Long Nguyen Ba

Enterprise-grade CI/CD pipeline for mobile development

Metropolia University of Applied Sciences

Bachelor of Engineering

Name of the Degree Programme

Bachelor's Thesis

3 May 2022

Abstract

Author:	Long Nguyen Ba
Title:	Enterprise-grade CI/CD pipeline for mobile development
Number of Pages:	57 pages + 0 appendices
Date:	3 May 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Solutions
Supervisors:	Toni Spännäri

Nowadays, software plays an indispensable role in human life. It exists almost anywhere people go and in anything people do. Testing and deploying pieces of software are also vital parts of software development. As a process, software development is also time-consuming and an error-prone processes. Thankfully, CI/CD technologies are built to automate mundane and labor-intensive parts.

The final technical target of this project is to introduce a fully operational and production-ready CI/CD pipeline for mobile application development. GitHub Actions as a CI/CD service and Fastlane as an assisting tool will be adopted and highlighted. Moreover, this thesis discusses common architecture approaches in mobile development as well as examine the comparison between each approach. The report also reflects the importance of how a good software architecture benefits the overall quality of the product while improving CI/CD workflow.

By leveraging modern CI/CD technologies, the objective of the report was successfully accomplished.

Keywords: CI, CD, Continuous Integration, Continuous Delivery

Contents

List of Abbreviations

CI:	Continuous integration is a set of practices in which frequent, small changes are immediately tested when migrating into a larger codebase. The developer will be notified about the test status.
CD:	Continuous delivery is a set of practices used to automate application re-leasing processes in short cycles.
GUI:	Graphical user interface, a user interface that enables end-users to interact with lower-level API.
API:	Application programming interface, a specification on how different software can discuss and communicate with each other.
OS:	Operating system, software that controls computer hardware.

1 Introduction

During the past years, software has emerged as an indispensable part of almost every business. (Herbsleb, J.D. et al, 2001). The process of developing software usually comprises analyzing requirements, planning, design, development, testing, deployment, and maintenance. Software testing is the process of verifying if the software's behavior aligns with the product's requirements (Srinivasan, D et al, 2007). With the help of increasingly advanced testing tools, it establishes confidence that the product is performing as expected and discovers errors before shipping to customers (William, E. Lewis, 2017). By properly implementing testing automation, a business will benefit from saving time and costs, enhancing software quality, and performing challenging tasks that can hardly be achieved with manual testing. (Bernie, G et al 2009)

1.1 Technical objectives

The final goal of this report is to introduce the modern CI/CD workflow used for mobile development. By analyzing GitHub Actions as the CI/CD service and Fastlane as the supporting tool, the considerate benefit of CI/CD will be explained and compared with the traditional methodologies.

1.2 Report structure

The table below outlines the main points of the report and the associated content with each chapter.

Section	Title	Contents
Section 1	Introduction	Illustrating the main technical objectives of the report. Summarizing the main points of the report.
Section 2	Theoretical background	Presenting theoretical background on CI/CD Analyzing GitHub Actions and Fastlane
Section 3	Conclusion	Recapping the report’s goals and main outcomes. Evaluating the report.
	References	Listing all resources used in this report.

Table 1. Report structure

2 Theoretical background

In this chapter, fundamental theoretical concepts will be provided for a detailed understanding of the research topic. Furthermore, the manual methods of conducting testing and delivering software are also discussed and compared with an automated CI/CD pipeline. Based on that, the clear benefits of adopting CI/CD will be highlighted.

2.1 Manual testing and deployment processes

Figure 1 demonstrates exemplary steps to produce software. Firstly, the requirements are gathered and documented. It is vital that the requirements are collected properly based on customer needs. The next step is planning. It involves project scheduling based on collected requirements and team resources. As the last two phases are finished, it is time for the developers to produce software. Depending on the software project, different programming languages could be used. This stage may also include documentation creation.

Once the programs are implemented, they need to be tested to verify whether they function correctly. The testing process may include different testing approaches: white box test, black-box test, integration test, performance test, and regression test. (Srinivasan, D et al, R. 2007). After being thoroughly and extensively tested, the programs are deployed to the customers for actual usage and the maintenance phase is entered. As observed, the software development process is complicated and labor-intensive, especially in the testing and deployment stage. Nowadays, customer demand is growing at a rapid speed that drives the need for software to be shipped with high quality and more frequently. (Bernie, G & Elfriede, D & Thom, G .2009). As a result, a tester team will face increasing workloads with limited man-hours. Those constraints once again emphasize the importance of an automation pipeline for testing and delivering a software product.

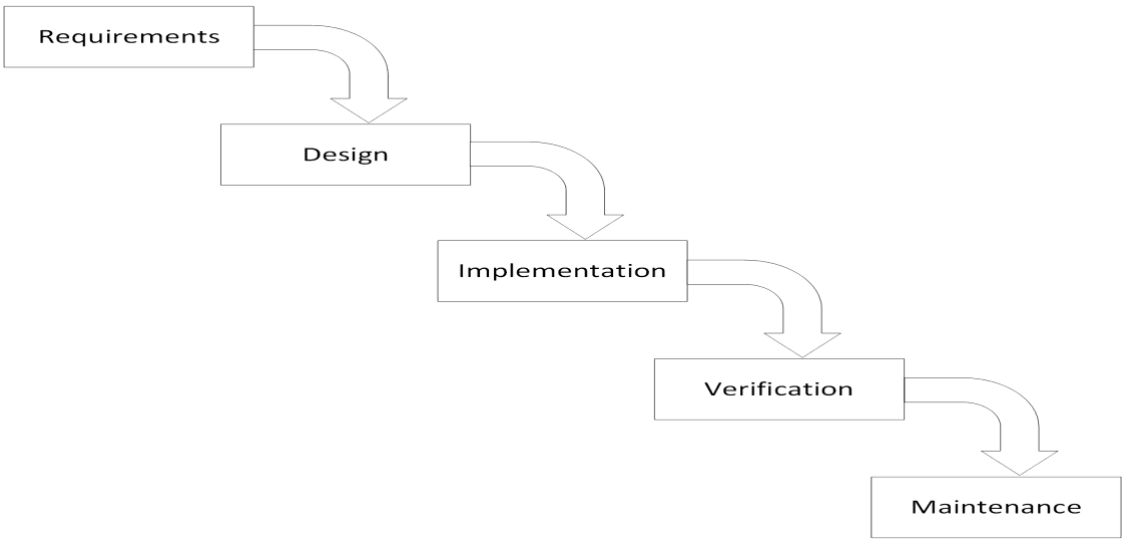


Figure 1. Software development processes (Joseph, I. 2018)

2.2 Continuous integration and continuous delivery/deployment

2.2.1 Continuous integration

The existence of continuous integration has been around for years. Fundamentally, continuous integration is a set of rules applying to the working procedures of engineering teams. The central idea of continuous integration is that all software developers commit to the master branch regularly preferably on daily basis. When team members create and integrate smaller changes into the master branch repeatedly, the value that it brings exceeds just the development operation. (Mathias, M. 2014). Continuous Integration improves feedback cycles. As a result, the application state will be closely monitored several times per day. It also can be used as an early-detection bug tool to ensure developers will be immediately notified in case new defects are introduced, thus enhancing the general software quality. (Andrew, G et al .2007)

From figure 2, the continuous integration pipeline involves a few components:

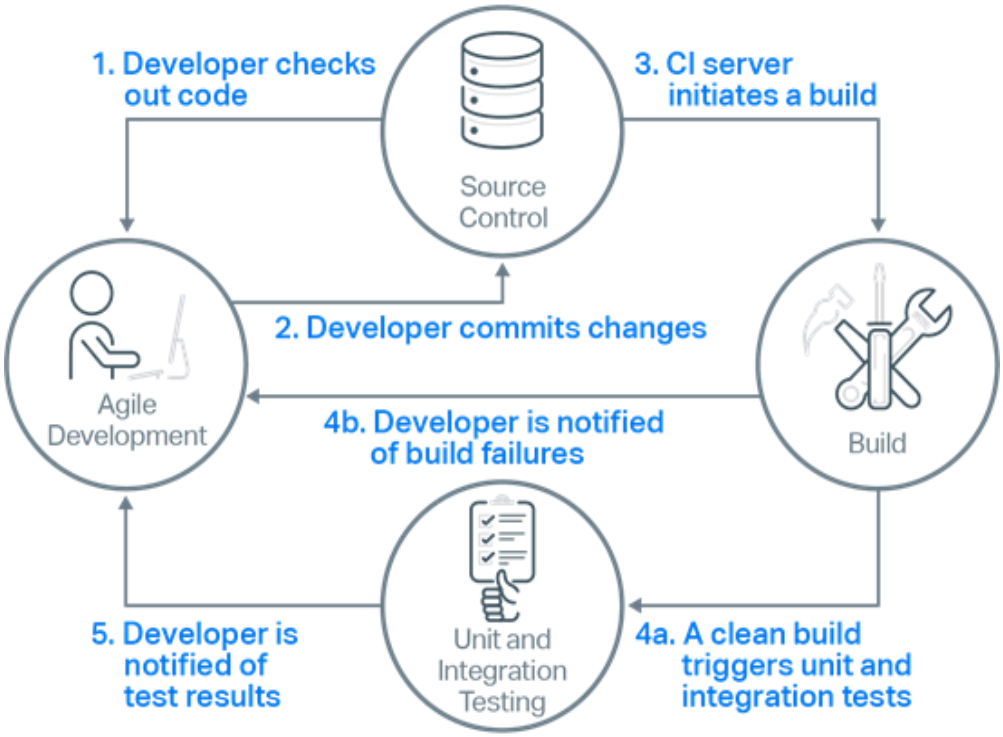


Figure 2. CI workflow (Microfocus 2020)

The CI server

The CI server is considered the heart of a CI flow. The CI server's job is to observe and be notified if there is a change in the code repository. It will automatically fetch the latest changes and try to build the project and execute any linked unit/integration tests. After that, it will send a notification to the developers about the state of the current code. From that, developers can choose to proceed, or fix caught bugs depending on the result. With different configurations, the CI server can additionally create planned, recurring builds from the current code in the repository. (Brent, L. 2020.)

Unit tests

To assist the CI server on whether a build is considered successful, we need tests. Developers, as the ones who understand how blocks of code and services interact with each other the most, will be responsible for this task. In the usual workflow of Continuous Integration, source code is updated and modified by multiple developers in their local environment. As code changes to the repository can be regular, it is required that unit tests are fast to run. It should not involve asynchronous jobs such as network requests, external database queries, etc. If such asynchronous actions are compulsory, then mocked object methodology could be used to mimic the behavior of the blocks of code/services without waiting for the responses. (Brent, L. 2020.). By following this change, the waiting time for the CI server to compile will be dramatically reduced which overall boosts feedback iteration for developers.

Prechecks

Prechecks are an additional feature that can be used on the local codebase before local changes are pushed to the remote repository. Hooks or triggers are some simple forms of prechecks. They are operations that can be executed before or after a source code event. Nowadays, big hosting sites offer an additional variant of prechecks. When a pull request is made, it automatically triggers the CI server to build and run tests that ensure the proposed changes will not break the codebase.

2.2.2 Continuous delivery

Continuous delivery is a collection of processes that enable delivery teams to rapidly distribute software deliverables to customers (Pete, H. 2020). Ultimately, the sole purpose of Continuous Delivery is to analyze and optimize the software release process (Eberhard, W 2017). The whole Continuous Delivery operation involves fetching code changes, building, testing, and lastly packaging to produce a final build. No human intervention is usually needed at this stage (Brent, L.

2020). With the help of Continuous De-livery, several enhancements have been achieved:

Firstly, more deployments will be taken sometimes up to several times a day which consequently reduces the development time of new features. More deployments also mean faster evaluation of new features and code improvements. As a result, the software developers do not have to remember the code that they delivered a few months ago. (Eberhard, W. 2017.)

Secondly, the more frequent release also results in a better-quality product. With the help of automation tools and automated testing, the software will become more resilient to potential technical errors which, in turn, will enhance the user experience as well as user engagement of the product. (Doron, K. 2018.)

Thirdly, Continuous Delivery improves competitiveness significantly. Thanks to CD, it is now possible to push new features more frequently which helps to test new ideas and business models. Thus, competitive advantage will be boosted as more ideas can be assessed. It is easier to choose the right path for the business. (Eberhard, W. 2017.)

Finally, continuous delivery can be seen as a method to mitigate the risk when releasing software. Manual release processes are complicated and error-prone. It requires efforts and co-communication between several parties. Each release is dependent on multiple manual adjustments which trivially leads to errors. If the errors are not caught before the new software version is launched, the consequences can be radical. (Eberhard, W. 2017.)

Figure 3 illustrates the main benefits of continuous delivery.

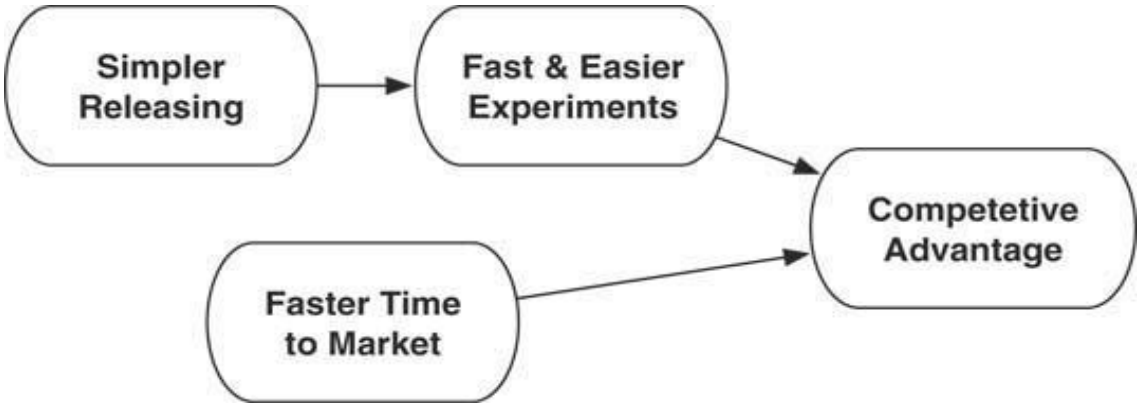


Figure 3. CD benefits in enhancing competitive advantage

2.2.3 Continuous delivery pipeline workflow

Figure 4 indicates a typical workflow of the continuous delivery pipeline.

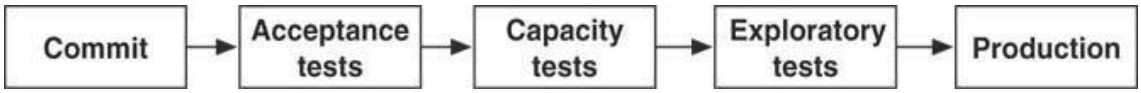


Figure 4. Continuous delivery pipeline (Eberhard, W. 2017)

Activities at the commit stage are included in Continuous Integration processes such as building, running unit tests, linting checks, and code analysis (Eberhard, W. 2017).

The next step is acceptance testing which mainly involves automated tests. The test type ranges from unit tests to automated tests via API and tests via GUI. The tests via GUI or API play a tremendous role. They, in the end, represent the system requirement while being a valuable addition to unit tests. (Eberhard, W. 2017.)

Capacity tests assure that an application can perform effectively for an expected number of users. Automated tests should be leveraged to precisely reveal whether the software is adequately fast. It is necessary to have the capacity tests performed in a production-like environment with full data volume.

However, due to hardware constraints, this is often not feasible. Apart from the environment, the user data need to also reflect the one in production. Otherwise, the outcome of the tests will not be helpful. Besides performance, scalability is also considered in this step. Scalability will indicate how many requests a system can handle at the same time. Poor scalability will lead to the system being accessed by only a few users at once. (Eberhard, W. 2017.)

During the exploratory tests phase, conducted tests are mainly manual. With most of the tests being automated at the acceptance test, the main target at this point is to carefully test the new feature functionalities. The distinctive feature of manual exploratory tests is to understand software from a domain viewpoint. Besides, the exploratory tests are utilized to improve the software in terms of usability, security, or overlooked corner cases. It can be a beneficial tool to closely investigate an unrealizable part of the product to identify uncaught errors. (Eberhard, W. 2017.)

The final deployment into production solely involves the installation of the software into the production environment. It is, therefore, considered to be a somewhat low risk process. To further minimize the potential risk during the release to the production process, different approaches are applied and discussed below as the consequences of a breakdown system at this point are significant. (Eberhard, W. 2017.)

Software deployment is a risky process that results in usage outages for the end-users (Eberhard, W. 2017). In the following sections, various means to protect the production environment while deploying updates will be discussed.

Rollout and Rollback

Risk management is especially essential during the software rollout. When a newer version of the product is released, it has the option to roll back to the older version in case problems occur. The main benefit of this approach is that it is a simple operation. At the end of the day, the old version of the software was deployed into production and the needed processes should not be different from

the one for the new release. The only uncertain work is the handling of the database, particularly in the case of huge datasets. (Eberhard, W. 2017.)

Roll forward

Roll forward can be used as an alternative for rollback. Instead of rolling back to an older version of the product, with this approach, a newer release will be tested and then deployed in the case of appeared errors. The associated cost of this approach is not pricey thanks to the continuous delivery pipeline. The advantage of roll forward is primarily the speed of deployment and simplicity. A typical rollback would require a lot of working hours. Additionally, dealing with database changes in a rollback is not an effortless task, whereas, in the case of roll forward, the database often remains as it was. After all, a roll forward is also a sign that the continuous delivery pipe has been performed properly. (Eberhard, W. 2017.)

Blue/Green Deployment

With this approach, the newer version will be deployed on an entirely detached system. To finalize the deployment process, a switch from the current environment to the newer environment must be triggered. The biggest benefit of Blue/Green Deployment is zero downtime. Moreover, pushing to a completely different system will enable more thorough testing on the new environment in terms of performance as well as function. (Eberhard, W. 2017.)

Canary releasing

Named after a tactic used in coal mining, Canary releasing is another choice to lessen the releasing associated risks (Eberhard, W. 2017). Firstly, a new version of the application is installed partly on several devices (mobiles) or servers (web services). The update will gradually be effective over a 7-day period until a point that all devices will use the new version. When problems occur with the update, the phased release can be paused. (Apple Authors. 2020.)

2.2.4 Continuous deployment

Continuous deployment is a set of practices to enable the code from the delivery pipeline to be deployed to production continuously. Different software products will have a distinctive approach on how to release the application, it may range from updating to the cloud, providing new updates to generating a new build in App Store/Google Play. (Brent, L. 2020.)

Being able to utilize the continuous deployment pipeline does not mean that every executable from the continuous deployment should be released. A team can decide themselves whether they need continuous deployment or not. (Brent, L. 2020.)

The main benefit of leveraging continuous deployment is that the required time for fixing bugs is further reduced. After deployment, a new bug fix can be trivially generated and automatically deployed to production in case of errors occur. Time is not the only thing that is beneficial from the continuous deployment, the product's overall quality will also be improved thanks to this practice. As changes are continuously ended up in production, it will force every team member to think twice before pushing code. Any careless change might lead to dreadful bugs for end users. Therefore, deployment pipeline and integration protection require heavy investments to catch potential issues as far as possible. With the new features, software under development can be instantaneously tested and released, user feedback and feature evaluation can be gathered rapidly which, in turn, enhances product flexibility based on market demand. (Eberhard, W. 2017.)

Besides the advantages that continuous deployment delivers, it also has several limitations that would need discreet consideration. As discussed, implementing a proper continuous deployment pipeline is costly. Additionally, the application architecture should be designed to support CI/CD from the beginning otherwise hidden problems will appear and eventually impose a negative effect on the team. This topic will be later discussed in section 2.3. The

cost of continuous deployment is not the only barrier, this practice also requires a high level of trust between developers as, in theory, anyone could release new code to production. It is preferable that the person who is responsible for the re-lease work in a different organizational unit. This prerequisite could trivially be integrated into the continuous deployment by applying the two-person principle. However, developers' responsibility will be declined as they would feel subjective and believe the tests should be able to find the potential bugs. (Eberhard, W. 2017.)

In figure 5, the distinction between continuous delivery and continuous deployment is outlined. In fact, both approaches share multiple common steps. The only factor that differs is in deployment to the production stage.

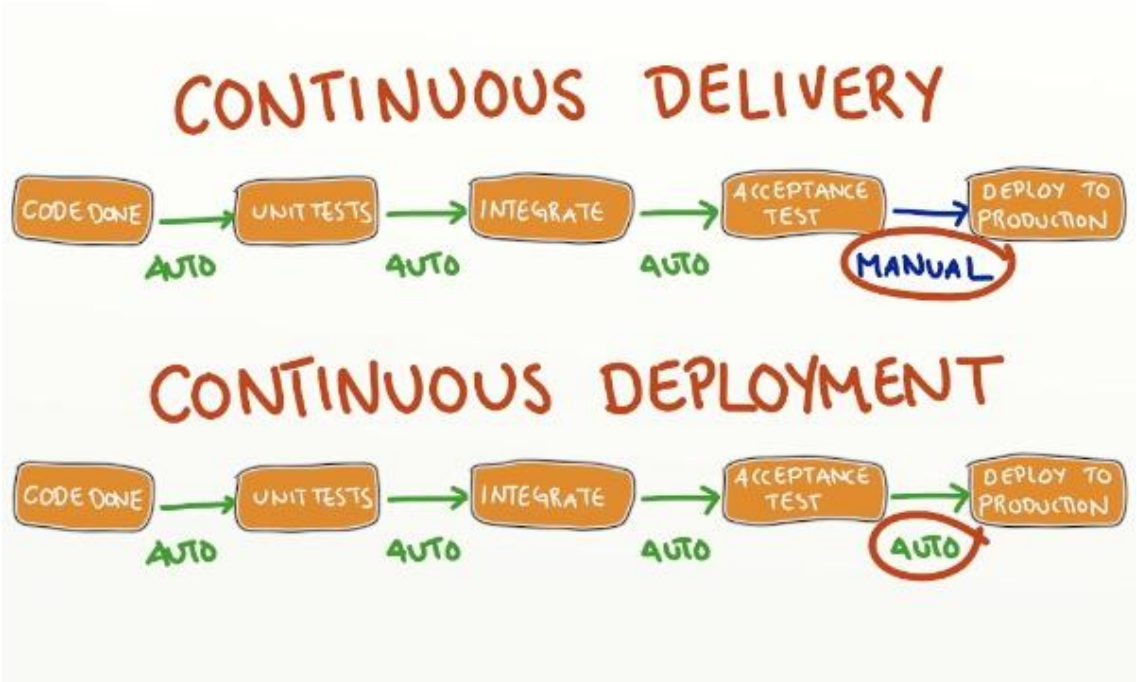


Figure 5. Difference between continuous delivery and continuous deployment.

2.2.5 GitHub Actions

GitHub Actions is an online platform that offers continuous integration and continuous delivery service used for automatically building, testing, and deploying software. Workflow is a building block that differentiates GitHub Actions from other players. Workflow can be leveraged to build and test open pull requests or to trigger the continuous delivery pipeline. (GitHub Actions Authors. 2022)

To configure an example workflow, a yml file is compulsory in the root directory of the project under the .github/workflows/ folder. The .yml file will be triggered by predefined events such as opening a pull request, commits merged to a certain branch, or simply based on a schedule. (GitHub Actions Authors. 2022)

One repository can own multiple workflows for various tasks. Tasks can range from running tests and deploying builds to external services to adding a label when a new issue is opened. (GitHub Actions Authors. 2022)

From listing 1, all necessary information that is related to how an application is built is listed. It defines the required OS, and condition that triggers the workflow, tools, and scripts that should be run. GitHub Actions components will be explained in more depth during the later section for conducting the actual project.

```
name: learn-github-actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

Listing 1. An example of content in a .yml file (GitHub Actions Authors. 2022).

2.2.6 Fastlane

For years, iOS developers have been struggling when dealing with Apple services such as App Store, iTunes Connect, and the Apple Developer portal. Dealing with provisioning profiles or granting access for a new teammate to code signing are frustrating tasks that nobody wants to do. Furthermore, pushing new screenshots to App Store every time an app's user interface changes is a laborious task. If the application supports multiple localities, then the work may even become more mundane. (Doron, K. 2018.)

Besides, releasing the staging app to testers or production app to App Store are tedious tasks that no developer would be willing to work on. To release the staging app, the app's bundle version number needs to be incremented when a new version is pushed to Git. After that, the app is signed with a proper provisioning profile. Once the signing is done, the developers need to generate an ipa file and then upload the executable to TestFlight. (Doron, K. 2018.)

Thankfully, all the labor-intensive tasks can be now automated with a technology called Fastlane. It enables a smooth flow and lifts any barrier to a continuous delivery pipeline by utilizing commands to achieve expected results with minimal effort (Doron, K. 2018).

Fastlane was created and open-sourced on GitHub by Felix Krause in 2014. After receiving recognition from the developer community as more people and companies start to adopt Fastlane in their CD pipeline, it was acquired by Twitter in the year 2015. One year after, it was also bought by Google for the Firebase mobile development platform. Regardless of Fastlane's ownership, it remains open-source and active ever since. (Doron, K. 2018.)

Fastlane is usually regarded as the simplest way to automate building and releasing mobile applications. Fastlane offers a wide range of features that support dispatching work autonomously such as (Doron, K. 2018):

- Automate building and packaging of iOS apps.

- Automate the taking screenshots process which can be configured to different screen sizes and languages.
- Automate upload artifacts such as .ipa files, and screenshots to iTunes Connect without using XCode.
- Automate signing certificates, provisioning profiles, and push notification profiles management.
- Synchronize certificates and profiles across multiple different teams.
- Automate managing testers in TestFlight.
- Automate test running in the application.

The heart of Fastlane is a Fastfile, which is a Ruby-powered configuration file. It supports multiple lanes for different targets. From listing two examples, the content of Fastfile serves two targets of the application CD process, one for beta users and one for production users. One lane combines a set of tasks that should be executed such as incrementing build numbers, installing dependencies, running tests, and even uploading app binary and artifacts to the App Store. (Doron, K. 2018.) Listing 2 indicates a typical format of a Fastfile.

```
lane :beta do
# Increment build number in XCode
  increment_build_number
# Build your app
  gym
# Upload to TestFlight
  testflight
end
lane :appstore do
  increment_build_number
# Run cocoapods install
  cocoapods
# Run tests
  scan
# Take screenshots
  snapshot
# Provisioning
  sigh
# Upload app, screenshots and meta-data
  deliver
# Run your own custom script
  sh "./customScript.sh"
  ...
# Notify your contacts on Slack
  slack
end
```

Listing 2. Example content of a Fastfile file (Doron, K. 2018)

2.3 Software architectures

Causing tons of confusion in the industry, the term “Software architecture” is usually regarded to define the skeleton of the whole system. It defines how each component in a system interacts with others. When architecting the software, four factors should be considered: structure, design principles, architecture characteristics, and architecture decisions as shown in figure 6. (Mark, Richards, et al, 2020.)

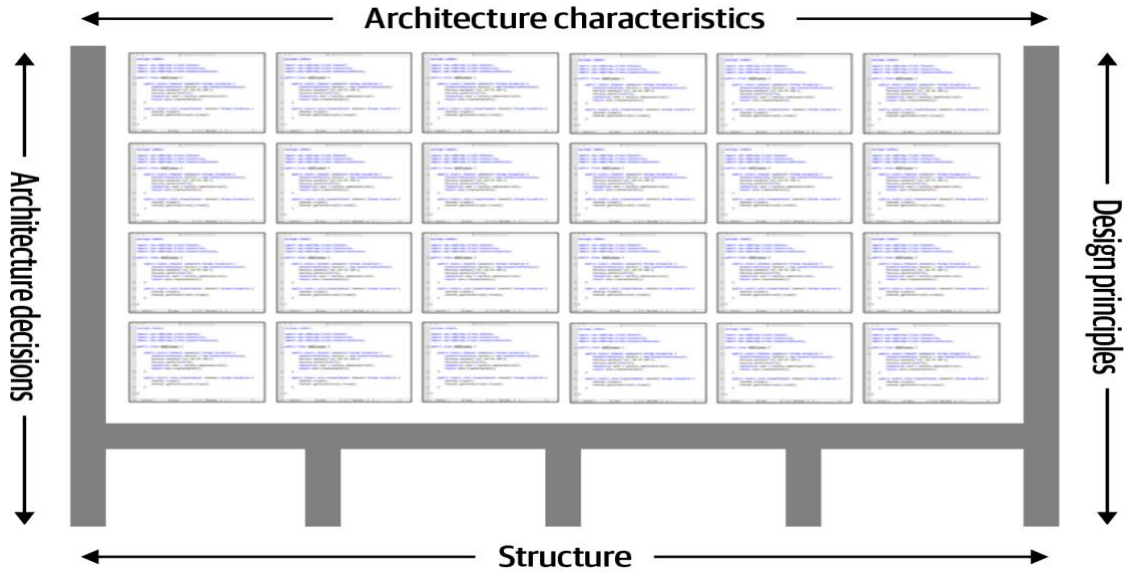


Figure 6. Elements that should be evaluated while architecting software projects (Mark, Richards, et al, 2020)

Structure

The structure indicates the architectural approach the system is using such as layered, microkernel or microservices. Structuring the software does not fully encapsulate the main goal of architecture. To implement an outstanding system, knowledge of design principles, architecture decisions, and architecture characteristics are required as well. (Mark, Richards, et al, 2020.) Figure 7 shows different structural types in software architecture.

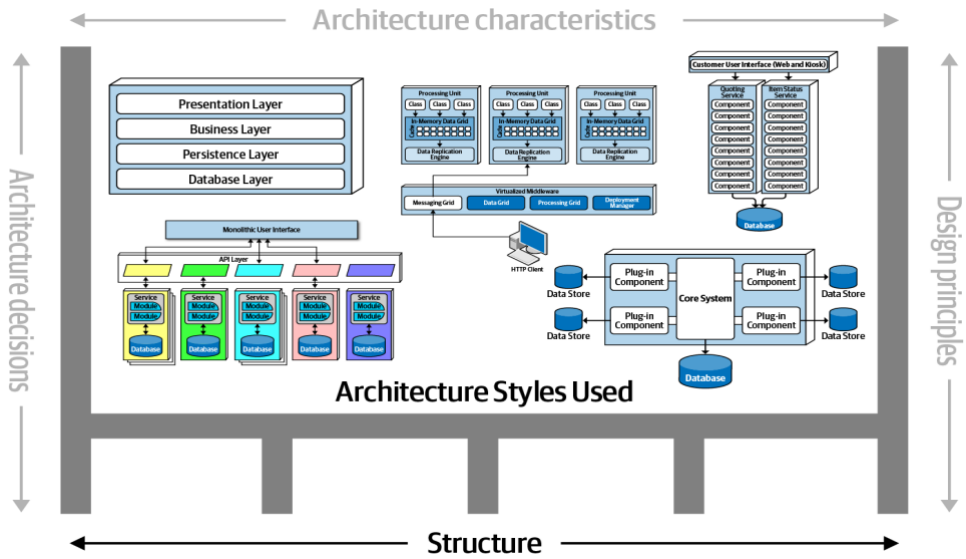


Figure 7. Structural types used in the system (Mark, Richards, et al, 2020)

Architecture characteristics

Another element that should be considered while deciding on software architecture is architectural characteristics. It defines how a system could be evaluated as a successful system by examining it from different angles such as availability, reliability, testability, etc. These metrics are commonly independent of software features. (Mark, Richards, et al, 2020.) Figure 8 shows the architectural characteristics aspect of software design.

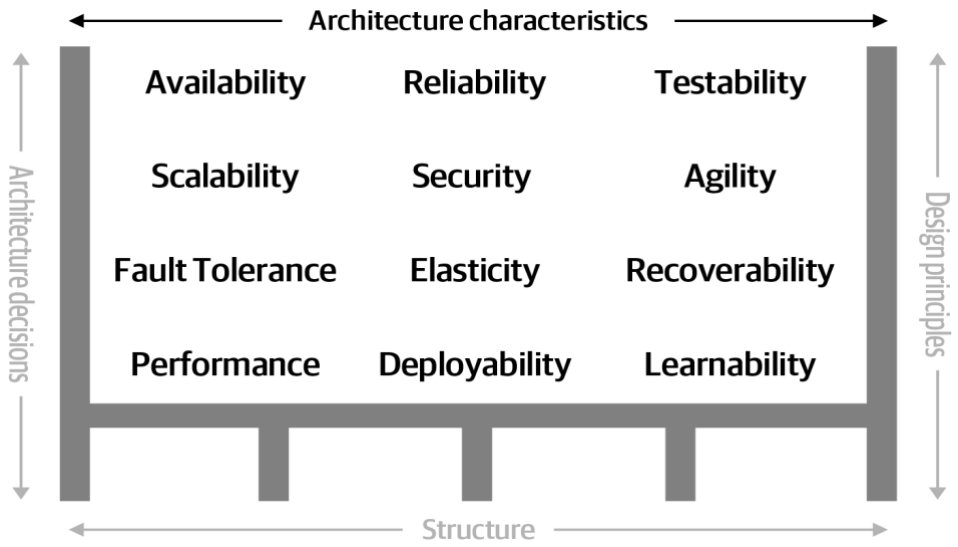


Figure 8. Architecture characteristics in the system (Mark, Richards, et al, 2020)

Architecture decisions

Architecture decisions specify the formulas a system should be composed of. It might involve setting separation between different layers in the application and only allowing data access from a certain place in the software. Architecture decisions construct the restraints of the system and act as a guide for the development team on what is and is not possible. (Mark, Richards, et al, 2020.) Figure 9 illustrates the architectural decisions in software design's angle.

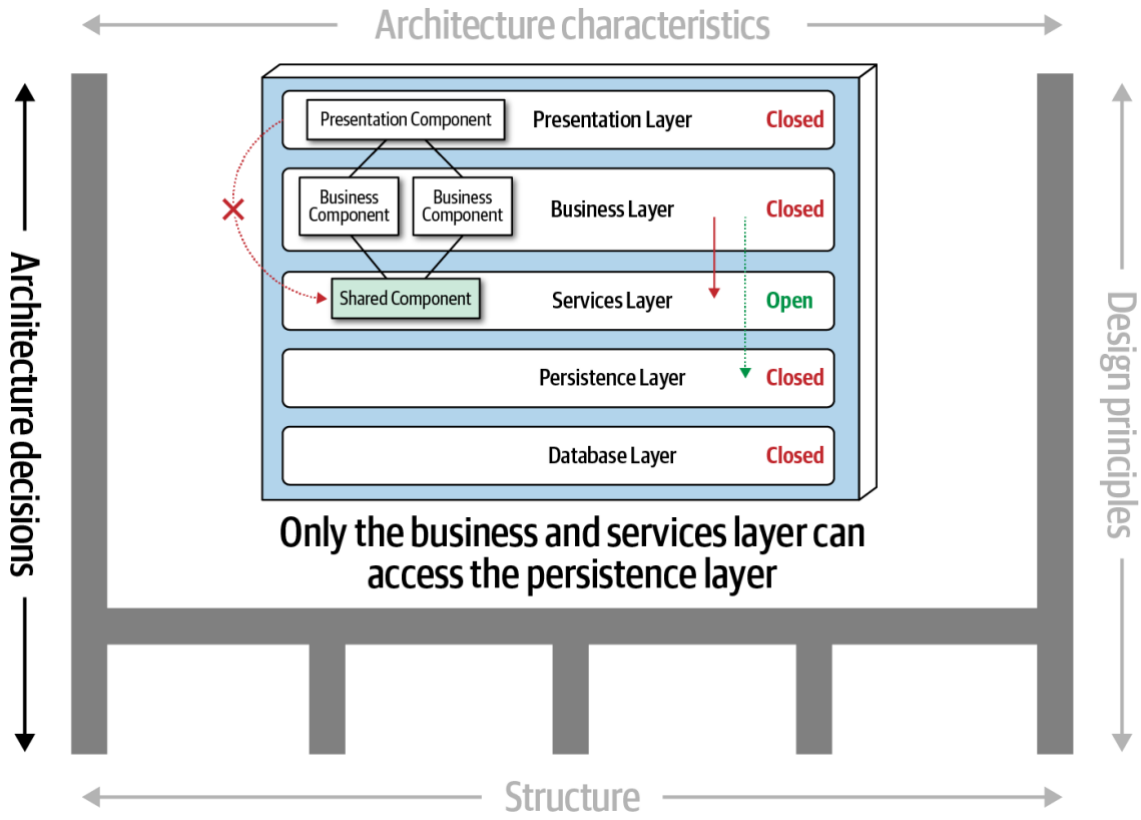


Figure 9. Architecture decisions in the system (Mark, Richards, et al, 2020)

Design principles

As a last factor to consider when architecting software, design principles are used as guidance instead of compulsory rules. For instance, when defining the

communication method between microservices in a microservice architecture, design principles can be leveraged to recommend the development team to use asynchronous calls to improve efficiency. The development team then can decide the suitable communication method such as gRPC or REST for different use cases. (Mark, Richards, et al, 2020.) Figure 10 outlines vital design principles in software architecture.

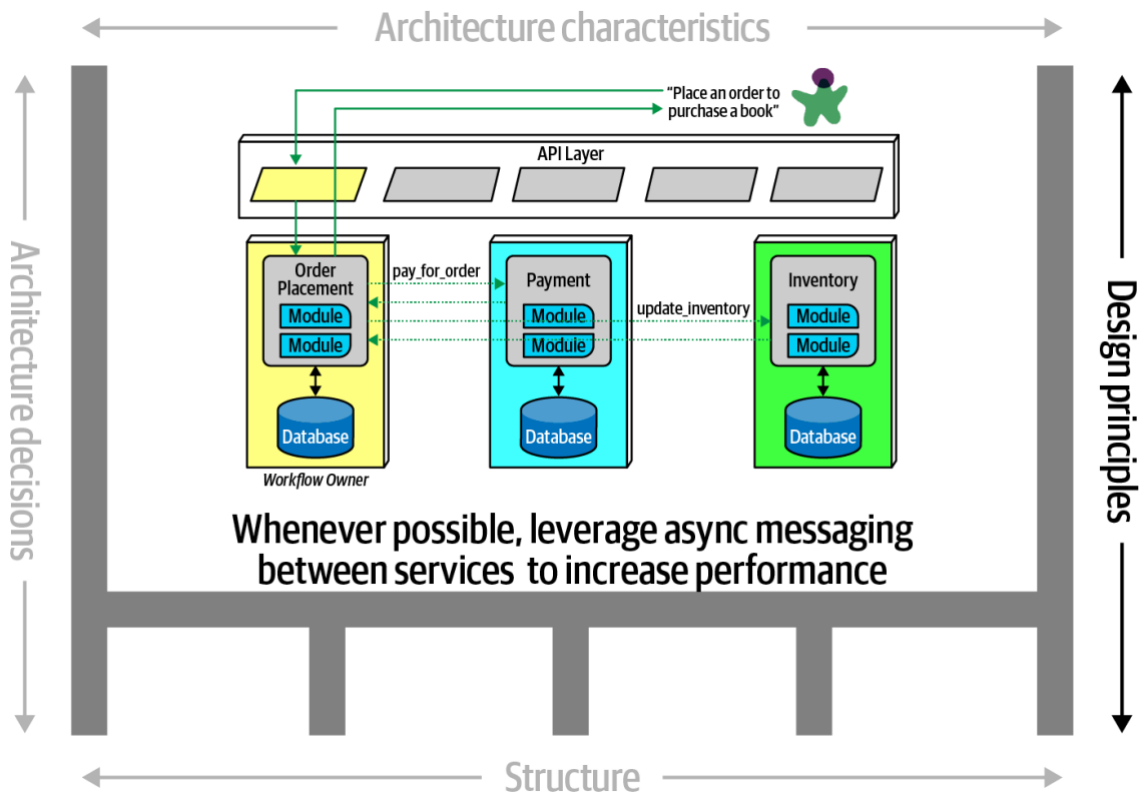


Figure 10. Design principles in the system (Mark, Richards, et al, 2020)

2.3.1 The MVC architecture

The MVC is a typical architectural approach in a Cocoa application. The MVC pattern has three different elements: model (M), view (V), and controller (C). By applying the MVC pattern, not only each role in the software application is clearly defined but it also encapsulates the logic of how each component interacts with each other. Each component in the MVC pattern has its unique job and is separated by an abstract border. (Apple, 2021.) MVC is extensively

adopted to create a rich user interface. The web and mobile OS are mostly utilizing this approach. The clear benefit of MVC brings is the loosely coupled models. View and Controller components can then be tested separately. (John Kouraklis, 2016.) It enables reusability among objects in the application and defines the interface more clearly (Apple, 2021).

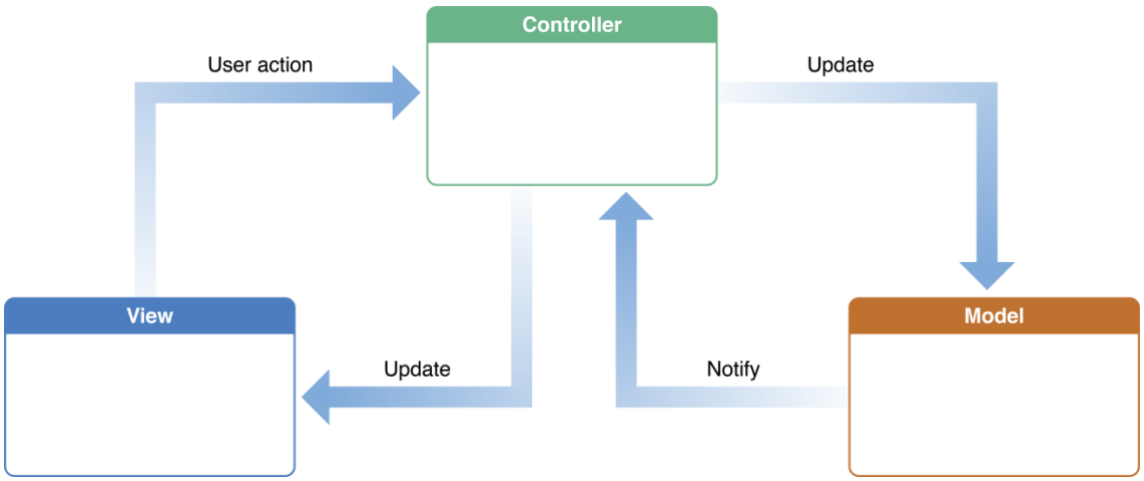


Figure 11. Model (M) – View (V) – Controller (C) workflow (Apple, 2021)

Model Objects

Model objects contain application-specific data. It also manages the data manipulation process. For example, a to-do list application may contain a to-do model, or an address book app may have a contact model. A model object can have a one-to-one or one-to-many relationship with other model objects which forms a model layer that contains one or more object graphs. Preferably, an object model should not be concerned or connected in any way to the view that uses the model data. A good rule to follow is to always divide the model layer from the view layer. (Apple, 2021.)

Communication: After an action is recognized in the presentation layer, it will be passed through the controller object which leads to model data creation or update. Once the model object data is updated, the controller object will be

notified, which then updates the view objects to present the correct state. (Apple, 2021.)

View Objects

A view object, which is usually regarded as UI, is an entity that can be seen and interacted with by users. The user interface comprises multiple view objects. The main goal of the UI is to present the model object data and facilitate the modification ability of that data. Despite the close relationship with data, the view object is generally loosely coupled with model objects in an MVC pattern. (Apple, 2021.)

Because view object is highly reusable, consistency between software applications is significantly enhanced. UIKit and AppKit are Apple view frameworks for iOS and macOS eco-system which provide collections of view classes to build a presentation layer. (Apple, 2021.)

Communication: Controllers receive events initiated by users and communicate back to update data in the model layer. In turn, once changes happen in the model object, it will trigger the updates to the controller objects which will lead to the presentation layer redrawn itself to correctly reflect the status of data. (Apple, 2021.)

Controller Objects

A controller object is seen as a bridge between view objects and model objects. Hence, it is used to pass object information to view objects and translate users' actions to modification to model objects. Setup and coordinating tasks as well as controlling application lifecycles can also be implemented by the controller (Apple, 2021.)

Communication: A controller object receives user actions from the UI layer and communicates updates to model objects. Similarly, when changes are recognized in the model layer, they will be transmitted back to the controllers for

processing and updating the view to reflect the current state of the data correctly. (Apple, 2021.)

2.3.2 The MVP pattern

The MVP (Model – View - Presenter) pattern is an interesting variation of the MVC approach. Similarly, there is also a clear separation between the presentation user interface and business logic. However, the role of the controller (C) in MVC was shifted to the presenter (P). (Joseph, I. 2018).

Model

The model in MVP plays a similar role as it does in MVC where it encapsulates the data and business models. Another responsibility would be interacting with external services such as databases or web services to obtain data and receive orders to fetch new data. When the data is obtained, it will be reported back to the presenter. The model itself should not hold any direct communication channel with the view under any circumstances. (Joseph, I. 2018).

View

The view component accounts for displaying the user interface as well as handling user interaction. When interaction occurs, the view is responsible to send the signal to the presenter for further processing such as fetching new data or update the current one. (Joseph, I. 2018).

Presenter

The presenter acts as a bridge to connect the model and the view. Each view following MVP should have a direct dependency on the presenter. The presenter, at the same time, owns the communication with models to fetch and update data from. One notable difference between MVP and MVC is that in MVP the view is not aware of the model. All the main logic was dispatched inside the presenter and then later communicate back to the view to update

itself accordingly. (Joseph, I. 2018). The overall interaction between components in MVP is shown in figure 12 below.

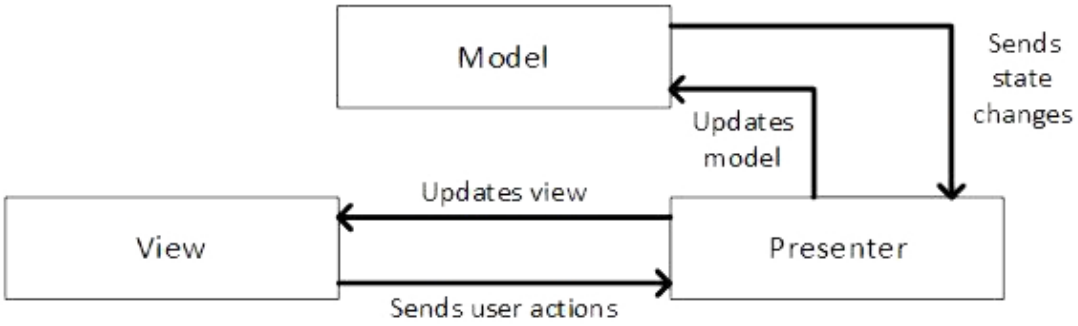


Figure 12. MVP (Model – View - Presenter) interaction. (Joseph, I. 2018)

2.3.3 The MVVM pattern

Being regarded as an option for MVC, MVVM comprises three parts: Model (M) – View (V) – ViewModel (VM). SmallTalk was the first to introduce this approach using Presentation Model as a name. The most noticeable benefit that MVVM brings is that it breaks the interlink that still exists in MVC between the view and the model. In the MVVM pattern, controller (C) is replaced by the ViewModel (VM) which leads to different accountability between View and ViewModel. (John Kouraklis, 2016.)

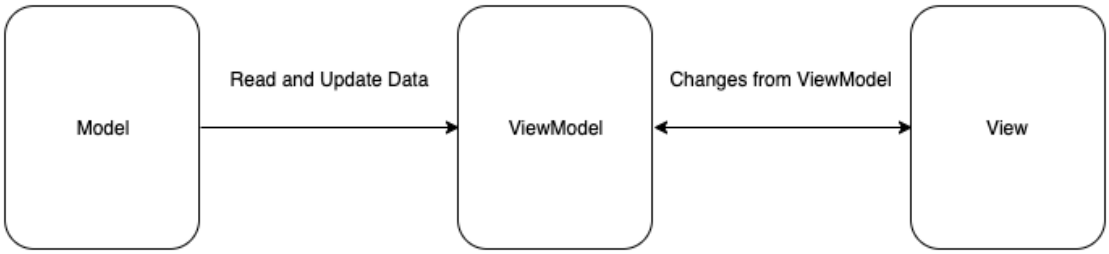


Figure 13. MVVM (Model-View-ViewModel) interaction (Chris Barker, 2020).

Model

The model part in the MVVM pattern is relatively comparable to the counterpart in MVC. It still contains different data sources (e.g., databases, files, or external API). Ideally, the model implementation should be as simple as possible. (John Kouraklis, 2016.)

View

The view illustrates data in the correct form, describing the data state, and it is also responsible for handling user interaction and events. Views in MVVM similarly involve minimal code, only necessary components are required to successfully display the UIs and handle events. (John Kouraklis, 2016.)

ViewModel

In MVVM pattern, the view model is the component that is ultimately responsible for representing the state of the view as well as handling the communication between the presentation layer and the data model. When user actions are detected in the view layer, they will be transmitted to the view model object for further processing which could involve data validation or getting new information from the server. Similarly, updates coming from the data layer will result in view state updates in the view model which will trigger UI is redrawn in the view layer. (John Kouraklis, 2016.)

Communication

In the MVVM pattern, the view has a direct dependency on the view model object. It sends events that are leveraged to update the view model states. Additionally, the view observes changes from the view model state and re-renders itself accordingly. The view model, on the other hand, should not have a dependency on the view at all. Similarly, the view model should own a connection with the model. This connection will allow the data to flow from the model to the view model without the model awareness of the existence of the view model. Any updates from the model will be transferred back to the view model so that it can update its state accurately. (John Kouraklis, 2016.)

2.3.4 Clean architecture

Clean architecture at the first glance

Over the past years, several different approaches to designing software have been created such as (Robert C. Martin, 2017):

- Hexagonal Architecture by Alistair Cockburn was later used in the book named Growing Object-Oriented Software with Tests written by Steve Freeman and Nat Pryce.
- DCI by James Coplien and Trygve Reenskaug.
- BCE, developed by Ivar Jacobson.

The mentioned-above architecture choices may vary in naming but share numerous common values as:

- Framework independence: Regardless of whichever framework is used inside the software, it should not dictate how software should be written. Frameworks should only act as a tool rather than a constraint that software depends entirely on. It will enable freedom of choice for the developers in case the framework needs to be replaced (Robert C. Martin, 2017).
- Testability: Any logic in the application can be tested without depending on any external sources (Robert C. Martin, 2017).
- UI independence: The representation layer should not affect the core logic of the software in case it needs to be altered, for example, from console to web UI. Changes in the UI shall not result in updates in other parts of the software (Robert C. Martin, 2017).
- Database independence: Ideally, the software should not rely on a specific type of database technology. Swapping between relational databases and non-relational databases is trivial without much extra effort (Robert C. Martin, 2017).
- Any external agency independence: Software business logic should not be aware of the outside world at all (Robert C. Martin, 2017).

Figure 14 provides an overview of the clean architecture.

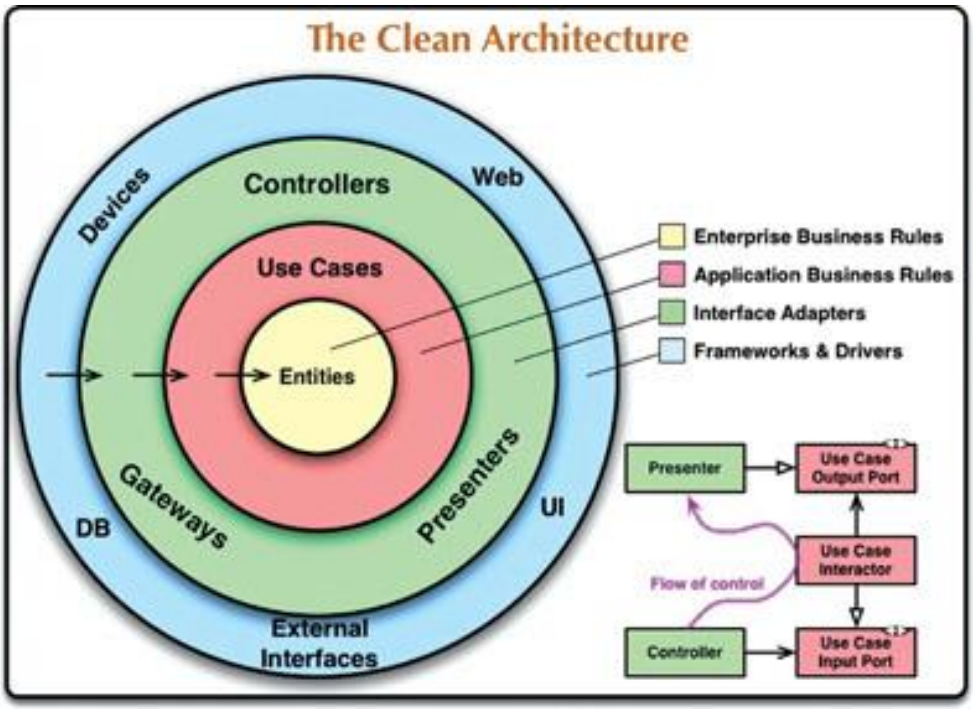


Figure 14. The clean architecture (Robert C. Martin, 2017).

The clean architecture dependency order

Figure 14 indicates different potential layers that might be included in the software. In general, the more inner the layer is, the higher level the logic will cover. The definite rule for following the clean architecture paradigm is an inward dependency which means that the outer layer should only depend on the next inner layer. Additionally, the inner layer knows absolutely nothing about the outer one which covers classes, functions, variables, etc. Data models that are declared in the outer layer shall not be consumed by the inner one. Generally, any links from the inner circle outward must be prevented (Robert C. Martin, 2017).

Entities

Entities could be an object with different functionalities or just a set of shared data structures or components inside your organization. If it is a commonly

shared piece of code that is leveraged by multiple software within the organization, it could be categorized as an entity. (Robert C. Martin, 2017).

When the software is written as a single application, entities will be the objects that cover the most bare-bone logic which has the least likelihood to change in case business requirements update. (Robert C. Martin, 2017)

Use cases

Use case layer will commonly include business-specific logic. It will serve all the use cases that the application needs to support and acts as a bridge of data flow from the entities layer. The existence of use case layers will ensure the application has a means of achieving its business targets. (Robert C. Martin, 2017)

Adjustment in the representational layer should not affect this layer and adjustment from the use case layer should not result in changes in the entities layer either. (Robert C. Martin, 2017)

Yet, changes to how software should be operated can lead to changes in this layer. (Robert C. Martin, 2017)

Interface adapters

As implied by the naming, the main responsibilities of the interface adapters layer will be to convert the data type from what is returned from the use case layer to that the application can consume. By having a separate layer for adapters, it is handy to use the data type that suits best for the entities or use case layer without worrying if the data type fits the external frameworks that the application relies on currently. (Robert C. Martin, 2017)

Frameworks and drivers

The frameworks and drivers layer host all the external dependencies that the application has such as web user interface, database, etc. The code in this layer only communicates to the inner layer. It is also the place where all the actual details are implemented. (Robert C. Martin, 2017)

By following the mention-above rules, it is trivial to maintain a clean software architecture that will benefit the application both in terms of cost of maintenance and testability. By dividing software into multiple independent layers which account for a unique responsibility, it will assist tremendously in creating scalable software where each component can be easily plugged out and replaced in case it becomes obsolete. (Robert C. Martin, 2017).

3 Case study

3.1 Case summary

This case study was executed to set up a production-grade continuous integration and continuous delivery pipeline as a start-up idea for a new travel assistant mobile application. The team size that is responsible for delivering the application is roughly four people with backgrounds mostly in software development. The travel assistant project is the first step in the start-up toward creating a seamless traveling experience for the end-users. The pressure to develop a minimum viable product is pressing as other competitors released similar tools. From the team members' experience, it is advisable to have a proper CI/CD pipeline set up from the start of the project to ensure the agreed timeline for the product to enter the market as well as the high quality of the new mobile application.

The main target of this case study is to automate the mundane parts of the software development cycle such as integrating new code into the codebase or deploying manually a newer version of the product to end-users. Hence, it assists developers to shift their energy into delivering new features and continuously improving the software. It is also equally important to set up a common workflow so everyone in the team understands and can work independently with a minimum fuss. After having a sync meeting within the team, the scope for the very first release will include a working CI/CD pipeline and project skeleton that strictly follows a clean architecture mindset for scalability. Having a full-featured application is not on the priority list now.

By conducting a clear planning session, a feasible timeline, as well as technology choices, were carefully chosen to meet the business requirements. Among numerous CI service providers, GitHub Actions was selected as it is handy to integrate the current repository hosted in GitHub. Fastlane which is

popular and well-documented among mobile developers was chosen as the tool that supports running tests and automating the generated build distribution process to Apple's TestFlight. It is agreed within the team to experiment the Apple's new UI engine called SwiftUI for declarative UI development. With the technology choices made, the first delivery was successful, and it demonstrates how effective a CI/CD pipeline can be to deliver more reliable software in a less time-consuming cycle.

3.2 Case challenges

Shortly after the project was initiated, the whole team had their initial meetings to collect business and technical requirements. Various aspects were discussed.

Firstly, it is critical to select the right technology used to develop the solution. In the current market, numerous players provide CI services for mobile development such as BitriseCI, TravisCI, and CircleCI. Each provider has its own strengths and weaknesses that require a careful evaluation. Unfortunately, not everyone has experienced all the CI providers which cause hesitation among team members to select the right one. Deciding the tool used for automating the testing, code signing, and distributing the build to Apple's TestFlight service is straightforward as Fastlane stands out in the crowd. It is an open-source tool that speeds up the whole development cycle and it can be seamlessly integrated into any CI service provider including GitHub Actions. However, the choice to select the UI engine to build the user interface in the iOS platform is no easy task. There is a debate on whether the team should stick to the UIKit framework which was proven to be stable and more time-consuming to build UI or experiment with SwiftUI which is newer, slightly less stable but can be used to rapidly develop the user interface.

Choosing the right application architecture is another challenge in the project. Traditionally, in the iOS world, a considerable number of applications were built using MVC architecture. MVC was not a bad choice for building apps. However,

when building software applications at a more global scale, it tends to lead to a massive controller problem where the controller knows too much of the details about the view and model. Consequently, the line of responsibility boundary fades and it will result in unmaintainable code. On the other hand, applying an extreme architectural approach where each component is strictly separated to own an excessively small responsibility will lengthen the new feature delivery time. Hence, a more balance choice needs to thoughtfully be considered.

Lastly, it is vital to analyze the current team capacity and adjust the plan accordingly. Even though everyone in the team has a software development background, the testing capacity is still lacking. It is rather easy to forget to run tests before opening any pull request to ensure no new defect is introduced to the codebase which emphasizes once again the significance of an automated pipeline.

3.3 Solutions

Despite the many challenges faced from the start of the project, the team managed to resolve all the impediments and successfully implemented the CI/CD pipeline.

Among numerous CI service providers, the whole team, after all, decided to use CI services from GitHub Actions to have the full CI/CD flow integrated into GitHub. As mentioned previously, the company codebase is hosted on GitHub. It is a major advantage as it would ease the integration process and allow developers to shift their focus on more important topics. With the native support from GitHub, it is easier than ever to build, test, and deploy software. Another advantage of GitHub Actions that is worth mentioning is that it supports open-source actions. Everyone can publish their actions which do a set of tasks and others can decide to adopt if desired.

To maintain consistent code quality, it is agreed within the team that the pull request opening event will trigger unit tests. By that mechanism, any new defect

will be caught before it finds its way to the main codebase. For the continuous delivery pipeline implementation, and merge to the main branch will trigger the pipeline to run through various types of tests and generate the build afterward. As the last step in the CD pipeline, the build will be uploaded to TestFlight for beta testing before releasing to production.

Swift was chosen as the primary programming language for developing the application thanks to the ease of use and the existing knowledge that current team members possess. As discussed previously, there is a controversy on whether to adopt Apple's new UI engine called SwiftUI or should the team continue to leverage UIKit. SwiftUI was after all selected as its benefits outweigh all the negatives. By using SwiftUI, the user interface can be rapidly developed without any hurdles. All the potential issues with adopting SwiftUI only arise when embedding the SwiftUI view inside a UIKit application. As the current application is still in its infancy, there is no legacy system to deal with which means that the application user interface can entirely be implemented using SwiftUI.

As for the application architectural choice, the development team decided to utilize the MVVM pattern. With MVVM, it is promising to have an application where each part has enough responsibility that should not be an overkill component to maintain in case the software scales. A part of the logic inside the controller in MVC will be transferred to the view model in MVVM which benefits the developers should there be a case for debugging or writing tests.

3.4 Implementation

The solutions in the previous section mitigate challenges faced during the planning phase. The overall implementation of the case study is illustrated in the following ways. First, the iOS application architecture will be addressed and explained. All the software layers and components will be revealed based on their necessity and significance. After that, the actual implementation of the pipeline will be the main discussion. Specifically, the GitHub Actions integration

process into an iOS project and assembling a production-grade CI/CD pipeline will be in detail discussed.

3.4.1 Application architecture

Software architecture plays a significant role in implementing an effective CI and CD pipeline. The application is a trip planning tool for iOS users which will be a valuable addition to the company's current app ecosystem. The application follows a clean architecture paradigm. Figure 15 describes the main components and layers inside it.

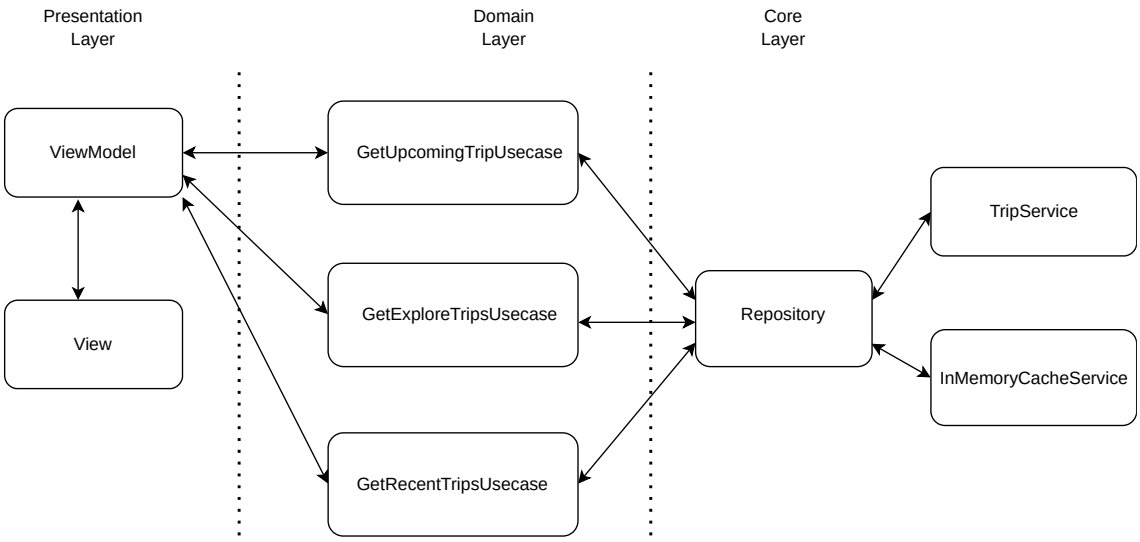


Figure 15. The iOS project's overall architecture

The application was divided into three distinctive layers. Each one owns different responsibilities that will be addressed sequentially.

The presentation layer is the foremost one. Its main job is to ensure the application has the necessary user interface based on the design. It contains two components: View and ViewModel. The View solely accounts for the UIs of the application as well as handles user interactions. All the responsibilities related to business logic will be executed inside ViewModel. The ViewModel

additionally owns the state of the and dictates which state the view should be in so it can render itself accordingly.

The next layer that requires attention is the domain layer. The goal of the domain layer is to act as a bridge between UIs and the core logic of the application. The domain layer generally comprises use-cases and models that are business-oriented. In the case study, three use-cases were implemented. Each has a unique job that it oversees.

The inner-most and last layer is the core layer. In the core layer, only the most fundamental services are included. In the case of the application, it supports Trip Service that supplies all information related to trips including upcoming, recent and explore ones. InMemoryCacheService object will provide the mechanism to cache the fetched trips in memory which reduces the need for firing constant network requests. Those services are the least likely to be affected by external factors.

On top of the services, an additional repository object is created to ease the work to connect different services. It will be liable for not only gluing services, but it should also contain business logic on how services should be connected.

Thanks to its different layers and clear responsibilities, it is trivial to start adopting unit tests in the application. All the components are protocolized which, in turn, ensures that each component can be mocked to create expected behaviors during testing processes. For this reason, the application in the case study can be unit-tested fully which will boost developers' confidence in the CI pipeline.

3.4.2 CI/CD pipeline implementation

Branching strategy

Before proceeding with the implementation of software, it is vital to adopt a suitable branching strategy. In the case study, the development team decided to

create two main branches: main and dev. It is agreed within the team that the release cycle will be two weeks. The main branch is considered the most stable code and it should always be the same as the current production version of the application. The dev branch is where all the development happens. From the dev branch, the feature branch is branched out and individual developers can focus on their tasks without worrying about disturbing other fellows. When the release happens, a separate release branch is established from the dev branch and the release manager will ensure the pull request from the release branch to the main one should not introduce any regressions thanks to CI/CD pipeline. Figure 16 below illustrates the general branching strategy of the development team.

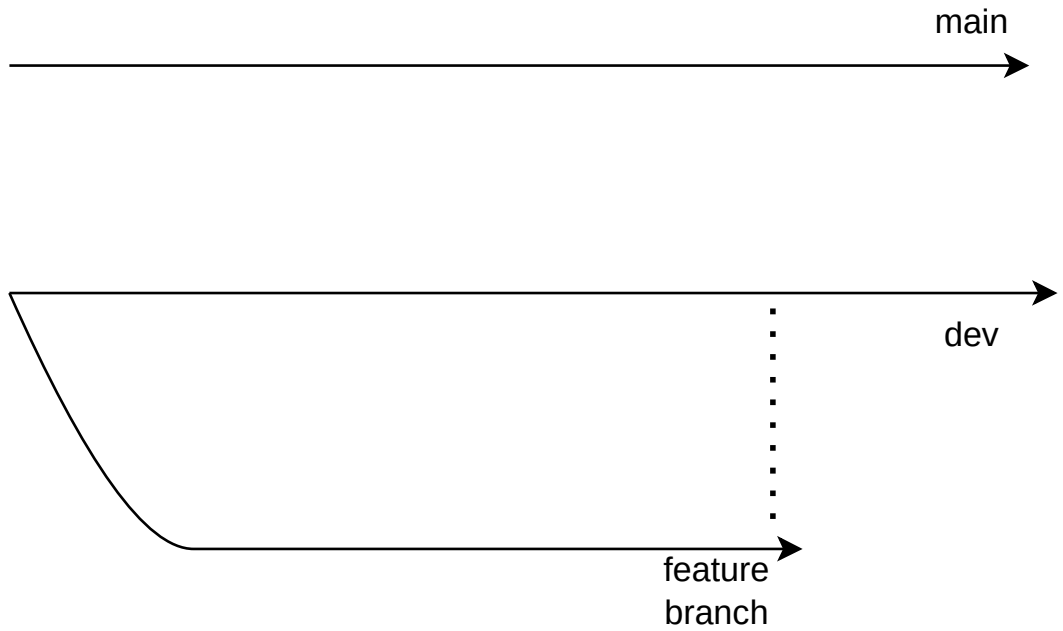
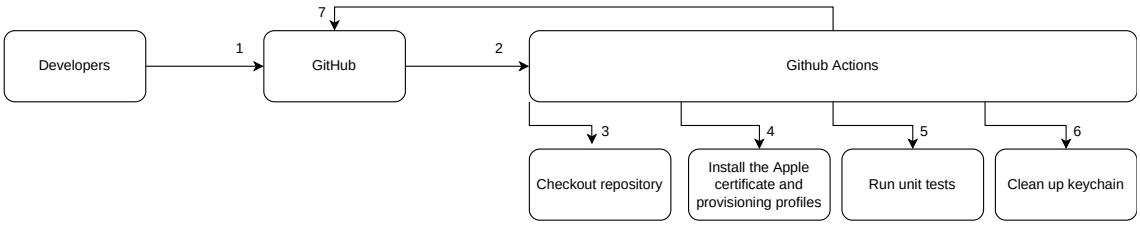


Figure 16. The application branch strategy

Continuous integration pipeline implementation

CI workflow

Figure 17 describes the main workflows of the continuous integration pipeline. The CI pipeline is a complex process, and it can be divided into multiple stages. Those stages will be further discussed in the following sections.



1. Developers open a pull request.
2. GitHub received a pull request open event and notify GitHub Actions.
3. GitHub Actions start checking the repository.
4. Install the necessary secrets in order to be able to run tests.
5. Run unit tests for the changes in the pull request.
6. Clean the secrets saved in the Keychain from step 4.
7. Notify GitHub about the status of the workflow.

Figure 17. CI workflow

The CI workflow starts when developers open a pull request against the dev branch. GitHub Actions is notified and triggered by the CI workflow defined in the .yml file. The first action from GitHub Actions is to check out the codebase of the branch. After that, the action of installing the Apple certificate and provisioning profiles will be executed. The definition of Apple certificate and provisioning profiles will be explained in detail in the later sections when setting up GitHub Actions. Next, the action of running unit tests is triggered as all the necessary secrets are installed in the build machine. To ensure the quality of the pull request, only unit tests were selected as they are lightweight and fast to run. UI tests were opted out as they are more time-consuming which results in a much longer wait time for developers. It is a pragmatic approach and receives general approval from the whole development team. After all the tests are executed, the last action is to clean up the secrets that are saved during the beginning phase of the CI workflow including the Apple certificate, provisioning profiles, passwords, etc. The status of workflow is then reported back to GitHub to let developers know if their changes break any part of the codebase and proceed to fix it if it is the case. The developers shall not merge the pull request if the workflow fails.

Fastlane configuration for CI pipeline

As discussed previously, Fastlane will be facilitated as a tool to automate the process of running unit and UI tests as well as signing and uploading artifacts to Apple's TestFlight service. To integrate Fastlane into an iOS project, several steps need to be followed. First, the local environment is required to have Fastlane installed which can simply be achieved by running the commands from the listing 3 in the project main directory:

```
brew install fastlane
fastlane init
```

Listing 3. Fastlane setup command

After running the commands, a fastlane folder will be created inside the project directory that contains all the necessary files for configuring Fastlane such as Appfile, Fastfile, etc. The content of the Fastfile where all lane declarations are revealed from listing 4.

```
update_fastlane
default_platform(:ios)

platform :ios do
  desc "Run unit tests"
  lane :unit_tests do
    scan(scheme: "TripPlannerTests")
  end

  desc "Run UI tests"
  lane :ui_tests do
    scan(scheme: "TripPlannerUITests")
  end
end
```

Listing 4. Content of the Fastfile

From listing 4, two lanes are declared. The first one is called unit_tests which accounts for triggering the unit tests process and the ui_tests lane is leveraged for UI tests. Each lane has its own description for explicit. At the beginning of the Fastfile, the update_fastlane command ensures that the Fastlane version is always up to date. Everything is now ready for running the lanes locally.

In the project main directory, try running the two commands:

```
fastlane unit_tests
fastlane ui_tests
```

Figure 18 indicates an example of a successful lane triggering for unit and UI tests.

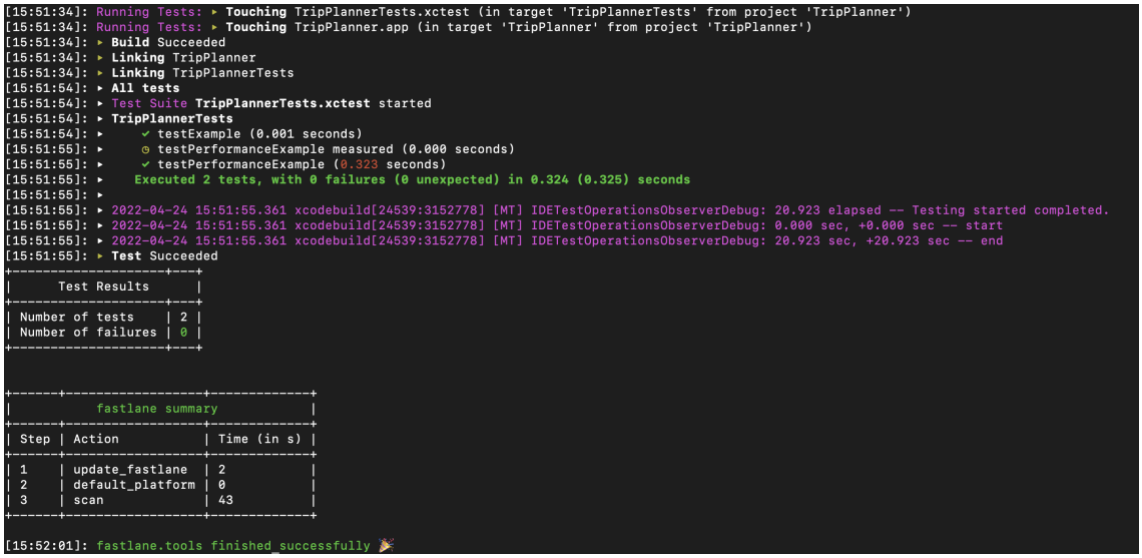


Figure 18. A successful lane results.

GitHub Actions configuration for CI pipeline

To configure GitHub Actions, two places require special attention. First, a local .github/workflows directory should be created to accommodate the .yml files. Listing 5 shows the content inside one of the .yml files that run during the CI process.

```
# Pull request workflows that are triggered when PR is made.

name: Pull request

# Controls the condition under which the workflow will be run
on:
  # Triggers the workflow on push or pull request events but only for the
develop branch
  pull_request:
    branches: [ develop ]

  # Allows you to run this workflow manually from the Actions tab
  workflow_dispatch:

# A workflow run is made up of one or more jobs that can run sequentially or
in parallel
jobs:
  build_with_signing:
    runs-on: macos-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v2
      - name: Install the Apple certificate and provisioning profile
        env:
          BUILD_CERTIFICATE_BASE64: ${{ secrets.BUILD_CERTIFICATE_BASE64 }}
          P12_PASSWORD: ${{ secrets.P12_PASSWORD }}
          BUILD_PROVISION_PROFILE_BASE64: ${{
secrets.BUILD_PROVISION_PROFILE_BASE64 }}
          KEYCHAIN_PASSWORD: ${{ secrets.KEYCHAIN_PASSWORD }}
        run: |
          # create variables
          CERTIFICATE_PATH=$RUNNER_TEMP/build_certificate.p12
          PP_PATH=$RUNNER_TEMP/build_pp.mobileprovision
          KEYCHAIN_PATH=$RUNNER_TEMP/app-signing.keychain-db

          # import certificate and provisioning profile from secrets
          echo -n "$BUILD_CERTIFICATE_BASE64" | base64 --decode --output
$CERTIFICATE_PATH
          echo -n "$BUILD_PROVISION_PROFILE_BASE64" | base64 --decode --output
$PP_PATH

          # create temporary keychain
          security create-keychain -p "$KEYCHAIN_PASSWORD" $KEYCHAIN_PATH
          security set-keychain-settings -lut 21600 $KEYCHAIN_PATH
          security unlock-keychain -p "$KEYCHAIN_PASSWORD" $KEYCHAIN_PATH

          # import certificate to keychain
          security import $CERTIFICATE_PATH -P "$P12_PASSWORD" -A -t cert -f
pkcs12 -k $KEYCHAIN_PATH
          security list-keychain -d user -s $KEYCHAIN_PATH

          # apply provisioning profile
          mkdir -p ~/Library/MobileDevice/Provisioning\ Profiles
          cp $PP_PATH ~/Library/MobileDevice/Provisioning\ Profiles
      - name: Run tests
        run: |
          bundle exec fastlane unit_tests
      - name: Keychain and provisioning profile clean up
        if: ${{ always() }}
        run: |
          security delete-keychain $RUNNER_TEMP/app-signing.keychain-db
          rm ~/Library/MobileDevice/Provisioning\
Profiles/build_pp.mobileprovision
```

Listing 5. Content of the .yml file that is triggered during the CI pipeline (GitHub Actions Authors. 2022)

Before the instruction .yml file can be executed. Numerous secrets need to be configured in GitHub Actions main console. Figure 19 shows obligatory variables.

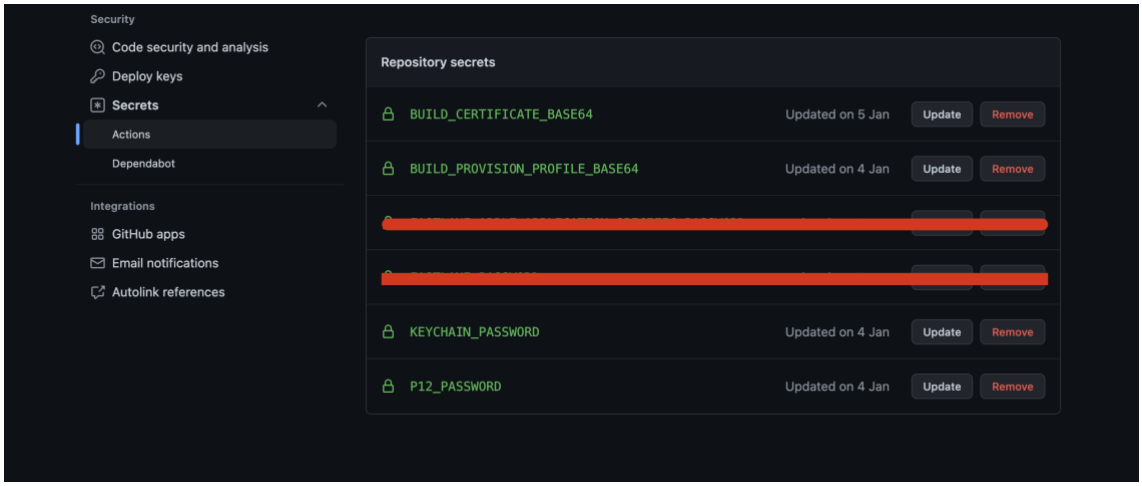


Figure 19. Compulsory secrets for CI actions

BUILD_CERTIFICATE_BASE64 is the based64-encoded string representation of the Apple certificate. The Apple certificate is used as a digital method for signing the application for both development and distribution purposes (Apple, 2022).

BUILD_PROVISION_PROFILE_BASE65 is similarly the based64-encoded string representation of the provisioning profile. The provisioning profile is leveraged to allow iOS applications installed on iOS devices or a Mac (Apple, 2022).

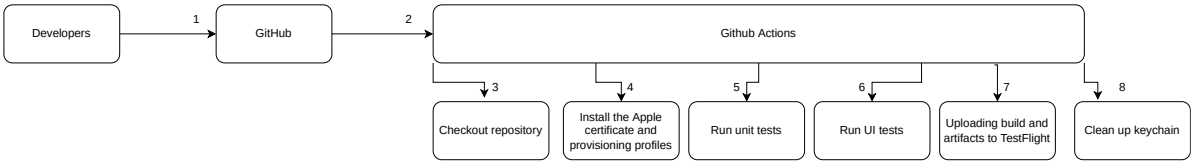
KEYCHAIN_PASSWORD is the password used for creating a temporary keychain that is responsible for storing all other secrets during the runtime of the CI pipeline.

P12_PASSWORD is the credential that is used to protect and ensure only ones that have the correct password can read the content of the BUILD_CERTIFICATE_BASE64

Continuous delivery pipeline implementation

CD workflow

Figure 20 illustrates the main steps involved in the continuous delivery pipeline. Like the CI pipeline, the CD flow includes several essential processes to assure a stable release for the end-users.



1. Developers merge/push commits to the main branch.
2. GitHub received the event and notify GitHub Actions.
3. GitHub Actions start checking the repository.
4. Install the necessary secrets in order to be able to run tests.
5. Run unit tests.
6. Run UI tests
7. Upload the build and artifacts to TestFlight
8. Clean the secrets saved in the Keychain from step 4.

Figure 20. Continuous delivery workflow

When developers push commits or a pull request is merged into the main branch, it will trigger the CD process in GitHub Actions. In the first stage, the status of the codebase is checked out in the main branch. After that, the installation of the necessary secrets into the build machine follows. Next, the whole test suite will be run including both unit and UI tests as it will guarantee the high quality of the application. After the quality of the application is checked, Fastlane will generate the build alongside the application artifacts and upload the whole package to Apple's TestFlight service. Lastly, all the saved secrets will be removed from the temporary keychain to guarantee no important piece of information is leaked inside public computers.

Fastlane configuration for CD pipeline

To support the CD process, the beta lane will be added to the Fastfile. Listing 6 indicates the content of the new beta lane.

```
desc "Push a new beta build to TestFlight"
lane :beta do
   increment_build_number(xcodeproj: "TripPlanner.xcodeproj")
   build_app(scheme: "TripPlanner")
   upload_to_testflight(
     skip_waiting_for_build_processing: true,
     apple_id: "1579411256"
   )
End
```

Listing 6. New lane added for CD pipeline

The newly created lane will account for numerous responsibilities. Initially, it increases the build number for the project which assures that we could never have a duplicate build number in TestFlight. Right after that, the process of archiving and building the application starts. When the build is ready, Fastlane will ease the process of uploading the build to TestFlight thanks to the upload_to_testflight command.

GitHub Actions configuration for CD pipeline

The configuration for the CD pipeline requires two additional changes from the current CI pipeline. Firstly, a new release.yml file is created and stored inside the .github/workflows folder specifically for the CD processes. Listing 7 shows the content of the file.

```
name: Release

# Controls when the workflow will run
on:
  # Triggers the workflow on push to main branch
  push:
    branches: [ main ]

  # Allows you to run this workflow manually from the Actions tab
  workflow_dispatch:

# A workflow run is made up of one or more jobs that can run sequentially or in
parallel
jobs:
  build_with_signing:
    runs-on: macos-latest
```

```
    steps:
      - name: Checkout repository
        uses: actions/checkout@v2
      - name: Install the Apple certificate and provisioning profile
        env:
          BUILD_CERTIFICATE_BASE64: ${{ secrets.BUILD_CERTIFICATE_BASE64 }}
          P12_PASSWORD: ${{ secrets.P12_PASSWORD }}
          BUILD_PROVISION_PROFILE_BASE64: ${{
secrets.BUILD_PROVISION_PROFILE_BASE64 }}
          KEYCHAIN_PASSWORD: ${{ secrets.KEYCHAIN_PASSWORD }}
        run: |
          # create variables
          CERTIFICATE_PATH=$RUNNER_TEMP/build_certificate.p12
          PP_PATH=$RUNNER_TEMP/build_pp.mobileprovision
          KEYCHAIN_PATH=$RUNNER_TEMP/app-signing.keychain-db

          # import certificate and provisioning profile from secrets
          echo -n "$BUILD_CERTIFICATE_BASE64" | base64 --decode --output
$CERTIFICATE_PATH
          echo -n "$BUILD_PROVISION_PROFILE_BASE64" | base64 --decode --output
$PP_PATH

          # create temporary keychain
          security create-keychain -p "$KEYCHAIN_PASSWORD" $KEYCHAIN_PATH
          security set-keychain-settings -lut 21600 $KEYCHAIN_PATH
          security unlock-keychain -p "$KEYCHAIN_PASSWORD" $KEYCHAIN_PATH

          # import certificate to keychain
          security import $CERTIFICATE_PATH -P "$P12_PASSWORD" -A -t cert -f
pkcs12 -k $KEYCHAIN_PATH
          security list-keychain -d user -s $KEYCHAIN_PATH

          # apply provisioning profile
          mkdir -p ~/Library/MobileDevice/Provisioning\ Profiles
          cp $PP_PATH ~/Library/MobileDevice/Provisioning\ Profiles
      - name: Run tests
        run: |
          bundle exec fastlane unit_tests
          bundle exec fastlane ui_tests
      - name: Upload to Testflight
        env:
          FASTLANE_APPLE_APPLICATION_SPECIFIC_PASSWORD: ${{
secrets.FASTLANE_APPLE_APPLICATION_SPECIFIC_PASSWORD }}
          FASTLANE_PASSWORD: ${{ secrets.FASTLANE_PASSWORD }}
        run: |
          bundle exec fastlane beta
      - name: Clean up keychain and provisioning profile
        if: ${{ always() }}
        run: |
          security delete-keychain $RUNNER_TEMP/app-signing.keychain-db
          rm ~/Library/MobileDevice/Provisioning\
Profiles/build_pp.mobileprovision
```

Listing 7. Content of the release.yml file that is triggered during the CD pipeline (GitHub Actions Authors. 2022).

Similarly, the release.yml workflow includes multiple stages required for the release. In the beginning, a checkout action is triggered to pull the latest status

of the main branch. The next phase is to install the required secrets as discussed in a previous section such as BUILD_CERTIFICATE_BASE64, BUILD_PROVISION_PROFILE_BASE64, P12_ PASSWORD, etc.

After configuring all necessary confidential tokens, the whole test suite will be activated which composes of unit and UI tests. Once tests are successfully passed. The action “Upload to TestFlight” will be triggered. The main workload in this phase is to execute the Fastlane beta lane. To complete the action, two new environment variables are added. Figure 21 illustrates the required secrets for completing the Fastlane beta lane run.

FASTLANE_APPLE_APPLICATION_SPECIFIC_PASSWORD is a secret that enables developers to safely access information from the Apple developer account if the access is from third-party apps. (Apple, 2022)

FASTLANE_PASSWORD is the password used for the current user in the App Store Connect. To configure the Apple user for Fastlane, an Appfile in the Fastlane folder is compulsory, and it will be explained in the next section.

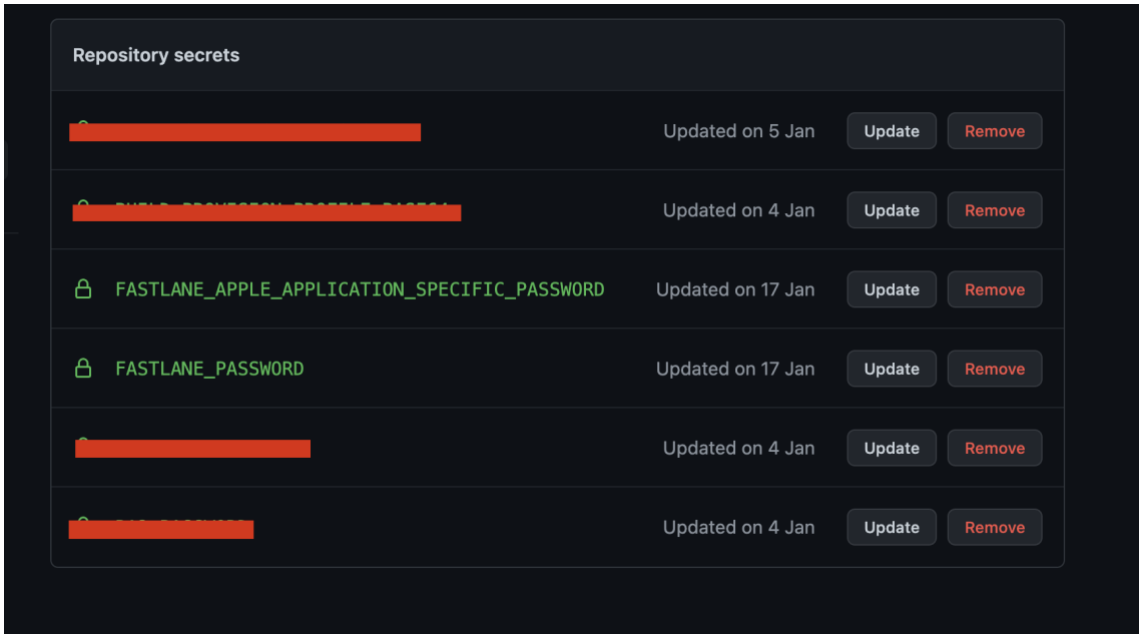


Figure 21. The newly added secrets for the CD process

As discussed, it is obligatory to have an Appfile configured so Fastlane can use the correct Apple user when running the lane. Listing 8 shows the content of the Appfile used for the project.

```
app_identifier("com.longnguyen.TripPlanningApp") # The bundle identifier of
your app
apple_id("Long.Nguyen@metropolia.fi") # Your Apple email address

itc_team_id("1710660") # App Store Connect Team ID
team_id("EVMSWPGYSG") # Developer Portal Team ID
```

Listing 8. Content of the Appfile

3.5 Results

With the detailed planning as well as expertise of the team members, the implementation of the CI/CD pipeline is inevitably a success. The power of GitHub Actions was leveraged and combined with Fastlane to construct a swift CI and CD pipeline. The case study sets a solid foundation for the upcoming development of the iOS application in the future. It solves the problem of unorganized software architecture and assists the developers to run the tests more often which results in more confidence in the product.

Figure 22 presents the main screen of the application for the first phase of the development cycle. As stated earlier, UI is not prioritized during the current stage.

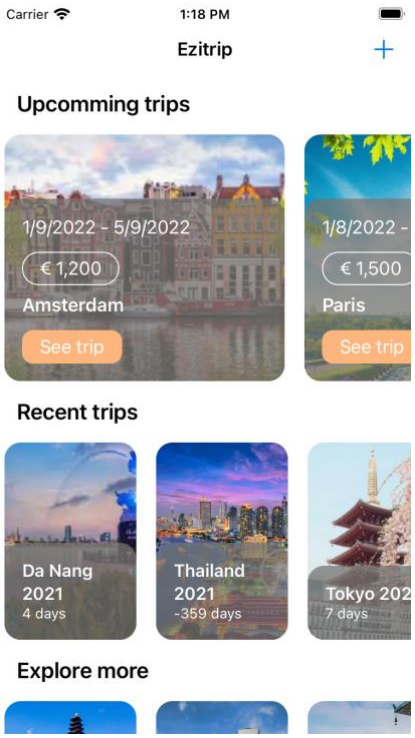


Figure 22. The home screen of the application

Figure 23 illustrates the successful workflow that is triggered for opening pull requests against the dev branch. It is additionally configured that each pull request requires at least one approval before merging which enhances the code quality significantly as well as prevents engineers from the accidental merge.

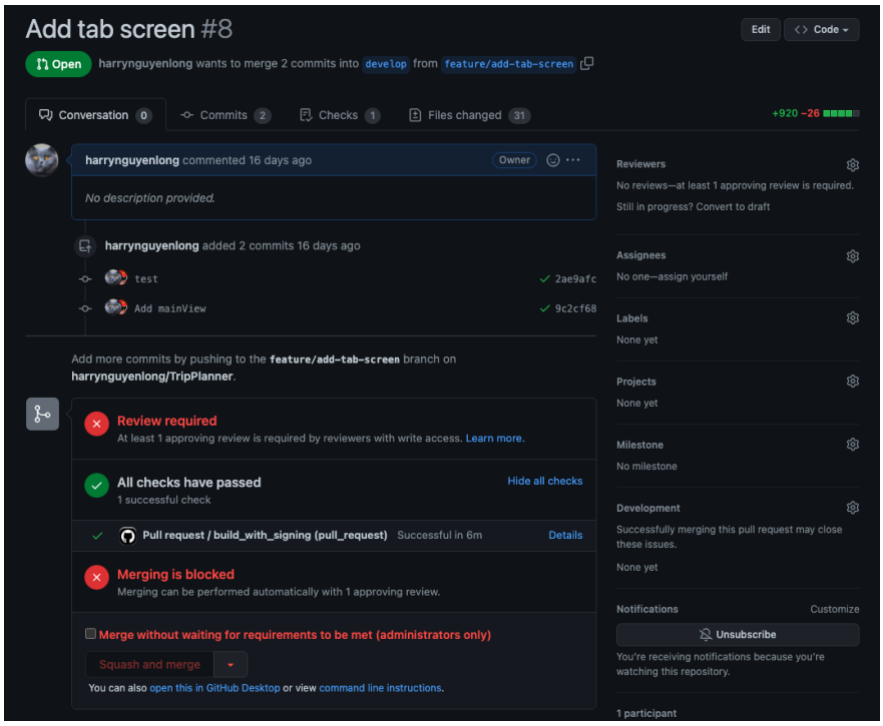


Figure 23. Checks run for one open pull request

Figures 24 and 25 outline steps involved during a CD flow as well as the result as the generated artifacts are uploaded automatically to TestFlight.

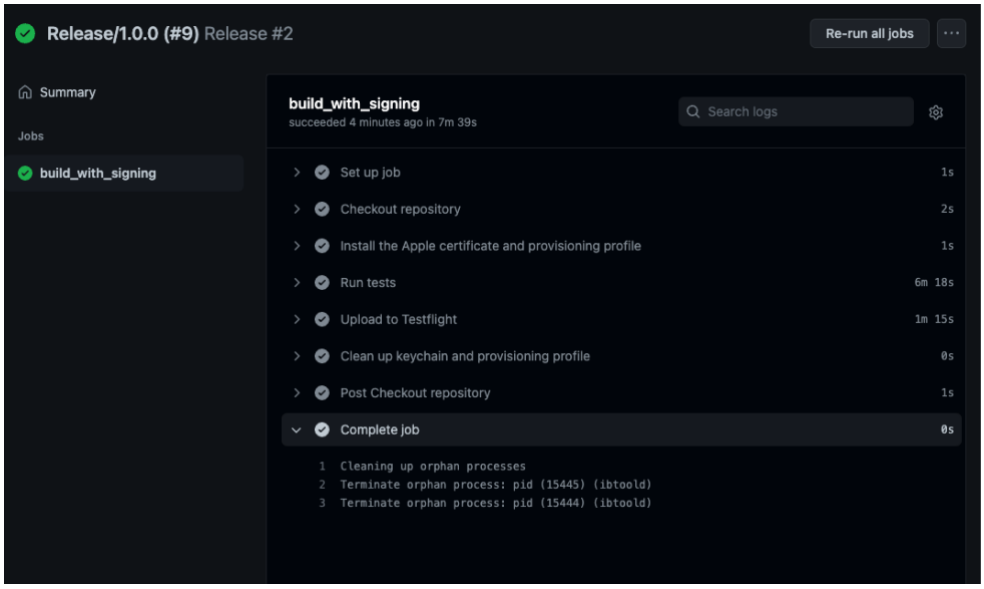


Figure 23. Actions run successfully for the CD pipeline

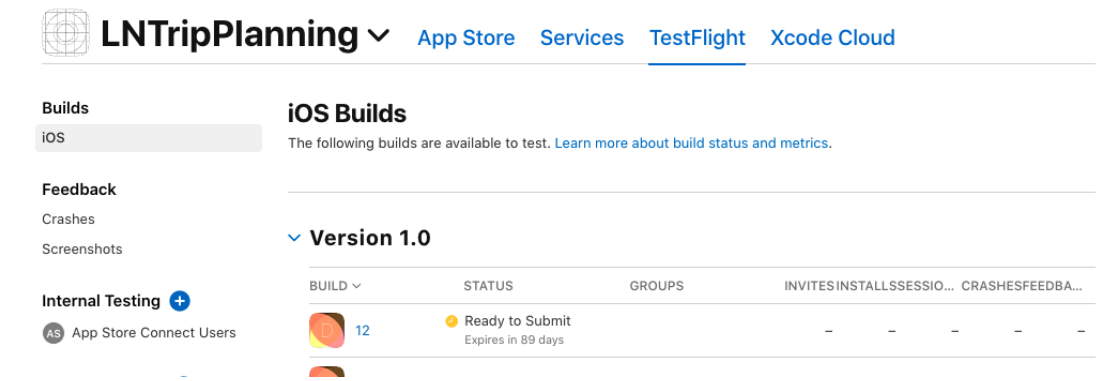


Figure 24. A successful build is uploaded to TestFlight

In general, the team manages to finish the first phase of the project with a success. By implementing the fully working CI/CD pipeline, the huge amount of precious time of the developers will be saved by not having to manually release a new version every time or test suites are automatically run every new code change is applied. It is also notable that the software architecture of the application is decided during the first phase which improves the testability of the application as well as enhances the scalability if the application grows in the future.

3.6 Future development

Despite the successful implementation during the initial stages of the project which guarantee the solid ground for further development, various aspects can be considered for the future of the application.

For the current pipeline, there is no good separation between different development environments. In practice, a more complex pipeline for alpha, staging, and production is preferable as it provides clear boundaries and avoids developers unintentionally mistaking actual user data with test data. Now, all builds are distributed via TestFlight which creates unnecessary constraints in case the developers would desire to share the staging/alpha build with internal people in the company or with third parties. Numerous services can help with distributing the alpha and staging version of an application that the team can

investigate further for future development such as Firebase App Distribution, HocketApp, TestFairy, etc. Finally, even though the architecture is designed with testability in mind, the number of tests is not at an adequate level. As the application grows, it is imperative that more tests would need to be integrated and run more frequently to maximize the benefits of the architecture choices as well as the time invested in implementing the CI/CD pipeline.

4 Conclusion

The project successfully delivered a basic understanding of how modern applications are built and provides an overview of the evolution of supporting tools for software packaging. CI/CD technologies are game-changing tools that improve a product's quality and save a considerable amount of developer hours. In the report, the idea behind terms such as CI, CD, and introduction to popular CI/CD tools are demystified. Additionally, the significance of proper software architecture is addressed and multiple different approaches are examined and compared.

The first phase of the project delivers a production-grade version of how modern CI/CD should be implemented. By leveraging GitHub Actions as a CI/CD service and Fastlane as an assisting tool, every change to the code base will be examined to ensure it matches the current standard. Furthermore, the pipeline automates the delivery process which is usually time-consuming, and error-prone as the local environment might differ. Countless hours will be saved which helps the organization to allocate more resources to compete with other players. In the competitive market, rather than wasting time on running tests manually and deploying the application, it is wiser to run all mundane work automatically and free the developers' resources to allow them to focus on the development of the new features.

Despite impediments such as high-cost expenditure and being tricky to implement properly, the enormous benefits that a CI/CD pipeline creates clearly outweigh the cons as the benefits will shine more once the application grows.

References

Andrew, G & Steve, M & Paul, M (2007) Continuous integration: improving software quality and reducing risk [Online] Available at: https://learning.oreilly.com/library/view/continuous-integration-improving/9780321336385/ (Accessed: 3 October 2020)

Apple Authors (2020). Release a version update in phases [Online] Available at: https://help.apple.com/app-store-connect/#/dev3d65fcee1 (Accessed: 10 October 2020)

Bernie, G & Elfriede, D & Thom, G (2009). Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality. [Online] Available at: https://learning.oreilly.com/library/view/implementing-automated-software/9780321619600/ (Accessed: 29 September 2020)

Brent, L (2020) Continuous Integration vs. Continuous Delivery vs. Continuous Deployment, 2nd Edition [Online] Available at: https://learning.oreilly.com/library/view/continuous-integration-vs/9781492088943/ (Accessed: 03 October 2020)

Doron, K (2018). Continuous Delivery for Mobile with Fastlane. [Online] Available at: https://learning.oreilly.com/library/view/continuous-delivery-for/9781788398510/ (Accessed: 10 October 2020)

Eberhard, W (2017) A Practical Guide to Continuous Delivery [Online] Available at: https://learning.oreilly.com/library/view/a-practical-guide/9780134691626/ (Accessed: 08 October 2020)

Ed, C (2015). Practical Software Development Techniques: Tools and Techniques for Building Enterprise Software. [Online] Available at: https://learning.oreilly.com/library/view/practical-software-development/9781484206201/ (Accessed: 30 October 2020)

Herbsleb, J.D. & Moitra, D. (2001). Global Software Development. IEEE Software 18(2): pp. 16-20. [Online] Available at: https://ieeexplore.ieee.org/abstract/document/914732 (Accessed: 26 September 2020)

Joseph, I (2018). Software Architect's Handbook. [Online] Available at: https://learning.oreilly.com/library/view/software-architects-handbook/9781788624060/ (Accessed: 30 October 2020)

Mathias, M (2014) Continuous Integration and Its Tools. IEEE Software 31(3): pp. 14-16 [Online] Available at: https://ieeexplore-ieee-org.ezproxy.metropolia.fi/document/6802994/ (Accessed: 3 October 2020)

Microfocus Plc (2020) Continuous Integration Workflow [Online] Available at: https://www.microfocus.com/documentation/visual-cobol/vc50pu5/EclWin/GUID-70375FE3-745F-4FA5-B28A-6F65953E562B.html (Accessed: 3 October 2020)

Pete, H (2020) Continuous Delivery in the Wild [Online] Available at: https://learning.oreilly.com/library/view/continuous-delivery-in/9781492077701/ (Accessed: 08 October 2020)

Srinivasan, D & Gopalaswamy, R. (2007). Software Testing: Principles and Practices. [Online] Available at: https://learning.oreilly.com/library/view/software-testing-principles/9788177581218/ (Accessed: 29 September 2020)

GitHub Actions Authors. (2022). GitHub Actions Documentation. [Online] Available at: https://docs.github.com/en/actions

William, E. Lewis (2017). Software Testing and Continuous Quality Improvement, 3rd Edition. [Online] Available at: https://learning.oreilly.com/library/view/software-testing-and/9781351722209/ (Accessed: 29 September 2020)

Mark, Richards & Neal Ford. (2020). Fundamentals of software architecture. [Online] Available at: https://learning.oreilly.com/library/view/fundamentals-of-software/9781492043447/ (Accessed: 24 Jan 2021)

Apple (2021) Model – view – controller. [Online] Available at: https://developer.apple.com/library/archive/documentation/Gensseral/Conceptual/DevPedia-CocoaCore/MVC.html#//apple_ref/doc/uid/TP40008195-CH32-SW1 (Accessed: 25 Jan 2021)

John Kouraklis (2016) MVVM in Delphi: architecting and building Model View ViewModel applications. [Online] Available at: https://learning.oreilly.com/library/view/mvvm-in-delphi/9781484222140/ (Accessed at 25 Jan 2021)

Chris Barker (2020) Learn SwiftUI [Online] Available at: https://learning.oreilly.com/library/view/learn-swiftui/9781839215421/ (Accessed at: 1 February 2021)

Robert C. Martin (2017) Clean Architecture: A Craftsman's Guide to Software Structure and Design [Online] Available at: https://learning.oreilly.com/library/view/clean-architecture-a/9780134494272/ (Accessed at: 17 April 2022)

Apple (2022) AppStore Connect API [Online] Available at: https://developer.apple.com/documentation/appstoreconnectapi/ (Accessed at: 24 April 2022)

Apple (2022) Apple app-specific passwords [Online] Available at: https://support.apple.com/en-us/HT204397 (Accessed at: 24 April 2022)