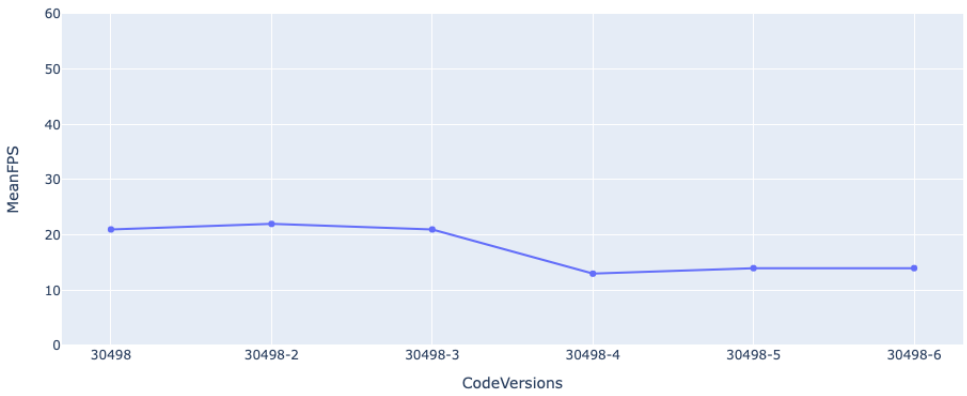



**Figure 4.1:** Graph of mean FPS

The graph 4.1 shows the mean FPS for a control version and the same version with the FPS bug implemented. The first 3 data points are from running the control and the following 3 from running the bug. This shows a clear decrease in FPS by about 8.

### 4.1.2 Memory Bug

In order to prove that our tool could detect changes in memory usage we also had to introduce a second bug. This bug simply created larger lists than necessary in the code which lead to the memory usage on the Dalvik heap to spike. This particular change was an extreme example of how not managing your data structures correctly and efficiently can lead to higher memory usage during run time. This is also something that can and probably will happen at some point during real development of an application. The memory bug is visualized in the graph 4.2.

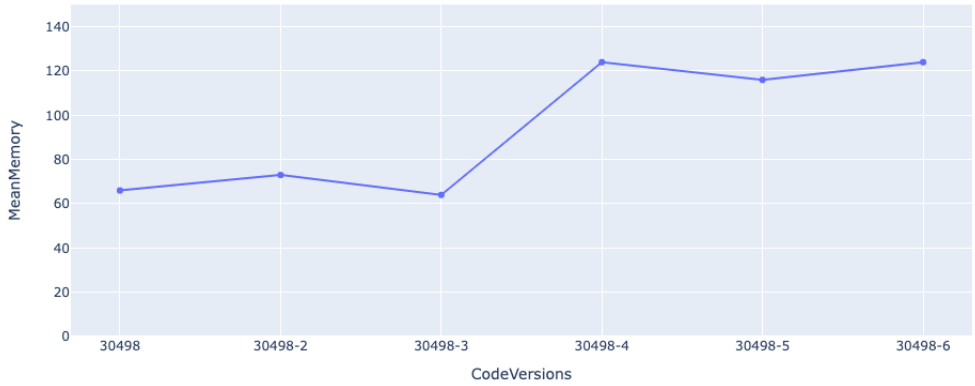



**Figure 4.2:** Graph of mean memory on Dalvik heap in mB

## 4.2 Survey

We sent out our survey on the performance of Arc to the entire Arc team and got 11 responses, mostly from developers and testers. The Arc team is a fairly small team of about 14 developers and 3 testers, so we think 11 responses is enough to encompass most of the people working directly with Arc. The people who responded had very different experience levels with Arc, ranging from 4 months to 5 years of experience with respondents evenly spread within the interval. When we asked them to categorize their own experience level with Arc 5 respondents saw themselves as experts, 4 as knowledgeable and only 2 as beginners. It was clear that the majority of the respondents most often run Arc in the emulator on their personal workstation and only two people answered that they most often run Arc on seatback devices.

When asked about how they have experienced the overall performance of Arc on a scale of 1 to 10, 1 being very bad and 10 being very good, most respondents gave high answers, correlating to a good overall performance of the app. This can be seen in figure 4.3. The lowest answer was 5 and 4 respondents graded overall performance as a 5 or 6 while 6 respondents graded the performance 8 or higher. The responses when asked about the stability of Arc over the last year were even more favourable as can be seen in figure 4.4. Out of 11 respondents, 10 graded stability at 7 or higher while only one person graded it at a 5.

We also asked the participants about how often they encounter performance bugs, both reported on master, which can be seen in figure 4.5, and in their own local environments, which can be seen in figure 4.6. From the first figure we can see that 9 respondents encounter reported bugs less than once a month with 7 of them stating that they encounter reported bugs several times a year. For bugs in the local environment respondents felt that they encountered even less bugs with 2 respondents stating they never encounter performance bugs and 3 saying that they encounter them about once a year. For both questions only one person each reported that they encounter bugs once a month and once a week. When asked about the last time they encountered a performance bug most respondents answered that they do not remember or that it was a month or more ago.

When asked about the character of these performance bugs many respondents stated low FPS, freezes and slow loading as common symptoms. The root causes for these performance

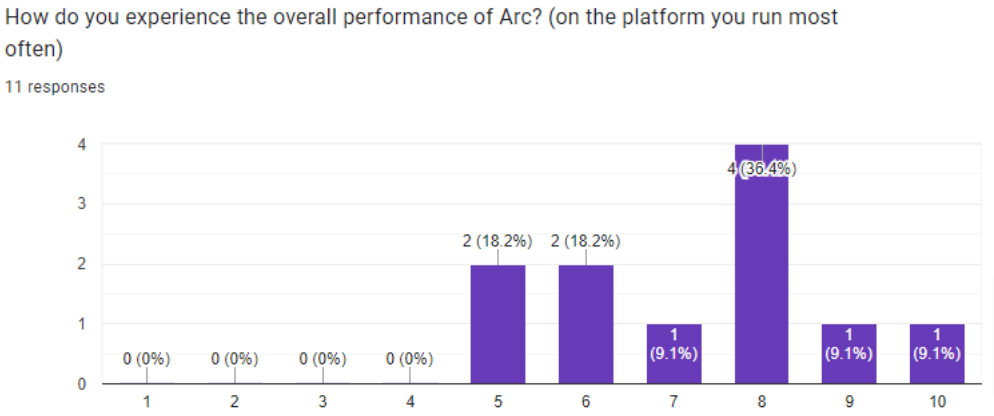



**Figure 4.3:** Overall performance of Arc

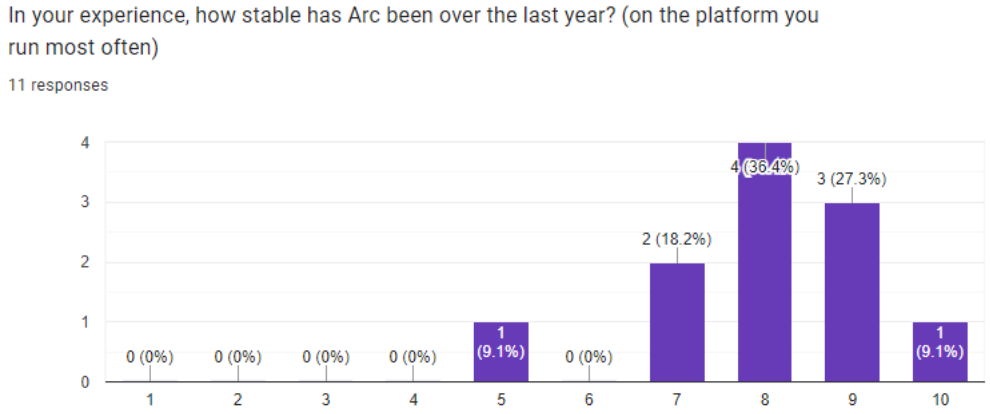



**Figure 4.4:** Stability of Arc over last year

issues varied greatly. Overriding animations, not deflating the view and unoptimized animations were some causes but several people also stated that they still do not know the cause of this performance issue. As a final question, we asked what they thought would help them discover performance bugs before pushing their local changes in the future. Many thought of automation as a possible solution, mentioning automated performance testing over time through Jenkins or other means. Some people also thought having a worse performance on their local environment would help since it frequently happened that their code ran without problem on the emulator and then ran into performance issues when ran on the seatback screens. Both the people who stated that they found performance bugs once a month or more characterized the bugs as issues with animations that could cause lag when run on the seatback screens.

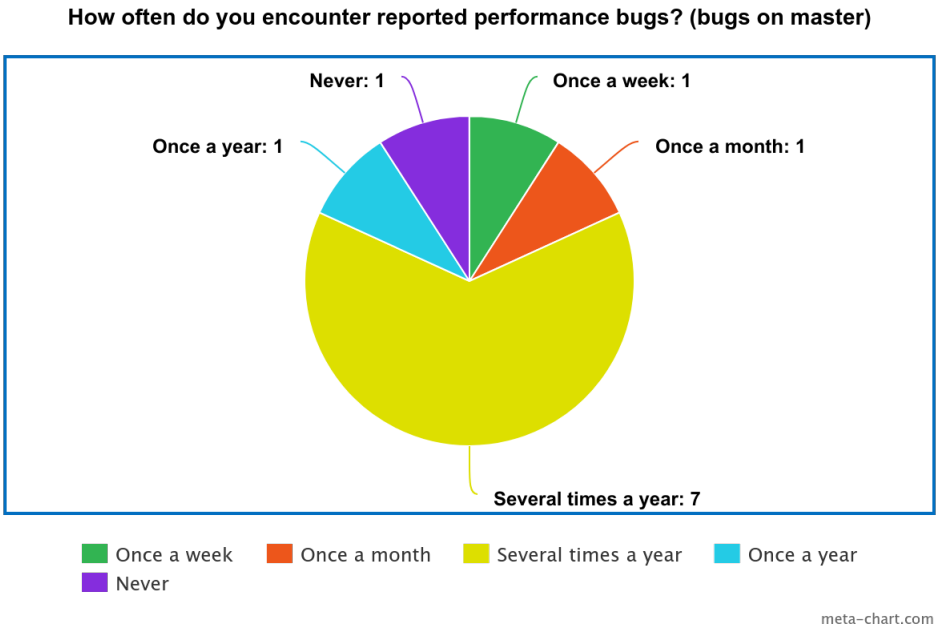



**Figure 4.5:** Encountered reported performance bugs

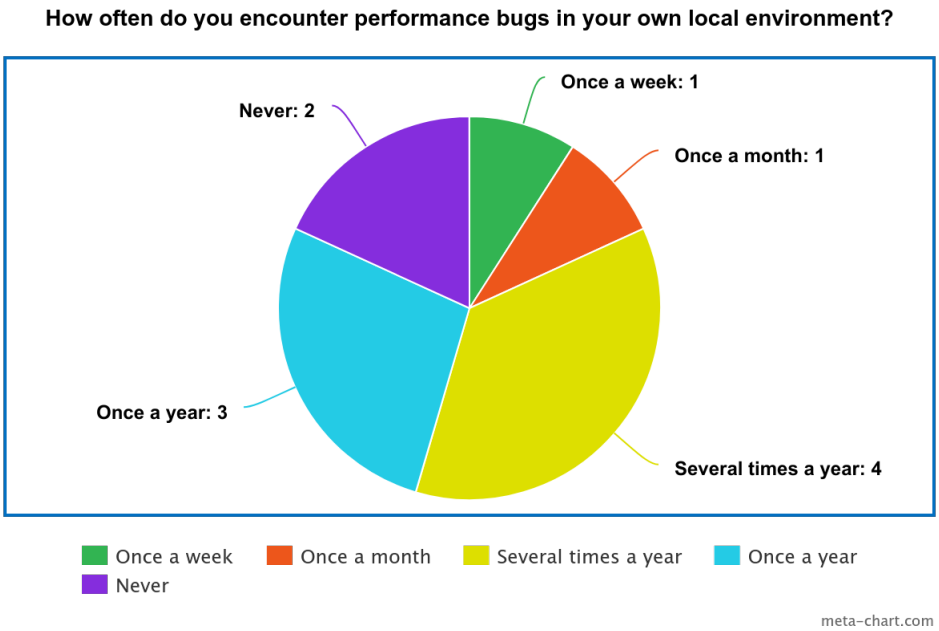



**Figure 4.6:** Encountered performance bugs in local environment

# Chapter 5
# Discussion

This chapter we will address the issues and limitations of our performance testing as well as describe what is needed to be able to perform performance testing of different code versions in a viable way. We will also talk about tools that we think would be useful if performance testing on a newer Android version. We then discuss some results from the case study and to conclude the chapter we will answer our research questions.

## 5.1 Performance Testing on Android Applications

This first section will discuss the challenges that arose when working with performance testing on an Android application. We will also highlight some limitations to have in mind when conducting performance testing in general as well as on different code versions.

### 5.1.1 Issues and Limitations

A problem we faced during our performance testing was that there were inconsistencies in the results that were not reproducible. However, these changes were relatively small as to not affect the conclusions that were being drawn. The cause for the changes in performance can be due to any number of reasons. Here are some possibilities that we discovered or discussed:

- **Hardware Limitations**
  We believe that the hardware played a role in the inconsistencies in the data. We believe this is because of the limitations of the CPU for our hardware. This is because we saw in our testing that the CPU was running at almost maximum capacity all the time.
- **UI Automation**
  When doing our UI automation the simulation had a varying degree of delay before

pressing the buttons. This was not a fault in the code since each button press was programmed to be clicked when the specified button was available. This inconsistency might be because of the hardware and the limitations of CPU performance available to the system. Or it might be because of the implementation of the UI Automator tool itself. It is more likely that it was because of the limitations of the CPU performance.

- **Background Activities**
  Even though we restarted the application before each run there can still be background tasks carried out by the system. And these are very hard to monitor and control since it will be the system that automatically performs these tasks.

We also faced an actual inconsistency in our testing that we were able to figure out the root cause of. When we started doing our performance testing we did not restart the hardware after each iteration which meant that the memory was not being freed in the same way a restart would. And we could notice when the garbage collector was running since our performance dropped. This was about every ten runs so it was not reproducible since it was not dependent on the actual code change. After we decided to add a restart of the device into our pipeline this issue was resolved.

Another issue we faced was the amount of time the pipeline took to run the simulations. On average it took about 30 minutes for a single commit to run. And the board we were using was also used during the night by the other developers to run different tests. That meant that we had to run our simulations during the day. This along with other issues like the version control meant that we were limited to a subset of all of the commits we initially wanted to run.

### 5.1.2 Atlas

As mentioned in the Atlas section in the Method chapter we decided to not have the airplane at the same point and time each performance test. And it was decided that this was an acceptable drawback since it would more accurately simulate the real world use. As can be seen from the results we found that this did not have any correlation to the results in a meaningful way. It can however be a contributing factor to the noise we found in the results.

This was positive since it would mean that the application is not really dependent on the location of the airplane and will have similar performance on different locations throughout the flight. It is still not clear if this is the case for all of the locations around the globe since we only tested one flight.

We also had another variable that could change the performance. This was the version of the Atlas service as well. And again it seemed to not have an effect on the results. This was also good since any performance change would have been a bug since the change of version should not impact the performance.

### 5.1.3 Version Control

Version control ended up being a very tedious endeavour for our thesis. As explained in the Arc section there are three different code versions that need to be taken into account. And to eliminate variables that might depend on each other, the correct way of testing would be

to only change one of the code versions, the engine, application or the configuration code. This is however not trivial since they all depend on each other and do not work if they are not compatible. The engine is not changed very often and it was easy too see if that would have affected the performance.

Changing the application means that another version of the configuration is pulled. And it was plausible that a change in performance would originate from the configuration and not the application. To see if this were the case, we needed to manually try different application versions and see if they were compatible with the configuration that was suspected to cause the issues. Sometimes they worked, sometimes they did not. This was not an effective way of performing performance testing since it was very manual.

This might be a problem in the future even if the use case would be to only run the latest version of the application. It would be desirable to have a some sort of system to see if an application is compatible with a configuration without testing it. Then a test could be implemented to automatically test a different configuration and see where the problem originated from.

## 5.2 Prerequisites for Performance Testing

The base for our case study was to explore the possibilities when it comes to performance testing different code versions. In our research we encountered many difficulties, some due to working with legacy Android versions but also general problems with getting past code versions to build correctly. We believe that setting up an automated pipeline for performance testing different code versions can be a great tool to discover what code changes have affected performance in the past and learn from them. However, there are two important criteria that have to be met to be able to do this without encountering too many problems slowing down development:

1. Robust version control
2. Reproducible builds

The project needs to have a robust system for version control and clear specifications for what is needed to build a specific code version. This includes configurations for the application, any engine or other program that the application is using and any servers that the application needs to connect to. If any of these criteria have changed and no longer are compatible with the application it will be difficult to build earlier versions and will take some time to develop workarounds, if at all possible. Release versions often follow these criteria and are packaged to be able to be built at any point in time. Unfortunately, releases are further apart in time compared to individual code versions and most bugs have probably been polished out by the time of the release. Therefore there is less to be gained from performance testing these versions to find bugs and learn from them.

Because of the problems that can arise with building earlier code versions of the application we would recommend to first develop performance testing for current versions before testing earlier code versions. We believe that there is a lot to be gained just by having any performance testing and by automatically performance testing each new version and logging the results it is possible to make a graph of historical performance. By knowing and analyzing the historical performance it is much easier to notice when performance changes happen and

hopefully try to find the causes. Only testing current versions also eliminates any problems with building the application and setting up a pipeline for it should be fairly easy for most projects.

## 5.3 Alternative Approaches to Performance Testing on Android

If we look at the limitations of our project, working with legacy Android versions, and striving for an external solution, there were several great tools available that we could not use. Perhaps the best example is Android's built in Macrobenchmark and Microbenchmark[29]. Both of these tools would have been very useful to us but we could not use them because of several reasons. Macrobenchmark is a tool that allows you to monitor performance while interacting with the UI. This would have provided us with all the functionality we were looking for and would have allowed us to measure a plethora of different performance metrics. Unfortunately it only supports API-level 23 or higher and it would have to run on the source code to function. Microbenchmark instead supports API-levels of 14 or higher which would have worked in our case if it were not for the fact that Microbenchmark tests code directly. Because of that it needs to be thoroughly integrated in the application source code and therefore did not suit our project. From our understanding, without having used these tools in practice, these are great tools to monitor and test performance of Android applications on different levels. Therefore we would recommend looking into these tools first if you are looking to performance test Android applications without our specific limitations.

According to our experiences with this project and the research we have done we believe that performance testing is an important part of testing a software project that we think most projects would benefit substantially from. Our recommended implementation for an Android application would be to use Android's own Microbenchmark or Macrobenchmark described above with a Jenkins pipeline to automate the process. This could then be run on each new version of the master branch with the results logged and visualized in a graph. This would most likely be fairly easy to implement for most Android projects and would provide a better understanding of the application's current and historical performance.

## 5.4 Case Study Bugs

In the case study multiple developers claim to have noticed bugs both on master and in their local environment. The bugs in their local environment can not be detected by our tool since it only test the things that are pushed to the master branch. The other bugs that the developers have seen on master should in theory be detectable by our tool yet we could not see any meaningful change in performance. Since we did not ask for the specific commits that where referenced it is hard to ascertain if we actually run our performance test on those bugs. If the commits were actually tested it might be that the performance degradation was too small to notice in the test. Another reason might be that the performance degradation was noticeable on an emulator but not on the hardware we tested on. That would make sense since most of the developers only use an emulator to run the program.

## 5.5 Answering Research Questions

To conclude the discussion we will look at our research questions and answer them.

**What are the challenges in performance monitoring of software that run in legacy Android environments?**
There will always be challenges when it comes to working with legacy versions of software. Our experiences showed us that performance testing is no different and might even be more affected. There were a lot of great tools that we could not use because they did not support our version, this lead us to have to use more manual methods that in some cases were unreliable. Generally we had to use several workarounds and building our tool involved many manual fixes like calculating FPS from the amount of frames that were rendered in a specific time, as well as checking memory usage more seldom because the function ran too slowly and hindered the rest of our script.

**What are the challenges in finding performance changes?**
Generally, to find performance changes you need to have a good idea of how the application usually performs. Without a frame of reference, no changes can be found. To obtain this frame of reference the application must be performance tested frequently over enough time to achieve a certainty of the application's general performance. Only after this has been done can you start looking for changes and to spot these changes the performance testing results need to be reliable. Depending on what kind of hardware you are running your application on there might be significant noise in the results and performance changes that cannot be attributed to any code changes but to how the hardware is performing. As this level of noise increases, the change also needs to be increasingly significant for it to be able to be noticed in the results.

**Is performance monitoring of different code versions a viable way to find code changes that affect performance?**
Yes, in certain cases. Performance testing of different code versions should not be the first type of performance testing that is implemented for a project. It is easier and more reliable to implement performance testing of current versions and log the results first. But in the case that this has already been implemented and there is a need for more information, especially about the historical performance of the application, testing earlier code versions can be valuable. As long as there is a robust way to build earlier code versions this type of performance testing can be more modular, new metrics could for example be introduced at any time and run on any code versions desired.

# Chapter 6
# Conclusion

The goal of this thesis was to get a better understanding of performance testing on Android and specifically research how performance testing of different code versions could be done. This was done by developing tooling to performance test a map application used on seatback screens in airplanes and perform a case study on the subject. The base for the case study was our own experiences and knowledge gained during the research and development of the tool as well as a survey performed on the development team for the application, focused on historical performance of said application.

Through strict limitations involving the Android version the application was running on we encountered many problems in the development of our tool which forced us to adapt and rethink our strategies. This led our research to encompass many different parts of performance testing on Android for different versions, while searching for solutions to the hurdles in development. In this thesis we describe the problems and the workarounds we used to solve them. We also show developers' views on performance testing through our survey in which they describe their relationship with performance issues and in what form they appear in their daily work.

All in all, we encountered many difficulties when trying to performance test different code versions on legacy Android versions but we think it could be very valuable to do if the conditions are right. Much of our problems arose because we had to adapt to the legacy Android version as well as older hardware. Another problem was that we ran into several hurdles when trying to build older code versions. Our experiences tell us that building a similar tool on new software and hardware with a robust system for building different code versions could be a valuable way to learn more about an application's performance.

# References


[1] David Curry. Android statistics (2024). https://www.businessofapps.com/data/android-statistics/. Accessed: 2024-02-05.

[2] Andrew Chen. New data shows losing 80% of mobile users is normal, and why the best apps do better. https://andrewchen.com/new-data-shows-why-losing-80-of-your-mobile-users-is-normal-and-that-the-best-apps-do-much-better/. Accessed: 2024-02-05.

[3] Appdynamics. The app attention span. https://info.appdynamics.com/rs/appdynamics/images/App%20Attention%20Span%20research%20report%20-%20final.pdf. Accessed: 2024-02-05.

[4] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empir. Softw. Eng.*, 14(2):131–164, 2009.

[5] Claire Suddath. Sunday strategist: Should airlines remove seatback screens? *Bloomberg*. https://www.bloomberg.com/news/newsletters/2019-10-13/PZB7W76KLVR501?embedded-checkout=true Accessed: 2024-02-05.

[6] Flexport Editorial Team. How are planes decommissioned, and how much value can be salvaged from their parts? https://www.flexport.com/blog/decommissioned-planes-salvage-value/#:~:text=AMARG%3A%20The%20World's%20Biggest%20Boneyard,approximately%2027%20years%20of%20service. Accessed: 2024-02-05.

[7] Eugene Belinski. Android api-levels. https://apilevels.com/. Accessed: 2024-02-05.

[8] Wikipedia. Android versions. https://en.wikipedia.org/wiki/Android_version_history. Accessed: 2024-02-05.

[9] Katie Hamilton Laura Adams, Elizabeth Burkholder. Micro-moments: Your guide to winning the shift to mobile. https://think.storage.googleapis.com/images/micromoments-guide-to-winning-shift-to-mobile-download.pdf. Accessed: 2024-02-05.

[10] bugsnag. Application stability index. https://www.bugsnag.com/wp-content/uploads/2023/06/Application-Stability-Index-2022.pdf. Accessed: 2024-02-05.

[11] Ian Molyneaux. *The Art of Application Performance Testing: From Strategy to Tools*. O'Reilly, 2014.

[12] Android. App startup time. https://developer.android.com/topic/performance/vitals/launch-time. Accessed: 2024-02-05.

[13] Jakob Nielsen. Usability engineering. pages 134–136. Academic Press, 1993.

[14] Anur Sijercic. Tiktok effects on the attention span. https://medium.com/digital-reflections/tiktok-effect-on-attention-span-12211b0a06a1. Accessed: 2024-02-05.

[15] Apple. Frame rate (ios and tvos). https://developer.apple.com/library/archive/documentation/3DDrawing/Conceptual/MTLBestPracticesGuide/FrameRate.html. Accessed: 2024-02-05.

[16] Dhivyabharathi G. Monkey testing: A guide for beginners. https://www.qatouch.com/blog/monkey-testing. Accessed: 2024-02-05.

[17] Tactel. Exploring the world below from the sky above. https://tactel.se/en/cases/exploring-the-world-below-from-the-sky-above/. Accessed: 2024-02-05.

[18] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1013–1024. ACM, 2014.

[19] Mario Linares Vásquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. How developers detect and fix performance bottlenecks in android apps. In Rainer Koschke, Jens Krinke, and Martin P. Robillard, editors, *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 352–361. IEEE Computer Society, 2015.

[20] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. Peass: A tool for identifying performance changes at code level. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 1146–1149. IEEE, 2019.

[21] Google. Android debug bridge (adb). https://developer.android.com/tools/adb. Accessed: 2024-02-05.

[22] Google. Write automated tests with ui automator. https://developer.android.com/training/testing/other-components/ui-automator. Accessed: 2024-02-05.

[23] Google. Basicsample. https://github.com/android/testing-samples/tree/main/ui/uiautomator/BasicSample, 2023. Accessed: 2024-02-05.

[24] openatx. uiautomator2. https://github.com/openatx/uiautomator2, 2023. Accessed: 2024-02-05.

[25] Google. Uiselector. https://developer.android.com/reference/androidx/test/uiautomator/UiSelector. Accessed: 2024-02-05.

[26] Flask. User's guide. https://flask.palletsprojects.com/en/3.0.x/. Accessed: 2024-02-05.

[27] Dash. Dash python user guide. https://dash.plotly.com/. Accessed: 2024-02-05.

[28] Plotly. Plotly express in python. https://plotly.com/python/plotly-express/. Accessed: 2024-02-05.

[29] Android. Benchmark your app. https://developer.android.com/topic/performance/benchmarking/benchmarking-overview. Accessed: 2024-02-05.

# Appendices

# Appendix A
# Division of Work

The work for this thesis was equally divided between the authors Jesper Grahm and Oskar Pott. Most of the work was able to be parallelized due to the nature of the development required. During development, Jesper mostly worked on the peformance testing script and the visualization website, while Oskar mostly worked on the UI Automator script and the Jenkins pipeline. Even though most of the development was parallelized all of the development was based on mutual discussions and decisions where we both were involved. Concerning this thesis, many sections were a collaborative process where we helped each other, but the majority of most sections were written by one person. Jesper wrote sections 2.2, 2.3, 3.1, 3.3, 3.7, 3.8, 4.1, 4.2, 5.2, 5.4, 6, 7 and Oskar wrote sections 2.1, 3.2, 3.4, 3.5, 3.6, 5.1. The sections not mentioned were a collaboration where we both wrote significant amounts.

**EXAMENSARBETE** Exploring Performance Testing for Different Code Versions in Legacy Android Environments
**STUDENTER** Jesper Grahm, Oskar Pott
**HANDLEDARE** Alexandru Dura (LTH), Niklas Hedström (Tactel)
**EXAMINATOR** Görel Hedin (LTH)

# Kan man testa hur väl en Android app fungerar? Hur gör man för att testa det?

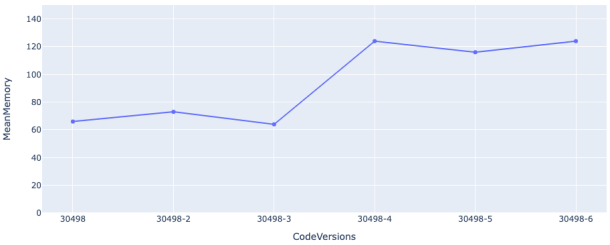
POPULÄRVETENSKAPLIG SAMMANFATTNING **Jesper Grahm, Oskar Pott**

Detta arbete gick ut på att göra ett verktyg som kunde testa hur bra en app presterade. Ett verktyg som automatiserade prestandatestning för en Android app utvecklades samt en undersökning i hur man bör göra denna typ av testning.

Mobilappar är något som används väldigt mycket varje dag. De flesta appar som vi använder fungerar också som de ska. En av anledningarna är för att de som utvecklar appar testar dem i förväg för att säkerställa att de presterar som förväntat. Det finns många olika typer av testning man kan göra av appar. Man kan dels testa prestandan och det är det som vi har fokuserat på, det innebär att man testar saker som laddningstid och hur många gånger bilden på skärmen uppdaterar sig.

Vi utvecklade ett verktyg som testade prestandan på en Android app, vi gjorde detta för att expandera på det verktyg som redan fanns på det företag där exjobbet utfördes. Vi ville också testa flera olika versioner av samma app för att märka vilka ändringar mellan versionerna som kan ha gett upphov till sämre prestanda. Det gjorde det genom att "låtsas" vara en människa och tryckte på skärmen åt oss medan ett annat program mätte prestandan. Verktyget kan visa en sekvens av versioner och deras repsektive prestanda via en hemsida. Man kan också se en mer detaljerad vy av varje version genom att bara klicka på en av versionerna på hemsidan. Vårt verktyg kan visa hur den versionen presterade under tiden för simuleringen.

Arbetet innefattade även en fallstudie som fokuserade på vår upplevelse av att utveckla verktyget. Som en del av fallstudien inkluderades även en enkät som utvecklarna fick svara på. Frågorna handlade om hur utvecklarna har upplevt prestandan i deras app.




Vårt verktyg fungerade och vi kunde se skillnader mellan olika versioner av appen. En version där vi introducerade en ändring som var menat att göra prestandan sämre kunde således upptäckas av vårt verktyg, som man kan se i bilden.