UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Markus Aksli

Barriers And Solutions In CI/CD Implementation for Unity Game Development

Bachelor's Thesis (9 ECTS)

Supervisor(s): Chinmaya Dehury Ph.D.
Ulrich Nobisrath Ph.D.
Martin Jeret

Tartu 2022

Barriers And Solutions In CI/CD Implementation for Unity Game Development

Abstract:

DevOps is a popular software development approach that promises efficiency and speedy development cycles. Implementing DevOps automation practices in a CI/CD plays a critical role in providing the promised improvements by automating the building, testing, and delivery software. One niche of software development that seems to be lagging in the adoption of CI/CD is game development. Video game projects seem to impose a unique set of technical implementation challenges for creating an effective CI/CD pipeline in practice, which is compounded by a lack of information. This thesis aims to provide insight into the technical challenges and currently used solutions in implementing a CI/CD pipeline for game development with the Unity game engine. This was studied by conducting semi-structured interviews with companies that used Unity to develop their products. The results describe some of the significant technical challenges and practical solutions in using version control software and automated building, testing, and delivery for game development with Unity.

Keywords:

DevOps, Agile Software Development, Continuous Practices, CI/CD, Game Development

CERCS: P170 Computer science, numerical analysis, systems, control

Probleemid ja lahendused CI/CD rakendamisel mänguarenduseks Unity mängumootoriga

Lühikokkuvõte:

DevOps on populaarne tarkvaraarenduse lähenemine, mis lubab kiiremaid ja efektiivsemaid arendustsükleid. DevOpsi praktilisel implementeerimisel omab eririlist tähtsust CI/CD, mis tagab lubatud kasu, automatiseerides tarkvara kompileerimise, testimise ja toimetamise. Üks tarkvara arenduse valdkond, kus CI/CD on vähem levinud, on mänguarendus. Mänguarenduses esinevad omalaadsed tehnilised raskused CI/CD rakendamisel, millele ei aita kaasa vähene teadmiste levik. Töö eesmärk on laiendada arusaamist nendest raskustest ning loetleda praktilisi lahendusi CI/CD rakendamisel mänguarenduseks Unity mängumootoriga. Selle nimel viidi läbi poolstruktureeritud intervjuud firmadega, mis kasutasid Unity mängumootorit oma toodete arendamiseks. Tulemusena kirjeldatakse olulisi tehnilisi raskusi ja mõningaid praktilisi lahendusi versioonihaldus tarkvara ning automaatse kompileerimise, testimise ja toimetamise rakendamisel mänguarenduseks Unity mängumootoriga.

Võtmesõnad:

DevOps, Agiilne tarkvaraarendus, Pidevad arenduspraktikad, CI/CD, Mänguarendus

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1 Introduction

DevOps is a popular software development approach that promises faster and more efficient development cycles. One of the most important practices driving these improvements is continuous integration and continuous delivery/deployment (CI/CD). At the heart of CI/CD is the automation of building, testing, and delivery of software. Many services, platforms, and tools like Jenkins[1], Buildbot[2], fastlane[3], Travis CI[4], CircleCI[5], GitLab[6], GitHub Actions[7], and Azure Pipelines[8] have been built to allow teams to adopt these practices quickly. There is a lot of widely available information about how to use these tools for mobile or web development, but one software niche that seems to have unique difficulty using this infrastructure is game development.

Game developers have to deal with large amounts of binary files resulting in large repositories, work with artists and sound designers, deliver game builds that can be tens of gigabytes, and overcome other unique challenges. These challenges also make it harder to implement DevOps practices like CI/CD, which is compounded by a lack of information about the tools and solutions used by game developers in practice. Still, the benefits of automated building, testing, and delivery are just as relevant as in the rest of IT [12].

The aim of this thesis was to gather information about the problems game developers face in implementing DevOps automation practices and to find out which solutions they use to solve said problems. This task is made more complicated by the differences in the game engines used by game developers. Game engines provide features like rendering 3D models and playing sounds to simplify game development. Each game engine has unique circumstances when it comes to using version control and CI/CD tools. This thesis only focuses on the Unity[9] game engine to reduce the scope of the research. Unity was chosen due to its popularity and the author's familiarity with the engine. The research question was explored by conducting semi-structured interviews with companies that used the Unity game engine to develop their products. The interview responses were recorded and later analysed to describe the useful findings.

Chapter 2 of the thesis describes the relevant concepts, the unique aspects of game development, the critical areas of inquiry, and the motivation for the research. The contributor selection, interview guide formulation, and interview process are described in Chapter 3. The interview results are described in Chapter 4 as they relate to using version control software and automating each of the steps in a CI/CD pipeline. The Appendix includes all of the interview questions.

[1] https://www.jenkins.io/
[2] https://www.buildbot.net/
[3] https://fastlane.tools/
[4] https://www.travis-ci.com/
[5] https://circleci.com/
[6] https://about.gitlab.com/
[7] https://github.com/features/actions
[8] https://azure.microsoft.com/en-us/services/devops/pipelines/
[9] https://unity.com/

2 Background

This chapter gives a brief overview of DevOps and the continuous software engineering paradigm (continuous integration, delivery, and deployment). The chapter defines each of the relevant concepts in the first two subsections and then describes the unique aspects of game development as they relate to adopting DevOps and continuous practices. This includes an explanation of game engines and is followed by an overview of the Unity game engine and its development ecosystem. Finally, the issue of video games with notable negative reception is described as a motivation for the thesis.

2.1 DevOps

DevOps is a popular software development approach focused on the automation of repetitive tasks and collaboration between the different teams involved in developing and maintaining software. The word itself is a portmanteau of the words *development* and *operations*. Despite DevOps being a relatively well-known and discussed development approach, Erich et al. [2] claim that it is hard to find a generally agreed-upon academic definition of DevOps and its principles. Leite et al. [5] mention the same issue and propose a consolidated definition based on the most cited definitions: “DevOps is a collaborative and multidisciplinary effort within an organization to automate continuous delivery of new software versions while guaranteeing their correctness and reliability.” This definition that focuses on automation will also be adopted throughout this thesis.
Puppet [3] does not regard automation as the only important aspect of DevOps, as most firms with highly evolved DevOps practices have automated most of their repetitive tasks but lack the organizational structure to reflect the collaborative aspect of DevOps. Microsoft Azure [7], as one of the largest DevOps service providers, also regards it as a development approach centred around encouraging cooperation between developers and system administrators. Azure says that despite these two departments making up the word DevOps, there are many more teams involved in software development and the broader goal of DevOps is to encourage collaboration between all of them. However, this aspect of collaboration can be mostly agnostic to the type of software being developed. This thesis is primarily concerned with the automation practices of DevOps, as the unique technical challenges of game development are the most relevant in this effort.

2.2 Continuous integration and delivery/deployment

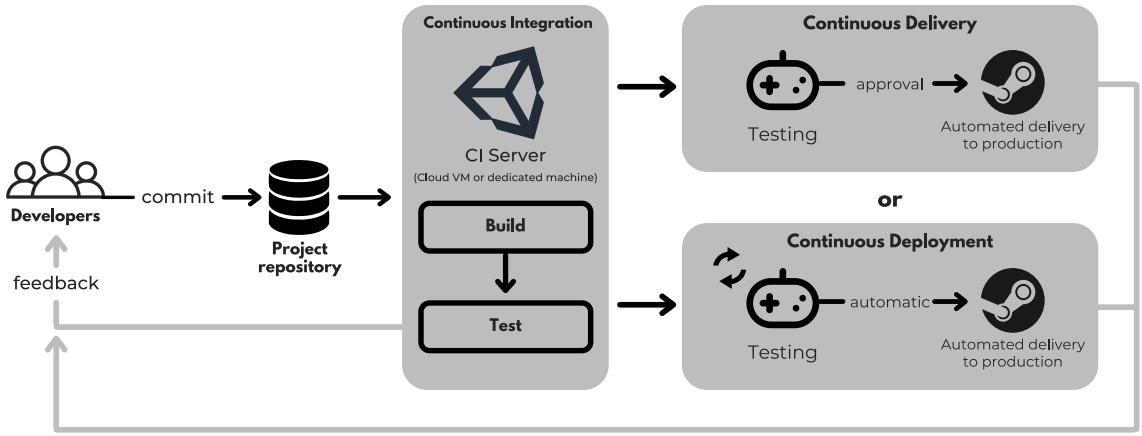

Figure 1. The relationship between continuous integration, delivery and deployment in game development [8].

While automating any repetitive tasks is seen as an essential component of DevOps practices, this also includes automating parts of core development tasks, namely building, testing, and delivery. The automation of these tasks as code changes are incorporated into a shared repository constitutes the area of continuous software engineering. This area includes the practices of continuous integration, continuous delivery, and continuous deployment [8].

Continuous integration (CI) is the practice of running automated checks and builds as code updates are merged in a shared repository [9]. These checks can include unit testing, integration testing, performance testing, as well as any other quality checks the developers deem necessary. Soares et al. [7] claim that in addition to the benefits of testing for safer and faster integration, CI promises to allow more teams to see these code changes, thereby improving communication among teams. They state that CI ensures that code produced by separate teams can be integrated into the product as smoothly as possible.

CI only refers to building and testing for the purpose of ensuring the quality and reliability of the software. Shahin et al. [8] describe continuous delivery (CDE) and continuous deployment (CD) as development approaches that involve the CI practice of automated building and testing but add the practice of automated delivery to production or customer environments. They state that the difference between the two is the presence of a manual approval step in CDE before automated delivery can commence. This means that CD necessitates automatic delivery of the build result from the CI server to production when code changes are merged in the repository and CI checks do not fail. In CDE, this build result can still be delivered automatically but only when the developers approve it.

The implementation of CI practices with the delivery automation of either CDE or CD is referred to as a continuous integration and delivery/deployment (CI/CD) pipeline. A CI/CD pipeline can include or omit some parts of the mentioned practices. CI builds and checks can be run automatically in a CI/CD pipeline on every pull request before merging, after merging, or on every commit to the repository. As mentioned, a CI/CD pipeline can implement either CDE or CD, and automatic delivery can be similarly triggered by a variety of actions in the repository. Arachchi et al. [10] state that CI/CD tools can include repository and version control tools, build tools, automation tools, test automation tools, and

monitoring tools. Figure 1 illustrates the relationship between CI, CDE, and CD as they could be implemented in a CI/CD pipeline for game development with Unity.

2.3 Game development

Developing video games poses a few unique challenges when compared to other software. Video games resemble other entertainment mediums such as movies or books in their disposability. They can be played multiple times if designed with that purpose in mind, but they are ultimately interactive experiences meant to entertain and as such are fundamentally different from other software used for practical purposes. Game development involves far more creative team members for the creation of assets such as music, 3D models, and 2D art. These assets are used to build experiences that can take the player up to hundreds of hours to complete. This means video game builds can vary in size from tens of megabytes to hundreds of gigabytes and can take up to hundreds of hours for players to experience fully.

The scope of video games also means longer development cycles, emphasizing the initial release. New features can rarely be introduced without needing time for the creative teams to create more assets. These features also need extensive testing due to all the possible ways players can interact with them in the game environment, all the different players the game is intended for, and all of the interactions with the other game features. These complexities have led game developers to rely heavily on QA testing [11].

The time investment for developing new features means they are often extensively designed and prototyped to avoid wasting effort before committing to development. Other types of software can be iteratively improved based on customer feedback during continued use. Games, however, often rely on an experience that is significantly less compelling when repeated and require more functionality at release to attract customers.

Video games also introduce various hardware performance requirements for developers and end-users. Game builds can take up to hundreds of gigabytes of room on the customer's device. Compiling and packaging software can reduce its size by orders of magnitude compared to the source repository. This should mean that game source projects are similarly much larger than game builds, although this is part of the research question. Games can also require significant graphical performance to display their environments during development, while running integration tests, during playtesting, and while playing the final product. These hardware requirements are compounded by the various devices, and operating systems games are built for. These can include personal computers, game consoles, and mobile devices.

Game engines

Starting from just a programming language and building up to a functioning game that can be built for all of the mentioned different platforms requires the developer to implement many features. These usually include displaying graphics, playing sounds, reading and responding to player input, interpreting 3D asset file formats, being able to manipulate and serialize the game environment, and running real-time physics calculations. The algorithms and optimizations used to implement these features have a storied past in the advancement of game development.

These functions have been bundled together into development tools called game engines to simplify the game development process. Game engines usually include some subset of the functionality mentioned above, a user interface, a framework or scripting language to use the engine features, and tooling to build and package the game project for the previously

mentioned target platforms. Every game engine is different, and they can provide much more functionality than what was listed in this chapter. The most popular game engines based on the number of released games on the popular digital game distribution platform itch.io [1] are Unity, Construct, GameMaker: Studio, Godot, Twine, and Unreal Engine. Itch.io is a web-based platform that has no fees for distributing free games, and as such, the popularity of the game engines is not representative of distribution platforms like Steam and Play Store, which do not publish this statistic for their platforms.

Each of the mentioned game engines has its own approach to providing the mentioned functionality, and they excel in different areas. They are also built using vastly different technologies and have different supported target platforms, build processes, licensing technologies, and surrounding ecosystems of tools, assets markets, and plugins. This places different technical requirements for interfacing with common CI/CD tools. This thesis focuses on the CI/CD solutions used with the Unity game engine because of its popularity and the author's familiarity with the engine.

2.4 Unity game engine

The Unity game engine was created by colleagues David Helgason, Joachim Ante, and Nicholas Francis in Denmark and launched on June 6, 2005 [13]. At the time of writing, it is the most popular game engine based on the number of published projects on itch.io [1]. The Unity game engine can be used to make 2D, 3D, AR, and VR games for Windows, Linux, Mac, iOS, Android, WebGL, PS4, PS5, Xbox One, Xbox Series S, and more [14]. Therefore, Unity is referred to as a cross-platform game engine. Unity has been used to make successful games such as Escape from Tarkov, Monument Valley 2, Hollow Knight, and Cuphead [15].
Unity provides the previously mentioned game engine functionality, a GUI application called the Unity Editor, and supports scripting and programming in the C# programming language. The Unity project source can be compiled into an executable with compressed assets and an included runtime called the Unity Player. This can be done using either just-in-time (JIT) compilation with the Mono[10] backend or ahead-of-time (AOT) compilation using the Intermediate Language To C++[11] (ILCPP) backend. These backends make it possible to compile the C# source code to different executables that can run on all supported platforms. Building a Unity project can only be done through the Unity Editor, which requires a valid license to use.

Unity License

Unity offers various tiers of licenses to use the Unity Editor. The Personal and Student licenses are free. The Personal license can only be used if the developer has received revenue or funding less than $100K in the last 12 months. If the developer isn't a student or doesn't meet the criteria for the Personal license, they will have to purchase one of the paid team licenses. The paid licenses also include access to parts of the Unity ecosystem and unlock editor functionality such as removing the requirement to show the Unity logo when the game is launched. The Unity Editor requires either an account with an active Unity Editor license to be logged in or an activation file that has information about a valid license. The Unity Editor cannot be used to develop or build a project without validation from an active license. All of the mentioned license tiers are restricted to being activated on 2 machines at once.

[10] https://www.mono-project.com/
[11] https://docs.unity3d.com/Manual/IL2CPP.html

Unity has a separate floating license option for easier build automation which is called Unity Build Server[12]. This licensing solution can be deployed in a server and used to activate Unity Editor instances without GUI functionality to build a project [16]. Unity projects can be built without activation in the cloud using the Unity Cloud Build service, where additional concurrent build capability can be purchased at a much cheaper rate compared to buying more editor licenses or a floating license.

Unity Cloud Build and Plastic SCM

As mentioned, the restrictions of the licensing system can be averted by just automating builds and testing using Unity Cloud Build, which is a DevOps service platform created by Unity Technologies. This is part of a larger ecosystem of services and tools provided which includes the Unity Asset Store, Unity Teams, Unity Gaming Services, Unity Industrial Applications, machine learning agents, and Unity learning resources [16]. Unity also provides an integrated version control functionality with Unity Plastic SCM[13]. Unity Cloud Build and Unity Plastic SCM can already provide most of the functionality of a CI/CD pipeline and are tightly integrated with the Unity Editor. This means developers can set up version control and automated building and testing with minimal effort through the Unity Editor and the Unity Dashboard on their website [11].

2.5 Research objectives and motivation

The purpose of the research was to identify the significant barriers to implementing a CI/CD pipeline for game development. This was further narrowed by focusing on game development with the Unity game engine to reduce the scope of the required research. While Chapter 2 has already described some of the unique challenges in game development, there is little information on how relevant they are for the research question. There is also very little information, academic or otherwise, about how developers are solving the relevant challenges and with which tools. Therefore, the key areas of inquiry in this thesis were:

- the most relevant technical challenges and solutions in implementing a CI/CD pipeline for game development with Unity;
- the tools and services used in practice to automate building, testing, and delivery;
- developer sentiment towards Unity Cloud Build compared to more common CI/CD services and tools like GitHub Actions and Jenkins;
- developer sentiment towards Unity Plastic SCM compared to more widely used version control software like Git[14];
- developer sentiment towards automated testing compared to QA testing;
- the tools and services used to automate delivery to game distribution platforms.

Gaining insight into the developer sentiment towards Unity Cloud Build and Unity Plastic SCM was one of the most important areas of inquiry due to the ease of integration. Silva et al. [24] propose Unity Cloud Build as the tool for automating building and testing in their DevOps methodology, and Sakharov [25] implements it successfully. However, Thobari et al. [12] instead implemented automated builds using Docker hosts after searching forums on the internet, although they do not comment on why they did not use Unity Cloud Build.

Jussila [11] states that in practice, builds in Unity Cloud Build are very slow for mobile game development. Jussila also describes how additional tools such as Bitbucket Pipelines,

[12] https://unity.com/products/unity-build-server
[13] https://unity.com/products/plastic-scm
[14] https://git-scm.com/

Webhook Relay, and fastlane were essential for bridging the gaps in the functionality of Unity Cloud Build, although there was little readily available information on how to integrate these tools.

These few existing academic works do not give a good overview of the practicality of Unity Cloud Build, especially for developing larger games. None of the mentioned works discuss the use of Unity Plastic SCM. This lack of academic insight is reflected in the lack of information about the subject matter on the internet. These factors were the main motivation for the research.

Video games with notable negative reception

The research was also motivated by the decades-long continuing trend of high-profile negatively received video game releases. Video games can receive negative reception for a variety of reasons such as design issues, high pricing, or controversial content. However, there were a number of games with large development budgets that were reviewed poorly in significant part due to technical issues such as performance problems or bugs at release. These games included Fallout: New Vegas (2010), Elder Scrolls: Skyrim (2011), Diablo 3 (2012), SimCity (2013), Battlefield 4 (2013), Assassin's Creed Unity (2014), Halo: The Master Chief Collection (2014), No Man's Sky (2016), Mass Effect: Andromeda (2017), Fallout 76 (2018), WWE 2K20 (2019), Cyberpunk 2077 (2020), Battlefield 2042 (2021), and more [17][18][19][20][21][22][23]. It is unclear whether these games were developed with or without DevOps principles and CI/CD pipelines. However, their failures demonstrate a lack of reliability in game releases that the described DevOps practices aim to remedy. By identifying the key technical challenges along with the current solutions, the author hopes to promote further research and more adoption of CI/CD in game development to mitigate this issue.

3 Methodology

The subject matter was researched by conducting semi-structured interviews with companies that used the Unity game engine to develop their products. This method was chosen due to the exploratory nature of the research question.

3.1 Interview questions

Appendix I includes an interview guide with 31 questions, which was formulated to provide structure to the interviews. Some questions on the guide included additional follow-up questions. The interview guide was followed for all contributors, but questions were skipped when not relevant and further probing questions were asked where useful.

Questions were formulated through discussions between the author and the supervisors to establish how each of the contributors approached version control and automating building, testing, and delivery. Further questions were included to establish general background details such as team size, project size, and development process specifics. These were included to provide context for the approaches chosen by each participant. Similar questions were included to inquire about build frequency, testing frequency, and delivery frequency.

3.2 Participant selection

Interview participants were chosen through voluntary response sampling. This was done by going through lists of popular games made with the Unity game engine and finding the contact information of the companies that worked on them. These companies were contacted through email and asked to participate in the research. Additional interview candidates were recommended by the industrial supervisor from Codemagic[15] and contacted directly on Slack[16].

5 participants responded and agreed to the interviews. 3 of the participants were from companies that had been developing games, and the remaining 2 were working on AR mobile apps using Unity. These 2 companies were targeting mobile app stores rather than PC or console game distribution platforms. 2 of the game development companies had released very successful games on the popular PC game digital distribution platform Steam, while one of them had yet to release their game. Further background information is included on Table 2 in Chapter 4.

3.3 Interview process

5 interviews were conducted in February and March of 2022 to gather information for the thesis. Participants were given an outline of the interview topics in advance in order to prepare. The interviews lasted an average of 1 hour and were conducted through Google Meet[17]. The interviews were recorded for later analysis and machine transcribed using Otter.ai[18]. The interview guide was refined after the first interview on February 22.

[15] https://codemagic.io/start/
[16] https://slack.com/
[17] https://meet.google.com/
[18] https://otter.ai/

4 Results

Table 1. Overview of contributors.

Company	Interview Time	Project type	Distribution	VCS	Solution	Build location	Tests	Source Size (GB)	Build time (min)
A (anonymous)	March 10, 2022	Game	Steam	Git	Buildbot	Local	Some	500	30
Event Horizon	March 2, 2022	Game	Steam	Git	Jenkins	Local	Full	500	300
Plop	February 22, 2022	AR app	Google Play (Android), App Store (iOS)	Plastic SCM	Unity Cloud Build, Manual	Both	None	11	5
Furyion Games	February 22, 2022	Game	Consoles, Steam,	Git	Manual	Local	None	320	35
Mobi Lab	February 18, 2022	AR app	Google Play (Android), App Store (iOS)	Git	Codemagic	Cloud	None	0.08	5

This chapter will describe the results of the interviews, outlining the relevant findings. The aim is to list the encountered problems and solutions as they relate to using version control software and the various stages of automated tasks in a CI/CD pipeline: building, testing, and delivery. An overview of the background and solutions used by each contributor is included in Table 1.

4.1 Version control

4 out of the 5 teams used Git with Git LFS for version control. Some of these teams also used GitLab and one of the teams also opted to self-host their Git repository to save on costs. Familiarity and ubiquity were the stated reasons for opting for Git. They did not want to retrain their developers to use a new system because they were yet unconvinced of the benefits or did not believe the additional costs of Plastic SCM hosting were justified.

Only one team was using Unity Plastic SCM, which is the version control system recommended by Unity. This team stated that they had run into issues with syncing binary files with Git and had found the Unity-supported solutions easier to use (previously Unity Collaborate, currently Unity Plastic SCM). They stated that they had lost progress due to automatic merges with Unity Collaborate but weren't experiencing any issues with Unity Plastic SCM.

Merge conflicts in scenes and prefabs

A common problem in working with version control for Unity projects was merge conflicts in scene and prefab files. Unity stores information about how objects are placed in an environment (commonly known as a level in video games) in a structure called a scene. Smaller collections of objects that can be reused in multiple scenes and changed in one file without having to change every scene they appear in are called prefabs. See Figure 2 for an example of the structure and possible size of scene and prefab files.

```
73412  --- !u!65 &4009518631293673150
73413  BoxCollider:
73414    m_ObjectHideFlags: 0
73415    m_CorrespondingSourceObject: {fileID: 0}
73416    m_PrefabInstance: {fileID: 0}
73417    m_PrefabAsset: {fileID: 0}
73418    m_GameObject: {fileID: 4009518631293673145}
73419    m_Material: {fileID: 0}
73420    m_IsTrigger: 1
73421    m_Enabled: 1
73422    serializedVersion: 2
73423    m_Size: {x: 20, y: 63.92856, z: 20}
73424    m_Center: {x: 0, y: 31.96428, z: 0}
73425  --- !u!114 &4009518631293673151
73426  MonoBehaviour:
73427    m_ObjectHideFlags: 0
73428    m_CorrespondingSourceObject: {fileID: 0}
73429    m_PrefabInstance: {fileID: 0}
73430    m_PrefabAsset: {fileID: 0}
73431    m_GameObject: {fileID: 4009518631293673145}
73432    m_Enabled: 1
73433    m_EditorHideFlags: 0
73434    m_Script: {fileID: 11500000, guid: b890677eb46812849b37ba536434f9e7, type: 3}
73435    m_Name:
73436    m_EditorClassIdentifier:
73437    antiGravity: 1.5
```

Figure 2. Example of scene file content.

Usually, version control software can apply the difference from a commit to only the lines that changed in a text file. This allows multiple developers to work on the same file and merge their changes without having to manually resolve the differences in the file. If multiple developers change the same lines in the same file and then attempt to merge their changes this however results in a merge conflict.

Unity scenes and prefabs are stored in files that can be generated as either binary files or text files that use a custom subset of the YAML data serialization language. By default, they are stored as text files to increase compatibility with version control. Despite this, it is very easy to create merge conflicts in these files by having multiple developers edit the same scene or prefab in the Unity Editor. This is because these files are automatically generated, and developers are unaware of which lines they are changing by editing the object in the editor. The resulting merge conflicts are difficult to resolve because the generated text files are hard to read and understand. This results in wasted time and effort in resolving the conflicts or even lost progress due to overwriting changes.

All of the contributors were aware of or had experience with this issue. One team had experienced a loss of multiple days of progress due to this issue. They all avoided creating these merge conflicts by avoiding a situation where multiple developers change the same scene or prefab at the same time. This was done by either assigning commit privilege of each scene or prefab to a single developer, having general awareness among the team about which developer was currently working on which scene or prefab, and by having each developer make a new temporary local scene whenever possible instead of working in an existing one.

Repository size

Teams that were developing games for PC and console had source projects that reached hundreds of gigabytes in size. The main reason for this was a large amount of 3D models, textures, and sound files. Teams would reduce the detail of these assets to improve the performance and size of the built game client but still needed to keep the source-quality assets. They would also create or download packs of multiple different 3D models and then only use some of the included assets but still keep the entire pack for possible future use.

This led to teams creating a separate repository for just the source-quality asset files. The company that self-hosted their repository kept their source-quality assets in Dropbox and Syncthing instead. These teams would then only commit the reduced quality assets to their game project repository. This approach significantly reduced the size of their Unity project repositories but not by an order of magnitude.

This explains why these teams could not use Unity Cloud Build for automated testing, building, or distribution as Unity Cloud Build has a project size limit of 25 GB. Some of the contributors mentioned this as a big limitation for not just Unity Cloud Build but for many of the CI/CD services they had looked at. The large repository size introduced a high bandwidth cost and increase in build time. Needing to download a project hundreds of gigabytes in size and import it into Unity for every build made using most cloud services infeasible if they did not provide a dedicated machine option. Unity Cloud Build also does not provide this option.

4.2 Building

Each of the contributors had a different approach to regularly building their Unity project. Two of the teams did not automate builds at all and only built manually through a Unity Editor instance on a local machine. One of these teams had tried Unity Cloud Build but mostly relied on manual builds due to how long builds took in the Unity Cloud Build environment. One team used the dedicated CI/CD tool Codemagic. The remaining two contributors used Jenkins and Buildbot respectively to schedule automatic builds on local machines. This subchapter will describe each of these approaches in more detail along with the reasons the teams mentioned for using them along with general issues encountered.

Unity Cloud Build

As outlined in Chapter 2, Unity Cloud Build is the CI/CD platform recommended by Unity and should be the most frictionless during setup due to it being tightly integrated with Unity. Despite this only one of the contributors used it to any degree. The larger teams simply stated that they had not even tried it or could not use it due to the repository size restriction. Other contributors did not use it for the reasons mentioned in this chapter.

Building manually

The two contributors that mostly relied on building manually mainly stated that automating the process would take too much time or effort. One of these teams could not use Unity Cloud Build due to the repository size restriction and the other team found that manual builds were much faster. Both contributors expressed that they did not see enough value in seeking out a different solution to automate builds, as their team was able to provide builds to testers at a sufficient rate.

Codemagic

One of the contributors had tried Unity Cloud Build to automate their builds but had run into issues like slow updates to Xcode[19] in the cloud build environment and hard-to-diagnose issues. They made the decision to switch to the CI/CD service provider Codemagic because it was always up to date for supporting the latest mobile development tool versions and allowed them to log into the build machine to have oversight over the build process. This team already was using Codemagic for their other projects as well. Codemagic support helped the team set up their project on their platform.

[19] https://developer.apple.com/xcode/

Scripting builds

Building a Unity project can be done by opening said project in an instance of the Unity Editor manually and using the build functionality from within. Unity projects can be built for multiple different platforms (e.g., PC, game consoles, mobile devices) but this requires switching the target platform from within the editor. Building a Unity project can also be done by using the command-line arguments for the Unity Editor executable. This allows one to script Unity builds without opening the editor GUI and can be done for multiple platforms without having to manually switch. These scripts can be used with build scheduling tools like Jenkins to automate builds.

Two of the teams opted to schedule their builds on their own local machines through Jenkins and Buildbot respectively. These teams also had projects that were too large for Unity Cloud Build. The contributor that opted to use Jenkins stated that they found the solution to be very powerful. It allowed them to have full control over all parts of their building and testing. They found the tool itself a bit cumbersome and outdated as using it requires the developer to learn about its features and scripting language. However, they said that the knowledge on the internet along with the plugins allowed them to do everything they needed.

These teams scheduled builds on their local machines instead of using a cloud service. One contributor said that this is a common solution because development teams with 20 or more members can end up with leftover machines anyway. They valued the ability to use more powerful hardware as needed compared to Unity Cloud Build despite the more complicated setup. The restrictions of the Unity Editor license mentioned in Chapter 2 meant that these contributors had to include additional steps in their build scripts. These included activating the Unity Editor with a license file and an extra step that returns the license after building.

Build times

Build times varied greatly among the contributors from 5 minutes to 10 hours. This could be somewhat accounted for by differences in the size and complexity of their projects, but the teams also identified factors that weren't unique to their projects. One team also mentioned that their build time had gone from 7 minutes to 35 minutes just by upgrading to the 2021 Unity versions. All of the contributors reported having generally experienced multiple times shorter builds on their local hardware compared to cloud services.

One of the contributing factors was the availability of build cache and generated libraries. Libraries are generated when first importing a Unity project and build cache is created during the first build on a machine. Once generated these resources can be re-used and updated as needed unless they are deleted from the machine. These steps can be cached to greatly speed up subsequent builds (unless a clean build is needed). Providing a cache server for these resources is more relevant in cloud build platforms where the environment is likely to be completely wiped after each build. Unity Cloud Build is an example of a service that has implemented this feature. Building on dedicated local machines provided this benefit, as the contributors did not delete caches on the machine unless they encountered build issues.

The contributors were not aware of the exact hardware differences between their local machines and the hardware used by cloud services like Unity Cloud Build but still believed this to be a large contributing factor in the difference between build times. Game development introduces a lot of tasks that rely on GPU performance such as baking lighting. This difference in developer hardware and cloud hardware can be mitigated if the cloud build service provides the option to pay for building on better hardware but platforms like Unity Cloud Build and GitHub Actions don't have this feature.

One of the contributors from a larger team said their builds took 5-6 hours and with more internal tasks like generating caches or pre-rendering static shadows (baking lightmaps) it could easily go over 10 hours. They said that this was not unique among projects of their size and meant they could not build on-demand. Their only option was nightly builds. Once most of the developers had finished working during the day, they would automatically trigger a build that would run through the night. Then testers would download the finished build the next morning to go through their testing. Two of the contributors used this approach over regular builds triggered by commits or merges.

4.3 Testing

All of the contributors had manual QA testing but only some used automated testing in the Unity project. The smaller teams that were working on AR projects and iterating quickly did not feel the need to write tests for features they might soon discard. One of the game development teams planned to use tests at some point but was more focused on their main production effort and another one only used automated testing for a few features like their save/load system.

However, one of the contributors from a larger game development team was of the opinion that automated testing is critical. Their team had a few hundred tests including unit tests, integration tests, graphical asset tests, and asset naming checks. It would take an average player tens of hours to play through all of their game. This made playtesting difficult so they created a test that would automatically play through their game at increased speed to make sure all parts of it were completable.

They ran these tests on every Bitbucket pull request using Jenkins as their scheduler. Pull requests could not be merged unless all tests had passed. Developers would get automatically notified of test results and receive feedback directly on their pull requests through webhooks.

This contributor also broke their tests into batches because their entire test suite started to take longer than 45 minutes to run. Integration tests that required the Unity Editor to load a scene environment took much longer to run than unit tests that only required a C# runtime. They found that some of the tests that were the fastest to run also failed the most often so they ran those tests first so developers would receive feedback faster.

4.4 Delivery

Two of the contributors targeted their product for distribution to mobile app stores and the other teams all focused on the PC game digital distributor Steam and/or game consoles. Each of the specific distribution platforms has its own processes and tools for automating delivery. The use of said tools overlaps with the rest of the software development industry in the case of mobile app stores like Play Store and App Store since these platforms distribute more than mobile games. While platforms like Steam also host development software, media, and other product categories, they still mainly promote and sell games. This is why their delivery steps and tools are mostly used by game developers.

Steam for example has its own command-line tool for delivering new builds for distribution. This tool can be used in a post-build script to automate deployment to Steam. Steam can also be used to deliver builds to only testers instead of customers similar to Play Store testing groups or Testflight. Although all of the game development teams distributed or were planning to distribute to Steam some of them still preferred directly sharing build archives to testers instead of using this feature.

As mentioned, deployment to the relevant game distribution platforms can require the use of uncommon tools. This meant that contributors could not use deployment integrations. These are a feature on CI/CD platforms like Codemagic that allow developers to automatically deploy to mobile app stores for example. The feature simplifies a process that would otherwise have to be manually scripted. Unity Cloud Build has this feature, but the only integrations are with smaller app stores. Platforms like Steam, itch.io, Play Store, and App Store are missing from the feature. Contributors named this as another reason they were dissatisfied with Unity Cloud Build.

Another unique aspect of the delivery step was frequency. The mobile AR contributors released builds to production on a biweekly or slower basis depending on the number of new features or fixes in each sprint. This was not the case for the game development teams. All of the contributors that were working on games focused their efforts on releasing their product and then only delivering infrequent updates to fix issues as needed. None of these teams had a regular post-release update schedule for new features and fixes. Because of this they also did not have much incentive to automate their delivery process.

4.5 Response to issues with Unity Cloud Build

The previous subchapters outline the shortcomings of Unity Cloud Build that prohibited or dissuaded the contributors from using it. These included the 25 GB repository size limit, lack of hardware options, and lack of deployment integrations to popular game distribution platforms. Unity Technologies were contacted to comment on these issues and their plans for developing the service further.

In their response, they stated that they have identified the mentioned limitations and more and are working to resolve them. They plan to dramatically increase repository size limits, offer different build machine sizes for faster builds, introduce more advanced caching systems, and work on other improvements to be able to service customers of any size on the Unity Cloud Build platform. Their goal is for the platform to be a “one-stop-shop for all real-time 3D DevOps needs”.

5 Conclusions

5.1 Summary

The aim of the thesis was to gather information about how game developers approached using version control software and automating building, testing, and delivery in a CI/CD pipeline. Over the course of this thesis, 5 companies that used the Unity game engine were interviewed. They were asked about how they have implemented mentioned DevOps practices and overcome the difficulties of using CI/CD for game development. The information provided by their responses was analyzed and key findings were highlighted.

The use of version control software and repository hosting was significantly impacted by the size of the projects. The contributors provided explanations of how game projects can reach hundreds of gigabytes in size and how they mitigate the issue by separating source-quality assets from the Unity project to reduce the size of their main repository. Most of the contributors explained that they chose Git with Git LFS as their version control software over Unity Plastic SCM due to familiarity. One contributor did use Unity Plastic SCM and did not encounter any significant issues.

The research uncovered major shortcomings in the Unity-provided CI/CD platform Unity Cloud Build. Game development teams that were working on projects larger than 25 GB simply could not use it. These teams opted instead to either use third-party platforms like Codemagic, build their own custom solutions using tools like Jenkins and Buildbot, or to not automate builds at all. The contributors described build times in the CI/CD environment as a hindrance. This issue was mitigated by using dedicated machines or a build cache server.

All of the contributors relied on frequent QA testing and most of them did not feel the need to implement automated testing. However, one of the larger teams described their extensive automated testing setup with a custom test that played through their entire tens-of-hours-long game. Automating delivery saw a similar trend of most contributors not seeing it necessary, however, explanations were given for the uncommon tools involved and the uniquely infrequent releases of games.

5.2 Suggestions for further research

Similar research could be done to inquire about the solutions used by teams working on games with a significant server infrastructure for a multiplayer component or teams that release frequent content updates for PC and console games. The small number of contributors and the qualitative nature of the thesis also place restrictions on the nature of the conclusions that can be drawn. Further quantitative research could be done to identify the most common CI/CD tools used by game developers. Finally, case studies could be done on implementing solutions for the most technically challenging issues mentioned in the thesis (hosting large repositories, implementing build cache).

References

[1] Most used Engine; itch.io https://itch.io/game-development/engines/most-projects (10.05.2022)

[2] Erich, FMA, Amrit, C, Daneva, M - A qualitative study of DevOps usage in practice. J Softw Evol Proc. 2017; 29: e1885. https://doi.org/10.1002/smr.1885

[3] 2021 State of DevOps Report; Puppet; https://puppet.com/resources/report/2021-state-of-devops-report (10.05.2022)

[4] What is DevOps?; Microsoft Azure; https://azure.microsoft.com/en-us/overview/what-is-devops/ (10.05.2022)

[5] Leite, L., Rocha, C., Kon, F., Milojicic, D. and Meirelles, P., 2019. A survey of DevOps concepts and challenges. ACM Computing Surveys (CSUR), 52(6), pp.1-35.

[6] Soares, E., Sizilio, G., Santos, J., da Costa, D.A. and Kulesza, U., 2022. The effects of continuous integration on software development: a systematic literature review. Empirical Software Engineering, 27(3), pp.1-61.

[7] Foxman, M., 2019. United we stand: Platforms, tools and innovation with the unity game engine. Social Media+ Society, 5(4), p.2056305119880177.

[8] Shahin, M., Babar, M.A. and Zhu, L., 2017. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE Access, 5, pp.3909-3943.

[9] Yarlagadda, R.T., 2018. Understanding DevOps & bridging the gap from continuous integration to continuous delivery. Understanding DevOps & Bridging the Gap from Continuous Integration to Continuous Delivery', International Journal of Emerging Technologies and Innovative Research (www. jetir. org), ISSN, pp.2349-5162.

[10] Arachchi, S.A.I.B.S. and Perera, I., 2018, May. Continuous integration and continuous delivery pipeline automation for agile software project management. In 2018 Moratuwa Engineering Research Conference (MERCon) (pp. 156-161). IEEE.

[11] Jussila, T., 2021. DevOps in mobile game development: An action research on applying DevOps practices in a mobile game development project.

[12] Thobari, A.J.A., Sa'adah, U., Hardiansyah, F.F. and Putra, R.C.A., 2021, November. Toolchain Development for Midcore Scale Game Products through DevOps and CI/CD Approach. In 2021 5th International Conference on Informatics and Computational Sciences (ICICoS) (pp. 81-86). IEEE.

[13] Haas, J., 2014. A history of the unity game engine. Diss. WORCESTER POLYTECHNIC INSTITUTE.

[14] Unity; Unity Technologies; https://unity.com/ (10.05.2022)

[15] Made with Unity; Unity Technologies; https://unity.com/madewith (10.05.2022)

[16] Unity Featured Products; Unity Technologies; https://unity.com/products (10.05.2022)

[17] Why you should use CI/CD for Unity games; Markus Aksli; https://blog.codemagic.io/why-to-use-cicd-for-unity-games/ (10.05.2022)

[18] List of video games notable for negative reception; Wikipedia; https://en.wikipedia.org/wiki/List_of_video_games_notable_for_negative_reception (10.05.2022)

[19] 10 Most BROKEN Video Game Launches Of The Generation; Greg Hicks; https://whatculture.com/gaming/10-most-broken-video-game-launches-of-the-generation-2 (10.05.2022)

[20] 10 Games That Were Broken At Launch; Nick Steinberg; https://electronics.howstuffworks.com/10-games-that-were-broken-at-launch.htm (10.05.2022)

[21] 8 Games With Completely Broken Releases; Mike Williams; https://www.usgamer.net/articles/8-games-with-completely-broken-releases (10.05.2022)

[22] Video Games That Were Released Totally Broken; Zack Millsap; https://www.cbr.com/5-video-games-released-totally-broken/ (10.05.2022)

[23] 10 Video Games That Were Horribly Broken at Launch; Jasmine Henry; https://gamerant.com/broken-games-launch-batman-halo-diablo-155/ (10.05.2022)

[24] The worst launches in PC gaming history; PC Gamer; https://www.pcgamer.com/the-worst-pc-game-launches/ (10.05.2022)

[25] da Silva Lima, G.B., de Araújo, C.S., Rodriguez, L.C., Pinheiro, C.L. and da Silva Junior, J.M., DEVOPS METHODOLOGY IN GAME DEVELOPMENT WITH UNITY3D.

[26] Sakharov, V., 2019. Workflow optimisation in unity engine.

Appendix

I. Interview questions

1. What are you developing?
2. Are you working on one primary application or multiple?
3. What stage of development are you at?
4. Where do you (plan to) distribute?
5. How many developers do you have that interact with the Unity project source?
6. Which VCS do you use and why?
    a. Have you tried Plastic SCM or Unity Collaborate?
7. How large is your primary application repository?
    a. What makes it big?
8. What do you not check into version control?
    a. If you use git is your .gitignore similar to the default one for Unity?
9. How much art assets are you dealing with and how?
10. Do artists use version control directly?
11. What does your branching structure look like?
12. Do you do Pull Requests?
13. What is the procedure for merging pull requests/branches?
14. How do you deal with merge conflicts in the Unity project, especially scenes and prefabs?
15. What CI/CD tool do you use and why? Have you always used this tool? (Why not Unity Cloud Build?)
    a. Do you use the Build Server license or the regular editor license?
16. Do you upgrade Unity during the development cycle of a project?
17. How do you manage Xcode updates or Unity updates in the CI/CD environment?
18. Do you use cloud builds or a dedicated machine? Why?
19. Do you automatically trigger builds from commits/merges?
20. How often do you build?
21. Do you manage different flavors or your games (e.g. staging, production, dev)?
22. How often do you release new versions to production?
    a. If not released do you plan to run content updates? (what is your plan for them?)
23. Have you had to do hot-fixes? How quickly can you release hot-fixes/patches to production?
24. If you could change anything about your current solution what would you change?
25. Do you run any tests on the Unity source?
    a. Do you run just C# unit tests or also integration tests that run a scene?
    b. Describe your test suite and how it is run.
    c. How long does it take to run your tests compared to build time?
    d. Do you run CI checks on every Pull request?
    e. How do you deal with flaky tests?
26. Do you feel like tests (could) help merge features more consistently?
27. When does it make sense to build out a test suite?
28. Do you use any metrics to measure developer productivity? Do you have something like a project manager?
29. Do you run into any build issues close to launching a game?
    a. Do you have any inabilities/limitations with your current CI/CD setup?
    b. How do you deal with remote build failures?
    c. Are builds too long?

    d. Does it make more sense to build locally?
    e. Do you have build caching to improve from first-build time?
30. What do you think about the costs/benefits of DevOps practice? How did you decide that your solution made sense?
31. When does CI/CD become valuable?

II. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Markus Aksli**,

(*author's name*)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Barriers And Solutions In CI/CD Implementation for Unity Game Development,

(title of thesis)

supervised by Chinmaya Dehury, Ulrich Nobisrath, and Martin Jeret,

(supervisor's name)

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **11.05.2022**