

## Deploying Software in the Cloud with CI/CD Pipelines

Ramesh Ghimire

Bachelor's Thesis  
Degree Programme in  
Business Information Technology  
2020





<b>Author(s)</b> Ramesh Ghimire	
<b>Degree programme</b> Business Information Technology	
<b>Report/thesis title</b> Deploying Software in the Cloud with CI/CD Pipelines	<b>Number of pages and appendix pages</b> 42+ 6
<p>This thesis deals with the practice of software deployment using Continuous Integration and Continuous Delivery pipelines. It introduces modern practices related to System Development Life Cycle such as version control, testing, delivery and deployment of software in the cloud managed under Agile framework and demonstrates an example which can be picked up by any software developer without a hassle.</p> <p>Emergence of cloud service providers and ever-growing usage of their offerings have shifted the information technology landscape and created more opportunities for software developers and usage of CICD Pipelines have become a standard industry practice. There are different approaches to CICD in the industrial settings and different companies have different tools and workflow set up to run this process. Some companies have dedicated departments for example, DevOps engineers are those technicians who configure the different moving parts of CICD pipeline. However, any small development team can operate with DevOps mentality themselves using the available tools and technology and DevOps principle promotes this. This thesis demonstrates the concepts and example implementation of CI/CD pipelines and the emphasis is given to deploying the software in the cloud using as less tools as possible so that it integrates well with the workflow of software developers and small companies as well as does not incur additional large sum of costs.</p> <p>To do so, theoretical background as well as different approaches and possibilities of the CI/CD Pipeline implementations for software deployment in the cloud are explored from available academic sources. This includes examining the diversity of technology and choices for a software developer, followed by implementing an example solution. And for the purpose of this implementation, AWS cloud and Bitbucket git repository are used. AWS provides a wide range of cloud services for different purposes including software deployment and Bitbucket provides git version control services in the cloud. Git is a technology used for software version control by developers. These tools are not the only ones available for software developers and the demonstrated approach can easily be adapted to any other alternative tools of choice.</p>	
<b>Keywords</b> Continuous Integration, Continuous Delivery, Pipeline, Cloud, Software	

## Table of contents

Abbreviations Used .....	1
1. Introduction .....	2
1.1. Purpose of this Work .....	4
1.2. Structure of this Work .....	4
2. Theoretical Framework .....	5
2.1. Early era of Computing .....	5
2.1.1. Physical Limitation of Computing Resources .....	6
2.1.2. Virtualisation Technology .....	6
2.1.3. Cloud Computing .....	7
2.1.4. The Docker Container .....	7
2.2. System Development Life Cycle .....	9
2.3. Agile Methodology .....	9
2.4. Deploying Software in the Cloud .....	12
2.4.1. Version Control with Git .....	12
2.4.2. Continuous Integration .....	13
2.4.3. Continuous Integration as a development process .....	13
2.4.4. Continuous Delivery .....	15
2.4.5. Continuous Deployment .....	16
2.4.6. Continuous Delivery vs Continuous Deployment .....	16
2.4.7. Automated CICD Pipelines .....	17
2.4.8. Advantages of automated CICD Pipelines .....	18
2.5. Containerised Deployment .....	18
3. Proof of Concept .....	19
3.1. Short Introduction of the Technology in Use .....	19
3.2. Process Outline .....	21
3.3. The Architecture .....	22
3.4. Implementation Details .....	22
3.4.1. Development Environment .....	23
3.4.2. Git Repository and the CICD Pipeline .....	23
3.4.3. Production Environment .....	24
3.4.4. IAM instance profile Configuration .....	25
3.4.5. EC2 Instance Creation .....	26
3.4.6. S3 Storage Configuration .....	27
3.4.7. CodeDeploy Service Configuration .....	27
3.4.8. Configuring EC2 instance as production server .....	28
3.4.9. Environment Variable Configuration in Bitbucket Pipelines .....	29
3.4.10. Adding features to the CICD pipeline .....	30

3.4.11. Pipeline optimisation .....	32
4. Conclusion.....	37
5. Afterword and Recommendations.....	38
5.1. CICD Pipeline Adoption Strategies .....	38
5.1.1. Bespoke Solution.....	39
5.1.2. DevOps Department.....	39
5.1.3. Temporary Hire.....	39
5.1.4. Inhouse Setup .....	39
5.2. Commercial and Free CICD Pipeline Providers.....	40
Appendices.....	43
Appendix 2. Images and screenshots.....	44

## Abbreviations Used

Abbreviation	Meaning
AMI	Amazon Machine Image
API	Application Programming Interface
ARN	Amazon Resource Name
AWS	Amazon Web Services
CD	Continuous Delivery (also Continuous Deployment)
CI	Continuous Integration
DevOps	Development Operations
EU	European Union
FTP	File Transfer Protocol
FTPS	File Transfer Protocol with Transport Layer Security support
GCP	Google Cloud Platform
GNU	Gnu IS NOT UNIX
HDD	Hard Disk Drives
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IaaS	Infrastructure as a Service
IAM	Identity and Access Management
ICT	Information and Communication Technology
IDE	Integrated Development Environment
SDLC	System Development Life Cycle
SN	Serial Number
SSH	Secure Shell
URL	Uniform Resource Locator
VIM	VI Improved

## 1. Introduction

There are many steps involved between a developer's text editor where the code is written to the end users' machine where the software runs. The process of making a software ready for usage for end users, as well as for beta-testers and other intermediary stakeholders, is called deployment or delivery. These steps collectively form the part of SDLC. Various stages in the SDLC mandate relevant tasks to be performed in those stages. Because managing those stages without a clear standard or industry practice often is exceedingly difficult for most of the projects, there are different practices in use to successfully manage the SDLC. These practices are referred to as methodologies or frameworks. Two of the most common methodologies which are in use are Waterfall and Agile. In this work the focus will be mostly given to Agile methodology.

"Software methodologies now focus on developing software applications that are highly adaptable and can be changed easily over time, resulting in less of a need for rigid, up-front planning." (Ingeno, 2018)

"Agile is a set of methods and methodologies that are optimized to help with specific problems that software teams run into and kept simple so they're relatively straightforward to implement. These methods and methodologies address all of the areas of traditional software engineering, including project management, software design and architecture, and process improvement. Each of those methods and methodologies consists of practices that are streamlined and optimized to make them as easy as possible to adopt." (Stellman & Greene, 2017)

As referenced above, agile is a framework that emphasizes on repetitive development in small incremental well-defined chunks. Agile has been gaining popularity in recent years and it is proven to be remarkably effective in handling extraordinarily complex to simple software development projects. Because Agile methodology emphasises on feedback, it allows improvement of in the overall workflow by gathering and applying the knowledge of limitations, benefits, or any other issues, roadblocks etc. related to the work set up and supplies a chance to continuously improve the process itself. Administrative tasks related to the software development requires mandatory participation of developers and other technicians in Agile methodology which often results in clear specifications and very well-defined road map of the software development.

The various stages of the SDLC and the tasks involved there need to be somehow integrated well to make sense and deliver a product. Some of the most common tasks except

the administrative tasks in the SDLC are writing code, maintaining the code-base in a repository, testing the code, testing the functionality of the software, building the software with build tools, adding features to the software, fixing bugs, realising new versions and managing versions, maintaining the infrastructure, moving to different infrastructure etc. These tasks are separate tasks and all of them are particularly important for a successful software development project. Managing these tasks manually is very time consuming and resource intensive and is often prone to errors. Hence, we need automated system which takes care of connecting these tasks and running them based on the defined set of rules. This system is analogous to an automated Continuous Integration and Continuous Delivery (and Continuous Deployment) pipeline.

A practice which simplifies the process of performing the development tasks of SDLC and focuses on quality assurance and successful release of the product is called Continuous Integration and Continuous Delivery (and Continuous Deployment). As the name suggests, it is a continuous process and a part of the overall software development project. A set of tasks which are well defined and are automatically triggered based on the defined rules and correspond to the CICD practices is a CICD pipeline. The practice of Continuous Integration and Continuous Delivery (and Continuous Deployment) is perfectly compatible with Agile methodology of software development project management.

“Continuous Integration, Delivery, and Deployment are relatively new development practices that have gained a lot of popularity in the past few years. Continuous Integration is all about validating software as soon as it's checked in to source control, more or less guaranteeing that software works and continues to work after new code has been written. Continuous Delivery succeeds Continuous Integration and makes software just a click away from deployment. Continuous Deployment then succeeds Continuous Delivery and automates the entire process of deploying software to your customers (or your own servers).” (Sander Rossel, 2017)

One important purpose of adopting a CICD pipeline in the industrial scene is to effectively integrate code changes, effortlessly perform the tests and reducing the time for the software delivery into production so that the new features are available for the end users in the earliest manner and the company stays ahead of the competition. For the software developers, adoption of CICD practices frees them from mundane repetitive tasks and supplies them more time to focus on developing new features at the same time reduces the risk of introducing many bugs and misconfiguring the infrastructure.



## **1.1. Purpose of this Work**

The purpose of the work is to present the opportunities presented by the CICD pipelines and DevOps practices for software developers and small development teams. Since the practice of DevOps, along with adoption of CICD pipelines and deployment in the cloud, is an emerging trend, not only big organisations but individual developers and small organisations should also adopt the practices and leverage the technology for high quality software production. Once the DevOps mindset is in place, setting up the pipeline and using it can be remarkably simple and easily integrate with the existing workflow.

This work goes through the theoretical background and demonstrates an implementation of a fully automated CICD pipeline which is capable of integration, delivery, and deployment of software. In order to demonstrate that any software developer or a small team can easily implement such technology, a bare minimum number of tools are used, and the setup is done in a fashion that does not incur too much extra cost and can be spun up very quickly. Hence, the intention behind example implementation of the discussed ideas is to demonstrate how easily, quickly, and cost-effectively a CICD pipeline can be created and integrated in the workflow of individual developers and small teams.

The type of research conducted for this work is qualitative research where contextual phenomenon are described in a narrative and explorative fashion and the scope of the inquiry revolves around the thematic concerns of the topic. Existing literature on the topic are the primary source of information that are considered in order to establish the theoretical knowledge base. The theoretical background thus established are explained in the following chapter. Moreover, third party publications and research papers, developer surveys, official documentations from service providers etc. are extensively referred and cited in the reference page.

## **1.2. Structure of this Work**

Theoretical background is first established by the next section titled that is followed by other sections which probe into discourse on conceptualisation and an implementation as the proof of concept, further discussions, conclusion, and foreword. The next section, Theoretical Framework, gives a short presentation of the technology and establishes ground for the implementation of the theory which will be presented in the subsequent sections.

An example software is written in flask framework and delivered using an automated CICD pipeline which is described in the section 3. Code samples of the software, design diagrams and other specifications, the usefulness of the software and the discussion on

architecture of the software design, however, is out of scope of this work. Chapter 4 follows with general discussion and ends with few take away ideas for the reader.

## 2. Theoretical Framework

Information Technology landscape has been changing exponentially by introducing contemporary trends and technologies every year. However, we still need to write code, test code, deploy the code and deliver the software. This task was vastly different than how it is today during the pre-agile era of software development.

### 2.1. Early era of Computing

During early era of computing before the advent of Personal Computers, computing mostly happened using terminals which would connect to the mainframe computers. It was very limiting, and the computing was still a scientific equipment accessible to some elite group of people. Information Technology industry was ridiculously small and industry practices were in the early stage of evolution. Then came the era of personal computing and revolutionised everything. Computers became personal and the golden era of Information Technology started.

“However, mainframes were not suitable for all workloads, or for all budgets, and a new set of competitors began to grow up around smaller, cheaper systems that were within the reach of smaller organisations. These midcomputers vendors included companies such as DEC (Digital Equipment Corporation), Texas Instruments, Hewlett Packard (HP) and Data General, along with many others. These systems were good for single workloads: they could be tuned individually to carry a single workload, or, in many cases, several similar workloads. Utilisation levels were still reasonable but tended to be around half, or less, of those of mainframes.” (Clive Longbottom, 2017)

When computers became personal, and popular, the industry expanded with wide range of services and infrastructures started relying on the Information Technology. Invention of the Internet took it to the next level and business processes started relying on the internet. Until this point in time we relied solely on Client – Server infrastructure with *one server application* serving many clients. Although the application would not need all the resources available on the server, it would just occupy it which was expensive and restrictive was of doing things. And organisations often need multitude of applications running on multitude of corresponding servers. Also, these servers were physical devices which required effective storage, maintenance, and security.

### **2.1.1. Physical Limitation of Computing Resources**

During the 80s and early 90s the scalability in large scale applications had the physical limitation of hardware scaling, which often took a lot of time, resources and caused low latency and multiple service breaks. Scalability of applications relied on physical availability and configuration of, mainly:

1. Processing (Compute)
2. Storage (HDDs)
3. Networking (bridges, routers etc.)

Industry standard of the time and the technology in use was not very efficient for globalised ICT industry. As a result of that innovative technologies started to appear and we entered the era of virtualisation. IBM and VMware were among the first ones who invented and used the virtualisation technology.

### **2.1.2. Virtualisation Technology**

Using virtualisation technology, not only, hardware resources can be sliced, added, distributed, merged etc. as per requirement and on the fly, but also these operations can all be managed from the application layer without a need of having to configure the hardware. This presented with new opportunities for the software developers who can now fully focus on the software development and forget about all the hardware configurations while being able to treat infrastructure as a program or code. "Virtualization is a disruptive technology, shattering the status quo of how physical computers are handled, services are delivered, and budgets are allocated." (Portnoy, 2012)

In recent years, the price of infrastructure has gone down significantly due to virtualisation. Also, the credit for making cloud computing happen goes to the virtualisation technology. Companies now could sell IaaS offerings (e.g. Google Cloud, Microsoft Azure, Amazon AWS etc.) which could be bought and used as if it were the real hardware without even having to know where the real hardware lies. Virtualisation also brought an opportunity to build and deploy software in a whatever hardware configuration needed for a very affordable price which led to subsequent decrease in infrastructure costs for application developments.

### **2.1.3. Cloud Computing**

Reduced costs of infrastructure, Virtualisation technology that enabled virtualisation of not only the compute resources but storage, networking, and services as well, helped in the emergence of the Cloud. The cloud is a service, where IT infrastructure is provided in subscription basis where these infrastructures are usually virtualised hardware resources. This service may include, without limitations, storage, network, memory etc. upon which any kind of software services may be built.

These cloud services are used by individuals and organisations for various purposes. Using the cloud liberates organisations from setting up and supporting the infrastructure themselves which in turn reduces the business costs and increases profitability. Big tech giants such as Google, Amazon, Microsoft, and IBM provide their commercial public cloud offerings which have a wide range of services for commercial use.

The industry trend now is shifting towards containerised deployments of software in the cloud. This means that a virtualised infrastructure resource in the cloud can be shared by many applications running in their own private environments. This also addresses the limitation that we would otherwise have for implementing Service Oriented Architecture. A Service Oriented Architecture is a pattern of software design where application components are built around their functionality and inter-networked. Details on Service Oriented Architecture and Microservices pattern is not in the scope of this work.

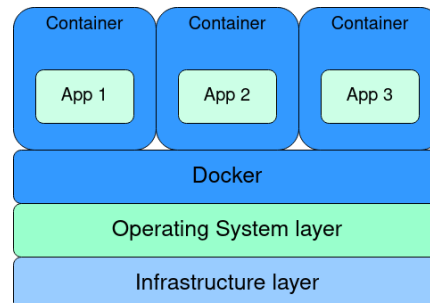
There are quite a few tools and technologies a developer can leverage for containerised deployments of software. The most common technology, which is used in the example implementation of a CI/CD Pipeline for the purpose of this work, is Docker. Other similar technologies include Podman, Vagrant, CoreOS rkt, Mesos, LXC, OpenVZ etc.

### **2.1.4. The Docker Container**

Docker can be defined as the technology that enables developers to encapsulate code, its dependencies and other environment variables into a package called an image that can be used to instantiate the application as a container.

“Developers use Docker containers to package their applications, frameworks, and libraries into them, and then they ship those containers to the testers or operations engineers. To testers and operations engineers, a container is just a black box. It is a standardized black

box, though. All containers, no matter what application runs inside them, can be treated equally. The engineers know that, if any container runs on their servers, then any other containers should run too. And this is actually true, apart from some edge cases, which always exist.” (Schenker, 2020)



*Figure 1: Containerised Applications with Docker*

“A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.” (Docker Inc., 2013)

“Docker provides an abstraction. Abstractions allow you to work with complicated things in simplified terms. So, in the case of Docker, instead of focusing on all the complexities and specifics associated with installing an application, all we need to consider is what software we’d like to install.” (Nickoloff et al., 2019)

Using the docker technology, multiple containers can be run on top of an infrastructure and the resource can be shared. This also adds the benefit of portability as these images can be easily transferred to various locations and new containers can be spawned from these. Which makes it whole lot easier to migrate and replicate infrastructures and software, as well as allows us to save the state of an application as snapshots. Figure 6 represents a simplified version of containerised deployment implementing the Docker technology.

“Innovation in many domains is creating compound effects, given the shared platforms of PCs, smartphones, and the Internet. Looking briefly at a range of technologies as resources for innovation should help drive further breakthroughs.” (Jordan, 2012)

## **2.2. System Development Life Cycle**

Like the construction of real-world objects, information systems are conceptualised, commissioned, planned, designed, built, tested, implemented, and maintained. This process is true for every information system and only varies in internal stages and diversity of tools, technology and frameworks applied there. This process is commonly referred to as the System Development Life Cycle. A project may be made up of a single completion of cycle, multiple cycles or confined in one or multiple stages of a life cycle.

Most common stages of SDLC which most of the projects consist of are: Initiation, Requirement Analysis, Design, Development, Testing, Implementation and Maintenance. In agile framework, these stages occur often and are applied repetitively to small chunks of work called Scrum in small periods called Sprint rather than the whole project. DevOps often incorporates the agile way of working, however, in modern technology landscape there are challenges that a DevOps team faces such as:

- I. Regular integration needs more resources.
- II. Regular testing also requires more resources.
- III. Regular integration and maintenance require more resources.
- IV. Development work may slow down during testing and maintenances.
- V. System interruption / failures become frequent.
- VI. Task switching and reassignment may lower developer morale.

These issues are effectively solved by the practice of Continuous Integration and Continuous Delivery which is described in later chapters.

## **2.3. Agile Methodology**

Agile methodology introduced several practices such as incremental development, repetitive testing, flexibility, increased feedback loops and customer focus etc. which organised the development more effectively than its predecessor frameworks at the same time presented us with challenges of needing frequent releases, frequent tests, and frequent deployments.

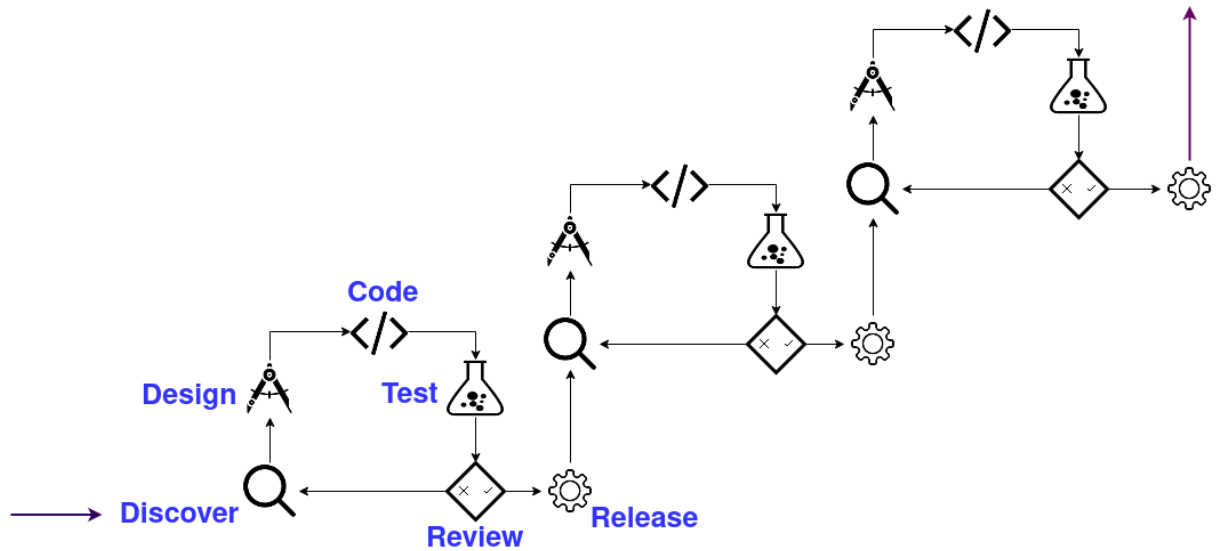


Figure 2: A typical system (software) development process under Agile framework.

Figure 2 above represents Agile framework applied to software development project with six most common stages repeating in cycles. The multiple releases can be different versions of the software or new features and bug fixes. This is an over simplified representation of actual software development process which often may include many more different intermediary stages, several types of tests and releases etc.

Moreover, emergence of cloud technologies, reduction in infrastructure costs, architecture designs including service federation, and organisations adopting newer way of working such as adoption of DevOps practices etc. mandate automation of repetitive tasks for effective and efficient project management. As a result of that, automated CICD pipelines are being developed and adopted by software development teams. As represented by the following figure, already over 50% companies are shifting towards adopting DevOps practices, including agile framework for software development project management, and adopting CICD pipelines for the development automation. And this trend is increasing every year.

As represented by the exhibit bellow, DevOps practices are becoming the industry norm:

## DevOps titles are the norm...

Organizations use DevOps in titles – 69% have a team with DevOps in the name and/or employees with DevOps in their titles.

- Larger organizations with 500+ employees are especially likely to have a team with DevOps in the title (57%).
- Organizations that have been using DevOps for 3+ years are also more likely to have a team with DevOps in the name (57%).

Which of the following best describes your organization's management of DevOps? Please select all that apply.

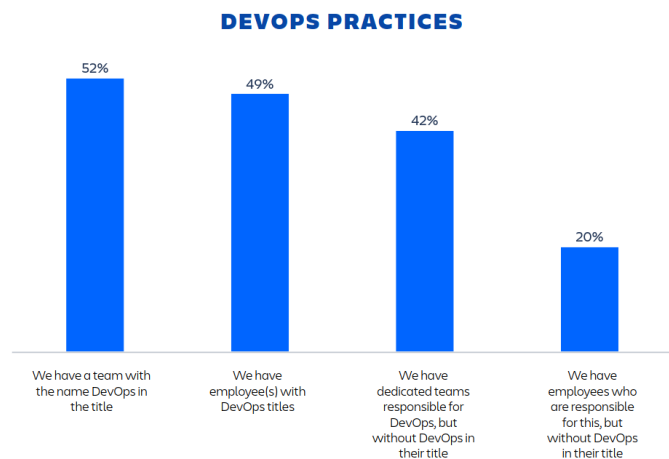


Figure 2.1: Exhibit extracted from 2020 DevOps Trends Survey by Atlassian & CITE Research

CICD pipelines are becoming increasingly commonplace in companies which want to automate the software development process nowadays. In a quite common and emerging DevOps practice, pipelines supported by dedicated technicians. The main principle behind the DevOps practice is to incorporate the development and operation process in a meaningful way and reduce barriers between the development and operation. DevOps related practices including design and maintenance of CICD pipelines are particularly important skill to gain as a developer as well because many small companies may not have a dedicated system administrator or DevOps engineers dedicated to creating and supporting the continuous integration and continuous deployment pipelines.

“DevOps—derived from Development Operations — culture came in to being to increase collaboration between development and operations. Organizations built DevOps teams with engineers from development and operations backgrounds to eliminate the communication barrier between these groups. Besides, many practices and tools are implemented to increase automation and decrease the delivery times and minimize the risks. Eventually, this culture shift in organizations fostered quality and reliability with reduced lead times. In these new teams, developers acknowledged operational knowledge such as cloud providers and customer environments.” (Onur Yilmaz, 2018)

“We’ve seen how DevOps has grown from a term only familiar to technical teams to becoming part of the C-suite vocabulary. Practices like CI/CD and automation have become



the norm in every engineering organization.” (2020 DevOps Trends Survey, Atlassian & CITE Research)

## **2.4. Deploying Software in the Cloud**

Various stages of operation such as development, version control, Continuous Integration, Continuous Delivery, and Continuous Deployment collectively form the CI/CD pipeline. The pipeline is used to deploy software in the cloud. Continuous Integration is the first stage of the pipeline which helps developers to integrate their code into the repository often. Continuous Delivery on the other hand, as the name suggests, is the process of making the code ready for deployment into production and Continuous Deployment takes care of automatic deployment of ready code. The whole pipeline can be triggered by a pre-defined set of rules and developers can work on their code and the integration, delivery and deployment can happen as often as required.

“By creating fast feedback loops at every step of the process, everyone can immediately see the effects of their actions. Whenever changes are committed into version control, fast automated tests are run in production-like environments, giving continual assurance that the code and environments operate as designed and are always in a secure and deployable state.” (Kim et al., 2017)

### **2.4.1. Version Control with Git**

Version Control is the technology that allows tracking and managing changes to the code, enables developers to collaborate on a common code base, supplies detailed insights on code changes and makes it possible to roll back to the earlier state effortlessly.

“As the name implies, Version Control is about the management of multiple versions of a project. To manage a version, each change (addition, edition, or removal) to the files in a project must be tracked. Version Control records each change made to a file (or a group of files) and offers a way to undo or roll back each change.” (Mariot Tsitoara, 2020)

It is almost unimaginable for developers nowadays not to make use of a version control technology. Every developer team works with version control systems. There are several types of version control systems and Git is the most popular one. According to the Stackoverflow’s Developer Survey, 90% of the developers use Git version control system.

## 2.4.2. Continuous Integration

Continuous Integration is the practice of *continuously integrating* code into the repository. A software repository is where all the code written by every developer is stored and maintained. Often a repository contains many temporary sub-repositories which are called branches. Branches can be formed based on the developer where each developer works on a dedicated branch, or on a feature where every feature is developed on its own branch or according to the development convention of the team. The primary repository from where these branching off is done is called the master branch. Practice of Continuous Integration means that the branches are *merged* into the master branch often. This can happen even multiple times a day.

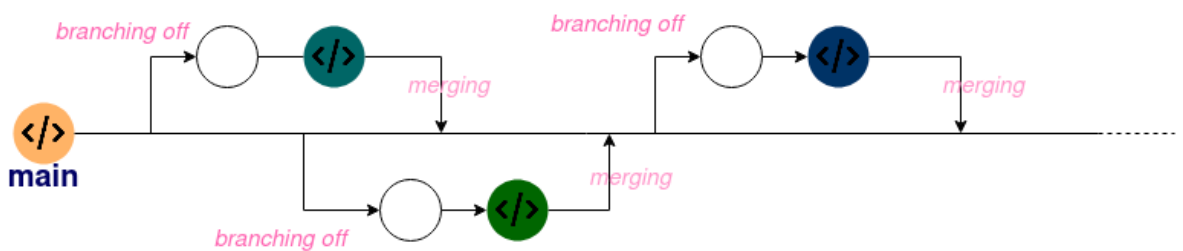


Figure 3: Simplified Representation of Continuous Integration in Practice.

When developers perform integration often, or when the integration happens frequently, the integration cost is reduced. A small chunk of code is easy to integrate, test and debug than a large codebase. Doing so, conflicts between the code, unexpected behaviours, unintended side-effects etc of the integrating code can be easily and early caught and resolved.

## 2.4.3. Continuous Integration as a development process

Figure 3 above represents a typical Continuous Integration process in a simplified way running tests, conflict resolution etc. are not included for the sake of simplicity. Main branch represents the master branch in Figure 3. In the typical practical scenario, codes are tested with defined tests, checked against conflicts with existing code in the master branch and pull requests are created before merging with the master branch. Figure 4 below demonstrates some control measures such as running tests and resolving conflicts before merging.

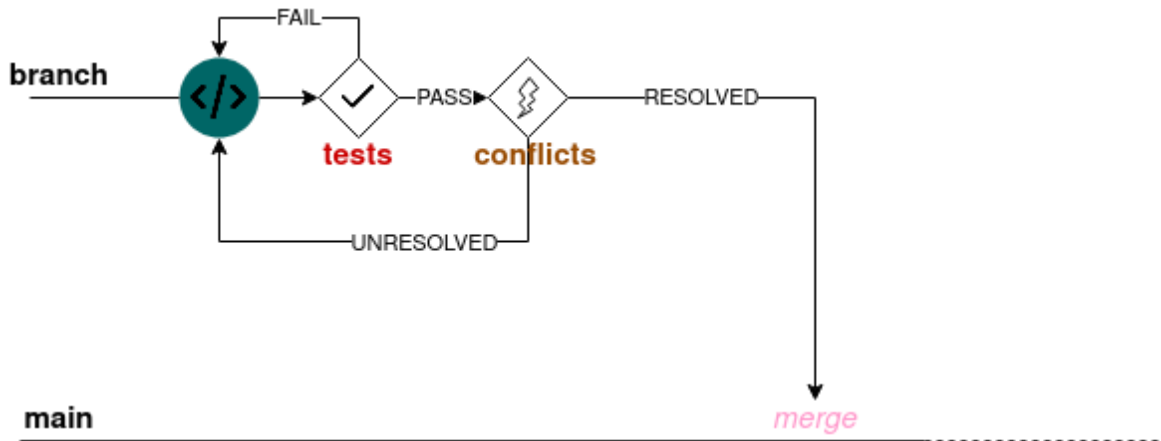


Figure 4: Running Tests and Resolving Conflicts before Merge in CI Process.

Often development teams may integrate code review and check against the quality control measures in the CI process. Details on quality control measures, types of tests and repository practices etc. are not in the scope of this work.

“A CI scenario starts with the developer committing source code to the repository. On a typical project, people in many project roles may commit changes that trigger a CI cycle: Developers change source code, database administrators (DBAs) change table definitions, build and deployment teams change configuration files, interface teams change DTD/XSD specifications, and so on.” (Duvall et al., 2007)

In the Book Continuous integration: improving software quality and reducing risk, Steve Matyas, Paul M. Duvall and Andrew Glover explain the typical process of CI as:

1. *First, a developer commits code to the version control repository. Meanwhile, the CI server on the integration build machine is polling this repository for changes (e.g., every few minutes).*
2. *Soon after a commit occurs, the CI server detects that changes have occurred in the version control repository, so the CI server retrieves the latest copy of the code from the repository and then executes a build script, which integrates the software.*
3. *The CI server generates feedback by e-mailing build results to specified project members.*
4. *The CI server continues to poll for changes in the version control repository.*

Hence, Continuous Integration starts from developers' initiative and is easily integrated with their existing workflow. Adoption of CI not only makes developers more efficient but also increases the quality of the code and aids in the overall project management.

#### 2.4.4. Continuous Delivery

Continuation Integration is extended by the Continuous Delivery process which includes the automation of the software delivery process. This means that the software is ready for deployment into production any time. Some companies prefer to manually deploy the software into production, however, the next stage in the pipeline Continuous Deployment enables us to deploy the software automatically if we chose to do so. Continuous Delivery benefits developers by reducing the deployment failure risk and smaller deployable software means frequent feedback from the customer which subsequently increases the quality of the software.

“Continuous Delivery is a set of technical practices that allow delivery teams to radically accelerate the pace at which they deliver value to their users. The core tenet of Continuous Delivery is keeping your codebase in a state where it can be shipped to production at any time. By working in this way, you can quicken the tempo of production changes, going from infrequent, big, and risky deployments to deployments that are frequent, small, and safe.” (Hodgson, 2020)

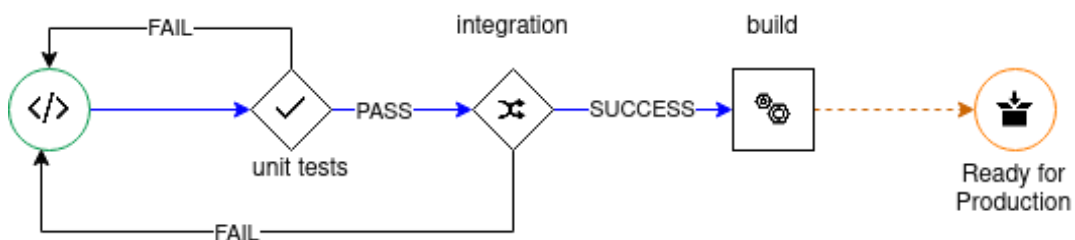


Figure 5: A Typical Continuous Delivery Process

A typical Continuous Delivery process replicates the environment that is close or identical to the production environment and builds the software. Like Continuous Integration, this happens in small chunks repeatedly over time. This enables software developers to always be certain that their code is always deployable in the production environment. Continuous Delivery usually contains test suites which are run against the build, and deployment processes. Figure 5 above depicts a typical Continuous Delivery process.

## 2.4.5. Continuous Deployment

Continuous Deployment is the process of automatically deploying to the production environment when the software is built successfully by preceding Continuous Delivery process and all the tests are passed. Some organisations may decide to not use the Continuous Deployment because they may want to deploy manually based on their deployment schedule. However, there is a huge benefit in adopting the Continuous Deployment process because it enables organisations to reach the users, fix bugs, release new versions faster.

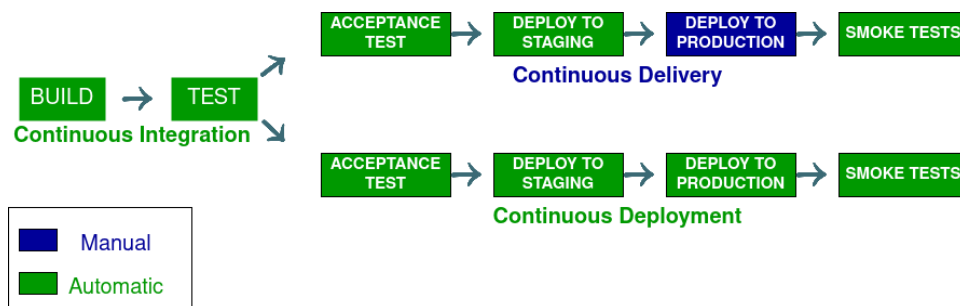


Figure 6: Simplified Representation of Continuous Deployment Process

## 2.4.6. Continuous Delivery vs Continuous Deployment

Figure 5 above depicts the Continuous Deployment process being triggered after successful build from Continuous Delivery stage of the pipeline. Continuous Deployment, if implemented, also happens repetitively in small chunks. In typical deployment scenario, Continuous Deployment process gets triggered as soon as the Continuous Delivery process is complete, and the software is ready for the deployment. This process takes care of moving the code into the production environment, often server infrastructure in the cloud in modern organisations, and making the new software available for the users.

A typical difference between Continuous Delivery and Continuous Deployment often, hence, is about how the software makes its way into production. This means that not all steps of the pipeline are automated and some, especially deployment into production, is handled manually by man intervention. Depending on the situation, this may give more control to the developers, but at the same time add some overhead and interruption to the complete process. "However, most of the organizations that I talked with avoid full-on Continuous Deployment. Instead, they institute some sort of manual gate, requiring an engineer to explicitly promote their changes into production from a preproduction environment. This wouldn't be considered Continuous Deployment, but it is still a form of Continuous Delivery." (Hodgson, 2020)

Although often confused, Continuous Delivery and Continuous Deployment are two separate processes. Sometimes these two steps are combined into one as a part of an automated pipeline and the explicit distinction may not exist. However, they both are important processes and commonly complete the CICD Pipeline.

#### **2.4.7. Automated CICD Pipelines**

An automated CICD pipeline is formed by creating a single system which combines Continuous Integration, Continuous Delivery and Continuous Deployment into one and is fully automated so that it does not require much human intervention when it is triggered. Leveraging an automated CICD pipeline can reduce costs and eliminate human errors. It may also provide important feedbacks about the code, checkpoint for passing the quality code, quick reach to the end users and opportunity for developers to focus on writing good software.

In their book titled Continuous Delivery, Jez Humble and David Farley argue that the manual deployment of software is an antipattern and must be avoided. They point out that this may lead to many human errors but also acknowledge that it is, however, common across organisations. Moreover, they mention that:

*Most modern applications of any size are complex to deploy, involving many moving parts. Many organizations release software manually. By this we mean that the steps required to deploy such an application are treated as separate and atomic, each performed by an individual or team. Judgments must be made within these steps, leaving them prone to human error. Even if this is not the case, differences in the ordering and timing of these steps can lead to different outcomes. These differences are rarely good.*

Furthermore, they go on and emphasize that the software deployment should be an automated process:

*Over time, deployments should tend towards being fully automated. There should be two tasks for a human being to perform to deploy software into a development, test, or production environment: to pick the version and environment and to press the “deploy” button.*

#### **2.4.8. Advantages of automated CICD Pipelines**

Adoption of an automated CICD Pipeline changes the development, testing and deployment process completely. Workflow are highly optimised, and errors are significantly reduced this is made possible because the code must go through the defined stages of the pipeline subjected to automated tests which ensure errors are caught and rolled back before they ever reach production.

It is estimated that the usage of such pipelines will only increase. The major advantages of adopting such a pipeline can be outlined as:

- i. More Efficient Development
- ii. Frequent Deployments
- iii. Quick and Efficient Testing
- iv. Less Bugs
- v. Better Developer Experience
- vi. Reduced Costs
- vii. Standardised Process
- viii. Better Software Quality

#### **2.5. Containerised Deployment**

Containerised Deployment means to deploy software in their own private environments called containers. These containers often are minimal version of operating systems with added software needed to run the deployed software. Combined with virtualised servers in the cloud, it created immense possibilities for the developers and organisations. To understand the technology behind containers and containerisation, it is mandated to look at the development of the Information Technology itself.

### 3. Proof of Concept

This project implements the theoretical framework established in the previous chapters. The implementation contains a software, a fully automated CICD pipeline, an instance of a production environment in the cloud. Since this work is focussed on the cloud, most of the tools used are cloud based except the text editor an IDE and local environment for writing the code. There are unlimited possibilities of tools and technology, the following are the tools and technologies that are used considering the ease of integration with existing tools for developers, pricing and difficulty of set up:

- Flask Framework.
- Local Environment:
  - A GNU/Linux Operating System with VIM for the text editor and PyCharm for the IDE.
  - Git Verson Control System.
- Cloud Software Repository:
  - Bitbucket (URL: <https://bitbucket.org>)
- CICD Pipeline
  - Bitbucket Pipelines
  - AWS CodeDeploy
  - AWS S3
  - Docker
- Production Server
  - AWS EC2

#### 3.1. Short Introduction of the Technology in Use

Flask is a Python framework used to build web services and applications. In this implementation, flask is used to create a backend API. This implementation can be tweaked to be used for any other software framework and programming language.

This implementation is system agnostic with respect to the operating system even though GNU/Linux is used solely because of the availability and familiarity. GNU/Linux is free and open-source software with many derivatives.

VIM is a free and open-source text editor available for many operating system, however, popular among Unix-like operating systems such as UNIX, GNU/Linux and macOS.



Git version control system is the most popular software version control system. Git is a free and open-source software. There are other version control systems, however not as much in use nowadays, such as Subversion (SVN), Mercurial, GNU Bazaar etc. Git is used exclusively in this implementation.

Bitbucket is a cloud-based Git repository by Atlassian. There are other popular, if not more, Git repositories as well such as GitHub, Gitlab, opensource.com's Projects etc. This implementation is based on bitbucket and pipeline feature of it is also used.

AWS is a cloud service that provides enterprise cloud services including server instances, storage, networking services, Content Delivery Network Services, Email Services, Repository Services etc. For the purpose of this implementation Elastic Compute Service (EC2) is used as our main production environment, CodeDeploy as Continuous Deployment part of the pipeline and S3 for the storage of artifacts (artifacts are objects which are created as part of the pipeline). An Amazon EC2 instance, where EC stands for Elastic Compute, is a virtual computing environment which can be used as a virtual server and can be scaled resource wise according to the need.

There are alternatives to AWS as well as such as Microsoft Azure (also Azure Cloud Platform), Google Cloud Platform (GCP), Alibaba Cloud, IBM Cloud etc. Among these services, the most popular ones now of this writing are AWS, Azure and GCP followed by IBM cloud. Since cloud services industry is booming now, there are new offerings coming up all the time and the popularity might change soon in the future. Services provided by AWS are generally available in GCP and Azure as well. Azure DevOps from Azure cloud is also quite popular and GCP is credited for invention of Kubernetes.

These cloud service providers, including AWS, also provide the complete pipeline solution including the code repository, CI tools, Delivery Tools, Deployment Tools, Test Tools and Container Services etc. which often are very tightly integrated and setting up a CI/CD pipeline can be done following their instruction documentations. Some of these cloud services also provide training and certifications regarding DevOps practices using their tools and technology, not to mention general usage of their platform. These are often quite useful for companies and the industry trend shows that large organisations often tend to invest in such services and use the tools offered by them.

Docker is a virtualisation technology that can be used to generate an environment based on the Operating System level virtualisation. For this implementation, the software is deployed in containerised environment inside the docker platform in the EC2 instance.

Docker also has alternatives, as mentioned in the Containerised Deployments section above, and there are some utility tools which can leverage the Docker's container technology without using its interface as well. However, Docker is the most popular technology for containers as of now.

Other minor tools and technology used is explained later contextually as they appear.

### **3.2. Process Outline**

The following steps are decided in order to create the necessary components of this implementation:

- Creating a simple software project locally on the computer.
- Testing and selecting a cloud software repository for the project: Bitbucket is selected.
- Selecting a cloud service provider and a virtual machine instance for software deployment: AWS and EC2 are selected.
- Creating an initial version of the CI/CD Pipeline connecting the local code, the repository, and the cloud service provider.
- Using a testing framework and write tests for the software: Pytest is selected as the testing framework.
- Re-implementing the pipeline with the integrated testing.
- Configuring the cloud services needed for the pipeline such as artifact repository, deployment service etc: S3 and CodeDeploy are configured.
- Adding a containerization capability to the pipeline.
- Adding the containerization capability to the cloud virtual machine instance.
- Using the pipeline to run automated tests and deployments in the cloud.
- Deploying using containerized environment within the cloud virtual machine.
- Testing the software as a user.

The following architecture is constructed implementing these steps and the Implementation details are explained in the following Implementation sections.

### 3.3. The Architecture

The following diagram illustrates the architecture of the pipeline:

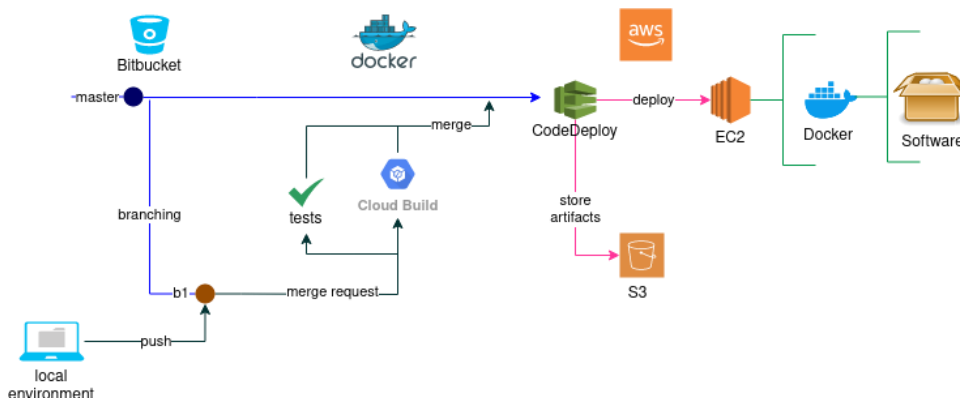


Figure 7: Architecture of the Implemented CI/CD Pipeline

As illustrated in the figure 7 above, the entry point of the pipeline is the local environment which is connected to the bitbucket git repository. The repository contains a master branch and other branches. A branch b1 is denoted in the picture as the working branch where the code locally produced are pushed into.

Depending on the requirement or convention followed a merge request can be made to the master branch and this triggers the Pipeline. As the pipeline is triggered, predefined sets of tests are run, and all tests must pass. After the tests are run, a build process is triggered and further integration tests and building process is carried out inside a temporarily spawned container.

When all the tests and build tasks are successfully completed, a merge is made. Completion of this step further triggers the CodeDeploy service which pulls the code runs integration tests and builds the software. Finally, the artifacts generated are stored in the S3 storage and the software is deployed into the production inside a dockerised environment in EC2 instance.

### 3.4. Implementation Details

As outlined in the process outline above the implementation is carried out in the following way:

### 3.4.1. Development Environment

There is not strict rules or guidelines how the Development Environment should be configured and there are countless possibilities. In this example, simplicity and ease of duplication is given priority rather than establishing or following conventions.

First a directory named CICD is created inside Project directory. The CICD directory contains two directories inside it, **Credentials** where all the credentials such as ssh keys and user info are stored and **project\_root** where all the application and configuration files exist.

The **project\_root** directory contains **bitbucket-pipelines.yml** file, **README.md** file and **app** directory as shown in the figure below. Our main pipeline configuration file is **bitbucket-pipelines.yml** and **app** is the directory where our application exists. The directory **app** is renamed to **apps** later when the pipeline is developed further to support multiple applications. The file **README.md** is where information and instruction for the users of the pipeline is written.

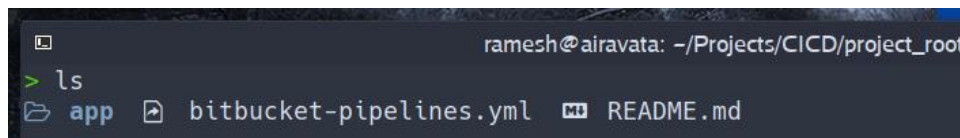


Figure 8: The main working directory in the local environment.

The pipeline configuration file, **bitbucket-pipelines.yml** is written with the initial configuration (Appendix 1) that has three steps, **Build**, **Upload to S3** and **Deploy with CodeDeploy**.

Finally, the local git repository is initialised by running command:

```
git init
```

This command initialises the whole directory under git version control system.

### 3.4.2. Git Repository and the CICD Pipeline

Since Bitbucket repository is selected for the project's git repository, a new project called **Thesis** is created in the bitbucket repository and an empty repository called **project\_root** is created. The project can be called anything, and it is not needed to call the repository **project\_root** but it is done to keep things simple.

The repository URL (shown as `$REPOSITORY_URL` below) of the new repository (Bitbucket) is copied and the new repository is connected to the local repository using the git command:

```
git remote add origin $REPOSITORY_URL
```

Finally, the content of the local `project_root` directory is pushed to the remote `project_root` (bitbucket) using the following git commands:

- Add all files for git to track : `git add .`
- Commit the files : `git commit -m 'Initial Bitbucket Pipeline configuration'`
- Push the files : `git push -u origin -all`

Lastly, Bitbucket automatically recognises the configuration file when the pipeline feature is enabled in the repository settings. The pipeline gets triggered at this point and fails because the production environment which is already a part of the pipeline is not setup yet.

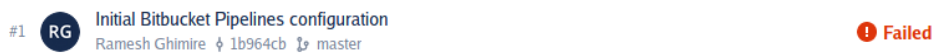


Figure 9: Pipeline failing

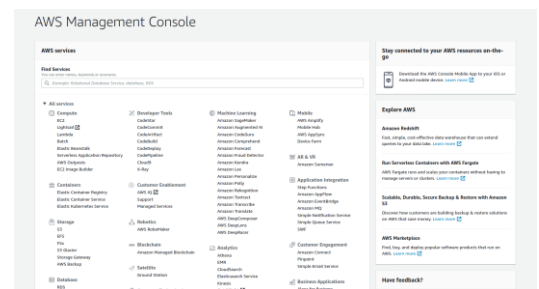
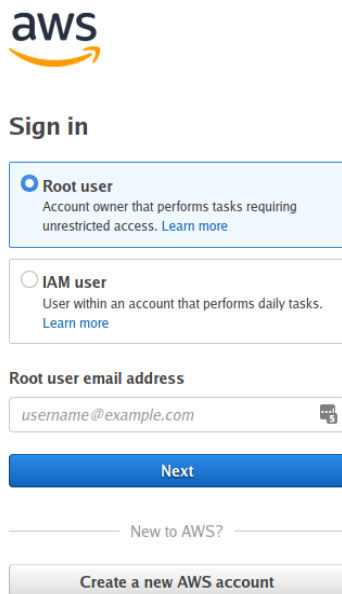
The next step is to setup the Production environment and connect the missing link to the pipeline.

### 3.4.3. Production Environment

This is probably the most important piece of the puzzle. The earlier two steps are more or less familiar with developers and development teams even if they have never used a CI/CD pipeline (exception can be the pipeline feature of the Bitbucket repository) because version control technology and different environment set up are common practices of software development.

The configuration starts from spinning up (allocating and starting) an EC2 instance in AWS cloud. To do so, AWS account is first created and logged in to the console using the

root username and password. A console or AWS Management Console is the UI that allows to start using different services offered by the AWS cloud.



Figures 10 (left) and Figure 11 (right): AWS Sign in page and AWS Management Console

Some resources are created in the production environment for use such as the deployment server, services which handle different parts of the deployment process and storages for artifacts. These resources are configured in a way that these can function autonomously with defined set of rules. The details of creating and configuring these are explained in the next sections.

### 3.4.4. IAM instance profile Configuration

After successfully logged in, an IAM profile and EC2 instance is created. IAM instance profile is required to work with CodeDeploy, which is the deployer service, and EC2 instance where the application gets deployed. During the configuration of the IAM instance profile it is required to set up a policy for the profile with sufficient privileges. The policy is configured using a JSON file and a sample policy looks like:

```

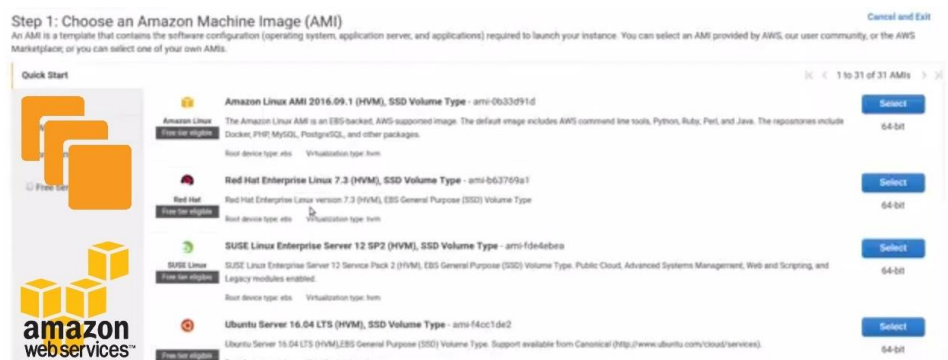
{"Version": "2012-10-17",
 "Statement": [{"Action": ["s3:Get*", "s3:List*"],
 "Effect": "Allow",
 "Resource": "*"}]
}

```

Where Resources must contain ARN of Amazon S3 buckets your Amazon EC2 instances must access. During the creation of the IAM instance profile, a role also must be attached to the profile where the role defines the use case, here EC2. After the policy is attached to the instance, the IAM instance is then attached to the EC2 instance during configuration. The IAM instance profile is used by CodeDeploy and therefore it needs to be configured to have a programmatic access to resources allocated. During the process of creating the IAM instance profile, the user's (instance profile) credentials are also created. These credentials are used in the pipeline setup as the environment variables.

### 3.4.5. EC2 Instance Creation

When creating the EC2 instance, there are many choices of images to select from such as Amazon Linux, Red Hat Enterprise Linux, Ubuntu etc., and most of them work just fine with the set up. However, for this implementation an Amazon Linux AMI is chosen.



Figures 12: Choosing AMI for EC2 instance.

For simple use cases such as this implementation and small web projects, a free tier plan can be sufficient. Hence, a free tier plan is chosen in the next screen following the previous process when creating an EC2 instance as shown in Figure 13 below. In the figure, the t2 micro plan, which is the free tier plan, is highlighted with red and additionally marked with number 1 and the button that takes to the next step, which is the configuration step, is also highlighted and marked with number 2:

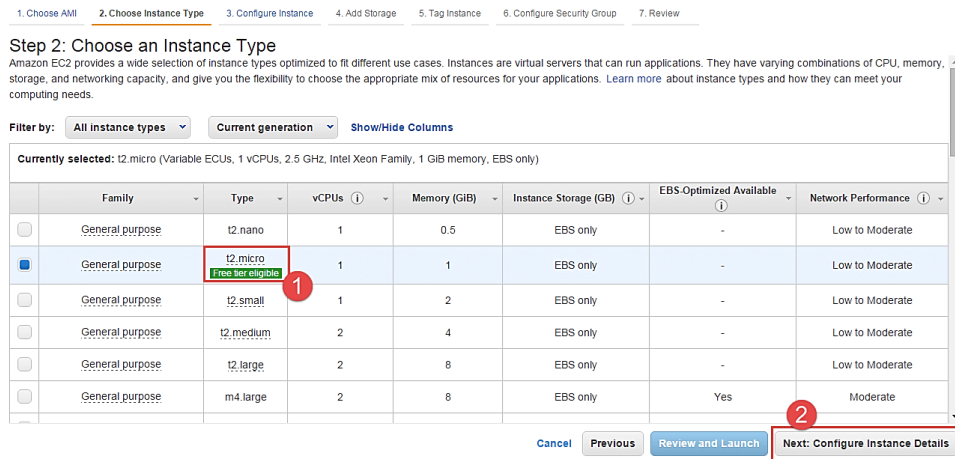


Figure 13: Choosing a free tier plan for EC2 instance.

Following along the configuration steps as guided by the UI creates an EC2 instance. One specific step that must not be avoided in the creation process is to attach the IAM role for CodeDeploy on the EC2 instance. The IAM role must have a policy attached to access S3 storage. EC2 instance thus created needs to be accessed later for configuration hence the SSH key is downloaded and stored in the secure location for using later during the configuration (later explained in section 3.4.8).

### 3.4.6. S3 Storage Configuration

As S3 storage with versioning enabled is created to be used as the storage for artifacts. Since, this storage does not need to be accessed publicly and only be accessed programmatically by the CodeDeploy service, it is configured to reflect these properties. During the creation of the S3 storage it is very important to create it in the same region where the EC2 instance is created. For this implementation EU-North-1 (Stockholm) is the chosen region.

### 3.4.7. CodeDeploy Service Configuration

A deployer application is created under the CodeDeploy service for the deployment tasks. The deployer service needs the deployment group set up for the deployments. The deployer configuration also needs other certain options set up such as the compute platform, in this case EC2, deployment type and the environment configuration. This is a very well guided process that enables connecting the deployer with EC2 instance.



### 3.4.8. Configuring EC2 instance as production server

A secure connection is made from the local terminal to the EC2 instance using the SSH key. After the successful connection, the first step that it takes is the updating of the system. This is not a required step, but it is recommended as a good practice.

After the system update, a few utility software packages are installed because these software packages are needed by the code deploy agent that is installed later. These software packages are: Ruby, a programming language, and wget a GNU software utility that is widely used to retrieve files via HTTP, HTTPS, FTP and FTPS protocols.

These steps are carried out with issuing the following commands:

```
sudo yum update
sudo yum install ruby
sudo yum install wget
cd /home/ec2-user
wget https://artifacts-storage-itp.s3.eu-north-1.amazonaws.com/latest/install
```

Where artifacts-storage-itp is the name of the S3 storage unit, also called bucket, and eu-north-1 is the region name where the resources including EC2 and S3 are configured. After the last line is executed, the file that is needed for installing the code deploy agent is downloaded. An execute permission that is required for this file is given by the following command:

```
chmod +x ./install
```

Finally, the following command will install the code deploy agent application:

```
sudo ./install auto
```

The code deploy agent must be running automatically after the installation, however, to be extra certain that it is running the following commands is executed:

```
sudo service codedeploy-agent start
```

### 3.4.9. Environment Variable Configuration in Bitbucket Pipelines

The next step is to set up the environment variables in the bitbucket pipeline settings. The pipeline settings can be found inside the repository settings option and Repository variables is where the environment variables are configured. Figure: 14 shows the location of the Repository variables in the Bitbucket repository settings and Figure 15: shows the variables that are configured.

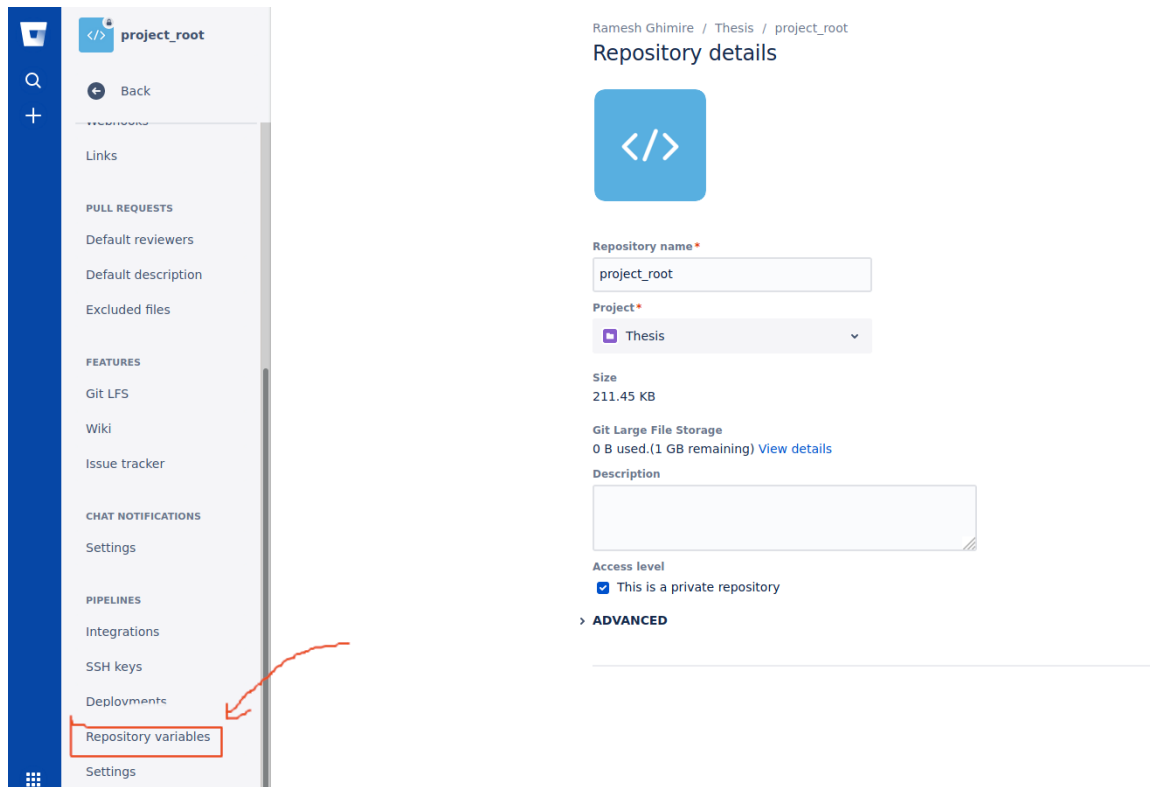


Figure 14: Environment variable settings location for Bitbucket Pipelines.

Clicking the Repository variables option presents the environment variable setup screen where the information about user and its credential to access AWS resources are added. This information must be in the exact format that is received during the resource's setup process within AWS console.

## Repository variables

Environment variables added on the repository level can be accessed by any users with push permissions in the repository. To access a variable, put the \$ symbol in front of its name. For example, access AWS\_SECRET by using \$AWS\_SECRET. [Learn more about repository variables.](#)

Repository variables override variables added on the workspace level. [View workspace variables](#)

If you want the variable to be stored unencrypted and shown in plain text in the logs, unsecure it by unchecking the checkbox.

Name	Value	<input type="checkbox"/> Secured	Add
S3_BUCKET	artifacts-storage-1tp	<input type="checkbox"/>	
AWS_ACCESS_KEY_ID	AKIA4LRL7VIXNLZIWI7R	<input type="checkbox"/>	
AWS_SECRET_ACCESS_KEY	•••••	<input checked="" type="checkbox"/>	
AWS_DEFAULT_REGION	eu-north-1	<input type="checkbox"/>	
APPLICATION_NAME	deployer	<input type="checkbox"/>	
COMMAND	deploy	<input type="checkbox"/>	
DEPLOYMENT_GROUP	deployers	<input type="checkbox"/>	

Figure 15: Setting up environment variables.

As shown in the Figure: 15, the name of the environment variables is in uppercase letters and they correspond the resources in the AWS cloud which are already configured such as the deployer application, deployment group, S3 bucket information, the IAM user profile etc. This information is basically needed for the deployment part of the CICD pipeline which happens in the production environment in the AWS cloud. This concludes the first complete set up of the CICD pipeline and the next step is to add functionalities in the pipeline such as tests, deployment as containerised deployments with Docker and ability to use the pipeline for multiple application.

### 3.4.10. Adding features to the CICD pipeline

To test if the pipeline executes successfully with the configuration thus made an empty file called **emptyfile** is added to the **project\_root** directory which is the working directory for this project and the pipeline executes successfully as shown in the Figure 17. The pipeline is configured to run every time there is a code push on the remote branch of the repository. Figure 16 shows the commands executed in the local development environment to, sequentially, navigate to the working directory, list the existing files, create the **emptyfile**, check the version control status, add the **emptyfile** to the version control system, commit the file and push the file to the remote branch hence triggering the pipeline execution.

```

> cd Projects/CICD/project_root
> l
├─ app
├─ bitbucket-pipelines.yml
└─ README.md
> touch emptyfile
> git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>.." to include in what will be committed)
        emptyfile

nothing added to commit but untracked files present (use "git add" to track)
> git add emptyfile
> git commit -m 'added the emptyfile to test the pipeline execution'
[master a6a137e] added the emptyfile to test the pipeline execution
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 emptyfile
> git push origin master
Password for 'https://goonda@bitbucket.org':
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 304 bytes | 304.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
To https://bitbucket.org/goonda/project_root.git
   6e0c9c9..a6a137e  master -> master

```

Figure 16: Triggering the pipeline execution by pushing an empty file to the remote repository.





Pipeline	Status	Started	Duration
#10  added the emptyfile to test the pipeline execution Ramesh Ghimire  a6a137e  master	 Successful	4 minutes ago	44 sec

Figure 17: Successful pipeline execution

The pipeline is now configured and works. It can be used as it is, but it is limited to one application and there are no testing and containerised deployments. To make it more featureful and usable, the following additional improvements are necessary:

- Restructuring the pipeline to support multiple applications.
- Adding a pre-build test suite to the pipeline.
- Deploying in docker containers.

The next sections explain how these can be built into the pipeline.

### 3.4.11. Pipeline optimisation

Currently pipeline just gets triggered whenever any change is pushed to the remote repository, then it builds the software and deploys the application to the production environment. This is not very practical since it forces developers to test the software locally before pushing to the remote repository, because of which the two repositories may never in synchronisation while a work is incomplete. This can be handled simply by enhancing the pipeline setup a little and make it only get triggered when intended, usually when we want to test and deploy the software.

Instead of triggering the pipeline on every push, we can modify it so that it triggers on every pull request to the master branch of the remote repository. This way any incomplete development work can be carried out in related branches and when the software is ready for testing and deployment, a pull request to the master branch can be created and the branches can optionally be merged. The pipeline is then triggered only on such events. Also, instead of one application, the pipeline must support multiple applications.

The current bitbucket-pipelines.yml file looks like the following:

```
image: atlassian/default-image:2
pipelines:
  default:
    - step:
      name: Build
      script:
        - cd app && zip -r ../myapp.zip *
      artifacts:
        - myapp.zip
    - step:
      name: Upload to S3
      services:
        - docker
      script:
        - pipe: atlassian/aws-code-deploy:0.2.10
      variables:
        AWS_ACCESS_KEY_ID: ${AWS_ACCESS_KEY_ID}
        AWS_SECRET_ACCESS_KEY: ${AWS_SECRET_ACCESS_KEY}
        AWS_DEFAULT_REGION: ${AWS_DEFAULT_REGION}
```

```

        COMMAND: 'upload'
        APPLICATION_NAME: ${APPLICATION_NAME}
        ZIP_FILE: 'myapp.zip'
        S3_BUCKET: ${S3_BUCKET}
    - step:
        name: Deploy with CodeDeploy
        deployment: production
        services:
        - docker
        script:
        - pipe: atlassian/aws-code-deploy:0.2.10
        variables:
            AWS_ACCESS_KEY_ID: ${AWS_ACCESS_KEY_ID}
            AWS_SECRET_ACCESS_KEY: ${AWS_SECRET_ACCESS_KEY}
            AWS_DEFAULT_REGION: ${AWS_DEFAULT_REGION}
            COMMAND: 'deploy'
            APPLICATION_NAME: ${APPLICATION_NAME}
            DEPLOYMENT_GROUP: ${DEPLOYMENT_GROUP}
            IGNORE_APPLICATION_STOP_FAILURES: 'true'
            FILE_EXISTS_BEHAVIOR: 'OVERWRITE'
            WAIT: 'true'
            S3_BUCKET: ${S3_BUCKET}

```

First, before defining the steps of the pipeline, the information about which branch this pipeline should run on is defined by simply adding:

```

branches:
  master:

```

The next step is to add a testing suite for the software. Since the intended software in this case are flask applications, pytest framework can be integrated in the pipeline to run the tests using it. Also, to support multiple applications, the app directory is renamed **apps**. Now the first 11 lines of the bitbucket-pipelines.yml file looks like:

```

image: atlassian/default-image:2
pipelines:

```

```

branches:
  master:
    - step:
      name: Testing
      image: python:3.5.1
      script:
        - apt-get update
        - pip install -U pytest
        - pytest apps/*/tests/*.py

```

This also enforces all the tests to be written inside the **tests** directory under for and long with every software. All the python files inside the **tests** directory is executed with **pytest** command. The test suite integration part is now complete.

Now the next step of the pipeline, which is the build step will not happen if all the tests are not passed. But if the pipeline is run in the current state it will fail nevertheless because our working directory structure does not reflect the changes. So, the following changes must be made to our working directory:

- The **app** directory is renamed to **apps**
- The **scripts** directory is moved to the **project\_root** directory.
- The **appspec.yml** is moved to the **project\_root** directory.
- A **dockerise.sh** file is created inside the **scripts** directory with the following content (The destination directory for the deployment is **/opt/app** directory, not to be confused with local **apps** directory):

```

#!/bin/bash
cd /opt/app
sudo docker-compose up -d --build

```

- The **appspec.yml** is modified with the added containerisation step (dockerise):

```

version: 0.0
os: linux
files:
  - source: /apps
    destination: /opt/app
hooks:
  AfterInstall:
    - location: scripts/dockerise.sh
      timeout: 300

```

```

ApplicationStart:
  - location: scripts/start_server.sh
    timeout: 300
    runas: root
ApplicationStop:
  - location: scripts/stop_server.sh
    timeout: 300
    runas: root

```

- The dockerise.sh will be looking for docker configuration files needed for containerisation so the apps directory must have a docker-compose.yml file that looks like the following:

```

version: '3'
services:
  testapp:
    build: ./testapp
    network_mode: bridge

```

The **testapp** is the name of the software in the code above. All the software in the **apps** directory must be defined in this file as services. Additionally, a file named Dockerfile must be present inside the individual software directory. The Dockerfile looks like:

```

# python version to use
FROM python:3.7.6
# set up the work directory
WORKDIR /app
#copy the required files to the app folder
COPY main.py ./
CMD ["python3", "main.py"]

```

On the code above, main.py is the main file of the flask application. Details on docker configuration files and related technology is not in the scope of this literature. Figure 18 below shows the final directory structure after all the modifications:

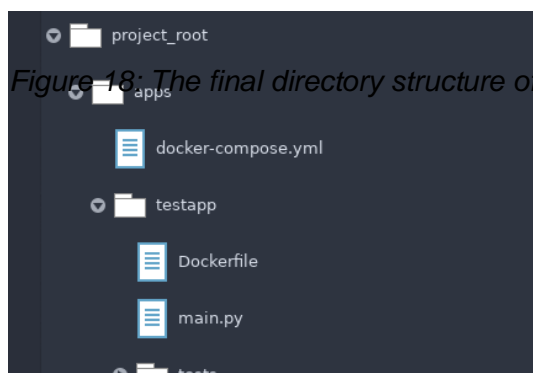


Figure 18: The final directory structure of the implemented CICD Pipeline.



This concludes the implementation of an automated CICD pipeline that is capable supporting unlimited number of software projects, running pre-build tests, running builds, storing artifacts to S3, moving files to the production environment, dockerising and deploying in containers.

## 4. Conclusion

The concept of this thesis work originated because of an observation made by the researcher in his circle of contacts in the software development community. The researcher has seen the growing trend of DevOps practices and commercial CICD offerings. The researcher has also seen that there is a fear of unknown amongst many software developers who have not yet made a leap towards the DevOps mentality. As a result, a tremendously beneficial practice such as adoption of automated CICD pipelines is not adopted as much as it should be. One of the goals of this thesis, along with others as explained in detail in earlier chapters, was also to demonstrate that this leap is not so hard to make. The researcher has also gained more in-depth insight on the topic during this research.

The implementation explained in the previous chapters is a complete CICD pipeline which requires a minimal setup, integrates well with existing tools and technology of the most common software development practices, supports multiple software or a software with multiple components, runs tests automatically before building, stores different versions in the artifact storage, containerises the software and deploys in the defined production environment.

From the theoretical background established in this thesis and the implementation of the idea as a proof of concept, it is now fairly clear that any software developer or a development team can simply create a minimalistic, cost-effective and efficient Continuous Integration and Continuous Delivery / Deployment pipelines with existing knowledge and skills. As demonstrated by the implementation above the CICD pipeline creation and implementation is not so difficult and provides with a lot of benefits. Practice of setting up and using such automated pipelines is compatible with Agile way of working and DevOps mentality of software development as discussed in the earlier chapters.

However, since all software development work is different and there are differences in preferences of tools and technology used among developers themselves, the architecture of a potential CICD pipeline must be carefully thought of. Also, a CICD pipeline can be simpler, if the project does not require all the aspect of the pipeline, or quite complex if the project requires many steps and iterations and involved a lot of intermediary processes. Often, a simple CICD setup can sufficiently work as a starting point for a beginning of the software development project, and as the team grows and / or the development work advances forward, more components can easily be added and the pipeline can be modified.

Some of the components of the CICD pipeline such as deployment in a containerised environment may not even be necessary for a single software development project. Also, all the tools used have their alternatives which may have certain benefits over the tools used in this implementation. Specifically, our choice of version control system GIT can be easily replaced by SVN (However not so popular), Bitbucket can easily be replaced by GitHub, Gitlab or other cloud repository providers, AWS can easily be replaced by Azure, GCP or some other etc. The choice is, and always should be, upon the developer, depending on the requirement specifications, to decide what kind of tools and technology should be used.

Working with DevOps mentality, adopting Agile way of working and using CICD pipelines, hence, has clear benefits and improves the quality of not only the software but the software development process as well. It is financially beneficial for companies to adopt these practices and encourage their developers to use automated CICD pipelines. It has also been clear from this research that the automated CICD pipelines can reduce the risk of project failure, create more feedback loop enhancing the communication and keep increase the overall efficiency of software development teams.

## **5. Afterword and Recommendations**

For aspiring software developers, development teams or any enthusiasts who want to start using CICD pipelines to deploy software in the cloud, the researcher would like to make some recommendations. If a company or team has not adopted a CICD pipeline for their projects yet, it can be done in various ways. For the purpose of simplicity these different ways are regarded as *strategies* here.

### **5.1. CICD Pipeline Adoption Strategies**

First, the adoption strategy depends on the project or the company commissioning or running the project. The strategy for adopting a CICD pipelines can be, but not limited to, for example:

- Bespoke Solution from a Solution Provider
- Formation of a DevOps department
- Temporarily hiring an expert for the set-up
- Inhouse set up from existing developers

Which strategy to adopt depends on a few factors such as the size of the company and the budget allocation, size and complexity of the software project, expertise in the team, requirement specification, complexity of the pipeline etc.

#### **5.1.1. Bespoke Solution**

This strategy may be suitable for medium to large size companies which may require complex solution with many moving parts, the setting up such solution would take time and resources and they lack inhouse expertise. Often these companies may not want to go through recruitment process to hire expertise for the CICD setup because the recruitment process is time consuming and costly.

This is not a very preferable solution because this solution will work but there are some caveats such as the developers will need trainings and the maintenance and support from the solution provider may be required for a while. However, it is the researcher's recommendation that the companies adopting this strategy should look for establishing their own DevOps department in the long run.

#### **5.1.2. DevOps Department**

A dedicated DevOps department maybe suitable for a medium to large size companies which can allocate a budget and have corporate will for such a set-up and restructuring. This maybe a prescribed strategy in many situations such as when developers are involved in multiple projects and there are multiple number of software in different stages at a given time. Since, restructuring of companies (setting up new departments) takes time, this solution is only recommended for companies which do have such time for implementation of CICD pipelines.

#### **5.1.3. Temporary Hire**

Temporarily hiring an expert is also a good strategy for companies which may not have budget for a persistent department but want to use such practices. A temporary DevOps engineer can then set up the CICD system, train existing developers and help the adopting company transition towards DevOps practices.

#### **5.1.4. Inhouse Setup**

This is a prescribed strategy for small companies and teams as this strategy involves the software developers own learning which in turn enhances the capabilities of the adopting company or team. This strategy is suitable for simple setup requirements and companies with a few software development projects going on at a given time. Adoption of this strategy also means learning and setting up the CICD pipeline and practicing DevOps related practices for individual developers.

## 5.2. Commercial and Free CICD Pipeline Providers

There are magnitudes of possibilities for CICD pipelines with so many different commercial, as well as free, CICD solutions available. Often, these solutions are specialised in special stage or set of tasks. There are also suites of applications provided by cloud service providers such as Amazon AWS, Microsoft Azure, Google Cloud Platform etc. An example of such suites is Azure DevOps Solution (using GitHub, Jenkins, Azure Container and Azure Web App), AWS CICD Pipeline (using CodeCommit, CodeDeploy, Beanstalk, ECS/ECR) etc.

There are also individual CI solutions such as Travis, Jenkins, GitLab, BitBucket Pipelines, CircleCI, Bamboo etc which are specialised in the integration and testing part of the pipeline. However, most of these can be configured to use continuous Delivery and Continuous Deployment and made into a full-fledged pipeline. What kind of tools to use and what kind of set-up to perform is and always should be developers' choice based on the requirements of projects and business interests.

## References

- Atlassian Corporation Plc, CITE Research. (2020). *2020 DevOps Trends Survey*. Atlassian Corporation Plc.
- Clive Longbottom. (2017). *Evolution of cloud computing - how to plan for change*. Bcs Learning & Development Lim.
- Docker Inc. (2013). *What is a Container?* Docker. <https://www.docker.com/resources/what-container>
- Duvall, P. M., Matyas, S., & Glover, A. (2007). *Continuous integration : improving software quality and reducing risk*. Addison-Wesley.
- Hodgson, P. (2020). *Continuous Delivery in the Wild*. O'Reilly Media, Inc.
- Ingeno, J. (2018). *Softward architect's handbook : become a successful software architect by implementing effective architecture concepts*. Packt Publishing Ltd.
- Jez Humble, & Farley, D. (2011). *Continuous delivery*. Addison-Wesley.
- Jordan, J. M. (2012). *Information, technology, and innovation : resources for growth in a connected world*. Wiley.
- Kim, G., Debois, P., Willis, J., Humble, J., & Allspaw, J. (2017). *The DevOps handbook : how to create world-class agility, reliability, and security in technology organizations*. It Revolution Press, Llc.
- Mariot Tsitoara. (2020). *BEGINNING GIT AND GITHUB : a comprehensive guide to version control, project management, and... teamwork for the new developer*. Apress.
- Nickoloff, J., Kuenzli, S., & Fisher, B. (2019). *Docker in action*. Manning Publications Co.
- Onur Yilmaz. (2018). *CLOUD-NATIVE CONTINUOUS INTEGRATION AND DELIVERY : discover how you can efficiently build, deploy ... and test your own cloud-native applications*. Packt Publishing.
- Portnoy, M. (2012). *Virtualization essentials*. Wiley/Sybex.

Sander Rossel. (2017). *Continuous Integration, Delivery And Deployment*. Packt Publishing Limited.

Schenker, G. N. (2020). *Learn Docker : fundamentals of Docker 19.x : build, test, ship, and run containers with Docker and Kubernetes*. Packt Publishing.

Stack Exchange, Inc. (2020). Stackoverflow Developer Survey 2020. In *Stackoverflow*.  
<https://insights.stackoverflow.com/survey/2020#overview>

Stellman, A., & Greene, J. (2017). *Head First Agile : a brain-friendly guide to Agile and the PMI-ACP certification*. Beijing ; Boston ; Farnham ; Sebastopol ; Tokyo O'reilly September.

## Appendices

### Appendix 1. Software codes snippets

#### I. Policy for IAM Role

```
{ "Version": "2012-10-17", "Statement": [ { "Effect": "Allow", "Action": [ "s3:Get*", "s3:List*" ], "Resource": [ "arn:aws:s3:::artifacts-storage-itp/*", "arn:aws:s3:::aws-codedeploy-eu-north-1/*" ] } ] }
```

#### II. Optimised and cleaned up Bitbucket-pipelines.yml file

```
pipelines:
  branches:
    master:
      - step:
          name: Testing
          image: python:3.5.1
          script:
            - apt-get update
            - pip install -U pytest
            - pytest apps/*/tests/*.py
      - step:
          name: Deployment
          image: python:3.5.1
          script:
            - apt-get update
            - apt-get install -y zip
            - pip install boto3==1.3.0
            - zip -r /tmp/artifact.zip *
            - python codedeploy_deploy.py
```

#### III. CodeDeploy script (codedeploy\_deploy.py used by the bitbucker-pipelines.yml file)



```

# Copyright 2016 Amazon.com, Inc. or its affiliates. All Rights
Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License").
You may not use this file
# except in compliance with the License. A copy of the License is
located at
#
#     http://aws.amazon.com/apache2.0/
#
# or in the "license" file accompanying this file. This file is
distributed on an "AS IS"
# BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either ex-
press or implied. See the
# License for the specific language governing permissions and li-
mitations under the License.
"""
A BitBucket Builds template for deploying an application revision
to AWS CodeDeploy
narshiva@amazon.com
v1.0.0
"""
from __future__ import print_function
import os
import sys
import json
from time import strftime, sleep
import boto3
from botocore.exceptions import ClientError

VERSION_LABEL = strftime("%Y%m%d%H%M%S")
BUCKET_KEY = os.getenv('APPLICATION_NAME') + '/' + VERSION_LABEL + \
    '-bitbucket_builds.zip'

def upload_to_s3(artifact):
    """
    Uploads an artifact to Amazon S3

```

```

"""
try:
    client = boto3.client('s3')
except ClientError as err:
    print("Failed to create boto3 client.\n" + str(err))
    return False
try:
    client.put_object(
        Body=open(artifact, 'rb'),
        Bucket=os.getenv('S3_BUCKET'),
        Key=BUCKET_KEY
    )
except ClientError as err:
    print("Failed to upload artifact to S3.\n" + str(err))
    return False
except IOError as err:
    print("Failed to access artifact.zip in this directory.\n"
+ str(err))
    return False
return True

def deploy_new_revision():
    """
    Deploy a new application revision to AWS CodeDeploy Deployment
    Group
    """
    try:
        client = boto3.client('codedeploy')
        print("created boto3 client")
    except ClientError as err:
        print("Failed to create boto3 client.\n" + str(err))
        return False

    try:
        response = client.create_deployment(
            applicationName=str(os.getenv('APPLICATION_NAME')),
            deploymentGroupName=str(os.getenv('DEPLO-
YMENT_GROUP_NAME')),

```

```

revision={
    'revisionType': 'S3',
    's3Location': {
        'bucket': os.getenv('S3_BUCKET'),
        'key': BUCKET_KEY,
        'bundleType': 'zip'
    }
},
deploymentConfigName=str(os.getenv('DEPLOYMENT_CONFIG')),
description='New deployment from BitBucket',
ignoreApplicationStopFailures=True
)
print("response obj create successful")
except ClientError as err:
    print("Failed to deploy application revision.\n" +
str(err))
    return False

"""
Wait for deployment to complete
"""
while 1:
    try:
        deploymentResponse = client.get_deployment(
            deploymentId=str(response['deploymentId'])
        )
        deployment_status_detail = "".join([str(k)+"":
"+str(v)+"\n" for k, v in deploymentResponse['deploymentIn-
fo'].items()])
        deploymentStatus=deploymentResponse['deploymentIn-
fo']['status']
        if deploymentStatus == 'Succeeded':
            print ("Deployment Succeeded")
            return True
        elif deploymentStatus == 'Failed':
            print("Deployment Failed\n"+deployment_status_de-
tail)

```

```

        return False
    elif deploymentStatus == 'Stopped':
        print("Deployment stopped")
        return False
    elif (deploymentStatus == 'InProgress') or (deploymentStatus == 'Queued') or (deploymentStatus == 'Created'):
        continue
    except ClientError as err:
        print("Failed to deploy application revision.\n" +
str(err))
        return False
    return True

def main():
    if not upload_to_s3('/tmp/artifact.zip'):
        sys.exit(1)
    if not deploy_new_revision():
        sys.exit(1)

if __name__ == "__main__":
    main()

```

#### IV. Flask Application main file

```

from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/')
def hello_world():
    return {"api_v": 1, "api_type": "Mult_14", "api_description":
"Multiply by 14"}

@app.route('/mul14', methods=['POST'])
def run_analysis():

```

```
assignment_data = request.json
data_to_multiply = assignment_data["parameters"].items()
if data_to_multiply:
    return jsonify({"multiplied": {w: (x * 14) for w, x in
data_to_multiply}})
    else:
        return "No data to multiply"

if __name__ == '__main__':
    app.run(debug=False, port=5010)
```