Bachelor's thesis

Information and Communications Technology

2019

Vladimir Sakharov

WORKFLOW OPTIMISATION IN UNITY ENGINE

TURKU AMK

TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor of Engineering, Information and Communications Technology

2019 | 21

Vladimir Sakharov

WORKFLOW OPTIMISATION IN UNITY ENGINE

The commissioner of this thesis, Flatfish Games Ltd, is a gaming company that focuses on business-to-business contracts. The process of working on multiple projects for different customers needs a stable way of delivering the software to the customers for assesment

The purpose of this thesis was to implement an automated software delivery pipeline to make the development process more robust, manageable and configurable.

The pipeline consists of 4 parts: a Version Control System(VCS) hosted on the GitLab platform, an automated build system in the form of Unity Cloud Build(UCB), a web application written in JavaScript utilizing NodeJS framework hosted on the Heroku platform, and Visual Studio Appcenter distribution platform . The setup was conducted via configuring the services separately and setting up communication between them through HTTP requests.

The logic of the configuration was implemented in a way that instead of every step of the deployment process executed in the same platform with the service responsible for each step. The information was transferred from one service to another through an HTTP trigger.

The VCS was configured to increment the project version on each change to the files in the repository. After completing the file modification, the VCS transfers the public address for the files to UCB. After receiving the location of the files required for the build, the build system creates a new virtual environment, executes all the automated tests, publishes the new virtual environment to a publicly-accessible location and notifies the web server of its location. After receiving a notification containing the download link for the build package, the web server downloads it, uploads it to the Visual Studio Appcenter distribution platform and notifies the customer of the new version.

The time calculations concluded that the deployment system is implemented according to the needs of the company and it has been used in the production environment since January 2019. However, the implementation can be improved upon by setting up a local build environment and a proprietary distribution platform.

KEYWORDS:

Continuous integration, continuous delivery, test automation, test driven development, build automation, Unity, game development.

CONTENTS

TABLES

LIST OF ABBREVIATIONS (OR) SYMBOLS

AI	Artificial Intelligence
API	Application Programming Interface
CD	Continuous Deployment or Continuous Delivery
CI	Continuous Integration
CLI	Command-Line Interface
CVCS	Centralized Version Control System
DVCS	Distributed Version Control Systems
FaaS	Function as a Service
IAP	In-App purchase
PaaS	Platform as a Service
TDD	Test-Driven Development
UI	User Interface
UCB	Unity Cloud Build
URL	Uniform Resource Locator
VCS	Version Control System

1 INTRODUCTION

Flatfish Games Ltd is a gaming company focused on providing other gaming companies with solutions for their programming and art-related needs. With a growing number of customers who require constant updates on their product, it is essential that the current solution delivery solution is adequately flexible to accomodate a varying number of targets.

The objective of this thesis is to implement a software delivery pipeline that requires minimal work to maintain and reconfigure. The pipeline will also need to be easily reproducable to allow for quick setup and takedown. At the same time, each member of the development team needs to have access to every step of the pipeline to minimize dependency on a single person.

In the first chapter of the thesis, the author aims to provide a background for the project and provide support for his claim for automation of the delivery process. In the second chapter the author aims to provide the reader with the theoretical background necessary to understand the practical implementation of the system, which will be discussed in the third chapter.

Before attempting to implement such a system, a survey was carried out to identify the delivery steps common among the industry professionals and the amount of time these steps take. The Thus 21 game development professionals working in the game development industry in Finland participated in the survey. The participants were presented with a Google Forms survey, which asked them to describe the steps of their delivery process and the time eacj of these steps take. After receiving the results, the author combined the steps into generalized categories and calculated the mean time for each category. Table 1 summarizes the results of this survey.

Table 1. Results of developer survey.

Step	**Manual workflow(daily, hours, average)**	**Automated workflow(daily, hours, average)**
Bringing the project up-to-date for the build	0.3	0
Cleaning up the project of development files and making sure no artifacts are left	0.15	0
Switching between deployment targets	2	0
Running tests for each platform	0.5	0
Building for each platform	1	0
Distributing the builds for the test team	0.3	0
In-house acceptance testing	0.3	0.3
Deploying to customer	0.5	0
Total time spent daily on testing, building and deploying the product	5.05	0.3

• Bringing the project up-to-date for the build – 10 minutes

When building a project, the developer has to switch to the branch dedicated for outside builds and make sure their project is up-to-date with the rest of the team

• Cleaning up the project of development files and making sure no artifacts are left – a few minutes to half an hour

Usually, manual deployment builds are carried out at the end of the working day. Before the build can be started, the developer must perform several actions to eliminate any errors in the building process. First, the developer has to sort through the files leftover from changes added to the repository throughout the day and delete development artifacts unnecessary for the build. After cleaning the environment, the developer has to commit the changes to an appropriate branch and stash the work that needs to be kept but does not belong in any of the existing branches.

• Switching between deployment targets – 30-40 minutes per platform

The process of switching between build targets is the greatest time consumer, according to the survey. Switching between PC, MacOS, Android and iOS requires Unity to switch many of its settings and essentially reimport every part of the project again.

• Running tests for each platform – 5 minutes per platform with good test separation

Since all the In-App Purchases(IAP) - specific libraries are platform specific and device-specific information is too, tests usually have to be run on each platform to exclude edge cases of build only working on one.

• Building for each platform – 20-30 minutes per platform

The building process takes longer for mobile devices.

• Distributing the builds for the test team – 10-20 minutes

Whether the team is using an on-site server or external testing platform, uploading the builds there and downloading them takes a significant amount of time

• In-house acceptance testing – 20 minutes

Most teams considered a 20-minute daily testing session to be optimal for assessment of the state of the game

• Deploying to customer – 5 minutes per platform

Deployment is heavily dependent on the distribution platform the customer requires, but overall this usually consists of uploading a file and writing a short description of the build.

As can be seen from the results, the most time-consuming steps (Switching between deployment targets and the actual build process) are the most easily automatable ones – if the development team has a build server, they can just specify the build environment and that by itself eliminates all the human resource that goes into these steps.

Another interesting outcome that becomes apparent is that the time going into the deployment process grows per platform. If the main product of the company is an in-house project for one platform, the company might require an external commission to fund it. In such a scenario, the difference in time required to deliver the product to an external customer without an automated delivery system in place can offset the influx of funds to a point of non-profitability.

A common point indicated by many participants of the survey was that nobody was able to quantify was the so called “debugging efforts”. Such complications are not usually present at the beginning of the project but grow exponentially with the size of the project. Some participants indicated that in the larger project where the pipeline and the workflow were not setup at the beginning, reported a “tipping point” that would happen around the 7-month mark in the project timeline. At this point in the production cycle, the debugging and the maintenance of the project would start taking more development time than the implementation of new features and content.

Essentialy, in aperfect world, the development team has never had to concern themselves with how their work is delivered to the customer.

Unfortunately, at this point in history, software that would let creators do what does not exist. A feasible alternative exists in the form of automating a significant portion of the deployment process.

Assessing the steps of the manual workflow, the steps that take the most time are the easier ones to automate. By automating just the parts of the workflow that are related to the building of the product, the development team can eliminate at least half of the time associated with the deployment process. Considering the fact that the building process has to include Unity and the running unit and that integration tests requires just one CLI command, that part is essentially included with the builds, increasing the time savings even further.

The most important part of automating the deployment process is the fact that with a fully automated pipeline and a dedicated team of testers, the development team can be fully eliminated from the process. This means that not only time is being saved, but the productivity increases due to the elimination of the constantly on-call mentality and the stress associated with that.

From the result of the survey of teams with at least some parts of the workflow, it becomes apparent that the only part of the deployment process that cannot be automated is the acceptance testing part and everything else is reduced to require no human interaction.

Unfortunately, some common tasks that still plague the professionals surveyed all have to do with the lack of time to set up a properly configured pipeline. Jenkins is considered to be the most used open source automation server (Jenkins, 2019), which means most teams rely on plugins for integrations with other services. The fact that Jenkins has to be run on provided hardware means acquiring an Apple device for iOS builds. Since plugins are not easily modifiable and builds are ran on an environment different from the development one, unexpected behaviours occur often, so teams have to dedicate a person to be responsible for the system.

Nevertheless, teams with even a partly-automated deployment process report much smaller time investment requirements, most a 30-minute a day range, when this time is dedicated to pipeline maintenance and human testing of the product.

2 THEORETICAL BACKGROUND

To understand which technologies are needed for proper workflow optimization, the developer needs to understand the concepts underlying each step and have a good grasp on the different ways of implementing them.

Overall, the steps of the pipeline follow the same pattern: Every developer performs their part of the work on their machine, syncing with the rest of the team through VCS(Version Control System). An external version control system runs the tests on each commit, which is a snapshot of the project's changes, to let the developer know that their work has not broken anyone else's. For distribution purposes, project properties such as version number and build identifier usually have to be constantly incremented, so doing that on each project iteration – a commit to the development branch, is considered to be a good practice. A dedicated server or cloud platform is usually set up as the build environment to keep it separated from the development artifacts. It listens for a trigger of some sort and after the tests have passed, it builds the project and runs the UI and Load tests. After the build is complete and tested, depending on the strategy the team has decided on, the product is either delivered to the distribution platform and awaits a release trigger, or is deployed straight to the customer.

The important part to notice is here that in a properly setup pipeline, the average developer's attention is not required on a regular basis for any steps after the local work is pushed to the external version control system and the basic test suite is completed.

2.1 Version Control

Version Control Systems are a method for a development team to keep a history of changes to the source code over time. A VCS keeps track of each and every change made to the code base in a special database-type structure, so that if something goes wrong, the developers can go back to previous versions of the code and find the mistake they have made. Unlike a simple backup system, a VCS usually tracks not just the latest working version of the file, but a list of all the versions of it, so even if the developer does noy find their mistake straight away, the history is still there for later examination (Kim et al. 2017)

CVCS (Centralized version control systems) go one step further and store these changes on a single remote server for the whole team. Having a remote point of connection with the rest of tne team is incredibly useful in keeping everyone up-to-date on the state of the project and making sure everyone's work is in sync.

DVCS (Distributed Version Control Systems) go even further by letting clients have a full copy of the remote repository on their machine, thus making sure that any one of those copies can be used as the new version of the repository in case something happens to the server, thus minimising the risks.

. If there has ever been a tool that can be described as"old, simple and reliable" CVS Oldest centralised CVCS is this. Unfortunately, being quite an old system it has many downfalls. The dirst downfall that turns off development teams is the centralised nature of it. If the server that hosts the files goes down, nobody can connect to it, therefore, the team loses their ability to work on the project. On the other hand, the management of the files has a more top-down approach, where the managers have full control over the system, which provides a much higher level of security. Secondly, the changes are handled on a per-file basis, which constitues a difference in the whole paradigm, not just a minor workflow difference. In other VCS, reverting changes and tracking the history is much easier due to this change. Another great hurdle for teams to overcome is the branching in CVS. In a centralized system, changes are stored on a per-file basis, forcing the developer to manually bundle changes under a "tag", which means more conditions to keep in mind during the development process.

Apache Subversion (SVN) is the most popular CVCS on the market right now. Being a CVCS, SVN faces all the problems of a centralised system as well. Unlike CVS though, SVN can handle changes on a per-directory basis, so many problems CVS has are eliminated here. Unfortunately, having a central point of connection is the downfall of SVN for game development because most game companies require many branches for different features. Having to acquire even per-directory permissions can be too much work for a smaller company to handle due to the practices of rapid prototyping and the reality of changing milestones being too common in the industry. Another notable difference of SVN from CVS is the inability to rollback. Unfortunately, SVN does not support reverting commits, which leads to complications if mistakes happen – the proposed solution is to take a good revision and put it on top of the bad one in the history, which unfortunately creates a very messy history and complicates the process of fixing bugs.

Mercurial is a DVCS and thus comes with all the advantages it provides: scalability, maintainability and lack of centralised point of failure. This means the developers do not have to depend on the server and the internet connection to do their work. No need to configure firewalls and VPNs to work with it. Every member of the team has a backup of the whole project and can commit as much as they want without ruining other people's work. Mercurial allows for more people to have ”commit access”, lowering the contribution barrier and letting the maintainers decide what changes to pull. Contributors can separate into teams, handle their own branches and merging, which decentralises the workflow and requires less maintenance from the managers.(Sink 2011)

Git, like Mercurial, is a DVCS, but with a few major changes. One of the largest shifts from previous VCS is the way Git stores changes. Unlike Mercurial, git never really lets changes into the project, it creates new objects instead. Combined with the fact that git stores references to modification in the commits and references to those commits in the branches, and the fact that git refuses to delete anything that it can still reference, mean that the history contains all the changes ever made to the project. Next comes the ability to rewrite history. Git does not put any restrictions on its history, so the developer has full control over it. Of course that comes with downsides when other people become involved, because rewriting the history that someone else has copied already results in conflicts. However, if used correctly, it lets the user be as careless with their local work as possible and then craft it into a perfectly readable and understandable story when sharing it. Combined, the storage format and the history manipulation function mean that there is almost never a situation that there will be no recovery from. The feature of git that is driving every other VCS down is branching. Git lets its user create, delete, and modify branches as freely as they want as well as making sure branches exist in corresponding namespaces. A branching system like that lets the user easily identify which branch they are on and avoid confusion and conflicts. The most polarizing feature that git introduced is the index or the “staging area”. The staging area is a virtual place between the user's work and the commits they are creating where all the changes they have made exist. Although confusing at first, this is the feature that takes so much anxiety out of the process of sharing work. Git allows the developer to select which files they want to commit or which parts of file they want to patch. This eliminates unnecessary testing and logging out of history and also lets them stash the changes if the developer is not sure if they have committed everything that is needed. By combining all these features together, git eliminates all the moments where the users feel like the technology is to blame. It gives the developers full control and if proper security measuere are taken, nothing will ever be lost. Unfortunately, that is also where the greatest issue people have with git comes from – the user has full control. As any tool with extensive functionality, git is as far from beginner friendly as possible. Extensions for git can be written in many languages, each command comes with a range of options and like most bash or shell applications, git commands and their results can be used or piped inside other commands. From an experienced developer’s point of view, these are all positive features, but git’s learning curve would be better described as a giant vertical wall, which turns off many novice developers and can be a huge/enormous hurdle for teams with people lacking that expertise. (Chacon 2009)

2.2 Tests

Test-driven development (TDD) or as it is lovingly known ”red – green – refactor” is a software development process based on repeating the same short development cycle over and over again:

- Turn requirements into tests
- See those tests fail (Red)
- Write minimum functionality to see tests pass (Green)
- Rewrite that functionality to fit into the overall structure of the project(Refactor)
- Repeat

TDD has been widely accepted in the software development industry not because it is a great theoretical concept, but because its benefits are hard to dispute. The more tests a programmer writes, the fewer unhandled errors will emerge and the easier it will be to pinpoint the cause of the ones that do. In addition, if the structure of the product is driven by tests, the product itself becomes more user-oriented. By focusing on the requirements and turning those into tests, the developer has to imagine how the functionality will be used by the clients. Thinking about the concrete use cases makes the product less based on theory and assertions. The limitations come, as usual, from the human point of the equation: the only way to do proper TDD is if everyone involved believes in it and follows the structure. The time it takes to learn how to do it is precious development time not every team is able to afford. For smaller, faster projects the benefits do not always outweigh the negatives. Properly implemented TDD will always save time in the long run, but might not justify itself with a project designed to be completed in 2 weeks. In addition, moving a project that was not started with TDD in mind to use tests properly is not an easy tasks which only expands with time. (Beck, 2014)

The major problem with even properly implemented tests is that they have to be run by someone. A normal manual testing workflow usually consists of:

1. Cloning the source
2. Installing all the needed dependencies
3. Running all the tests, which could include hundreds of unit tests, hundreds of integration tests and dozens of UI tests.
4. If tests succeed, deploying the application. If not, seeing which tests failed and notifying the people whose work is affected

(Mosley & Posey 2002)

Not only does this process require the developer to actually put time and energy aside for running the tests, this practice will likely involve multiple people any time tests fail. Furthermore, it is prone to human error in cases where the developer is tired, stressed or is just not paying enough attention. Machines are designed to perform similar tasks over and over again, so one wonders why waste precious development time of humans who are notorious for being bad at routine tasks.

When deciding on the test automation tool, it is important to know which exact freamework to look for. In total, there are six common categories automation frameworks can fall into, each with their own benefits and disadvantages:

1. Linear Automation Framework is the simplest framework to setup, also known as record-and-playback. Linear automation lets the testers records the test steps and the framework repeats them. The issue with this approach is that the ”record” step has to be redone with each dataset or change to the application.
2. Modular Based Framewor aims to provide functionality similar to Linear Automation on a smaller scale. By separating the project into ”modules”, the tester can create test sets for each module individually, thus requiring a rework only of the modules affected by changes to the application.

3. Library Architecture framework takes the modular approach further and aims to separate the application into tasks and group those tasks by their function. After the separation, the tasks are kept in a separate library. The library is then used by the test scripts whenever they're run.
4. Data-driven frameworks aim to separate the tests and the data. Linear, Modular and Library-based frameworks all fall prey to the same problem: the tests are hard-coded. Therefore similar operations on multiple datasets are impossible without writing a specific taste case for each change in data. The separation present in Data-Driven frameworks allows the same tests to be run on multiple sets of data. Being used by the same tests means that the data needs to be indentified, formated and stored in specific ways. The ways of connecting that data to the tests need to be written by a progreammer, which results in a significant jump in time required to set up this frameworks of this type.
5. Keyword Driven Framework takes the concept of the data-driven framework one step further by separating more functionality away from the test tool. Such frameworks require creation of a set of keywords that describe certain actions. These keywords then drive the test scripts in accordance with these keywords. By doing this, after the keyword table is set up, the testers have to just specify the code that will prompt the required action based on the description of the keyword. This framework allows to reuse multiple actions for different test scripts and even application itself.
6. Hybrid frameworks are a combination of a number of previously mentioned frameworks, designed to use parts of one category to mitigate the weaknesses of another. (Aebersold 2019)

Automated testing in game development brings a whole new level complexity with a large number of input combinations and rendering. Starting with the lowest level of testing, Unity provides the creators with a good inbuilt tool for writing both unit and integration tests. This testing framework is based on an open-source unit testing framework NUnit. (Unity 2018) The tests can then be run by either ticking a checkbox in Unity Cloud Build or passing a ”runTests” command to the Unity process through the command line on the dedicated build machine.

A solution to commonly occurring rendering problems is setting up test scenes which will then be used by image recognition software to pixel-compare their results to pre-recorded images depicting the correct rendering. t´The difference between pre-recorded and newly generated images can be usedto detect issues in the rendering pipeline.

The most complicated part of testing a game is assessing the gameplay. For testing various in-game situations, a common practice is to create AI agents simulating specific player behavior. The most common situation in which this practice is applicable is movement: to detect any problems with traversing terrain, developers create an AI that moves around the play area. While moving, the agent sets nodes wherever it goes and marks the connection between the nodes as successful or failed. After completing such a test, the tester is presented with a map of the area with possible and impossible paths clearly defined.

Another useful tool available to developers in modern times is Machine Learning. Unity has an official reinforcement learning package. This package lets developers either model the agents' behavior after the players or predefine it and let the agents explore the game world. This is not something that is required for any project due to the time to set up and the test run time being significantly larger than traditional testing. However, such a tool lets the developers explore highly unexpected player behaviours in a controlled environment.

2.3 Build Automation

The larger the development team is, the larger the project and the more time the team goes with no full build, the greater the chances of a bug sneaking into it are. Therefore, the longer it will take to understand where the issue is coming from. Build automation tries to mitigate this issue by letting people schedule their builds at specific times or even triggering a build with each change to the codebase. The benefits of automatic builds are numerous:

- By always creating the build from the same environment, the risk of failed build is reduced.
- By automating all the build steps, the risk of failure due to human error is eliminated
- When a bug is fixed, the testers obtain a new version quickly for retesting

- The build can be picked up by other parts of the company quickly without the need for one of the developers to manually transfer it to them
- By keeping a history of builds, pinpointing a build which introduces a bug becomes easier and seeing if the fix for it is in the version a person is testing becomes instantaneous.

Overall, Build Automation saves the development team a lot of time on performing the steps manually, reduces the number of points of failure and provides the developers with a lot more sense of security. Reducing the amount of times developers have to verify their work and worry if they forgot something crucial lets them focus on creating content, verifying already implemented features.(Mosley & Posey 2002)

More than 50% of the game market nowadays are mobile games (newzoo Global Games Market Report 2018), so it is safe to assume that any game development team will have to deliver products to mobile platform. This means the team will have to include iOS builds into their workflow, limiting their builder choices significantly. There are no legal ways of building iOS application on non-MacOS devices. Additionally, it is illegal to run MacOS on non-Apple hardware. Considering that less than 15% of Unity developers use Macs for development (Unity Stats 2015), most teams have to either acquire a Mac and dedicate it for building or use a remote provider that rents out the hardware. To mitigate the issue, Unity introduced Unity Cloud Build in 2017, letting developers build Unity applications through Unity-owned cloud services.

3 PRACTICAL IMPLEMENTATON

This thesis is based on work done for Flatfish Games as part of the process of increasing the overall productivity of the team. The project consisted of researching parts of the company's workflow that could be removed, automated or improved on in terms of time consumed. The result had to provide every developer with access to the infrastrucure of the pipeline, as well as minimise human involvement in such steps as build or testing the project. The final pipeline consisted of a GitLab hosted git server responsible for version control, unit tests and file manipulation, Unity Cloud Build(UCB) platform monitoring the git repository and automatically building the application from the files hosted in there, and a Heroku-hosted NodeJS web application delivering the builds to Appcenter for testing. Apple TestFlight and Google Play were chosen as platforms for open testing due to their wide availability. Although proprietary software is often considered to be suboptimal, the choice to use testing platforms maintained by the same companies that own the stores was made. Using these platforms assures that part of the acceptance testing is automatically performed when delivering the builds to the open testing environment.

The reason for choosing git was its wide adoption and ability to support multiple developers working on the same files. GitLab as service provider was a big deciding factor, due to its inclusion of a configurable CI/CD pipeline, which let the team modify the version files without leaving the VCS environment.

There are many platforms for beta testing applications, where most of the PC and MacOS ones are fairly easy to set up and manage. Most problems arise when developers need to test their applications on a mobile platform. The ones considered for this project were Testflight by Apple, available only for iOS development,, Google Play Internal Test Track by Google,available only for Android development and App Center, formerly HockeyApp, by Microsoft, available for both iOS and Android. All of these platforms are easily distributable to via applications like Jenkins or Fastlane.

App Center is a platform designed by Microsoft to ease the process of delivering the product to the customer both in testing and production enviornment. It supports Windows, MacOS, iOS and Android applications. The platform also has support for Windows App store, Apple App Store and Google Play Store. The process of delivering the applications is very straightforward: App Center requires 4 requests to be sent for the distribution to go through:

1.A post request with the credentials for the project to get the upload url

2.A post request to the upload url with attached file

3.A patch request to the release url with the status of the release

4.A patch request with the release group and the release notes specified

To automate those requests, a simple web app was created that downloads the finished build from Unity Cloud Build and then performs the required App Center upload request. Additionally, the app uploads builds directly to Google Play and iOS testflight if instructed to do so.

For Android, Google play has 3 different testing tracks. Internal - designed for distributing the app for in-house testing. Closed - available so that the developers can invite a larger test base to participate in the testing. Closed testing is often used to make the application available to other departments of the company and a small group of trusted users. Last testing track available is Open Testing. Open testing is the last testing stage before an official release is made. During this stage the application becomes available on the Play Store as a test version. Any person residing in the regions published to will be able to test it and provide feedback for the development team. The feedback is kept private not to contaminate the rating and public opinion of the application.

Google provides the developers with a simple API mirroring the functionality of the Google Play Console. Similar to the way App Center handles uploads, a developer needs to pass a request to get the information on where to upload the file, pass the file to the corresponding location and then update the testing tracks with the resource URL referencing the newly uploaded application.

Similar to the building process, Apple has restricted the App Store upload process to its proprietary software called Application Loader. This means that if the development team does not want to use any middle man platforms like App Center, they are restricted to using Apple hardware as the distribution server. Considering the fact that apple does not produce server-grade hardware, this choice severely limits the processing power and software stack of the project. If owning the hardware is necessary, applications such as Jenkins and Fastlane that let developers deploy through Application loader with no manual work. Native Apple Developer Tools or XCode Server are proprietary Apple services that provide developers with the ability to not rely on third-party software, but lack support for other platform distributions.

Setting up and building the project will probably be the most time-consuming parts of the process, due to the fact that Apple requires a fully configured Apple Developer account for any build.

To configure the Apple Account, it is required to:

• Sign up for the Apple Developer Program

• Get an Apple ID

• Create a Certificate Request file

To get a certificate request, one needs to use the ”Keychain” application on a MacOS machine or use the OpenSSL library

• Upload the certificate request to the Apple Developer Console

• Create a certificate from the Certificate Request

• Download the certificate file

• Install the certificate into the ”Keychain” application if it is available

• Create a .p12 file corresponding to the certificate either from the ”Keychain” or by generating one with OpenSSL using the certificate

If the .p12 file is created using OpenSSL and the team plans to distribute the product through any microsoft services, it is required to give it a “friendly” name, otherwise it will be rejected. This can be done by passing a “–name” argument during the creation process

After obtaining both the certificate and the .p12 file, they have to be provided it to the build and distribution platforms. A very important thing to keep in mind is that there is no way to obtain the password from the .p12 file, so if it is lost, the process has to be repeated from the beginning.

One of the main reasons why automating the building and deployment of an iOS application is advised is the fact that Apple rejects applications with a build number equal or lesser than any of the previously uploaded ones. A small step like this is exactly the kind of thing a human mind is exceptional at forgetting about, resulting in many situations where the whole deployment has to be repeated many times if the incremention is not automated.

If the team opts out of having a dedicated iOS building machine or purchasing access to a cloud-based Mac solution, the company are locked into using UCB, which lacks integrated support for App Store and Play store. However, UCB has support for webhooks, which expose all the necessary information about the build in the request. This information lets the developer set up their own external server that is listening for that request and then deploys the build.

For setting up the web server, NodeJS environment with the Express framework was chosen for this project. The choice of a less modular, but easier to setup approach simply was made because one of the goals of this project was to let anyone on the team modify the pipeline according to their demands, thus limiting some technical aspects of it.

The broad functionality of the app can be described as:

1. Listen for the trigger from UCB

2. Parse the content of the request for necessary information

3. Upload the build to specified platforms according to the infromation in the request

Most modern web application frameworks come with many libraries designed specifically for parsing information out of an incoming request, while javascript has that functionality built into the language. The information that needs to be extracted from the incoming request is:

• Number of the build

To use for different deployment platforms that require build number to be in strict succession, as mentioned in the Apple distribution section, App Store does not let applications with build number lower or identical to one of previous revisions to go through)

• Target name of the build

The easiest way to differentiate between deployments for different platforms is to setup separate build pipelines for them, letting the flow of each platform be separate.

• Platform

Even if a universal distribution system like App Center has separate routes for the builds for different platforms. This means that web application needs differentiate between different build packages.

• File location

Possibly the most important piece of information UCB provides – the external download path for the build, necessary for the download of the file that will be distributed.

• Revision number

This piece of information Is completely optional, but useful for debugging the build process as well as identifying issues with the product itself. Most distribution platforms allow for “release notes” to be attached to the release and having the revision number there allows for identification of specific commits the build is coming from.

For the upload part, distribution platforms such as App Center or Google Play have an API that can be manipulated through web requests with specific information inside. Such functionality is present in NodeJS, thus no extra dependencies were required.

Results

At the end of the project, the finalized pipeline was highly configurable, had almost no unknown variables and was only tied to one proprietary service – Unity Cloud Build. Each step was kept dependency-free and enviornment agnostic to the best of the author’s ability. Additionally, the source code was made available to every member of the team. In conjunction, these factors minimized the number of possible points of failure and gave every team member the ability to fix and modify the functionality. Avoiding complicated plugin-heavy software required a significant amount of time investment upfront. However, this presented additional freedom in customising the pipeline for different platforms. Writing the software independently of the envrionment makes it possible to move parts of the pipeline around without the fear of them breaking due to unexpected environment changes. Another big positive of the final configuration is that the pipeline was 100% linear. This factor made sure that once an environment has served its purpose, other steps do not require it to be setup again to receive further instructions or information.

Using Gitlab CI meant that the file manipulation for version increments and build triggering was running on a clean Debian setup, with the possibility of including Docker images if the need arises. This ensures that the most crucial point of the pipeline the most flexible one. If something goes wrong with this step, any team member has the ability to turn it off, ignore it or move it to another Unix-like evnironment without impeding the workflow for other people.

Using UCB as the building solution provided the ability to completely separate the building and development environments. At the same time, UCB allows building for any platform without the need to acquire extra hardware. In addition, the fact that it is a service means the team can modify the build templates but the environment maintenance is not a concern. The need to acquire a dedicated build server that the company is in charge of maintaining and upgrading is eliminated.

Setting up the deployment environment on a Node.js server meant full control over how many dependencies the team wants to rely on. Node.js being a well-supported environment made all the functionality easily accessible and modifiable by any member of the team. By keeping the dependency amount low, and the dependencies themselves low-level, server size was kept low, ensuring easy deployment to any platform. Heroku was chosen as the hosting service due to the ease of setup and maintenance. However, the same application can be run from a dedicated server or any cloud-hosting platform with almost no modifications.

Overall, the current system turned out to be exceptionally flexible, with only the build machine being tied to a specific provider. Every other part of the pipeline is 100% modifiable and can be switched off, moved to a different platform and turned back on in a matter of hours, without affecting other steps of the pipeline.

Assuming an average working day of 8 hours and a cost per working hour to the employer of 30eur. Referencing the amount of time it took the author to set the system up with no knowledge, the assembly of the pipeline in a worst-case scenario would cost the company roughly 4500eur. The quickest setup using already existing software could take as little as 450eur(Table 2). Considering these numbers, it would take 180 days for a team with no experience or 12 for a reasonably-experienced team to outweigh to investment cost, thus saving the company anywhere from 5000eur to 20000eur a year.

Additional savings that were hard to calculate include the development time spent on finding unexpected bugs related to the build process. Bugs avoided due to implementation of tests and their automation are almost impossible reduce the time saved on finding the cause of an. Although estimates for these costs are difficult to calculate they add up quickly and at a certain point start taking up more time than the actual development of the product, as reported by many surveyed developers.

Table 2. Cost comparison between non-automated and automated workflows with different team experience levels, assuming an average developer cost of 30eur/h

Task	**No automation**	**Automation(from scratch, no experience)**	**Automation(experienced developer)**
Setting up testing environment	0	32	2
Writing tests	0	1	0.5
Manual testing	2	0.25	0.25
Setting up build automation	0	56	8
Manual building	1	0	0
Manual distribution	0.3	0	0
In-house acceptance testing	0.3	0.3	0.3
Deploying to customer	0.5	0	0
Setting up distribution	0	56	5
Total time spent daily	2.1	1.3	0.8
Once-per-project time commitment	0	144	15
Up-front investment €	0	4320	450
Daily Spending	63	39	24

4 CONCLUSION

The main objective of the thesis was to create a product delivery process that removes the need for human attention from all the steps possible and requires minimal maintenance. This objective was fulfilled. Every part of the delivery beyond development requires no developer interaction unless a step needs to be reconfigured. The pipeline has been used in production with positive results.

The system was designed in accordance with initial requirements, based on the current best CD practices. A version control system runs on GitLab platform and handles file manipulation and test automation. The build server utilizing Unity Cloud Build platform listens for a trigger from GitLab and creates build packages in a clean environment. After the package is finished, a Node.js application is notified of its location and the package is uploaded to the needed distribution platforms.

The application is used in the production environment, which helped the author identify points of possible improvement for the system. The pipeline will not be improved on further unless requested by the commissioner of the thesis. However, possible features added or modified are: implementing web application in the Function as a Service structure, transferring the build process to owned hardware to improve performance, and setting up a proprietary distribution platform.

REFERENCES

Beck, K. (2014). Test-driven development by example. Boston: Addison-Wesley.

Chacon, S. (2009). Pro Git. Praha: CZ. NIC.

Jenkins.io Jenkins Press Information. Retrieved from https://jenkins.io/press/

Kim, G., Debois, P., Willis, J., Humble, J., & Allspaw, J. (2017). The DevOps handbook: How to create world-class agility, reliability, and security in technology organizations. Portland, OR: IT Revolution Press, LLC.

Kim, G., Behr, K., & Spafford, G. (2018). The Phoenix Project: A novel about IT, DevOps, and helping your business win. Portland, OR: IT Revolution Press.

Mosley, D. J., & Posey, B. (2002). Just enough software test automation. Upper Saddle River, NJ: Prentice Hall PTR.

Newzoo.com. Global Games Market Revenues 2018 | Per Region & Segment. Retrieved from https://newzoo.com/insights/articles/global-games-market-reaches-137-9-billion-in-2018-mobile-games-take-half/

Sink, E. (2011). Version control by example. Champaign, IL: Pyrenean Gold Press.

Smartbear.com. Test Automation Frameworks. Retrieved January 15, 2019, from https://smartbear.com/learn/automated-testing/test-automation-frameworks/

Unity Technologies. Building for iOS. Retrieved February 10, 2019, from https://docs.unity3d.com/Manual/UnityCloudBuildiOS.html

Unity Technologies. Unity Test Runner. Retrieved January 10, 2019, from https://docs.unity3d.com/Manual/testing-editortestsrunner.html