Mag. Matthias Fuchs, Bakk. BSc

Maintaining a Continuous Integration System for the Free and Open Source Software Project Catrobat

Master's Thesis
to achieve the university degree of
Diplom-Ingenieur
Master's degree programme: Softwareentwicklung-Wirtschaft

submitted to

Graz University of Technology

Supervisor
Dipl-Ing. Dr.techn. Christian Schindler

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Slany

Graz, April 2019

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

Division of labour is a necessary ingredient for any complex software system. However, such division of labour comes at a price: There is a need for coordination. Especially the integration of separately developed software pieces into one system proved problematic historically. The continuous integration method avoids typical integration issues via automation and frequent repetition. Instead of performing tedious and error-prone integration steps manually at rare instances they are performed frequently and automatically.

This thesis describes the maintenance of the Jenkins continuous integration system for the free and open source software project Catrobat, which includes two Android applications. After changes to external Android dependencies in 2015 the Catrobat tests did not work reliably anymore, leading to unstable Jenkins builds and practically an abolishment of continuous integration.

The main part of this thesis focuses on how these dependencies can be handled, so that stable builds are possible again. This includes the use of the modern container technology Docker to isolate resources, such as the Android Emulator, between concurrent build jobs. The installation of the Android dependencies as well as the management of the Android Emulator are investigated too.

Another key aspect of the thesis is improving the maintainability of the Catrobat Jenkins system. The best practice of configuration as code is applied to create the build jobs on Jenkins. The importance of documentation and policies is also discussed.

With continuous integration for Catrobat operating again further needs of the developers and product owners like performance of build jobs and code coverage tracking are considered.

This thesis highlights the complexity and fragility of continuous integration for Android applications and also emphasises the necessity of considering both the requirements of users of a continuous integration system and its maintainability.

Kurzfassung

Arbeitsteilung ist eine Voraussetzung für komplexe Softwaresysteme. Diese Arbeitsteilung hat jedoch einen Preis: Es entsteht Koordinierungsaufwand. Historisch betrachtet führte vor allem die Integration unabhängig entwickelter Software in ein System zu Problemen. Continuous Integration automatisiert diesen Integrationsprozess in Form von Builds und erhöht dessen Frequenz wodurch Integrationsprobleme vermieden werden können. Mühsame und fehleranfällige manuelle Integrationsschritte fallen weg.

Diese Diplomarbeit beschreibt die Wartung von Jenkins, einem Continuous Integration System, für das Open-Source-Software-Projekt Catrobat. Catrobat beinhaltet unter anderem zwei Android-Applikationen, deren Tests nach der Änderung von externen Android-Abhängigkeiten ab 2015 nicht mehr stabil liefen. Das führte auch zu instabilen Resultaten auf Jenkins und praktisch der Rücknahme von Continuous Integration.

Der Hauptteil der Arbeit befasst sich mit der Verwaltung dieser Abhängigkeiten um wieder stabile Builds zu ermöglichen. Dabei kommen moderne Technologien wie etwa Docker zum Einsatz, was ermöglicht Prozesse, wie den Android Emulator, zwischen gleichzeitig laufenden Build-Aufträgen zu isolieren. Auch die Installation von Android-Abhängigkeiten und das Verwalten des Android Emulators wird behandelt.

Ein anderer Schwerpunkt ist die Verbesserung der Wartbarkeit von Catrobat Jenkins. Dafür werden bewährte Praktiken wie das sogenannte Configuration-as-Code verwendet um die Jenkins-Build-Aufträge zu erzeugen, sowie die Dokumentation erweitert.

Mit schließlich wieder funktionierender Continuous Integration werden weitere Anforderungen der Entwickler und Product Owner behandelt, wie einer Verringerung der Build-Dauer oder das Sammeln der Testabdeckung.

Diese Diplomarbeit hat die Komplexität und Fragilität von Continuous Integration für Android-Applikationen aufgezeigt und dabei die Notwendigkeit hervorgehoben sowohl auf Nutzeranforderungen als auch auf die Wartbarkeit des Systems zu achten.

Contents

List of Figures

List of Listings

1 Introduction

Software development is a complex endeavour. To manage the often large teams developing software, different methodologies have been envisioned and applied in the last decades. No matter which software development process is used, there is a set of tools that helps developers perform their work.

When many people work together they have to integrate their pieces of work into a whole system. If such an integration is done rarely the risk increases that the integration does not work flawlessly. Instead, with continuous integration developers integrate their work regularly, ideally multiple times a day, into the mainline of a version control system. Their changes are then build and a set of tests is run. A continuous integration system is used to automatically perform these steps on every change. As such, continuous integration can be seen as a best practice in the software development process (Leffingwell, 2007, pages 169–177), that reduces the risks of teamwork with the help of tools that automate most of the necessary steps (Fowler, 2006).

This thesis discusses the challenges faced and the implemented solution approaches while maintaining a continuous integration system for the free and open source project Catrobat. Most parts of the challenges as well as the solutions apply to any continuous integration system and not just Jenkins[1], which was used here.

[1]Jenkins website: `https://jenkins.io`

1.1 Motivation

The motivation of this thesis is twofold. On the one hand the motivation is to provide developers a continuous integration system that helps them again in their workflow, that even becomes an integral part of that workflow. On the other hand the motivation is to ensure that the resulting continuous integration system is maintainable, and therefore feasible in the long-run.

Jenkins was used as continuous integration system for Catrobat (Reisenberger, 2014). Yet by the end of 2016 the Catrobat Jenkins system was in disarray. Many of the build jobs took hours to complete with hundreds of failed tests. The build results were mostly ignored by developers. At the same time the Jenkins version was outdated, as were its plugins, and the used operating systems, which posed security issues. So while the system was once usable and beneficial it deteriorated due to lack of maintenance and rigour.

This highlights that there are trade-offs between providing a usable system that is also maintainable. Programmers have to face similar trade-offs: They have to provide their customers with new features in the short term and at the same time ensure that they will be able to do so in the foreseeable future. In the case of continuous integration the customers are the developers and managers whose programs are built and tested on the system. For the Catrobat project the developers are internal customers and the continuous integration system is not a product on its own. Therefore, adding new features to the continuous integration system is part of maintenance.

The concept of technical debt introduced by Ward Cunningham also applies to continuous integration when loosening the meaning of code:

> "Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. [...] The danger occurs when the debt is not repaid. Any minute spent on not-quite-right code counts as interest on that debt." (Cunningham, 1993)

This notion was considered in most decisions that lead to this thesis, balancing the promptly implementation of new features with long-term maintainability.

1.2 Thesis Outline

The following chapter describes continuous integration with a definition first, preconditions for continuous integration second, and finally benefits of continuous integration like the possibility for continuous delivery.

Chapter 3 describes the Catrobat project and introduces the Catrobat Jenkins team and their focus of activities.

The main part of the thesis is Chapter 4, which discusses a subset of the challenges faced maintaining the continuous integration system for Catrobat. It highlights that documentation and policies have a place, even in agile teams, countering possible knowledge loss due to high turnover. Section 4.2 shows that configuration as code can help reduce documentation and at the same time lead to better maintained systems. Section 4.3 focuses on how to configure build jobs a maintainable way. Following sections discuss the measures taken to improve the stability of the build jobs by detecting and handling flaky tests, making the build jobs more independent, and by improving the interaction with the Android Emulator. Section 4.7 describes the steps taken to improve the performance of the build jobs.

In Chapter 5 a conclusion of the thesis is given.

2 Continuous Integration

This chapter provides an overview of continuous integration. A definition, preconditions, and benefits of continuous integration will be discussed. Consequently, this chapter provides a foundation for the rest of this thesis.

2.1 Definition

Continuous integration is a high-level term. Its meaning and scope evolved over time (Duvall, Matyas and Glover, 2007, pages 36–37). Yet despite of the time, the main issue is how multiple people work together on one project or product, without causing each other too many issues. Especially when they combine the results of their work to create one system. This is one of the drawbacks of division of labour: People are working mostly independently on software components that have to be integrated at some point into a whole system.

This notion is reflected in what Brooks Jr (1995, page 133) refers to as integration. In the Mythical Man-Month integration is to integrate a small program or software component a person works on into a whole system. The system can then be tested and deployed.

A similar sentiment is also shared by Beck and Andres (2005, pages 49–50). For them division of labour in programming is a "divide, conquer, and integrate problem" where integration is the most riskiest aspect. To reduce that risk Beck and Andres (2005) suggest to integrate every couple hours, which they call continuous integration. After committing (integrating) code changes to the mainline a build is triggered automatically. Such a build includes building the most recent state of the mainline and running tests automatically. On build failures the responsible developer, the one who committed the changes, is ideally informed automatically. Nonetheless, having working builds is the responsibility of the whole team, fixing broken builds has highest priority. To be able to integrate every few hours the work

needs to be split into small parts that can each be tackled on their own. This is where the other extreme programming practices come into play.

So instead of performing the cumbersome task of integration even less frequently continuous integration suggests the opposite. This follows a notion of Fowler (2011b): "if it hurts, do it more often". Only then measures will be implemented to reduce the burden of integration. The higher frequency of integration provides fast feedback leading to further improvements, increased experience, and enhanced automation.

Yet continuous integration is not just committing source code changes to mainline. It also includes the steps triggered automatically by the commit (Fowler, 2006). An integration can only be considered successful if the integrated code built, if all the tests kept working, and if any further checks succeeded. Nonetheless, continuous integration does not relieve developers of the responsibility to try and test their code changes locally. Otherwise, there would be many failed builds.

There are tools that help with tasks related to continuous integration, which in general run on a their own server. Such a continuous integration system supports developers with far-reaching automation, avoiding tedious and error-prone manual work. The terms continuous integration and continuous integration system are often used interchangeable. This thesis tries to differentiate between the two, considering continuous integration as concept that is practically applied with the help of a continuous integration system. The used references are often not that strict, which leads to some compromises in this regard.

2.2 Preconditions

For continuous integration to work several preconditions have to be fulfilled in practice (Duvall, Matyas and Glover, 2007, pages 3–12):

1. A stand-alone script is used to build the software.
2. A version control system is applied.
3. A continuous integration system runs round-the-clock, therefore on its own server.
4. Feedback mechanisms are in place.
5. Developers abide by basic processes.

2.2.1 Build Script

Continuous integration systems need to build software, execute tests, perform static analysis and further checking. All of these steps have to be automated. These steps should also work on local machines, so that developers can run them easily during development. Having support directly in an integrated development environment such as IntelliJ Idea[1]is also beneficial for productivity.

This is where build scripts (Duvall, Matyas and Glover, 2007, page 10), also called build automation tools, come in. They are used for all of these steps. Some of the build automation tools even handle the retrieval and installation of dependencies, which is especially useful for continuous integration systems. There are many different build automation tools, often directly related to a single programming language. Gradle[2], Apache Maven[3], and Apache Ant[4] are commonly used for the Java programming language and also for languages relying on the Java Virtual Machine.

For example, with Gradle assembling the programs can be done via the `gradle assemble` command (Nizet et al., 2018). Executing all tests is done with the intuitive command `gradle test`, while `gradle check` additionally also executes static analysis. The `gradle build` command includes assembling, testing, and also static analysis.

2.2.2 Version Control System

Continuous integration only works when changes to the source code are detected automatically. Naturally the integration process, including building and testing the software, would not be triggered otherwise.

The attribution of changes is also important for continuous integration: Who changed what, and when? This information is necessary to inform the originator of a change that broke a build.

Version control systems provide both of these features and multiple more. A version control system keeps track of all the files under its supervision in the form of a repository. Developers, and also a continuous integration

[1]IntelliJ Idea website: `https://www.jetbrains.com/idea`
[2]Gradle website: `https://gradle.org`
[3]Apache Maven website: `https://maven.apache.org`
[4]Apache Ant website: `https://ant.apache.org`

system, can retrieve the current state of the software from a version control system by performing a checkout. Changes are tracked in the form of commits. A commit consists in general of the following information:

- A unique identification of the commit, for example, in the form of an increasing revision number or a unique checksum.
- The author who performed the commit.
- A timestamp when the commit was performed.
- The changes, that is the addition of new files and the removal or modification of existing files.

Commits can be reverted, which is especially useful for commits that broke builds and where writing a proper fix for the build would be too time consuming.

Continuous integration systems can be informed of changes directly by version control systems such as git[5] via hooks (Chacon, 2009, pages 190–202). As fallback version control systems can be polled frequently to check for new changes, in case they do not provide hooks or writing hooks is too complicated.

Most version control systems support independent work in the form of branches. There is a mainline branch, often called trunk, master, or develop, that will be used as base for future releases. At the same time developers can work in other branches, without affecting the mainline. Like any changes branches can be integrated themselves into other branches, which is called merging.

Merging branches was very complicated historically, especially for long-lasting branches. This was caused by so called merge conflicts that arise, for example, when files were modified at similar locations in both the target branch of the merge and the branch to merge. For the version control system it is not clear which change to apply. The user has to decide in such cases. Newer version control systems like git improved these merge algorithms leading to fewer conflicts compared with older version control systems like Subversion[6] (Chacon, 2009, page 57).

Traditional version control systems such as Subversion were centralized (Chacon, 2009, page 3), that means there was a dedicated server with the

[5]Git website: `https://git-scm.com`

[6]Subversion website: `https://subversion.apache.org`

version control system. This server contained the complete commit history and was considered the single source of truth for the code. Everyone else had only the current state locally and committed their changes directly to this server. Newer systems like git are distributed (Chacon, 2009, page 4–7). Every checkout contains the complete history. Developers can commit locally and also create branches locally.

With the newer tools also new workflows evolved. For example, the pull request feature introduced by code hosting sites (Gousios, Pinzger and Deursen, 2014). A pull request is when users work in their own branches and even repositories and then do not integrate these branches into the mainline or a different branch themselves. Instead they request others to integrate their changes in form of a pull request they create on a code hosting site.

The git-based hosting service GitHub[7] creates an internal branch for each pull request which acts as copy of the target branch with the changes integrated. Continuous integration systems can then build these pull requests and detect issues before they affect the mainline. The changes can also be reviewed by maintainers, before they enter the mainline. All modifications to the target branch are tracked. Thus merge conflicts that did not exist initially but were later caused by interim commits are detected. When all issues have been ironed out the pull request is accepted and merged into the mainline. Therefore, pull requests can be thought of as a transparent dry run of an integration with increased transparency and opportunities for communication and community interaction.

There are discussions whether creating branches for features can still be considered continuous integration, as they result in committing to the mainline less regularly, often not daily anymore (Fowler, 2009). The same arguments could apply to pull requests. Yet pull requests are a form of integration themselves. And this discussion was lead in the late 2000s when pull requests were a very new feature. Since then pull requests have been applied industry-wide and are commonly supported by tools like GitHub, Bitbucket[8], and GitLab[9]. All of these tools support the interaction of continuous integration systems with pull requests, including checking

[7]GitHub website: `https://github.com`
[8]Bitbucket website: `https://bitbucket.org`
[9]GitLab website: `https://about.gitlab.com`

out the code and updating the pull request with the build job status. The author of this thesis considers a pull request workflow still as continuous integration. Provided that pull requests do not stay open for too long. When multiple open pull requests accumulate the risk of integration issues between them increases again. Pull requests can be a considered a trade off between having a mainline that potentially always works and the fast integration of changes. Thus continuous integration not only depends on the tooling but also on the way the tooling is applied, see also Section 2.2.5

In conclusion, version control systems are not only a necessity for continuous integration, they are also a best practice for software development in general (Hunt and Thomas, 2000, pages 86–89).

2.2.3 Continuous Integration System

As discussed above continuous integration involves many steps. These steps have to run automatically. Therefore, a system to execute them is necessary. This continuous integration system could be realised with custom scripts, or better, existing continuous integration systems can be used (Duvall, Matyas and Glover, 2007, pages 8–9). In any case a continuous integration system has to run round-the-clock. This alone mandates a dedicated computer, a server, to run the software.

There are many different continuous integration systems, some run on dedicated machines, others run in the cloud[10], while some can combine both modes of operation. Also, the feature set and pricing models differ. There are more than 50 systems available (Stackify, 2017), for example: Jenkins[11], Travis[12], TeamCity[13], and CircleCI[14].

This thesis will focus on the application of the very popular free and open source software Jenkins.

For continuous integration one of the many existing systems should be selected, instead of writing a customised solution. This both safes time and gives access to the large communities of existing users

[10]For a definition of cloud computing refer to Mell and Grance (2011).
[11]Jenkins website: `https://jenkins.io`
[12]Travis website: `https://travis-ci.org`
[13]TeamCity website: `https://www.jetbrains.com/teamcity`
[14]CircleCI website: `https://circleci.com`

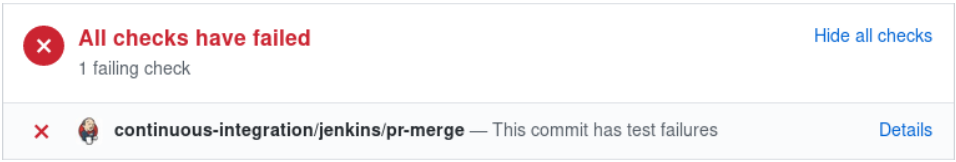


Figure 2.1: The continuous integration job failed for this Catroid pull request. [Screenshot taken by the author of this thesis.]

2.2.4 Feedback Mechanisms

In case of a build failure or when a given quality threshold is breached it is important to inform the right people as fast as possible (Duvall, Matyas and Glover, 2007, pages 10–11, pages 203–222). Duvall, Matyas and Glover (2007) call this continuous feedback. Therefore, it is even beneficial to give feedback right away and not to wait for a build to finish if issues were found already. That way the right people can take action to resolve an issue as early as possible.

The feedback can be provided by different means, for example, via e-mail, RSS-feeds, hooks in collaboration systems such as Slack[15], or directly in the web interface of a source hosting platform such as GitHub.

Especially the integration with pull requests is useful as there is a small and accurate target audience for the feedback. The right person is informed. Figure 2.1 shows an example for a pull request where the Jenkins build failed. This information is placed at the bottom of the pull request and therefore reaches the correct audience: The person working on the pull request, potential reviewers, and the person to merge the pull request. GitHub also shows check marks in the commit history for each commit built by the continuous integration system. In Figure 2.2 a Paintroid commit is shown that built without issues on Jenkins. Both of these communication channels are subtle and directed, which makes them comfortable to work with. In these cases Jenkins for Catrobat provides the information as soon as an issue is detected, even before the build finished. This reduces the time developers have to wait before they are informed of problems they have to fix.

Informing the right people in cases of build failures or other issues is important. Otherwise, the mainline might be broken for long durations

[15]Slack website: `https://slack.com`

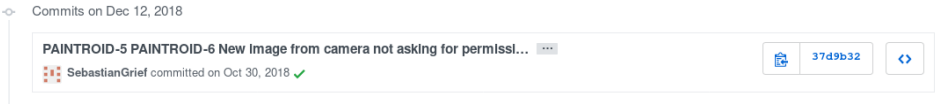


Figure 2.2: For this specific Paintroid commit all the checks on Jenkins passed which is shown by the green check mark. Clicking the check mark provides more details, such as a direct link to the build results. In case of a build error there would be a red cross. [Screenshot taken by the author of this thesis.]

before people recognise. All relevant continuous integration systems provide ample ways of informing its users.

2.2.5 Developers

At the heart of any process are the people executing it. Despite all the automation continuous integration is no exception. The developers have to follow many practices for continuous integration to work (Duvall, Matyas and Glover, 2007, pages 39–45).

For continuous integration to work properly developers have to build the code locally and also execute tests locally before committing any changes to mainline. This practice is called private build (Duvall, Matyas and Glover, 2007, pages 26–28). Broken code should never be committed. Otherwise, the mainline would be in a state of constant breakage if people used it as testbed. Developers would have to take care to avoid updating to a broken version of the mainline. Very important in this regard is that developers write tests for their code. The benefits of continuous integration heavily rely on the quality and quantity of tests written. If there are only few tests it is unlikely that the continuous integration system will detect issues.

Whenever a build is broken it has to be fixed immediately. Broken builds should be the exception, not the norm. As mentioned above private builds and pull requests are a preventative measure in this regard. The continuous integration system will only be recognised by developers if the builds are in working state in general. If the builds always fail developers will start to ignore the build results.

Another important aspect is to limit the size of changes. This makes reviewing the code easier and also decreases the risk of long lasting feature branches. These small changes should regularly be committed to the mainline or integrated via pull requests. When pull requests take long to be

merged there is a risk that their size increases: When a developer is blocked by an unmerged pull request they continue their work, accumulating more code that has not been integrated yet. Therefore, it is important that pull requests are acted upon quickly.

In summary, continuous integration is a practice that requires discipline and constant reflection. What was continuous integration yesterday might silently deteriorate into a different practice with fewer benefits.

2.3 Benefits

Continuous integration provides many benefits (Duvall, Matyas and Glover, 2007, pages 29–32, pages 39–40, pages 47–63), some of them are explored in this section.

Probably the most apparent benefit is automating otherwise tedious, mundane, and error prone tasks. People can work on value generating tasks instead. This automation also enables further practices that were not possible before. For example, trends can be tracked reliably: Is the code quality corresponding a given measure improving? Is the test coverage increasing? Is the time between broken builds increasing? Continuous integration systems even support thresholds for quality metrics. A build would fail if static analysis tools detected issues.

This leads to better transparency and understanding of the quality of a product and also to increased trust in the product. This transparency reduces risks in planning, helps managers to form decisions and can also lead to more satisfied customers. Furthermore, the increased quality can have positive effects on motivation of the staff as DeMarco and Lister (2013, pages 19–23) pointed out.

Performing the integration frequently reduces the integration related risks, like merge conflicts. At the same time the number of commits increases, while their size decreases. These smaller commits make the changes easier to review, they tell a clear story of what was changed. When an issue was introduced reverting such a small commit keeps the negative side-effects low.

Rapid feedback is also a huge benefit: Issues are detected very early and the correct people are informed. That way the developers can fix an issue while they are still familiar with the relevant source code. The confidence of

all stakeholders in the product increases. For developers it is some form of insurance that their changes did not break any of the tested functionality. Especially for refactoring tests in combination with continuous integration act as safety-net (Fowler, 1999, page 7).

The jobs of a continuous integration systems are executed in a defined environment. That reduces the risks of misconfiguration or stale caches that exist on local machines. Given well written tests this leads to reproducible builds (Fowler, 2010), see also Section 4.5. Continuous integration can even help detect the case when developers forgot to commit some necessary files.

Especially for management it is important to know if their product can be released. Here continuous integration gives indications with broken or working builds. Artefacts of the the build, such as executables, can be tested manually, deployed to test machines, and even deployed directly to customers.

The practice to also automate the deployment pipeline is called continuous delivery (Humble and Farley, 2010, pages 3–4). Continuous delivery can be thought of as continuous integration with additional release steps, such as releasing an alpha version that can then be promoted to a beta or even a release version directly in the continuous integration system. Suddenly releases become a regular event that is one or a few mouse clicks away, instead of massive endeavours in the last moments of a project. Of course, as mentioned before, the quality of tests and their coverage is very important in this regard.

Overall, continuous integration leads to reduced risk and increased transparency. Managers and developers can have confidence in their software.

3 The Catrobat Project

The Catrobat project[1] is an agile free and open source software (Free Software Foundation, 2017) project started in 2010 (Slany, 2012). Mostly students of the Technical University of Graz are working on Catrobat (Müller, Schindler and Slany, 2019).

The main products of the project are the Pocket Code and Pocket Paint applications. The focus of Pocket Code is to introduce children into the world of programming, using the Catrobat visual programming language (Slany, 2012). With Pocket Paint pictures can be drawn that can then be used in Catrobat programs, for example, drawing the content of a small Tic Tac Toe game.

Pocket Code can be run as an Android application[2], called Catroid internally, or from within the web browser, there is also a version for iOS[3] called Catty internally. The Android version of Pocket Paint[4] is called Paintroid internally.

In support for the developers there is a Jenkins team that works on continuous integration. Previous members include Burtscher (Burtscher, 2016) and Reisenberger (Reisenberger, 2014). The author of this thesis started working on the Jenkins team in November 2016 with a roughly six month hiatus at the end of 2017. During most of the time in the project the author had the role as coordinator, setting goals and tasks for the Jenkins team, coordinating its members, communicating with other teams, and of course also working on these tasks. The main focus was continuous integration, which is also discussed in this thesis. Michael Musenbrock, who was coordinator during the hiatus of the author, and further team members had a

[1] Catrobat website: `https://www.catrobat.org`

[2] Pocket Code for Android: `https://play.google.com/store/apps/details?id=org.catrobat.catroid`

[3] Pocket Code for iOS: `https://itunes.apple.com/app/pocket-code/id1117935892`

[4] Pocket Paint for Android: `https://play.google.com/store/apps/details?id=org.catrobat.paintroid`

similar focus. Kirshan Kumar Luhana (Luhana, Schindler and Slany, 2018) and Rainer Lankmayr focused instead on support for continuous delivery of Pocket Code and also Pocket Paint.

The Catrobat staff consists mostly of students performing their work in the course of a practical, a thesis, or a bachelor's project. In general this leads to a short and defined participation within the project. The amount of time people invest weekly heavily depends on the current state of their studies and their personal motivation. At the same time there are moments when smaller Catrobat teams have barely any members. As a result, successful knowledge transfer is especially important and training new team members is common. This adds additional challenges, not regularly faced in the industry.

4 Practical Challenges

Despite all efforts, at the end of 2016 the Jenkins system for the Catrobat project was in a state of disarray. This chapter describes issues that the Catrobat Jenkins team faced and measures that were taken to improve the situation.

At the beginning of any change there needs to be an assessment of the current situation. These were some of the encountered issues:

- The installed version of Jenkins had not been updated in over a year.
- The installed plugins were out-of-date.
- Different Linux distributions were used for the Jenkins master and Jenkins slave installations.
- The configuration needed for the master and the slaves was not fully documented.
- The configuration steps needed for Jenkins were not fully documented.
- The jobs were manually created and had little common behaviour.
- What jobs were still needed was not clear, neither who created them initially and for what purpose.
- Most of the jobs had not run successfully in months. They failed constantly.
- Unclear direction of the project.
- Unclear what people were working on.
- The Android Emulator used was outdated since the current version was not supported by the used Jenkins plugin.
- The Android Emulator crashed regularly during the execution of tests.
- There were hundreds of failing tests for Catroid.
- A custom test runner was used for Catroid to avoid issues during test result collection and merging faced on Jenkins with the default test runner. This blocked progress to move to more modern test runners.
- The jobs had very long execution times, ranging to many hours, occasionally even more than a day.

- Jobs were influencing each other.

Most of these issues impacted the usefulness of Catrobat Jenkins. Especially tests that had worked before did not work anymore. The problem of broken and flaky tests started to rapidly increase in 2015. Analysis was difficult but it was at least related to updates for the Android SDK, which also contains the Android Emulator, in combination with the custom test runner and to issues with the Android Emulator Jenkins plugin[1]. The success of tests also depended on the Android API in use. Tests would fail both locally and on Jenkins with a given API while the same set of tests would work with another.

The issues lead to long debugging sessions where different solution approaches were tried. Unfortunately, continuous integration could not be practised anymore. Without continuous integration regressions were not found automatically and thus the quality of both tests and code decreased.

The outdated version of Jenkins and the plugins also had security implications: Having up-to-date installations of Jenkins is essential for security, given the regular disclosures of security issues and their fixes (Jenkins Team, 2018). Yet updating these plugins was made harder by interdependencies

4.1 Policies and Transparency

Most of the issues the Catrobat Jenkins team faced were of a technical nature. The issues themselves and progress on their solution was discussed during meetings. The outcomes were tracked in the meeting notes. There was no overview of all the problems faced and the approaches already taken, which made it harder for newcomers to understand the whole scope of issues. At the same time there was no overview of the jobs on Jenkins, the installed plugins, and the configuration. The measures taken to improve transparency and visibility are described below.

Many of the encountered issues were not new, they were known. Yet the issues were not documented at a single location. Instead the issues were part of meeting notes. As a result, starting with the end of 2016 the Catrobat Jenkins team documented the issues on an own challenges page in the wiki software Confluence, see Figure 4.1 for an extract. Each of these tasks has

[1]For example: `https://issues.jenkins-ci.org/browse/JENKINS-27456`

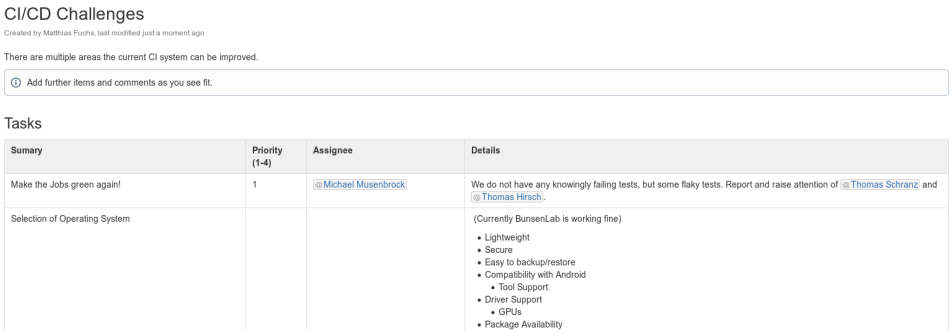


Figure 4.1: An extract of the the Confluence page to collect areas of improvement for Catrobat Jenkins. [Screenshot taken by the author of this thesis.]

a summary, a priority, zero or more assignees, and a detailed description. The idea of the challenges page was to support planning by providing an overview. Additions to the challenges were welcome and the overhead to add them was low. This helped communicating the vision of the project and gaining support of the team to realise that vision (Kotter and Cohen, 2012, page 83). The issue tracking software Jira[2] was specifically not used to track the challenges. Since the focus of the challenges page was planning, giving an overview and communication, Confluence was preferred over the issue tracking software Jira. The work on these challenges was then tracked on Jira.

The challenges page already improved transparency. Most of them were realised, like code quality measurements, automated configuration of slaves, and nightly build jobs.

For keeping an overview of a project is is important to know what the team members are working on. Seventy issues were created before 2017 on Catrobat Jira for Jenkins, the first in 2014. In the following two years alone nearly 200 further issues were added, of which 155 were resolved in the same time span. Participation on Jira increased. Consequently, what team members were working on became more transparent.

Another area with little documentation was research done by the team. With technology constantly advancing research is needed to figure out which technologies exist and whether they are feasible in their current state for Catrobat Jenkins. Without documentation of research findings the

[2]Website of Catrobat Jira: `https://jira.catrob.at`

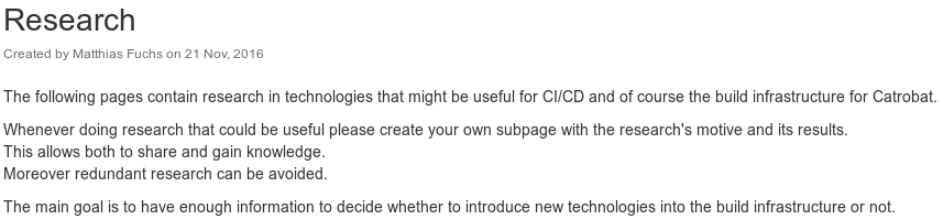


Figure 4.2: The parent page of the research space highlights the benefits of documenting research related to Catrobat Jenkins. [Screenshot taken by the author of this thesis.]

knowledge is not persisted and might be lost easily. Especially in a university project like Catrobat with a high turnover. The reasons why a technology was not used might be forgotten, potentially leading to confusion and extra work.

As a result, the author created a specific research space on Confluence to track research findings. Figure 4.2 shows the text of the homepage of the research space. It highlights that the focus is to share and gain knowledge and to avoid redundant research.

Consider the research on Firebase Test Lab for Android (Google Developers, 2018e) in November 2016 as an example. Firebase Test Lab provides the ability to run tests on physical or virtual devices in a Google data centre. The large amount of physical devices available[3] makes this an interesting option to test applications in a realistic setting. There were different pricing plans, including a free option for hobbyists[4]. The free option had a restricted daily quota of devices and practically also time the devices could be used. The result of that research was that Firebase Test Lab was not feasible for Catrobat. The free option was too restricted, especially with Catrobat test execution times of multiple hours at that time. The other pricing options were too expensive. In the future the conditions for Firebase Test Lab might change making it feasible for Catrobat. To aid such future research all findings were documented on Confluence. Figure 4.3 shows an excerpt of the Firebase Test Lab research. At the top of the page emphasis is given on when the research was performed and that the research results might be different now. Thus the research of a technology is not seen as one-time

[3]Physical devices for Firebase Test Lab: `https://firebase.google.com/docs/test-lab/android/available-testing-devices`

[4]Firebase pricing: `https://firebase.google.com/pricing`

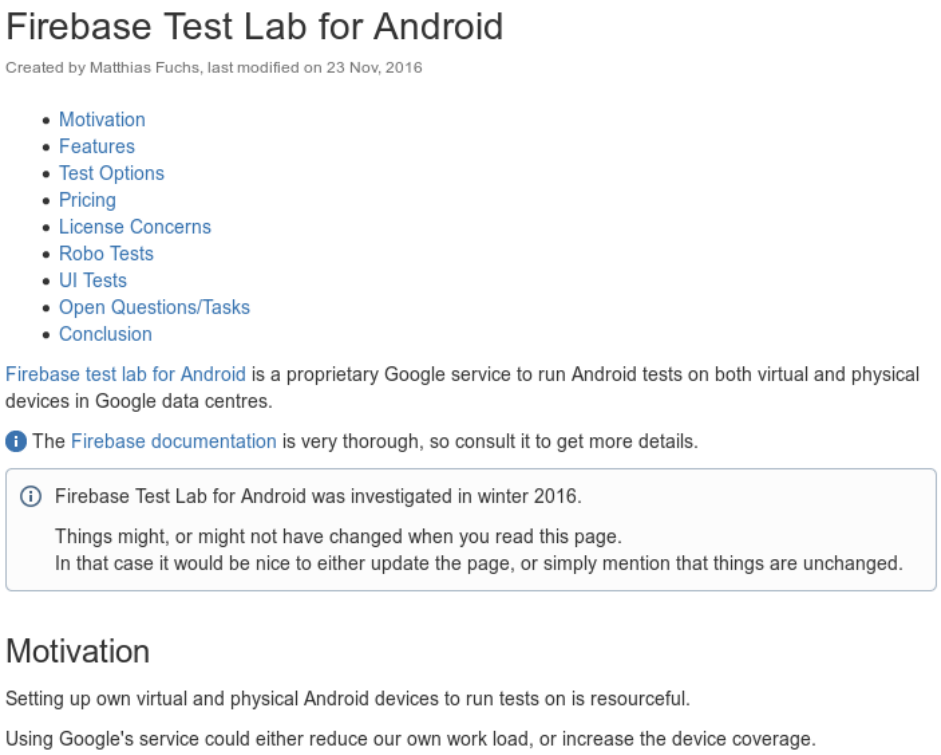


Figure 4.3: An excerpt of the Confluence page that tracked the research on Firebase Test Lab for Android. All findings of the research, including the outcome and its motivation are documented to ease future research, to share knowledge, and to avoid redundant work. [Screenshot taken by the author of this thesis.]

event, taking into consideration that technologies evolve with time.

As previously mentioned the team faced multiple issues at the end of 2016. With so many issues to combat the author decided it would be best to sidestep them all together and work on a new Jenkins system from scratch. That system would be well documented as a whole from the beginning. The system was installed on a test server and was supposed to replace the operational Jenkins when completely configured. The transition to the test server never happened though. As it turned out it was easier to clean up the operational server instead, keeping it available.

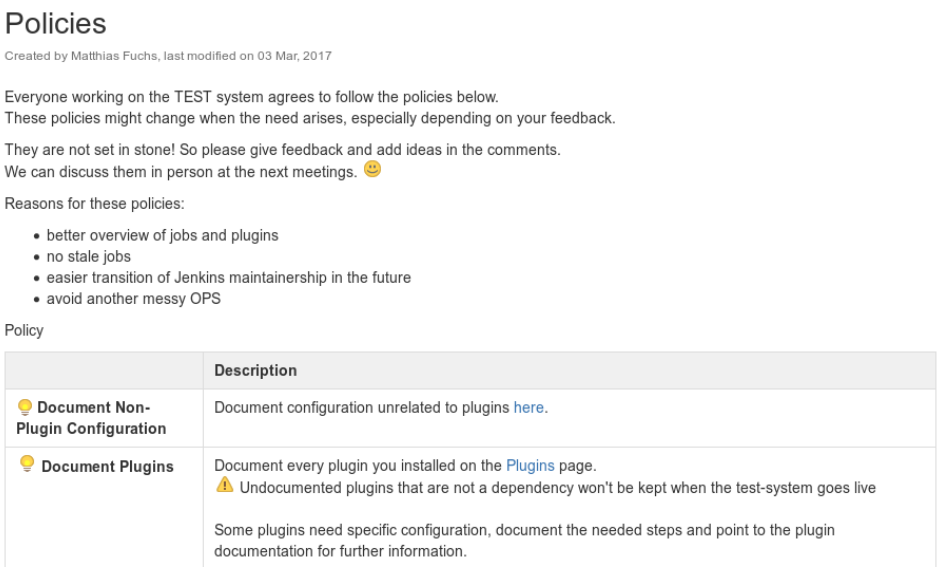
To improve the documentation and to avoid jobs with unknown origin or obsolete jobs one of the first steps for the test server was to introduce

policies that every Catrobat Jenkins team member had to follow. Each policy had a short name and a detailed description. The policies should be self-explanatory, therefore very easy to understand and follow. Policies should not incur too much complexity, as that would have adverse effects (Morieux, 2011). To increase acceptance reasons for the policies were given. The policies should lead to a system that is better understood by the team. Figure 4.4 shows an excerpt of the initial policies. Already with the initial version it was highlighted that the policies are not set in stone, that they are open for discussion. It was understood that the policies will change with gained knowledge, changes in technology, changes to the team structure, and the state of Jenkins Catrobat. Such an approach is similar to continuous improvement of lean methods, like Kanban (Kim et al., 2016, page 6).

It was very important to have as few policies as necessary. The policies should never become an end in themselves. Otherwise, this might lead to frustration, show a lack of trust, and reduce productivity eventually (Biro, 2018). If there are too many policies some of them might be ignored, or even worse there could be malice compliance. Such malice compliance refers to working by the book, which can even stop any progress while showing the abundance of useless policies (DeMarco and Lister, 2013, page 176). As a result, it is very important to keep the number of policies to the absolute minimum.

Initially there were eight policies:

- *Document Non-Plugin Configuration*: This refers to document any configuration needed that is not related to plugins. A link to the Confluence page to track the configuration is provided in the policy description.
- *Document Plugins*: In 2016 a vast amount of plugins was installed. Many of them unused or used by jobs that did not run for a long time. Why a plugin was installed initially or what steps were necessary to configure it was not documented. With this policy every newly installed plugin needs to be documented. The documentation includes a link to the plugin, its identifier, who installed the plugin and why, and what configuration steps are necessary to use the plugin. Plugins that are not documented on this page and are not required as dependencies can be removed without notice, to ensure that Jenkins remains clean. Later not only installed plugins were documented but also plugins that had been uninstalled. That way reasons of why a plugin is not



Policies

Created by Matthias Fuchs, last modified on 03 Mar, 2017

Everyone working on the TEST system agrees to follow the policies below.
These policies might change when the need arises, especially depending on your feedback.

They are not set in stone! So please give feedback and add ideas in the comments.
We can discuss them in person at the next meetings.

Reasons for these policies:

- better overview of jobs and plugins
- no stale jobs
- easier transition of Jenkins maintainership in the future
- avoid another messy OPS

Policy

	Description
Document Non-Plugin Configuration	Document configuration unrelated to plugins here.
Document Plugins	Document every plugin you installed on the Plugins page. Undocumented plugins that are not a dependency won't be kept when the test-system goes live Some plugins need specific configuration, document the needed steps and point to the plugin documentation for further information.

Figure 4.4: An excerpt of the initial policies for the Catrobat Jenkins team. Emphasis was given that policies are not in a steady state. Instead they would be adapted depending on feedback and needs. [Screenshot taken by the author of this thesis.]

used anymore would be kept.

- *Use Maintained Plugins:* Jenkins has more than 1000 plugins[5]. Many of these plugins are unmaintained or have few users. Hardly used plugins or unmaintained plugins can lead to security issues or they might become incompatible with Jenkins in the future. Therefore, this policy was added to only rely on maintained and somewhat popular plugins.
- *Own Views:* In 2016 and 2017 the jobs were created on the top-level of Jenkins and then grouped via views into related units, like Catroid, Paintroid, LeeroyCatroid, and LeeroyMultiJobs. To track who is responsible for a job each Jenkins team member would create a view of the form `FirstnameLastname`, for example, `MatthiasFuchs`, that listed all the jobs they were responsible for. With the retirement of a member it would be clear when jobs would need a new maintainer or could be removed.
- *Life-Time:* In 2017 there were more than 80 jobs. Most of them had not

[5]Jenkins plugins: `https://plugins.jenkins.io`

been run on that installation in months. Some of the jobs had been created years ago for testing purposes but were still kept around. This lead to the idea of life-times. Each job on the test system should have an associated life-time. The life-time would specify when a job could be removed. For example, with a life-time of two months a job could be removed if it did not run in the last two months. This measure should help reduce the amount of unmaintained jobs.
- *Job Description:* Each job should have a description that describes its purpose, its life-time, and possible shortcomings.
- *Consistent Styling:* The job description should have a consistent styling to emphasise whether a job is broken, undergoing rework, or working as intended.
- *Commit Finished Jobs:* When work on a job was finished and it worked as intended it should be moved to the corresponding view. For example, initially the job would be shown in the `MatthiasFuchs` view as mentioned above. When ready for the public it should be shown in the `Catroid` view. Before doing such a *commit* a discussion should happen with the affected stakeholders: The other Jenkins team members and representatives of the team who would use the job in the future.

Over the following months five policies were added, four policies were removed and many of the polices were refined. Overall, there were never more than nine policies active at the same time. The following list describes the added and removed policies:

- *Job DSL:* Initially all jobs were configured directly in the web interface of Jenkins, most of them were freestyle jobs. In mid of 2017 all jobs were configured via the Job DSL. That enabled to put the job configuration in the Catrobat Jenkins repository[6]. The jobs were still freestyle jobs, but Job DSL enabled to share code and make the jobs more similar. Now it would also be clear who added a job, due to the version control system used. At the same time the own views policy was removed, since now it would be clear who created a job. It was also encouraged to develop new jobs on a local Jenkins installation and performing a pull-request when finished. At that point also folders were used

[6]Job DSL definitions for Catrobat Jenkins: `https://git.io/fhZMV`

to group jobs. This lead to shorter names of the jobs as the project prefixes were not needed anymore.
- *Pipeline/Jenkinsfile:* In 2018 Michael Musenbrock converted the freestyle jobs to pipeline jobs specified via Jenkinsfiles placed next to the Catroid and Paintroid source code, see Section 4.3. To signify this change a policy was added to mandate that future jobs should be specified in a Jenkinsfile. The Job DSL would then be used to create the pipeline jobs.
- *Test Jobs:* Michael Musenbrock also added a policy that specified how to add prototype jobs on Jenkins. They should be placed in a lab directory and either should be prefixed with `FirstnameLastname` or be placed in a folder `FirstnameLastname`. Jobs that are not managed by Job DSL and not placed in the lab folder would be removed.
- *Working with Catroid/Paintroid Team:* Both the Catroid and Paintroid team started to use a new workflow in fall 2018 that mandates heavy involvement of product owners. This policy describes the necessary steps so that changes to the Jenkinsfile in the Catroid and Paintroid repositories can be merged.
- *Security Updates / Updating Jenkins:* Describes the necessary steps to update Jenkins, especially in case of security updates. There is also an automatic notification of Jenkins security advisories on the internal Slack channel of the Catrobat Jenkins team.
- *Job Description:* The job description policy was later merged with the Job DSL policy. The description of jobs should be done via the Job DSL. Jobs in the lab directory do not need a description.
- *Life-Time:* The life-time policy was removed later as well, since it became redundant with the job configuration being in a git repository.

To further emphasise the importance of communication, especially when mistakes happen, the author added a further introductory paragraph to the policies page as shown in Figure 4.5. The notion is that errors are part of a learning process that is enhanced by communication. This is similar to what DeMarco and Lister (2013, page 8) described as quota for errors in Peopleware. Punishing errors only leads to defensiveness and reduction in communication.

The test system never replaced the operation system as that would have been a huge change. Instead with the knowledge gained of working on the

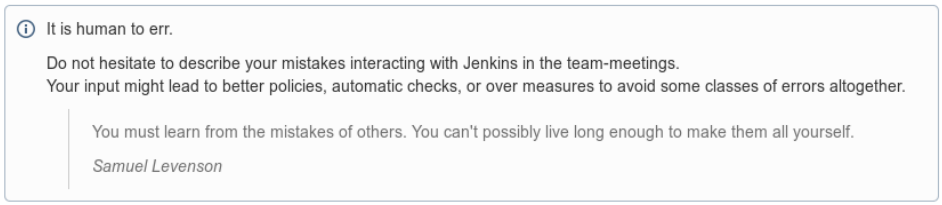


Figure 4.5: Text block added at the beginning of the policies page to emphasis that mistakes are part of a learning process. Communicating own mistakes can lead to overall improvements. [Screenshot taken by the author of this thesis.]

test system the operational system was adapted. So instead of transitioning directly to the test system smaller steps were performed directly on the operational system, mostly by Michael Musenbrock and Kirshan Kumar Luhana. Eventually this lead to a cleaned-up system. The policies were then applied to the operational system, changing the way of how to work with Jenkins. The end result was a well maintained Jenkins installation with a quite thorough documentation. This lead to more transparency and made it also easier for newcomers to the team to understand Jenkins and set up a local Jenkins instance themselves.

4.2 Configuration as Code

The previous section highlighted the importance of documentation to improve the situation. This section describes an approach that can avoid much of the documentation by treating configuration as code.

Despite all the improvements realised by documentation, documentation also has some disadvantages that have to be considered. A very common problem is that documentation becomes outdated. For example, by not documenting newly installed Jenkins plugins and their configuration, or not documenting configuration changes on the slaves.

Programmers have to face similar issues with comments in their code. In that case Martin (2008, pages 53–74) suggests to avoid most comments and to write self explanatory code instead. Unfortunately, for very long there was no way to translate the same approach to configuration. This is where configuration management, infrastructure as code, or more generally configuration as code came in.

Configuration as code refers to putting the configuration of something, for example, infrastructure, on a repository itself. There the configuration is not necessarily stored in its native format. Instead, domain-specific languages can be used to simplify writing configurations and to facilitate code reuse. When used correctly configuration as code makes it easier to recreate a system than trying to repair it (Kim et al., 2016, pages 118–119). Simply because setting up the system became easy.

Ansible[7], Chef[8], and Puppet[9] are very popular tools to manage the setup of systems, such as the Jenkins server infrastructure. With a domain specific language all common administrative tasks for a system can be handled, among others: The systems can be configured, packages can be installed, services such as Jenkins can be started, or users can be created on the system.

After research it was decided to manage the configuration of the test system[10], introduced in Section 4.1, with Ansible. The main advantage of Ansible in this regard was that it did not require a server to run and did not need specific software installed on the managed systems, other than Python. Instead Ansible could be triggered from any machine. Yet the resulting Ansible configuration was only used for the test system, but not the operational system. Much of the infrastructure of the Catrobat team was managed with Puppet, which was unfortunately not maintained well. With the creation of a designated infrastructure team for Catrobat in 2018 the process started to move the configuration of the Jenkins infrastructure to Puppet. The complete Jenkins infrastructure then used the Debian Linux distribution, reducing the complexity of the system. The complexity might be further reduced by relying on the Jenkins Evergreen project[11] once it leaves its beta phase. This project was started by the Jenkins team to reduce the burden of keeping Jenkins up to date and simplifying its maintenance.

While these tools work well to configure a system their support to configure Jenkins itself is lacking. Jenkins, as well as its plugins, are written in Java or languages supported by the Java Virtual Machine. With the script

[7]Ansible website: `https://www.ansible.com`
[8]Chef website: `https://www.chef.io`
[9]Puppet website: `https://puppet.com`
[10]Ansible configuration of the test system: `https://git.io/fhXa9`
[11]`https://jenkins.io/projects/evergreen`

console[12] Groovy scripts can be executed that directly interact with Jenkins. For a proof of concept the author wrote Groovy scripts that would configure some parts of Jenkins, see also Listing 4.1. In this case the usage statistics would be turned off, but only if they are turned on right now. So the configuration is only modified when changes are necessary. Writing such configuration requires detailed knowledge of Jenkins and the involved plugins. There is a risk that this code would be broken by future changes to the plugins and Jenkins. As a result, this approach was discontinued early on.

```
config('Not Sending Usage Statistics',
       { it.isUsageStatisticsCollected() },
       { it.setNoUsageStatistics(true) })
```

Listing 4.1: Excerpt of the code needed to configure parts of Jenkins with the Groovy programming language. [Source code taken from a commit by author of this thesis: `https://git.io/fhXwC`]

There was a common need for configuration as code and by the end of 2017 an enhancement proposal for Jenkins JEP201 was made to remedy this situation (Wilkosz, 2017). Eventually this lead to the development of the Jenkins Configuration as Code plugin[13] that saw a release of version 1.0 in September 2018. Using this plugin Jenkins can be configured with files in the YAML[14] format. Currently there are plans to use the plugin to configure Jenkins for Catrobat.

Closer communication with the Jenkins community would have avoided the research that lead to Listing 4.1, instead waiting for JEP201 to become usable. Similarly, it might have made sense to try and improve the situation of Puppet in use for Catrobat instead of experimenting with Ansible. Nonetheless, working on configuration as code provided many insights and underlines the future of how servers will be configured.

With the help of configuration as code the amount of documentation can be reduced. At the same time the Jenkins infrastructure will be easier to set up, both locally and on the servers. Eventually there will be one

[12] `https://wiki.jenkins.io/display/JENKINS/Jenkins+Script+Console`

[13] Website for the Configuration as Code Jenkins plugin: `https://plugins.jenkins.io/configuration-as-code`

[14] YAML format: `https://yaml.org`

single source of truth for the configuration of the Jenkins infrastructure. This leads to a reduction of manual steps needed and thus to well defined environments, avoiding snowflake servers (Fowler, 2012a). There is still much work needed before the full potential can be applied to Catrobat Jenkins. Especially considering the vast improvements the configuration as code plugin saw in the last months. Overall, configuration as code will make the Jenkins system more maintainable.

4.3 Configuring Build Jobs

As with any other continuous integration system the main focus of Jenkins are the jobs that it runs. Therefore, configuring these jobs is an integral part of maintaining Jenkins. For the users of Jenkins running the jobs and checking their results is the most common task. A clear structure of the jobs and a similarity between the jobs is not only important for users of Jenkins but also for its administrators. This section focuses on the configuration of the build jobs. Advantages and disadvantages of different approaches are discussed using issues faced by the Catrobat Jenkins team as example.

From the perspective of users jobs shall provide all the functionality they need. This includes building the source, running tests, running static analysis tools, providing artefacts such as executables that were built, collecting code coverage, collecting other statistics like the size of the resulting executable, checking dependencies for security issues, and of course all with decent performance. Usually users are not aware of the capabilities of the tools available, including continuous integration systems. They might use them, yet they do not want to create jobs themselves. Instead their focus is naturally the actual software they work on. As a result, the maintainers of the continuous integration system, also in the case of Catrobat Jenkins, create the build jobs and add further steps to them when needed. Therefore, a key to useful jobs is communication and close interaction between the people providing the jobs and its users. This includes reacting to requests by users, but also to suggesting tools users might not be aware of.

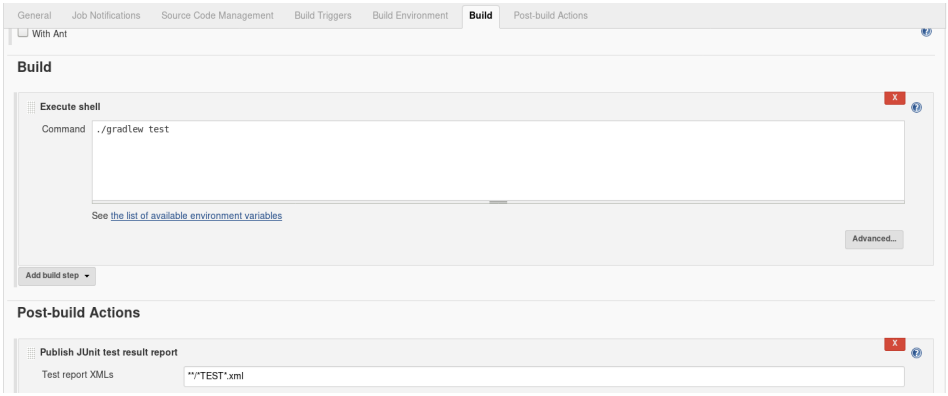


Figure 4.6: Small parts of the configuration of a freestyle Jenkins job. Further build steps can be added by selecting them from a drop-down menu. This adds additional forms that have to be filled in. The forms of even the simplest freestyle job need multiple screens for the whole configuration. [Screenshot taken by the author of this thesis.]

4.3.1 Configuration with the Jenkins User Interface

Historically the way to create jobs in Jenkins was via its web browser user interface. A job type, such as a freestyle job, has to be selected, a name provided, and the steps of the job have to be specified in HTML[15] forms (Laster, 2018, page 1) such as in Figure 4.6. Everything would be done directly in the web browser by filling in a huge amount of forms. The configuration of the resulting job is then stored on Jenkins master as XML[16] files. This lead to some practical issues the Catrobat Jenkins team faced:

- Who is the owner of a job?
- What is the purpose of a job?
- How to revert configuration changes that lead to faulty jobs?
- How to backup jobs?
- How to avoid snowflake jobs?
- How to provide the needed functionality without incurring too much complexity in the resulting jobs?

[15]World wide web consortium HTML website: `https://www.w3.org/html`
[16]XML standard: `https://www.w3.org/TR/2006/REC-xml11-20060816`

Attributing the owner and purpose of a job can be done via a policy, see also Section 4.1. Of course policies can be ignored or forgotten, therefore automation is desirable.

The JobConfigHistory[17] plugin helps to attribute who changed what and when of the configuration of a job. This plugin compares the resulting XML file of the job after a configuration change with a previous state. Faulty changes can be reverted directly in the user interface of the plugin.

Yet looking directly at the differences of the XML files forces implementation details upon the user. Suddenly they are confronted with how Jenkins stores the jobs and what each step looks like in XML. Especially when many configuration changes were made the differences between the XML files become overwhelming.

Unfortunately, any change to the XML file is treated as configuration change, which leads to many entries in the job configuration history. For example, some plugins write their version to the configuration file. Updates of the plugin lead to updated job configuration XML files, which leads to new entries in the job configuration history view. Even worse is the Job DSL plugin, see below, which changes the XML files regularly.

For the Catroid job there were more than 250 entries in the job configuration history between its creation in February 2018 and January 2019. Only 29 of these changes were performed by actual users, all the other changes were performed automatically by Jenkins.

The usefulness of the JobConfigHistory therefore depends on what plugins are installed, how often the plugins are updated, and on the amount of changes to the structure of the XML file.

Whenever manual steps are needed to configure something there is a high risk that similar systems are configured differently. Fowler describes snowflake servers as servers with a different configuration each. Their configuration is not reproducible since it is not clear what was configured manually in the first place. The same applies not only to servers, but to configuration in general. For instance, many jobs on Catrobat Jenkins could have been considered snowflake jobs.

In 2017 there were more than 80 jobs on Jenkins. All of them were placed on the top-level of Jenkins with no hierarchical structure, other than views.

[17] Plugin for Jenkins to track changes to the job configuration: `https://plugins.jenkins.io/jobConfigHistory`

All of the jobs were configured manually via the Jenkins user interface. This lead to snowflake jobs. Some of the jobs automatically cleaned up the workspace before builds, leading to reproducibility. Other jobs ensured that each log line in the build log was timestamped. Some jobs only kept a history of the 30 last builds, while others did not have a limit. Another common distinction was that jobs had different security settings.

All the solutions described above work somewhat. They share a common disadvantage though: All of them rely directly on Jenkins. That means everything has to be done in the Jenkins user interface. There is no single place to handle all these issues at once. Different plugins have to be installed and configured, different policies have to be enforced. These solutions do not cover a catastrophic failure of the Jenkins infrastructure. For example, when the hard drives fail the configuration as well as the backups created by Jenkins[18] might be lost. This is where configuration as code can be applied.

4.3.2 Job DSL

The Job DSL plugin provides a domain-specific language using Groovy[19] to specify jobs in a configuration as code manner. Basically any kind of Jenkins job can be represented, including freestyle jobs.

The basic workflow is to create Groovy files in a version control system that contain the specification of the jobs and views. These Groovy files are then executed on Jenkins creating these jobs and views. The execution is done by a specific job which is in general called seed job (Spilker, 2016). The seed job itself has to be created manually first, but can then also be created via the Job DSL, which allows bootstrapping.

To support the development of configuration as code an interactive reference is included in the user interface. Furthermore, a public reference[20] is available. The reference includes documentation for Job DSL methods, depending on the installed plugins.

Since Groovy is a regular programming language best practices of software development can be employed. This includes avoiding repetition by

[18] Plugin for Jenkins to backup the configuration: `https://plugins.jenkins.io/thinBackup`

[19] Groovy programming language: `http://groovy-lang.org`

[20] Job DSL reference: `https://jenkinsci.github.io/job-dsl-plugin`

using abstractions like functions and classes, which is often referred to as *don't repeat yourself* (DRY) principle (Hunt and Thomas, 2000, pages 26–33). Common code can be placed into own Groovy files that can then be included automatically by the Job DSL plugin.

Listing 4.2 gives an example for the Job DSL syntax. A freestyle job is created that retrieves the source code from a git repository, builds the project, and runs the tests. The git repository is checked every 15 minutes for changes.

```
job('DSL-Tutorial-1-Test') {
    scm {
        git('git://github.com/quidryan/aws-sdk-test.git')
    }
    triggers {
        scm('H/15 * * * *')
    }
    steps {
        maven('-e clean test')
    }
}
```

Listing 4.2: A Job DSL example for a freestyle job that checks a git repository every 15 minutes for changes, builds the source, and executes tests (Spilker, 2016).

The Job DSL resolves all issues identified before. The specification of all build jobs, except the initial seed job, can reside externally of Jenkins in an own source repository. With a version control system in place it is easy to attribute who changed what for a given job. Of course then it is also easy to undo faulty changes. Policies like good commit messages can also be enforced manually when reviewing pull requests. Creating own Groovy abstractions makes it possible to create jobs with similar settings, avoiding snowflake jobs altogether.

When managed properly the resulting job specifications can be clearer than the configuration created directly in the Jenkins user interface. The latter relies on many check boxes, text fields, advanced settings hidden behind buttons, and further forms which do not fit on a single screen.

4.3.3 Jenkins Pipeline

With Jenkins 2 pipelines have been added as preferred way of creating jobs (Laster, 2018, pages 1–22). Jenkins pipelines are a domain-specific Groovy language that makes the creation of complex jobs possible. The definition of a job can be put into a so called Jenkinsfile directly in the source code repository, next to the source that it will build. New job types have been added that read these Jenkinsfiles and execute their build steps. To avoid repetition common pipeline code can be placed in an own library that is then shared between jobs (Laster, 2018, page 185) . Jenkins pipeline adds a flexibility to Jenkins that was previously unachievable. This is also the main contrast to Job DSL. The Job DSL is not a job type itself and therefore does not add any flexibility on its own. It is constrained by the types of jobs it creates.

In Jenkins 2 there are three new job types directly related to pipeline:

- Pipeline
- Multibranch Pipeline
- GitHub Organisation

The `Pipeline` job works either on a Jenkinsfile in a specified branch of a repository or has the pipeline domain-specific language embedded. In contrast, the `Multibranch Pipeline` job always needs a repository. The job then automatically detects all branches with a Jenkinsfile and creates jobs for them. As such the multibranch job acts as if it was a folder, only that all elements of that "folder" are added automatically and cannot be modified manually. With the correct settings even pull requests are detected and cleaned-up after merging. Webhooks can be created automatically too. That way GitHub informs Jenkins whenever there was a change to the repository or any of the pull requests. The *GitHub Organisation* job is similar to the multibranch pipeline job. The difference is that it works on all repositories of a GitHub organisation and not just on a single repository. In terms of the folder analogy there would be a top-level folder for the organisation with a sub folder for each repository, which in turn consists of the jobs for all branches and pull requests.

What these jobs enable is to have one repository of truth (Kim et al., 2016, pages 115–119). This repository would then not only include the source code but also a Jenkinsfile that specifies all the steps necessary for continuous

integration and continuous deployment. Changes to the Jenkinsfile in one branch do not affect other branches. Consequently, older branches will continue to work in the future. No additional versioning is required. Of course there are some limitations, for example, when jobs share external resources like directories. Nevertheless in general jobs should not share any resources, as is explored in Section 4.5. Another huge advantage of these job types is the automatic detection of branches and pull-requests. There only needs to be one job that can handle most if not all use cases.

There are two different syntaxes for Jenkins pipeline (Laster, 2018, pages 4–5):

- Scripted Pipeline
- Declarative Pipeline

Scripted pipelines enforce little structure and are executed top-down, like any other Groovy script. The stages of a build are executed on specific nodes. Regular Groovy can be used in the scripts, including its stages. Therefore, flow control with if-conditions and loops is possible. Error handling is done by acting on exceptions, for example, catching an exception to then inform people via mail of the failed build.

Each build stage consists of the build steps to execute, such as `sh` to execute shell commands and `junit` to collect and publish the JUnit[21] test results. Each Jenkins plugin can provide pipeline steps, like the JUnit plugin[22] with the `junit` step.

The flexibility of scripted pipeline makes it useful for very complex jobs but incurs additional work for regular tasks. For example, typical post-build actions like collecting JUnit results, informing people of the build result, and publishing coverage results have to be handled manually. They have to be in the `finally`-block of a `try-catch-finally` to be always executed, while steps that should only be executed on success have to be placed inside of the `try`. Listing 4.3 illustrates such a pipeline that executes tests within a given environment on a node (Laster, 2018, pages 32–34) with the label or name `java`. The test results are always collected, even if the Gradle step fails.

[21] JUnit test framework: `https://junit.org`
[22] Plugin for Jenkins to collect JUnit test results: `https://plugins.jenkins.io/junit`

```
node('java') {
    stage('Test') {
        try {
            withEnv(["GRADLE_USER_HOME=/.gradle/
               $EXECUTOR_NUMBER", 'ANDROID_SDK_ROOT=/usr/
               local/android-sdk']) {
                sh './gradlew test'
            }
        } finally {
            junit '**/*TEST*.xml'
        }
    }
}
```

Listing 4.3: Scripted pipeline that runs on a node named `java` or on any node that has a `java` label. There is one stage that executes tests whose results are always collected by the `junit` step. Gradle is executed within a specified environment. [Source code written by the author of this thesis.]

With pipeline version 2.5 the declarative syntax was added (Bayer et al., 2018). In contrast to scripted pipeline some of the flexibility is sacrificed by enforcing a clear structure that aims to represent typical Jenkins workflows.

To illustrate the declarative syntax the scripted pipeline example of Listing 4.3 was rewritten declaratively in Listing 4.4. In this case the resulting code is longer, though the intent is arguable clearer. Notably the same build steps can be used in both the scripted and the declarative pipeline.

All the build steps are placed in a hierarchical structure. A `stages`-block contains at least one `stage`-block that in turn contains build steps grouped in a `steps`-block. Further structuring is possible by nesting multiple sequential stages inside of a stage or by employing parallelism to some degree.

The `agent` directive specifies where to execute the build, which could include a Docker container as is described in Section 4.5. Above all each `stages`- and `stage`-block can have its own agent to execute on. This makes it very easy to, for instance, execute stages on different operating systems.

There is a specific `environment`-block where the environment variables are set for the whole build. In the same way there can be an `environment`-block for each stage and stages, furthermore `withEnv` can still be used.

With the `post` block it is clear which steps are part of the post processing, such as collecting test results or code coverage results. In this case the `junit` step is always executed, even on failure, which is made explicit. A multitude

of different conditions is supported, for instance, always, failure, success, changed, and fixed.

```
pipeline {
    agent {
        label 'java'
    }

    environment {
        GRADLE_USER_HOME = "/.gradle/$EXECUTOR_NUMBER"
        ANDROID_SDK_ROOT = '/usr/local/android-sdk'
    }

    stages {
        stage('Test') {
            steps {
                sh './gradlew test'
            }

            post {
                always {
                    junit '**/*TEST*.xml'
                }
            }
        }
    }
}
```

Listing 4.4: Declarative pipeline that runs on an agent with the label `java`. There is one stage that executes tests whose results are always collected by the `junit` step in the `post`-block. The whole build is executed within a specified environment. [Source code written by the author of this thesis.]

Development of both scripted and declarative pipelines is aided by references accessible in the user interface, including generators for build steps. Another form of creating basic declarative pipelines is the Blue Ocean Editor (Laster, 2018, pages 344–378) as part of the recently introduced Blue Ocean interface (Laster, 2018, pages 317–379).

The Blue Ocean interface is a new graphical interface with the goal to be cleaner than the traditional Jenkins interface. Although it does not yet support all features like displaying code coverage, displaying the results of static analysis, or creating complex declarative pipelines. Yet it provides a

Figure 4.7: The structure enforced by freestyle jobs. [Screenshot taken by the author of this thesis.]

graphical view of the pipeline in form of a graph, which makes its structure clearer.

The newer declarative pipeline is somewhat endorsed in the official Jenkins documentation by describing it first and often hiding scripted pipeline examples behind an additional mouse click (Gaskell et al., 2018). Clearly an advantage of the declarative pipeline is that the Blue Ocean Editor is supported and that its structure helps in common tasks and is also familiar with freestyle jobs (Laster, 2018, pages 25–26, pages 218–219).

Figure 4.7 shows the structure of a freestyle job. There are different sections like build triggers, build environment, build, and post-build actions that can be mapped directly to sections in the declarative pipeline. Such direct mapping is not possible for scripted pipeline, which is closer to imperative programming.

In contrast, the scripted pipeline is often mentioned in combination with advanced and complex build jobs. It provides more flexibility than the declarative pipelines due to its simpler structure (Bayer et al., 2018).

The Jenkins pipeline is a good way to avoid the limitations of freestyle jobs altogether. While the Job DSL can mitigate some of the limitations of freestyle jobs, freestyle jobs in themselves remain inflexible. Consequently, it is beneficial to use Jenkins pipeline for all kind of jobs and only to rely on the Job DSL to create these pipeline jobs. Probably a sensible approach is to start with declarative pipeline first and to only use scripted pipeline when not possible otherwise.

4.3.4 Build Jobs on Catrobat Jenkins

At the end of 2016 there were roughly 50 freestyle jobs with either Catroid or Paintroid in their name grouped into multiple views. The views had names like Catroid, Catroid-multi-job, LeeroyCatroid, Paintroid, and Paintroid_NEWView. There was no further indication about the state of these jobs. All the jobs were configured via the Jenkins user interface. None of the jobs were stored in a repository. Therefore, improving the situation

was an early goal.

Transforming existing jobs to pipeline jobs was a natural contender with all its advantages mentioned above. Yet in the beginning of 2017 Jenkins pipeline was still relatively new. For instance, Blue Ocean did not exist yet. Furthermore, the Android Emulator Jenkins plugin that was used was not supported yet. This plugin managed everything related to the emulator, including the installation of its dependencies.

For this reason it was decided to stick with freestyle jobs and to configure them via the Job DSL as a stopgap measure. The hope was that the Android Emulator plugin would be supported in the near future, as was planned by its maintainer, making a seamless transition to pipeline possible. Section 4.6 discusses the issues faced with the Android Emulator, explains in more details the reasoning to select Job DSL at first, and describes further measures taken to work with the emulator.

To remedy the situation of too many jobs with unknown purpose only jobs with a known purpose that were actually used were deemed to be created by the Job DSL. Nonetheless with freestyle jobs many specialised jobs were used. This was partially done since it was easier to have small freestyle jobs with as little conditional logic as possible. Otherwise, the jobs would become very complex and confusing. In October 2017 Job DSL managed 10 freestyle jobs for Catroid in an own folder on Jenkins[23]:

- `SingleClassEmulatorTest` executed a test class of the parameterised git branch and repository on the Android Emulator.
- `SinglePackageEmulatorTest` same as `SingleClassEmulatorTest` only that here a test package was executed in contrast to a test class.
- `PullRequest` executed the instrumented unit tests for Catroid pull requests.
- `PullRequest-Standalone` ensured that standalone APKs could still be created for pull requests.
- `PullRequest-UniqueApk` built for every pull request a Catroid APK with a unique application name. This enabled parallel installation of multiple versions of the Catroid application.
- `PullRequest-Espresso` ran Espresso user interface tests for every pull request.

[23]Catroid build jobs in October 2017: `https://git.io/fhGbo`

- `Nightly` ran tests and static analysis for Catroid every day at midnight and made the generated APK accessible.
- `Continuous` ran on any change to the develop branch of the Catroid repository. Similar to the nightly, but did not make the resulting APKs accessible.
- `Standalone` was triggered automatically by the Pocket Code website[24] to build and publish APKs for user generated applications.
- `Standalone-Nightly` built the same user application of the Pocket Code website every midnight to ensure that standalone applications could still be built.

At the same time five freestyle jobs were managed for Paintroid, with similar naming and meaning[25]:

- `SingleClassEmulatorTest`
- `SinglePackageEmulatorTest`
- `PullRequest`
- `Nightly`
- `Continuous`

This means that during 2017 it was possible to reduce the number of jobs in views for Catroid and Paintroid from roughly 50 jobs to 15. All managed with the Job DSL and all with a documented purpose. Initially these jobs only existed on a test server, to avoid negative effects on the operational system.

Listing 4.5 shows the code necessary to create the `Continuous` job for Paintroid. That only 12 lines of code are necessary is made possible by abstractions implemented on top of the Job DSL. Without these abstractions, in plain Job DSL syntax, the same job would need more than 100 lines of code. Instead the resulting job definition is very readable and compact. The Job DSL was also used to create the seed job in a bootstrapping fashion.

[24]Pocket Code website: `https://share.catrob.at`
[25]Paintroid build jobs in October 2017: `https://git.io/fhGbK`

```
paintroid.job("Continuous") {
    htmlDescription(['Job runs continuously on changes.'])

    jenkinsUsersPermissions(Permission.JobRead)

    git()
    continuous()
    androidEmulator()
    gradle('connectedDebugAndroidTest',
           '-Pjenkins -Pandroid.
              testInstrumentationRunnerArguments.class=org.
              catrobat.paintroid.test.espresso')
    junit()
}
```

Listing 4.5: Job DSL definition of the continuous job for Paintroid. [Source code written by the author of this thesis.]

The transition to pipeline jobs triggered by Michael Musenbrock further reduced the number of jobs needed. Eventually for Paintroid a single Multibranch Pipeline job remained[26] handling pull requests, nightlies, and building on changes. The need for further jobs to build specific branches was reduced, since the Multibranch Pipeline job detects branches and all pull requests automatically. At the same time the Paintroid team improved its tests, which is why `SingleClassEmulatorTest` and `SinglePackageEmulatorTest` were not needed anymore.

For Catroid four pipeline jobs remained[27]:

- `Catroid` is a Multibranch job handling pull requests, branches, nightlies, continuous integration, and optionally building APKs for all flavours that can be installed in parallel.
- `Catroid-SensorBoxTests` a new Multibranch job to run a subset of tests on a real Android device connected to a sensor box. This sensor box (Lesser, 2018) detects hardware effects like flash activation and vibration of the device.

[26] Paintroid build jobs in December 2018: `https://git.io/fhGbP`
[27] Catroid build jobs in January 2019: `https://git.io/fhDxE`

- `Catroid-ManualEmulatorTest` takes user input in the form of a git branch and repository, Android Emulator parameters, as well as the test class or test package to execute.
- `Build-Standalone` job with the same behaviour as the `Standalone` job previously.

The flexibility of pipeline lead to three jobs where there had been 10 before with the addition of the new job `Catroid-SensorBoxTests`, which had no counterpart before the transition. The jobs also provided more functionality.

An early goal when introducing pipeline jobs was to keep the Catroid and Paintroid jobs similar. Initially this resulted in similar, but not identical, build stages. The main difference was in the environment section where the differences were mapped to environment variables with project specific values, see Listing 4.6 for Catroid. These environment variables were then referred in the build steps. With increasing work the build steps of both Catroid and Paintroid diverged, especially since Catroid supports different flavours and has its tests structured differently. The environment variables were only used in the Jenkinsfile, while they were also accessible from within the build. They polluted the environment of the build and also made the build steps harder to understand. Many of the steps were also duplicated, increasing the size of the Jenkinsfile. As there are multiple Jenkinsfiles for Catroid to realise the different jobs there was also duplication.

The Jenkinsfile was refactored by creating helper functions and using regular Groovy variables instead of environment variables. Basically applying best practices of programming. Eventually no environment variables were needed directly in the Jenkinsfile, instead some of them were moved to the Dockerfile, see Section 4.6.

Further work by the author of this thesis included nightlies for the develop branch, code coverage collection, support to trigger builds directly from comments on GitHub, optionally building all Catroid flavours, building Catroid packages that point to the test server instead of `share.catrob.at`, improved independent APKs, and general cleanup.

Some of these features were added to support the needs of the stakeholders, for example, the Catrobat Web Team can now test the compatibility of Catroid with changes they made on their test server. Instead of hoping that changes to their live system are compatible with Catroid they can try beforehand on their test server, thereby reducing the risk drastically.

```
//////// Build specific variables ////////
////////// May be edited by the developer on changing the
    build steps
// modulename
GRADLE_PROJECT_MODULE_NAME = "catroid"

// APK build output locations
APK_LOCATION_DEBUG = "${env.GRADLE_PROJECT_MODULE_NAME}/build
    /outputs/apk/catroid/debug/catroid-catroid-debug.apk"
APK_LOCATION_STANDALONE = "${env.GRADLE_PROJECT_MODULE_NAME}/
    build/outputs/apk/standalone/debug/catroid-standalone-
    debug.apk"

JACOCO_XML = "${env.GRADLE_PROJECT_MODULE_NAME}/build/reports
    /coverage/catroid/debug/report.xml"
JACOCO_UNIT_XML = "${env.GRADLE_PROJECT_MODULE_NAME}/build/
    reports/jacoco/jacocoTestCatroidDebugUnitTestReport/
    jacocoTestCatroidDebugUnitTestReport.xml"

// place the cobertura xml relative to the source, so that
    the source can be found
JAVA_SRC = "${env.GRADLE_PROJECT_MODULE_NAME}/src/main/java"
```

Listing 4.6: Parts of the `environment` section of the Catroid Jenkinsfile in August 2018. [Source code taken from `https://git.io/fhSpH`]

Independent APKs were a feature introduced with the Job DSL and further improved for the pipeline jobs. A common use case of the Catroid and Paintroid product owners was to install the APKs of a pull request to try out the changes. Android only allows to have one installation of an application. The solution was to assign each APK a unique application name, so that they could be installed independently of APKs provided by other builds. For the pipeline job the naming was improved by considering the flavours and using a name that easily identifies the relevant branch or pull request that an installed application originated from.

Performance of the build jobs was also improved, see Section 4.7. Furthermore, Kirshan Kumar Luhana (Luhana, Schindler and Slany, 2018) and Rainer Lankmayr introduced support to deploy Catroid and Paintroid directly to the Google Play Store, drastically reducing the manual steps needed. During all this work the complexity of the Jenkinsfile increased, yet the jobs

are still easy-to-use.

Configuration as code for build jobs proved a success. In hindsight it would have made more sense though to directly use Jenkins pipeline and not to hope that the Jenkins Android Emulator plugin would be supported in the future. Nonetheless, configuration as code made transparent what the configuration of a job is and who created that configuration. Placing the job configuration next to the source code of the builds means that even older branches can be build in the future, no matter if the most recent job configuration would be incompatible. Notable of the build job definitions is that they were actively maintained by adding more features, refactoring the code, and improving performance and stability. Excluding merge commits there were more than 40 git commits that modified any of the Jenkinsfile for Catroid and more than 35 for Paintroid, which is a testament to a well maintained and actively used system.

4.4 Flaky Tests

For both Catroid and Paintroid tests are executed on Jenkins. In 2016 most of these tests were executed on the Android Emulator. Unfortunately, many tests failed and one reason brought forward by the Catroid team was that they were flaky. This section describes the problem of flaky tests and discusses the steps taken by the Jenkins, the Catroid, and Paintroid teams to understand and improve the situation.

4.4.1 Impact of Flaky Tests

Tests are at the core of continuous integration: All tests have to succeed for a build to pass (Duvall, Matyas and Glover, 2007, page 42). The precondition is that tests are deterministic. That means without a change to the system under test the test will either always succeed or always fail. Tests that are non-deterministic are often called flaky tests (Luo et al., 2014). Such flaky tests are a common problem as Luo et al. (2014) found out: Industry leaders like Google are affected as are open source projects like Jenkins.

Flaky tests have a very negative impact (Fowler, 2011a). Tests are supposed to act like a safety guard that detects regressions. Thus when a build failed the developer whose source code changes were built is supposed to

investigate the failing tests. Developers lose trust in the tests when it turns out that their own source code changes were unrelated to the failing tests, that the tests were flaky. Eventually developers will ignore the results of alleged flaky tests. These flaky tests would then not act as safety guard anymore. Bugs could be introduced without anyone noticing. Even worse, developers might start to ignore all tests results. For them it might be a waste of time to check the test results to always find the same set of tests failing. This is why Fowler (2011a) calls flaky tests virulent. They degrade the value of any test suite.

4.4.2 Causes for Flaky Tests

There are multiple causes for flaky tests like the so called asynchronous wait (Luo et al., 2014). An asynchronous call is performed to then check the results. The test fails when the result of the asynchronous work is checked before it finished, which in general is referred to as race condition. Such race conditions can easily happen when testing user interfaces (Thorve, Sreshtha and Meng, 2018), as they are often run in their own thread.

In a typical flaky test first a user interface action is scheduled in the user thread. Second the test thread is manually paused for a given time, ideally sufficient time for the user interface thread to complete its work. Such a pause is called sleep. After the sleep the results are checked in the user interface.

Listing 4.7 shows code of the Paintroid project that had this issue. If the sleep is too short the following assertion will fail, since the asynchronous work is not finished by then. Therefore, the test depends on the performance of the machine it is executed on. Continuous integration systems are often under heavy load, thus a longer sleep might be needed. A longer sleep increases the overall test execution time which is detrimental for overall performance. In general tests should work independently of the performance of the machine they are executed on.

Another cause for flakiness can be lack of isolation (Fowler, 2011a). This refers to tests depending on each other to some degree, for example, when a test does not reset some global state, like resources, after its execution. With such a dependency one test might handle resources incorrectly which then affects another test. Suddenly the test execution order matters. This

behaviour was often witnessed in the Paintroid drawing application when a test did not correctly reset the colours and brushes used.

```
mSolo.drag(bottomrightCanvasPoint.x / 2,
    bottomrightCanvasPoint.x / 2, bottomrightCanvasPoint.y /
    2, bottomrightCanvasPoint.y / 2 + canvasCenterTollerance,
    1);
PointF canvasCenter = new PointF((bottomrightCanvasPoint.x +
    widthOverflow) / 2, newBitmapHeight / 2);

mSolo.sleep(SHORT_SLEEP);
assertTrue("Center not set", PaintroidApplication.
    drawingSurface.getPixel(canvasCenter) != Color.TRANSPARENT
    );
```

Listing 4.7: Segment of the `BitmapIntegrationTest.java` file of the Paintroid project in March 2017. The sleep used might be too short in some cases leading to flaky tests. [Source code taken from `https://git.io/fhZpR`]

An often overlooked cause for flakiness are external dependencies (Thorve, Sreshtha and Meng, 2018). Both Catroid and Paintroid used the Robotium[28] test framework. Unfortunately, Robotium tests tended to be slow and flaky (Genco, 2015). One cause was usage of sleep as mentioned above but also the framework itself. Another cause for flakiness is the Android Emulator. The Android Emulator is started once, then all tests are executed on the emulator instance. Consequently, the emulator can be regarded as global state that is shared between all tests. Since the emulator is complex software itself, bugs can lead to flaky tests. Unfortunately, the lack of isolation in case of the emulator is mandated by performance and practical considerations.

A typical source for flakiness are also tests that rely on the network (Thorve, Sreshtha and Meng, 2018). The Catroid team had some network related tests that tendend to be flaky, like the `testLoginWithNotExistingUser` in `ServerCallsTest` that tried to login on the test version of the Pocket Code Website[29] with an invalid account.

[28] Robotium: `https://github.com/RobotiumTech/robotium`
[29] Pocket Code website: `https://share.catrob.at`

4.4.3 Handling Flaky Tests

There are different ways of handling flaky tests. Some of the approaches try to fix the underlying issue, while other approaches can be considered a workaround.

One common workaround is to mark flaky tests as such. When a marked test fails it is rerun a couple of times up to a limit and treated as passed if any of the test runs succeeded (Luo et al., 2014). Such a test would still find bugs that always lead to a test failure. Bugs with flaky behaviour themselves would not be found by such a test. Another approach is to remove flaky tests, since they have questionable benefit. As Fowler (2011a) points out this can lead to test removal as habit when developers lose discipline and do not analyse the test results anymore. A further workaround is to move flaky tests into an own quarantine test suite (Fowler, 2011a), to separate between deterministic and non-deterministic tests. The danger exists that this quarantine suite grows and failures there are discarded as flaky and are not investigated at all.

Up until 2017 whole test packages with failures were rerun for Catroid. This increased the test execution time dramatically to five hours on average.

With actual flakiness it becomes likely that a test package with many flaky tests will never succeed. Rerunning the whole package would only work if there were very few flaky tests. Suppose every flaky test has a probability of failure p_{tf} and that the flakiness for each test is independent. With $p_{tf} = 0.2$ and $n_{ft} = 5$ flaky tests the probability of the test package to succeed would be $p_{ps} = (1 - p_{tf})^{n_{ft}} = (1 - 0.2)^5 = 33\%$. The probability that a package succeeds at least once in three runs would then be $p_s = 1 - (1 - p_{ps})^3$ and in the example 69.6%. So in this example the test package would show as failed for roughly 30% of the builds, leading to additional overhead. The problem of test execution times becomes more apparent when test packages contain tests that always fail, which unfortunately was the case.

In 2018 a quarantine test suite was added for Catroid that included very flaky tests. This test suite was only run during nightlies, since there were always failures. The quarantine test suite even lead to Android Emulator crashes more than half of the time.

Then there are the actual solutions. That means improving the tests to some degree (Thorve, Sreshtha and Meng, 2018). Listing 4.7 gave an example of potential race conditions and wasted test performance. Instead of having

one sleep which might be too short or way too generous polling could be used. Robotium provides wait functions to wait for a certain outcome (Joshi, 2014). These wait functions check a condition and if the condition is false wait for a short amount of time to check again. The wait functions try this until success or a timeout is reached. This leads to two advantages: First the overall timeout can be very high, which reduces the risk of failures on slow machines. And second the interval between checking the condition can be low. That reduces the overall test duration as less time is spent sleeping. Listing 4.8 is adapted to use these wait functions. Now no manual sleep is necessary anymore. Albeit the code grew a little bit, which is a hint that refactoring is needed. Common use cases should be put in the test base class, for example, an `assertCondition` or even an `assertColorUnequalAt` function could be provided.

```
mSolo.drag(bottomrightCanvasPoint.x / 2,
    bottomrightCanvasPoint.x / 2, bottomrightCanvasPoint.y /
    2, bottomrightCanvasPoint.y / 2 + canvasCenterTollerance,
    1);
PointF canvasCenter = new PointF((bottomrightCanvasPoint.x +
    widthOverflow) / 2, newBitmapHeight / 2);

assertTrue("Center not set", mSolo.waitForCondition(new
    Condition() {
     @Override
     public boolean isSatisfied() {
         return PaintroidApplication.drawingSurface.getPixel(
             canvasCenter) != Color.TRANSPARENT);
     }
}));
```

Listing 4.8: Adapted test code of Listing 4.7 to not rely on sleeps and thereby removing the race conditions. The resulting code could be refactored to move common assert checks into the base class. [Source code written by the author of this thesis.]

Yet, even with polling, the tests were not running stable. Robotium was not stable. So both the Catroid and the Paintroid team moved to a different testing framework: Espresso. Espresso has a different approach from Robotium. Instead of polling, callbacks can be registered that are called automatically on defined events. This speeds up test execution drastically as there is no sleeping involved. Furthermore, Espresso leads to more

stable tests than Robotium (Genco, 2015), which was also experienced in the Catrobat project. The changes also included usage of JUnit 4 and the official JUnit test runner. Previously a custom test runner had been used due to deficits of Jenkins in the test result collection, that were not relevant anymore.

Finally, the test structure can be changed: Writing more unit tests rather than the brittle user interface tests. This can be seen both as workaround or as proper solution. As workaround since the actual tests are not fixed, but as a proper solution since the program is still tested but there are fewer flaky tests.

Cohn (2010, pages 311–314) coined the term test automation pyramid. At the base of the test automation pyramid in Figure 4.8 are unit tests. Cohn (2010) places service tests in the middle of the pyramid, it is also common to place integration tests and component tests in the middle (Genco, 2015). For Cohn (2010) a service is anything that is done in response to user input, for example, parsing a string. At the top of the pyramid are user interface tests and often also end-to-end tests (Genco, 2015). End-to-end tests test the whole system which can include user interface tests (Vocke, 2018). Most tests should be unit tests as they are in general cheap to write (given testable code), they run stable, and execute very fast. In contrast, user interface tests tend to be brittle: When the user interface changes many tests need to be adapted. User interface tests using the Android Emulator are also very flaky and take long to execute (Genco, 2015).

In practice there is often a reverse pyramid in the form of an ice-cream cone (Fowler, 2012b). That means most tests are user interface tests or end-to-end tests with very few unit tests.

The Test test structure of both Catroid and Paintroid were in the form of such ice-cream cones. Most existing unit tests required the Android Emulator, so-called instrumented unit tests (Google Developers, 2018c), and thus were very similar to user interface tests. There was also a massive amount of user interface tests. This lead to the aforementioned issues.

For unit tests to be easy and therefore cheap to write the source code needs to be testable. Making existing code testable can be quite laborious but leads to better designs and improves the product overall as Feathers (2004) pointed out in *Working Effectively With Legacy Code*.

The Paintroid team invested heavily in improving their code to make it more testable. This lead to roughly 250 unit tests and 370 end-to-end tests

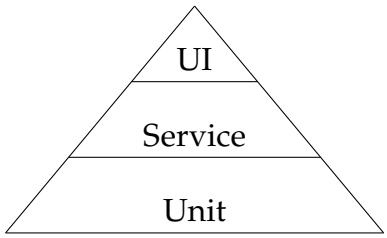


Figure 4.8: Test automation pyramid as described by Cohn (2010, pages 311–324). The focus should be mostly on unit tests. In contrast, user interface tests are brittle, expensive to write and need longer to execute. [Diagram drawn by the author of this thesis.]

which improved the test execution speed. Google itself suggests a share of 70% unit tests, 20 % integration tests, and 10% user interface tests, so there is still much room for improvement (Google Developers, 2018f).

To avoid long transitions to more unit tests a framework like Robolectric[30] can be used. This framework simulates a subset of the Android API, allowing to write user interface tests without any connected device. Tests that rely on functionality not supported by Robolectric would still be written as regular user interface tests to run on a device. For these remaining tests using Android Test Orchestrator can be useful to reduce the shared state between tests, since the instrumentation is not shared and application data is cleared between test runs (Zawadzki, 2018). In 2018 neither the Android Test Orchestrator not Robolectric were used by the Catrobat project.

4.4.4 Detection of Flaky Tests

With a description of what flaky tests are and how to handle them the question still remains of how to detect them on Jenkins. For example, in 2016 the build job `CatroidEmulatorAllTestsSerialNightly` executed roughly 2500 tests of which 40% failed every time. Both the total number of executed tests and the number of failing tests varied during build runs. This could be an indication of Android Emulator crashes. Yet, just by these numbers it is not clear how many of these tests were actually flaky. This sections investigates one manual method to detect flaky tests that can be used on Jenkins.

[30] Robolectric framework: `http://robolectric.org`

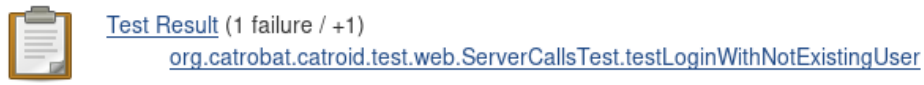


Figure 4.9: The test results overview provides insufficient information to determine whether the failed `testLoginWithNotExistingUser` test was flaky or not. [Screenshot taken by the author of this thesis.]

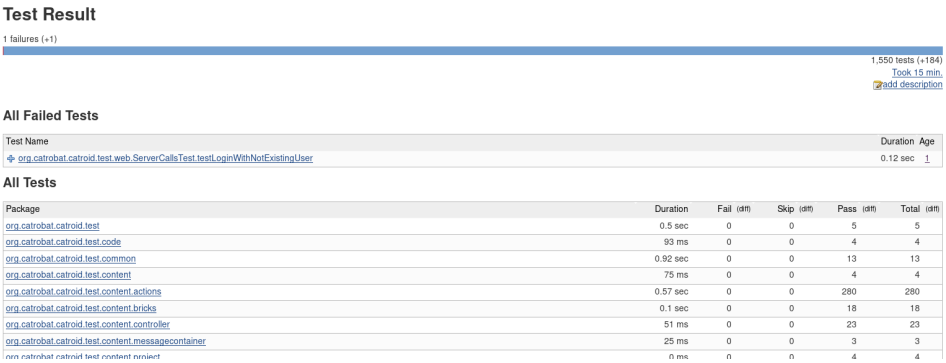


Figure 4.10: The test results view only refers to the current and the previous build. For test failures it tracks for how many builds the test was failing. In this case the age of `testLoginWithNotExistingUser` is one, since the test ran successfully in the previous build. [Screenshot taken by the author of this thesis.]

Figure 4.9 shows the test section of the build results. For this specific build there was one failed test overall. That test did not fail in the previous build which is indicated by the +1. This might be an indication for flakiness but could also indicate a bug. The test overview is insufficient to determine whether there are flaky tests or not. The detailed test results view in Figure 4.10 does not help in this regard either. It shows an age for failed tests, which is the continuous number of builds the test failed. Since `testLoginWithNotExistingUser` succeeded the previous build the age is one. Clearly more information is needed to determine flaky tests.

To determine whether a test is probably flaky more historic information is needed. Especially the state changes from passing to failing are important. This is where the test results analyzer plugin[31] comes in.

By default the test results analyzer provides an overview of the last ten builds as in Figure 4.11, which shows the results for an example project

[31] Test Results Analyzer plugin for Jenkins: `https://plugins.jenkins.io/test-results-analyzer`

New Failures	Chart	See children	Build Number ⇒ Package-Class-Testmethod names ⇓	15	14	13	12	11	10	9	8	7	6
⚠	☐	⊕	org.somedomain.producta.packagea	FAILED	PASSED	FAILED	FAILED	PASSED	PASSED	FAILED	PASSED	FAILED	PASSED
	☐	⊕	org.somedomain.producta.packageb	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED

Figure 4.11: View of the aggregated test results of the last ten builds of an illustrative example for flaky tests. Tests failed in `packagea` while there were no failures in `packageb`. Detailed information can be shown by pressing the plus icon in the *See children* column. [Screenshot taken by the author of this thesis.]

New Failures	Chart	See children	Build Number ⇒ Package-Class-Testmethod names ⇓	15	14	13	12	11	10	9	8	7	6
⚠	☐	⊖	org.somedomain.producta.packagea	FAILED	PASSED	FAILED	FAILED	PASSED	PASSED	FAILED	PASSED	FAILED	PASSED
⚠	☐	⊖	AnotherFlakyTest	FAILED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
⚠	☐		testFlaky1	FAILED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
	☐	⊖	FlakyTest	PASSED	PASSED	FAILED	FAILED	PASSED	PASSED	FAILED	PASSED	FAILED	PASSED
	☐		testFlaky1	PASSED	PASSED	FAILED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
	☐		testFlaky2	PASSED	PASSED	FAILED	FAILED	PASSED	PASSED	FAILED	PASSED	FAILED	PASSED
	☐	⊕	org.somedomain.producta.packageb	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED

Figure 4.12: View of all failing tests of the last ten builds using the test results analyzer Jenkins plugin on an illustrative example. The package and the test classes were manually expanded to show the failing tests. The resulting view can be helpful in finding potential flaky tests as the history of the last ten builds is included. [Screenshot taken by the author of this thesis.]

to illustrate flaky tests. There the aggregated results of the executed test packages are shown. If one test in a package failed then the whole package is treated as failed. In the example `packagea` had failed in the current build 15, which is indicated by the warning sign, and also failed regularly before. It might be that there are flaky tests in `packagea`.

To investigate a test package with failures in more detail the view can be expanded, see Figure 4.12. The test `testFlaky2` in the class `FlakyTest` appears to be flaky since there were many transitions from pased to failed and vice versa. This is not clear for both `testFlaky1`, for these tests the history needs to be increased to consider more than ten builds, which can be configured. In any case the failed tests need to be investigated further to determine the actual issues, which can be done by clicking on the `FAILED` text.

When only few tests fail overall then the top 10 most failed tests view shown in Figure 4.13 is very useful. It can be sufficient to manually detect probable flaky tests.

Test Name	Times Failed	Recent Failed Builds
org.somedomain.producta.packagea.FlakyTest.testFlaky2	4	13, 12, 9, 7
org.somedomain.producta.packagea.AnotherFlakyTest	1	15
org.somedomain.producta.packagea.AnotherFlakyTest.testFlaky1	1	15
org.somedomain.producta.packagea.FlakyTest.testFlaky1	1	13

Figure 4.13: View of the top 10 most failed tests of the test results analyzer Jenkins plugin. A project is used that illustrates flaky tests. [Screenshot taken by the author of this thesis.]

The test results analyzer plugin for Jenkins is a useful tool for finding probable flaky tests as it includes previous builds in its visualisations. There were some disadvantages though in version 0.3.4 of the plugin:

- There was no information on the number of transitions from failed to passed and vice versa. Users had to retrieve that information manually.
- There was no indication in the collapsed view of how often a test package, a test class, or a test failed in the investigated builds. The test package needed to be expanded to retrieve that information by manually looking at previous builds.
- Moreover, in the collapsed view there was no indication of how many tests of a package failed. In this case also expanding the test packages and test classes was necessary to then manually determine the severity of the number of failed tests.
- The user interface was not very compact. Much space was wasted by the *new failures* and the *see children* column. This became impractical when viewing more than just the last ten builds as vertical scrolling was necessary.
- The performance of the user interface was detrimental for projects with a large number of test, like Catroid.
- The performance on Jenkins master was also bad for for projects like Catroid.
- The test results were not collected for Jenkins jobs that start child jobs to delegate the work. The `CatroidEmulatorAllTestsSerialNightly` was such a build flow job that delegated the work to child jobs. The plugin could not be used for that case. Even though it would have been especially useful there to investigate the over 2500 tests with over

1000 failing ones.

The test results analyzer plugin is released under the Apache license, which supports users modifying its code (Apache Software Foundation, 2004). The author of this thesis created pull requests to improve the plugin.

One of the first pull requests made the plugin work with the `Catroid-EmulatorAllTestsSerialNightly` job by aggregating tests of child jobs[32].

Afterwards the focus moved to the performance of the plugin. The creation of the JSON file consumed by the clients took seconds on Jenkins master. The performance was improved by removing unused content and by reducing indirections[33]. Notably, the changes even lead to clearer code including unit tests.

Some of the user interface interactions were very slow too, which was improved by applying more efficient algorithms[34]. For example, collapsing the root node with a fully expanded tree improved from 50 seconds to 250 milliseconds. Some interactions such as expanding all nodes also improved, but to a smaller degree: From more than eight seconds to roughly three seconds. The performance of the plugin could be further improved for better interactivity, but at least the plugin became usable.

The user interface was also improved as shown in Figure 4.14. Tooltips describe each element, enabling simplified user interfaces. For example, the `New Failures` and the `See children` columns were combined with the `Package/Class/Testmethod` column[35]. Two new columns were added as well: A `Passed` and a `Transitions` column. The passed column shows the percentage of succeeding runs of the test package, the test class, or the tests. In the parenthesis the percentage of all succeeding tests that belong to the node is shown. Users can look at the passed column to identify packages with a low percentage of succeeding tests or packages that succeeded rarely. The transitions column lists the number of transitions from failed to succeeding and vice versa, which can be used as indication for flakiness.

With the help of the test results analyzer it turned out that most of the more than 1000 failing Catroid tests were not flaky. They never worked. This is an example of how the quality of a project deteriorates when continuous

[32]Pull request by the author of this thesis: `https://git.io/fhDFI`
[33]Pull request by the author of this thesis: `https://git.io/fhDFs`
[34]Pull request by the author of this thesis: `https://git.io/fhDF2`
[35]Pull request by the author of this thesis: `https://git.io/fhDFo`

Chart	Package/Class/Testmethod	Passed	Transitions	15	14	13	12	11	10	9	8	7	6
☐	org.somedomain.producta.packagea	50% (80%)	7	FAILED	PASSED	FAILED	FAILED	PASSED	PASSED	FAILED	PASSED	FAILED	PASSED
☐	AnotherFlakyTest	90% (90%)	1	FAILED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
☐	testFlaky1	90% (90%)	1	FAILED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
☐	FlakyTest	60% (75%)	6	PASSED	PASSED	FAILED	FAILED	PASSED	PASSED	FAILED	PASSED	FAILED	PASSED
☐	testFlaky1	90% (90%)	2	PASSED	PASSED	FAILED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
☐	testFlaky2	60% (60%)	6	PASSED	PASSED	FAILED	FAILED	PASSED	PASSED	FAILED	PASSED	FAILED	PASSED
☐	org.somedomain.producta.packageb	100% (100%)	0	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED

Figure 4.14: The author made some user interface improvements compared with Figure 4.12. More information is included while the view became more compact at the same time. [Screenshot taken by the author of this thesis.]

integration is not practised anymore. A combination of bad maintained tests and a bad maintained continuous integration system with changes of the environment such as newer Android versions lead to this state.

The test results analyzer is a useful tool to investigate test failures and to detect flaky tests. Its free and open source software license made improvements easy to realise. With its help it could be shown that many flaky test candidates were not flaky anymore by the end of 2016. They always failed. The plugin also proved useful in finding flaky tests created with the Espresso framework. Overall, the stability of the tests was improved massively, mostly by using the Espresso test framework and by applying better testing patterns.

4.5 Independent Jobs

Independent build jobs are a necessity for continuous integration. Reproducible builds are not achievable without them. For Martin Fowler reproducible builds are not a key practice, they are an underlying assumption of continuous integration (Fowler, 2010). If one build job can negatively affect a different build job or even the same build job running at a different time then there is a dependency. Such dependencies need to be avoided.

In 2016 the Catrobat build jobs ran directly on the Jenkins slaves. Many resources were shared between the build jobs:

- user home directory
- Gradle

- Android

All jobs on a slave shared the same user home directory. In that directory programs store their configuration and quite often also persistent data that is created during the usage of the program. This includes, for example, the `$HOME/.java` directory.

Gradle does not only build the Android projects but also handles dependencies. The `GRADLE_USER_HOME` environment variable contains the location where gradle stores its cache, like dependencies it downloaded. If this environment variable is not set then Gradle stores its cache automatically at `$HOME/.gradle`, that means a `.gradle` directory in the user's home directory. Furthermore, the jobs share the Gradle daemon if possible. So instead of a distinct Gradle process per build the builds shared a single Gradle daemon.

There are multiple resources related to Android. The `ANDROID_SDK_ROOT` and the `ANDROID_HOME` environment variables point to the location where the Android SDK is stored. At that location there is also the Android NDK and packages that were installed with the `sdkmanager` executable provided by Android SDK. Some of these Android SDK packages are installed automatically by Gradle using the `sdkmanager` executable (Google Developers, 2018b). The SDK also includes executables that are used during building, and executables that are used to start the Android Emulator and interact with all running devices, including emulators. Then there is also the `$HOME/.android` directory:

- It contains by default the Android Virtual Devices (AVDs), unless the `ANDROID_AVD_HOME` environment variable is set. An AVD represents a mobile device to emulate, including configurations, the system image, and so forth (Google Developers, 2018d).
- The Android specific build cache (Google Developers, 2018a).
- Settings and keys.

Sharing resources can be harmful. For example, when the most recent version of a resource is not compatible with older versions and thereby breaks the build for old branches.

The r18 release of the Android NDK in 2018 caused build errors when the Android Gradle plugin used was of version 3.0 or older (Prichard and Albert, 2018). This affected Catroid, even though Catroid did not need the Android NDK, simply because the Android NDK was installed.

Per Jenkins slave multiple build jobs can run concurrently, potentially all accessing these shared resources at the same time. To work reliable Jenkins depends on all these resources to implement concurrent access correctly, for present and future versions. If builds fail sporadically is this caused due to bad tests, or maybe due to concurrency issues affecting the shared resources? Analysing such cases increases the maintenance load and uncertainty.

As previously mentioned, in 2016 the builds were very unstable, with roughly 1000 Catroid test failures every nightly. The failure causes were unclear, especially since there was a time when the tests had worked.

There were different theories of what caused these issues. Possible causes were newer versions of the Android SDK, the old Android Emulator used by the Android Emulator Jenkins plugin (see Section 4.6), Gradle causing issues when runnning concurrently, and the possibility of increased resource requirements by the newer versions of the software. Also graphic driver updates increased the instability. Another theory was that the adb daemon, part of the Android SDK, was handled incorrectly by the Android Emulator plugin, which often stopped adb while jobs were still running. The long job execution times also made it hard to investigate the issue, especially since they could not be reproduced consistently. At the same time the tests were not maintained well and often depended on timing, see Section 4.4

When the cause of a problem is not clear it is often helpful to simplify the problem by eliminating some possible causes, following the process of elimination. As a result, the author started to research Docker as one method to isolate jobs.

4.5.1 Docker

Docker is a software that supports the isolation of processes from each other by running them in a container. The initial release was in 2013, since then Docker saw huge growth and is very popular today (Cito et al., 2017).

In contrast to virtual machines containers share the kernel of the host system (Docker Inc, 2018c) and do not run a full-fledged operating system (Merkel, 2014). As a result, a Docker container is arguable less secure than a virtual machine, but still quite secure (Docker Inc, 2018b; Walsh, 2014). Yet at the same time sharing the kernel leads to better performance compared with virtual machines (Herrera-Izquierdo and Grob, 2017) and performance

comparable to running software directly on the host (Preeth et al., 2015).

On Linux Docker uses the namespace feature of the Linux kernel (Kerrisk and Biederman, 2018) to isolate the processes within a container from processes of other containers or processes running directly on the host system (Agarwal, 2017). That way processes can have the same process identifier and also provide the same network ports in multiple containers. This leads to simpler deployments. For example, a server process can run in multiple containers and use the same port everywhere, simplifying the code that interacts with the server.

To keep the host filesystem untouched each docker container has its own filesystem that is provided by the image it is based on. Such an image contains everything that is needed to run the desired process, like system libraries and in the case of a Linux image a Linux distribution installed on its root. Many of the Linux images use Debian Linux or Alpine Linux as their base, for example, the OpenJDK images[36].

Docker applies copy-on-write to efficiently share the filesystem of one image. Multiple containers that all use the same image can be started at the same time. These containers cannot affect each others files. Any change to a file is done solely in the container instance where the change originated (Agarwal, 2017).

Docker also supports the rationing of system resources using the so called `cgroups` provided by Linux (Agarwal, 2017). The system memory, swap memory, and CPU time a Docker container uses can be limited, enabling to partition the overall resources of a system. In Section 4.7 the cgroups feature is used to provide better overall performance a deterministic way.

In summary, Docker provides the isolation of processes that is desirable for a continuous integration system. This is the reason why many continuous integration systems like Jenkins (Croy et al., 2018) and Bamboo (Prichard, 2018) support Docker.

A huge benefit of Docker is the ease of creating and running custom-tailored Docker images (Merkel, 2014). The most common approach is to use a Dockerfile that describes the necessary steps for the image creation, so that the image includes all necessary dependencies. Such a Dockerfile can be considered as infrastructure as code.

Listing 4.9 shows a Dockerfile that creates an image based on the Debian

[36]Official OpenJDK Docker images: `https://hub.docker.com/_/openjdk`

Linux distribution version 9.5 and installs the Java 8 SDK using the distribution provided apt-get package manager.

```
FROM debian:9.5

RUN apt-get update && apt-get install -y --no-install-
    recommends openjdk-8-jdk && rm -rf /var/lib/apt/lists/*
```

Listing 4.9: Dockerfile that describes an image based on the Debian Linux distribution with Java 8 installed. [Source code written by the author of this thesis.]

The image can then be created by running `docker build -t my_java8:1 .` in the directory where the Dockerfile is placed. To start a container `docker run` can be used, for example, `docker run -ti my_java8:1 /bin/bash` starts the container and provides an interactive shell for the user.

Docker Hub provides an even easier way to retrieve an image with Java 8 installed: Using one of the official openjdk images[37] directly or as base, for example, `FROM openjdk:8-jdk`. On Docker Hub there are many official and custom images of common software, which simplify deployment of custom applications relying on them.

An import feature of `docker run` is to mount directories or files of the host into the Docker container (Docker Inc, 2018a). This can be done with the `--volume /HOST-DIR:/CONTAINER-DIR` parameter or `-v` in short where `/HOST-DIR` is an absolute path to a directory or file on the host that will be readable and writable from the absolute path `/CONTAINER-DIR` inside of the container. The `--device` parameter exposes specific devices to the container, for example, `--device /dev/kvm:/dev/kvm` exposes the kernel-based virtual machine used by the Android Emulator to the container.

Docker containers isolate processes leading to more security than running the same processes directly on the host machine. The images these containers are based on can be specified easily in a textual configuration format stored in a so called Dockerfile. Therefore, Docker can also be used for packaging all necessary dependencies in the images. On the central Docker Hub repository official images are available for common use cases and custom images can be uploaded as well. Overall, Docker is a very popular and useful software that provides way more than just isolation of processes.

[37] Official OpenJDK Docker images: `https://hub.docker.com/_/openjdk`

4.5.2 Docker and Jenkins

Jenkins 2 provides different ways to interact with Docker (Laster, 2018, page 499):

1. A cloud can be used as Jenkins node to provide Docker containers on the fly as agent to execute jobs.
2. Docker agents can be started automatically on existing Jenkins nodes to run the steps of a declarative pipeline.
3. Global variables exposed by Jenkins can be used to interact with Docker, or alternatively shell commands can be used to interact with Docker directly. This is more complicated than interacting with Docker agents and thus not of interest here.

Once a cloud service has been configured using it from within a Jenkinsfile is as easy as specifying a label in the agent directive (Laster, 2018, pages 499–511). Based on the label the cloud service would provide a Docker container on the fly. Such a cloud node has a dynamic number of executors. When more performance is needed only the capabilities of the cloud need to be extended. For an external cloud provider this would be as simple as changing the cloud configuration or switching to a different payment plan.

```
agent {
    docker {
        image 'openjdk:8-jdk'
    }
}
```

Listing 4.10: Agent directive in a Jenkinsfile that runs the build steps inside of the Docker container based on the image named `openjdk:8-jdk`. In this case the image is retrieved from Docker Hub automatically. [Source code written by the author of this thesis.]

The declarative pipeline, see Section 4.3.3, provides very convenient ways to interact with Docker (Laster, 2018, pages 511-513). In Listing 4.10 the agent directive of a Jenkinsfile is shown. With this directive all build steps would run automatically inside of the Docker image `openjdk:8-jdk` retrieved from Docker Hub. Next to the `image` function call further functions are possible, such as handing the arguments to `docker run` via the `args` function. Jenkins

then runs the Docker container automatically by calling the command `docker run` in the background.

```
agent {
    dockerfile {
        dir 'docker'
        filename 'Dockerfile'
    }
}
```

Listing 4.11: Agent directive in a Jenkinsfile that runs the build steps inside of the Docker container created from a Dockerfile that is placed in a directory called docker. [Source code written by the author of this thesis.]

The `image` invocation relied on an image existing already, either locally or on a Docker registry like Docker Hub. It is also possible to dynamically create an image based on a Dockerfile, which is shown in Listing 4.11. Here the Dockerfile is placed inside of the repository that is checked out and built by Jenkins in the directory called docker.

When building a Docker image Docker includes all files and directories as part of its build environment. These files are not part of the resulting image, unless otherwise specified. Yet including the files in the build environment takes time, depending on their number and size. Therefore, it is beneficial to place the Dockerfile in its own directory to reduce the Docker build environment, as was done above.

Jenkins automatically builds a Docker image using the `docker build` command in the background. Further arguments can be provided to `docker build` with `additionalBuildArgs`. In this case the `args` call is supported for `docker run` as well. Building the Docker image is a one-time cost. Once it was built further calls to `docker build` will use the existing image automatically. Docker also detects modifications to the Dockerfile, which would lead to the creation of a new image.

4.5.3 Docker and Catrobat

Initial Docker support for Catrobat was introduced by relying on a custom image hosted on DockerHub[38]. Later a Dockerfile located in the repository was used[39], with the benefits described above.

One of the disadvantages of this solution was that many resources were still shared, as can be seen by the very long list of arguments handed to `docker run` in Listing 4.12. Resources such as the Android SDK were shared, as well as the Gradle cache, the Java home, the Android home, and further files.

```
args "--device /dev/kvm:/dev/kvm -v /var/local/
    container_shared/gradle/:/.gradle -v /var/local/
    container_shared/android-sdk:/usr/local/android-sdk -v /
    var/local/container_shared/android-home:/.android -v /var/
    local/container_shared/emulator_console_auth_token:/.
    emulator_console_auth_token -v /var/local/container_shared
    /analytics.settings:/analytics.settings"
```

Listing 4.12: Arguments handed to the `docker run` command. [Source code taken from a commit by Michael Musenbrock: `https://git.io/fhDhK`]

To ensure that these resources were actually used inside of the container and that the build job could run successfully, environment variables were set in the environment directive of the Jenkinsfile shown in Listing 4.13.

One reason to share so many files and directories was the location of the home directory inside of the Docker image. By default the home directory is set to the root directory `/`. Yet the user inside of the Docker container used by Jenkins only had reading rights for `/`. When the build job tried to create files in its `$HOME`, such as `$HOME/.java` it failed. So all the locations the build job tried to write to were handed in and thereby shared between the build jobs.

Another reason for sharing directories with the container are performance considerations. Recreating the Gradle cache and redownloading the Android SDK for every build would take very long. So it is sensible to benefit of caching instead.

[38]Commit for Paintroid by Michael Musenbrock: `https://git.io/fhDhK`
[39]Commit for Paintroid by Michael Musenbrock: `https://git.io/fhDhD`

```
ANDROID_SDK_ROOT = "/usr/local/android-sdk"
// Deprecated: Still used by the used gradle version, once
   gradle respects ANDROID_SDK_ROOT, this can be removed
ANDROID_HOME = "/usr/local/android-sdk"
ANDROID_SDK_HOME = "/"
// This is important, as we want the keep our gradle cache,
   but we can't share it between containers
// the cache could only be shared if the gradle instances
   could comunicate with each other
// imho keeping the cache per executor will have the least
   space impact
GRADLE_USER_HOME = "/.gradle/${env.EXECUTOR_NUMBER}"
// Otherwise user.home returns ? for java applications
JAVA_TOOL_OPTIONS = "-Duser.home=/tmp/"
```

Listing 4.13: Environment variables used in the Jenkinsfile so that the handed in arguments of Listing 4.12 are found by the build job inside of the container. [Source code taken from a commit by Michael Musenbrock: `https://git.io/fhDhK`]

What is also notable in Listing 4.13 is the comment for the Gradle cache environment variable `GRADLE_USER_HOME`. Each executor of a Jenkins node needs its own Gradle cache. Gradle does not support concurrent access to its cache across Docker containers[40].

The easiest way to reduce the amount of shared resources was to change the location of the home directory inside of the container to a location that is writable by the user. Further sharing could be avoided by moving the Android dependencies inside of the container, see Section 4.6. Eventually this lead to the simplified list of arguments shown in Listing 4.14. None of the environment variables of Listing 4.13 were needed in the Jenkinsfile anymore. The Gradle cache is still shared, though a different host location has to be used as the Gradle cache is also not relocatable[41].

The usage of Docker for Catrobat changed quite a bit from its introduction to the current state. This highlights the importance of constantly improving the infrastructure whenever issues or possibilities for refactoring arise.

[40] Gradle issue report: `https://git.io/fhye0`
[41] Gradle issue report: `https://git.io/fhyvf`

```
args '--device /dev/kvm:/dev/kvm -v /var/local/
    container_shared/gradle_cache/$EXECUTOR_NUMBER:/home/user
    /.gradle'
```

Listing 4.14: Reduced number of arguments handed to the `docker run` command. [Source code taken from a pull request by the author of this thesis: `https://git.io/fhZDC`]

Docker proved very beneficial for Jenkins Catrobat. The build dependencies became transparent, independent of other build jobs, and with Dockerfile also tied to the source code to build. Thereby there is one single source of truth.

Section 4.6 will describe how Docker can be used for Android dependencies, while Section 4.7 will further investigate Docker usage to achieve a more deterministic and improved performance of the build jobs.

4.6 Android Emulator Handling

Typical Android integration tests and user interface tests need a device to be executed on (Google Developers, 2018f). This can be a physical, therefore real, device or alternatively an emulated device. In the case of Catrobat most devices were emulated.

Emulating devices makes scaling easier, for example, testing different form factors or growing the infrastructure by adding new slaves, without having to bother acquiring new Android devices. Furthermore, managing real devices adds own burdens, such as failing batteries, or state that is kept between build jobs. Of course, emulated devices also have disadvantages, such as emulation bugs, or hiding behaviour real devices exhibit.

Handling the Android Emulator was surprisingly complicated though. The emulator and its dependencies have to be installed, kept up-to-date, configured, and have to be booted and shut-down correctly.

How parts of the dependencies can be managed was already described in Section 4.5. This section focuses on the Android Emulator and its direct dependencies. Different approaches and considerations of how to manage the emulator will be investigated, leading to the solution that is used today.

4.6.1 Android Emulator Jenkins Plugin

For Jenkins there is the Android Emulator plugin that handles all emulator related tasks (Orr, 2017). The plugin can install the Android SDK automatically, create the emulator, start the emulator, remove previously installed APKs, and can also stop the emulator. The plugin has some issues on its own though, which affected Catrobat.

The plugin installed an old version of the Android SDK, which also lead to an old version of the emulator. Stability fixes and performance enhancements of future Android SDK versions were not accessible on Jenkins[42]. There were also issues[43] when trying to use newer versions of the Android build tools.

A disheartening situation resulted where the software used to execute the tests would differ between Jenkins and setups of Catrobat developers. The Catrobat developers would use newer versions of the Android SDK, the emulator, and other Android dependencies, compared with the versions used by the Android Emulator plugin on Jenkins.

This inconsistency made debugging test failures on Jenkins harder and lead to distrust of the Jenkins results. Michael Musenbrock improved the plugin by transitioning to newer versions of the dependencies, which turned out to be a massive endeavour that fixed many long-standing issues[44].

Occasionally the Android Emulator did not shut down correctly, leading to defunct processes. The Android Emulator plugin did not forcefully clean up these processes (Orr, 2017), which lead to resource outages on long running Jenkins slaves.

Probably the most significant issue was that the plugin did not support pipeline jobs. It only worked with freestyle jobs.

This known issue was supposed to be fixed in early 2017[45] after the Free and Open Source Software Developers' European Meeting[46]. Yet no progress

[42]Story to track support for newer Android Emulators in the Android Emulator Jenkins plugin: `https://issues.jenkins-ci.org/browse/JENKINS-40178`

[43]Bug for incompatibilities between the Android build tools 26.0.2 and the Android Emulator Jenkins plugin: `https://issues.jenkins-ci.org/browse/JENKINS-44490`

[44]Pull request by Michael Musenbrock to add Android Emulator 2.0 support to the Android Emulator Jenkins plugin: `https://git.io/fhZDg`

[45]Story to track Jenkins pipeline support for the Android Emulators plugin: `https://issues.jenkins-ci.org/browse/JENKINS-33156`

[46]Post FOSDEM 2017 Hackathon agenda: `https://bit.ly/2ZymFYp`

was made due to colliding ideas of the implementation of some necessary dependencies[47] as the author of the plugin Christopher Orr mentioned in the Internet Relay Chat of Jenkins.

The author of this thesis initially intended to convert the freestyle jobs to pipeline jobs using Jenkinsfile, since the latter was the preferred way of creating jobs in Jenkins 2 (Laster, 2018, page 2). The only obstacle was the Android Emulator plugin dependency that did not support pipeline jobs.

With no progress on the pipeline support of the plugin at the beginning of 2017 it was necessary to decide whether the Catrobat Jenkins team should manage the emulator and everything related themselves, or whether to wait for the support to arrive.

To avoid additional maintenance costs it was decided to keep using the Android Emulator plugin in the hopes that support would arrive later in 2017. As a stop-gap measure the jobs would be managed with the Job DSL instead, see Section 4.3.

4.6.2 Managing the Android Emulator with Bash and Python

In 2018 Michael Musenbrock decided to port the existing jobs to pipeline jobs. As a result, the Jenkins Android Emulator plugin could not be used anymore, as there still was no pipeline support present. Instead Bash scripts were written by Musenbrock to perform the necessary steps[48].

These Bash scripts could also run on the machines of the developers. Yet Windows was not supported by default, as it does not come with a Bash installation. This lead to porting the scripts to Python to add Windows support.

Listing 4.15 provides an example of the usage of the Bash helper scripts to manage the emulator. There are steps to create the AVD for the emulator which is then started to, finally, execute some tests. Not shown here are the steps to install the Android SDK and other dependencies which was also done by the helper scripts.

[47] See pull request: `https://git.io/fhZDM`

[48] See the source code at `https://git.io/fhDXV`.

```
// create emulator
sh "jenkins_android_emulator_helper -C -P 'hw.camera:yes' -P
    'hw.ramSize:800' -P 'hw.gpu.enabled:yes' -P 'hw.camera.
    front:emulated' -P 'hw.camera.back:emulated' -P 'hw.gps:
    yes' -i '${ANDROID_EMULATOR_IMAGE}' -s xhdpi"
// start emulator
sh "jenkins_android_emulator_helper -S -r 768x1280 -l en_US -
    c '-gpu swiftshader_indirect -no-boot-anim -noaudio'"
// wait for emulator startup
sh "jenkins_android_emulator_helper -W"
// Run Unit and device tests for package: org.catrobat.
    catroid.test
sh "jenkins_android_cmd_wrapper -I ./gradlew test
    connectedCatroidDebugAndroidTest -Pandroid.
    testInstrumentationRunnerArguments.package=org.catrobat.
    catroid.test"
```

Listing 4.15: Parts of a Jenkinsfile to create and start the Android Emulator via Bash helper scripts. [Source code taken from `https://git.io/fhZpM`]

To simplify the steps needed for AVD creation a config file was created, as shown in Listing 4.16, that would be read by the Python helper scripts. The Python helper scripts were placed alongside the project they were used on. That means they were both placed in the Catroid and the Paintroid repositories. To make it easier to run the same steps locally and on Jenkins all the steps were moved into their own scripts. Now the code from Listing 4.15 could be simplified to just one line as shown in Listing 4.17.

This solution worked very stable. It allowed to transition the jobs to Jenkins pipeline. Yet there were also some disadvantages. The most obvious disadvantage was that Python, an external dependency, needed to be installed. In general Python is installed on Linux machines, while it is absent by default on Windows installations. Developers who want to use these scripts on Windows need to install and configure Python themselves. Another disadvantage was that the scripts existed in both the Catroid and Paintroid repository. This duplication made maintenance harder, since issues needed to be fixed in both repositories. Furthermore, only one configuration could be specified at once. For a different emulator configuration the `emulator_config.ini` file needed to be modified. The complexity of build jobs increased when they performed modifications of the emulator configuration.

```
# AVD creation
system_image=system-images;android-24;default;x86_64
## properties written to the avd config, prefix here with
   prop, so the script knows where to use them
prop.hw.camera=yes
prop.hw.ramSize=2048
prop.hw.gpu.enabled=yes
prop.hw.camera.front=emulated
prop.hw.camera.back=emulated
prop.hw.gps=yes
prop.hw.mainKeys=no
prop.hw.keyboard=yes
prop.disk.dataPartition.size=512M
## dpi
screen.density=xxhdpi
## sdcard
sdcard.size=200M
## AVD startup
screen.resolution=1080x1920
device.language=en_US
```

Listing 4.16: Configuration for Android Emulator used by the Python helper scripts. [Configuration taken from `https://git.io/fhZpw`]

```
"./buildScripts/build_step_run_tests_on_emulator__test_pkg"
```

Listing 4.17: Parts of a Jenkinsfile to create and start the Android Emulator via Python helper scripts. The code is simpler compared with the initial Bash implementation in Listing 4.15. [Source code taken from a commit by Michael Musenbrock: `https://git.io/fhZpH`]

The Python implementation was basically a direct port from Bash. There were many global variables and the code was both hard to read and maintain. Just by looking at the Jenkinsfile it was unclear which steps were performed. For example, in Listing 4.17 it was not clear which gradle tasks were executed. For this the code had to be executed, or the scripts had to be investigated manually.

4.6.3 Managing the Android Emulator with Gradle

To avoid the disadvantages of the Bash and Python implementations the author of this thesis decided to implement the management of the Android Emulator directly within a Gradle plugin.

At first the already existing Gradle plugin `android-emulator-gradle`[49] was investigated for its feasibility. Unfortunately, it did not provide sufficient features. The plugin did not handle the installation of the Android SDK and the Android NDK, furthermore only one emulator could be specified at the same time.

In the first iterations the plugin was only implemented for Paintroid. The Groovy code was placed inside the `buildSrc` directory of the Paintroid repository. This `buildSrc` directory is recognised by Gradle automatically and can then be used in the other Gradle build files (Wendelin et al., 2018). Later the plugin was implemented as its own project[50] that was released on JFrog Bintray[51] and also accessible on JCenter.

To use the plugin only an additional dependency needs to be added to the `build.gradle` file. As a result, there is no duplication between Catroid and Paintroid, both projects rely on released versions of the plugin.

The Gradle plugin does not add any further external dependencies, Python is not necessary anymore. Similar to the Bash and the Python code the plugin can install the Android SDK, the Android NDK and other dependencies like the Android Emulator. The emulators to use can be managed directly in the `build.gradle` file using a custom domain-specific language shown in Listing 4.18.

For clarity the configuration of an emulator is split into an `avd` and a `parameters` part. The `avd` part rerfers to the parameters for the AVD image creation, while `parameters` refers to the parameters the emulator is then started with.

With the Gradle plugin the Jenkinsfile could be cleaned up as well. In Listing 4.17 the work of the build step was delegated to a shell script. This is not necessary anymore. The Jenkinsfile becomes clearer by directly calling

[49] Gradle plugin for Android Emulator: `https://github.com/gocal/android-emulator-plugin`

[50] Gradle project for Catrobat initiated: `https://github.com/Catrobat/Gradle`

[51] Android Emulators Gradle plugin published on JFrog Bintray: `https://bintray.com/catrobat/Gradle/org.catrobat.gradle.androidemulators`

Gradle, see Listing 4.19. The plugin reduces code duplication and also makes it possible for external projects to rely on its functionality.

```
apply plugin: 'org.catrobat.gradle.androidemulators'

emulators {
    install project.hasProperty('installSdk')

    dependencies {
        sdk()
        ndk()
    }

    emulator 'android24', {
        avd {
            systemImage = 'system-images;android-24;default;
               x86_64'
            sdcardSizeMb = 200
            hardwareProperties += ['hw.ramSize': 800, 'vm.
               heapSize': 128]
            screenDensity = 'xhdpi'
        }

        parameters {
            resolution = '768x1280'
            language = 'en'
            country = 'US'
        }
    }
}
```

Listing 4.18: Configuration of the Android Emulator in the `build.gradle` file of Paintroid using the Gradle plugin. All necessary steps are specified via a Groovy domain-specific language. No additional configuration, like in Listing 4.16, is necessary. [Source code taken from a pull request by the author of this thesis: `https://git.io/fhZpK`]

Multiple emulators can be specified in the `build.gradle` file. The desired emulator is then selected via the `-Pemulator` Gradle parameter as is done in Listing 4.19 with `-Pemulator=android24`. The same build can use different emulators, for example, to perform regression tests for a specific Android version only.

```
sh '''./gradlew -PenableCoverage -Pemulator=android24
    startEmulator createCatroidDebugAndroidTestCoverageReport
    \
     -Pandroid.testInstrumentationRunnerArguments.package=org.
        catrobat.catroid.test'''
```

Listing 4.19: Parts of a Jenkinsfile to create and start the Android Emulator via the android-emulators-gradle plugin. In contrast to Listing 4.17 the necessary Gradle parameters are transparent, since there are no external dependencies. [Source code taken from a pull request by the author of this thesis: `https://git.io/fhZpP`]

When defining emulators templates can be used for common settings. Such a template is specified exactly like an emulator, but instead of the function `emulator` in Listing 4.18 the function `emulatorTemplate` has to be called. To refer to a template the `emulator` function also accepts a third parameter: The name of the template. An example is shown in Listing 4.20 where an emulator with the name `android19` is created. The emulator inherits the settings of the template with the name `template1` and only specifies a system image for Android 19. The templates themselves can also be based on other templates. Emulators can be treated as templates themselves, so it is possible to base an emulator `android20` on `android19` of Listing 4.20. This supports complex scenarios directly via the domain-specific language and enables the usage of groovy abstractions where further complexity is needed.

```
emulator 'android19', 'template1', {
    avd {
        systemImage = 'system-images;android-19;default;x86'
    }
}
```

Listing 4.20: Creating an emulator `android19` based on a previously defined template `template1`. Every setting of the template is inherited, just the `systemImage` is overriden. [Source code taken from a pull request by the author of this thesis: `https://git.io/fhZpP`]

Another change compared with the previous solutions is the existence of tests. Parts of the code are unittested and were written in a test driven design fashion. As such best practices can also be applied to this repository. This includes the presence of continuous integration support. Pull requests trigger automatic builds that run the unit tests and archive the resulting

plugin. Further work is needed to both track the test coverage and to increase the coverage.

Yet the Gradle plugin has some disadvantages itself. Now it is not possible to run the same emulator concurrently. This disadvantage is alleviated by running the builds in Docker. As Docker isolates the builds multiple builds can use an emulator with the same specification, see Section 4.5.

4.6.4 Android Dependencies in Docker

So far, it was described how the Android dependencies are installed. This section highlights were these dependencies are installed to, how the build jobs access them, what trade-offs to consider, and how Docker can improve the situation.

The Android SDK, the Android NDK, the emulator and other Android dependencies are searched for at the location specified by the environment variable `ANDROID_SDK_ROOT` outlined in Section 4.5. This is also the location these dependencies are installed to by the scripts and the Gradle plugin described above.

Part of the issue with the install location is how to manage different builds. Section 4.5 emphasised the importance of independent builds. At the same time performance is crucial for the acceptance of a continuous integration system. Therefore, not every build can download and install these dependencies. There needs to be some form of sharing the Android dependencies to improve the performance.

The first step of sharing the Android installation was by defining one `ANDROID_SDK_ROOT` per slave. With a single location for multiple jobs concurrent access needs to be managed though. There would be race conditions otherwise. The Android Emulator Jenkins plugin took care of this by employing a mutex to only allow exclusive access during installation of the dependencies[52]. Users of the plugin do not need to adapt their jobs to avoid race conditions. In contrast, job modifications are needed for the solutions relying on the Bash scripts, the Python scripts, and the Gradle plugin discussed previously.

[52]Confer the code in `SdkInstaller.java`: `https://git.io/fhZDd`

```
stage('Setup Android SDK') {
    steps {
        // Install Android SDK
        lock("update-android-sdk-on-${env.NODE_NAME}") {
            sh './gradlew -PinstallSdk'
        }
    }
}
```

Listing 4.21: Stage of the Catroid Jenkinsfile to install Android dependencies with the `android-emulators-gradle` plugin. The installation step is protected by a lock on that slave (node). This avoids race conditions of multiple jobs performing the installation at the same time. [Source code taken from `https://git.io/fhnfm`]

An own stage in the Jenkinsfile takes care of locking the resource. The resource in this case is the `ANDROID_SKD_ROOT` directory per slave. In Listing 4.21 the resource is locked to then perform the installation of the dependencies with the Gradle plugin. Such an extra step adds complexity to the Jenkinsfile and thus makes creating jobs with Jenkins pipeline harder. Forgetting to manually lock the correct resources for just one job can lead to a corrupted installation of the Android dependencies which might require manual intervention. Even when locking the resources during installation there is a potential of race conditions: Only the writing to the `ANDROID_SKD_ROOT` directory is locked, not the usage of files in that directory when creating and starting the emulator. This race condition did not lead known to issues yet though.

Another disadvantage of having just one location per slave for the Android dependencies was discovered early. Catroid and Paintroid might have different requirements for the dependencies, for example, recent Catroid does not need the Android NDK, or a different version for the Android SDK might be necessary. This resulted in three directories per slave for the Android dependencies:

- `android-sdk` was initially used by both Catroid and Paintroid. The Android NDK version was fixed to `r16`. Newer versions of the NDK did not support MIPS[53] which was necessary for Catroid (Albert, 2018).

[53]MIPS processors: `https://www.mips.com/products`

- `android-sdk-paintroid` supported the newest Android NDK for Paintroid. Paintroid did not have the MIPS support issues and thus could support newer NDK versions already.
- `android-sdk-ndk-latest` used later by both Catroid[54] and Paintroid[55]. Paintroid did not face the MIPS support issues and with the MIPS support removed from Catroid[56] both could share the same dependencies again.

Even worse was the situation when the newer version r18 of the Android NDK was incompatible with the current source code (Prichard and Albert, 2018). In `android-sdk-ndk-latest` always the latest version of the Android NDK was installed. That lead to build errors on Jenkins. A fix to the source code for Paintroid[57] sorted out the build errors. Of course the fix only affected the source code after and including that commit. Unfortunately, older Paintroid source code cannot be built on Jenkins anymore. The exact same issue also applied to Catroid[58].

One reason for placing the Jenkinsfile in the source code repository was to keep older revisions of the code working on Jenkins. The Android dependencies though worked against backward compatibility. To solve this issue all the dependencies were moved to the Docker file.

Thus the `android-emulators-gradle` plugin was only used to create the emulator and to start and stop it. It is not used anymore to install the dependencies for Catroid[59] and Paintroid[60].

The version of the Android SDK and the Android NDK are pinned inside of the Docker image. Other Android dependencies, like the version of the emulator, are not pinned at this time, as no version incompatibilities have been found there so far. Pinning the version of a dependency is much work that is better avoided when not necessary.

[54] Confer pull request for Catroid by Michael Musenbrock: `https://git.io/fhZDy`
[55] Confer pull request for Paintroid by Michael Musenbrock: `https://git.io/fhZDS`
[56] Pull request by Thomas Schwengler to remove MIPS support from Catroid: `https://git.io/fhZDu`
[57] Pull request by Thomas Schwengler to make Paintroid work with Android NDK r18: `https://git.io/fhZD0`
[58] Commit by Thomas Schwengler to make Catroid work with Android NDK r18: `https://git.io/fhZDB`
[59] Pull request for Catroid by the author of this thesis: `https://git.io/fhDAH`
[60] Pull request for Paintroid by the author of this thesis: `https://git.io/fhZDC`

Now it is transparent which version of the SDK and the NDK is used for a given source code revision. Pull requests can try to increase the version of the SDK or NDK without affecting any other builds. Consequently, issues like encountered with Android NDK r18 are avoided. Furthermore, these changes simplified the local replication of the environment that is used on Jenkins, without needing multiple `ANDROID_SKD_ROOT` directories. A further advantage is that no new release of the `android-emulators-gradle` plugin is necessary to pin newly released versions of the Android SDK or the Android NDK.

Race conditions for the Android dependencies are impossible now, as they are not shared anymore. This decreases the complexity of the Jenkinsfile, as no locking of Android dependencies is necessary anymore. A disadvantage is the increased complexity of the Dockerfile. Changes of the Android dependencies have to be placed there in contrast to the build.gradle file.

4.6.5 Discussion

The Android Emulator proved to be a main source of issues. On the one hand it often ran very unstable and unpredictable, depending on the version used, which lead to flaky tests and even crashes. Updates to the related Jenkins plugin improved the situation a bit.

On the other hand the emulator was hard to manage. The emulator was not directly supported by Jenkins pipeline, which slowed down the adoption of Jenkins pipeline. The alternative, for the team to manage the emulator and all its dependencies, was very complex and time consuming. Testament to this complexity are the four different approaches tried: First using Bash scripts to manage the emulator, then Python, then Gradle, and finally Gradle in combination with Docker.

The complexity of the build jobs increased by directly managing the Android Emulator. At the same time that step proved successful: The emulator became more stable, different versions of the dependencies could be used, and a transition to Jenkins pipeline was possible.

While in hindsight relying directly on Docker and Gradle would have saved time it would have been very unlikely for such a complex solution to be considered initially. The gained experience was key in devising the final solution.

4.7 Performance

Build performance is crucial for continuous integration. Duvall, Matyas and Glover (2007, pages 87–88) even suggested that no build should take more than 10 minutes to complete. In contrast, the `CatroidEmulatorAllTests-SerialNightly` job took five hours on average to execute. With multiple-hour build times and all the other issues it is no surprise that continuous integration was not really practised in 2016. This section discusses improvements to the build performance and provides outlooks for further improvements.

Profiling is at the starting point of optimisations. For both Catroid and Paintroid most of the build time was used for end-to-end tests. Improving their performance would have the greatest potential.

Trying to use the build infrastructure more efficiently might sound like the best approach to improve performance in general. For example, executing tests in parallel to benefit from the multiple CPU cores modern servers have. Not surprisingly though better hardware or better utilisation of hardware only leads to linear improvements of performance: Work performed on one single CPU core will take at best one fourth of the time if performed on four CPU cores. For computation-heavy tasks often the most significant performance gains can be achieved by using better algorithms, not better hardware (Skiena, 2008, pages 51–54).

Applied to testing this means to use a different and especially faster test that performs equivalent checks. So instead of writing slow end-to-end tests rather unit tests should be written, as mentioned in Section 4.4. In 2019 the 250 Paintroid unit tests took roughly 10 seconds to execute, while the 370 end-to-end tests took more than 10 minutes, a difference of two orders of magnitude.

When unit tests are not feasible using faster test frameworks is beneficial. Both Catroid and Paintroid transitioned their end-to-end tests to the Espresso framework. This not only improved the stability of the tests but also significantly improved the performance. With Espresso a Catroid build with more than 1500 tests took roughly 30 minutes, half of which was spend on end-to-end tests. With Robotium in 2016 this still took five hours on average, though packages with failing tests were rerun back then, see Section 4.4.

Parallelism can only be beneficial if enough hardware is available. In general parallelising tasks incurs an overhead, perfect scaling is hardly achievable. Up until 2017 each Jenkins node had six executors and thus

could theoretically execute six build jobs at once. Unfortunately, hardware requirements increased. This was probably caused by newer Android and Gradle versions being more demanding and by performance degradations caused by fixes for hardware related security issues such as Meltdown (Lipp et al., 2018) and Spectre (Kocher et al., 2019).

By the end of 2018 each node was restricted to just one executor. Using more executors lead to unstable builds as the 16 gigabyte of system memory was insufficient. With just three nodes available for both Catroid and Paintroid continuous integration jobs the potential benefits of parallelism were rather low. Furthermore, Gradle, Java, as well as the Android Emulator already benefited from multiple CPU cores. That meant that running more tasks on a single Jenkins slave would reduce the performance of each task, but might lead to better performance overall.

As previously mentioned the build jobs were very greedy in terms of system memory, prohibiting two concurrent jobs on a single slave. Restricting the system memory available to the build jobs was necessary to increase the number of executors to two per slave. This can be done by a Docker feature that abstracts cgroups provided by the Linux Kernel. With the `docker run` parameter `--m=6.5G` a container would use at most 6.5 gigabyte of system memory. Java version 8 used inside of the Docker container does not recognise this value, it needs additional settings inside of the Docker file (Smith, 2017). With these adaptations two jobs could run concurrently without stability issues[61].

At the beginning of 2019 a Catroid build took between 26 and 30 minutes: The end-to-end tests took between 14 and 16 minutes, the static analysis between 5 and 8 minutes, instrumented unit tests around 2 minutes, code coverage collection roughly 1:30 minutes, and the rest was used by the remaining tasks such as APK generation.

The end-to-end tests set the limit of what could be achieved by parallelising the tasks of the Catroid job. One improvement was to execute the end-to-end tests on one node and all the other tasks on another[62]. This lead to build times between 15:30 and 17:30 minutes, as the code coverage report still had to be done at the end of the build job.

The code coverage is collected by a plugin for Gradle. This plugin creates

[61]Commit for Catroid by the author of this thesis: `https://git.io/fhSaU`
[62]Commit by the author of this thesis: `https://git.io/fhSnS`

an HTML and an XML report. Jenkins read this XML report and then created its own HTML report by also reading all the source files. A change was to publish the HTML report directly instead[63], which had some usability restrictions, but saved 1:30 minutes.

Finally, an improvement was to optimise settings for the Android Emulator, for example, reducing the resolution, which lead to end-to-end tests completing in 13 to 15 minutes. The builds themselves also completed in this time, thus the build times were cut in half.

Paintroid jobs already finished in 14 minutes after the transition to Espresso end-to-end tests. Further speed-ups would be possible by parallelising the tests themselves. This can either be done by putting them manually into multiple test suites, each with roughly the same amount of work, or by relying on Jenkins to split up the tests with the Parallel Test Executor plugin[64]. Combining both approaches does not work, as that plugin does not work with test suites, such as used for Catroid. Unfortunately, a bug for the JUnit Jenkins plugin undermined the usability of parallelising tests. Even though the bug had been fixed[65] it took very long for the changes to be merged. In the future also the performance of the Paintroid job can be improved, potentially leading to builds below 10 minutes.

The build performance of the Catroid and Paintroid jobs improved vastly over the last years. The most significant gains were caused by improved tests, which is also the area with most potential still. Further optimisations and better utilisation of the resources available lead to jobs that execute in less than 15 minutes. With additional hardware it should be feasible to achieve builds in under 10 minutes.

[63] Commits by the author of this thesis: `https://git.io/fhSnH`, `https://git.io/fhSn7`

[64] Parallel Test Executor Jenkins plugin: `https://plugins.jenkins.io/parallel-test-executor`

[65] Pull request by the author of this thesis: `https://git.io/fhSam`

5 Conclusion

The origin of this thesis was the dire state the Jenkins system for Catrobat was in at the end of 2016. At that time the method of continuous integration was practically not applied anymore due to technical issues. Therefore, the goal was to provide a continuous integration system that would be used again and that would stay maintainable at the same time. The maintainability should enable a system that would be feasible in the long run. Of course, key for the usability of a system would be close cooperation with its users.

Chapter 2 described the concept of continuous integration, highlighting the necessary preconditions, and listing benefits. For continuous integration to work properly a system, like Jenkins, would execute jobs that build and test software on any change. Such a build would only be treated as success when all tests and additional checks passed.

For Catrobat the focus was on the Android applications Catroid and Paintroid, described in Chapter 3 along with the Catrobat project.

The main part of the thesis was Chapter 4, which focused on the issues faced by the Catrobat Jenkins team and the solutions devised. One of the first steps taken was to improve the documentation and to add a small set of policies to avoid similar situations in the future, as described in Section 4.1.

The amount of issues seemed overwhelming, which is why a separate test server was installed, to create a clean Jenkins configuration from scratch. At a later stage the knowledge gained was transferred to the operational Jenkins system. An advantage of that approach was the learning experience for the team. Nonetheless, a disadvantage was the long time it took to translate the introduced changes to the operational server to benefit the users. Especially, since the test server was only intended for the Catrobat Jenkins team. As a result, both the operational and the test server were long in a state that did not support the continuous integration needs of the users. A potentially better approach would have been to clean up the operational server directly.

Configuration as code was researched for the server infrastructure and

Jenkins configuration in Section 4.2 and for build jobs in Section 4.3. This approach lead to better accountability of configuration changes, decreased the number of jobs required, and also reduced the amount of necessary documentation. There were two domain specific languages to configure Jenkins jobs: The Job DSL and Jenkins pipeline.

The Jenkins pipeline was the recommended system to create jobs for Jenkins version 2. Unfortunately, it did not support the Android Emulator Jenkins plugin used by the existing jobs. As a result, the Job DSL was used initially. The hope was to transition to Jenkins pipeline once the plugin was supported, which was planned for 2017. However, support was never added. The Catrobat Jenkins team still transitioned to pipeline, with the increased burden of managing the Android Emulator with custom scripts, as described in Section 4.6. In retrospect, using Job DSL was beneficial, yet even better would have been to apply Jenkins pipeline directly and only using Job DSL to create these pipeline jobs, as is the current state.

Probably the main issue from the perspective of the users was that a tremendous amount of Catroid and Paintroid tests failed on Jenkins, leading to broken builds. With more than 1000 Catroid test failures and very long execution times continuous integration was not practised anymore. Initially it was believed that most test failures were caused by flaky tests. Consequently, it was investigated in Section 4.4 whether the tests were flaky or broken.

Most of the flaky test candidates failed always at the end of 2016, they were thus not flaky. Although it was unclear what the cause was, since they had worked once. Most likely the Android Emulator and changes to the Android SDK caused these issues. To reduce the potential causes of test failures it was tried to make the build jobs as independent of each other as possible, which was the focus of Section 4.5. The container system Docker was applied to reduce the resources shared between the build jobs.

Much work was put into improving the situation of the Android Emulator, see Section 4.6. This included work by Michael Musenbrock on the Android Emulator Jenkins plugin, as well as his work on managing the emulator directly via Bash and Python scripts. Eventually the emulator was managed directly with a Gradle plugin inside of a Docker container.

Section 4.7 highlighted the importance of performance of build jobs. Measures were described that halved the Catroid build times. Above all, the most beneficial improvement was the transition of the Catroid and Paintroid

teams to the Espresso test framework, which not only reduced the test execution times but also increased the test stability.

Although the initial scope seemed well-defined the existing issues lead to researching many different technologies, which was very time-consuming. For instance, technologies to manage the server installations and their configuration. Probably better would have been to postpone the setup of the servers and to focus solely on the job stability first, although it was unclear initially whether these were related. In contrast, using configuration as code for the build jobs worked very well. It enabled a previously unknown traceability and flexibility that could be considered future-proof. Providing users with new functionality became easy. Docker also proved successful in serving build environments, that were independent of the slaves and other build jobs. This was especially useful for isolating the Android Emulator and also for installing the complex dependencies of the release jobs.

In the future Docker will support continuous integration for the Catrobat web infrastructure, which was already planned for 2018, but then postponed due to a lack of resources in the web team. Recent developments in the Jenkins community will probably be beneficial as well and would have saved much research time if they existed two years ago. For example, the Jenkins Evergreen project might save time to keep Jenkins updated and well maintained. Jenkins configuration as code will help to keep track of the configuration. Another interesting area of research that was not pursued due to lack of time is running Jenkins on the cloud for better scaling.

From 2016 to 2019 the situation of Jenkins for Catrobat improved massively. Continuous integration was practised again. The different stakeholders were also provided with tools that made their work easier and more enjoyable. Some of the Catrobat Jenkins team even implemented the foundations for continuous deployment, which already improved the release process. At the same time the system became well documented and there were steps in process to improve the situation even further. There was a long journey from a dysfunctional system to a functional one, including fruitless approaches and numerous hours of debugging, research, and implementation efforts. The end result is a quite satisfactory continuous integration system for both its users and maintainers that provides the foundation for future developments.

Bibliography

Agarwal, Nitin (2017). *Get Started, Part 1: Orientation and setup*. URL: https://medium.com/@nagarwal/understanding-the-docker-internals-7ccb052ce9fe (visited on 03/12/2018) (cit. on p. 58).

Albert, Dan (2018). *Changelog r17*. URL: https://github.com/android-ndk/ndk/wiki/Changelog-r17 (visited on 21/12/2018) (cit. on p. 73).

Apache Software Foundation (2004). *Apache License, Version 2.0*. URL: https://apache.org/licenses/LICENSE-2.0.html (visited on 07/01/2019) (cit. on p. 54).

Bayer, Andrew et al. (2018). *Pipeline Syntax*. URL: https://jenkins.io/doc/book/pipeline/syntax (visited on 07/01/2019) (cit. on pp. 36, 38).

Beck, Kent and Cynthia Andres (2005). *Extreme Programming Explained: Embrace Change*. 2nd ed. Addison-Wesley. ISBN: 0321278658 (cit. on p. 5).

Biro, Meghan M. (2018). *Secret Productivity Killer: Too Many Rules in the Workplace*. URL: https://www.huffingtonpost.com/meghan-m-biro-/secret-productivity-kille_b_14254046.html (visited on 07/01/2019) (cit. on p. 22).

Brooks Jr, Frederick P (1995). *The Mythical Man-Month: Essays on Software Engineering*. 2nd ed. Addison-Wesley. ISBN: 0201835959 (cit. on p. 5).

Burtscher, Daniel (2016). 'Introduction of a Continuous Integration Process in an Open Source Project'. MA thesis. Graz University of Technology (cit. on p. 15).

Chacon, Scott (2009). *Pro Git: Everything you need to know about the Git distributed source control tool*. Apres. ISBN: 9781430218333 (cit. on pp. 8, 9).

Cito, Jürge et al. (May 2017). 'An Empirical Analysis of the Docker Container Ecosystem on GitHub'. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 323–333. DOI: 10.1109/MSR.2017.67 (cit. on p. 57).

Cohn, Mike (2010). *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley. ISBN: 9780321579362 (cit. on pp. 49, 50).

Croy, R. Tyler et al. (2018). *Using Docker with Pipeline*. URL: https://jenkins.io/doc/book/pipeline/docker (visited on 07/01/2019) (cit. on p. 58).

Cunningham, Ward (1993). 'The WyCash portfolio management system'. In: *ACM SIGPLAN OOPS Messenger* 4.2, pp. 29–30 (cit. on p. 2).

DeMarco, Tom and Timothy Lister (2013). *Peopleware: Productive Projects and Teams*. 3rd ed. Addison-Wesley. ISBN: 9780321934116 (cit. on pp. 13, 22, 25).

Docker Inc (2018a). *Docker run reference*. URL: https://docs.docker.com/engine/reference/run (visited on 03/12/2018) (cit. on p. 59).

Docker Inc (2018b). *Docker security*. URL: https://docs.docker.com/engine/security/security (visited on 31/01/2019) (cit. on p. 57).

Docker Inc (2018c). *Understanding the Docker Internals*. URL: https://docs.docker.com/get-started (visited on 07/01/2019) (cit. on p. 57).

Duvall, Paul M., Steve Matyas and Andrew Glover (2007). *Continuous Integration*. Addison-Wesley. ISBN: 9780321336385 (cit. on pp. 5–7, 10–13, 44, 76).

Feathers, Michael C. (2nd Oct. 2004). *Working Effectively With Legacy Code*. Prentice Hall PTR. ISBN: 0131177052 (cit. on p. 49).

Fowler, Martin (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley. ISBN: 0201485672 (cit. on p. 14).

Fowler, Martin (2006). *Continuous Integration*. URL: https://martinfowler.com/articles/continuousIntegration.html (visited on 07/01/2019) (cit. on pp. 1, 6).

Fowler, Martin (2009). *Feature Branch*. URL: https://martinfowler.com/bliki/FeatureBranch.html (visited on 20/01/2019) (cit. on p. 9).

Fowler, Martin (2010). *Reproducible Build*. URL: https://martinfowler.com/bliki/ReproducibleBuild.html (visited on 07/01/2019) (cit. on pp. 14, 55).

Fowler, Martin (2011a). *Eradicating Non-Determinism in Tests*. URL: https://martinfowler.com/articles/nonDeterminism.html (visited on 07/01/2019) (cit. on pp. 44, 45, 47).

Fowler, Martin (2011b). *Frequency Reduces Difficulty*. URL: https://martinfowler.com/bliki/FrequencyReducesDifficulty.html (visited on 19/01/2019) (cit. on p. 6).

Fowler, Martin (2012a). *Snowflake Server*. URL: https://martinfowler.com/bliki/SnowflakeServer.html (visited on 07/01/2019) (cit. on pp. 29, 31).

Fowler, Martin (2012b). *Test Pyramid*. URL: https://martinfowler.com/bliki/TestPyramid.html (visited on 07/01/2019) (cit. on p. 49).

Free Software Foundation (2017). *What is free software?* URL: https://www.gnu.org/philosophy/free-sw.en.html (visited on 07/01/2019) (cit. on p. 15).

Gaskell, Giles et al. (2018). *Using a Jenkinsfile*. URL: https://jenkins.io/doc/book/pipeline/jenkinsfile (visited on 07/01/2019) (cit. on p. 38).

Genco, Sebastian Lobato (2015). *The evolution journey of Android GUI testing*. URL: https://medium.com/@sebaslogen/the-evolution-journey-of-android-gui-testing-f65005f7ced8 (visited on 07/01/2019) (cit. on pp. 46, 49).

Google Developers (2018a). *Accelerate clean builds with the build cache*. URL: https://developer.android.com/studio/build/build-cache (visited on 27/11/2018) (cit. on p. 56).

Google Developers (2018b). *Auto-download missing packages with Gradle*. URL: https://developer.android.com/studio/intro/update#download-with-gradle (visited on 27/11/2018) (cit. on p. 56).

Google Developers (2018c). *Build instrumented unit tests*. URL: https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests (visited on 27/11/2018) (cit. on p. 49).

Google Developers (2018d). *Create and manage virtual devices*. URL: https://developer.android.com/studio/run/managing-avds (visited on 27/11/2018) (cit. on p. 56).

Google Developers (2018e). *Firebase Test Lab*. URL: https://firebase.google.com/docs/test-lab (visited on 07/01/2019) (cit. on p. 20).

Google Developers (2018f). *Fundamentals of Testing*. URL: https://developer.android.com/training/testing/fundamentals (visited on 27/11/2018) (cit. on pp. 50, 64).

Gousios, Georgios, Martin Pinzger and Arie van Deursen (2014). 'An Exploratory Study of the Pull-based Software Development Model'. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM, pp. 345–355 (cit. on p. 9).

Herrera-Izquierdo, Luis and Marc Grob (Nov. 2017). 'A performance evaluation between Docker container and Virtual Machines in cloud computing architectures'. In: *Maskana* 8, pp. 127–133. URL: https://publicaciones.ucuenca.edu.ec/ojs/index.php/maskana/article/view/1457 (cit. on p. 57).

Humble, Jez and David Farley (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley. ISBN: 9780321601919 (cit. on p. 14).

Hunt, Andrew and David Thomas (2000). *The Pragmatic Programmer: From Journey to Master*. Addison-Wesley. ISBN: 020161622X (cit. on pp. 10, 33).

Jenkins Team (2018). *Security Advisories*. URL: https://jenkins.io/security/advisories (visited on 07/01/2019) (cit. on p. 18).

Joshi, Sachin (2014). *15+ Useful Robotium Code Snippets for Android Test Automation*. URL: https://www.javacodegeeks.com/2014/06/15-useful-robotium-code-snippets-for-android-test-automation.html (visited on 07/01/2019) (cit. on p. 48).

Kerrisk, Michael and Eric W. Biederman (2018). *namespaces – Linux Programmer's Manual*. URL: http://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 07/01/2019) (cit. on p. 58).

Kim, Gene et al. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution. ISBN: 9781942788003 (cit. on pp. 22, 27, 34).

Kocher, Paul et al. (2019). 'Spectre Attacks: Exploiting Speculative Execution'. In: *40th IEEE Symposium on Security and Privacy (S&P'19)* (cit. on p. 77).

Kotter, John P. and Dan S. Cohen (2012). *The Heart of Change: Real-life Stories of How People Change Their Organizations*. Harvard Business Review Press. ISBN: 9781422187333 (cit. on p. 19).

Laster, Brent (2018). *Jenkins 2 Up & Running*. O'Reilly. ISBN: 9781491979594 (cit. on pp. 30, 34, 35, 37, 38, 60, 66).

Leffingwell, Dean (2007). *Scaling Software Agility: Best Practices for Large Enterprises*. Pearson Education. ISBN: 9780321458193 (cit. on p. 1).

Lesser, Joachim Andreas (2018). 'NFC extension for Catrobat'. MA thesis. Graz University of Technology (cit. on p. 41).

Lipp, Moritz et al. (2018). 'Meltdown: Reading Kernel Memory from User Space'. In: *27th USENIX Security Symposium (USENIX Security 18)* (cit. on p. 77).

Luhana, Kirshan Kumar, Christian Schindler and Wolfgang Slany (May 2018). 'Streamlining mobile app deployment with Jenkins and Fastlane in the case of Catrobat's pocket code'. In: *2018 IEEE International Conference on Innovative Research and Development (ICIRD)*. DOI: 10.1109/ICIRD.2018.8376296 (cit. on pp. 16, 43).

Luo, Qingzhou et al. (2014). 'An Empirical Analysis of Flaky Tests'. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, pp. 643–653 (cit. on pp. 44, 45, 47).

Martin, Robert C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. Prentice Hall. ISBN: 978-0-13-235088-4 (cit. on p. 26).

Mell, Peter and Timothy Grance (2011). 'The NIST Definition of Cloud Computing'. In: (cit. on p. 10).

Merkel, Dirk (Mar. 2014). 'Docker: Lightweight Linux Containers for Consistent Development and Deployment'. In: *Linux Journal* 2014.239. ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2600239.2600241 (cit. on pp. 57, 58).

Morieux, Yves (2011). 'Smart rules: Six Ways to Get People to Solve Problems Without You'. In: *Harvard Business Review* 89.9, pp. 78–86 (cit. on p. 22).

Müller, Matthias, Christian Schindler and Wolfgang Slany (Jan. 2019). 'Engaging Students in Open Source: Establishing FOSS Development at a University'. In: *Proceedings of the 52nd Annual Hawaii International Conference on System Sciences*, pp. 7721–7730 (cit. on p. 15).

Nizet, Jean-Baptiste et al. (2018). *The Java Plugin*. URL: https://docs.gradle.org/current/userguide/java_plugin.html (visited on 20/01/2019) (cit. on p. 7).

Orr, Christopher (2017). *Android Emulator Plugin*. URL: https://plugins.jenkins.io/android-emulator (visited on 07/01/2019) (cit. on p. 65).

Preeth, E. N. et al. (Nov. 2015). 'Evaluation of Docker containers based on hardware utilization'. In: *2015 International Conference on Control Communication Computing India (ICCC)*, pp. 697–700. DOI: 10.1109/ICCC.2015.7432984 (cit. on p. 58).

Prichard, Ryan (2018). *Docker Runner*. URL: https://confluence.atlassian.com/bamboo/docker-runner-946020207.html (cit. on p. 58).

Prichard, Ryan and Dan Albert (2018). *Changelog r18*. URL: https://github.com/android-ndk/ndk/wiki/Changelog-r18 (visited on 21/12/2018) (cit. on pp. 56, 74).

Reisenberger, David (2014). 'Continuous Integration in an Android Open Source Project'. MA thesis. Graz University of Technology (cit. on pp. 2, 15).

Skiena, Steven S. (2008). *The Algorithm Design Manual*. 2nd ed. Springer. ISBN: 978-1-84800-069-8 (cit. on p. 76).

Slany, Wolfgang (Sept. 2012). 'A mobile visual programming system for Android smartphones and tablets'. In: *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 265–266. DOI: 10.1109/VLHCC.2012.6344546 (cit. on p. 15).

Smith, Donald (2017). *Java SE support for Docker CPU and memory limits*. URL: https://blogs.oracle.com/java-platform-group/java-se-support-for-docker-cpu-and-memory-limits (visited on 03/02/2019) (cit. on p. 77).

Spilker, Daniel (2016). *Tutorial Using the Jenkins Job DSL*. URL: https://github.com/jenkinsci/job-dsl-plugin/wiki/Tutorial---Using-the-Jenkins-Job-DSL (visited on 07/01/2019) (cit. on pp. 32, 33).

Stackify (2017). *Top Continuous Integration Tools: 51 Tools to Streamline Your Development Process, Boost Quality, and Enhance Accuracy*. URL: https://stackify.com/top-continuous-integration-tools (visited on 21/01/2019) (cit. on p. 10).

Thorve, Swapna, Chandani Sreshtha and Na Meng (2018). 'An Empirical Study of Flaky Tests in Android Apps'. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, pp. 534–538 (cit. on pp. 45–47).

Vocke, Ham (2018). *The Practical Test Pyramid*. URL: https://martinfowler.com/articles/practical-test-pyramid.html (cit. on p. 49).

Walsh, Daniel J., ed. (2014). *Are Docker containers really secure?* URL: https://opensource.com/business/14/7/docker-security-selinux (visited on 31/01/2019) (cit. on p. 57).

Wendelin, Eric et al. (2018). *Writing Custom Plugins*. URL: https://docs.gradle.org/current/userguide/custom_plugins.html (visited on 07/01/2019) (cit. on p. 69).

Wilkosz, Ewelina, ed. (2017). *JEP-201: Jenkins Configuration as Code*. URL: https://github.com/jenkinsci/jep/tree/master/jep/201 (visited on 29/01/2019) (cit. on p. 28).

Zawadzki, Piotr (2018). *Android Test Orchestrator unmasked*. URL: https://medium.com/stepstone-tech/android-test-orchestrator-unmasked-83b8879928fa (visited on 07/01/2019) (cit. on p. 50).