

**M A S A R Y K
U N I V E R S I T Y**

FACULTY OF INFORMATICS

**Optimizing React Native application
deployment processes with
automation**

Bachelor's Thesis

JAN GLOS

Brno, Spring 2024

**M A S A R Y K
U N I V E R S I T Y**

FACULTY OF INFORMATICS

**Optimizing React Native application
deployment processes with
automation**

Bachelor's Thesis

JAN GLOS

Advisor: prof. RNDr. Tomáš Pitner, Ph.D.

Brno, Spring 2024



Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

While preparing this thesis, I used Grammarly for grammar check and ChatGPT to improve my writing style and generate documentation. I declare that I used these tools in accordance with the principles of academic integrity. I checked the content and took full responsibility for it.

Jan Glos

Advisor: prof. RNDr. Tomáš Pitner, Ph.D.

Acknowledgements

I would like to express my gratitude to prof. RNDr. Tomáš Pitner, Ph.D. for his time and valuable advice. My thanks also go to Mgr. Jan Ondruch from InQool and all the people who were by my side when writing this thesis.

Abstract

The aim of this thesis is to analyze tools for deploying React Native applications, such as Expo or Fastlane, and to use these tools to implement a CI/CD pipeline that can be integrated into both new and existing projects. Addressing the challenges of lengthy pipeline configuration, shell and Python scripts associated with the pipeline are designed to automate and assist in managing the configuration process. The result of this thesis is an extendable architecture that automates the deployment processes using a CI/CD pipeline equipped with tools that enhance its configuration routine.

Keywords

React Native, deployment, automation, CI/CD, DevOps, Android, iOS, Expo, Fastlane, InQool

Contents

Introduction	1
1 Company context	2
2 React & React Native	3
2.1 React	3
2.1.1 JSX	3
2.1.2 Virtual DOM	4
2.2 React Native	5
2.2.1 Architecture of mobile React Native applications	6
2.2.2 New architecture	6
3 Continuous Integration and Continuous Delivery	8
3.1 Continuous Integration	8
3.2 Continuous Delivery	8
3.3 Continuous Deployment	9
3.4 CI/CD pipeline	9
4 Existing tools for React Native deployment	11
4.1 Fastlane	11
4.1.1 Plugins	11
4.2 Expo	12
4.3 Jenkins	12
4.4 GitHub Actions	13
4.5 GitLab CI/CD	14
5 Analysis in company context	17
5.1 Current deployment processes in the company	17
5.1.1 Deployment of Android applications	18
5.1.2 Deployment of iOS applications	19
5.1.3 Reasons for automation	20
5.2 Utilizing automation	21
5.2.1 Potential areas for automation implementation .	21
5.3 Choosing the right tools	23
6 Implementation	25

6.1	Accounts setup	25
6.1.1	Expo	25
6.1.2	Google service account	26
6.2	Building of the applications	27
6.2.1	Local builds	27
6.2.2	Builds on EAS servers	27
6.3	GitLab configuration	28
6.3.1	GitLab runners	28
6.3.2	CI/CD pipeline	28
6.4	Signing and credentials	32
6.5	Versioning	32
6.6	Integration and automation scripts	34
6.6.1	Architecture	34
6.6.2	Pipeline configuration	35
6.7	Extendability	38
6.8	Proof of concept	38
6.8.1	Issues	39
	Conclusion	41
	Bibliography	42
	A Implementation	47

List of Tables

- 4.1 Comparison between Jenkins and GitHub Actions [35] . . 14
- 4.2 Comparison between Jenkins and GitLab CI/CD [37] . . 16

List of Figures

2.1	React Native’s architecture [18]	7
4.1	Fastlane action results in Jenkins [32]	13
6.1	Expo dashboard	26

Introduction

The deployment phase is an integral part of every mobile application's lifecycle. This includes the bundling of the source code into executable applications and releasing them to their consumers. For companies, the ability to quickly and reliably deploy applications to their customers is crucial.

This thesis is conducted in collaboration with InQool, a.s. – a software company focused on developing robust information systems and applications. The company's mobile application deployment workflows are mostly manual, which slows down the process and introduces a higher risk of human error.

The aim of this thesis is to analyze the available tools that can be used to automate deployments, pinpoint the processes to automate, and design a CI/CD pipeline tailored to the company's needs.

In the first chapter, I will describe the company's background and processes. The second chapter will focus on the tools used to develop mobile applications and introduce React Native. The third chapter will be dedicated to describing continuous integration and continuous delivery. The fourth and fifth chapters will analyze the existing tools for the deployment of React Native applications and put them into the company's context. The sixth and final chapter will describe the implementation of the CI/CD pipeline and other automation utilities.

1 Company context

InQool, as a company, specializes in delivering custom applications and information systems to various businesses and government agencies from the Czech Republic and Slovakia. These systems include complex web applications and powerful applications for mobile devices that are tailored to meet specific customer requirements.

Mobile applications are written in React Native, emphasizing a unified development process across the major mobile platforms. However, the build and deployment processes are not yet fully standardized across all projects. Some of the applications the company develops are distributed by the customers themselves, whereas the deployment and delivery process of other applications is completely directed by developer teams.

Until now, the deployment processes were completely manual. Hence, all the application bundles needed to be manually built and then uploaded to either the Play Store for Android applications or the App Store for iOS applications. Small partial automation utilities, such as custom building scripts, were used. However, the scripts only handled the bare minimum, as our deployment automation requirements were of a bigger scale.

2 React & React Native

To fully grasp the essence of React Native [1], it is essential first to get familiar with React [2]. This chapter, therefore, provides a comprehensive overview of the core principles both technologies build on, introducing how React Native extends and applies these concepts in the field of mobile application development.

2.1 React

React is a JavaScript **library** used to build *reactive* user interfaces on the web. There are key differences between frameworks and libraries that determine how developers ought to use them.

Frameworks are full-fledged tools that enable users to build complete applications without using additional external libraries. To word it differently, frameworks usually have all the paramount resources already bundled within them. Hence, tasks such as routing, state management, or data fetching are handled by the framework itself. Consequently, companies often utilize frameworks to build large-scale applications that make use of all the functionality contained in them [3]. Some of the most used JavaScript frameworks on the web are, e.g., Vue.js [4] and Angular [5].

Libraries, on the other hand, offer only limited functionality. React makes it easy to build a scalable dynamic user interface (*UI*), but does not handle other necessary aspects of the application, such as routing, form validation, etc. Thus, developers either come up with their own solutions to these problems or install other third-party React-specific libraries, that tackle these issues [3]. Some may view this as additional overhead since all the tools have to be selected and installed manually by the developers. On the contrary, this gives the developers complete control over the tools they use and prevents bloating applications with unused functionalities.

2.1.1 JSX

JSX is an ECMAScript [6] extension used extensively in React. It provides an XML (*Extensible Markup Language*) [7] interface to build UI

blocks. It serves as a *syntactic sugar*, meaning that it does not add new functionality that ECMAScript does not already contain. Subsequently, all of the JSX is at some point transpiled into ECMAScript [8].

Transpilation is the process of translating a program written in one (often high-level) language into an equivalent program in another (usually high-level) language [9]. That is exactly what is happening behind the scenes with React JSX code. Transpilation is most often performed automatically. An example of such a transpiler is Babel [10].

Below is an example of code written with JSX:

```
<div className="container">
  <div className="child" />
</div>
```

And here is the equivalent code in ECMAScript:

```
React.createElement(
  "div",
  { className: "container" },
  React.createElement(
    "div",
    { className: "child" }
  )
);
```

As you can see above, JSX offers a concise and readable way of expressing UI with a syntax that is rather similar to the syntax of HTML [11], which tends to be familiar to many developers. Moreover, it provides a more intuitive and visual way to structure the layout of a UI. The benefits of JSX become more apparent as the complexity of the UI increases.

2.1.2 Virtual DOM

DOM (*Document Object Model*) is a tree-like representation of web documents. The provided interface allows for the programmatic modification of the nodes and objects that make up the document's structure. As a result, with the help of a scripting language such as JavaScript, programmers can navigate the DOM, manipulate, and interact with the elements of a web page in real-time [12].

Virtual DOM is an *abstract* DOM that is used by React. React does not always manipulate the DOM of a web document directly; it creates and modifies a virtual version of this DOM. Each object present in the DOM is bound to an object in the virtual DOM [13].

Given a React web application whose state changes rather frequently, direct manipulation of the DOM would not be scalable and performant, since DOM manipulation is an expensive operation [14]. Modifications to the virtual DOM, on the other hand, are fairly inexpensive operations since a virtual DOM is just a lightweight JavaScript object. This is the reason why React chose to use virtual DOM to manage the real DOM.

All the changes that would otherwise have to be made to the DOM directly are made to the corresponding objects in the virtual DOM. Consequently, the virtual DOM is then effectively compared to the previous version of the virtual DOM, and if any updates to the DOM are necessary, React performs them while only updating the relevant parts of the DOM tree [13].

2.2 React Native

React Native's motto "Learn once, write anywhere" [15] exceptionally sums up the motivation behind it. It is an open-source multi-platform JavaScript *framework* that builds on the foundations laid by React. Specifically, it brings all the fundamental React concepts and its mind-set into the world of native application development.

It is mostly used to develop mobile applications both for iOS and Android. Additionally, even web pages, TV apps, and many others can be developed with React Native. This means that one codebase can support virtually all the commonly used platforms and even more. React Native allows for the creation of smooth and responsive user experiences that closely mimic those of native applications [16].

Both React and React Native use a component-based structure, where components are the atoms of UI. The components manage their state and should be encapsulated to achieve atomicity. The basic building blocks of UI in React Native are Views. Views are rather similar to div elements one might know from web development. They

are rectangular containers that can contain any content (text, images, videos) or more containers.

2.2.1 Architecture of mobile React Native applications

Due to the nature of this thesis, it is essential to mention how React Native mobile applications' architecture works internally.

Since React Native is used to develop cross-platform mobile applications, the process of running the applications on devices is somewhat unique compared to traditional native mobile apps. A React Native application consists of two main layers: the JavaScript layer and the native code layer. The simplified process of writing and delivering an application to a mobile device goes as follows:

1. A developer writes code in JavaScript/TypeScript.
2. This code is bundled by a JavaScript bundler (such as Metro [17]).
3. The native code in Java, Objective-C, or Swift gets compiled; it is stored and run separately from JavaScript code.
4. JavaScript code is run on a JavaScript virtual machine and interpreted at runtime. JavaScript **does not** get translated into native programming languages like Java, Objective-C, or Swift. Both native code and JavaScript code are run on a separate thread. Communication between the native and JavaScript threads happens via a bridge. More about bridges in subsection 2.2.2.
5. An executable React Native application is generated [18].

2.2.2 New architecture

The bridge is the medium of communication between the JavaScript layer and the native layer. It is used to translate JavaScript into something that can be easily interpreted by the native languages – JSON (*JavaScript Object Notation*). Subsequently, the bridge is used to handle the serialization of data into JSON so that both layers can communicate with each other in an effective manner.

However, this approach has some drawbacks and requires additional computation time, making React Native perform worse than regular native applications [19]. The process involves converting JavaScript objects and function calls into a format that is understandable by the

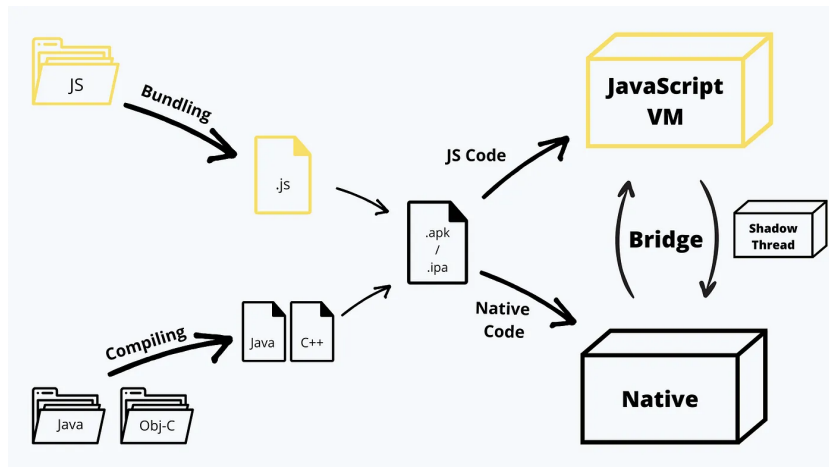


Figure 2.1: React Native's architecture [18]

native layer – JSON. This serialization process is computationally intensive since it requires parsing and converting the structure and data of the JavaScript objects into a string-based JSON format. After that, it is necessary to transfer the serialized data to the native thread. Once the native layer has finished processing the request, the result must be converted back into a format that the JavaScript layer can understand, requiring another round of serialization and a round trip of the data back to the JavaScript layer.

The React Native team wanted to tackle the overhead brought by the use of bridges and speed up React Native applications in other aspects as well. Consequently, in version 0.68, the team introduced a so-called "new architecture" [20], which is supposed to make React Native more performant and responsive by introducing turbo native modules and a new renderer [21, 22]. It is worth noting that the new architecture is still in the development phase, and as such, it isn't yet a well-established concept. Furthermore, the complete details of this architecture are outside the scope of this thesis. Despite it being in the initial stage, mentioning the new architecture of React Native is worthwhile as it sheds light on the potential future enhancements and the evolving direction of the platform.

3 Continuous Integration and Continuous Delivery

This chapter provides an overview of CI (*continuous integration*) / CD (*continuous delivery*) and summarizes the motivation behind these processes.

3.1 Continuous Integration

Continuous integration is an automation process mainly designed to mitigate the amount of tedious tasks that developers have to perform. The main purposes of CI are to:

- maintain united coding conventions and formatting,
- statically analyze new code,
- build code,
- run tests.

When CI is put in place, each submission of new code (*usually through a merge/pull request to a git repository*) triggers an automated set of tasks that ensure the code is properly formatted, obeys coding conventions, has desired functionality, and is generally ready to be introduced into a repository.

As the amount of people working on a software project rises, the complexity of keeping the codebase stable, united, and in control increases. A well-designed CI pipeline should scale together with the size of the team, ensuring that integration issues are automatically detected and addressed adequately. This is where CI is essential since it provides a solid backbone for managing the increased complexity.

3.2 Continuous Delivery

Continuous delivery is an extension of continuous integration. Therefore, these two processes are often bound together and tend to be executed sequentially.

Continuous delivery expands on continuous integration by *delivering* the tested code into a repository. The key aspect of continuous delivery is that the code present in the repository is at **all times** ready

to be pushed to the production environment and to be delivered to customers [23].

3.3 Continuous Deployment

Continuous deployment is a term that is sometimes considered as a part of continuous delivery depending on the context [23].

Should continuous deployment be examined as a separate automation process, it can be defined as the final step in which the new changes get deployed to end users without any human interaction/approval. Subsequently, this practice allows rapid delivery of granular changes in small batches.

The frequency of the delivery is a major advantage since it allows blazing-fast bug fixes or new feature updates to be distributed to customers in a fast fashion.

From another perspective, the quality of the updates heavily depends on the thoroughness of the continuous integration process that precedes the deployment process itself. Thus, it is crucial to ensure that rigorous and effective integration processes are put in place.

3.4 CI/CD pipeline

CI/CD pipeline is a process that is constituted of the sequence of the processes mentioned above.

It begins with CI where the newly added (*or modified*) code can be formatted, statically analyzed, built, and tested. This process ensures that the code abides by a set coding style and verifies that the old functionality remains intact and that the newly added functionality works as expected.

Then comes the CD – the code that previously passed through the CI phase is pushed into a repository and is ready to be released into a production environment at any moment. The repository at this stage can further be tested to guarantee the correctness of the changes and validate the quality even beyond the automated tests that were a part of the CI process.

In summary, the CI/CD pipeline can offer a seamless way of automating one's integration and deployment workflows. Moreover, it

3. CONTINUOUS INTEGRATION AND CONTINUOUS DELIVERY

minimizes the risk of human error since human intervention is not needed for these processes to operate. By introducing thorough testing and strict processes, the code that passes through these processes tends to be more reliable and of higher quality. On the other hand, maintaining a well-designed pipeline requires a significant amount of effort. The tests must be well-written and up-to-date and should cover a remarkable part of the codebase. For a CI/CD pipeline to be effective, it is absolutely crucial for it to be meticulously planned, designed, implemented, and maintained.

4 Existing tools for React Native deployment

This chapter explores existing tools and practices used when deploying applications written in React Native.

4.1 Fastlane

Fastlane is an open-source automation toolset for automating and simplifying the process of building, testing, and deploying mobile applications, including those developed with React Native. It is a bundle of scripts written in Ruby [24] that relieves developers from the burden of manually building and deploying applications.

As its name suggests, Fastlane uses individual lanes that enable users to specify the actions that they want to take when executing said lane. A lane can be viewed as a collection of tasks that constitute a workflow. It is essentially a function, and actions are the code blocks. Below you can find an example of a lane from the official Fastlane documentation [25].

```
lane :beta do
  increment_build_number
  build_app
  upload_to_testflight
end
```

Actions can either be **built-in** or completely **custom**. Under some circumstances, these custom actions can be submitted to the Fastlane codebase via a pull request [26].

4.1.1 Plugins

Fastlane offers an extensive amount of plugins developed by the community. These plugins allow Fastlane users to extend the built-in capabilities further and share their actions with other developers. Developing a Fastlane plugin could serve as an alternative if the action fails to meet the prerequisites for direct integration into the Fastlane codebase. This approach provides a structured method for extending

Fastlane's functionality while maintaining adherence to its standards. Moreover, the process of creating a plugin is much easier and swifter.

4.2 Expo

Expo is another open-source framework that is capable of automating the deployment of mobile applications written with JavaScript and React. Moreover, it serves as a platform for developing these applications as well. Utilizing Expo's CLI (*Command Line Interface*), users can both develop and deploy their application to App Store¹, Google Play² and the web with one codebase.

EAS (*Expo Application Services*) is a cloud-based platform that offers additional services to Expo users, such as:

- application building,
- conveniently sharing preview builds with testers,
- handling of secret credentials,
- automating store submissions,
- automatic version management,
- OTA (*Over-the-Air*) updates [27].

EAS uses Fastlane for iOS under the hood thus extending and simplifying its use [28]. Expo excels at simplifying many platform-specific details and offers numerous pre-built ready-to-use tools, thereby considerably reducing development time.

On the other hand, the lack of control over the build process might be perceived as a limitation. Additionally, incorporating third-party libraries that use custom native code not included in the Expo SDK can introduce unneeded complexity and require additional configuration for projects managed by Expo [29].

4.3 Jenkins

Jenkins is a widely used open-source automation server powered by Java. It is extremely flexible and acts as an automation tool for a broad range of different codebases as it is platform-agnostic. Jenkins is exceptionally useful at automating code integration, such as building

1. App Store: <https://www.apple.com/app-store/>

2. Google Play: <https://play.google.com/>

and testing projects every time new code is pushed to a Git [30] repository. Subsequently, this allows for smooth and continuous delivery of smaller parts of new code into existing software projects [31].

Jenkins is an automation server. Therefore, it can be used in synergy with other automation tools, such as Fastlane. Fastlane itself allows the definition of the specific actions that constitute the piece of work consumers want to get done. Combining it with Jenkins, users can enrich Fastlane's capabilities, trigger certain lanes on events in their Git repositories, and overall manage the complete flow of actions in a reliable fashion [32].

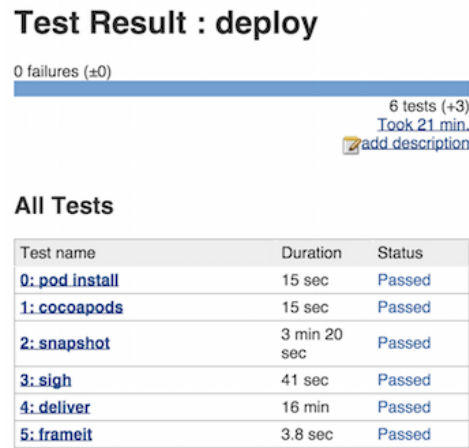


Figure 4.1: Fastlane action results in Jenkins [32]

4.4 GitHub Actions

GitHub Actions is a free open-source CI/CD platform used to automate building, testing, and deploying code. It is integrated into GitHub [33] repositories directly, hence, it is a rather popular tool among users in the GitHub community. The workflow that is executed when an action is run is defined in a YAML [34] file. This YAML file may resemble a lane within a Fastlane setup. Similar to a Fastlane lane, a GitHub Actions workflow specifies a sequence of tasks to be executed automatically upon certain triggers, such as pushing new code or creating a pull request. GitHub Actions may serve a similar

purpose as Jenkins. Below, you can find a table that underlines the key strengths and weaknesses of both tools to determine which tool to use for which purpose.

Table 4.1: Comparison between Jenkins and GitHub Actions [35]

Feature	Jenkins	GitHub Actions
Software model	Open source	Open source
Hosting	On-premise and cloud-based	On-premise and cloud-based
Supported OS	Linux, Windows, MacOS	Linux, Windows, MacOS
Ease of use and Setup	Medium	Easy to use
Installation	Required	Not required
Scalability	Highly scalable	Limited by GitHub's infrastructure
Control	Doesn't provide full control over CI/CD pipelines	Full control over CI/CD pipelines
Paid plan details	Free to use, requires a dedicated administrator	Both free and paid plans with varying features and usage limits

4.5 GitLab CI/CD

Similarly to GitHub Actions, GitLab CI/CD is a platform that automates development and deployment processes using CI/CD pipelines. Workflows are based on YAML files as well. Both platforms offer similar functionality, and the decision between them often comes down to personal preference. Teams or individuals who already utilize GitHub

or GitLab to manage their codebase may naturally lean towards the platform they are already familiar with. This familiarity stems from the seamless integration that both GitHub and GitLab provide with other tools and workflows.

Regarding security, GitLab CI/CD facilitates a secrets management system that is able to securely store private or sensitive data which is essential when executing a workflow. Besides that, CI/CD variables are also available to the developer to further configure and tailor the specified workflow to their specific needs [36]. In the context of mobile application deployment, this is where all the necessary credentials and API keys would be stored in order to access individual services such as Google Play, App Store, Google Drive, and others.

Table 4.2: Comparison between Jenkins and GitLab CI/CD [37]

Feature	Jenkins	GitLab CI/CD
Open-source or Commercial	Open-source	Open-source
Product Type	Self-hosted	Self-hosted/SaaS
Built-in CI/CD	Plugin-dependent	Built-in
Unique Feature	Extensive plugins	AutoDevOps, Unified CI & code management
Support / SLA	None	Available
Setup and Installation	Easy	Easy
Self-hosted Option	Only self-hosted	Available
Build Pipelines	Custom via DSL	Supported
App Performance Monitoring	Not available	Performance metrics provided
Ecosystem	1000+ plugins	Extensive
Comprehensive API	Yes	Yes
Language Support (JavaScript)	Yes	Yes
Integrations	Many (e.g., Slack, GitHub)	Numerous, incl. GitHub, Kubernetes
CI/CD Deployment Dashboard	Partial support	Customizable per user
API	REST	REST & GraphQL
Code Quality	Via plugins (e.g., Sonarqube)	Built-in

5 Analysis in company context

To adequately build new automation processes in a company, thorough research of the company's needs must be conducted. Subsequently, it is essential to identify areas where automation can improve efficiency and where it may cause bottlenecks due to undesirable lengthy configuration.

After identifying areas where automation can be beneficial, the next step involves researching and selecting technologies that fit the company's needs and already established processes/infrastructure best. Crucial factors to consider when choosing automation technologies are:

- their cost – including both financial and resource costs,
- scalability – so that the tools can handle increasing amounts of workload,
- ease of integration, and others.

The next step involves the actual implementation of new automation solutions. This might involve configuring some of the automation tools mentioned in chapter 4, creating new scripts or workflows, and integrating them into a CI/CD pipeline.

The goal of the automation processes is to boost productivity and mitigate human error. After the successful integration of new automation processes, it is paramount to continuously monitor their effectiveness and the benefits they bring to the company and individual developers. To ensure that these processes are effective and up-to-date, they ought to be kept well-maintained and expanded over time if need be.

5.1 Current deployment processes in the company

As a member of the React Native development team, my insights and descriptions relate solely to my team's deployment processes.

In the company's context, mobile apps developed in React Native are often developed with both iOS and Android in mind, leveraging the benefits of React Native being a multi-platform library. Some of our applications, however, are developed solely for one of the platforms. Moreover, the distribution point is not always Google Play or App

Store. Depending on the nature of the project and its intended user base, we also utilize alternative distribution methods such as Samsung KNOX, Google Drive, etc. In summary, there is not a one-size-fits-all solution when it comes to our deployment processes.

Generally, the building of our applications is done mostly manually, with some trivial automation in place.

5.1.1 Deployment of Android applications

The process of building and deploying Android applications within the company is primarily manual, with some steps incorporating minimal automation. The following outlines our current workflow:

1. Versioning

- The build number and version are incremented manually.
- These changes are made directly in the `build.gradle` file.

2. Environment configuration

- Environment variables are manually adjusted for the build process.
- The developer has to choose the appropriate environment for the current build.

3. Building the application

- The developer must navigate to the `android` folder and execute a built-in Gradle build script `assembleRelease` to generate an Android Package Kit file (*APK – format used to install Android applications*).
- When generating an Android App Bundle (*AAB – a format used when uploading applications to Google Play*) the developer follows similar steps to generating an APK but uses the `bundleRelease` built-in Gradle script.

4. Signing the application

- Mobile applications require signing to ensure their authenticity and integrity. Overall, it helps keep users safe from attackers. It guarantees that the application source is uniquely

identified and that a third party or an attacker did not alter the APK or AAB.

- My company uses Google's Play App Signing utility, which handles the signing for us. However, signing every application bundle with an upload key is essential before uploading it to Google Play [38].
- We generate the upload key manually or via Android Studio¹.
- This upload key is stored securely but must be manually retrieved and configured for each project.

5. Distribution

- The APK or bundle is manually distributed to the intended platforms or users. Depending on the project's requirements, this can include uploading it to Google Play, Samsung KNOX, Google Drive, etc.

After an application has been submitted to Google Play, it is necessary to manually request a review from Google so that the application can be distributed to our end users.

5.1.2 Deployment of iOS applications

The deployment process for iOS applications is similar to Android, with some manual steps involved. On the other hand, the iOS ecosystem has unique requirements that differ slightly from those of Android. Generally, our workflow includes the following steps:

1. Versioning

- Adjusting the version is a manual process.
- We use Xcode² to edit the corresponding settings of the application build.
- Apple handles incrementing the build number automatically with each new build.

2. Environment configuration

1. an Android integrated development environment (IDE)
2. Xcode - Apple IDE: <https://developer.apple.com/xcode/>

- Environment variables are manually adjusted for the build process.
- The developer has to choose the appropriate environment for the current build.

3. Building the application

- The build process is usually initiated and executed through Xcode.
- The developer may choose to upload the application bundle to TestFlight³ to distribute it to testers.

4. Signing the application

- Signing is handled by Apple completely.
- There is no need to generate an upload key for submission.

5. Distribution

- We distribute iOS applications through TestFlight and App Store Connect⁴.

Similarly to our workflow for Android applications, the newly uploaded version must be manually submitted to Apple for review. Only after a successful review is the application distributed to users.

5.1.3 Reasons for automation

Human error

Based on the deployment workflows described above, it is evident that there is a large possibility of a human error occurring somewhere during the process. Some of the most prevalent human defects that occur regularly are version mismatches, incorrectly set environment variables, and distribution attempts of non-signed application bundles to Google Play.

Furthermore, these oversights may result in major issues with the distribution of the company's applications. For instance, failing to

3. A platform for testing and submitting iOS applications: <https://developer.apple.com/testflight/>

4. Platform to manage iOS applications on the App Store: <https://developer.apple.com/help/app-store-connect/>

configure environment variables correctly for production could unintentionally grant users access to test or development environments. This would prevent transactions from processing correctly, leading to outages and, ultimately, financial losses on behalf of our customers. These incidents may irreversibly negatively influence the company's or our customer's reputation.

Human resources

Each new build requires a significant amount of time dedicated to ensure everything is set up correctly. Consequently, it leads to wasting human resources on tasks that could potentially be at least partially automated.

Scalability

As the number of developed and maintained projects increases, scalability becomes an essential concern. Manual deployment processes that are easy to manage for a small number of projects become unreasonable as the company scales up. This is indeed the case in my company, as both the amount and the complexity of our projects are constantly increasing. This increase can quickly overgrow human resources, causing bottlenecks, delays, and even errors. Scaling up efficiently is nearly impossible without automation, as manual processes do not scale proportionately with the number of projects.

5.2 Utilizing automation

Given the detailed overview of our current deployment processes and the mentioned challenges, it is clear that introducing automation into our workflows can provide substantial benefits.

5.2.1 Potential areas for automation implementation

Versioning

Automation can be introduced into the process of versioning. However, versioning might not be completely automatable. That is because

the developer must specify the version that is being bumped – major, minor, or patch (assuming semantic versioning [39] is used). Nevertheless, it is possible to unify the process of bumping the version for both iOS and Android so that the developer is not required to adjust it in two different places.

Environment configuration

Environments can be distinguished by individual branches in a Git repository where our projects are stored. Each project would then have a few special branches all defining a different environment, such as test, development, and production. Hence, the environment will be automatically established based on the branch the code is a part of. This will completely mitigate manual manipulation of the environment configuration and ideally prevent accidental bugs.

Signing the application

The process of signing an Android application bundle can be automated by either:

- retrieving the signing key from a secure location automatically if it already exists,
- generating the signing key and uploading it securely for future use if it does not already exist.

Consequently, the signing process will be less prone to errors and fundamentally more secure, assuming the signing key is handled securely according to common modern security standards – e.g., not committed to a repository, stored securely, and not exposed to unauthorized entities...

Delivering the application

Methods to automatically upload new versions to a distribution point can be utilized to mitigate manual uploads. Many popular deployment tools compatible with React Native directly support uploading application bundles to Google Play and the App Store. As mentioned earlier, our needs go beyond the two stores as we need to be able to deliver to Google Drive and potentially other platforms. Consequently, a more

customized approach is needed to fulfill our needs. This indeed plays a vital role in choosing the right tool for the final implementation.

Testing

As of now, the company's applications are mostly manually tested. However, the company aims to integrate automatic tests in the near future. Therefore, it is essential to consider this foresight when choosing the right automation solution.

5.3 Choosing the right tools

After assessing the areas where automation is a sane and beneficial approach in the company context, the concrete tools that will be used are to be analyzed and chosen carefully to fit the company's needs and requirements. Overall, the selected tools ought to be versatile, scalable, and maintainable in the long run.

Fastlane is the best tool for executing scripts that automate the deployment process of React Native applications in this context. It is both simple and easy to use, yet also extensible and powerful since it supports completely custom actions written in Ruby. On top of that, the community around Ruby and Fastlane is large and active. Numerous basic actions tied to mobile application development are already implemented and bundled with Fastlane, ensuring a swift start.

Moreover, Fastlane is not only limited to React Native – it can be used with a mobile application written in any multiplatform language/library/framework. Therefore, should the company change its development stack, the workflows defined with Fastlane should still be valid and usable. Subsequently, Fastlane lanes can easily be executed in any CI/CD pipeline as they are run from the terminal.

Despite these benefits, setting up all the credentials needed to build and publish iOS and Android applications has proven to be a rather difficult task with Fastlane. Furthermore, utilizing the same (or slightly altered) CI/CD pipeline in many different projects would become harder as the process of setting up the pipeline would not be unified.

This is where Expo comes into play – specifically, the Expo EAS platform which supports building and delivering both iOS and Android applications. The key aspect of EAS is that it handles credentials in a centralized location in the cloud and is, therefore, usable for a large amount of projects. Nevertheless, it is beneficial to combine different tools in a pipeline to achieve desired goals – utilizing both EAS and Fastlane in synergy will bring the most flexible (in terms of customizability) and reliable build/deploy results. Expo’s services are preferred due to the company’s planned transition to Expo-powered projects, which brings substantial tools and convenience to React Native mobile application development.

With EAS utilized for the build process and Fastlane for other handling of the builds (such as uploading them to Google Drive), a versatile, flexible, and easy-to-maintain pipeline can be constructed. Because of the fact that the company uses an instance of GitLab for versioning and collaborative work, GitLab CI/CD is the best pipeline tool to use. Moreover, it supports storing secret environment variables and integrates well with the company’s Git workflow overall.

6 Implementation

This part of the thesis describes the specific implementation of a React Native CI/CD pipeline for automatic builds and deployments. The setup involves setting up necessary accounts and configuring a GitLab repository with GitLab runners and the pipeline itself. The next step is to handle sensitive credentials such as API tokens, private keys, or passwords. Furthermore, the chapter covers new automation scripts that can be used to share common pipeline logic between different projects and make the setting up of a pipeline more feasible. The challenges that arose are mentioned and described at the end of the chapter.

6.1 Accounts setup

Since the implementation uses services of third parties (*such as Expo or Google*), it is essential to configure accounts for use in the automation solution appropriately. This involves integration with EAS and Google Play Developer APIs. Therefore, this section will cover the creation and configuration of both of the accounts.

6.1.1 Expo

An Expo account is needed to utilize Expo's services, such as EAS. I configured one central account that is used for all of the company's React Native projects with their corresponding credentials.

Expo offers an easy-to-read dashboard that summarizes various projects' build and deploy actions. Each of the projects can be viewed and modified separately. Subsequently, all projects are managed and manageable in one place, making it a really powerful solution for teams where the number of projects is rather large.

Moreover, the dashboard offers the possibility to download individual builds. Consequently, a manual submission to the stores is also possible through the Expo dashboard.

Expo offers a free plan that limits the number of builds per month to 30. Nevertheless, this is a sufficient amount that should not limit the company's workflow. A paid Expo plan might be considered in

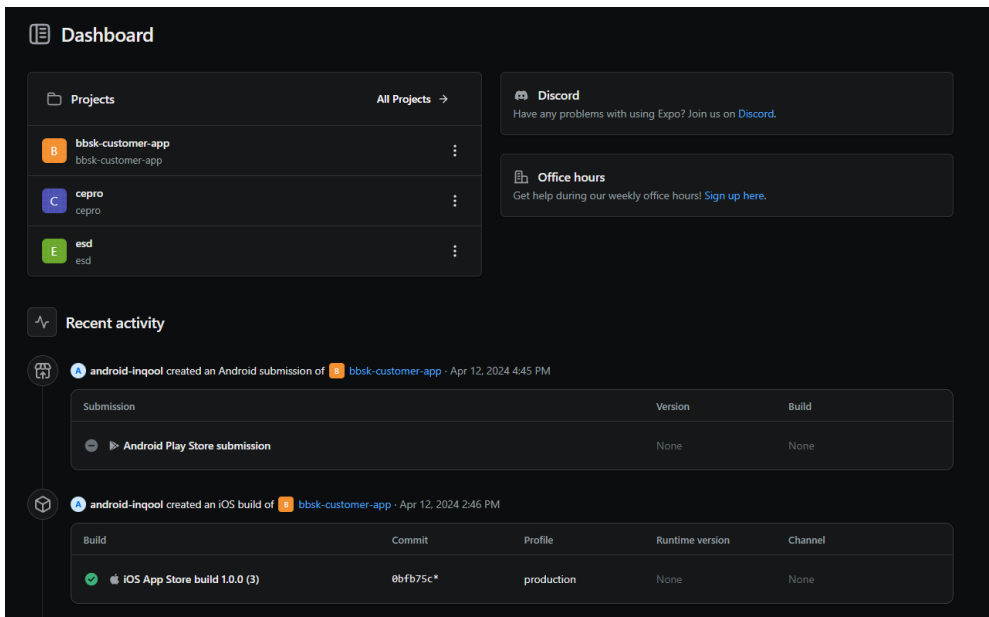


Figure 6.1: Expo dashboard

the future, should the company benefit from it. As of now, the free plan is completely abundant as local builds (*which are not limited*) are still a viable option if the limit is reached.

6.1.2 Google service account

A Google service account is needed to upload builds to the Google Play store. Moreover, enabling a Google service account also makes uploading new builds to Google Drive possible. I created a new Google Cloud project (with enabled Google Drive API) and configured a service account according to Google’s developer instructions [40].

The newly generated public/private key pair is stored in a secure location that is accessible by Expo and developers, who might utilize this service account to upload builds via the Google Drive API.

The service account’s access to resources is granted on an individual basis, where the resource has to be specifically shared with the service account. Thus, resources are easily shareable, and only the resources that need to be shared are actually shared. Granting

permissions to publish and upload new versions of applications to the Play Store involves a similar process, where the service account is specifically invited to the project and granted said permissions.

6.2 Building of the applications

EAS is utilized to build the applications for a unified process for both platforms. Different build profiles can be set up and enabled in the `eas.json` configuration file to customize the build properties [41]. In order to keep the configuration generic, only a basic set of build profiles is utilized by default. Subsequently, the `eas.json` file may be further edited and adjusted to suit individual projects. There are two types of builds that the solution utilizes, namely:

- local builds,
- builds on EAS servers.

6.2.1 Local builds

The solution uses local builds to build for non-production environments. Since these builds run on the company's infrastructure, they are not limited by waiting in build queues or running out of the number of builds per billing period. Hence, local builds are great for verifying whether the application is even buildable – many bugs and errors might be discovered this way. On top of that, all local builds can also be directly submitted to the stores by utilizing the command `eas submit`.

A significant downside to local builds is that the environment for the builds must be set up on the machine where the build runs. This involves installing all the EAS build dependencies listed in the official documentation [42]. Consequently, to build iOS applications, the build must be executed on a machine running macOS. Thus, developers without access to a macOS device may not be able to perform the build process of iOS applications locally.

6.2.2 Builds on EAS servers

Builds in the cloud executed on EAS servers are preferred for production as the build environment is standardized. Subsequently, all

production builds performed on EAS servers are visible and manageable through Expo's dashboard, bringing more flexibility and clarity.

6.3 GitLab configuration

GitLab plays a crucial role in the proposed solution as it serves as the CI/CD service where essential workflows are executed. Each GitLab repository supports CI/CD by default and offers extensive options to tailor it to fit users' needs. It is necessary to configure GitLab runners and the pipeline configuration itself to implement a working CI/CD pipeline.

6.3.1 GitLab runners

When a CI/CD job is triggered a GitLab runner is necessary to execute it. It is a piece of software that runs on any machine. There are shared GitLab runners that may be used across many different projects and also specific runners that are usually unique to a single project.

For Android jobs in the pipeline, no specific operating system is necessary. I utilized one of our company's shared runners which runs a Linux environment for these jobs.

On the other hand, a specific runner with a macOS operating system was needed to build applications for iOS. I decided to install and configure a GitLab runner on a personal macOS device to handle tasks associated with the building of iOS applications [43]. I then proceeded by assigning this runner to repositories where iOS builds need to be performed.

6.3.2 CI/CD pipeline

After configuring runners to pick up the pipeline jobs, these jobs must be configured in the `.gitlab-ci.yml` file that GitLab uses for CI/CD pipeline definition.

The `GIT_STRATEGY` variable is set to `clone` as it always ensures a completely clean repository.

In the context of established company processes, I decided to define four basic stages that can be further built upon, namely:

- linting,

- version code and build number bump,
- build,
- deploy.

Linting

In the linting stage, the code is statically analyzed with ESLint [44] which helps discover overlooked bugs in the earliest stages. This stage can be extended in the future by including automatic code formatting, for instance.

Version code and build number bump

This stage is crucial to ensuring the correct management of Android applications' version codes and iOS applications' build numbers. Its corresponding job uses a custom Fastlane lane that increments both of these properties. Furthermore, these changes are then pushed to the current branch and to the main branch to synchronize these changes. Without this stage succeeding, the pipeline cannot proceed.

Build

The build stage handles the transformation of the code into application bundles that can either be installed on mobile devices directly or uploaded to the corresponding stores for distribution.

The build stage consists of four different jobs, where two jobs handle the building process of the iOS version of the application, and the two remaining jobs handle building for Android. For both platforms, jobs are further divided into one job for the test environment and one job for the production environment.

Production builds for both platforms utilize the EAS command line interface's command:

```
npx eas-cli@latest build --platform [android |  
ios] --profile production --non-interactive
```

This command uploads the application code to EAS servers and initializes the build on EAS's cloud infrastructure. The `-non-interactive` option is essential for the command to be executed in a CI/CD pipeline

environment. Production jobs are triggered on each push to the *production* branch.

Builds for the test environment utilize EAS build with the `-local` option. The command then looks like this:

```
npx eas-cli@latest build --platform [android |  
ios] --profile test --non-interactive --local
```

For these local builds, the environment of the runner needs to be set up as mentioned in subsection 6.2.1. Test jobs are triggered on each push to the *test* branch.

Deploy

Once the builds are successfully generated, the pipeline proceeds to the deployment stage. This is where the application bundle is submitted to a distribution point. The same number of jobs as for the build stage is utilized – two for deploying Android applications and two for deploying iOS applications. All the deployment jobs have their corresponding build counterparts set as a dependency, since no deployment shall happen while the build stage has not finished or should the build stage fail entirely.

For production deployment, the following command is executed:

```
npx eas-cli@latest submit -p [android | ios] --  
latest --non-interactive --no-wait
```

EAS then picks the most recent build and attempts to submit it to the corresponding store. All of the necessary credentials should be stored in EAS at this point in time.

Test deployment for iOS tends to differ from test deployment for Android in the company. When it comes to iOS applications, we usually upload them to App Store Connect as we would for production applications. Therefore, the process is almost the same as for the production build. The main difference is that the locally built *.ipa* file is submitted instead of the most recent build on EAS servers. On the other hand, test versions of Android applications are more likely to be distributed as APK files through Google Drive. Therefore, Fastlane is used to upload the *APK* to Google Drive through the configured service account. The Google Drive location to upload the *APK* can be

configured by specifying the `GOOGLE_DRIVE_FOLDER_ID` environment variable in the pipeline. Moreover, the `GOOGLE_SERVICE_KEY` variable must also be set to provide access to the service account that will be used to perform the upload.

```
if [[ -n "$GOOGLE_DRIVE_FOLDER_ID" && -n "$GOOGLE_SERVICE_KEY" ]]; then
    echo "$GOOGLE_SERVICE_KEY" >
        google_service_key.json
    bundle install

    apk=$(find . -name "*.apk" -print0 | xargs -r
        -0 ls -1 -t | head -1)

    bundle exec fastlane upload_apk folder_id:"
        $GOOGLE_DRIVE_FOLDER_ID" apk_path:"$apk"
fi
```

Environment

The environment of the application is fully determined by the branch the code was pushed to. The pipeline itself sets the environment variable `EXPO_PUBLIC_ENVIRONMENT` to the current environment – either *test* or *production* (or any other possible environment). It does so by adjusting the environment variable in a `.env` file.

```
set_env() {
    if grep -q "EXPO_PUBLIC_ENVIRONMENT=" .env;
    then
        sed -i "s/^EXPO_PUBLIC_ENVIRONMENT=.* /
            EXPO_PUBLIC_ENVIRONMENT=$1/" .env
    else
        echo "EXPO_PUBLIC_ENVIRONMENT=$1" >> .env
    fi
}
```

The application is therefore aware of the environment it was built in and will appropriately configure everything that depends on the current environment – API URLs, debugging info... No manual manipulation of either the `.env` file or the shell environment is required.

6.4 Signing and credentials

EAS handles the application signing completely. This means that the developer is seldom required to manage any signatures or credentials on their own.

For the credentials to be managed by EAS, it is essential that they are uploaded there first. Consequently, every time an application is built and submitted for the first time, the corresponding *eas build* and *eas submit* commands should be executed manually. Expo then walks the developer through the process of creating new signing credentials for the current project. After this step, all of the builds and submissions can be triggered from the pipeline without the need for human intervention.

Should it be necessary to use existing credentials, Expo offers a way of supplying them to EAS. The path to the local credentials may be specified in a *credentials.json* file. These credentials are then used when building or submitting the application. By running *eas credentials*, the credentials specified in *credentials.json* can be uploaded to EAS. EAS will then store and manage them on its own [45].

Authenticated communication with EAS servers is secured with an access token generated in the Expo's dashboard in section *Access tokens*. This token enables robot users to perform authenticated actions on behalf of the account. This is crucial, as the pipeline environment is not interactive, and therefore, signing in to the Expo account cannot be handled by entering the account credentials on each login. The access token is stored inside the variable system of each git repository as `EXPO_TOKEN`. Expo uses this variable as default to look for a provided access token before attempting a manual login.

6.5 Versioning

An automated versioning process is put in place to unify the process of bumping versions for both iOS and Android projects of the same application. The pipeline *does not* automatically bump versions on its own, since it is impossible for it to know how to bump the version. Hence, the developer pushing new code to either the *test* or *production* branch is responsible for bumping the version.

I created a Fastlane automation that bumps the versions of both Android and iOS in unison. This is achieved by storing the version in one file that serves as the single source of truth. That is the `package.json` file. It is independent of the specific platform and the current versions set for the specific platforms. This is not ideal as having this version dictate the current version to both platforms is desirable. Henceforth, I needed to synchronize the versions with each other.

This is where Fastlane comes into play. I created a lane called `set_version` that first reads the contents of `package.json` (using a Fastlane plugin `load_json`) and extracts the current version. After that, the version of the native project is set to the previously read version from `package.json` with `increment_version_name` for Android and `increment_version_number` for iOS. Both of these actions accept a parameter that makes it possible to *set* the version rather than *increment* it.

The process described above only handles synchronizing the native versions with `package.json`, but not the incrementing of the version itself. Thus, the Fastlane utility is combined with the `npm version` package [46]. Three new scripts have been added to the *scripts* section of `package.json`.

By running any of the scripts, the `package.json` version is incremented accordingly, and after that, the synchronization process described above is executed with Fastlane. It is vital to execute one of these scripts before attempting to process a new build which will be uploaded to the Play Store or App Store.

A custom lane `bump_build_number` works similarly to the lanes mentioned above. It increments the build number for the iOS project and the version code for the Android project ensuring that these properties are unique for each new build. It is run automatically in the pipeline before each build.

```
{
  "bump_patch":
    "npm version patch --no-git-tag-version
    && bundle exec fastlane set_version",
  "bump_minor":
    "npm version minor --no-git-tag-version
    && bundle exec fastlane set_version",
  "bump_major":
    "npm version major --no-git-tag-version
    && bundle exec fastlane set_version"
}
```

6.6 Integration and automation scripts

As many of the processes described above require some manual steps and configuration to put them in place, it was necessary to consider how to make the ease of integration of the pipeline as intuitive and fool-proof as possible. Moreover, the ease of configuration is also a critical factor in choosing the specific automation technologies, which was mentioned in chapter 5.

6.6.1 Architecture

The initial concern was how and where to store the necessary parts of the pipeline and its tools. The concern is resolved by storing all related code, scripts, and READMEs in a single public ¹ GitLab repository. Making the repository public makes it easier for developers to access the pipeline's resources and does not pose critical security risks, since the repository does not contain any sensitive information. The repository can then be cloned, forked, or downloaded by individual files using tools like curl [47].

Having all the related files in one centralized location has its benefits:

- the repository is the single source of truth,

1. the repository is still accessible only through the company's *Virtual Private Network*

- the pipeline's development is well-versioned and can be tracked,
- new versions of the pipeline and its setup tools can easily be downloaded and/or updated.

Nevertheless, choosing to store the pipeline utilities in a public Git repository means that no sensitive information can be stored alongside it. Therefore, all the secrets (like authorization tokens) must be obtained somewhere else.

6.6.2 Pipeline configuration

The pipeline may be configured with a custom configuration script I created. The script serves as a guide to the developer configuring the pipeline and offers significant support throughout the whole process. The script supports:

- automatic script updates,
- dependency check,
- current CI/CD pipeline YAML file download,
- initializing a new EAS project,
- prompting for iOS application ID and setting it automatically,
- automatically configuring the standard build profiles – test and production,
- setting up Fastlane,
- reminding the developer to add an Expo authorization token and other secrets to GitLab CI/CD variables,
- `app.json` automatic configuration,
- preparing automatic versioning scripts,
- displaying instructions for Google integration configuration,
- downloading a shell script for manual Android builds.

Utilizing the script, the developers introducing the React Native pipeline into their project have a significant amount of configuration done automatically. Moreover, the steps of the configuration, when manual actions need to be taken, are well-documented, and the developer is thoroughly guided with specific URLs and instructions.

The workflow of using the script goes as follows: After the script is run in the root of the React Native project, the current version of the script is checked against the version stored in the remote GitLab repository. The version of the script is stored inside the script itself as a variable. The GitLab repository hosts a `VERSION` file where the latest

version is specified, and these values are compared. Should the script be outdated, the developer will be prompted whether or not they want to update it. The update is then either performed or ignored and the script proceeds. The versioning of the script is automated as well with a Python [48] script. Upon executing the Python script, the developer is prompted to specify how and if the version should be increased. Husky [49], a Node Package Manager (*npm*) package, is used to run this script before each commit to the central repository. Automating the versioning of the pipeline configuration script preemptively mitigates human error or mismatches in versions.

An important step is to check whether all the necessary dependencies are installed on the device running the script. The script checks for the presence of Node.js, Expo CLI, Ruby, and Bundler. Some of these dependencies, such as Expo CLI or Bundler, may be installed automatically, while others require manual installation from the developer.

After checking for the presence of `.gitlab-ci.yml` in the project root, the script downloads the CI/CD pipeline YAML file. If the file is already present, the script prompts the developer to decide whether to overwrite it. This is necessary as the pipeline configuration file itself will most probably be modified to suit each project the best. Overwriting it would cause the loss of the modified version of the pipeline.

Moving forward, an EAS project needs to be configured to use EAS utilities. The script attempts to initialize a new EAS project with the command `eas build:configure`. EAS then seamlessly guides the developer through configuring builds for the new project. To run builds from the pipeline, it is necessary to specify the application ID for iOS applications. Therefore, the user is prompted to provide it or skip the step and provide it manually later. A Python script parses `eas.json` and inserts the provided application ID into the corresponding fields. Subsequently, the test build profile utilized by the pipeline for test builds is configured, utilizing a one-line Python script to adjust the `eas.json` file.

```
import json, sys; fp = 'eas.json'; data = json.
    load(open(fp)); data['build'] = data.get('
    build', {}); data['build']['test'] = {'android
```

```
'': {'buildType': 'apk'}}} if 'test' not in data
['build'] else data['build']['test']; json.
dump(data, open(fp, 'w'), indent=2)
```

The next step is to set up Fastlane, which is used for versioning and APK distribution to Google Drive. A Gemfile [50] is generated to contain the Fastlane dependency. The Fastfile configured in the remote repository is downloaded and stored in the fastlane directory at the root of the project. After that, Fastlane and the necessary plugins are installed.

```
bundle install
bundle exec fastlane add_plugin
  increment_version_name increment_version_code
  load_json google_drive
```

The script then continues by prompting the user to add the Expo authentication token to the GitLab CI/CD variables, which is a crucial step in making the builds work correctly.

The owner field is added to app.json using sed [51] with the nickname of the shared Expo account utilized for all React Native projects. This step is necessary since the Expo builds fail without it.

To automatically set up the versioning scripts, the npm-add-script npm package is utilized alongside the npm version command and the Fastlane set_version lane that was previously configured. Versioning was described more thoroughly in section 6.5.

Google integration instructions are then printed to the developer running the script. The developer is provided a link to the Google service account's private key (the key is safely stored and only authorized developers can access it). This key should be in the local repository to set up Play Store and Google Drive uploads. This file is also automatically added to the .gitignore file to prevent accidental commits to version control.

The last step involves downloading a shell script that may be used for manual Android application builds.

After the whole setup is complete, the developer is notified that it is necessary to build and submit the application manually at least once for both platforms so that EAS can obtain or generate the credentials needed for the build process. Subsequent builds will be fully performable through the pipeline environment.

6.7 Extendability

The extent to which the whole solution can be extended was a major concern when designing it. Consequently, I put emphasis on the solution being well-documented and completely customizable.

Each pipeline job can be enhanced with custom project-specific adjustments. These enhancements can be done either locally or directly in the Git repository that is hosting the pipeline and its utilities. EAS offers numerous additional configurations that enable completely custom build or deployment settings where the developer has complete control over what EAS build produces.

Since the configuration script that prepares the environment and downloads the necessary files has an automatic update system built in, new extended versions of the script can be distributed to its users automatically with ease.

A well-organized project documentation is paramount for other developers' ability to contribute and extend it. Hence, I documented the project thoroughly and appropriately with README files and occasionally inline comments directly in the code.

Utilizing Git for version control opens up easy ways to extend the implementation. Merge requests can be opened to either fix bugs in the code or submit completely new functionalities. Should any developer utilizing this tool find a bug or miss a feature, they can easily integrate their new proposed changes into my code.

This thesis mainly focuses on introducing pipeline automation to already existing projects. Nevertheless, all the implemented tools may be even more powerful when paired with custom project initialization tools that may be introduced in the future. Subsequently, all the new React Native projects in the company would be standardized and utilize the same core automation principles.

6.8 Proof of concept

A proof of concept was conducted with another developer from the React Native team in the company. He was tasked to configure the pipeline in a new project without knowing anything about the process

beforehand. The only information source provided to him was the repository's README file and the configuration script itself.

The chosen project is a multi-platform mobile application developed for both iOS and Android. At the time of writing this thesis, it is still in development. iOS test versions for customers are uploaded through App Store Connect to TestFlight. Android APK files, on the other hand, are distributed to a Google Drive folder.

6.8.1 Issues

During this proof of concept, we encountered a few smaller issues that prevented a smooth integration of the pipeline. These issues were mostly minor and related to issues with dependencies and access restrictions that were resolved on the spot. The implemented versioning and updating system in the setup script significantly increased my work pace when resolving bugs, since new versions of the script could be distributed swiftly.

Nevertheless, a bigger issue regarding the pipeline surfaced. Many of the company's projects consist of a frontend part and a backend part. These two (predominantly *independent*) blocks are sometimes stored in the same repository in different directories. The GitLab pipeline configuration file needs to be stored in the root of the project², and our backend already has an established pipeline architecture. Consequently, the React Native pipeline configuration either had to be merged with the backend pipeline (which would be non-trivial to automate and prone to errors), or a different approach had to be used.

I chose to utilize GitLab's downstream pipelines [52]. They allow us to define a job in the pipeline configuration file that triggers a child pipeline. In our scenario, the child pipeline is the React Native pipeline, and the parent pipeline is either the root backend pipeline or a new pipeline created at the root of the Git repository. This is achieved with a bash script combined with a Python script. Adding a new *mobile* stage and new *mobile trigger* job to the root pipeline configuration is sufficient for this use case.

2. Technically, the path to the file can be configured in settings, but this does not handle multiple pipeline files.

```
mobile_trigger:
  stage: mobile
  trigger:
    # Path CUSTOM_PATH is adjusted
    # automatically with Python when
    # inserting
    include: CUSTOM_PATH/.gitlab-ci.yml
    strategy: depend
  rules:
    - changes:
      - CUSTOM_PATH/**/*
```

Nevertheless, after resolving the newly discovered issues, the pipeline was successfully configured and introduced into an actual commercial project of the company.

Conclusion

This thesis explores and analyzes automation tools for building and deploying React Native mobile applications and implements an automation toolset specifically designed for InQool's needs.

The core of the implementation is the architecture which uses Expo EAS to handle application builds, application distribution, and credentials management. A custom Fastlane integration complements EAS by automating versioning and uploads of application bundles to Google Drive. The tools are combined and utilized within a CI/CD pipeline alongside accompanying shell and Python scripts that automate its integration.

The designed pipeline substantially contributes to the ease of mobile application deployment in InQool by automating repetitive tasks associated with application building and application distribution. Consequently, development teams' time is freed up as the pipeline significantly reduces the amount of manual handling the developers had to perform in the past. Thus, optimizing the distribution of human resources in the company. Moreover, the risk of human error during the deployment processes is minimized, ensuring a reliable and consistent delivery of applications to end users.

The pipeline is integrated into a commercial project of the company, demonstrating the benefits of an automated deployment process in a real-world professional scenario.

Last but not least, the implemented tools can be effortlessly customized and expanded to scale with the scope of InQool's projects and evolving workflows in the times yet to unfold. Moreover, defining new pipeline jobs is straightforward, as this thesis establishes a solid foundation for future automation enhancements.

Bibliography

1. *React Native · Learn once, write anywhere* [online]. Meta Platforms, Inc. [visited on 2024-05-03]. Available from: <https://reactnative.dev>.
2. *React* [online]. Meta Platforms, Inc. [visited on 2024-05-03]. Available from: <https://react.dev/>.
3. BARGER, Reed. Is React a Library or a Framework? Here's Why it Matters [online]. 2021 [visited on 2024-01-24]. Available from: <https://www.freecodecamp.org/news/is-react-a-library-or-a-framework/>.
4. *Vue.js - The Progressive JavaScript Framework | Vue.js* [online]. Evan You [visited on 2024-05-03]. Available from: <https://vuejs.org/>.
5. *Angular* [online]. Google [visited on 2024-05-03]. Available from: <https://angular.io/>.
6. *ECMAScript® 2025 Language Specification* [online]. Ecma International [visited on 2024-05-03]. Available from: <https://tc39.es/ecma262/>.
7. *What is XML? - XML File Explained - AWS* [online]. Amazon Web Services, Inc [visited on 2024-05-03]. Available from: <https://aws.amazon.com/what-is/xml/>.
8. *JSX* [online]. Meta Platforms, Inc., 2022-08-04 [visited on 2024-01-25]. Available from: <https://facebook.github.io/jsx/>.
9. *Learn Build Tools: Browser Compatibility and Transpilation Cheatsheet | Codecademy* [online]. Codecademy [visited on 2024-01-25]. Available from: <https://www.codecademy.com/learn/learn-build-tools/modules/learn-javascript-transpilation/cheatsheet>.
10. *Babel · Babel* [online]. [visited on 2024-05-03]. Available from: <https://babeljs.io>.
11. *HTML: HyperText Markup Language | MDN* [online]. Mozilla Corporation [visited on 2024-05-03]. Available from: <https://developer.mozilla.org/en-US/docs/Web/HTML>.

BIBLIOGRAPHY

12. *Document Object Model (DOM) - Web APIs* | MDN [online]. MDN [visited on 2024-01-26]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.
13. TEAM, Codecademy. *React: The Virtual DOM* [online]. [N.d.] [visited on 2024-01-26]. Available from: <https://codecademy.com/article/react-virtual-dom>.
14. PUROHIT, Rakesh. *The Power of React Virtual DOM: Revolutionizing DOM Manipulation* [online]. 2023 [visited on 2024-01-26]. Available from: <https://www.dhiwise.com/post/the-power-of-react-virtual-dom-revolutionizing-dom-manipulation>.
15. *React Native · Learn once, write anywhere* [online]. Meta Platforms, Inc. [visited on 2024-01-24]. Available from: <https://reactnative.dev>.
16. *Core Components and Native Components · React Native* [online]. Meta Platforms, Inc. [visited on 2024-01-27]. Available from: <https://reactnative.dev/docs/intro-react-native-components>.
17. *Metro | Metro* [online]. Meta Platforms, Inc. [visited on 2024-05-03]. Available from: <https://metrobundle.dev/>.
18. KOSMAL, Jakub. *How does React Native work? Understanding the architecture* [online]. 2022 [visited on 2024-03-03]. Available from: <https://medium.com/front-end-weekly/how-does-react-native-work-understanding-the-architecture-d9d714e402e0>.
19. RAHMAN, Anisur. *React Native — Ultimate Guide on New Architecture in depth* [online]. 2023 [visited on 2024-03-03]. Available from: <https://medium.com/@anisurrahmanbup/react-native-new-architecture-in-depth-hermes-jsi-fabric-fabric-renderer-yoga-turbo-module-1284a192a82b#58a0>.
20. SUSLA, Samuel. *The New Architecture Benchmarks* [online]. 2023 [visited on 2024-05-03]. Available from: <https://github.com/reactwg/react-native-new-architecture/discussions/123>.
21. *What Composes the New Architecture · React Native* [online]. Meta Platforms, Inc. [visited on 2024-03-05]. Available from: <https://reactnative.dev/docs/the-new-architecture/pillars>.

BIBLIOGRAPHY

22. *Introduction · React Native* [online]. Meta Platforms, Inc. [visited on 2024-03-03]. Available from: <https://reactnative.dev/docs/the-new-architecture/landing-page>.
23. *What is CI/CD?* [online]. RedHat, 2022-05-11 [visited on 2024-01-12]. Available from: <https://www.redhat.com/en/topics/devops/what-is-ci-cd>.
24. *Ruby Programming Language* [online]. Ruby [visited on 2024-05-22]. Available from: <https://www.ruby-lang.org>.
25. *fastlane - App automation done right* [online]. fastlane, 2022 [visited on 2024-01-09]. Available from: <https://fastlane.tools>.
26. *Create Your Own Action - fastlane docs* [online]. fastlane, 2023 [visited on 2024-01-09]. Available from: <https://docs.fastlane.tools/create-action/#local-actions>.
27. *Expo Application Services - Expo Documentation* [online]. Expo, 2023 [visited on 2024-01-09]. Available from: <https://docs.expo.dev/eas/>.
28. *iOS build process - Expo Documentation* [online]. Expo [visited on 2024-04-04]. Available from: <https://docs.expo.dev/build-reference/ios-builds/>.
29. *Adding custom native code - Expo Documentation* [online]. Expo, 2023 [visited on 2024-01-09]. Available from: <https://docs.expo.dev/workflow/customizing/>.
30. *Git* [online]. [visited on 2024-05-07]. Available from: <https://git-scm.com/>.
31. MAAYAN, Gilad David. *The Future of Jenkins in 2024* [online]. 2023 [visited on 2024-01-10]. Available from: <https://devops.com/the-future-of-jenkins-in-2024>.
32. *Jenkins - fastlane docs* [online]. fastlane, 2023 [visited on 2024-01-10]. Available from: <https://docs.fastlane.tools/best-practices/continuous-integration/jenkins/>.
33. *GitHub* [online]. GitHub, Inc. [visited on 2024-05-07]. Available from: <https://github.com/>.
34. *The Official YAML Web Site* [online]. [visited on 2024-05-07]. Available from: <https://yaml.org/>.

BIBLIOGRAPHY

35. SHEREMETA, Olga. Jenkins vs GitHub actions: Which CI Tool Is Right? [online]. 2023 [visited on 2024-03-05]. Available from: <https://testomat.io/blog/jenkins-vs-github-actions-which-cicd-tool-is-right/>.
36. *Pipeline security* [online]. GitLab [visited on 2024-03-20]. Available from: https://docs.gitlab.com/ee/ci/pipelines/pipeline_security.html.
37. JAIN, Rahul. Jenkins vs GitLab CI: Battle of CI/CD Tools [online]. 2020 [visited on 2024-03-20]. Available from: <https://www.lambdatest.com/blog/jenkins-vs-gitlab-ci-battle-of-ci-cd-tools/>.
38. *Sign your app* [online]. Google [visited on 2024-03-22]. Available from: <https://developer.android.com/studio/publish/app-signing>.
39. *Semantic Versioning 2.0.0 | Semantic Versioning* [online]. [visited on 2024-05-07]. Available from: <https://semver.org/>.
40. *Using OAuth 2.0 for Server to Server Applications* [online]. Google [visited on 2024-04-16]. Available from: <https://developers.google.com/identity/protocols/oauth2/service-account>.
41. *Configure EAS Build with eas.json - Expo Documentation* [online]. Expo [visited on 2024-04-16]. Available from: <https://docs.expo.dev/build/eas-json/>.
42. *Run builds locally or on your own infrastructure - Expo Documentation* [online]. Expo [visited on 2024-04-16]. Available from: <https://docs.expo.dev/build-reference/local-builds/>.
43. *Install GitLab Runner | GitLab* [online]. GitLab [visited on 2024-04-17]. Available from: <https://docs.gitlab.com/runner/install/>.
44. *Find and fix problems in your JavaScript code - ESLint - Pluggable JavaScript Linter* [online]. OpenJS Foundation [visited on 2024-04-17]. Available from: <https://eslint.org/>.
45. *Use existing credentials - Expo Documentation* [online]. Expo [visited on 2024-04-18]. Available from: <https://docs.expo.dev/app-signing/existing-credentials/>.

BIBLIOGRAPHY

46. *npm-version* | *npm Docs* [online]. npm, Inc. [visited on 2024-04-19]. Available from: <https://docs.npmjs.com/cli/v8/commands/npm-version>.
47. *curl* [online]. curl [visited on 2024-04-19]. Available from: <https://curl.se/>.
48. *About Python™* | *Python.org* [online]. Python Software Foundation [visited on 2024-05-02]. Available from: <https://www.python.org/about/>.
49. *Husky* [online]. typicode [visited on 2024-05-02]. Available from: <https://typicode.github.io/husky/>.
50. *CocoaPods Guides - Using a Gemfile* [online]. CocoaPods [visited on 2024-04-24]. Available from: <https://guides.cocoapods.org/using/a-gemfile.html#what-is-a-gemfile>.
51. *sed, a stream editor* [online]. GNU [visited on 2024-04-24]. Available from: <https://www.gnu.org/software/sed/manual/sed.html>.
52. *GitLab* [online]. Downstream pipelines | GitLab [visited on 2024-05-11]. Available from: https://docs.gitlab.com/ee/ci/pipelines/downstream_pipelines.html.

A Implementation

The implementation part of the thesis is submitted in an archive as an attachment to this thesis. The documentation can be found in the `README.md` file. To see an example output and interaction with the `setup-pipeline.sh` script, run `cat transcript.txt` in a terminal.