Tampere University

Henri Salminen

DESIGN OF DEVELOPMENT ENVIRONMENT FOR MOBILE APPLICATIONS

Faculty of Computing and Electrical Engineering
Master of Science Thesis
February 2019

ABSTRACT

HENRI SALMINEN: Design of development environment for mobile applications

Tampere University
Master of Science Thesis, 46 pages, 14 Appendix pages
February 2019
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiners: Professor Kari Systä and University Lecturer Terhi Kilamo

Keywords: Continuous Integration, Continuous Deployment, Continuous Delivery, CI/CD, iOS, Android, Git, Gitlab, branching, release management, mobile application development

A well-designed and functioning development environment is crucial for every software company to push the productivity of the development team to the max. With the appropriate development environment, the limited time of the development team can be used for productive work. In this thesis, a successful development environment for mobile application development is designed and implemented for the Finnish company called Piceasoft.

Piceasoft has been increasingly investing in the development of mobile applications over the last few years. The mobile application development team has identified problems with the development environment. In this thesis, these flaws of the development environment are pinpointed and solved.

The thesis declares a whole new Version Control System (VCS) and branching model for the mobile applications codebase. This VCS is deployed with self-hosted Gitlab instance that runs in the internal network and integrates with existing Lightweight Directory Access Protocol (LDAP) authentication system. With integrated Continuous Integration and Continuous Delivery (CI/CD) system of Gitlab, a fully automated CI/CD pipeline for mobile applications is created. The transition to the new system from old Subversion VCS is described.

The system implemented in this thesis turned out to be well suited for Piceasoft. The system was evaluated by interviewing the developers from the mobile application development team and PC development team as well as Quality Assurance (QA) engineers from the testing department. Additionally, data about integrated alpha builds of the applications during development was collected. The data shows a significant increase in deliverables available for integration and testing.

TIIVISTELMÄ

HENRI SALMINEN: Kehitysympäristön suunnittelu mobiiliapplikaatiokehitykseen
Tampereen yliopisto
Diplomityö, 46 sivua, 14 liitesivua
Helmikuu 2019
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: Professori Kari Systä ja Yliopistonlehtori Terhi Kilamo

Avainsanat: jatkuva integraatio, jatkuva julkaisu, jatkuva toimitus, CI/CD, iOS, Android, Git, Gitlab, kehityshaarojen hallinta, julkaisun hallinta, SDK, mobiilikehitys

Tarkkaan mietitty ja toimiva kehitysympäristö on elintärkeä kaikille ohjelmistoalan yrityksille, jotta kehitystiimien tuottavuus saadaan maksimoitua. Kunnollisen kehitysympäristön avulla kehitystiimin on mahdollista optimoida käytettävissä olevat resurssit tuottavan työn tekemiseen. Tässä diplomityössä rakennetaan toimiva kehitysympäristö Piceasoft nimisen Suomalaisen ohjelmistoalan yrityksen mobiilikehitystiimille.

Viime vuosien aikana Piceasoft on satsannut yhä kasvavassa määrin mobiilikehitykseen. Mobiilikehitystiimi on kuitenkin todennut ongelmia ja puutteita Piceasoftin mobiilikehitysympäristössä. Tässä diplomityössä nämä ongelmat tuodaan esille ja ratkaistaan.

Diplomityössä esitellään kokonaan uusi versionhallintajärjestelmä ja tähän liittyvä kehityshaaramalli mobiilisovellusten lähdekoodipohjalle. Tämä uusi versionhallintajärjestelmä toteutetaan itse julkaisun ja yläpidetyn Gitlab instanssin avulla. Tämä Gitlab instanssi pystytetään Piceasoftin sisäverkkoon ja se integroituu olemassa olevaan LDAP autentikaatio järjestelmään. Täysin automatisoitu jatkuvan integraation ja jatkuvan julkaisun järjestelmä mobiilisovelluksille toteutetaan Gitlabin tarjoamilla työkaluilla. Siirtyminen vanhasta versionhallintajärjestelmästä uuteen kuvaillaan.

Diplomityössä luotu järjestelmä osoittautui erittäin toimivaksi ja vaatimuksia vastaavaksi. Järjestelmän toimivuutta ja soveltuvuutta tarkoitukseensa arvioitiin haastattelemalla kehittäjiä mobiili- ja PC kehitystiimistä, kuin myös haastattelemalla laadunvarmistusinsinöörejä laadunvarmistusosastolta. Tämän lisäksi aikaisemmasta ja uudesta järjestelmästä kerättiin dataa mobiilisovellusten esiversioiden julkaisujen määristä. Tämä data osoitti merkittävän kasvun julkaistujen esiversioiden määrissä mahdollistaen ketterämmän integraation ja testauksen mobiilisovelluksille.

PREFACE

This Master of Science Thesis was written as an assignment from a Finnish company called Piceasoft. The idea behind the thesis was to pinpoint the most significant flaws from the existing mobile applications development environment and design and implement successful development environment.

I want to thank Piceasoft, especially CTO Jani Väänänen for providing such a fantastic opportunity and tools to create this kind of system. I want to thank the Tampere University of Technology, especially Professor Kari Systä for all the guidance and knowledge during the studies and the writing of the thesis.

Special thanks belong to my wife Erika, who has been supporting and encouraging me during the studies and writing of the thesis. I also want to thank my parents for gently pushing me towards this degree.

" Love the Problem, Not the Solution." - Ash Maurya

In Tampere, Finland on February 23, 2019

Henri Salminen

CONTENTS

1. INTRODUCTION 1
2. BACKGROUND 3
2.1 Development environment 3
2.2 Scope of the thesis 4
2.3 Products 4
2.4 Mobile applications 5
2.5 Android 5
2.6 iOS 6
2.7 Codesigning 6
2.7.1 Signing the Android applications 7
2.7.2 Signing the iOS applications 7
2.8 Subversion and Git Version Control Systems 8
2.9 Continuous Integration, Delivery and Deployment 8
3. ENVIRONMENT 10
3.1 Mobile applications high level architecture 10
3.1.1 Android architecture 10
3.1.2 iOS architecture 11
3.2 Previous development environment 12
4. VERSION CONTROL SYSTEM 14
4.1 Selecting the Version Control System 14
4.2 Branching model 15
4.3 Configuration management 17
5. GITLAB SETUP 20
5.1 Configuring SSL 20
5.2 LDAP integration 21
5.3 Transition from Subversion to Gitlab 21
5.4 Setting up Gitlab Runners 22
5.4.1 Setup Gitlab Runner host machines 23
5.4.2 Registering Gitlab Runners 23
6. CONTINUOUS INTEGRATION, DELIVERY AND DEPLOYMENT 26
6.1 Analysis of requirements 26
6.2 Selecting tools for CI/CD system 27
6.2.1 Setup accounts 27
6.3 Relationship between branching model and configuration management 28
6.4 Implementation of the CI/CD system 29
6.4.1 Managing versioning 30
6.4.2 Building 35
6.4.3 Testing 36
6.4.4 Code signing 36
6.4.5 Deployment and delivery 39

7. EVALUATION 43
8. CONCLUSION 46
9. REFERENCES 47

APPENDIX A: Snippet from iOS low memory report defining kernel version

APPENDIX B: Gradle build script implementation for Android application version management

APPENDIX C: Lanes for building PiceaOne iOS application variants

APPENDIX D: Lanes for building PiceaOne Android application variants

APPENDIX E: CI/CD pipeline

APPENDIX F: Survey results for mobile application development team

APPENDIX G: Survey results for PC application development team

APPENDIX H: Survey results for PC QA team

LIST OF FIGURES

Figure 1. *Continuous Delivery and Continuous Deployment processes. 9*
Figure 2. *PiceaOne high-level software architecture on Android platform. 10*
Figure 3. *PiceaOne application high-level software architecture on iOS platform 12*
Figure 4. *Git flow branching model. 17*
Figure 5. *Target configurations for PiceaOne Xcode project. 19*
Figure 6. *Build scheme configurations for PiceaOne Xcode project. 19*
Figure 7. *Relationship between branches and application variants. 29*
Figure 8. *Layers defined in CI/CD system implementation. 30*
Figure 9. *Outputs of the fastlane build scripts for iOS application variants. 35*
Figure 10. *Outputs of the fastlane build scripts for Android application variants. 36*
Figure 11. *Increase of alpha build during release development cycle before and after new development environment. 44*
Figure 12. *Deployed alpha release counts for PiceaOne (Beta) application. 45*

LIST OF SYMBOLS AND ABBREVIATIONS

VCS	Version Control System
IDE	Integrated Development Environment
USB	Universal Serial Bus
SDK	Software Development Kit
AAR	Android Archive
ART	Android Runtime
JIT	Just-in-Time (compiler philosophy)
AOT	Ahead-of-Time (compiler philosophy)
NDK	Native Development Kit (Android)
XNU	Computer operating system kernel (Originally developed by Apple Inc.)
i386	Microprocessor architecture
x64_64	Microprocessor architecture
API	Application Programming Interface
CVCS	Centralized Version Control System
DVCS	Distributed Version Control System
CI	Continuous Integration
CD	Continuous Delivery or Continuous Deployment
QR	Quick Response (in context of QR codes)
QR code	Quick Response Code
CI/CD	Continuous Integration and Continuous Delivery or Continuous Deployment
LDAP	Lightweight Directory Access Protocol
DNS	Dynamic Name Services
SSH	Secure Shell
HTTPS	Hypertext Transfer Protocol Secure
TLS	Transport Layer Security
SSL	Secure Sockets Layer
NGINX	Web-server widely used as a reverse proxy
URL	Uniform Resource Locator
HTTP	Hypertext Transfer Protocol
IT	Information Technology
JRE	Java Runtime Environment
JDK	Java Development Kit
DMG	Apple Disk Image
AWS	Amazon Web Services
UI	User Interface
QA	Quality Assurance
dSYM	Set of files containing application debug symbols

1. INTRODUCTION

Mobile devices have become a more and more popular development platform in the software business. Most of the software companies work directly or indirectly with the mobile applications. Two of the most common platforms for mobile devices are Android and iOS. The application development mainly focuses on these two platforms as they cover 98.9 % of the mobile device platform market share [1].

Companies developing mobile applications, have to manage different variants of them. When mobile applications are developed natively for both of the platforms and applications need to support multiple versions of the operating systems, the number of different application variants raises rapidly. Application variants include different platform versions, different release versions of the application, different build variants of the application and different release configurations of the application. As the number of individual applications goes up, it is crucial to divide some of the functionalities to independent modules that can be shared across the applications. This results in even more complex project structures resulting in even more difficult management of the variants.

To keep track of changes in the codebase, sophisticated and well-designed Version Control System (VCS) is needed. With a well-designed VCS, it is easy for the development team to manage upcoming releases, implement new features and release hotfixes to the production in agile manners. The amount of manual labor required to maintain and manage the different versions of the software becomes a real problem as the development work continues. Manually managing all of the releases, alpha and beta builds and testing have a significant negative impact on the performance of development and testing teams.

In this thesis, a fully functioning development environment is designed and implemented. The environment covers VCS and Continuous Integration and Continuous Delivery system. The thesis tackles the problems identified in the existing development setup and makes a great effort to improve the productivity of the mobile application development team.

The thesis is written for a company called Piceasoft Ltd. Piceasoft is a Finnish company that focuses on life cycle management of mobile devices. Piceasoft delivers software products for big global operators, retailers, repair centers and recyclers. During the writing of the thesis, Piceasoft worked with one mobile application that is developed natively for both iOS and Android platforms. The application supports Piceasoft PC and

Mac solution by providing services for device hardware and software diagnostics. This thesis tackles the problems identified in the mobile application development environment.

2. BACKGROUND

To deploy any software to production, the software must first be defined, designed, implemented, tested and deployed by some development team. The simplest example is one person with a laptop. Everything that is needed is a text editor for writing code and a manual process of building and publishing it. Software companies need a well-designed and functioning development environment to ensure their productivity.

2.1 Development environment

The development environment can be described as a set of tools and processes that are used to create the final software product. The standard development environment should at least contain tools and methods for version control, building and compiling, testing and delivery.

Version Control Systems (VCS) are used to manage the source code [2]. Version Control System keeps track on changes in source code and preserves the history of the changes helping the developers to work together on the same codebase [3].

Another mandatory requirement for the development environment is the toolset used for compiling and building the source code into a complete deliverable software product. Toolset can be as simple as Integrated Development Environment (IDE) [4] on the developer's laptop which is capable of compiling and building the final product with a push of a button. However, projects that have multiple people working on this solution is far from ideal. The target is that the deliverable is compiled and build automatically when the codebase changes. This way the up-to-date version of the software is always available for integration and testing.

Before the compiled and built product can be released to the waiting customers, it is almost a mandatory practice to run tests for it to make sure it functions as intended. Indeed, the developer who compiled and built the product could test the solution before publishing it, but that is not the most effective way of doing things. The tests need to be run every single time the software is released to make sure that the changes have not broken the existing functionality. Software companies nowadays prefer to have automated testing at least for the core functionality of the software. Automatic testing saves a lot of effort from the testing department as each new version of the software is going through core functionality testing at least.

Finally, as the codebase is built into deliverable software product and it has passed the core functionality testing, it is time to release it into its environment. The environment usually depends on the stage of the software readiness, and the product is typically

deployed into a staging environment before the actual deployment to the production environment. In the staging environment, final integration and end-to-end testing, where the software is tested together with its related components [5] can be done. After the software is tested in staging environment, it can be deployed to the production environment.

2.2 Scope of the thesis

Software companies would prefer to have a proper development environment set up to boost the development work. With the appropriate development environment, the productivity of the company increases, and the quality of the software goes up. In practice, most often the somehow working VCS and narrow set of unit tests is the only thing that software projects have. The lack of a proper development environment is most often caused by the lack of resources or the fact that the company is not willing to invest in building this kind of setup which is not directly profitable.

In this thesis, a functional development environment is designed and implemented for developing native mobile applications. The thesis will pinpoint the problems with the existing setup and tries to resolve these as efficiently as possible. The focus will be in the development environment itself as well as in tools used to implement it. The thesis does not cover the testing, but it will take it into account for further development of the system.

2.3 Products

Piceasoft initially focused on software running on PC and Mac where the communication with the mobile devices was done via different media like Universal Serial Bus (USB) cable or Bluetooth. At first, the software was capable of transferring content from one device to another regardless of the source and target platforms. As the product line-up expanded, and more features were added to the software the simple connection to the device was no longer enough and the need for native mobile applications arose. A native application running on the mobile device with the capability to communicate with the PC software opened new possibilities.

At first, Piceasoft launched an application to the Android platform only. The name of the application was *Piceaconnector*. The application collected more detailed information about the device and was able to communicate with the PC software through a USB connection. The application was released to the Google Play Store, but it was possible to install it from PC application using a USB connectivity as well.

Later on, the product line-up expanded to the device hardware diagnostics. *PiceaDiagnostics* named applications were used to run the diagnostics in mobile devices with a help from PC solution. *PiceaDiagnostics* applications were developed for both Android and iOS platforms and they were able to communicate with the PC and Mac

software. At this point, there were three applications in total, *Piceaconnector* for Android and *PiceaDiagnostics* for Android and iOS. At the same time, the market started to move towards entirely mobile application-based solutions.

The development work of the mobile-based diagnostics evoked a new idea of having only one application per platform which would work in the combination of the PC software as well as individual mobile application. The mobile applications' software was divided into smaller individual components which were responsible for logical parts of the functionality. New PiceaOne applications were built on top of these smaller logical components to be later used in different applications or to be licensed separately to Piceasoft's customers as a Software Development Kit (SDK) [6].

2.4 Mobile applications

Mobile applications are usually designed to implement some specific task and users have multiple applications installed on their devices. Mobile applications are typically built by combining and using smaller components of software. Each of these components is responsible for implementing some specific task on the application. For example, one component of the application can be responsible for drawing notifications on the screen. Many of these components are written with good abstraction level to make it possible for developers to reuse them across multiple projects and applications. These components are usually implemented as Android Libraries [7] or iOS frameworks [8].

Android libraries and iOS frameworks are software modules that can be used in multiple projects. The modules can include everything needed to build and use them including source code, resource files and information related to module [7] [8]. Android libraries are packaged as Android Archive (AAR) files [7], and iOS frameworks are packaged as hierarchical directories with .framework file extension [8].

2.5 Android

Android is an operating system for mobile devices developed by Google [9]. It is optimized and designed to be used through touchscreen instead of conventional input devices like a mouse and a keyboard.

Android was initially built on top of the Linux kernel, and it used a Dalvik virtual machine to run the applications. Dalvik virtual machine was later replaced by Android Runtime (ART)**.** Dalvik virtual machine used a Just-in-Time (JIT) compilation to execute the programs. That means that the execution of the computer code requires compilation during the time of execution. Compared to the JIT compilation the Android Runtime compiles the programs during the installation which results in better performance at run time. This is called Ahead-of-Time (AOT) compilation [10]. As the programs are compiled into native machine code instead of interpreting each line of byte code, the start-

up time of the programs is reduced, and the battery life of mobile devices is improved [11]. These benefits come with the price of increased installation times and internal storage usage. However, the gains are more significant than the drawbacks in modern mobile device operating system.

Android applications are most often written in Java, however, in 2017 Google revealed full support for Kotlin programming language together with Android Studio 3.0 release [12]. Android Native Development Kit (NDK) makes it possible to write native C code to the Android applications as well [13]. The Piceasoft product line-up needs all of these languages and technologies.

2.6 iOS

iOS is an operating system for mobile devices developed by Apple [14]. Like Android, iOS is optimized for touchscreen usage. iOS was initially designed for iPhone devices, but Apple nowadays uses it within iPads and iPods as well.

iOS uses Darwin's XNU kernel [Appendix A] developed by Apple. iOS runs on ARM processor architecture, but when the developers run applications in the iOS simulator on macOS, the processor architecture is either i386 or x64_64 depending on the Mac computer's processor architecture. The processor architectures need to be considered when building frameworks for iOS as the target application needs to run in physical devices as well as the simulators. iOS frameworks can be built to contain binaries for both processor architectures, but the simulator architecture needs to be stripped out, before the application can be submitted to the App Store.

iOS applications used to be written in Objective-C, however new programming language called Swift has gained popularity amongst the developer community and is advertised as a preferred language for new applications for iOS. Additionally, some of the Application Programming Interfaces (API) require pure C or C++ code. Piceasoft product line up contains hardware diagnostics tests for the mobile devices; hence low-level device access is required, and C/C++ code is needed.

2.7 Codesigning

Both Android and iOS platforms require that the applications are digitally signed before installation. By signing the applications with the trusted certificates, the user can trust that the applications are not altered by any third parties or corrupted after the trusted developer has released the software. By allowing the installation of only signed applications the operating system can validate the authenticity and integrity of the software. This is a security mechanism that is designed to protect the user as well as the original author of the software.

2.7.1 Signing the Android applications

Android platform only allows the installation and update of signed and trusted applications. Unsigned applications and application updates are rejected by the Google Play Store [15] or Android's package installer. In most cases, applications developed for the Android platform are distributed through Google Play Store which is Google's digital distribution service for Android applications. This is the distribution channel Piceasoft uses to distribute Android applications as well.

Android Studio is the IDE for Android development. When developing applications for the Android platform using Android Studio, the Android Studio automatically manages the signing of the applications before installation on the device. The Android Studio automatically generates a signing certificate to make it possible to test the application on the device while developing it. The Google Play Store does not approve these development certificates, and the application needs to be signed with proper production certificates before submitting the builds to the Play Store [16].

To retrieve the needed certificates, developers must create a Google account and login to Google's developer portal. From the developer portal, the developer can generate the required credentials to sign the applications. Android development environment provides two ways to manage the credentials. The preferred approach is to let Google Play securely manage the application signing keys [17] while another option is to control the signing keys by yourself. For legacy reasons, signing keys are managed by the Piceasoft, and it is crucial to handle these keys safely in the development environment.

2.7.2 Signing the iOS applications

As the Android platform, the iOS platform is strict when it comes to the integrity and authenticity of the applications. iOS allows only the installation of digitally signed and trusted applications. Unsigned applications are rejected by the Xcode [18] or Application Loader [19] before the application even reaches Apple's services. Xcode and Application Loader are the default tools Apple provides for uploading new application binaries to the App Store [20]. App Store is Apple's digital distribution service for iOS and macOS applications.

During the development of the applications, debug versions are signed before they can be installed to the device. Xcode is the official IDE for the iOS applications. It can automatically manage application signing for developers. To deploy applications to the App Store, the developer must enroll in Apple's developer program which has a fixed annual charge [21]. From the development portal, application signing certificates and provisioning profiles can be created and managed. Xcode can automatically handle the signing of the applications but is it is also possible to manually select the proper provisioning profile when signing the applications. The certificates and provisioning

profiles are managed by Apple's developer portal and are accessed via Apple developer accounts. The macOS default Keychain Access application controls private keys for the certificates. The private keys are secrets and must be treated accordingly.

2.8 Subversion and Git Version Control Systems

Two of the most common VCS are Subversion and Git. Subversion was the most commonly used for a long time, but in last few years Git has become more and more popular choice. According to Eclipse Community study back in 2014, for the first time Git was more used VCS in Java projects than the Subversion. [22]

Subversion is developed as a project of the Apache software foundation [22]. According to the Apache software foundation: Subversion is *"Enterprise-class centralized version control for the masses"* [23]. A Centralized Version Control System (CVCS) tracks the changes on a central server that developers use to retrieve and publish changes to the codebase. Each commit is published on the central server as the developer commits the changes and the changes are immediately available for the other developers to pull [24].

As contrary to the Subversion, Git is a Distributed Version Control System (DVCS) [24]. This means that each developer has a full local copy of the entire repository, including the history. Instead of checking out the most recent codebase from the Subversion in Git the whole repository is cloned from the remote repository to the local computer. This is a significant benefit when it comes to working offline because the majority of Git commands can be executed even without connection to the remote server.

2.9 Continuous Integration, Delivery and Deployment

Continuous Integration (CI) is a practice and process that aims to regular code integration to the development and delivery environments [26]. In the Continuous Integration process, a code change from the developer is integrated into the shared codebase. During the integration of the changes, a set of tests are executed to ensure that the code change under integration is functioning as expected [27]. The tests are usually simple, quick and easy to run and test the code changes in isolation. These tests are often referred as *unit tests* or *commit tests*. In unit testing code is tested in individual component level and the testing aims to make sure each component in the software functions as expected [28]. The primary purpose of the state is to quickly catch any errors in the code and to ensure the code quality [27].

Continuous Delivery (CD) and Continuous Deployment (CD) are practices and processes that aim to regular delivery and deployment of the software [29]. The difference between these two is that in the Continuous Delivery the version of the software is prepared for the deployment [30] while in the Continuous Deployment the version of the software is automatically deployed [31]. In Continuous Delivery, the code is automatically built,

tested and delivered to different environments. This makes it possible to extend the testing of the software from unit testing to more extensive User Interface (UI) and integration testing. Often the software is automatically deployed to the R&D or staging environment by Continuous Deployment and prepared for the production deployment by the Continuous Delivery. It reduces the workload of the operations team by automating frequently needed manual steps in the process leading to better effectiveness of the delivery and better availability of the new features. Continuous Delivery and Continuous Deployment processes are described in figure 1 below.

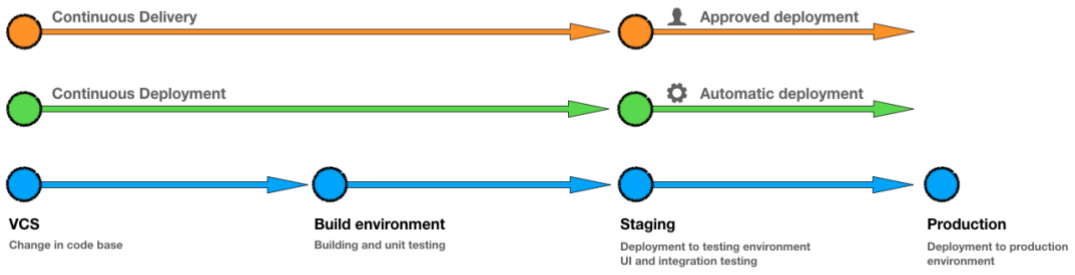


Figure 1. *Continuous Delivery and Continuous Deployment processes.*

3. ENVIRONMENT

3.1 Mobile applications high level architecture

Piceasoft has created a custom SDK consisting of multiple individual libraries and frameworks. The SDK includes tools for harvesting information about the device, checking different statuses of the device settings, communicating with Piceasoft PC or online solution and running various diagnostics tests for the device. By combining and using the components of the SDK, a single multifunctional mobile application can be created with less effort than by writing everything from scratch. The SDK is designed in a way that multiple applications can efficiently utilize the technologies. It can be shipped to Piceasoft's customers or partners, to allow them to build custom solutions utilizing technologies from Piceasoft. *PiceaOne* application has been developed on top of this SDK.

3.1.1 Android architecture

PiceaOne Android application utilizes five components from the Piceasoft's Android SDK. Each of these components is written in Java and can be included as a dependency to any Android application. High-level architecture of *PiceaOne* Android application is defined in figure 2 below.

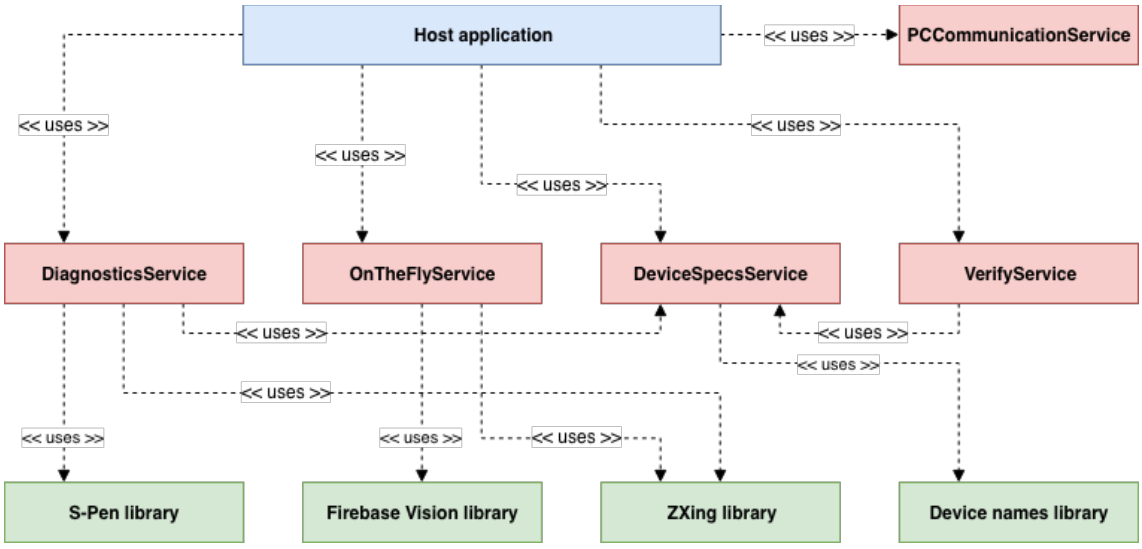


Figure 2. *PiceaOne high-level software architecture on Android platform.*

Services in Piceasoft SDK are marked with red boxes. The host application can be any mobile application utilizing Piceasoft SDK, but the architectural image describes the dependencies of *PiceaOne* application. The green boxes are external 3rd party dependencies that the Piceasoft SDK uses.

DeviceSpecsService is responsible for collecting the device identification information from the device. The service provides easy-to-use classes for accessing the device information by abstracting the tasks of obtaining the data from the different devices. Other services rely on this service when the device identification information is needed.

DiagnosticsService implements a large number of different hardware and software diagnostics tests. The service provides API which can be used to manage diagnostics sessions and the results of the tests. With this API it is possible to detect problems on the devices and provide possible solutions to resolve the issues found by the service. The component uses *DeviceSpecsService* as an internal dependency and usage of the *DiagnosticsService* requires including the *DeviceSpecsService* to the project as well.

VerifyService provides *DiagnosticsService* like API to be used to detect different values of the device settings. The API can be used for example to check if the device has Device Protection [32] enabled. The information is valuable for Piceasoft partners running buyback for used devices as Device Protection will not allow the device re-activation for a new user.

OnTheFlyService provides APIs for scanning QR codes and handling deep linking [33] to the applications. Deep linking offers a capability to link to the application via weblink. With the service, additional data can be passed to the application when it is installed and started via a deep link.

PCCommunicationService provides APIs for establishing a bidirectional communication channel between Piceasoft PC solution and mobile application. With the help of this service, Piceasoft PC solution and mobile solutions communicate with each other.

3.1.2 iOS architecture

PiceaOne iOS application utilizes all five components from the Piceasoft SDK as well. The components are written in Objective-C and can be included as a dependency on any iOS application. High-level architecture is defined in figure 3 below.

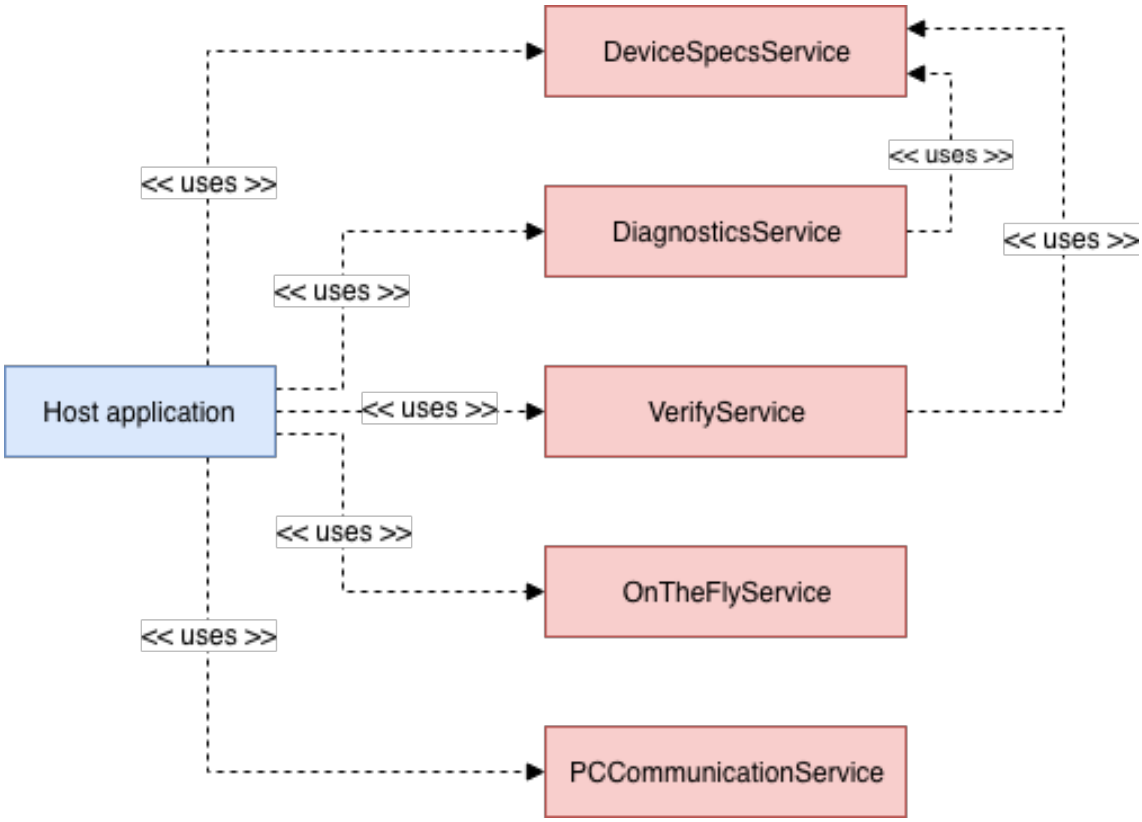


Figure 3. *PiceaOne application high-level software architecture on iOS platform*

The services in Piceasoft iOS SDK are as similar as possible to the Android ones introduced before. These services have the same functionalities and use cases as Android ones. Unlike the Piceasoft SDK for Android, the iOS versions do not have any external dependencies to 3rd party frameworks.

3.2 Previous development environment

The version control setup before this thesis used Subversion Version Control System (VCS) [23]. Piceasoft used Subversion in a traditional way of having Trunk as an unnamed base branch for the projects and having a new release branch created for each new release. Release branches diverged from the Trunk when the features for the release were mostly implemented. After branching, all of the fixes and features that were included in the release were merged from the Trunk to the release branch. Mobile development projects were managed by the same Subversion project and were bind to the branching of the PC software codebase.

Continuous Integration for PC solution was implemented by running commit checker for every commit to the VCS. Additionally, a new version of the product was built on a nightly basis. Continuous Integration for mobile applications did not exist. New versions of the mobile applications were manually triggered to build, and the system always used the Trunk branch as a codebase for a new build.

The most significant problem with the previous development environment was the branching model and how the build automation system was bound to it. The codebase of the mobile applications was located under the same Subversion project as the PC solution codebase; hence the branching was dependent on the branching of the PC application's codebase. In the branching model, each release meant a new release branch, which was then left alive. Development continued in Trunk after the release branch was created. In many cases, the branching point was not optimal for the mobile application development team as the PC development team fixed the branching point. It made it almost impossible for the mobile application development team to manage the codebase before the upcoming release because the build automation used Trunk as a codebase for building release versions of the mobile applications. Committing changes that were not supposed to be included in the next release, easily ended up there if the changes were committed to the Trunk before the release.

Another problem in the setup was the missing Continuous Integration and Continuous Delivery system. Integration of changes in mobile application's codebase required a lot of manual effort from the mobile application development team, and the changes were not going through any testing. This can result in poor software quality before the software makes its way to the quality assurance team. The poor quality of the software can cause issues during the release process, as a lot of fixes are done after the initial start of the release process. As a result, multiple iterations of bug fixing, and testing can lead to repetitive basic acceptance testing. The testing department needs to make sure that the changes in codebase do not affect to other functions of the software. These operational models can hurt the productivity of both the development and testing teams.

4. VERSION CONTROL SYSTEM

All of the well-known VCSs in the market can handle the most crucial tasks they are designed to do; manage your codebase history. However, there are a lot of fundamental differences between the systems. The selection of the VCS in this thesis focuses on selecting between the two most commonly used VCSs, Subversion, and Git. The reason why any other candidates are dropped out from the selection is the fact that Piceasoft has experience from working with Subversion and Git but not from any others.

The most important selection criteria for the VCS are the following:

1. The team must be familiar with the selected VCS.
2. VCS needs to integrate well into CI/CD systems.
3. VCS must provide well-established branching support.
4. Context switching needs to be easy and efficient.

4.1 Selecting the Version Control System

The most important aspect of choice was that the development team must be familiar with it. The mobile application development team had experience in Git for many years, but Subversion had only been used in the office and team had no experience from branches or tags in Subversion. Another significant factor was the feedback from the developers indicating a strong will to move away from Subversion. Selection strongly leaning towards Git based on the development team input changed the evaluation process a little bit to the direction of investigating the possible pitfalls if the transition to new VCS would be made.

When evaluating Git integration abilities to CI/CD systems integration to Jenkins [34] was evaluated as the Jenkins is the current CI/CD system server for Piceasoft. Jenkins does not support Git out-of-the-box, however, there is a plugin that can be used to integrate Jenkins and Git VCS [35]. For CI/CD system integration Gitlab [36] was evaluated as well. Gitlab is a single application that can manage the whole software development lifecycle including Git repository hosting and CI/CD support. Gitlab provides everything that was needed to set up the development environment with a Git repository and integrated CI/CD system.

The last criterion from the identified requirements was the well-established branching support and easy and efficient context switching. Both Subversion and Git support branching and functional branching model can be created with each system. In Subversion most of the developers find context switching difficult according to Muşlu et al [37].

Context switching is a lot easier with Git as it supports local branches and stashing [38]. Easier context switching supports the decision to transition from Subversion to Git.

When considering the speed of VCS nearly all of the operations to the Git are done locally on the computer which leads to an increased speed of the many operations compared to Subversion which requires access to the centralized server to operate. Git manages the repositories efficiently and the cloning of the entire Git repository is almost as fast as checkout operation from the Subversion server [39].

Git is by nature more fault tolerant than the Subversion. The problem with the Subversion is that the history is only in the centralized server and the fault in the system can be catastrophic if no proper backup system is set up. When working with the Git, each developer has a full clone of the repository locally so in a case of failure on a remote server the work is preserved in the local copies of the repository.

The problems in the previous development environment could have been solved by using either Git or Subversion. By only updating the build scripts of the prior automation server and designing a better branching model for the Subversion, significant improvements could have been achieved. The decision to move mobile application's codebase under Git version control system, design entirely new branching model and create a proper CI/CD system that integrates to the Git repositories was made mostly by respecting the desires of the development team who works daily with the selected tools. It is crucial to understand that the transition to modern DVCS does not resolve any of the issues if no proper branching model and processes are built around it.

4.2 Branching model

Promising branching model that supported the desired workflows was found from the blog post by Vincent Driessen [40]. The blog post described *a successful Git branching model* featuring some Git workflows. The branching model from the blog post was designed to help developers to stay on track of stages in the development. It featured workflows for development, features, hotfixes, and releases.

The basic idea is to have two main branches with an infinite timeline. First and the most active branch is called a *development* branch. The second one is the *master* branch. These branches will be named with the following naming conventions in the implementation:

- Development branch: development
- Master Branch: master

Master branch always reflects the state of currently released software. *Development* branch, on the other hand, works as a branch for daily development. By defining branches this way, the team will work in the *development* branch by default, push to the *development* branch and share the code between a team using the *development* branch.

The code change in the *master* branch always means a new release. With this workflow, the CI/CD system can even automatically build and release a new version of the application to the production environment if the codebase in *master* branch changes. The alpha builds can be built and delivered from the *development* branch to the testing department through internal testing channels.

The branching model defines three other branches that have a finite length timeline. These branches are called *feature*, *release* and *hotfix* branches. The following naming conventions are established for these branches:

- Feature branches: feature/<feature description>
- Release branches: release/<release version number>
- Hotfix branches: hotfix/<hotfix description>

The *feature* branches are used when the development of some major feature is started or if the developer works with the feature that is not part of the next upcoming release. *Release* branches are created once the *development* branch contains all of the features planned to the next release. The *release* branch is created from the *development* branch, and after that point, only bug fixes are allowed to be committed to the *release* branch. The bug fixes committed to the *release* branch are merged back to the *development* branch to make sure they are included in the later releases as well. Once the *release* branch codebase is stable and ready to be released the *release* branch is merged into the *master*, the *master* branch is tagged with the version number and the *release* branch is back merged to the *development*. After that, the *release* branch is deleted the *master* branch reflects the state of the released codebase.

When the development team needs to make hotfixes between the releases, a *hotfix* branch diverges from the *master* branch. On this *hotfix* branch, the critical bugs are fixed. Once the *hotfix* branch contains all of the fixes it is merged back to the *master,* and the *master* branch is tagged with the updated version number. Next, the *master* branch is back merged into the *development* branch to ensure that the upcoming releases will have the fix and then the *hotfix* branch is deleted.

The branching model is presented in figure 4 below.

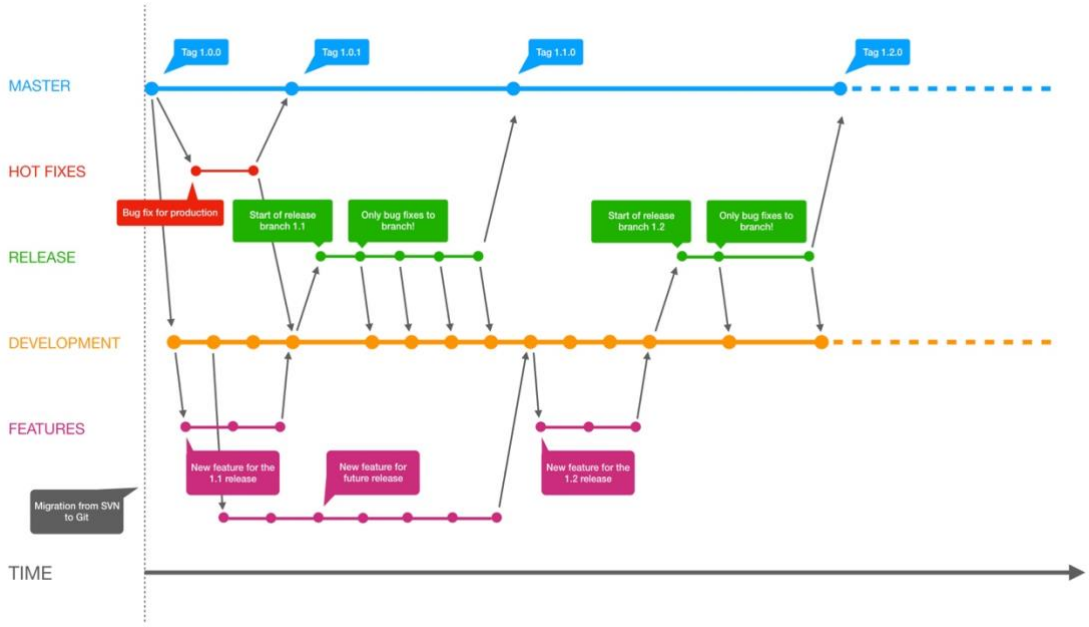


Figure 4. *Git flow branching model.*

4.3 Configuration management

Software project level configurations manage variants of the applications on both Android and iOS platforms. Both project structures support building applications with different settings from the same project. The dependencies of each configuration can be managed in the project level as well. In PiceaOne project, different configurations contain different dependencies.

Three different variants of PiceaOne application are set up for PiceaOne Android and iOS projects: *PiceaOne*, *PiceaOne (Beta)* and *PiceaOne (Enterprise)*. Each of these application variants has a release and debug configurations to helps the team to debug issues in different application variants. The purpose of different configurations is to manage which features are released and which features are still under testing and development. From different configurations, unnecessary dependencies can be left out at compile time. The configurations for Android project are defined in the Gradle [41] build file as described in the program 1.

```
buildTypes {
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android.txt'),
        'proguard-project.txt'
        signingConfig signingConfigs.release
    }
    debug {
        resValue "string", "application_type", "RD"
    }
```

```
}

flavorDimensions "tier"

productFlavors {
    beta {
        dimension "tier"
        applicationIdSuffix = ".beta"

        buildConfigField 'boolean', 'BETA', "true"
        buildConfigField 'boolean', 'ENTERPRISE', "false"

        resValue "string", "application_id", "XXX"
        resValue "string", "application_type", "Beta"
    }
    store {
        dimension "tier"
        buildConfigField 'boolean', 'BETA', "false"
        buildConfigField 'boolean', 'ENTERPRISE', "false"

        resValue "string", "application_id", "XXX"
        resValue "string", "application_type", "Official"
    }
    enterprise {
        targetSdkVersion 22

        dimension "tier"
        applicationIdSuffix = ".enterprise"

        buildConfigField 'boolean', 'BETA', "false"
        buildConfigField 'boolean', 'ENTERPRISE', "true"

        resValue "string", "application_id", "XXX"
        resValue "string", "application_type", "Official"
    }
}
```

Program 1. *Defining different configurations of PiceaOne application in Gradle build file.*

The configurations for the iOS project are configured from the Xcode GUI. The configurations are shown in figures 5 and 6 below.

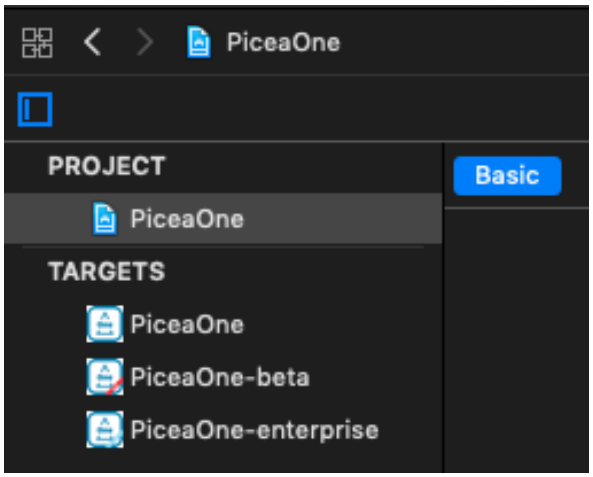


Figure 5. *Target configurations for PiceaOne Xcode project.*

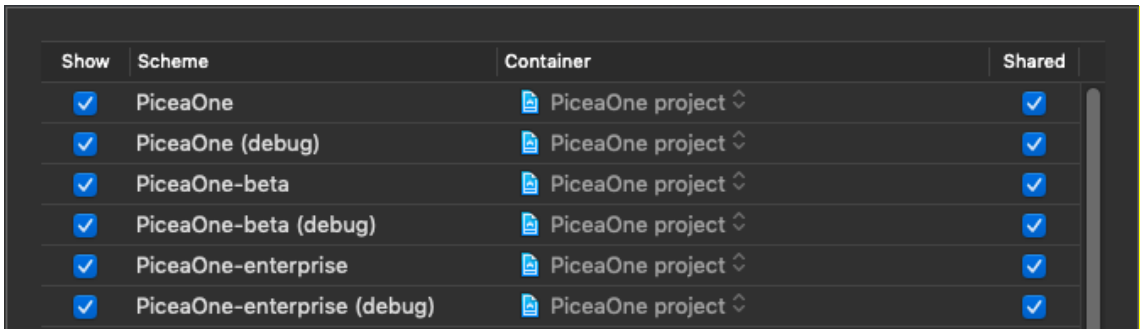


Figure 6. *Build scheme configurations for PiceaOne Xcode project.*

PiceaOne configuration is the default configuration of the applications. It is the official version distributed through Google's Play Store and Apple's App Store to the end-users. This configuration is always built from *release* and *master* branches as described in figure 4 earlier. The *store* flavor declares the configuration in Android Studio project. *PiceaOne* target and scheme declares the configuration in iOS Xcode project.

PiceaOne (Beta) configuration is the configuration for daily builds from the *development* branch. This app configuration is only distributed through internal test tracks. This configuration is never built from any other branch than *development* as described in figure 4 earlier. The *beta* flavor declares the configuration in Android Studio project, and *PiceaOne-beta* target and scheme declare the configuration in iOS Xcode project.

The third configuration for both platforms is the *PiceaOne (Enterprise)* configuration. This configuration is for application variants distributed as part of Piceasoft's PC solution. Application built with this configuration, is built from *development*, *release* and *master* branches and delivered to PC software team's VCS. The source branch of the build defines the target branch in VCS of PC software team. The configuration is declared by the *enterprise* flavor in Android Studio project and by *PiceaOne-enterprise* target and scheme in iOS Xcode project.

5. GITLAB SETUP

Gitlab Enterprise Edition [42] was selected for a Git repository hosting solution. It is free open-core software based on completely open-source Gitlab Community Edition [43]. The Enterprise Edition functions as Community Edition without a license key, still providing an easy update of the subscription in the future [44]. Features that supported the selection of the Gitlab were easy integration to Piceasoft's existing Lightweight Directory Access Protocol (LDAP) authentication system, built-in support for CI/CD pipelines and the fact that the development team was used to work with the Gitlab.

Setting up the Gitlab started with the setup of a dedicated Debian virtual machine in Piceasoft's internal office network. The hostname of the server was set to gitlab.piceasoft.com. After setting up the Dynamic Name System (DNS) records for the machine, Secure Shell (SSH) connection to the server could be established to perform Gitlab Enterprise Edition software basic installation.

5.1 Configuring SSL

After the basic installation, it was time to configure Gitlab to use Hypertext Transfer Protocol Secure (HTTPS). The need for the Transport Layer Security (TLS) / Secure Sockets Layer (SSL) was not critical as the service is only accessible from company's internal network, but most of the modern web browsers display warning for the user when login credentials are sent over non-secured connections. To enable HTTPS for Gitlab instance, a NGINX [44] configuration is required. Gitlab provides an easy and automated setup using free certificates from Let's Encrypt [45] [46]. However, Piceasoft already has a wildcard certificates for the *.piceasoft.com domain; hence the manual setup for the HTTPS was selected as a preferred method.

The setup of the HTTPS was started by modifying the Gitlab instance configuration file from default installation location "/etc/gitlab/gitlab.rb". From the configuration file field "external_url" was changed from http://gitlab.piceasoft.com to https://gitlab.piceasoft.com. After setting the instance Uniform Resource Locator (URL) from Hypertext Transfer Protocol (HTTP) to HTTPS protocol, TSL/SSL certificates were added for the Gitlab instance. Gitlab instance needed the full chain of the certificates to allow Gitlab runners used by the Gitlab's CI system to register from macOS operating systems. The chained certificate "gitlab.piceasoft.com.chained.crt" was created. This certificate included all of the CA certificates after which the Gitlab runners were able to connect the Gitlab instance from macOS platform as well. The certificates were introduced for NGINX instance by adding the "ssl_certificate" and "ssl_certificate_key" to Gitlab's configuration file as the following program 2 shows.

```
nginx['ssl_certificate'] = "/etc/gitlab/ssl/gitlab.piceasoft.com.chained.crt"
nginx['ssl_certificate_key'] = "/etc/gitlab/ssl/gitlab.piceasoft.com.key"
```

Program 2. *Gitlab NGINX certificate configurations*

After configuring the certificates, the SSL was enabled by writing the following parameters from program 3 to Gitlab's configuration file.

```
nginx['redirect_http_to_https'] = true
nginx['ssl_protocols'] = "TLSv1.1 TLSv1.2"
```

Program 3. *Gitlab NGINX SSL configurations*

5.2 LDAP integration

After the following set up the Gitlab instance was up and running and accessible from the *https://gitlab.piceasoft.com* URL. The final step on the setup was to integrate the Gitlab instance with Piceasoft's LDAP server to control access on the Gitlab resources. By integrating the Gitlab instance to the existing LDAP server, the same accounts Piceasoft employees already use to access the internal services can be used to access the resources in Gitlab. Gitlab installation provides tools for the integration out-of-the-box, and the integration can be done by simply providing LDAP instance details for Gitlab instance in Gitlab's configuration file. Piceasoft's Information Technology (IT) department provided needed LDAP service details to allow Gitlab integration and the following configuration from program 4 was written to the configuration file of Gitlab.

```
gitlab_rails['ldap_enabled'] = true
gitlab_rails['ldap_servers'] = {
'main' => {
  'label' => 'Piceasoft account',
  'host' =>  'ldapmaster.piceasoft',
  'port' => 389,
  'uid' => 'uid',
  'encryption' => 'plain',
  'verify_certificates' => false,
  'bind_dn' => 'uid=**********,ou=Admins,dc=piceasoft',
  'password' => '****************',
  'active_directory' => false,
  'base' => 'ou=Users,dc=piceasoft'
  }
}
```

Program 4. *Gitlab instance LDAP integration configuration.*

5.3 Transition from Subversion to Gitlab

The migration from Subversion to Git can be done with several different tools. There is built-in support to migrate codebase from Subversion repository into a Git repository in Git command line tools for example [48]. The Subversion repository, in this case, had a long history which had a very little to do with the mobile applications. The decision was

made not to try a migration of Subversion repository to Git repository. The Subversion repository will be kept in use for the PC / Mac software development; thus, the full history of the mobile applications will be preserved as well. Additionally, the branching of the mobile application projects codebase is messy and does not include any usage of tags. These facts supported the decisions of not running any migrations from the Subversion repository to Git repository.

The transition to Git was done by creating a new project to the Gitlab from the web interface. After the project set up in the Gitlab, the empty repository was cloned to the local machine with the instructions provided in the Gitlab project stub. The empty repository was initialized with README.md file. The steps are declared in program 5 below.

```
git config --global user.name "Henri Salminen"
git config --global user.email henri.salminen@piceasoft.com

git clone git@gitlab.piceasoft.com:hsalminen/test-gitlab-project.git
cd test-gitlab-project
touch README.md
git add README.md
git commit -m "add README"
git push -u origin master
```

Program 5. *Cloning empty Gitlab project repository*

After the Git repository was cloned and was ready to be used in a local machine, the Subversion repository was checked out to the local computer in a different location than the Git repository. Git repository was initialized with a created *.gitignore* file, which tells the Git repository to ignore a particular type of files that do not need to be stored in the repository [48]. After adding the *.gitignore* file to the repository a new branch called *development* was created from the Gitlab web interface. It was marked as a default branch of the project, and the *master* branch was protected from project settings. After checking out the *development* branch on the local machine, the project files were copied from the Subversion repository to the local Git repository, excluding the .svn directory.

5.4 Setting up Gitlab Runners

Gitlab Runner is an open source software that integrates to the Gitlab server installation and executes CI/CD system pipelines from the Gitlab [49]. In the setup, Gitlab installation is referred to a coordinator, which can coordinate multiple Gitlab Runners. The Gitlab Runner software is written in Go programming language, and it can run in Linux, macOS and Windows environments as a self-contained binary without language specific dependencies [49].

The Gitlab Runner can execute jobs coordinated by Gitlab coordinator [50] either locally or by using Docker containers [51]. The Gitlab Runner can be set up as a project-specific

runner which can pick-up jobs only from specific projects, or it can be set up as a shared runner which can pick-up jobs from any project in the Gitlab [50].

In this setup, two Mac Mini machines are used as Gitlab Runner machines. These machines are referred to as *gitlabrunner1* and *gitlabrunner2* in the future. Additionally, there will be a third Mac Mini machine which will be dedicated for running UI automation tests for the application. The runners need to run on macOS platform to be able to build, test and deploy native iOS applications.

5.4.1 Setup Gitlab Runner host machines

Both machines need to be initialized with some software before the environment can run the Gitlab CI/CD pipeline build scripts. Initial macOS setup is performed, and *gitlab* is selected as a primary user for both machines. The important part on the setup is to not enable two-factor authentication for Gitlab's iCloud account. It is documented by the Fastlane build tools which we are going to be using later to build and deploy the applications [52].

For building, testing and deploying Android applications Java development environment and Gradle build tools needs to be installed. The Java Runtime Environment (JRE) and Java Development Kit (JDK) packages were installed to both machines using official Oracle installation packages. After setting up the Java development environment for both machines, Android Studio installation can be started. Android Studio installs required Gradle build tools and can be later on used to debug issues in the build environment or update the Gradle build tools.

For building, testing and deploying iOS applications Xcode and its command line tools needs to be installed. The Xcode was installed on both machines from Apple AppStore for macOS.

5.4.2 Registering Gitlab Runners

First, the Gitlab Runner software needs to be installed on *gitlabrunner1* and *gitlabrunner2* machines. The installation is simple and detailed instructions are provided in the Gitlab's documentation [56].

After the installation, Gitlab Runners need to be registered to the Gitlab coordinator instance to acknowledge it about the new runners. Two runners per host machine were registered; one for PiceaOne iOS project and one for PiceaOne Android project per machine. This way the runners can pick up jobs concurrently and the wait time for the jobs reduces. During the registration details declared in table 1, table 2, table 3 and table 4 were provided for the Gitlab Runner instances.

Table 1. *PiceaOne iOS Gitlab Runner registration parameters for Gitlab Runner 1 host machine.*

Gitlab CI coordinator URL	https://gitlab.piceasoft.com/
Gitlab CI token	XXXXXXXXXXXXXXXX
Runner description	PiceaOne iOS (Gitlab Runner 1)
Runner tags	iOS
Runner executor	shell

Table 2. *PiceaOne iOS Gitlab Runner registration parameters for Gitlab Runner 2 host machine.*

Gitlab CI coordinator URL	https://gitlab.piceasoft.com/
Gitlab CI token	XXXXXXXXXXXXXXXX
Runner description	PiceaOne iOS (Gitlab Runner 2)
Runner tags	iOS
Runner executor	shell

Table 3. *PiceaOne Android Gitlab Runner registration parameters for Gitlab Runner 1 host machine.*

Gitlab CI coordinator URL	https://gitlab.piceasoft.com/
Gitlab CI token	XXXXXXXXXXXXXXXX
Runner description	PiceaOne Android (Gitlab Runner 2)
Runner tags	Android
Runner executor	shell

Table 4. *PiceaOne Android Gitlab Runner registration parameters for Gitlab Runner 2 host machine.*

Gitlab CI coordinator URL	https://gitlab.piceasoft.com/
Gitlab CI token	XXXXXXXXXXXXXXX
Runner description	PiceaOne iOS (Gitlab Runner 2)
Runner tags	Android
Runner executor	shell

After registering the runners, the Gitlab coordinator distributes jobs to the runners based on the project the runner is bind to and the tags the jobs are assigned.

6. CONTINUOUS INTEGRATION, DELIVERY AND DEPLOYMENT

To improve the productivity of the development team the CI/CD system needed to automate the building, testing, and deployment of the mobile applications. The CI/CD system needs to build all variants of the applications, run regression testing, manage the versioning, deploy the applications to the internal testing tracks and deliver the applications to the production tracks for easy deployment.

6.1 Analysis of requirements

Building step of the CI/CD system needs to be able to build every variant of the PiceaOne application. In the future, the CI/CD systems need to build individual components of the applications to framework and library packages as well, but this is not relevant at the time of writing the thesis. The variants of the application are *PiceaOne*, *PiceaOne (Beta)* and *PiceaOne (Enterprise)*. *PiceaOne* variant is the official version of the application, and it communicates with Piceasoft production back-end environment. This official version is distributed through official distribution channels for Android and iOS. *PiceaOne (Beta)* is the R&D variant of the application which communicates with Piceasoft R&D back-end environment. *PiceaOne (Beta)* app is used for testing the new unreleased features before official deployment and is only distributed through beta testing channels. *PiceaOne (Enterprise)* variant is signed for installation via USB connection and is used by Piceasoft PC solutions. *PiceaOne (Enterprise)* application packages are deployed to VCS used by the PC software team.

The testing stage needs to validate the quality of the changed codebase by running unit tests and more comprehensive UI tests for regression. Both Android and iOS development environments provide tools for writing and running unit tests for the software components. Each component is tested with unit tests as individual components detached from the host application. The application combines these different components to the final product and the functionality of the final product is verified with more comprehensive UI testing. These UI tests are implemented with a combination of Robot Framework [59] and Appium server [60] in the future.

Deployment stage needs to automatically deploy the new version of the *PiceaOne (Beta)*. The latest version of the application is needed by the Quality Assurance (QA) team. The QA team needs to be able to install the application to the multiple testing devices easily. *PiceaOne (Enterprise)* variant needs to be deployed to the VCS used by PC software development team. PC development team regularly needs new versions of the applications to integrate mobile applications to PC solution.

6.2 Selecting tools for CI/CD system

Gitlab has excellent built-in support for CI/CD processes [61]. The most significant advantages for the Gitlab CI/CD tools are the integration to the Gitlab installation, low learning curve, and good scaling capabilities with independent Gitlab runners [49].

To build applications automatically in CI/CD system of Gitlab the projects need to support building from the command line. To build the applications from the command line Xcode and Gradle tools can be used. Xcode has xcodebuild command line tool that can be used to build and archive iOS applications. For the Android, Gradle build tools are the preferred way to build apps [62].

Google offers API to upload application binaries to the Google Play Store and to distribute them for testers. To upload iOS application binaries to the AppStore Connect is a more complicated task as Apple does not provide any documented way to do this from the automated environment. Within the Xcode installation comes the *altool* command line tool which can be used to upload binaries to the AppStore Connect [63]. However, the official documentation for this tool was not available at the time of the writing this thesis.

To make it easier for developers, an open source project called fastlane has been developed by a broad community of developers [64]. *Fastlane* is a platform which aims to simplify the deployment of Android and iOS applications [64]. It provides a set of tools written in Ruby [65] for building, testing, signing and deploying the applications. The project has a vast community of developers behind it and at the time writing the thesis it had over 880 contributors and almost 15 000 commits in is public Github repository [66]. The tools provided by *fastlane* were so comprehensive that it was selected as a toolset for implementing the CI/CD scripts to support the Gitlab's CI/CD pipeline scripts.

6.2.1 Setup accounts

To build, test and deploy Android and iOS applications Google and iCloud accounts are needed. Delivery of the Android applications is done with Google Play Store Console web service, and delivery of the iOS applications is done with App Store Connect and Apple Developer Portal web services. The CI/CD system needs to have both of these accounts with correct access rights to be able to deliver the applications. To set up these accounts, Piceasoft IT department provided a specific @piceasoft.com email account. With this email account, both Google and iCloud accounts were created.

To provide access to Google Play Console for CI/CD scripts API access needs to be enabled. Fastlane tools are used to communicate with this API, and the Fastlane documentation provides excellent instructions on how to set up the API access [67]. By

following the instructions, the API access is granted for the build scripts and the binary upload to the Google Play Console is possible.

The created iCloud user is added to the development team in the Apple Developer Portal to grant access to the App Store Connect services. Additionally, the account is added to the App Store Connect service with “App Manager” role. This way the account that CI/CD system uses can access the required provisioning profiles and APIs that are needed during the building, testing, signing and deploying the applications.

6.3 Relationship between branching model and configuration management

The CI/CD system will use the *development* branch to continuously build, test and deliver alpha versions of the application which will only be delivered to the test department internally. The pipeline for the CI/CD system can be triggered every time the codebase in the *development* branch changes. In this way, all of the new features committed to the *development* branch are instantly tested by the test automation and available for the other developers and testing department. For this purpose, the *PiceaOne (Beta)* application variant is used. From the *development* branch, only the *PiceaOne (Beta)* application variant is built, tested and distributed.

Once all of the features for the upcoming release are completed and *release* branch is created, CI/CD system can start building beta builds of official *PiceaOne* applications from the *release* branch and distribute the official builds through beta testing channels. Finally, when the *release* branch is merged into the *master* branch the CI/CD system can build an official *PiceaOne* application and deliver it to the production. The relationship between branches and application variants is described in figure 7 below.

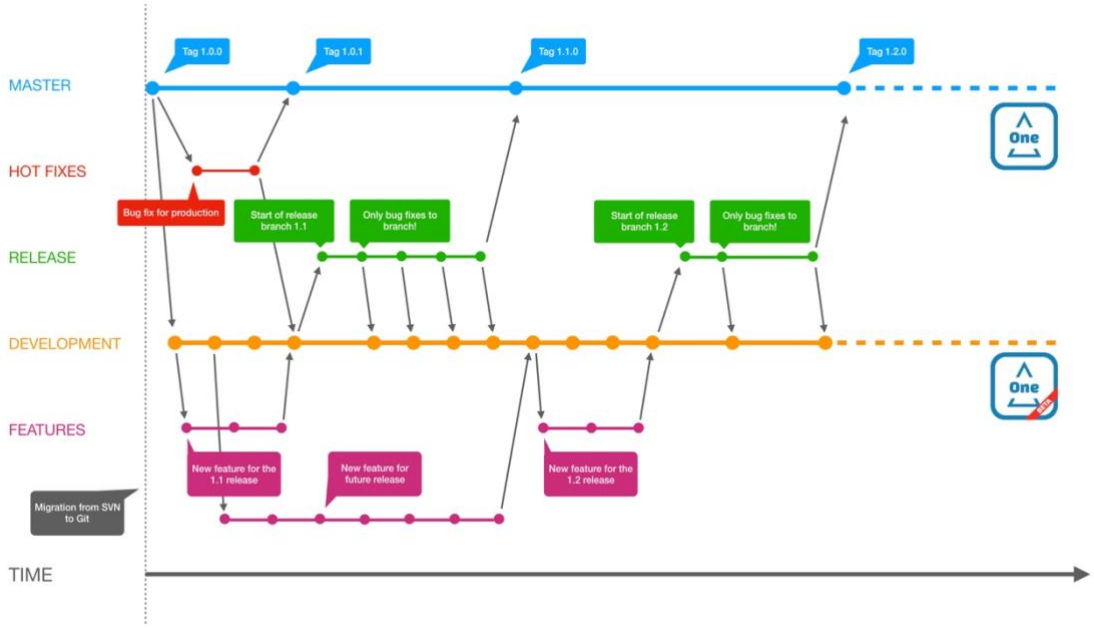


Figure 7. *Relationship between branches and application variants.*

6.4 Implementation of the CI/CD system

Figure describing the CI/CD pipeline is declared in appendix E.

Implementation of the created system consists of two layers. The second layer provides an abstraction for complicated tasks of managing versioning, testing, building, and deployment of the project. The second layer is implemented with *fastlane* by defining tasks as *lanes* in *Fastfile*. Each of these *lanes* implements some task related to the CI/CD system. The first layer uses the *lanes* provided by the second layer to manage the configurations of the applications. The first layer integrates seamlessly to selected VCS as the layer is implemented with Gitlab CI/CD tools. Gitlab's CI/CD pipeline is described by *.gitlab-ci.yml* file. In this file, multiple stages or jobs are configured for registered Gitlab runners to pick up and execute. Each job can be defined to be executed conditionally. To complete the CI/CD system multiple different *lanes* needs to be implemented to support Gitlab CI/CD pipeline configuration file. The layers are defined in figure 8 below.

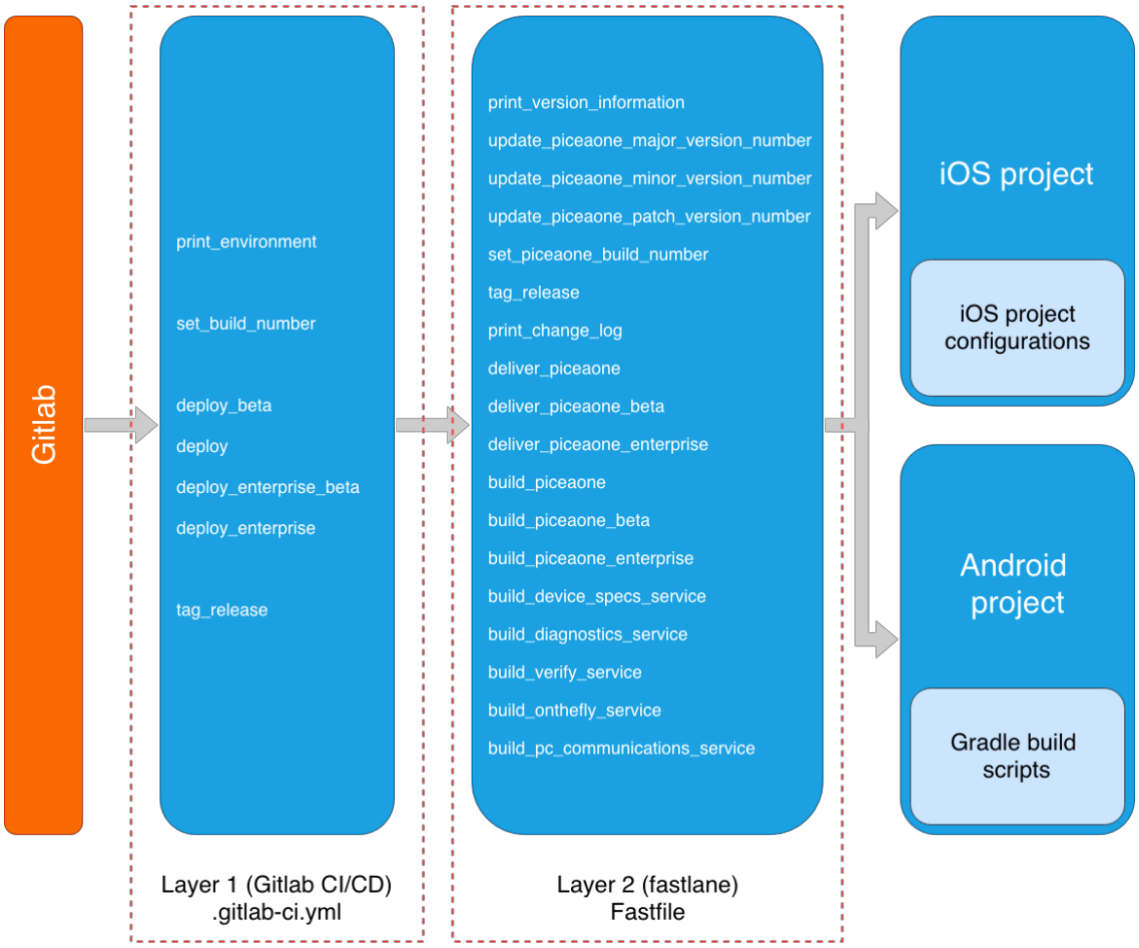


Figure 8. *Layers defined in CI/CD system implementation.*

6.4.1 Managing versioning

To manage the PiceaOne applications versioning the CI/CD pipeline in Gitlab is configured to include a *set_build_number* stage which will update the PiceaOne application build number when the new build is triggered. Build number needs to increment every time the application is deployed to the AppStore Connect or Google Play Store service. The build number is set based on Gitlab CI/CD pipeline variable *CI_PIPELINE_IID*. This variable provides project-specific incrementing integer which is ideal for build number of the applications [68]. The stage is defined to be executed for all main branches in the branching model; *release, development*, and *master*. The stage for setting the build number is declared in the programs 6 and 7 below.

```
################################################################################
# Incrementing build number
################################################################################

set_build_number:
  stage: set_build_number
  tags:
    - Android
  script:
    - |
      bundle exec fastlane set_build_number \
      build_number:$CI_PIPELINE_IID         \
  only:
    - /^release\/.*|^master$|^development$/
```

Program 6. *set_build_number stage for Android in Gitlab CI/CD pipeline configuration file.*

```
################################################################################
# Incrementing build number
################################################################################

set_build_number:
  stage: set_build_number
  tags:
    - iOS
  script:
    - |
      bundle exec fastlane set_build_number \
      build_number:$CI_PIPELINE_IID         \
  only:
    - /^release\/.*|^master$|^development$/
```

Program 7. *set_build_number stage for iOS in Gitlab CI/CD pipeline configuration file.*

The Programs 14 and 15 above defined the Gitlab's CI/CD pipeline stage for setting the build number for the build the CI/CD system is building. The program starts *fastlane* lane

named *set_build_number*. This lane is defined in the *fastlane* configuration file called *Fastfile*. To manage versioning four different lanes are implemented to *fastlane Fastfile*.

1. update_major_version_number
2. update_minor_version_number
3. update_patch_version_number
4. set_build_number (options)

The first one updates the application major version number and resets both minor and patch versions. The second one updates the application minor version number and resets the patch version number. The third one updates the application patch version number. The last one sets the build number to the build number provided for the lane in options. The version update is then committed to the Git repository and finally pushed to the remote repository. For iOS platform fastlane provides commands for updating version numbers out of the box: *incement_version_number* [69], *commit_version_bump* [70] and *push_to_git_remote* [71]. The iOS implementation is declared in the program 8 below.

```
def update_version_number_major
  increment_version_number(
    bump_type: "major",
    xcodeproj: "PiceaOne.xcodeproj"
  )
end
def update_version_number_minor
  increment_version_number(
    bump_type: "minor",
    xcodeproj: "PiceaOne.xcodeproj"
  )
end
def update_version_number_patch
  increment_version_number(
    bump_type: "patch",
    xcodeproj: "PiceaOne.xcodeproj"
  )
end
def set_build_number(build_number)
  increment_build_number(
    xcodeproj: "PiceaOne.xcodeproj",
    build_number: build_number
  )
end

def commit_version_update
  commit_version_bump(
    xcodeproj: "PiceaOne.xcodeproj",
    message: "Updated version to #{get_version} [skip ci]"
  )
end

desc "Update major version number"
lane :update_major_version_number do
  update_version_number_major
  commit_version_update
```

```
  push_to_git_remote
end

desc "Update minor version number"
lane :update_minor_version_number do
  update_version_number_minor
  commit_version_update
  push_to_git_remote
end

desc "Update patch version number"
lane :update_patch_version_number do
  update_version_number_patch
  commit_version_update
  push_to_git_remote
end

desc "Update build number"
lane :update_build_number do
  update_build_number
  commit_version_update
  push_to_git_remote
end

desc "Set build number"
lane :set_build_number do |options|
  set_build_number(options[:build_number])
  commit_version_update
  push_to_git_remote
end
```

Program 8. *Updating version numbers in iOS project using Fastlane.*

For Android, the same kind of complete commands are not provided by the *fastlane* tools. The versioning of the Android project is completed with the help of Gradle build tools. Version information of the Android applications is stored in the repository in file called *version.properties* which is readable and writable for the Gradle build tools. In the Gradle build scripts the following methods are define and implemented.

1. readVersionPropertiesFile
2. readVersionProperties
3. getMajorVersion
4. getMinorVersion
5. getPatchVersion
6. getBuildNumber
7. readVersionCode
8. incrementMajorVersion
9. incrementMinorVersion
10. incrementPatchVersion
11. setBuildNumber(buildNumber)

The implementation of these methods is provided in the appendix B. After declaring the methods above following tasks are implemented in the Gradle build script.

1. incrementMajorVersionNumber
2. incrementMinorVersionNumber
3. incrementPatchVersionNumber
4. setBuildNumber

The implementation of these Gradle tasks is declared in the following program 9.

```
task('incrementMajorVersionNumber') {
    doLast {
        println "Increment major version number..."
        incrementMajorVersion()
    }
}
task('incrementMinorVersionNumber') {
    doLast {
        println "Increment minor version number..."
        incrementMinorVersion()
    }
}
task('incrementPatchVersionNumber') {
    doLast {
        println "Increment patch version number..."
        incrementPatchVersion()
    }
}
task('setBuildNumber') {
    doLast {
        println "Set build number to: ${buildnumber} / " + buildnumber
        setBuildNumber(buildnumber)
    }
}
```

Program 9. *Managing Android version numbers in Gradle build scripts.*

With these Gradle tasks, it is trivial to implement similar *fastlane* command for Android project that the *fastlane* provides for iOS out of the box. Same versioning lanes were implemented to the Android project *Fastfile* that were implemented for iOS.

1. update_major_version_number
2. update_minor_version_number
3. update_patch_version_number
4. set_build_number

The functionality of these lanes is the same as for the iOS project, but the implementation differs as we need to use Gradle build scripts to manage the versioning. Additionally, *fastlane* does not provide a command for committing Android version update to the repository so the plain *git_commit* command can be used directly [72]. Implementation of the lanes is defined in the following program 10.

```
def update_version_number_major
  gradle(
    task: 'incrementMajorVersionNumber'
  )
end
def update_version_number_minor
  gradle(
    task: 'incrementMinorVersionNumber'
  )
end
def update_version_number_patch
  gradle(
    task: 'incrementPatchVersionNumber'
  )
end
def set_build_number(build_number)
  gradle(
    task: 'setBuildNumber',
    properties: {
      'buildnumber' => build_number
    }
  )
end

def commit_version_update
  git_commit(
    path: "./version.properties",
    message: "Updated PiceaOne version to #{get_version} [skip ci]"
  )
end

desc "Update major version number"
lane :update_major_version_number do
  update_version_number_major
  commit_version_update
  push_to_git_remote
end

desc "Update minor version number"
lane :update_minor_version_number do
  update_version_number_minor
  commit_version_update
  push_to_git_remote
end

desc "Update patch version number"
lane :update_patch_version_number do
  update_version_number_patch
  commit_version_update
  push_to_git_remote
end

desc "Update build number"
lane :update_build_number do
  update_build_number
  commit_version_update
  push_to_git_remote
end
```

```
desc "Set build number"
lane :set_build_number do |options|
  set_build_number(options[:build_number])
  commit_version_update
  push_to_git_remote
end
```

Program 10. *Updating version numbers in Android project using Fastlane.*

With this implementation, the build number of the PiceaOne applications are automatically managed by the CI/CD system. The development team manages the major, minor and patch version updates, but the update is easily carried out by using the implemented *fastlane* scripts.

6.4.2 Building

The next step is to provide simple commands for building the applications. To build all the variants of the PiceaOne application the following lanes are defined to the *Fastfile*.

1. build_piceaone
2. build_piceaone_beta
3. build_piceaone_enterprise

These lanes build the different variants of the PiceaOne application and conveniently copy the build outputs to the "./Outputs" directory in the project's root directory as described in figure 9.

```
 | salminen : ~/Repositories/Gitlab/PiceaOne-iOS  tree ./Outputs/
./Outputs/
├── Beta
|   ├── PiceaOne-iOS-Beta-information.json
|   ├── PiceaOne-iOS-Beta.app.dSYM.zip
|   └── PiceaOne-iOS-Beta.ipa
├── Enterprise
|   ├── PiceaOne-iOS-Enterprise-information.json
|   ├── PiceaOne-iOS-Enterprise.app.dSYM.zip
|   └── PiceaOne-iOS-Enterprise.ipa
└── Official
    ├── PiceaOne-iOS-information.json
    ├── PiceaOne-iOS.app.dSYM.zip
    └── PiceaOne-iOS.ipa
```

Figure 9. *Outputs of the fastlane build scripts for iOS application variants.*

Each of the subfolders contains necessary files related to a certain variant of the application. The JSON files include version information of the app. dSYM packages include application binary debug symbols to deobfuscate crash reports stack traces [73]. The last file is the IPA package of the application that can be distributed through multiple

delivery channels to iOS devices. Android build scripts produce similar file structure to the root of the repository as declared in figure 10.

```
 | salminen : ~/Repositories/Gitlab/PiceaOne-Android  tree ./Outputs/
./Outputs/
├── Beta
|   ├── PiceaOne-Android-Beta-information.json
|   └── PiceaOne-Android-Beta.apk
├── Enterprise
|   ├── PiceaOne-Android-Enterprise-information.json
|   └── PiceaOne-Android-Enterprise.apk
└── Official
    ├── PiceaOne-Android-information.json
    └── PiceaOne-Android.apk
```

Figure 10. *Outputs of the fastlane build scripts for Android application variants.*

As the iOS build scripts, the Android build scripts produce JSON files that includes information about the application as well as the application package itself. The implementation of these *fastlane* build scripts are defined in the appendixes B and C. The *lanes* for building the applications are not used directly in the Gitlab's CI/CD pipeline, but the deployment and delivery lanes use these lanes to build the deliverables.

6.4.3 Testing

Testing stages run tests for the applications to make sure the commit triggering the pipeline did not break any functionality of the software. In the *unit_tests* stage, all of the applications and its components' unit tests are executed by firing the testing lanes from *Fastfile*. In the *ui_tests* stage, more complete and time-consuming tests are performed. UI tests have own shell scripts for running and require a specific environment with dedicated devices to work correctly. To accomplish this, a dedicated Gitlab runner can be set up. The unit tests are executed for all of the main branches and commits while UI tests are executed only nightly triggered CI/CD pipelines. The testing in detail is out of scope for this thesis, and only the crucial parts were described.

6.4.4 Code signing

To be able to distribute Android and iOS applications the applications must be code signed before they can be delivered to the Google Play Store or Apple App Store. Once the codebase is built in the CI/CD system the deliverable .apk and .ipa packages are created.

To code sign the PiceaOne Android application, signing configuration is defined to application level Gradle build script in Android Studio project. Signing configuration

needs the keystore file, the keystore password, the key alias and the key password. The keystore file is saved on the machines that are running gitlab runner processes for the CI/CD system. Rest of the values are read from environment variables set by Gitlab CI script. The Gradle script implementation is defined in the program 11 below.

```
signingConfigs {
    release {
        storeFile = file('../config/piceasoft.keystore')

        if(System.getenv("KEYSTORE_PASSWORD")) {
            storePassword System.getenv("KEYSTORE_PASSWORD")
        }

        if(System.getenv("KEY_ALIAS")) {
            keyAlias System.getenv("KEY_ALIAS")
        }

        if(System.getenv("KEY_PASSWORD")) {
            keyPassword System.getenv("KEY_PASSWORD")
        }
    }
}
```

Program 11. *Setting code signing configurations for PiceaOne Android application build script.*

File *piceasoft.keystore* is saved to the gitlab runner host machines and copied to project level config directory during the start of CI/CD pipeline to make sure that the Gradle script has access to it. Rest of the code signing values are stored as Gitlab project level secret variables [74], and CI/CD pipeline script sets them to the environment variables before the Gradle script is triggered. The CI/CD pipeline variables are set as the following program 12 describes.

```
################################################################################
# CI Pipeline for PiceaOne (Android)
################################################################################

variables:
  LANG              : "en_US.UTF-8"
  LC_ALL            : "en_US.UTF-8"
  GIT_STRATEGY      : clone
  KEYSTORE_PASSWORD : $KEYSTORE_PASSWORD
  KEY_ALIAS         : $KEY_ALIAS
  KEY_PASSWORD      : $KEY_PASSWORD
```

Program 12. *Setting PiceaOne Android signing config secrets to environment variables at the start of CI/CD pipeline execution.*

The keystore file is copied in *before_script* stage in the *.gitlab-ci.yml* file to make sure the keystore file is available in the working directory of CI/CD pipeline. The *before_script* stage is described in the program 13 below.

```
################################################################################
```

```
# Setting up the environment
################################################################################

before_script:
  # Install bundler and install gem dependecies (fastlane):
  - sudo gem install bundler && sudo bundle install

  # Configure git:
  - git config --global user.email "$GIT_CI_EMAIL"
  - git config --global user.name "$GIT_CI_NAME"

  # Checkout branch and fetch tags to be able to commit version bump:
  - git status
  - git fetch
  - git checkout $CI_COMMIT_REF_NAME
  - git branch
  - git remote set-url origin git@gitlab.piceasoft.com:mobile/PiceaOne-
Android.git
  - git fetch --tags

  # Give gradle wrapper execution permission:
  - chmod +x ./gradlew

  # local.properties has to be generated as Gradle does not support ANDROID_HOME
  # environment variable despite the documentation...
  - echo "sdk.dir=/Users/$USER/Library/Android/sdk" > $PWD/local.properties

  # Create config directory
  - mkdir $PWD/config

  # Copy all required configurations to working directory from user's home
  # directory:
  - cp ~/Configurations/PiceaOne-Android/piceasoft.keystore
$PWD/config/piceasoft.keystore
  - cp ~/Configurations/PiceaOne-Android/google-credintials.json
$PWD/config/google-credintials.json
  - cp ~/Configurations/PiceaOne-Android/PiceaOne/google-services.json
$PWD/app/google-services.json
  - cp ~/Configurations/PiceaOne-Android/PiceaOne-Beta/google-services.json
$PWD/app/src/beta/google-services.json
  - cp ~/Configurations/PiceaOne-Android/PiceaOne-Enterprise/google-
services.json $PWD/app/src/enterprise/google-services.json
```

Program 13. *Setting up Android environment before each CI/CD stage.*

Code signing PiceaOne iOS application can be managed by the Xcode entirely automatically. In this setup, more control over code signing was required, and it was achieved by creating and defining provisioning profiles for each build variant manually. The provisioning profiles that are used for code signing iOS applications were created from the Apple Developer portal [75], downloaded and installed to the machines running CI/CD pipeline jobs and defined for *Fastfile* so that the command line build knows which provisioning profile to use. Additionally, the signing certificates that were attached to the provisioning profiles needed to be installed to the machines running the build tasks. Following program 14 declares the provision profile configuration in *Fastfile.*

```
# Build the PiceaOne app
build_app(
  workspace: "PiceaOne.xcworkspace",
  scheme: "PiceaOne",
  output_name: "PiceaOne-iOS.ipa",
  output_directory: "./Outputs/Official",
  export_method: "app-store",
  export_options: {
    provisioningProfiles: {
      "com.piceasoft.piceaone2" => "PiceaOne App Store distribution profile",
      "com.piceasoft.piceaone-beta" => "PiceaOne (Beta) App Store distribution
profile",
      "com.piceasoft.piceaone-enterprise" => "PiceaOne Enterprise distribution
profile"
    }
  }
)
```

Program 14. *Setting provisioning profiles for PiceaOne iOS application code signing in Fastlane build script.*

6.4.5 Deployment and delivery

For deployment, two different jobs are declared in the Gitlab CI/CD pipeline configuration. The first one is executed only for *development* branch while the other one is executed for *release* and *master* branches. By declaring the jobs this way, the pipeline can act differently depending on the source branch. The pipeline can build and deliver *PiceaOne (Beta)* application variant from the *development* branch and build and deliver official *PiceaOne* application variant from *release* and *master* branches. The stage only triggers a different lane from the *Fastfile*. Deploy stage is declared in the programs 15 and 16 below.

```
################################################################################
# Deployment
################################################################################

deploy_beta:
  stage: deploy
  tags:
    - Android
  script:
    - |
      bundle exec fastlane deliver_piceaone_beta  \
      ci_commit_sha:$CI_COMMIT_SHA                \
      ci_commit_before_sha:$CI_COMMIT_BEFORE_SHA  \
  artifacts:
    paths:
      - Outputs/
  only:
    - /^development$/

deploy:
  stage: deploy
  tags:
```

```
    - Android
  script:
    - |
      bundle exec fastlane deliver_piceaone       \
      ci_commit_sha:$CI_COMMIT_SHA                \
      ci_commit_before_sha:$CI_COMMIT_BEFORE_SHA  \
  artifacts:
    paths:
      - Outputs/
  only:
    - /^release\/.*|^master$/
```

Program 15. *deploy stage for Android in Gitlab CI/CD pipeline configuration file.*

```
################################################################################
# Deployment
################################################################################

deploy_beta:
  stage: deploy
  tags:
    - iOS
  script:
    - |
      bundle exec fastlane deliver_piceaone_beta  \
      ci_commit_sha:$CI_COMMIT_SHA                \
      ci_commit_before_sha:$CI_COMMIT_BEFORE_SHA  \
  artifacts:
    paths:
      - Outputs/
  only:
    - /^development$/

deploy:
  stage: deploy
  tags:
    - iOS
  script:
    - |
      bundle exec fastlane deliver_piceaone       \
      ci_commit_sha:$CI_COMMIT_SHA                \
      ci_commit_before_sha:$CI_COMMIT_BEFORE_SHA  \
  artifacts:
    paths:
      - Outputs/
  only:
    - /^release\/.*|^master$/
```

Program 16. *deploy stage for iOS in Gitlab CI/CD pipeline configuration file.*

The *deploy_enterprise* stage is designed in the same way as the *deploy* stage by defining two different jobs that are executed depending on the branch. Both of the jobs build and deliver the *PiceaOne (Enterprise)* application variant, but the target branch in the Subversion is different. The details of the Subversion are provided to the delivery script via Gitlab CI/CD pipeline secret variables that can be set for project scope [74]. This way

the secrets do not need to be kept in the repository and the risk of compromising them reduces. The *deploy_enterprise* stage is declared in the programs 17 and 18 below.

```
################################################################################
# Deployment (Enterprise)
################################################################################

deploy_enterprise_beta:
  stage: deploy_enterprise
  tags:
    - Android
  script:
    - |
      bundle exec fastlane deliver_piceaone_enterprise  \
      ci_commit_sha:$CI_COMMIT_SHA                      \
      ci_commit_before_sha:$CI_COMMIT_BEFORE_SHA        \
      svn_url:$CI_SVN_URL                               \
      svn_checkout_path:$CI_SVN_CHECKOUT_PATH           \
      svn_path:$CI_SVN_PATH                             \
      svn_user:$CI_SVN_USER                             \
      svn_pw:$CI_SVN_PW                                 \
      copy_from:$CI_COPY_FROM                           \
  artifacts:
    paths:
      - Outputs/
  only:
    - /^development$/

deploy_enterprise:
  stage: deploy_enterprise
  tags:
    - Android
  script:
    - |
      bundle exec fastlane deliver_piceaone_enterprise  \
      ci_commit_sha:$CI_COMMIT_SHA                      \
      ci_commit_before_sha:$CI_COMMIT_BEFORE_SHA        \
      svn_url:$CI_SVN_URL_BRANCH                        \
      svn_checkout_path:$CI_SVN_CHECKOUT_PATH_BRANCH    \
      svn_path:$CI_SVN_PATH_BRANCH                      \
      svn_user:$CI_SVN_USER                             \
      svn_pw:$CI_SVN_PW                                 \
      copy_from:$CI_COPY_FROM                           \
  artifacts:
    paths:
      - Outputs/
  only:
    - /^release\/.*|^master$/
```

Program 17. *deploy_enterprise stage for Android in Gitlab CI/CD pipeline configuration file.*

```
################################################################################
# Deployment (Enterprise)
################################################################################

deploy_enterprise_beta:
  stage: deploy_enterprise
```

```
  tags:
    - iOS
  script:
    - |
      bundle exec fastlane deliver_piceaone_enterprise  \
      ci_commit_sha:$CI_COMMIT_SHA                      \
      ci_commit_before_sha:$CI_COMMIT_BEFORE_SHA        \
      svn_url:$CI_SVN_URL                               \
      svn_checkout_path:$CI_SVN_CHECKOUT_PATH           \
      svn_path:$CI_SVN_PATH                             \
      svn_user:$CI_SVN_USER                             \
      svn_pw:$CI_SVN_PW                                 \
      copy_from:$CI_COPY_FROM                           \
  artifacts:
    paths:
      - Outputs/
  only:
    - /^development$/

deploy_enterprise:
  stage: deploy_enterprise
  tags:
    - iOS
  script:
    - |
      bundle exec fastlane deliver_piceaone_enterprise  \
      ci_commit_sha:$CI_COMMIT_SHA                      \
      ci_commit_before_sha:$CI_COMMIT_BEFORE_SHA        \
      svn_url:$CI_SVN_URL_BRANCH                        \
      svn_checkout_path:$CI_SVN_CHECKOUT_PATH_BRANCH    \
      svn_path:$CI_SVN_PATH_BRANCH                      \
      svn_user:$CI_SVN_USER                             \
      svn_pw:$CI_SVN_PW                                 \
      copy_from:$CI_COPY_FROM                           \
  artifacts:
    paths:
      - Outputs/
  only:
    - /^release\/.*|^master$/
```

Program 18. *deploy_enterprise stage for iOS in Gitlab CI/CD pipeline configuration file.*

7. EVALUATION

The development environment designed and implemented in this thesis has been running for 6 months before writing the evaluation of the system. Overall, the development team has been delighted with the improvements that the system brought to daily development. The new VCS is more complex with branches and access controls, but as the team was already familiar working with the Git, pitfalls were quickly identified and overcome. Before writing this evaluation, around 10 official releases have been released from this new system, and everything has been working smoothly. Workflows are way better compared to the old system, and the team can release critical fixes in much more flexible manners.

The major problems with the old setup in Piceasoft were the lack of a proper branching model in the VCS and missing CI/CD system. This thesis tried to solve these issues by designing and implementing a new branching model and CI/CD system for Piceasoft's mobile application development team.

The VCS system and especially the branching model and processes built with it are proven to be functional. Before the upcoming release, the team is now able to start working with the features with zero headaches and without fear of affecting to the release. The use of release branches has improved the released software quality as the testing department has been able to focus on testing the new functionality instead of regression after the creation of the release branch.

Build automation, and automatic deploy and delivery have saved a significant amount of time and increased the versions available for testing department dramatically. The testing department is now getting the new version of the application automatically whenever when the developer completes the feature and publishes the code to the central repository. With the old setup the testing department did not receive new builds of the application even weekly. With the new system in place, multiple versions are deployed to the testing tracks daily. It has helped a lot on running the scrum ideology in software development. The following table in figure 11 shows the increase in deployed alpha versions of the *PiceaOne (Beta)* iOS application before and after the implementation of this development environment.

Release	First build date	Last build date	Builds	Days	Builds / day	Average builds per day
1.1.0	01/02/2018	15/05/2018	14	104	0,13	0,22
1.2.0	15/05/2018	12/06/2018	9	29	0,31	
1.2.1	28/06/2018	03/08/2018	38	37	1,03	1,56
1.2.2	10/08/2018	05/09/2018	25	27	0,93	
1.2.3	05/09/2018	10/09/2018	4	6	0,67	
4.3.0	10/09/2018	03/10/2018	33	24	1,38	
4.3.1	08/10/2018	22/10/2018	14	15	0,93	
4.3.2	23/10/2018	31/10/2018	9	9	1,00	
4.3.3	01/11/2018	01/11/2018	4	1	4,00	
4.3.4	01/11/2018	29/11/2018	18	29	0,62	
4.4.0	03/12/2018	13/01/2019	73	42	1,74	
	Increase of deployed alpha builds per day in percents					700 %

Figure 11. *Increase of alpha build during release development cycle before and after new development environment.*

From the table in figure 15 above can be seen that the builds per day ratio during the development cycle has increased **700 %** after the CI/CD system was brought into use during the 1.2.1 release. In the table, red cells mark the data that was collected before the new system and green rows the data that was collected after the new system was taken into use. When calculating the average count of builds per day only Piceasoft quartal major release values were used including releases 1.1.0, 1.2.0, 4.3.0 and 4.4.0. The release number was bumped up after 1.2.3 release to follow up the release numbers of Piceasoft PC releases to which PiceaOne app releases are loosely tied. The same data is presented in the following figure 12 below.

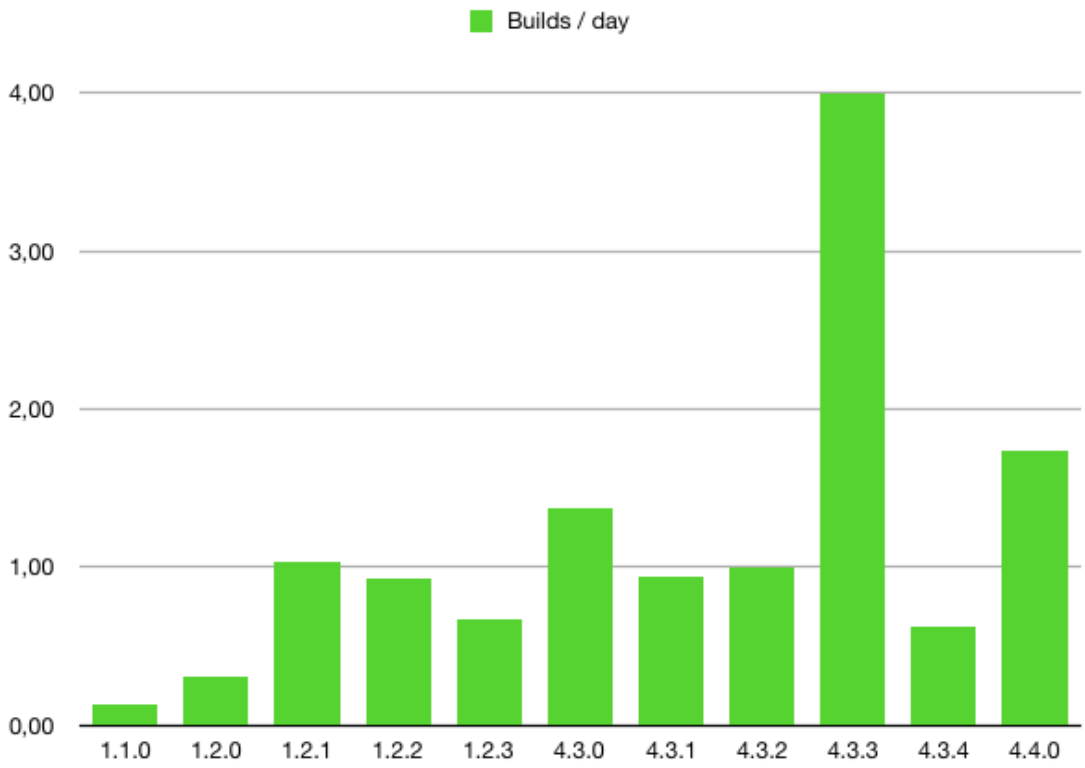


Figure 12. *Deployed alpha release counts for PiceaOne (Beta) application.*

Another method used for evaluation of the new development environment was surveys arranged for teams in Piceasoft. Customized surveys were created for the mobile application development team, the PC development team, and the quality assurance team. The purpose of these surveys was to determine if the new system has improved the software creation processes in Piceasoft. Additionally, PC team developers were asked to provide feedback regarding the PC team transition from Subversion to Git in the future. Survey results are presented in appendixes F, G, and H.

The feedback from the teams and the data from figures 15 and 16 above support the fact that the system designed and implemented in this thesis improved the mobile development environment significantly. The system was so well-functioning that Piceasoft made a decision to switch all projects from Subversion to Gitlab and implement proper CI/CD pipelines. The transition is ongoing during the finalize of this thesis and experiences are positive.

8. CONCLUSION

In this thesis a complete version control system, branching model, continuous integration, continuous delivery, and continuous deployment system was designed and implemented. The target was to improve the development environment for the mobile application development team by providing a functioning version control system and a successful branching model. Additionally, the goal was to reduce the amount of required manual effort when integrating, testing and releasing a new version of the mobile applications.

The solution implemented in this thesis included the transition from the Subversion version control system to a more modern version control system Git. The open source community edition of Gitlab was selected as a tool for hosting and managing the projects and Git repositories. To the Gitlab projects project specific CI/CD pipelines were created to automatically version, build, test and deploy & deliver the mobile applications. The implementation of a proper CI/CD pipelines for the projects significantly reduced the time the development team needed to use for integrating and deploying the solutions.

To support the implementation of CI/CD pipelines in the Gitlab fastlane tools were taken into use to provide a comprehensive set of tools for CI/CD pipeline system to manage the mobile applications. The selected tools turned out to be a superior choice for automating tasks related to mobile application development.

In the future, the mobile application development environment could be expanded to support Piceasoft's mobile SDK story better by adding a capability to manage individual software components as well as the complete mobile applications. The CI/CD system could manage the versioning, testing, building, and delivery of the SDK components in the same way it handles the tasks for the PiceaOne applications. Additionally, more comprehensive testing integration is ongoing for the created CI/CD system.

9. REFERENCES

[1] A. A. Forni and R. van der Meulen, "Gartner Says Worldwide Sales of Smartphones Grew 9 Percent in First Quarter of 2017," Gartner Inc., [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2017-05-23-gartner-says-worldwide-sales-of-smartphones-grew-9-percent-in-first-quarter-of-2017. [Accessed 20 September 2018].

[2] "Getting Started - About Version Control," Software Freedom Conservancy, [Online]. Available: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control. [Accessed 4 February 2019].

[3] "What is Version Control System," Atlassian, [Online]. Available: https://www.atlassian.com/git/tutorials/what-is-version-control. [Accessed 28 December 2018].

[4] "IDE (integrated development environment)," Gartner Inc., [Online]. Available: https://www.gartner.com/it-glossary/ide-integrated-development-environment/. [Accessed 4 February 2019].

[5] V. Shinde, "What is End to End Testing and How to Perform It (Quick Guide)," SoftwareTestingHelp.com, [Online]. Available: https://www.softwaretestinghelp.com/what-is-end-to-end-testing/. [Accessed 31 December 2018].

[6] "SDK (software development kit)," Gartner Inc., [Online]. Available: https://www.gartner.com/it-glossary/sdk-software-development-kit. [Accessed 31 December 2018].

[7] "Create an Android library," Google Inc., [Online]. Available: https://developer.android.com/studio/projects/android-library. [Accessed 9 January 2019].

[8] "What are Frameworks?," Apple Inc., [Online]. Available: https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPFrameworks/Concepts/WhatAreFrameworks.html#//apple_ref/doc/uid/20002303-BBCEIJFI. [Accessed 9 January 2019].

[9] "Android," Google Inc., [Online]. Available: https://www.android.com. [Accessed 31 December 2018].

[10] A. Community, "ART and Dalvik," Google Inc., [Online]. Available: https://source.android.com/devices/tech/dalvik#AOT_compilation. [Accessed 6 February 2019].

[11] A. Thakur, "Difference between Dalvik and ART runtimes in Android," [Online]. Available: http://opensourceforgeeks.blogspot.com/2015/02/difference-between-dalvik-and-art.html. [Accessed 16 July 2018].

[12] Maxim Shafirov, "Kotlin on Android. Now official," [Online]. Available: https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/. [Accessed 16 July 2018].

[13] "Android NDK," Google Developers, [Online]. Available: https://developer.android.com/ndk/. [Accessed 31 December 2018].

[14] "iOS," Apple Inc., [Online]. Available: https://developer.apple.com/ios/. [Accessed 31 December 2018].

[15] "Google Play Store," Google Inc., [Online]. Available: https://developer.android.com/distribute/google-play/. [Accessed 31 December 2018].

[16] "Sign your app," Google Inc., [Online]. Available: https://developer.android.com/studio/publish/app-signing. [Accessed 31 December 2018].

[17] "Manage your app signing keys," Google Inc., [Online]. Available: https://support.google.com/googleplay/android-developer/answer/7384423. [Accessed 31 December 2018].

[18] "Xcode 10," Apple Inc., [Online]. Available: https://developer.apple.com/xcode/. [Accessed 31 December 2018].

[19] "Application Loader overview," Apple Inc., [Online]. Available: https://help.apple.com/itc/apploader/#/apdS673accdb. [Accessed 31 December 2018].

[20] "App Store Find the apps you love. And the ones you're about to.," Apple Inc., [Online]. Available: https://www.apple.com/lae/ios/app-store/. [Accessed 31 December 2018].

[21] "How the Program Works," Apple Inc., [Online]. Available: https://developer.apple.com/programs/how-it-works/. [Accessed 31 December 2018].

[22] I. Skerrett, Eclipse Foundation Inc., [Online]. Available: https://www.slideshare.net/IanSkerrett/eclipse-community-survey-2014. [Accessed 21 February 2019].

[23] "Welcome to The Apache Software Foundation!," The Apache Software Foundation, [Online]. Available: https://www.apache.org. [Accessed 31 December 2018].

[24] "Apache™ Subversion®," The Apache Software Foundation, [Online]. Available: https://subversion.apache.org. [Accessed 31 December 2018].

[25] N. Zolkifli, A. Ngah and A. Deraman, "Version Control System: A Review," *Procedia Computer Science,* vol. 135, pp. 408-415, 2018.

[26] "What is Continuous Integration?," Amazon Web Services Inc., [Online]. Available: https://aws.amazon.com/devops/continuous-integration/. [Accessed 9 January 2019].

[27] B. Laster, "Continuous Integration," in *Continuous Integration vs. Continuous Delivery vs. Continuous Deployment*, O'Reilly Media, Inc., 2017, p. 12.

[28] "Unit Testing," Software Testing Fundamentals, [Online]. Available: http://softwaretestingfundamentals.com/unit-testing/. [Accessed 9 January 2019].

[29] "What is Continuous Delivery?," Amazon Web Services Inc., [Online]. Available: https://aws.amazon.com/devops/continuous-delivery/. [Accessed 9 January 2019].

[30] B. Laster, "Continuous Delivery," in *Continuous Integration vs. Continuous Delivery vs. Continuous Deployment*, O'Reilly Media, Inc., 2017, p. 12.

[31] B. Laster, "Continuous Deployment," in *Continuous Integration vs. Continuous Delivery vs. Continuous Deployment*, O'Reilly Media, Inc., 2017, p. 12.

[32] "Help prevent others from using your device without permission," Google Inc., [Online]. Available: https://support.google.com/nexus/answer/6172890. [Accessed 3 August 2018].

[33] "Create Deep Links to App Content," Google Inc., [Online]. Available: https://developer.android.com/training/app-links/deep-linking. [Accessed 3 August 2018].

[34] "Jenkins Build great things at any scale," Jenkins Open Source Community, [Online]. Available: https://jenkins.io. [Accessed 24 January 2019].

[35] "Git Plugin," Jenkins Open Source Community, [Online]. Available: https://wiki.jenkins.io/display/JENKINS/Git+Plugin. [Accessed 24 January 2019].

[36] "About GitLab," GitLab Inc., [Online]. Available: https://about.gitlab.com/stages-devops-lifecycle/. [Accessed 24 January 2019].

[37] C. B. N. N. J. C. Kıvanç Muşlu, " Transition from centralized to decentralized version control systems: a case study on reasons, barriers, and outcomes," *Proceeding,* pp. 334-344, 2014.

[38] "Git Tools - Stashing," Git community, [Online]. Available: https://git-scm.com/book/en/v1/Git-Tools-Stashing. [Accessed 24 January 2019].

[39] R. Somasundaram, "Performance," in *Git : Version Control for Everyone*, Packt Publishing Ltd, 2013, pp. 39-41.

[40] V. Driessen, "A successful Git branching model," [Online]. Available: https://nvie.com/posts/a-successful-git-branching-model/. [Accessed 24 January 2019].

[41] "Gradle User Manual," Gradle Inc., [Online]. Available: https://docs.gradle.org/current/userguide/userguide.html. [Accessed 18 February 2019].

[42] "Community Edition or Enterprise Edition," Gitlab Inc., [Online]. Available: https://about.gitlab.com/install/ce-or-ee/. [Accessed 24 January 2019].

[43] "GitLab Community Edition," Gitlab community, [Online]. Available: https://gitlab.com/gitlab-org/gitlab-ce/. [Accessed 24 January 2019].

[44] "Community Edition or Enterprise Edition," Gitlab Inc., [Online]. Available: https://about.gitlab.com/install/ce-or-ee/. [Accessed 24 January 2019].

[45] "NGINX settings," Gitlab Inc., [Online]. Available: https://docs.gitlab.com/omnibus/settings/nginx.html. [Accessed 9 February 2019].

[46] "SSL Configuration," Gitlab Inc., [Online]. Available: https://docs.gitlab.com/omnibus/settings/ssl.html#let-39-s-encrypt-integration. [Accessed 24 January 2019].

[47] "Let's Encrypt is a free, automated, and open Certificate Authority.," Internet Security Research Group (ISRG), [Online]. Available: https://letsencrypt.org. [Accessed 24 January 2019].

[48] M. Mombrea, "Migrating from SVN to Git version control - Part 2," *ITworld.Com,* 2015.

[49] R. Somasundaram, ".gitignore to the rescue," in *Git : Version Control for Everyone*, Packt Publishing Ltd , 2013, p. 76.

[50] "GitLab Runner," Gitlab Inc., [Online]. Available: https://docs.gitlab.com/runner/. [Accessed 24 January 2019].

[51] "Configuring GitLab Runners," Gitlab Inc., [Online]. Available: https://docs.gitlab.com/ee/ci/runners/. [Accessed 24 January 2019].

[52] "What is a Container," Docker Inc., [Online]. Available: https://www.docker.com/resources/what-container. [Accessed 24 January 2019].

[53] "Continuous Integration," fastlane, [Online]. Available: https://docs.fastlane.tools/best-practices/continuous-integration/. [Accessed 24 January 2019].

[54] "Install GitLab Runner," Gitlab Inc., [Online]. Available: https://docs.gitlab.com/runner/install. [Accessed 24 January 2019].

[55] "ROBOT FRAME WORK/," Robot Framework Foundation, [Online]. Available: http://robotframework.org. [Accessed 24 January 2019].

[56] "Automation for Apps," JS Foundation, [Online]. Available: http://appium.io. [Accessed 24 January 2019].

[57] "GitLab Continuous Integration & Delivery," Gitlab Inc., [Online]. Available: https://about.gitlab.com/features/gitlab-ci-cd/. [Accessed 24 January 2019].

[58] "The Gradle Wrapper," Gradle Inc., [Online]. Available: https://docs.gradle.org/current/userguide/gradle_wrapper.html#gradle_wrapper. [Accessed 24 January 2019].

[59] S. Jagtap, "xcodebuild: Deploy iOS app from Command Line," [Online]. Available: https://medium.com/xcblog/xcodebuild-deploy-ios-app-from-command-line-c6defff0d8b8. [Accessed 24 January 2019].

[60] "App automation done right," Google Inc., [Online]. Available: https://fastlane.tools. [Accessed 24 January 2019].

[61] "Ruby is...," Ruby community, Members of the Ruby community. [Online]. Available: https://www.ruby-lang.org/en. [Accessed 24 January 2019].

[62] "fastlane/fastlane," [Online]. Available: https://github.com/fastlane/fastlane. [Accessed 24 January 2019].

[63] "Getting started with fastlane for Android," fastlane, [Online]. Available: https://docs.fastlane.tools/getting-started/android/setup. [Accessed 24 January 2019].

[64] "GitLab CI/CD Variables," Gitlab Inc., [Online]. Available: https://docs.gitlab.com/ee/ci/variables/#predefined-environment-variables. [Accessed 4 February 2019].

[65] "increment_version_number," fastlane, [Online]. Available: https://docs.fastlane.tools/actions/increment_version_number. [Accessed 4 February 2019].

[66] "commit_version_bump," fastlane, [Online]. Available: https://docs.fastlane.tools/actions/commit_version_bump/. [Accessed 4 February 2019].

[67] "push_to_git_remote," fastlane, [Online]. Available: https://docs.fastlane.tools/actions/push_to_git_remote/. [Accessed 4 February 2019].

[68] "git_commit," fastlane, [Online]. Available: https://docs.fastlane.tools/actions/git_commit/. [Accessed 4 February 2019].

[69] "Understanding and Analyzing Application Crash Reports," Apple Inc., [Online]. Available: https://developer.apple.com/library/archive/technotes/tn2151/_index.html. [Accessed 4 February 2019].

[70] "GitLab CI/CD Variables | Variables," Gitlab Inc., [Online]. Available: https://docs.gitlab.com/ee/ci/variables/#variables. [Accessed 4 February 2019].

[71] "Creating a Distribution Provisioning Profile," Apple Inc., [Online]. Available: https://developer.apple.com/library/archive/recipes/ProvisioningPortal_Recipes/CreatingaDistributionProvisioningProfile/CreatingaDistributionProvisioningProfile.html#//apple_ref/doc/uid/TP40011211-CH3-SW1. [Accessed 4 February 2019].

[72] B. Laster, "Defining "Continuous"," in *Continuous Integration vs. Continuous Delivery vs. Continuous Deployment*, O'Reilly Media, Inc., 2017, p. 12.

[73] "Installation of the JDK and the JRE on macOS," Oracle, [Online]. Available: https://docs.oracle.com/javase/10/install/installation-jdk-and-jre-macos.htm#JSJIG-GUID-2FE451B0-9572-4E38-A1A5-568B77B146DE. [Accessed 24 January 2019].

[74] J. Levin, "Demystifying the DMG File Format," [Online]. Available: http://newosxbook.com/DMG.html. [Accessed 24 January 2019].

[75] "Gradle Features," Gradle Inc., [Online]. Available: https://gradle.org/features/. [Accessed 24 January 2019].

[76] "Registering Runners," Gitlab Inc., [Online]. Available: https://docs.gitlab.com/runner/register/. [Accessed 24 January 2019].

[77] "Limitations on macOS," Gitlab Inc., [Online]. Available: https://docs.gitlab.com/runner/install/osx.html#limitations-on-macos. [Accessed 24 January 2019].

[78] "Mobile Operating System Market Share Worldwide," StatCounter, [Online]. Available: http://gs.statcounter.com/os-market-share/mobile/worldwide. [Accessed 28 December 2018].

APPENDIX A: SNIPPET FROM IOS LOW MEMORY REPORT DEFINING KERNEL VERSION


```
{
  "bug_type": "298",
  "timestamp": "2018-12-24 23:11:59.34 +0200",
  "os_version": "iPhone OS 12.1.2 (16C101)",
  "incident_id": "<redacted>"
}

{
  "crashReporterKey": "<redacted>",
  "kernel": "Darwin Kernel Version 18.2.0: Mon Nov 12 20:32:02 PST 2018;
root:xnu-4903.232.2~1\/RELEASE_ARM64_T8015",
  "product": "iPhone10,4",
  "incident": "<redacted>",
  "date": "2018-12-24 23:11:59.32 +0200",
  "build": "iPhone OS 12.1.2 (16C101)",
  "timeDelta": 7,
  "memoryStatus": {
    "compressorSize": 21963,
    "compressions": 3538088,
    "decompressions": 2863309,
    "zoneMapCap": 745897984,
    "largestZone": "APFS_4K_OBJS",
    "largestZoneSize": 15319040,
    "pageSize": 16384,
    "uncompressed": 62779,
    "zoneMapSize": 112705536,
    "memoryPages": {
      "active": 40559,
      "throttled": 0,
      "fileBacked": 31901,
      "wired": 17369,
      "anonymous": 48609,
      "purgeable": 1011,
      "inactive": 36964,
      "free": 4278,
      "speculative": 2987
    }
  }
  ...
}
```

APPENDIX B: GRADLE BUILD SCRIPT IMPLEMENTATION FOR ANDROID APPLICATION VERSION MANAGEMENT

```
def readVersionPropertiesFile() {
    def versionPropertiesFile = file('../version.properties')

    if(versionPropertiesFile.canRead()) {
        Properties versionProperties = new Properties()
        versionProperties.load(new FileInputStream(versionPropertiesFile))

        if(versionProperties == null) {
            throw new GradleException("Cannot parse version.properties")
        } else if(!versionProperties.containsKey('VERSION_MAJOR')) {
            throw new GradleException("version.properties does not contain key
'VERSION_MAJOR'")
        } else if(!versionProperties.containsKey('VERSION_MINOR')) {
            throw new GradleException("version.properties does not contain key
'VERSION_MINOR'")
        } else if(!versionProperties.containsKey('VERSION_PATCH')) {
            throw new GradleException("version.properties does not contain key
'VERSION_PATCH'")
        } else if(!versionProperties.containsKey('BUILD_NUMBER')) {
            throw new GradleException("version.properties does not contain key
'BUILD_NUMBER'")
        } else if(!versionProperties.containsKey('VERSION_CODE')) {
            throw new GradleException("version.properties does not contain key
'VERSION_CODE'")
        }
    } else {
        throw new GradleException("Could not read version.properties file.")
    }

    return versionPropertiesFile
}
def readVersionProperties() {
    Properties versionProperties = new Properties()
    versionProperties.load(new FileInputStream(readVersionPropertiesFile()))

    return versionProperties
}

def getMajorVersion() {
    Properties versionProperties = readVersionProperties()
    return versionProperties['VERSION_MAJOR'] as int
}
def getMinorVersion() {
    Properties versionProperties = readVersionProperties()
    return versionProperties['VERSION_MINOR'] as int
}
def getPatchVersion() {
    Properties versionProperties = readVersionProperties()
    return versionProperties['VERSION_PATCH'] as int
}
def getBuildNumber() {
    Properties versionProperties = readVersionProperties()
    return versionProperties['BUILD_NUMBER'] as int
}
```

```
def readVersionCode() {
    Properties versionProperties = readVersionProperties()
    return versionProperties['VERSION_CODE'] as int
}

def incrementMajorVersion() {
    def majorVersion = getMajorVersion()
    def minorVersion = getMinorVersion()
    def patchVersion = getPatchVersion()

    ++majorVersion
    minorVersion = 0
    patchVersion = 0

    Properties versionProperties = readVersionProperties()

    versionProperties['VERSION_MAJOR'] = majorVersion.toString()
    versionProperties['VERSION_MINOR'] = minorVersion.toString()
    versionProperties['VERSION_PATCH'] = patchVersion.toString()

    def outputStream = new FileOutputStream(readVersionPropertiesFile())

    try {
        versionProperties.store(outputStream, null)
    } finally {
        outputStream.close()
    }

    def version = majorVersion + "." + minorVersion + "." + patchVersion
    println "Updated version to: " + version

    return version
}
def incrementMinorVersion() {
    def majorVersion = getMajorVersion()
    def minorVersion = getMinorVersion()
    def patchVersion = getPatchVersion()

    ++minorVersion
    patchVersion = 0

    Properties versionProperties = readVersionProperties()

    versionProperties['VERSION_MINOR'] = minorVersion.toString()
    versionProperties['VERSION_PATCH'] = patchVersion.toString()

    def outputStream = new FileOutputStream(readVersionPropertiesFile())

    try {
        versionProperties.store(outputStream, null)
    } finally {
        outputStream.close()
    }

    def version = majorVersion + "." + minorVersion + "." + patchVersion
    println "Updated version to: " + version

    return version
}
```

```
def incrementPatchVersion() {
    def majorVersion = getMajorVersion()
    def minorVersion = getMinorVersion()
    def patchVersion = getPatchVersion()

    ++patchVersion

    Properties versionProperties = readVersionProperties()

    versionProperties['VERSION_PATCH'] = patchVersion.toString()

    def outputStream = new FileOutputStream(readVersionPropertiesFile())

    try {
        versionProperties.store(outputStream, null)
    } finally {
        outputStream.close()
    }

    def version = majorVersion + "." + minorVersion + "." + patchVersion
    println "Updated version to: " + version

    return version
}
def incrementBuildNumber() {
    def buildNumber = getBuildNumber()
    def versionCode = readVersionCode()

    buildNumber ++
    versionCode ++

    Properties versionProperties = readVersionProperties()

    versionProperties['BUILD_NUMBER'] = buildNumber.toString()
    versionProperties['VERSION_CODE'] = versionCode.toString()

    def outputStream = new FileOutputStream(readVersionPropertiesFile())

    try {
        versionProperties.store(outputStream, null)
    } finally {
        outputStream.close()
    }

    println "Updated build number to: " + buildNumber
    println "Updated version code to: " + versionCode

    return buildNumber
}
def setBuildNumber(buildNumber) {
    def versionCode = (buildNumber as int) + 50000

    Properties versionProperties = readVersionProperties()

    versionProperties['BUILD_NUMBER'] = buildNumber.toString()
    versionProperties['VERSION_CODE'] = versionCode.toString()

    def outputStream = new FileOutputStream(readVersionPropertiesFile())
```

```
    try {
        versionProperties.store(outputStream, null)
    } finally {
        outputStream.close()
    }

    println "Set build number to: " + buildNumber
    println "Set version code to: " + versionCode
}
```

APPENDIX C: LANES FOR BUILDING PICEAONE IOS APPLICATION VARIANTS

```
################################################################################
# Build PiceaOne applications:
################################################################################

desc "Build PiceaOne"
lane :build_piceaone do
  # Install CocoaPods dependencies
  install_pods()

  # Build the PiceaOne app
  build_app(
    workspace: "PiceaOne.xcworkspace",
    scheme: "PiceaOne",
    output_name: "PiceaOne-iOS.ipa",
    output_directory: "./Outputs/Official",
    export_method: "app-store",
    export_options: {
      provisioningProfiles: {
        "com.piceasoft.piceaone2"   =>   "PiceaOne   App   Store   distribution
profile",
        "com.piceasoft.piceaone-beta"   =>   "PiceaOne   (Beta)   App   Store
distribution profile",
        "com.piceasoft.piceaone-enterprise"    =>    "PiceaOne    Enterprise
distribution profile"
      }
    }
  )

  appInfo = {
    "bundle_identifier" => "com.piceasoft.piceaone2",
    "version"           => "#{get_project_version_number}",
    "build"             => Integer(get_build_number),
    "version_code"      => Integer(get_build_number)
  }

  appInformationFileLocation       =       "../Outputs/Official/PiceaOne-iOS-
information.json"
  system("touch #{appInformationFileLocation}");

  File.open("#{appInformationFileLocation}","w") do |f|
    f.write(JSON.pretty_generate(appInfo))
  end
end

desc "Build PiceaOne (Beta)"
lane :build_piceaone_beta do
  # Install CocoaPods dependencies
  install_pods()

  # Build the PiceaOne (Beta) app
  build_app(
    workspace: "PiceaOne.xcworkspace",
    scheme: "PiceaOne-beta",
    output_name: "PiceaOne-iOS-Beta.ipa",
```

```
    output_directory: "./Outputs/Beta",
    export_method: "app-store",
    export_options: {
      provisioningProfiles: {
        "com.piceasoft.piceaone2"   =>   "PiceaOne   App   Store   distribution
profile",
        "com.piceasoft.piceaone-beta"   =>   "PiceaOne   (Beta)   App   Store
distribution profile",
        "com.piceasoft.piceaone-enterprise"    =>    "PiceaOne    Enterprise
distribution profile"
      }
    }
  )

  appInfo = {
    "bundle_identifier" => "com.piceasoft.piceaone-beta",
    "version"           => "#{get_project_version_number}",
    "build"             => Integer(get_build_number),
    "version_code"      => Integer(get_build_number)
  }

  appInformationFileLocation      =      "../Outputs/Beta/PiceaOne-iOS-Beta-
information.json"
  system("touch #{appInformationFileLocation}");

  File.open("#{appInformationFileLocation}","w") do |f|
    f.write(JSON.pretty_generate(appInfo))
  end
end

desc "Build PiceaOne (Enterprise)"
lane :build_piceaone_enterprise do
  # Install CocoaPods dependencies
  install_pods()

  # Build the PiceaOne (Enterprise) app
  build_app(
    workspace: "PiceaOne.xcworkspace",
    scheme: "PiceaOne-enterprise",
    output_name: "PiceaOne-iOS-Enterprise.ipa",
    output_directory: "./Outputs/Enterprise",
    export_method: "enterprise",
    export_options: {
      provisioningProfiles: {
        "com.piceasoft.piceaone2"   =>   "PiceaOne   App   Store   distribution
profile",
        "com.piceasoft.piceaone-beta"   =>   "PiceaOne   (Beta)   App   Store
distribution profile",
        "com.piceasoft.piceaone-enterprise"    =>    "PiceaOne    (Enterprise)
distribution profile"
      }
    }
  )

  appInfo = {
    "bundle_identifier" => "com.piceasoft.piceaone-enterprise",
    "version"           => "#{get_project_version_number}",
    "build"             => Integer(get_build_number),
    "version_code"      => Integer(get_build_number)
```

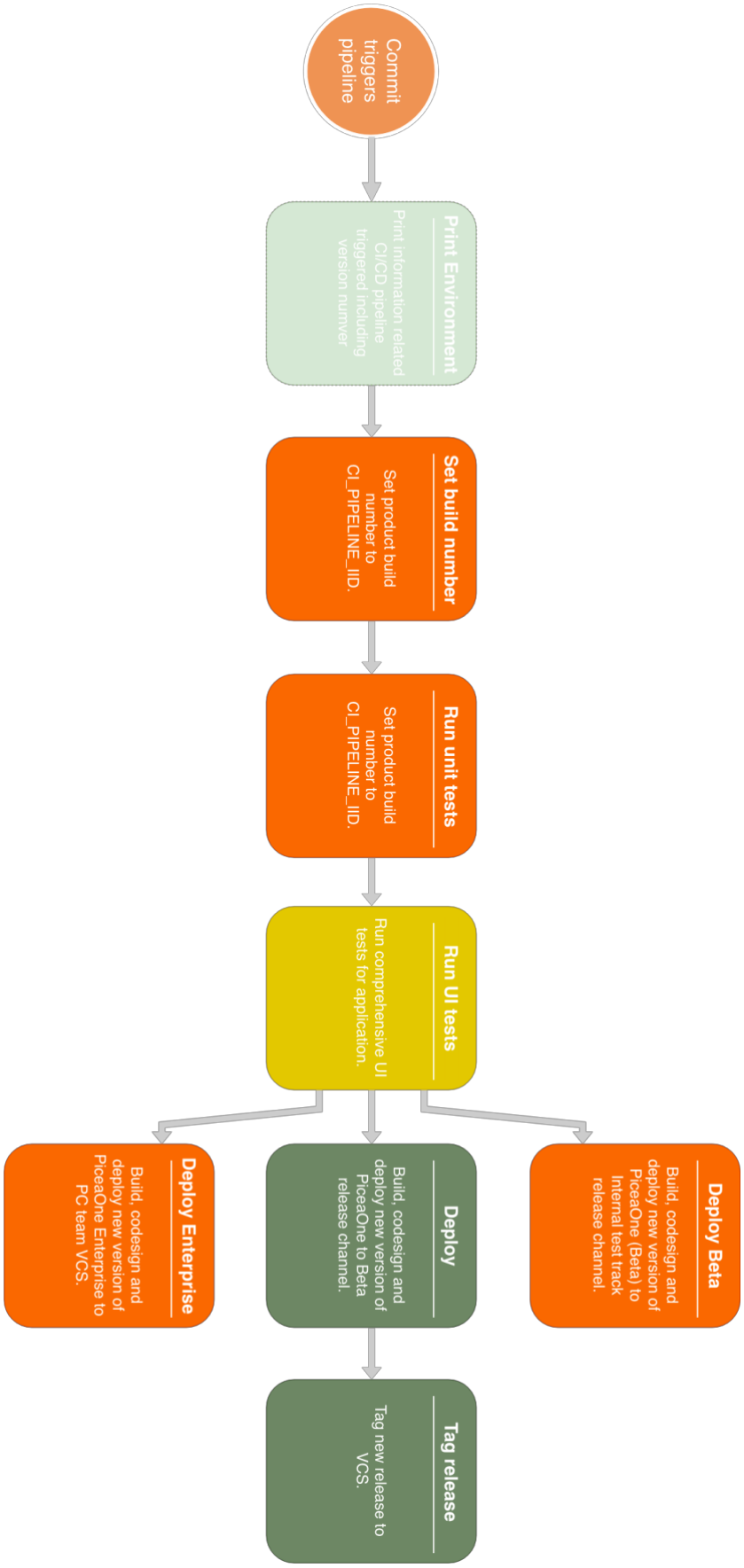
```
  }

  appInformationFileLocation      =      "../Outputs/Enterprise/PiceaOne-iOS-
Enterprise-information.json"
  system("touch #{appInformationFileLocation}");

  File.open("#{appInformationFileLocation}","w") do |f|
    f.write(JSON.pretty_generate(appInfo))
  end
end
```

APPENDIX D: LANES FOR BUILDING PICEAONE ANDROID APPLICATION VARIANTS

```
################################################################################
# Build PiceaOne applications:
################################################################################

desc "Build PiceaOne"
lane :build_piceaone do
  # Clean project
  gradle(task: 'clean')

  # Build project with Store configuration
  gradle(
    task: 'assemble',
    flavor: "store",
    build_type: 'Release',
    print_command: false,
  )

  # Copy build apk package to Outputs directory
  %x(mkdir -p ../Outputs/Official)
  %x(cp          ../app/build/outputs/apk/store/release/app-store-release.apk
Outputs/Official/PiceaOne-Android.apk)

  # Generate app information json file
  appInfo = {
      "bundle_identifier" => "com.piceasoft.piceaconnector",
      "version"           => "#{get_version_number}",
      "build"             => Integer(get_build_number),
      "version_code"      => Integer(get_version_code)
  }

  appInformationFileLocation     =     "../Outputs/Official/PiceaOne-Android-
information.json"
  system("touch #{appInformationFileLocation}");

  File.open("#{appInformationFileLocation}","w") do |f|
    f.write(JSON.pretty_generate(appInfo))
  end
end

desc "Build PiceaOne (Beta)"
lane :build_piceaone_beta do
  # Clean project
  gradle(task: 'clean')

  # Build project with Beta configuration
  gradle(
    task: 'assemble',
    flavor: "beta",
    build_type: 'Release',
    print_command: false,
  )

  # Copy build apk package to Outputs directory
  %x(mkdir -p ../Outputs/Beta)
```

```
  %x(cp            ../app/build/outputs/apk/beta/release/app-beta-release.apk
../Outputs/Beta/PiceaOne-Android-Beta.apk)

  # Generate app information json file
  appInfo = {
      "bundle_identifier" => "com.piceasoft.piceaconnector.beta",
      "version"           => "#{get_version_number}",
      "build"             => Integer(get_build_number),
      "version_code"      => Integer(get_version_code)
  }

  appInformationFileLocation    =    "../Outputs/Beta/PiceaOne-Android-Beta-
information.json"
  system("touch #{appInformationFileLocation}");

  File.open("#{appInformationFileLocation}","w") do |f|
    f.write(JSON.pretty_generate(appInfo))
  end
end

desc "Build PiceaOne (Enterprise)"
lane :build_piceaone_enterprise do
  # Clean project
  gradle(task: 'clean')

  # Build project with Enterprise configuration
  gradle(
    task: 'assemble',
    flavor: "enterprise",
    build_type: 'Release',
    print_command: false,
  )

  # Copy build apk package to Outputs directory
  %x(mkdir -p ../Outputs/Enterprise)
  %x(cp           ../app/build/outputs/apk/enterprise/release/app-enterprise-
release.apk ../Outputs/Enterprise/PiceaOne-Android-Enterprise.apk)

  # Generate app information json file
  appInfo = {
      "bundle_identifier" => "com.piceasoft.piceaconnector.enterprise",
      "version"           => "#{get_version_number}",
      "build"             => Integer(get_build_number),
      "version_code"      => Integer(get_version_code)
  }

  appInformationFileLocation    =    "../Outputs/Enterprise/PiceaOne-Android-
Enterprise-information.json"
  system("touch #{appInformationFileLocation}");

  File.open("#{appInformationFileLocation}","w") do |f|
    f.write(JSON.pretty_generate(appInfo))
  end
end
```

APPENDIX E: CI/CD PIPELINE



APPENDIX F: SURVEY RESULTS FOR MOBILE APPLICATION DEVELOPMENT TEAM

1. New VCS is complicated and difficult to use.

2. New VCS is missing crucial features.

3. New work flow is suitable for my daily development.

4. New VCS allows easier context switching compared to old system.

5. I prefer the new VCS and workflows over the old system.

6. Integrated CI/CD pipeline has saved me significant amount of time.

7. Managing code base near upcoming release is easier with new VCS.

APPENDIX G: SURVEY RESULTS FOR PC APPLICATION DEVELOPMENT TEAM

1. New VCS is complicated and difficult to use.

2. New work flow with mobile application development team is suitable for my daily development.

3. Features implemented into mobile applications are available to integrate quicker than before.

4. I would prefer to switch from Subversion to Git.

APPENDIX H: SURVEY RESULTS FOR PC QA TEAM

1. Features implemented to mobile applications are available for testing quicker than before.

2. Features implemented into mobile applications are integrated into PC solution quicker.

3. Integrated CI/CD pipeline has improved the overall software production process.