A Cloud-Based DevOps Toolchain for Efficient Software Development

Ruiyang Ding

Delft University of Technology

A Cloud-Based DevOps Toolchain for Efficient Software Development

Master's Thesis in Computer Science
EIT Digital Master's Programme in Cloud Computing Services
(Special. in track ICT innovation)

Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Ruiyang Ding

12th August 2020

Author
Ruiyang Ding

Title
A Cloud-Based DevOps Toolchain for Efficient Software Development

MSc presentation
31th August 2020

Graduation Committee

Prof dr. ir. D. H. J. Epema (chair)	Delft University of Technology
Mikko Drocan	Eficode Oy
Dr. J.S. Rellermeyer	Delft University of Technology
Prof. dr. M.M. Specht	Delft University of Technology

Abstract

In the traditional software development life cycle, development and operation are divided into different departments. The conflict between departments and, besides, the lack of automation usually leads to low software development efficiency and slow software delivery. Thus, the concept of DevOps is introduced which combines different departments and automates the process to make software delivery faster and easier. The DevOps toolchain is one important component for adapting DevOps. On the other hand, the adaptation of cloud technology, especially serverless computing makes it tempting for us to investigate what benefit serverless computing brings to the DevOps toolchain. In the first research question we examine the benefits that AWS serverless platforms bring to DevOps toolchain. To answer this research question, we (1) develop a DevOps toolchain hosted in Amazon Web Services (AWS), in addition, using the serverless computing service; (2) Examine what does each serverless computing service brings to the DevOps toolchain, examine how does the performance of the DevOps toolchain changes with or without using serverless computing service.

Our research shows that a part of the serverless computing service could reduce the cost, operation effort, and improves performance with the help of enabling parallel execution. The experiment shows that, in contrast to a toolchain hosted in a traditional cloud server vs the toolchain that was developed by us using serverless computing service could reduce the total runtime of parallel execution up to 65%. In the second research question, we focus on the integrated toolchain build with AWS DevOps tools from AWS serverless platform. We (3) build a demo integrated DevOps toolchain with AWS DevOps tools and (4) compare the integrated toolchain with the non-integrated toolchain we build in (1). We find out that the integrated toolchain significantly reduces the development time by providing an out-of-box solution for DevOps toolchain. In addition, the better integration with underlying cloud infrastructure provides more functionality such as global monitoring and blue/green deployment. However, we also find out from the experiment that the performance of the integrated toolchain is lower due to the limitation of resources which also come with a high cost.

Preface

Helping customers transform their software development practices to DevOps is one of the main business activities of Eficode, and DevOps toolchain is an essential part of the transformation process. On the other hand, as an advanced partner of multiple cloud providers, Eficode is interested in what cloud technologies could bring to the DevOps toolchain. In this thesis project, I focus on the AWS serverless platform[1] in Amazon Web Services and discuss what change, especially benefit can it bring to a DevOps toolchain.

The process of carrying out this project is not an easy task; setting up the cloud infrastructure requires an enormous number of operational and configuration tasks. The vast but unregulated plugin eco-system of Jenkins also leads to problems like lack of plugin documentation, dependency hell and unstable plugin such as plugin for using ECS as Jenkins agent. There does not exist that many previous researches about the serverless within DevOps toolchain. Moreover, as a student with a software development background, the lack of prior experiences in the related field means much study is needed before I can start the project. All these tedious and unexpected tasks above, plus the tight thesis schedule did make me frustrated in the middle phase of this project. Despite all of these, I managed to achieve the defined goal by answering all the research questions and implement the demo. Through this project, I familiarized with different exciting tools for cloud and DevOps; it opens up a whole new area for me, and will help me with my future career in Eficode. I hope the result will give Eficode more insight on the capability that AWS serverless platform could bring for the DevOps toolchain.

Now I'm at the end of this two year's journey, and I feel grateful that I made this life-changing decision to come and study in Europe. I have to say, it was not an easy decision to quit the ongoing research master's study back in Shanghai, China and start all over again in a brand new environment. The two year's study was a journey full of struggle – in both financial and study. However, what it brings is more than what I expected: international experiences by EIT Digital, great friends from different countries, two intern/work experiences, and precious knowledge that combines business and technologies. Most importantly, I could study the topic that I'm interested in and start a career within this field, which is what I was not able to

[1] https://aws.amazon.com/serverless/

do back in my previous research master's study.

I sincerely thank my supervisor Mikko Drocan from Eficode, Prof.dr.ir. D.H.J. Epema from TU Delft for their guidance and support in writing during this unprecedented time. Thanks for Eficode for giving me this precious opportunity and sponsoring this thesis project. Thanks for the EIT Digital master school for the scholarship which made it possible for me to finish the two-year's master's study. Lastly, special thanks to the thesis support team in Eficode for getting the weekly support on my thesis writing, and Eficode IT for providing the cloud sandbox environment.

Ruiyang Ding

Helsinki, Finland
12th August 2020

Contents

Chapter 1

Introduction

The Agile Manifesto [1] drafted by Kent Beck et al. in 2001, created the Agile software development method. Since then, this software development method has drawn attention to the industry. Agile has become a leading standard for the software development industry, with multiple further enhancements aiming to tackle certain business-specific challenges. The Agile method advocates the shorter development iteration, continuous development of software and continuous delivery of the software to the customer. The goal of Agile is to satisfy the customer with early and continuous delivery of the software [1]. The Agile, which aims at the improvement of the process within the software development team and the communication between the development team and customers [2] improves software development and makes it more iterative and thus faster. However, it does not emphasize the cooperation and communication between the development team and other teams. In real life, the conflict and lack of communication between the development team and operation teams usually becomes the barrier for efficient development of a software project [3].

Hence, in answer to how to solve the gaps and flaws when applying Agile into real-life software development, the concept of DevOps emerged. The term "DevOps" is created by Patrick Debois in 2009 [4], after he saw the presentation "10 deployments per day" by John Allspaw and Paul Hammond. While Agile fills the gap between software development and business requirement from the customer, the DevOps eliminates the gap between the development team and the operation team [5]. By eliminates the barrier we mentioned in the last paragraph, DevOps further enhances and smoothers software delivery. In conclusion, DevOps means a combination of practices and culture which aim to combine separate departments(software development, quality assurance and the operation and others) in the same team, in order fasten the software delivery, maximising delivered without risking high software quality [6][7].

In software engineering, the toolchain is a set of tools which are integrated for performing a specific objective. DevOps toolchain is the integration between tools that specialised in different aspects of the DevOps ecosystem, which support and coordinate the DevOps practices. The DevOps toolchain helps organizations in creating and maintain an efficient software delivery pipeline, automate the development process [8] which are the keys aspects in DevOps. On the other hand, DevOps relies strongly on tools. There exist specialised tools which helps teams adopt different DevOps practices [9].

Traditionally, a DevOps toolchain is to have individual tools which are standalone and from different vendors. The tools usually are on-premise. However, can also be deployed on cloud virtual machines. In this report we define this type of toolchain as **non-integrated toolchain**.

At the same period that the tools for DevOps emerged and developed, the cloud technologies also developed rapidly. This led to the emigrations of Serverless Computing. The Serverless Computing is a modern cloud computing model in which everything is build and executed in the applications running in the cloud environments without thinking about physical servers [10]. It also allows developers to build application with less overhead [10] and more flexibility by eliminating infrastructure management tasks [11]. With serverless computing technologies, many new cloud technologies emerged, which gives developers an alternative way from traditional cloud servers or cloud virtual machines. For example: Functional computing allows the application to be divided by functions and designed under the event-driving paradigm without managing the hardware infrastructures. The on-demand nature of the serverless computing could be used to deploy certain event-based component of a DevOps toolchain, such as post deploy testing and logging. Managed scalable container services in the cloud enable the user to run the container-based application directly on the cloud, which enables the toolchain scalability. DevOps tools as a service [12] allow the cloud provider to deliver DevOps tools directly on its cloud platform.

Helping the customer in there DevOps transformation is one of the main business activities of Eficode, the company which I am writing my thesis. TThe transformation is enabled, for example, by defining, developing and maintaining a DevOps toolchain at the customer. As mentioned in the last paragraph, the new changes brought by cloud may further improve the performance and lower the cost of DevOps toolchain development – both money and time. As part of my thesis work at Eficode, I will investigate how serverless computing enhances the DevOps toolchain,

1.1 Problem Statement

As per the last paragraph, serverless computing could brings enhancements to the DevOps toolchain. Currently, there are several cloud providers that providers cloud services using serverless computing technologies. Among them, Amazon Web Services (AWS)[1] has the largest market share and is the first cloud provider which provides serverless computing services. According to the report from Gartner [13],

[1]https://aws.amazon.com/

the market share of AWS was 47.8% in the year 2018, which makes it the largest cloud provider in the world.

Nowadays, the serverless computing services in AWS has already been expanded to a set of fully managed services called "AWS serverless platform" [2]. This platform includes new AWS cloud products that leverage the serverless computing technologies. These products include for instance, AMS Lambda[3] for function computing and AWS Fargate[4] for managed container services. AWS also gains the most popularity among the developers that use serverless technologies. The most recent survey report [14] from Cloud Native Computing Foundation (CNCF) shows that 51% of serverless users are using AWS Lambda, while 68% of developers who are not using Kubernetes are using AWS ECS to hosting their containers. As the Advanced AWS partner, AWS is being used as the main cloud providers in the customer projects by Eficode. Furthermore, the company keeps looking for ways to leverage serverless computing services in AWS in order to produce cost efficient solutions for the customers.

However, despite the serverless computing is be extensively used, and an enormous number of research papers about the use-cases or benefits of serverless in data analysis [15], for container-based microservices [16], or for IoT applications [17] [18], the benefits of serverless within DevOps has not yet been discussed. There is a paper [19] and a book [20] about DevOps toolchain for serverless applications. Nervertheless, there is still lack of research on how serverless helps DevOps toolchain itself. Thus, our first research question is to fill the gap by answering this question.

The second area we need to investigate in our project is the integrated DevOps toolchain that is powered by serverless DevOps tooling in AWS.

The **integrated DevOps toolchain** is delivered as a cloud-based single platform that allows development teams to start using DevOps toolchain without the challenge of having to choose, integrate, learn, and maintain a multitude of tools. In other words, the cloud based-integrated DevOps toolchain is to offer DevOps toolchain as a service. In AWS, this is offered by AWS CodePipeline (as the platform) and Several serverless tools that integrated with CodePipeline.

This integrated toolchain is one of the new changes that serverless computing brings, but it also leaves a question to the development team who is trying to build DevOps toolchain in AWS: which kind of toolchain should they select? Should they stick on the previous non-integrated toolchain or embracing the integrated one? The integrated DevOps toolchain provides an out-of-box integrated solution for the whole DevOps lifecycle, which is tempting. However, apart from the advertisement from the vendors of these "DevOps" platforms, we still lack third party researches about the comparisons between these two.

Based on the above, the research questions could be summarised as below:

[2]https://aws.amazon.com/serverless/

[3]https://aws.amazon.com/lambda/

[4]https://aws.amazon.com/fargate/

RQ1: *How can serverless computing services in Amazon Web Services enhance the DevOps toolchain?*
RQ2: *How does the integrated toolchain build with AWS DevOps Tools compare with the traditional non-integrated toolchain?*

1.2 Research Approach

To answer the RQ 1, we first investigate the current serverless offering in Amazon Web Services (AWS), which is one of the cloud services mainly used in Eficode. In chapter 3, we first introduce serverless computing services that we will use in our implementation. We also analysis what the role of each service within a DevOps toolchain.

To answer RQ1 and RQ2, we implement both traditional non-integrated and integrated toolchain based on the DevOps practices and tools used by Eficode. In the design and implementation of the toolchain, we focus on the following DevOps practices: Version Control, Configuration Management, Continuous Delivery and Monitoring. The goal of the implementation is: First, validate the availability of using AWS serverless computing services in the traditional non-integrated toolchain. Thus, in the process of develop and deploy the toolchain, we could already partly answer the RQ1 by answer how the serverless computing services be used in our DevOps toolchain. Second, the implementation served as the environment for experiments in Chapter 5, which could answer two RQs.

To further facilitate the answer to RQ1, our next step of the study is an experiment. The experiments are done by comparing the metrics measured from the toolchain with and without using certain serverless computing service from AWS. These metrics cover different perspectives which including cost, performance and ease of use.

To answer RQ2, The non-integrated toolchain is used to compare with the integrated DevOps toolchain built by the AWS DevOps tools. Besides, we conduct a study on a comparison between an AWS based traditional toolchain and this out-of-box integrated DevOps toolchain that is also provided by AWS. The reason that we keep the comparison scope within AWS is that by doing this, we make sure that hardware in both toolchains are from AWS, this could eliminate the errors caused by the hardware difference between vendors and focuses on the difference caused by toolchains themselves.

In the experiment for answering RQ2, we simulate the same DevOps lifecycle of a demo Spring Boot web app on both toolchains. We again measure the metrics in these two toolchains. The process is similar to what we do on RQ1. For software development teams, it could provide better insights on how to select the DevOps toolchains.

1.3 Thesis Structure and Main Contributions

In Chapter 2, we introduce concepts within the scope of DevOps. We also introduce the concepts in cloud computing which are related to our research. Chapter 3 is focusing on a survey on serverless computing technologies which the DevOps toolchain could make use of. Chapter 4 focuses on the design and the implementation of our DevOps toolchains(both non-integrated and integrated). Chapter 5 focuses on the experiments and evaluations, which show how the serverless computing services introduced in CH3 could benefit DevOps toolchain. We also compare integrated/non-integrated toolchain in CH5. We finally summarise our research and answer the research questions in Chapter 6.

The main contributions of this thesis project are:

- We provide a study on how could the DevOps tools leverage the cloud services to reduced development/deployment difficulties, lower the cost and improving the performance. This part of research could help the software team which is going to employ DevOps understand the practices needed. Besides, the research gives them a clearer scope of the tools needed for implementing the practices.
- We give the overview of 2 different types of DevOps toolchain. We also implement demo prototypes for each type of toolchain and conduct experiments with these prototypes. The experiment result shows a comparison between different toolchains. It could help the software team understand which toolchain cloud be selected based on the needs.

Chapter 2

Background and Concepts

In this chapter, I introduce several main concepts related to our study. Section 2.1-2.3 shows the definition of DevOps and two important concepts that DevOps is based. Section 2.4 introduces serverless computing.

2.1 Agile software development

The term ”Agile” represents the fast adaptation and response to the changes in the plan[21]. Agile software development is a method of software development that implements the ideology of ”agile”. Agile software development advocates the continuous development of software teams. The software development under this methodology will have shorter planning/development time before it delivers to the customers and could better adapt to changes in the environment and requirements.

Iterative Software Development: Agile software development uses an iterative way in the development process. The traditional software development process, like the waterfall method, requires the long and complicated planning process, and a complicated document. Once one phase of the development is done, the teams should not change the output (document and code) of this phase [22]. In contrast, agile software development aims to satisfy the customer with early and continuous delivery of the software [1]. Early means the shorter time before software delivery. Continuous means the development does not end with the delivery. Delivery means to end of an iteration, together with a demonstration to stakeholders. After delivery, the team continues to the next iteration according to the feedback it gets from stakeholders. In each iteration, the aim of the team is not to add major features to the software, rather is to have a working and deliverable release [23]. In the ideology of agile, the best design of the software product comes from the iterative development [1], rather than the tedious planning.

High Quality Software: The rapid development does not mean low software development quality. On the contrast, the quality of software design is highly appreciated in agile software development. The automated testing is widely used in Agile. The test cases will be defined and implements from the beginning of the development process. The test goes through the entire development iteration to ensure that the software is of high enough quality and can be released or shown to customers at any time during the iteration.[24].

Collaboration: The agile software development processes include collaboration across different groups, i.e. business development team, software development team, test team, and customers. It values face to face communications [25] and feedbacks. The purpose for these communications is first to let everyone in the multifunctional agile team understand the whole project, and second to receive feedback that helps the software in the right development track. The track which aligns with the requirement of the stakeholders [1].

According to the Manifesto for Agile Software Development, compared with traditional software development, the agile software development value these aspects [1]:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

2.2 Continuous Integration & Continuous Delivery

In the software development, CI/CD refers to continuous integration, continuous delivery and continuous deployment [26]. As I mentioned in 2.1, agile software development requires continuous software quality assurance and iterative development. Currently, CI/CD is one set of the necessary practices for the team to become agile by achieving the requirements above. Figure 2.1 shows the relationship between these 3 practices.

2.2.1 Continuous Integration

Continuous integration is the base practice of all practices within CI/CD, and continuous delivery/deployment is based on the continuous integration [26]. The continuous integration means the team integrate each team member's work into the main codebase frequently(multiple times per day). "Integrate" means merge the code into the main codebase [27]. The continuous integration rely on following practices: *Source Code Management Build Automation Visibility* and *Test Automation*. The definition of these practices are:

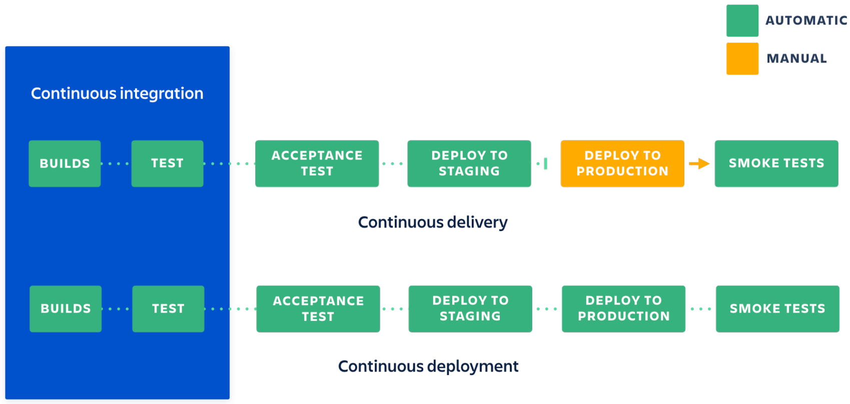


Figure 2.1: The relationship between continuous integration, continuous delivery and continuous deployment [26]

- *Test Automation:* Test automation means using separate software to execute the software automated without human intervention. It could help the team to test fast and test early [28]. There are 3 approaches of test automation: first, the automated unit tests which design by developers in the early stage of the project. Unit testing aims to verify if each component of the software project works properly; Second, the functional testing which tests the business logic behind the user interface. The function testing is a type of black box test which the tester only care about if the output is as expected under certain input; Third, the graphical user interface (GUI) testing test if the user interface meets the functionality requirement. The context of GUI testing is to simulate the user's operation on the user interface.

- *Build Automation:* Automate the process of creating software build. This means to automate the dependency configuration, source code compiling, packaging and testing. It is viewed as the first step to continuous integration [29]. There are 2 types of build automation tools, the build-automation utility and build-automation servers [30]. Build-automation utility means the tool to generate build artifacts by compiling the source codes. The common tools belong to this type include Cmake, Gradle and MSBuild. Build-automation server is the tool which executing build-utility tools, it allows build to be triggered from the outside or be scheduled on the time basis. Build-automation server is usually web-based. Continuous integration servers, such as Jenkins and Circle CI is considered as build-automation server.

- *Source Code Management:* In continuous integration, the team maintains a single source repository ans use version control system. In practise, this means one branch in the version control system act as the "mainline", while everyone works off this mainline [27]. However, everyone need to merge the code to the mainline everyday. For making sure that the mainline code still works after the merge, the mainline that merge the new code need to be built and tested. I will further introduce this practice in section 4.2.4.

With the help of these practices, for each developer in the software team, the workflow [27] in continuous integration as follows: In the development of each feature, the developer first pulls the code from the main codebase. During the development, new test cases should also be added to the automated test. Automated testing runs on the code after the developer finishes the feature development. This is for maintaining the code quality and minimise the number of bugs from the beginning. The actual practice for implementing this step is to have build automation tools compiled the code locally in the development machine.

After the step above, the developer already has the executable and the high quality (passed the automated test) code in the development machine before submitting the change to the repository. This represents the principle of quality and automation in agile software development. In the next step, the developer commits changes to the repository, which is the main codebase, and the system check the conflict and do the test/build again, to make sure that there are not any bugs missed in the test on the development machine. If the code passes this build and test, it will be merged to the main codebase, and the integration is done.

2.2.2 Continuous Delivery and Continuous Deployment

Continuous delivery is practice that software development team build a software that can be released at any time during the life cycle [31]. This practice ensures that the software always high-quality and in a deployable state [32]. Continuous delivery provides a clear way for software development teams to become agile [33][34]. In the last section, I introduce the concept of continuous integration. Continuous delivery is based on continuous integration, but it further automates the software deployment process. In the software deployment pipeline, the team divides the build into several stages, first build the product, and then push the product into a production-like environment for further testing. This ensures that the software can be deployed at any time. However, in continuous delivery, deploying software to production environment is done manually. The benefit [32][31] of continuous delivery includes:

- High quality of code: The automate and continuous testing ensure the high quality of code.
- Low risk: The software team could release the software at any time. The release process is easy, and it is also harder to make a mistake.
- Short time before going to the market: The iteration of software development is much shorter. The automated testing, deployment and environment

configuration shorts the development life cycle. The always ready-to-deploy status shorten the time from development to market.

The continuous deployment is based on continuous delivery. The only difference is continuous deployment automates the deployment process. In continuous delivery, the software is deployable but not deployed without manual approval. In the continuous deployment, each change that passed automated build and testing will be deployed directly. Continuous deployment is a relatively new concept, and most companies have not yet put this practice into production [35]. Although continuous delivery is a necessary practice for companies to become DevOps, it has been widely used.

2.3 DevOps

> The fundamental goal of DevOps is to minimise the service overhead so that it can respond to change with minimal effort and deliver the maximum amount of value during its lifetime.
>
> – Markus Suonto, Senior DevOps Consultant, Eficode

DevOps is a set of practices that aims to combine different, traditionally separated disciplines (e.g. software development, operations, QA, and others) in cross-functional teams with the help of automation of work to speed up software delivery without risking high-quality [36].

2.3.1 Emergence of DevOps

In the pre-DevOps era, development and operations were two different teams with different goals. The interface between them is based on the ticket system, and the operation team performs the ticket management. As I mentioned at 2.1, the goal of Agile is to shorten the deliver life cycle and quickly delivery software to the customers. Therefore, when practising agile development methods in this situation, developers try to deliver code and they will develop earlier. However, the operation team usually will delay the process for quality control or other reasons. In practice, this causes the delay between the code change and the software delivery to the customers [37]. The lack of communication and conflict between developers and the operation team slow down the software delivery process and also make it harder for the teams to be real Agile. Therefore the concept ”DevOps” is being proposed at 2008, for eliminating of the boundary between developers (Dev) and operation team (Ops).

2.3.2 Relationship with Agile

DevOps is the extension and evolution [39][37] of Agile. DevOps and Agile both driven by the collaboration ideology and the adoption of DevOps needs Agile as the

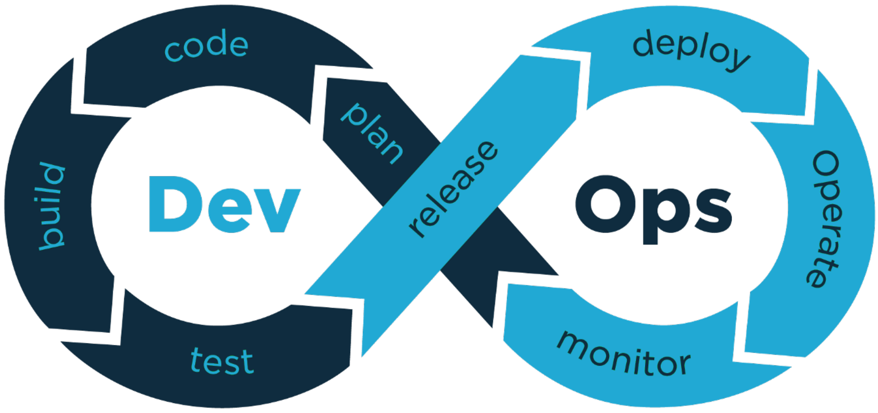


Figure 2.2: DevOps Practices and Workflow [38]

key factor [39]. DevOps has a different focus on agile. DevOps focus on the whole delivery and customer satisfaction while agile is focused on the development with the requirement and customer [40]. Figure 2.2 shows the workflow and practices of a team working under DevOps.

2.3.3 Elements

In this section, I will introduce the necessary elements that an organisation need to includes while introducing DevOps as a common practice. Eficode proposed it's DevOps capability model which act as the baseline for the DevOps transformation. This model consists following elements enablement, organization and culture, environments and release, builds and continuous integration, quality assurance, visibility and reporting, technologies and architecture. Combined with my research based on other materials related to DevOps, I summarise the DevOps capability model at a more general level, into four aspects.

Culture

According to Walls (2013), this is being done by promoting the culture with 4 characterises: open communication, incentive and responsibility alignment, respect and trust [41].

Open communication means open discussion and debate. Communication can help minimize the gap between developers and operations teams. Traditionally, communication within the team is carried out through a very formal and standardized ticketing system. However, in teams adopting DevOps, communication is not limited to the ticket system. Instead, the team will maintain free communication throughout the life cycle of the product, and they will discuss requirements, timetables, and anything else. In addition, information sharing is also very important[42]. Metrics and project status are available to everyone in the team , so each member can clearly understand the scope of the team's work.

The **incentive and responsibility alignment** mean the entire team (consisting Dev and Ops) has the same goals and assumes the same responsibilities. The transition from "Dev" and "Ops" to DevOps requires people who used in charges in only development and operation starting taking the responsibility in both side [42]. Such transition means if the product is failed, individuals or part of the team will be not solely blamed. This "no blame" culture could help each engineer be willing to take the development responsibility for the whole system [43].

Respect means all employees should respect and recognise the contribution of other teams members. A DevOps team is not a single team without any division of jobs, there is still an operation part within a team [44]. Therefore each part of the team need to **trust** the other parts is doing the best to benefit the whole team. On the other hand, the personal in the operation team will take development responsibility, and the developers will also put their hands-on operation and management[45]. To make people with different roles works in a team, trust and respect each other is critically important.

Organisation

In the organisational level, the DevOps emphasises the collaboration between different parts of an organisation. This is closely related to the "culture" part of this section. Within a team, each member should be a generalist who could understand all aspects of a project. There will not be a dedicated QA, operation or security team within a team. Instead, these are jobs that belong to everyone [43][4]. The structure and rule of the organisation should provide all members with opportunities to learn all skills needed for building the whole system.

The DevOps Handbook [4] published in 2016, proposed ways to organise a organisation for DevOps transformation. One of the principle in organise the teams is to keep the team boundary and comply the "two pizza rule" proposed Amazon in 2002. The rule is to keep the team size small for having more productive team meeting. A small team could help to reduce the inter-team communication, keep the team scope bounded and small [4]. Furthermore, smaller team also means less bureaucracy in team management. There are four benefits to have a small team:

- The smaller team allows each team member to understand the whole project easily.

- The smaller team could reduce the amount of communication needed. It could also limit the growth rate that the product could have.

- The smaller team could decentralise power. In DevOps, each team lead could define the metrics which become the overall criteria of the whole team's performance.

- In a smaller team, failure does not mean a disaster for the company. This fact allows the team to fail. Thus each employee could train their headship skill in the team without too much pressure.

Having a loosely-coupled architecture is another important organisational aspect for DevOps. The first benefit is the better safety. In the organisation with a tightly-coupled architecture, because each component is closely coped with each other, even a small change could results in large failure [4]. The second benefit of loosely-coupled architecture is productive. In a traditional organisation, the whole organisation shares a same development life cycle. The result of each team will be merged, tested together and deployed together, which is time-costly when configure the test environment and dependencies. A loose organisation enables each team to finish the development life cycle (from planning to deployment) independently. Each team could update their products independently, which gives the team more flexibility to align the product with the change in the customer requirement. The update of each team's product should not affects other teams product.

Automation

In the DevOps, automation means automation within the whole development and operation process. The organisations which employ DevOps aim for a high degree of automation[46]. With automation, people could be free from the repetitive work and reduce human error. It could help build the DevOps culture of collaboration, and it is seen as the cornerstone of the DevOps [47]. The main practices regarding Automation are the automated testing, continuous delivery and automated operation. The automation operation includes several practices such automated monitoring and alarming, automation infrastructure provision and environment configuration.

The continuous delivery pipeline is the core of automation within the scope of DevOps. As per discussed at 2.2.2, the continuous delivery will ultimately automate all steps between the developer to commit the code to the product in the production. In addition, the continuous pipeline brings together all automated steps within DevOps life cycle.

Infrastructure as Code is a practice which helps achieving automated operation part, specificity, environment configuration and infrastructure provision. The Infrastructure as Code (IaC) means everything at the software infrastructure level is defined as code [48]. Because it is code, the developer could use the automation methodology used in the software development to manages and deploy these codes. According to Christof et. (2016), under IaC, infrastructure can be shared, tested, and version-controlled [7]. This could help emphasize the automation within the operation scope. In addition, IaC the team could be free from the tedious environment configuration and shorten the product development lifecycle. Automating server configuration with IaC helps the developers and operation staff know the server configuration equally [47], which help build the culture of shared responsibility and trust.

Monitoring and Measurement

Monitoring is to continuously collect the matrices from the running system. Monitoring provides the team a good visibility on teh whole system. The team could get update on the system status, and find the problems in the system in time. To conducting the monitoring, the monitoring system needs to do the measurement, which is to collect data properly from the system. The measurement is defined as reducing the uncertainty through observation, which producing quantitative result [49]. The organisation should properly use the result (metrics).

In the DevOps way of development, the testing is the key to maintain the quality of the software continuously. However, when the product enters the production, I cannot test the software any more. So, I need monitoring to keep track of the status of the product [50]. According to State of DevOps report from DORA and Google Cloud, the good monitoring structure and the wisely usage of the data from monitoring for making business decision could improve the software delivery performance [51]. Thus, Monitoring is an important component of DevOps.

With monitoring, the software team could keep tracking the status, and maintain the quality of deployed production. Monitoring enables the management teams to track the KPIs during the production. The monitoring has also enabled the team to collect the data from customers' usage behaviour. This helps the agile development team to improve in the next iteration of the product [42].

The development of monitoring should be in parallel with the main product, and the monitoring system can be already be used against the "staging deployment" (see Figure 2.1) at the early stage of the iteration. By the practice of parallel development, the development team can improve the monitoring system continuously together with the main software system. In addition, the parallel development helps the team to find the gap in the monitoring earlier [50].

As I mentioned in the "Culture" section, the collaboration is an important part of the DevOps culture. Collaboration needs the communication and information sharing between the development(Dev) and operation(Ops) team. The monitoring could be one of the channels between the Dev and Ops since it can expose the information of the whole system, which helps team members to understand the system as a whole. This helps the team achieving the point I mentioned at 2.3.3 (Culture) that the project status and matrices should be available to every team members.

2.3.4 Toolchain

A DevOps toolchain is a set of tools that are integrated to aid the software development, deployment and management through the whole software development lifecycle. The goal of DevOps toolchain is to help the software development fits the DevOps principles [8][52][5]. Within DevOps toolchain, each tool in the toolchain related to a specific activity in DevOps, for example, version control, build, testing.

According to [5], DORA state of DevOps reports [51][53][54] and our previous definition of the DevOps, I summarise the essential component of a DevOps toolchain as below.

Project Management & Planning

Planning software development project, track the tickets and the issues, communication between and within the teams. The project management tools help to implement the DevOps culture, which enhances collaboration and knowledge sharing.

Configuration Management

Configuration management provides a central platform to manage the configuration across the assets. Such tool allow the team defines the desired state of the assets in a configure file. Then the tool automates the configuration process which reaching the assets to the defined status. In the cloud environment, a common practice of configuration management is through infrastructure as code, which is define the cloud infrastructure, services configuration and deployment orchestration as configuration file [55].

Continuous Integration

Continuous integration (in short: CI) is the top practice for improving the Deployment Frequency [53]. It is one of the most important parts of DevOps toolchain. As I introduced at 2.2.2, CI allows the developers to integrate their work more frequently to the production products, and it shortens the time to the market of the product. The automatic testing and code analysis integrated into the CI continuously maintain the quality of the product. CI tools also automated the most parts of the software development pipeline, In conclusion, CI helps the system fulfil the DevOps definition (2.3) by speed up the delivery by automation, maintain the quality by continuous quality assurance. The location of CI server is flexible, depends on the scenario, it could be either on-promise (on development machine or a local server) or deployed on cloud. Nowadays, some vendors provide CI as a service. In this case, the CI server is hosted and managed by the vendor, and provided to user as an online service. As I introduced in 2.2.2, CI brings together all automation tools, and automate the DevOps workflow which connects multiple automated processes. CI is the "confluence point" of the most DevOps tools, and thus the core of the whole DevOps toolchain.

Version Control

Version control is a important part of DevOps toolchain. It is a system that could record and track the changes in a set of files overtime. Version control simplifies the collaboration between team members. Furthermore, allow the simultaneous development of the different parts of a software system. According to [56] and [53], version control is the top practice when it comes to improving the multiple metrics in DevOps. Version control becomes the indicator of the software system performance [56] Infrastructure as code. An important DevOps practise I mentioned at 2.3.3 also relies on version control.

The version control is composed by a repository and the checkout. The repository is a database which record all history versions of the files. the checkout is the a local copy of the all the files. The user could edit the files in the checkout, then commit the change to the repository. Depends on the location of the repository, there are three types of version control systems [57].

- **Local Version Control Systems:** The repository located locally on the development machine where the user keeps the checkout. However, the repository is stored in an separated version database that keeps all changes of files.
- **Centralized Version Control Systems:** The repository located in a centralized server, while there are multiple checkouts on multiple development machines. This allows multiple developers work together under version control system. However, such setup is not fault-tolerated because to the VCS server is centralized.
- **Distributed Version Control Systems:** This is a type of VCS system that leverage the peer-to-peer approach. Most modern VSC system, such as Git, is using such approaches. The file history is not only kept in the server, but also in each development machine where has the checkout. Once the server dies, the history record will not be lost, and the development machine that retains the file history record will copy the file back after the server is up and running again.

Monitoring

The monitoring system is one of 4 basic elements of DevOps as I mentioned at 2.3.3. In the DevOps toolchain, the monitoring system detects the failure in the whole system and helps the software team find the problems earlier. The team could also collect performance related matrices with the help of monitoring system, which could be used for improving the . Moreover, monitoring system gives software team a better visibility to the system status. Monitoring system combines the data measured form the system and then visualizes these data on dashboard. The visualization helps people which is not in the operation team understand the data.

Test Automation

The test automation tool could verify the code before it being built. Such tools usually come either an independent tools, or an plugin that embedded within IDE, build server and continuous delivery pipeline. The integration of testing with other tools such as continuous integration pipeline makes it easy for the organisation to implement the quality gate in the software development [50]. The test automation tool runs on the local development machines after each build, after commit to the repository, and before the deployment to production. This policy makes sure that the testing ans quality control goes through the whole software life cycle. We will introduce when runs automation testing and what kind of testing will bu run in section 4.2.5.

2.4 Serverless Computing

In this section, I focus on the concepts of Serverless Computing. I will have more discussion regarding the new cloud service based on Serverless Computing in the next chapter.

Serverless Computing (in short: Serverless) is a cloud execution model in which the cloud provider manages the server and resources allocation. The popularity of serverless is precipitated by the development of microservices and container technologies [58]. A survey by the Cloud Native Computing Foundation (CNCF) showed that in 2019, 41% of respondents used serverless technology in production, compared with 32% in 2018[14]. The report of this survey also shows that serverless architectures and cloud functions are being used by 3.3 million developers [14] in 2019.

In traditional cloud computing services, users rent a fixed number of cloud servers from the cloud provider, and then the cloud provider charges users based on the lease period and server type (pay-as-you-go model). In serverless computing services, developers only pay based on the execution time of the program. Another difference between serverless computing and traditional computing method is that, in serverless computing, users doesn't need to care about the physical machine that runs the application. In addition, in serverless the environment that runs the application will be destroyed shortly after the application terminates. However, the task is still running on a physical cloud server that is fully managed by the cloud provider. This means that when serverless is used, the user leaves all server provisioning and management tasks to the cloud provider [59].

2.4.1 History

In the early days of cloud computing, the consideration behind cloud computing design was that developers only needed to transfer their deployment environment from a local server to a server on the cloud. Therefore, cloud virtual machines (for example, Amazon Web Service EC2) are the main form of providing cloud services. After Amazon Web Service started offering the service with the virtual machine, Google entered this field for competing with AWS, but in another direction. In 2008, Google released Google App Engine (GAE) [1][60]. The platform allows developers to run their code without managing the cloud virtual machine. This makes Google the first in the main cloud providers to allow the developer to run code on its cloud without provisioning and managing the cloud servers. However, the GAE only allows the developer to run the python code that is programmed with Google's framework, rather than running arbitrary Python code. Amazon Web Service (AWS) introduces AWS Lambda in 2014, make Amazon the first public cloud provider that provides serverless computing platform[61]. Since then the serverless computing starts its rapid commercial development. Following AWS, other providers also introduced their serverless computing platforms. Only in a single year (2016), Google [2], Microsoft [3], and IBM [4] released their serverless computing platform respectively. In the beginning, the serverless computing offering of vendors is limited to function as a service (FaaS), and company only use the serverless computing in some supportive components like scheduled tasks. Nowadays, the serverless is expending it's application scope together with the extension of serverless offering in cloud vendors. For example, AWS provides a serverless platform [5] with different component for a modern application, as well as tools and services for DevOps. These components are enough for a software team builds microservices architecture backend service for web applications, with DevOps toolchain that also build in AWS. In a word, the serverless cloud service cloud now covers the entire development life cycle.

2.4.2 Characteristics

I conducted research related to the main characteristics of serverless computing, and I summarise my finding in following four main characteristics based on materials [62][63][59] I read.

Event Driven

Event-Driven means the serverless applications is usually triggered and start running due to an event. There are different kinds of event that could act as a trigger. The first one is the HTTP request. When an HTTP request reaches the server, the serverless application could be triggered to reads the context of this request, execute the code, return the HTTP response to the frontend. This kind of pattern matched the nature of web application which allows the developer easily build serverless API for web/mobile applications on top of serverless cloud functions. The serverless application could also be triggered by changes in the database and object storage. This allows the serverless computing to be used as a background

[1] https://cloud.google.com/appengine
[2] https://cloud.google.com/functions
[3] https://azure.microsoft.com/en-us/overview/serverless-computing/
[4] https://www.ibm.com/cloud/functions
[5] https://aws.amazon.com/serverless/

task such as data processing. A good example is the serverless computing use case of Thomson Reuters in their social media data analysis project[64]. Thomson Reuters uses AWS Lambda to host a serverless application that triggers when new data is stored. The application processes the data real-time, extracts the hashtag trend data and stores it in Amazon DynamoDB, a database solution by AWS, which is also serverless.

Managed Resources Allocation

Managed resource allocation means that developers only need to deploy code without leaving operational tasks to the cloud. As I mentioned before The developer does not need provisioning or managing any server besides, the developer is not required to install any software or runtime [65] when deploying his/her application.

The scaling of the infrastructure which the developers are running their code is managed by the cloud provider. This also reflect the managed resource allocation of serverless. n traditional virtual machines, although some cloud providers (such as AWS and Azure) support automatic scaling; however, the scaling strategy must be defined by the user. Moreover, the user needs to set up the cloud infrastructure (such as Auto Scaling Groups and Elastic Load Balancing in AWS) for using autoscaling. In contrast, in serverless computing, the cloud provider will handle everything related to automatic scaling. Furthermore, together with other operational tasks, the availability and security issues of the underlying infrastructure are being taken care of by the cloud provider as well.

Pay-per-use

Pay-per-use is the significant characteristic of serverless computing form non-technical perspective. The traditional cloud server using pay-as-you-go mode. The billing is done based on the type of VM and the rental time of this VM. For the user, this is not economically flexible, because the user must pay the same price for an idle VM as when it is fully loaded. On contrast, in serverless computing, the users do not need to pay the idle time; they only pay for the time that the application is running. In many scenario, such payment mode could lower the cost. According to [59], serverless computing could be 6x cheaper than VM when doing on-fly video encoding, with 60x performance. A organisation could save up to 4x-10x when moves application to serverless [59][66].

Extensive Application Scenarios

Serverless computing has a wide range of applications. A common application is to deploy the runtime in a serverless environment. However, as mentioned in section 2.4.1, serverless computing is now not limited to deploy cloud functions but used in all the components that could be used when building modern applications. For example, beside serverless functions (AWS Lambda), the serverless offering in

AWS also includes the serverless database, container runtime services, data analysis and Kubernetes cluster. Google cloud also advocates ”full-stack serverless” [67]. Like AWS, Google Cloud also provides various serverless solutions ranging from computing and DevOps storage to AI and data analysis. In addition, Azure’s serverless products also cover a wide range of back-end components, including computing, storage, artificial intelligence, monitoring and analysis [68].

2.4.3 Limitations

Serverless computing is not the perfect solution. In some aspects, it still has it is limitations.

Performance

This is mainly the problem within the computing task that runs serverless. In the current serverless products of cloud providers, the computing power of serverless computing is limited. For example, in the virtual machine service (AWS EC2) provided by AWS, users can choose virtual machines with up to 96 CPUs and 192 GB RAM. In the serverless AWS computing engine, the maximum RAM size allocated is only 3008MB [69], and the maximum number of vCpus is not specified in the document. By making it unsuitable for heavy tasks, it limits the application scenarios of serverless computing to development team. In some cases (such as ML model training), the limitations of hardware selection are also mimicking performance. Research experiments [59] at the University of California, Berkeley show that because AWS Lambda does not support GPU computing, makes it 21 times slower than EC2 instances using GPU [70] when training deep learning models. In this case, longer execution times could make serverless servers more expensive, the research also shows serverless has poor performance in MapReduce and linear algebra computing. In conclusion, to a s development team, selecting serverless computing means limited hardware option, and poor performance with high cost in some scenarios.

Cold Start

The cold start is also a disadvantage of serverless. In when running a function on serverless cloud service, the functions are being served by container [71]. As long as the functions keep bring triggered, the container which hosting the functions will stay active. The cold start means the trigger event happens when the function is not being triggered for a too long time, so the cloud provider has already deactivated the container. In such a situation, the cloud has to deploy the code again and spin up a new container. This will significantly add overhead to the total execution time. Thus, if the development team needs to run a short task frequently, but not so frequently to keep the cloud function ”warm”, serverless is not the best option. This is because the cold start time could takes even longer than the actual runtime, which will lower the performance.

Fortunately, for AWS Lambda, there are plugins exists to solving this problem. The common practices of these plugins is to use CloudWatch to ping the function periodically.However, for other serverless services (such as AWS Fargate), there is no way to significantly shorten the cold start time.

Communication Pattern

The communication pattern between serverless services is limited: In current serverless computing offering from cloud providers, there is a lack of peer-to-peer networking between different running serverless instances [70]. This mean some heavy lifting inter-communication such as streaming content to another function [72] cannot be done efficiently. For example, in AWS Lambda, replacement of peer-to-peer networking between executing cloud functions is through slow cloud storage [70]. While the communication between virtual machines is through the network interface, which is much faster than cloud storage. Such limitation could further affect the performance of the distributed system that hosted by serverless since the distribute algorithm largely depends on the communication between nodes.

Another limitation is the communication pattern of serverless leads to more inter-instances communication. A good example is MapReduce [59]. While in VM part of shuffle and aggregation operation could be done within a VM instance, but between different tasks, such operations in Lambda must requires inter-instance communication, since each task is on an independent instance. This problem largely increases the need of network communication. The experiments from UC Berkeley shows during MapReduce operation, serverless functions cost 15% more than VM [59].

Chapter 3

Overview of Current Serverless Cloud Services Offering in AWS

In this chapter, I will introduce the serverless cloud services in Amazon Web Services (AWS) that I will use in the DevOps toolchain and the experiments. In 3.1, I introduce the AWS Elastic Container Services and AWS Fargate. In 3.2 I introduce the AWS Lambda and AWS CloudWatch in 3.3. I introduce AWS DevOps tools in 3.4.

3.1 AWS Elastic Container Services with Fargate

In our toolchain, I make use of Elastic Container Services with Fargate to run the Jenkins build agent. In this section, I introduce ECS and Fargate, and how do they combine to host the serverless Docker container.

3.1.1 AWS Elastic Container Service(ECS)

Amazon Elastic Container Service is a managed container orchestration service that runs Docker containers. AWS fully manages the ECS service, which means that AWS will be responsible for some operational tasks, such as automatically scaling the running container. In order to introduce how the container runs on ECS, I first introduce a few concepts.

Task: Task means a container instance that runs in the ECS cluster. A task is defined by task definition, which is a JSON file that contains the following information: container definition, network, hardware configuration and launch type. The task is the instantiation of a task definition [73]. The ECS task scheduler is responsible for putting the task to the cluster.

Service: A service is an abstraction of a set of tasks that include a specified number of tasks runs simultaneously.

Launch Type: Launch type defines on which infrastructure the task will run. Currently, there are two options, EC2 (virtual machine) and Fargate (serverless). The EC2 launch type refers to running the container (task) in a group of EC2 virtual machines. This launch type requires the user to manually create and managing EC2 VMs. The Fargate launch type means run containers in AWS Fargate, and Fargate is a serverless container service in AWS. This launch type does not require user provisioning and managing the infrastructure that runs the containers. Instead, AWS takes over these tasks. Our first experiment in chapter 5 will be related to a comparison between these two launch types.

The workflow for running an container in ECS is as follows: The first step is to have the task definition define the specification of the Docker container that is going to run. In the next step, a task defined by this task definition is created.

In the DevOps tool chain, ECS can be used to host Docker-based build agents in the continuous delivery pipeline. Docker-based build agent means running certain stages in the pipeline distributed in Docker containers. ECS supports the use of APIs to perform the two steps we mentioned in last paragraph, which makes it easy for DevOps tools to deploy Docker-based build agents to ECS cluster. Major continuous delivery tools support the Docker-based build agent. I will thoroughly introduce the Docker-based build agent in 4.2.

3.1.2 AWS Fargate

AWS Fargate is a serverless container service by AWS, and as I mentioned above, one of the launch types of ECS. It removes the need for provision, manages the server from the user's side. Fargate also follows the payment mode of serverless computing, which is paid for the runtime of each running container.

I notice that different from other serverless computing services in AWS, for example, AWS Lambda, Fargate cannot be used independently. To use Fargate, the user needs to select it as ”Launch type” in Elastic Container Service, which means the container runs under this task definition will run in Fargate. I call this method of using Fargate as ”Elastic Container with Fargate” in the following chapters. Another way of using Fargate is to run pods in Fargate when deploying Kubernetes cluster to AWS Elastic Kubernetes Service (EKS). This way user could run a fully serverless Kubernetes cluster in AWS, without managing backend infrastructure which runs each pod of the cluster.

3.2 AWS Lambda

AWS Lambda is AWS's first serverless service, It was launched in November 2014. In AWS Lambda, users can upload codes called ”Lambda functions” to AWS Lambda. AWS Lambda runs the code in its own hosting infrastructure. ”Managed” refers to AWS performing all management tasks of back-end services, including server and OS maintenance, server configuration, and scaling. In addition to server-related tasks, AWS will also be responsible for security, monitoring, and logging.

AWS Lambda is event-driven, which means that when an incoming event triggers the function, the deployed lambda function will start running. I introduced the characteristics and applications of even driving in 2.4.2. In addition, AWS allows user to associate Lambda functions with other AWS services. This means that changes in AWS services can be used to trigger our Lambda functions. The combination of even-numbered driving characteristics and association with AWS services allows user to extend the functionality of AWS services. I will introduce how to use this combination in Chapter 4.

3.3 AWS CloudWatch

As I mentioned in chapter 2, monitoring is one of the DevOps practises. AWS CloudWatch is a monitoring and observability service [74]. It providing an out-of-box monitoring solution for both infrastructure and deployed applications. CloudWatch could also helps on resource utilization and gives an uniform platform to monitoring operational health of the infrastructure in both AWS and on-premises. In addition, CloudWatch gives a complete visibility to AWS infrastructure status, because CloudWatch natively integrates with over 70 AWS cloud services [74].

The core function of CloudWatch is to collect metrics and logs of all running AWS services under the current user, display these data in real time and save the data for further analysis. CloudWatch supports monitoring all services running in serverless and server-based AWS. In addition, it can be used to monitor on-premises services. Monitoring in CloudWatch follows the following workflow:

1. **Collect:** CloudWatch gathers the log from services in AWS. In addition, it also gathers metrics include CPU/RAM utilization, network I/O e.g..

2. **Monitor:** CloudWatch visualizes application and infrastructure logs and metrics on the dashboard. Users can check the status from the dashboard and can also set CloudWatch alarms.

3. **Act:** CloudWatch continuously monitors the status of AWS services. When certain metrics reach the value set in the CloudWatch alarm, the alarm will trigger the action set by the user. A common use case is to set an alarm about CPU usage and use that alarm to trigger auto-scaling. Alarm actions may also trigger Lambda functions.

4. **Analyze:** CloudWatch can save logs and analyze them later. The analysis includes customizable indicators, contributor insights and log analysis.

I will introduce how the CloudWatch is being used in our toolchain in chapter 4.

3.4 AWS Developer Tools

AWS provides a set of cloud-based tools which helps user to build an integrated DevOps toolchain. These tools include the following four tools.

3.4.1 CodeBuild

CodeBuild is a fully managed build server in AWS. CodeBuild mainly takes care of the automated build and automated testing within configuration delivery. Same with all serverless services, CodeBuild frees the software team from building and managing build servers.

Although as a managed service, still, CodeBuild provides the user with a configurable build environment. Users are allowed to select the hardware configuration of the build machine. CodeBuild provides several out-of-box build environments which include build dependencies for the project in different programming languages. For example, a Java environment including JDK and Gradle; PIP and Python for Python development; Android build environment, etc.[1] Users can also use a self-defined build environment in compilable with their requirement. Furthermore, CodeBuild provides good integration with popular tools. For example, CodeBuild can be integrated into a continuous pipeline in Jenkins by acting as a Jenkins build agent. This could be achieved through Jenkins' plugin "AWS CodeBuild" [2] developed by AWS CodeBuild engineering team.

3.4.2 CodeDeploy

CodeDeploy is for automating the application deployment to both AWS services and on-premise services. Besides the basic functionality as automated deployment, CodeDeploy also minimized the downtime by using advanced deployment strategies (blue/green deployment and rolling update) and continuous health checking. CodeDeploy also allows users to continuously monitor the running status of deployed applications.

3.4.3 CodePipeline

CodePipeline is for modelling the workflow within the continuous delivery pipeline with both graphic interface and code. The user could use different DevOps tools from AWS or third party in each stage of the CodePipeline. In the other way around, CodePipeline could connect different DevOps tools I mentioned above into an integrated continuous delivery pipeline. These tools include AWS DevOps tools such as CodeCommit, CodeDeploy, CodeBuild; Third-party tools such as GitHub, Jenkins, XebiaLabs etc. [3]

[1] The full list can be found at https://docs.aws.amazon.com/codebuild/latest/userguide/build-env-ref-available.html

[2] https://plugins.jenkins.io/aws-codebuild/

[3] See https://aws.amazon.com/codepipeline/product-integrations/

3.4.4 CodeStar

CodeStar is a uniform platform that joins AWS DevOps tools as an integrated DevOps toolchain. I mentioned CodePipeline above which integrated different tools to an integrated continuous delivery pipeline, while CodeStar brings the integration to a further step. I will introduce what tools does CodeStar include in section 4.3.4 where I present our implementation of integrated DevOps toolchain in AWS.

Chapter 4

Design of DevOps Toolchains

In this chapter, we will introduce the design and implementation of DevOps toolchains. Note that for the experiment that is answering RQs in Chapter 5, we implement two different continuous delivery pipelines design with two sets of tools respectively, one with tradition non-integrated tool while another one with the serverless integrated DevOps tools from AWS. In conclusion, we introduce the design of both toolchains(integrated and non-integrated) and explain how we come to this implementation in this chapter. We will also compare these two types of toolchains within the scope of functionality and ease of implementation.

In Section 4.1, we present the case software project developed by us that will be built, tested, and deployed by our DevOps toolchain in the experiment. In section 4.2, we introduce the design and implementation of our non-integrated DevOps toolchain. Section 4.3 is related to the integrated toolchain, and section 4.4 is a comparison between Integrated and non-integrated toolchain. Lastly, in section 4.5, we talk about the challenges we met during the implementation.

4.1 Case Project

We first develop the case project. The case project is an example software project which will be used to test our implementation and run the experiments in which we simulate the DevOps development process of the case project on our DevOps toolchain. Although the type of our case project has no effect on our DevOps toolchain on the architecture level, the build dependencies and the software configuration inside our toolchain could be affected by it. Thus we must have an introduction to the case project.

4.1.1 Programming Language and Framework Considerations

Java is one of the most common languages used in commercial software development. According to the TIOBE index of programming language [75], Java is the most popular or the second most popular programming language in the world since the mid-1990s. Besides commercial software development inside companies, Java programming language is widely used in open-source software development. The report [76] from GitHub shows that Java ranked third most popular programming language in 2019, and it ranks second before 2018. Furthermore, Java has good versatility, which means it almost every kind of applications. For instance, web applications, desktop applications, besides, Java is the main development language for Android applications.

To the DevOps point of view, the Java programming language has a complete ecosystem. The complete ecosystem means there are tools for every phase of Java application development. These tools include: build, code analysis, testing frameworks, artifact management, build automation & dependency management et. These tools could be easily integrated and act as part of the DevOps toolchain.

Hence, due to the popularity, versatility and complete ecosystem of Java programming language, we select Java as the language of the case project.

One of the major application of Java in web development. Currently, 7 out of 10 [77] most popular website is using Java as a web development language (server-side). In the field of web development, Spring framework is the most popular framework for Java, and it is being used in many major internet companies including Google, Microsoft and Amazon [78].

So, we choose Spring the framework to build our application. To develop our Spring application, we use Spring Boot[1]. Spring Boot is a project under Spring, which, according to its documentation, is to allow the developer to create Spring application with the minimal effort [79], by simplifying the configuration of Spring framework.

4.1.2 Project Description

The case project is a simple REST API (Figure 4.1.2) which returns the info of all installed software packages in the host machine in JSON format when the frontend sends an HTTP GET request to the backend.

4.2 Design of Non-integrated DevOps Toolchain

In section, we present our design of non-integrated DevOps toolchain. Part of the components is still based on the virtual machine those we cannot call it a serverless toolchain as in the integrated toolchain. Each section is the introduction to the design of each component. We also present the consideration when a select tool for this part of the toolchain in each section. Besides, in each section, we introduce how could serverless computing be used by this component in general and the benefits to the specific tool we select.

[1] https://spring.io/projects/spring-boot

```
Method: GET
Endpoint: /packages
Success Response:
Code: 200
Content:
[
{
name : (Package name)
description : (Package description)
dependencies : (Dependencies)
}
]
Error Response:
Code: 500
Content: { msg: Server Error! }
```

Figure 4.1: RESTful API Interface of Case Project

4.2.1 Architecture

The toolchain implementation is based on the DevOps elements we presented in Chapter 2, and the DevOps practises from Eficode. Figure 4.2 shows the architecture of our DevOps toolchain. In here we are only presenting architecture on a more general level. The detailed architecture of each component will be introduced in the following sections, both text and graph.

When the developer pushes a new commit to the repository in GitHub [2], Github will send HTTP POST request that contains the necessary information to the Jenkins master node. Jenkins master, which triggered by the HTTP request, will create a new job for this project according to the information that the HTTP request contains. The job will first pull the latest code from the git repository, then runs the docker containers with required build environment and build the project. In the end, a docker image for running the project will be created and be pushed to the container registry of AWS. Depends on the git branch that the developer committed to, the project will be deployed to a different development environment.

Figure 4.2 shows the architecture of our DevOps toolchain. We can see except version control, and the whole environment is running in Amazon Web Services. Due to the limitation of space, the internal architecture of certain components is not shown in the graph. Instead, we show them in the following sections.

[2] https://github.com/

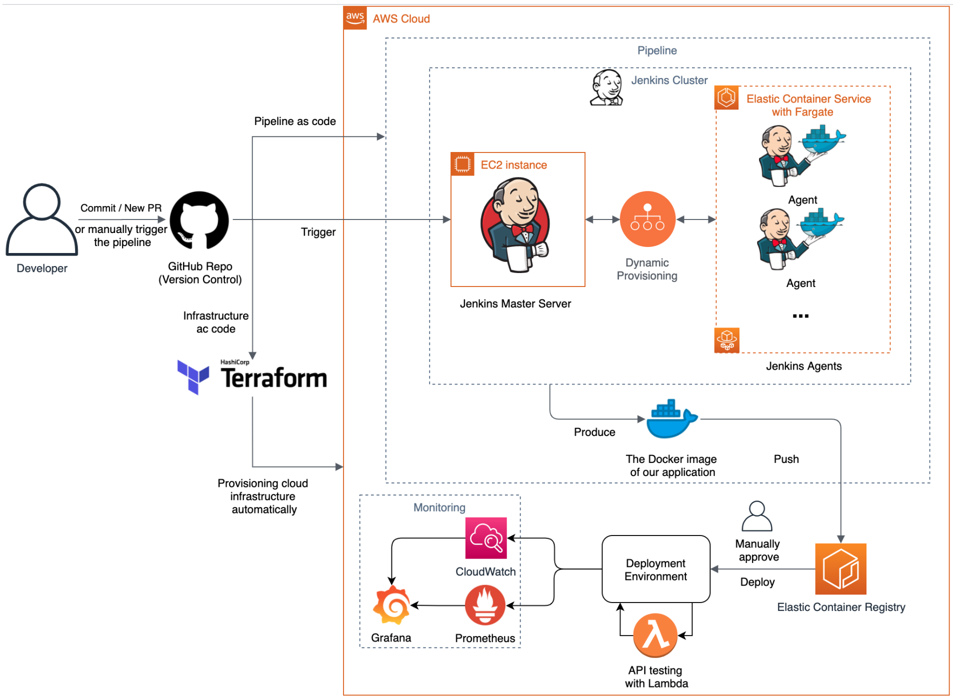


Figure 4.2: Architecture diagram of the non-integrated DevOps toolchain

4.2.2 Introduction to Tools Used in the Implementation

One of the essential steps to build the non-integrated toolchain is to select the proper tool for each component. In this section, we describe our consideration when we select tools.

Continuous Delivery Pipeline The most popular server-based tools for build continuous delivery pipeline are Jenkins[3], Drone[4], GoCD[5] and Circle CI[6]. Table 4.1 shows a comparison between these tools. As we can see from the table, Jenkins is the most popular option for CI/CD. Jenkins has wide application in the commercial use case, and the high popularity in the open-source community as well. Although compared with the other three newer tools, Jenkins is more focuses on the "Build" step within the continuous delivery pipeline since it was originally a build automation server. But, the open-source nature of Jenkins gives it a much wider selection of the plugin, which means Jenkins can be used for almost all steps in a continuous delivery pipeline.

[3]https://www.jenkins.io/
[4]https://drone.io/
[5]https://www.gocd.org/
[6]https://circleci.com/

	Jenkins	Drone	Circle CI	GoCD
Open Source	Yes	Yes	No	Yes
GitHub stars	15.7k	21.2k	-	5.7k
Github contributors	614	258	-	116
Plugin extensions	Over 1500 [7]	93 [8]	110 [9]	88 [10]
Price of self-hosted solution	Free	Free	$35 user/month	Free
Number of companies use it in the tech stack[11]	2634	82	1368	42

Table 4.1: Comparison of continuous delivery tools

Created by Kohsuke Kawaguchi in 2001, Jenkins is an open-source continuous integrating tool write with Java. It is suitable for a team of all sizes and varies of languages and technologies [80]. Furthermore, Jenkins also attracts software teams with its easy-to-use and high extendibility [80] with a thousand of the plugin. More plugin keeps coming since Jenkins has an active open-source community. These plugins help Jenkins keep up with the fast-developing DevOps practices, and help Jenkins integrate with the newly emerging tools and cloud services. The extendibility makes Jenkins still the most popular tool for DevOps toolchain even it is an aged software created when the term "DevOps" just appeared.

Our continuous delivery pipeline is developed with Pipeline plugin[12] in Jenkins. Pipeline plugin allows us to define a continuous delivery pipeline as code in Jenkinsfile. In the pipeline, a conceptually distinct subset of tasks within the continuous delivery pipeline [81] is defined as a "stage"[13] and each task within a step is called "step". Each pipeline is binding with a "project". An execution runtime of a project/pipeline is called "build", and the machine (virtual machine, container, e.t.) for running the build is called "agent".

Build & Test Automation Tool For the build stage within Jenkins pipeline, we use Gradle[14] as the build tool. Gradle is a powerful build tool initially designed for JVM based language, but now it also supports other programming languages, for example, C++ and Python. Like Jenkins, Gradle also has a dynamic ecosystem with thousands of plugin. This enables the possibility to use different kinds of tools such as unit testing and code analysis within a single pipeline of Gradle. Gradle also makes the dependency management easy, and dependencies could be easily

[7] https://plugins.jenkins.io/
[8] According to GitHub search result
[9] https://circleci.com/integrations/
[10] https://www.gocd.org/plugins/
[11] based on data from StackShare
[12] https://www.jenkins.io/doc/book/pipeline/
[13] For example, "Build", Test", "Deploy" step in a continuous delivery pipeline.
[14] https://gradle.org/

added to the project by editing the Gradle configure file of the project. Furthermore, Gradle supports configuration as code. This allows developers to define all the build configurations of a software project in a single file.

For unit testing within the build stage, we are using JUnit [15] as the tool for testing. For code analysis, we use SonarQube[16]. Both are one of the most common used tools in their specialized field in the Java ecosystem. Moreover, both tools have official Gradle plugin, which allows us easily use them with Gradle.

Deployment and Jenkins Agents We will widely use Docker [17] in our pipeline. Docker is an open-source software which could pack, deliver and run the software as a container. A container is a separate unit that includes the application and all its dependencies which allow application runs in the same way regardless of the host environment [82]. A container is the running instance of a Docker image that defined by Dockerfile.

There will be two main use cases of Docker in our toolchain. Firstly, we run the build stage within the container. Nowadays, Docker[18] is being widely used as build agents in continuous integration and continuous delivery (CI/CD) pipelines. This means the pipeline will execute specific steps inside ephemeral Docker containers [83]. It is easier to manage build dependencies in the Docker container. Besides, the container-based agent requires less effort to maintain.

In our case, to build the case application, the host machine needs to have JVM installed. However, we want to make our pipeline not only suitable for Java application but also easily be used to build an application in other programming languages. Docker solves this problem by provides excellent isolation from the host machine. Thus, we can configure the built environment (operating system version, dependencies) runs within a Docker container without actually install anything on the host machine by merely editing the Dockerfile.

We also use Docker to Dockerize our application which creates a Docker image of our application. Docker allows us to specify all system dependencies in a single file (Dockerfile), so there is no need to have any Java environment pre-installed in the deployment environment which runs our application. This is because all environment is already being packed in our Docker image. By doing this, firstly, we reduce the operational effort. Secondly, we improve compatibility since Docker makes sure that the docker image could run in the same behaviour no matter what host machine it runs on. Also, all major cloud computing providers support Docker. We could easily run the container from our Docker image on their VM, and they are serverless computing services. This means our Dockerized application could easily be cloud-native and be deployed across a multi-cloud environment.

[15] https://junit.org/junit5/
[16] https://www.sonarqube.org/
[17] https://www.docker.com/
[18] https://www.docker.com/

Monitoring We using CloudWatch for monitoring the status of the cloud infrastructure. CloudWatch is an tools provided by AWS, which we already introduced in Chapter 3. However, It's not possible for CloudWatch to give us insight about the status of Java SpringBoot application. More generally, the metrics within the application framework. In Spring Boot, these metrics are HTTP statistics, CPU load and JVM statistics. Prometheus [19], together with Grafana could fill this gap.

Prometheus is an open source monitoring and alerting solution initially built by SoundCloud in 2012 [84]. Prometheus is used for reading numeric metrics that are recorded in time series. In Spring Boot, there are some plugins ,for example, Micrometer [20] exist which could export all the Spring Boot specific metrics to Prometheus in time series. Thus Prometheus has perfect fit with our case project. In addition to collecting metrics, we could also set alarm within Prometheus, which could alert user when some metrics are not in normal range. We can also query the metrics from the past. Although Prometheus supports simple graph which show the metrics' change with time, it is not user friendly enough. Thus, we introduce Grafana to better visualizes the data collected by Prometheus.

Grafana is a observation platform which it's core feature is in the visualization. Users could define dashboards according to their needs with JSON files. Another main feature if Grafana is that it could gather data from different platform into one dashboard. In our case, with Grafana, we can display the metrics from Prometheus and CloudWatch in a single page. This could gives us great overview to both Spring Boot application and cloud infrastructure as the same time.

4.2.3 Infrastructure as Code (IasC)

Configuration Management is one of the component of DevOps toolchain that we mentioned in Chapter 2. Infrastructure as code the common practices to implement configuration Management in the cloud-based environment.

Terraform [21] is one of the most popular tools to manage cloud Infrastructure with IasC practice. It has the thorough support of AWS. In our implementation, we define our could infrastructure and all AWS resources including EC2 virtual machine, ECS cluster, security groups and network Infrastructures in a series of configuration files. Then we create the cloud environment by simply using CLI interfaces. Figure 4.3 shows the creation of the cloud environment with Terraform.

4.2.4 Version Control

Version Control System (VCS) is the process that record the changes in files set over time [85], and versioning the history of these files. VSC is suitable for track the development progress and manages the goal within a software development team [86]. Among all software for version control, Git is the most popular one

[19] https://prometheus.io/
[20] https://micrometer.io/
[21] https://www.terraform.io/

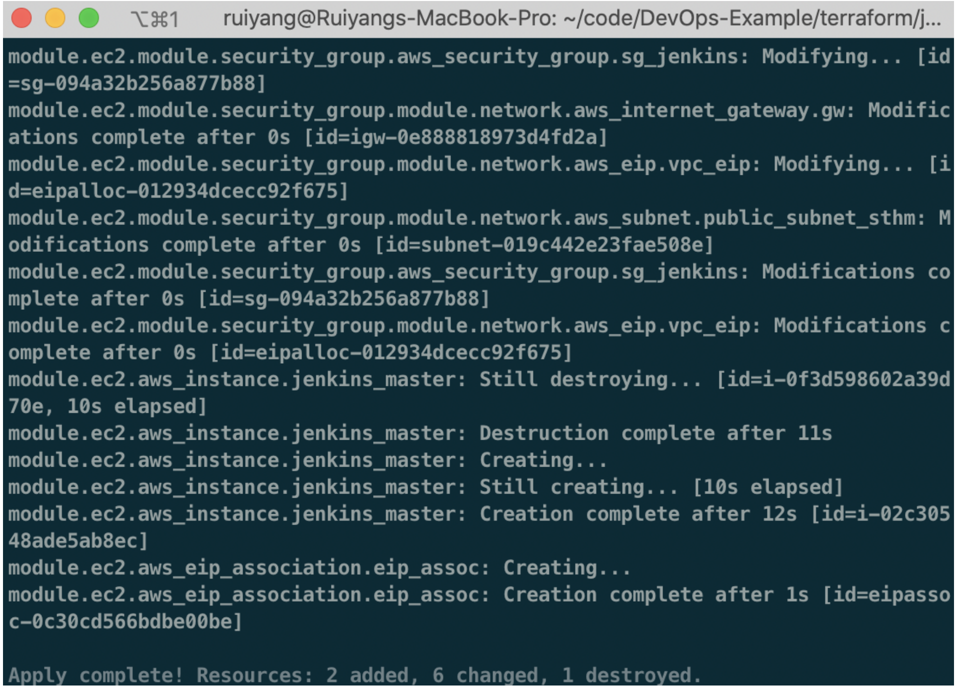


Figure 4.3: Creating a cloud environment with Terraform CLI

nowadays. The survey [87] from Synopsys shows that in 2019, 71% of the project today is using Git as it is versioning system while SVN that ranks in second only be used in 25% of the projects. We use Git as the version control system since it is used by most of the software development teams nowadays. We use GitHub for hosting the case project. Github is the biggest preform in the world that hosting a version-controlled software project for free using Git. It provides interfaces with different DevOps related tools which makes it easy to be integrated into all kinds of DevOps toolchains.

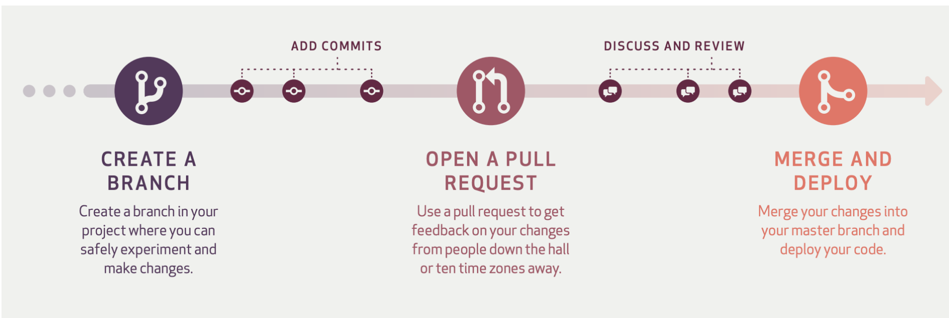


Figure 4.4: GitHub Workflow [88]

The Git flow [89] proposed in 2010 is a successful workflow for working with

Git. Git flow has already widely used and has been approved by the software industries. However, to better cope with the frequent release nature of DevOps, the Github workflow – a simplified version of Git flow is proposed by GitHub. Therefore, GitHub workflow [90] is being chosen as our workflow in the version control. The simplified version of this workflow is shown as in Figure 4.4

Several general principles followed by us when adapting GitHub flow, we refer to principals in [90] to design our workflow.

- Master branch is always deployable. This means when deploying the continuous delivery pipelines in our toolchain, only the master branch can be deployed. Moreover, there should not have any code which is not good to be deployed in the master branch.

- When working on the new feature, make a new branch for this feature. The name of this branch should be descriptive, which reflect the content of this feature. Commit the new code related to this feature to the feature branch. And push from this feature branch to the branch with the same name on the remote server (github.com) when necessary.

- Open a pull request[22] when the feature is ready to merge, or when developer feel that he/she need help or comments from other team means on this feature. Others also do the code review in the pull request.

- When the code is reviewed and is good to be merged, the developer should merge the code to the master.

- After the code of this feature is in the master, the code will and should be immediately deployed. There should not be any rollback in the master branch. If there are any issues within the newly merged code, a new commit or a new branch should be made to fix the issue rather than rollback on the master.

Note that in our Git workflow, there are several time points that we need to run the continuous delivery pipeline within the toolchain. The continuous delivery pipeline will also vary with the time point within the version control workflow. We will introduce this in detail on 4.2.5.

4.2.5 Continuous Delivery Pipeline

Figure 4.5 shows the six Jenkins stages in our pipeline. The bottom part of this Figure shows the task distribution between the master node and agent nodes. The master node is an EC2 virtual machine while agents run on Fargate instances within an ECS cluster. In section 2.3.1 we mentioned that the development of the monitoring system should be in parallel with the main software project, thus in our

[22] https://help.github.com/en/GitHub/collaborating-with-issues-and-pull-requests/about-pull-requests

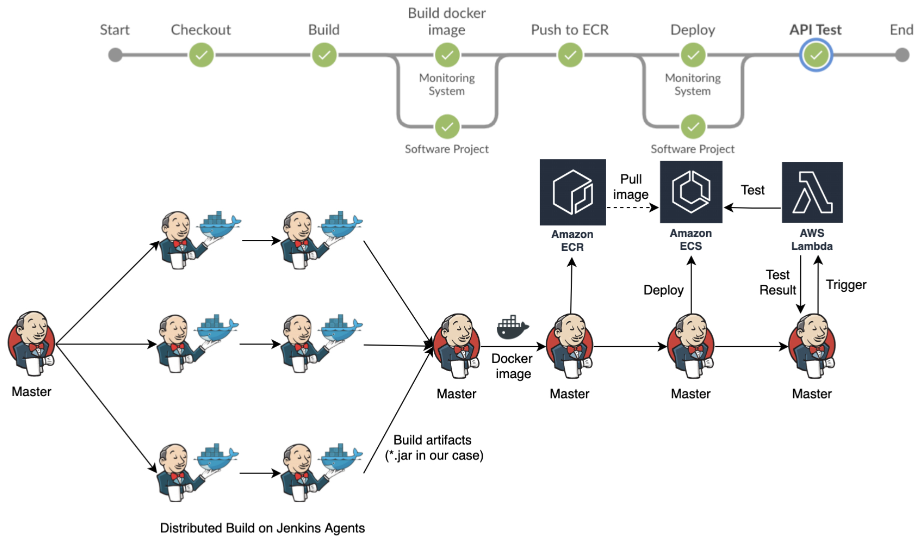


Figure 4.5: The Stages and Distributed Build in Our Pipeline

pipeline, the build and deploy of monitoring system is in parallel with the case project.

As we can see from the Figure, when the master node starts a job, it will create a Docker container in AWS Fargate as the agent. The agent will pull codes from VCS, build the code, and then send the build artifacts back to the master node. After this, the container will be terminated. The master node will continue the rest steps.

Build Agents Build agent is an independent computation unit (VM or Docker container) that could exchange data with the Jenkins master node and run a certain part of the pipeline. To implement a Jenkins build cluster, we need first to implement build agents. We discussed why we use Docker-based agent in our Jenkins build a cluster on 4.2.2 and we decide to it in our implementation. The first step of our implementation is to develop our own Docker image [23] of the Jenkins agent. We use the "jenkins/jnlp-slave"[24] as the base image, this allows our Jenkins agent to establish an inbound connection to the Jenkins master with TCP. The next step is to set up the built environment within the agent. We add shell script for auto-install all build dependencies of our case project when we build this Docker image. In the last step, we build the Docker image for build agent and push it to DockerHub.

We also discussed how Fargate allows us to run container serverlessly. To make use of Serverless offering of AWS, we let Docker-based Jenkins agents run on AWS Fargate to cut the operational effort and automate the scaling of Jenkins cluster. To implement this, we use Jenkins plugin "Amazon Elastic Container Service (ECS) /

[23]The Docker image we developed could be found at https://hub.docker.com/r/dry1995/jnlp
[24]https://hub.docker.com/r/jenkins/jnlp-slave/

Fargate", which is the only Jenkins plugin allow us to host Jenkins agent in Fargate.

Considerations in Designing the Workflow of Distributed Pipeline The considerations behind to our design are that the first two steps take most of the time in our pipeline and according to Figure 4.6 runs more frequently than other steps [25]. The running time will be further extended when building a larger project. These two stages will be the bottleneck of the pipeline if we have it on the master mode. So we need to offload these steps to Jenkins agents for better performance.

The second reason is: as we mentioned in our introduction of Docker at 4.2.2, the built environment inside the Jenkins agent that runs in Docker container is easier to be changed. When the team want to build the same code for different OS (Which happens in C/C++ development) or want to have a different build environment for different projects, they eliminate tasks such as configuration and installation different environment thanks to Docker. Instead, they can just modify the Dockerfile that defines the Docker image of the Jenkins agents. However, we cannot put the stage that builds a Docker image in Jenkins agents. This is because AWS Fargate does not allow building agent runs in a free container which means we cannot use Docker within the Jenkins agent's container that runs in Fargate. This is one significant limitation of Fargate, so we have to move the step back to the master node. Fortunately, in our case project, the Docker build only takes a short time (¡1s on average). Therefore this will not slow down the whole pipeline.

We also notice that the Deploy stage also takes a long time. Still, we do not have it in the distributed build because: first, it is on the end of a pipeline so it will not block the further steps, second, the pipeline runs the stage less frequently than first two stages as shown in Figure 4.6. Thus there will be less possibility that there are many jobs runs at "Deploy" stage in parallel.

Workflow for Continuous Delivery Figure 4.6 shows our proposed workflow of a project that goes through the continuous delivery pipeline. We can see when the event on the feature branch triggers the pipeline, and it only runs through the first two stages. This is because according to the practices of continuous integration mentioned by us in 2.2.2 and by Martin Fowler in [27], a developer should merge(the "integration" in continuous integration) his/her work couple times per day. Therefore the whole pipeline will run the code with this new feature at least several times a day. This already ensures the code could frequently be tested and deployed into the test environment. Thus, in the pipeline runs after the push to the feature branch, the further steps could be skipped.

The developer only commits to the feature branch. The pipeline runs first two stages after a developer pushes local commits to Git. It first pulls the newly pushed code, and then build. In the build stage, the code first is analyzed, then we do unit testing to make sure the code could pass the test cases defined by the developer during development. In the end, the code will be built into Java ARchive file (.jar).

[25]The reason will be discussed in next section "Workflow in Production"

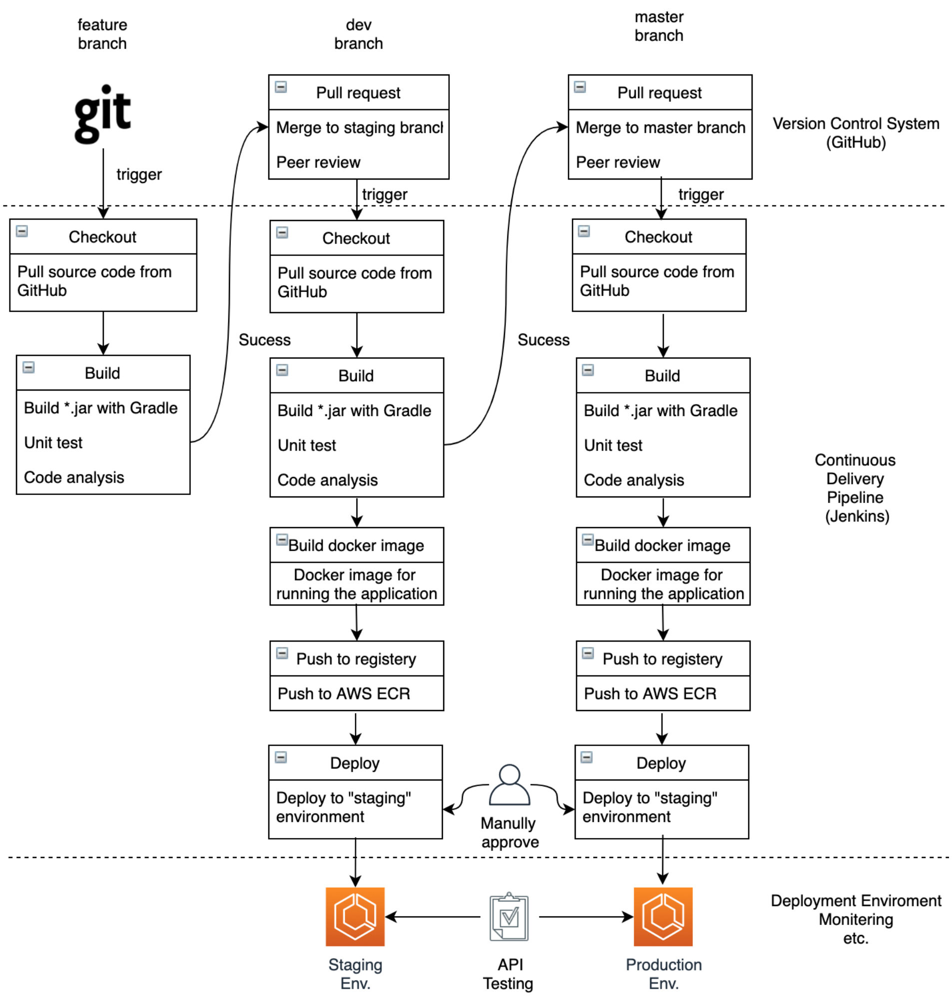


Figure 4.6: The Workflow of Continuous Delivery Pipeline in Our DevOps Tool-chain

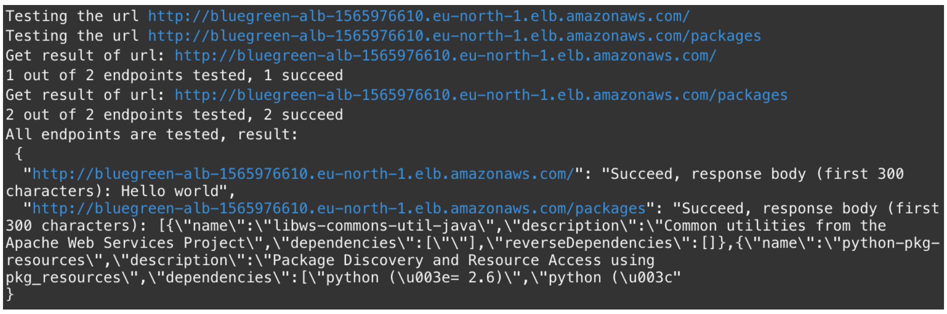
The purpose of putting code analysis step first is that the code analysis will check syntax error and bugs. We want to make sure the code is runnable, and no syntax error before put it into the build. So we can reduce the cost by lowering pipeline running time if there is error exists in the code.

If no error returns after finishing all the above steps, the developer can open a pull request view the code change and ready to merge the code to the dev branch. Before the merge, the pull request needs to pass the code review by another developer. Code review is to make sure that the automated tests do not miss any bugs. After the code review passed, the reviewer or the developer him/herself merge the code to the dev branch.

After the code merged to the dev branch, the pipelines run again, this time it runs the whole pipeline. First, the pipeline executes the first two stages as in the feature branch. Now we have the Java ARchive file. The Java ARchive is an executable package of our Spring Boot application. Next step is to Dockerizing our application which generates the Docker image our application. Then we push the image to the Amazon Elastic Container Registry AWS (AWS ECR) for further use.

The next step of the pipeline is deployment, the pipeline pull image in ECR that we pushed in the last stage, and then deploy it to the deployment environment in ECS with AWS CIL. The deployment strategy we are using is the rolling update. In the rolling update, we are gradually replacing instances in our deployment environment with the newer version of code.

In the dev branch, we deploy the application to the staging environment. The deployment to staging environment should be automated. This is because the staging environment is only for testing and only visible within the team. In the staging environment, we will conduct API testing (last stage shows in Figure 4.5) for test if our deployed API works and if it works as expected. The test is being done by trigger a Lambda function. The Lambda function sends test HTTP request to the deployed endpoints, and verify if the HTTP response is correct. When the test is done,



```
Testing the url http://bluegreen-alb-1565976610.eu-north-1.elb.amazonaws.com/
Testing the url http://bluegreen-alb-1565976610.eu-north-1.elb.amazonaws.com/packages
Get result of url: http://bluegreen-alb-1565976610.eu-north-1.elb.amazonaws.com/
1 out of 2 endpoints tested, 1 succeed
Get result of url: http://bluegreen-alb-1565976610.eu-north-1.elb.amazonaws.com/packages
2 out of 2 endpoints tested, 2 succeed
All endpoints are tested, result:
 {
  "http://bluegreen-alb-1565976610.eu-north-1.elb.amazonaws.com/": "Succeed, response body (first 300
characters): Hello world",
  "http://bluegreen-alb-1565976610.eu-north-1.elb.amazonaws.com/packages": "Succeed, response body (first
300 characters): [{\"name\":\"libws-commons-util-java\",\"description\":\"Common utilities from the
Apache Web Services Project\",\"dependencies\":[\"\"],\"reverseDependencies\":[]},{\"name\":\"python-pkg-
resources\",\"description\":\"Package Discovery and Resource Access using
pkg_resources\",\"dependencies\":[\"python (\u003e= 2.6)\",\"python (\u003c"
}
```

Figure 4.7: API Test Result (Jenkins log output)

If the deployed function passes the test, this shows the deployment works as expected and ready for the deployment. The developer could now open a pull request, merge code to master branch. The pipeline will runs again, and deploy the application to the production environment, which is visible to the customers.

4.2.6 Deployment Environment

We create a simple deployment environment with AWS Elastic Container Service and Elastic Load Balancer. Same with Jenkins agents, we use Fargate to host our containerized case project.

AWS Fargate allow us to run our containerized application without having to manage servers, makes it easier for us to build a functionality complete DevOps toolchain implementation. We choose ECS over EKS (Elastic Kubernetes Service) is because ECS is free of charge while EKS charges extra for the runtime of the

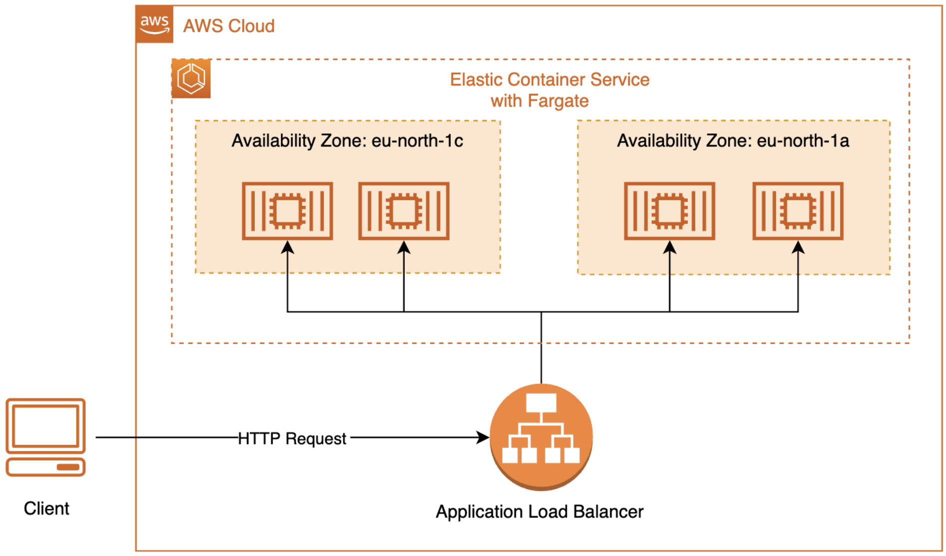


Figure 4.8: Deployment Environment

cluster. Compared with EKS, ECS also provides better integration with other AWS services, such as with AWS DevOps toolchain, and AWS CloudWatch monitoring.

Figure 4.8 shows our deployment architecture. The deployment region in Stockholm(EU-north-1). The Fargate instances in ECS cluster are automated scaled according to the number of incoming requests. To improve the availably of the product, we deploy the case project into two different availability zones within the region. When one availability zone is down, the load balancer can route the request makes sure the request can still reach the healthy availability zone. Besides the availability improvement, the load balancer also distributes incoming requests across Fargate instances which maximizing the resources rate within our ECS cluster.

4.2.7 Monitoring

Monitoring is one of the important components in the DevOps toolchain. Different from testing, which usually integrated with the continuous delivery pipeline, the monitoring is independent of the pipeline. Usually, monitoring does not act as one step within the continuous delivery pipeline but as an independent component.

In Chapter 3, we introduced AWS CloudWatch as one of the serverless services in AWS. In our toolchain, we will use it as the tool for monitoring the cloud infrastructure. With Cloudwatch, we not only can get the realtime log from our deployed container in the ECS but also the quantitative data, for example, memory utilization and network i/o, the monitoring dashboard can seem at figure 4.2.7

In the last paragraph of section 4.2.2, we mentioned that the CloudWatch is

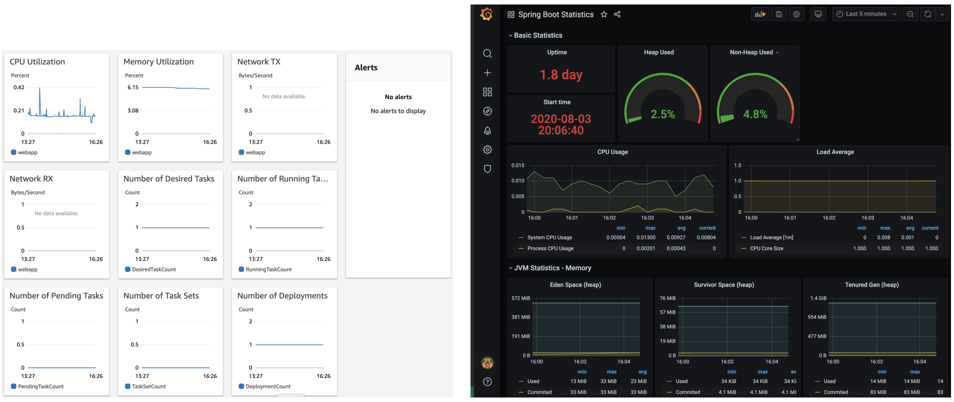


Figure 4.9: The Monitoring Dashboard of CloudWatch (Left) and Grafana (Right)

not capable to monitor what happens inside the application framework, in our case, Spring boot. Thus, we use Prometheus to gather the metrics within our Spring Boot application. Prometheus gather the realtime metrics with the help of Micrometer plugin of Spring boot. Micrometer exports all numeric metrics in realtime, these metrics includes, JVM statistics, CPU/RAM utilization, number of request to each API endpoint and the user's self-defined metrics within Java code. Prometheus records these metrics every second, and save the metrics in time series to Database. In addition to Prometheus, we use Grafana for read the metrics from Prometheus and display the metrics on the dashboard.

Another service we introduced in Chapter 3 is AWS lambda. It is the most important serverless service in AWS. We also discussed how could it be used in our DevOps toolchain in which monitoring is one of the use cases. In our monitoring system, AWS lambda is used as an extension for CloudWatch, and we use it for 2 cases.

Auto-Scaling the ECS Cluster with Custom Alarm in Cloudwatch As we mentioned in 4.8 Deployment Environment, The deployment could be auto-scaled by defining the auto-scaling policy within the ECS cluster. However, the scaling policy is not flexible enough; it only based on thresholds on certain metrics such as CPU utilization and memory utilization. Such auto-scaling in practice is: When the watched resources utilization is above/below a certain threshold, an alarm in Cloudwatch will be triggered. The alarm will further trigger the scaling event if the scaling policy was being set before.

Nevertheless, in real-life development, many projects are microservices architecture, rather than homogeneous architecture as we have in the case project. According to Luca Tiozzo's article [91], this means some service (container instance in ECS) could be CPU intensive while the others might be RAM intensive. In such a situation, with Cloudwatch alarm based scaling, we need two groups of alarm watching RAM and CPU, respectively. Nevertheless, the problem is, when the

ECS cluster lack of CPU resource but lack of RAM recourse, the CPU alarm is triggered then the ECS scaled up. Now the ECS cluster has enough CPU recourse, but it may have too much RAM resource so that it triggers the scale-in alarm in RAM. So the cluster will scale in again. This will cause the cluster to keep scaling up and back without finding and suitable size.

A good practice solves the problem is to use a single group of alarm that only triggered by single metrics. We can set an AWS lambda function that read different metrics and then aggregate it to a single custom metric. The software team determines the threshold and metrics according to the deployed project. Once the aggregated metric reach the threshold, the lambda function triggers an alarm that can trigger the scaling of ECS cluster.

Custom Project-Specific Metrics The second application scenario is related to the first one. The Cloudwatch has support on recourse utilization metrics. However, some metrics are project-specific and not related to resources utilization and performance. For example, the number of successful payment has been made in a payment service. In such a case, Lambda could fill the gap within the scope of CloudWatch. The team could set up a Lambda function which gets the number by monitoring the log with PutMetricData provided by CloudWatch. This Lambda can further forward the metrics to metrics analysis and visualization platform, for example, Grafana [26] to give the management team an overview of the KPIs.

4.3 Design of Integrated Serverless DevOps Toolchain

AWS provides a set of serverless DevOps tools which could help us build a completely serverless DevOps toolchain. We introduced these tools in Chapter 3. In the section, we introduce the design of Serverless toolchain based on DevOps tools of AWS. A certain part of the toolchain is the same with the non-integrated DevOps toolchain that introduced in the last section. Therefore we will not introduce these components again. Rather we only focus on how do we make use AWS DevOps toolchain. Figure 4.10 shows the general workflow of a project delivered by our integrated DevOps toolchain.

4.3.1 Continuous Delivery Pipeline with AWS CodePipeline

The workflow of our continuous delivery pipeline is the same as the pipeline in 2.2.2. Instead of Jenkins, which is server-based, we build the pipeline with AWS CodePipeline. Figure 4.10 shows the activity within the CodePipeline in a single graph. Different from Jenkins who can do the whole continuous delivery process solely with the help of plugins, the CodePipeline just provides a platform which the development team can configure a workflow with AWS DevOps tools or other third-party tools.

[26] https://grafana.com/grafana/

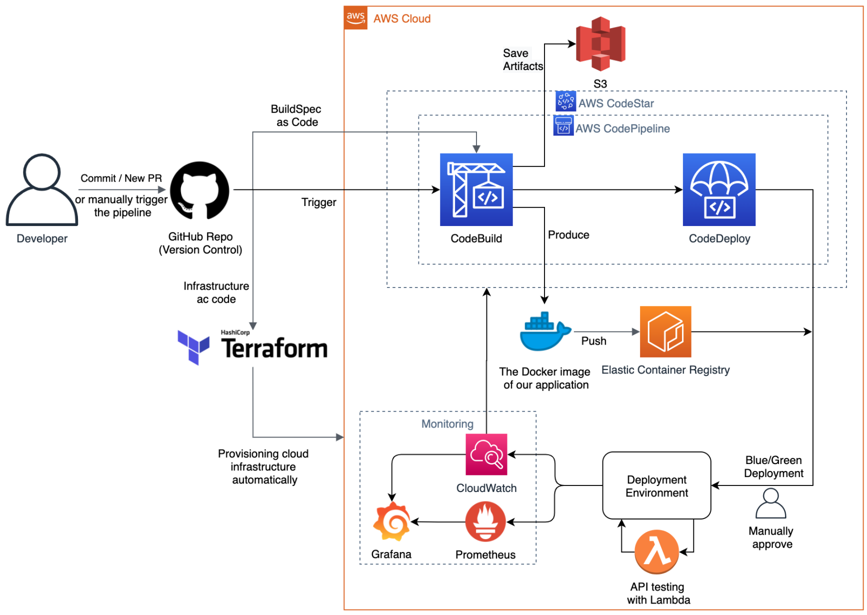


Figure 4.10: Integrated Serverless DevOps Toolchain

Same within the non-integrated version, we used GitHub as the version control system, and it has the same role as it has in the non-integrated toolchain. Although AWS also provides version control solution which is CodeCommit. It still lacks the functionality of collaboration compared with GitHub. Also, GitHub is already a serverless solution with good integration with AWS DevOps tools, so we do not need to change our version control system away from GitHub.

In the next step, we are using AWS CodeBuild, which we introduced in Chapter 3. AWS CodeBuild does the same procedure as in Jenkins pipeline. It does code analysis, unit test and builds the Java application with Gradle, build the Docker image of the application and push to the ECR. Different, the cloud deployment deploys our application to ECS with Blue-and-green deployment strategy.

The implementation of continuous delivery in CodePipeline is straightforward compared with Jenkins. In Jenkins, without the help of the plugin, the pipeline workflow can only be defined by groovy code, while CodePipeline natively provides a graphical user interface for workflow modelling. In our implementation, we just simply add each step with the graphical interfaces in CodePipeline.

4.3.2 Build and Test with AWS CodeBuild

Same with the design of our Jenkins pipeline, AWS CodeBuild also executes the build within Docker container. The image of the Docker container provided by AWS already contains environment for the build of different programming languages. It also includes the Java environment and Gradle which needed by our case project. Therefore we could save time in setting up the pipeline since we do

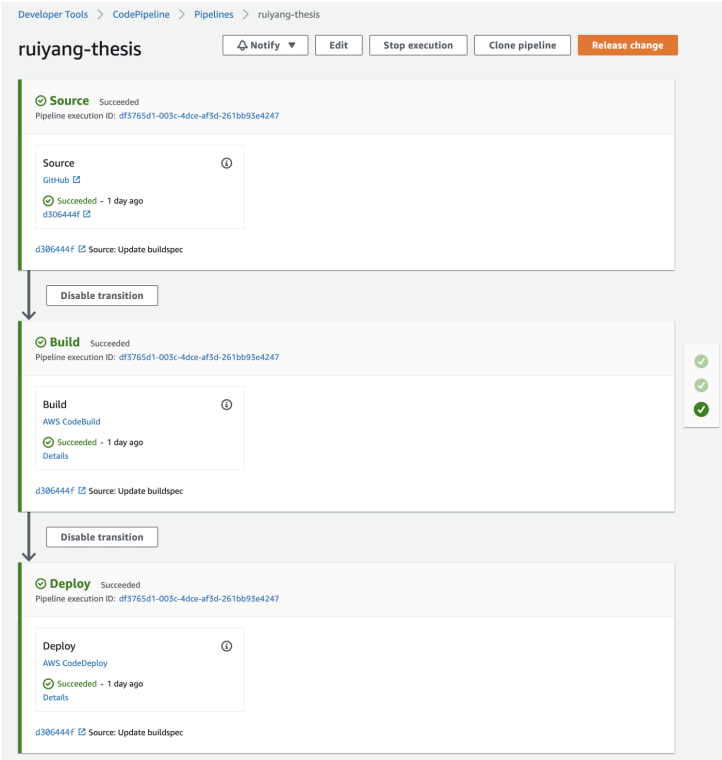


Figure 4.11: Our Workflow in CodePipeline

not need to define the Docker image for the build by ourself.

As we mentioned in 4.3.1, the process within CodeBuild is the same as we have in Jenkins before the stage "Deploy". We will not describe the process again here. Same with Jenkins, the workflow of CodeBuild is defined in a YAML configuration file. The only difference in term of build workflow CodeBuild is that we store the build artifact to S3, the build artifact is the configuration file defines the deployment configuration in CodeDeploy. This is because CodeDeploy requires the deployment configuration from the step before it to run automatically.

4.3.3 Blue/Green Deployment with AWS CodeDeploy

One of the advantages of AWS DevOps tools is good integration with other AWS services. During the design and implementation of our toolchain, this advantage shows in the deployment to ECS with CodeDeploy.

In Jenkins, there is a lack of specific plugin that helps us deploy the project into ECS or EKS. Thus we have to deploy our project to ECS with AWS command-line interface(CIL). The problem with AWS CIL is that it only supports the most basic deployment strategy, which is rolling update deployment. The rolling deployment strategy is to replace the old code running on the instances with new code gradually, instance by instance.

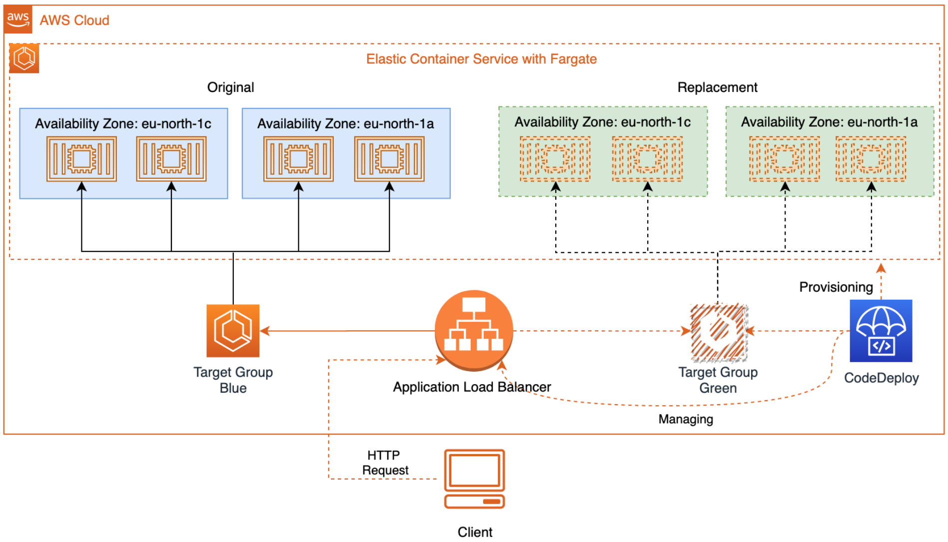


Figure 4.12: Blue and Green Deployment for Our Deployment Solution

In real-life production, the team would like to make sure the deployment is reliable with minimized downtime. Thus the safety is highly valued in deployment strategy. In answering this need, a strategy called blue/green deployment, which is now widely used in the industry. AWS CodeDeploy natively supports the blue/green strategy. A blue/green deployment is a deployment strategy that requires two sets of totally identical deployment environment that runs the new and original version of code respectively, while the load balancer is gradually routing more incoming requests to the environment that runs the newer version of code.

Figure 4.12 shows the visualisation of blue/green deployment. It also shows our design on how to implement blue/green deployment with CodeDeploy (shown before in Figure 4.8). CodeDeploy controls routing policy within the load balancer. When new deployment comes, CodeDeploy does the following steps:

- Provisioning new identical deployment environment (replacement environment) and deploy a newer version of code on it. In ECS, the deployment is called ”task set”.
- Control the load balancer, rerouteing incoming traffic gradually to replacement ECS task set. We set the rerouteing rule as 10% per minuets. We do not rerouteing all traffic at once to ensure the service will not fully down if the new deployed task set not works properly. This minimizes the downtime of our deployment.
- Wait for 5 minutes after rerouteing is done. During the rerouteing and this 5 minutes, the load balancer keeps doing the health check to the new deployment by sending a request to the health check API endpoint. CodeDeploy

read the health status from the load balancer. If the replacement tasks set is un-healthy, CodeDeploy does rollback by rerouteing incoming traffic back to origin tasks set.

- If the new deployment is still healthy after 5 minutes of waiting time, CodeDeploy terminates the origin tasks set. Now the whole deploy process is done.

Compare with the rolling update we are using in Jenkins pipeline, the better safety of blue/green deployment reflected at, when the error happens with a newer version of code, we can immediately roll back to the last version by switching the rerouteing to Blue [92] without redeploying the previous version. Under the same circumstance, the rollback with the rolling update is nevertheless taking a too long time, since we have to replace the already deployed code to the previous version. This could cause longer downtime of the server.

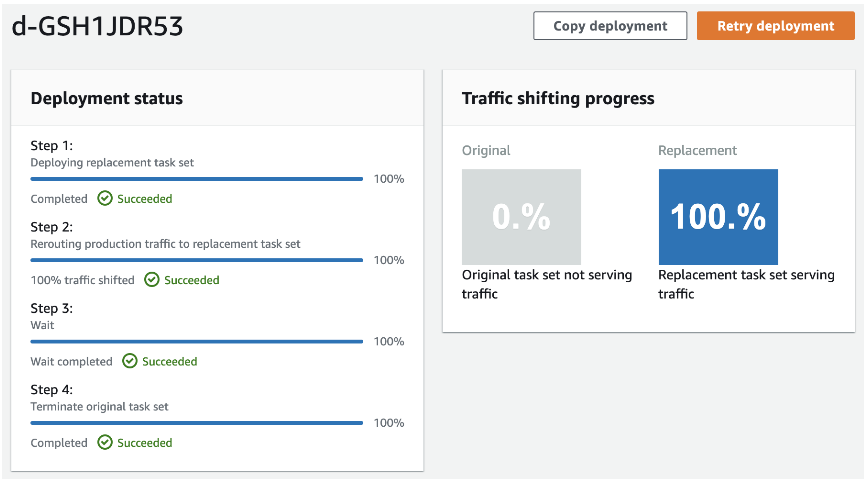


Figure 4.13: CodeDeploy Dashboard

The better integration of CodeDeploy with the rest of AWS also brings benefit in the monitoring of the deployed solution. Aside from the existing monitoring with CloudWatch, CodeDeploy also provides us with a dashboard to show the deploy progress and the traffic rerouteing process. Figure 4.13 shows the dashboard of CodeDeploy that shows the status of our case project during the deployment.

In comparison, with our non-integrated toolchain, we can not easily do the blue/green deployment. A possible solution is to set up an AWS Lambda function which will be triggered after the deployment. The invoked Lambda function control the load balancer to gradually redirect the traffic from the previous deployment to the new deployment. At the same time, the second lambda function continuously read the health status of the new deployment from the load balancer, and trigger the rollback if the new deployment is unhealthy. Besides all these, a dashboard is needed for developer to monitoring the rerouteing status. On the contrary, in CodeDeploy, all the above tasks are being taken care of.

4.3.4 Integration of AWS DevOps Tools using CodeStar

In 4.3.1, we introduce how do we integrate different stages in the continuous delivery pipeline (CD pipeline) with CodePipeline. However, an integrated continuous delivery pipeline is not called integrated toolchain. The DevOps toolchain is centred with the continuous delivery pipelines, but the pipeline is not all of the toolchains. To integrate other AWS tools with the CD pipeline and get the integrated toolchain as a single application, we use CodeStar.

Figure 4.14 show the user interface of the CodeStar dashboard. We can see, besides the CodePipeline, the CodeStar also integrate tools like monitoring, project management, and version control and then compose our integrated toolchain. In conclusion, the AWS integrated toolchain includes the following tools:

- **CodeStar**: Integrate below tools into a signal toolchain. With project management functionality. The project management could be extended by integrating with third-party tools, i.e. Jira.

- **CodePipeline**: Modelling Continuous Delivery Workflow, integrate tools that used within the CD pipeline.

  - **CodeBuild**: Build code of the software project.
  - **CodeDeploy**: Deployment and monitoring.
  - **GitHub**: Version control.

- **CloudWatch**: Monitoring deployed solution.

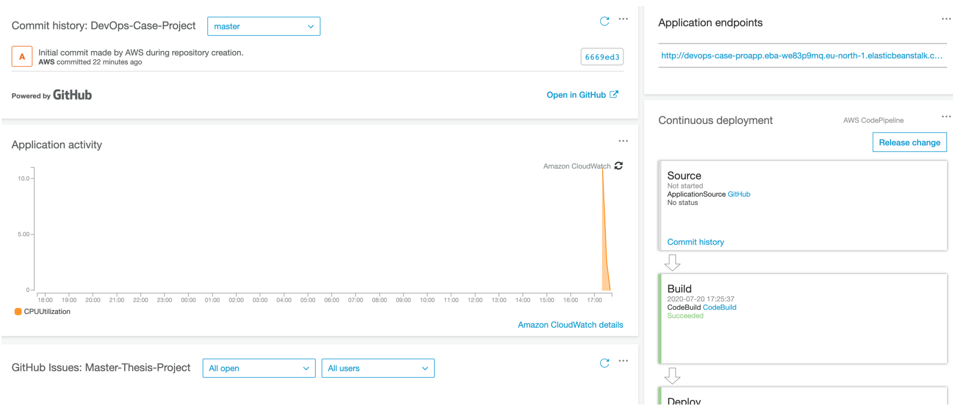


Figure 4.14: CodeStar Dashboard

4.4 Comparison between Integrated and Non-integrated Toolchain

In this section, we discuss the difference between these two kinds of toolchains. The scope of comparison will be limited within the scope of functionality and ease of implementation. Furthermore, it is only based on our experiences with the tools used in our implementation. We summarize the difference between these two toolchains as in Table 4.2. We will do more comparison related to the performance and cost in Chapter 5.

	Non-Integrated Toolchain	Integrated Toolchain
Open-source	Open-source solution existed	No, usually hosted commercial solution
Delivery method	Each part is a stand-alone tool either hosted or on-promised, depends on the tools selection	As a single cloud hosted software
Implementing time	Long	Short
Operational effort	High	Low
Visibility on status	Depends on tools, a well-integrated toolchain could gives good overview on the whole toolchain.	Easy to see the status as a whole without additional implementation effort, low visibility on under-laying server since it's hosted solution
Extendibility and tool selection freedom	Free to select tools for each part of the toolchain.	Limited integration with third-party tools

Table 4.2: Comparison of continuous delivery tools

4.4.1 Implementation and Cloud Deployment

The cloud based integrated toolchain is delivered as hosted solution in a serverless model. However we noticed that the non-integrated toolchain could be also completely serverless if we using hosted tools for all the components.

For example in our solution, we only have continuous integration pipeline which is on-promised and need to be deployed to the VM manually. If we replace Jenkins with some other hosted tools, for example, Travis CI[27] we can actually build a fully hosted but non-integrated DevOps toolchain. But, for following reasons, it is not a satisfactory solution thus a non-integrated toolchain usually has some on-promised modules that needs operational effort and cloud knowledge.

- The hosted tools, especially tools for continuous integration pipeline, are all closed-source commercial solution. This means there is no community support like in Jenkins, and it can usually integrates with the certain tools that supported by vendor. Besides, commercial user always need to pay for these hosted tools.

- Hosted tools runs in the vendor's server, and it requests user log in to use this tool which brings extra integration difficulty. This means for two hosted toolchain to integrated with each other, it not only need to do the integration in the data transfer, but also need to connect their account system, for example, with OAuth. This extra inconvenience makes most hosted DevOps tools only do the integration support with other most most popular tools which largely limited the extendibility.

Therefore, a non-integrated toolchain usually has some on-promised module in real-life use. In our deployment process, we find it requires lot of work to put an on-promised tool to cloud, especially if the developer is not familiar with the cloud platform that deploy this tools. For only deploying Jenkins we need to do following steps:

1. Create a cloud virtual machine (EC2 instances) for hosting the Jenkins master.

2. Setup IAM role for Jenkins master VM, make sure it has access to other AWS recourse that needed during build.

3. Setup security group and networking for the VM, makes sure it can be accessed form the internet but only accessible within company's IP range, and only port needed are opened to the public.

4. Install Jenkins in the VM. Research what plugin is needed and install necessary plugins.

5. An tedious set-up process for setup Jenkins cluster that supports the distributed build. This includes setup ECS cluster for build agent. Although Terraform makes the provision of cloud resources easier, still, prior knowledge for AWS is needed. The experiences in AWS is also needs to correctly configure ECS cluster that maximizing the performance of build agents.

6. Develop Docker image for the container that runs Jenkins agents.

[27] https://travis-ci.org/

7. Setup integration with other tools in toolchains by finding correct plugins and configure these plugins.

Only after these steps, we can start using Jenkins as part of our toolchain. In comparison, the core feature of the integrated toolchain in AWS is an out-of-the-box feature which means there is no previous cloud knowledge needed, and there is no deployment and environment configuration required before we use it. We are free from all the steps we mentioned above.

4.4.2 Extendibility and Flexibility

The integrated toolchain is a hosted platform that runs by a vendor. Similar to we mentioned in 4.4.1, We find all the currently integrated toolchain are all commercial and closed-source which has no community support. So the integration of their third-party tools usually only limited to popular tools. For example, AWS DevOps tools only support 21 tools within it is "DevOps Partner Solutions" [28]. If the user wants to use anything except these tools, it is not possible.

Nevertheless, different with the single hosted tools we mentioned in 4.1.1, a hosted integrated DevOps toolchain mostly has everything needed for DevOps lifecycle, so it is not mandatory for it to able to integrates with third-party tools. Still, the limitation in third party tool support might make the software team facing trouble when they want to use certain tools which are not very supported.

A non-integrated toolchain allows the software team to pick any tools for each component, as long as those tools can be integrated. The tools in the toolchains could also be open-source, which allow so the software team modify the tools according to their need. For example, develop a plugin for Jenkins that allows the integration of internal company tool with Jenkins.

In conclusion, in terms of extendibility and flexibility, non-integrated toolchains are better than integrated toolchain.

4.4.3 Integration Between Tools

As we mentioned in 4.4.1, sometimes it is hard for tools within a non-integrated toolchain to integrate, especially between the hosted tools. During our implementation, we also realized that, first, it requires some configuration work for tools to be able to work together. Secondly, sometime the Integrated could be buggy, for example, in our toolchain, the Jenkins sometimes does not react to the build triggering signal from GitHub. This means the software teams need further maintaining of the toolchain. In integrated toolchain, the toolchains are delivered as a single cloud-based software, each part of naturally coped with each other, which makes integration much more straightforward. The better integrating between each component also makes it easier to monitor the toolchain as a whole.

[28] https://aws.amazon.com/devops/partner-solutions/

4.4.4 Visibility

In 4.4.3, we mentioned that the integrated toolchain is easier to be monitored as a whole, however, when comes to every single component, in our implementation, we find out that integrated toolchain is lack of visibility. We met two difficulties when test two toolchain. The first one is that Jenkins master was having difficulty in the provision and connect to the agents. Since Jenkins is a web service deployed in our EC2 virtual machine, We solve the problem by reading the Error message within the Jenkins log file. The second problem we met was within AWS CodeDeploy. The CodeDeploy failed to deploy the case project to the ECS cluster. We could not find the reason at that time since we cannot find the log of CodeDeploy anywhere since it is not shown in the web interface, and we have no access to the underlying cloud infrastructure either. The lack of visibility is a problem with all hosted serverless services since the users do not have the visibility to the infrastructure behind the service.

4.5 Challenges in Implementation and Design of DevOps toolchains

In this section, we discuss the challenge that we met during the implementation.

4.5.1 Challenge I: The Enormous and Unregulated Jenkins Plugins System

Jenkins has more than 1600 plugins which brings the software an amazing extendibility, which is one of the main advantages of Jenkins. However, there are two problems with Jenkins' plugins; First, there are usually more than one plugins that have the same functionality, for example, there are at least five different plugins related to running Docker container as Jenkins agent. Second, most of the plugin is developed by the open-source community, so the quality of these plugins is not ensured. During our implementation, we find out there are two plugins that support run Jenkins agent in ECS cluster. However, we find only one work after we tried both plugins. Besides, the documentation of Jenkins plugins sometimes is abysmal. For example, the documentation of the plugin that we use for Jenkins agent is too brief to tell us how to use the plugin, and it is not even mentioned the security setting needed in Jenkins master node that allows agents to connect the master node.

As a result of the above three factors, we spent a very long time in selecting and configuring tools. Moreover, trying to solve the problem which nether mentioned in the documentation and on the internet.

4.5.2 Challenge II: Fargate Does not Supports Container runs in Privileged Mode

As said per title, this is some limitation in AWS Fargate for preventing container get permission to access the critical resource on the host(underlying server hosting Fargate instances in this case). As a result, we cannot use Docker within a Docker container that runs on Fargate. This makes it impossible for us to distribute the ”Build docker image” and ”Push to ECR” stages to agents. Instead, we have to run them on the master. Luckily, these two stages take a short time (¡5s in total), so this limitation will not slow down the pipeline too much when multiple builds run in parallel.

A possible solution solving this problem is to runs these two stages in AWS CodeBuild, AWS CodeBuild has support to Jenkins, which allow us to run certain Jenkins stages in CodeBuild. Moreover, CodeBuild supports fully parallel execution as in Fargate.

4.5.3 Challenge III: Slow Starting Time for Agents in AWS Fargate

On average it takes around 60s from sending a Jenkins job to agent, to running the job in an agent. To our case project that takes 90 seconds to go through the whole pipeline, this is a relatively long time. During this 60s, Jenkins master node sends task definition [29] to ECS, provision a Fargate instance within ECS, then pull the image we developed for Jenkins agent, star the agent container within Fargate and then connect to the Jenkins master. This challenge is due to the nature of the serverless computing that we discussed in Chapter 2, and we believe there is not an economical way to solve the challenge with the current setup.

4.5.4 Challenge IV: No Enough Visibility in AWS DevOps tools

As we mentioned in 4.4.4, the lack of visibility to the underlying process, especially in CodeDeploy, caused some obstacle for us to debugging our pipeline. There was a problem when we try to deploy the case project to the ECS with blue/green deployment, and the CodeDeploy sucked in the creating replacement service. We know there was something wrong within either the configuration of ECS, load balancer, health-check or security/network setting. Nevertheless, there is no log output of the underlying deployment process in CodeDeploy. In the end, we have to check everything that might be caused problem one by one and found it was the problem with failed health-check, which was very time-consuming.

[29]Define specification of a container runs in ECS.

Chapter 5

Performance Comparison and Evaluation

In this chapter, we describe our experiments regarding two research questions we proposed in chapter 1. The experiments based on the DevOps toolchains we implemented in Chapter 4.

In Section 5,1, we will examine how does the serverless compute engine for containers (Amazon ECS on AWS Fargate) could influence the performance of non-integrated toolchains. Thus, in the experiment, we implement the solution with a different type of cloud environment (with/without serverless) as a comparison group. In section 5.2, we focus on answering research question 2, in which we will compare the performance of continuous delivery pipeline composed of fully-managed serverless DevOps tools in AWS with our Jenkins-based pipeline that runs on the virtual machine.

5.1 Experiment 1: Experiment on Serverless Container Services

The Docker agent has already been supported by many CI/CD tools, for example container job[1] in Azure DevOps[2], Docker agent in TeamCity [3], Docker agent[4] in Jenkins and docker runner[5] in Drone[6]

The serverless container services in AWS (AWS Fargate) provides the possibility to ease the infrastructure management task for the Docker build agents. This experiment is a controlled experiment which examines whether serverless container service could improve the continuous delivery pipeline from various perspectives.

[1] https://docs.microsoft.com/en-us/azure/devops/pipelines/process/container-phases
[2] https://azure.microsoft.com/en-us/services/devops/
[3] https://www.jetbrains.com/help/teamcity/build-agent.html
[4] https://www.jenkins.io/doc/book/pipeline/docker/
[5] https://docs.drone.io/runner/docker/overview/
[6] https://drone.io/

5.1.1 Test Task and System Description

In this experiment, we run the continuous delivery process of a Spring Boot web application with our DevOps toolchain. From the experiments, we could verify our assumption in chapter 3 and better-answering RQ 1.

As we described in Chapter 4, the continuous delivery pipeline includes the following steps:

1. *Checkout*: Pull the most recent change from GitHub repository
2. *Build*: Build the application with Gradle, with automating testing with JUnit integrated into Gradle.
3. *Build the docker image*: Build the docker image of our Spring Boot application.
4. *Push to Container Registry*: Push the docker image from the last step to the AWS elastic cloud registry (ECR) for further deployment.

In these four steps, the step ”Build”, and ”Checkout” is being done in parallel within the ECS cluster. As we mentioned in CH4, when the new job started in the Jenkins master server, Jenkins will provision a new container instance within the ECS cluster. The container is managed directly by AWS, so we don’t need to create and manage the virtual machine that runs the container. We use this setup in our initial implementation as the control group.

In the experimental group, we replace AWS Fargate with traditional VM, which is EC2 in the Amazon Web Services. The parallelisation pattern remains the same; this means as in the control group, only the first two steps are being run distributively in the Jenkins nodes. The EC2 instances belong to an auto-scaling group that will scale up when CPU Utilisation rate reach 70%. The initial size for an auto-scaling group is 1.

Figure 5.1.1 shows the architecture of 2 groups in this experiment. The experimental group on the left is a Jenkins server with the traditional virtual machine as workers node that hosting the container agent. The architecture of the control group on the right has agent nodes dynamically provisioned as serverless containers hosed by AWS Fargate.

Hardware

The hardware of the instance that runs Jenkins agents is the independent variable that exposed to the change in the experiment.

The experiments are conducted on Amazon Web Services (AWS). The hardware of Jenkins master node in both experiment groups is the same, which is EC2 instance of type t3.medium with two virtual CPU, 4 GB RAM and 30 GB disk. The EC2 instances as worker node are type t3.small, with two virtual CPU and 2GB RAM. Each EC2 instance can run one container at the same time.

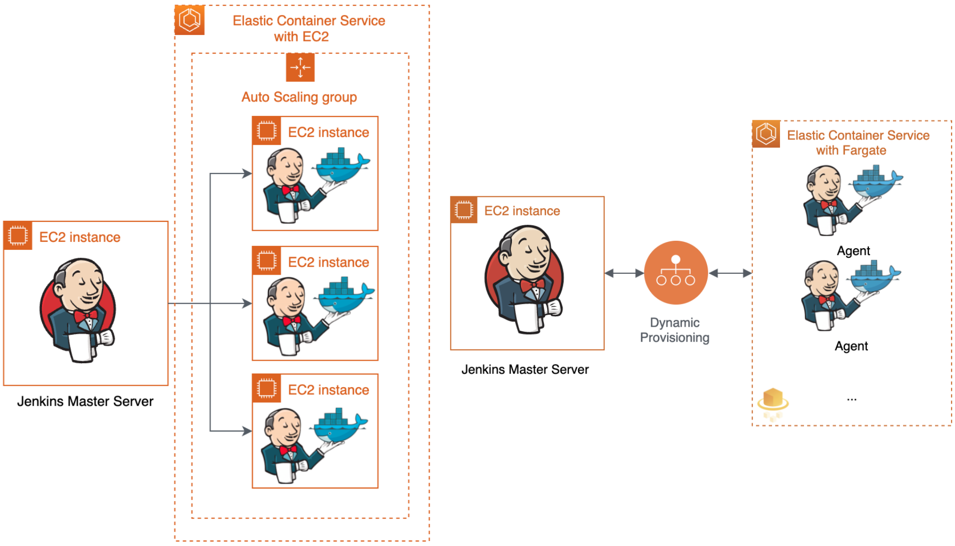


Figure 5.1: Architecture diagram of the test Jenkins cluster with agents running in traditional virtual machines (left) and on ECS with AWS Fargate (right)

In the control group, which is the implementation we presented in CH4, the Jenkins agents run on AWS ECS powered by AWS Fargate. The virtual hardware resources that are allocated to each serverless container is two virtual CPU, and 2 GB of RAM. The identical hardware setup between to groups makes sure that each container shares the same hardware resources as in another group, so the hardware will not affect the result.

Software

We maintain the same software setup in each group. The operating System for EC2 instance that runs Jenkins master node is Ubuntu Server 18.04. The version of Jenkins that runs on the server is 2.222.3. For connect ECS and Fargate which works as the Jenkins agents, we use Jenkins plugin ”Amazon Elastic Container Service (ECS) / Fargate”, version 1.34. The container in Fargate/EC2 for running the Checkout and Build steps is from our developed docker image, which can be seen at [7]. The docker image includes essential dependencies that will be used to build the Spring Boot application. It’s the base image include a program which allows container connects Jenkins master as an agent. The ”Build” step in our pipeline uses Gradle (version: 6.2.1) as the build tool for the application, with OpenJDK 1.8.0.252 as Java virtual machine (JVM). This step also includes the automated testing and code analysis, as plugins of Gradle.

To shows how does the 2 setups performance within the teams with different sizes, we run by run the different number of tasks parallel through the pipeline.

[7] https://hub.docker.com/r/dry1995/jnlp

This scenario simulates the different team size and shows the scalability when it comes to the need for task parallelisation in bigger organisations. It also simulates the DevOps process of a microservices software project, which could have multiple jobs for different service runs in parallel.

5.1.2 Performance Properties and Evaluation

We run the pipeline through 2 different setups, we will get the result of the following properties:

- *Runtime* describes the total time for finishing all the jobs. If the jobs run in parallel, the runtime is from the start of jobs until the end of the last finished job.
- *Cost Structure* describes the daily cost of 2 setups under the same workload, within the same period.
- *Resource Utilisation* describes the average CPU/RAM usage for each instance during a single run of the pipeline.

5.1.3 Result and Evaluation

Here shows the result of this experiment. We also evaluate our experiment result by analysing the factors that lead to the results.

Runtime

We first compare the runtime of these 2 setups. Except for test the runtime of single job runs with two setups respectively, we also test the runtime of each pipeline setup under different number jobs executed in parallel. Figure 5.2 shows the test result.

The test result shows that when it comes to the execution of a single task. The traditional VM has a faster delivery speed over serverless solution (AWS Fargate). However, with the number of jobs that run in parallel increases, the total runtime on the traditional VM decrease. On the contract, on the serverless solution(Fargate), the runtime remains almost the same.

We analyse the reason behind this result, and we found out that the longer runtime with the single job on Fargate is because the longer starting time of Jenkins agent. In EC2, the Jenkins will simply provision a Docker container within EC2 VM, and connect to the Jenkins master node. However, in Fargate, the Jenkins can only connect to the agent once AWS finishes the initialisation of underlay infrastructure that runs the serverless container, which takes a significantly longer time. The slow provisioning shows one of the limitations of serverless computing (cold-start) that we mentioned in 2.4.3.

When it comes to parallel job execution, the performance of Fargate is significantly better. The performance difference is because, in Fargate, each container runs

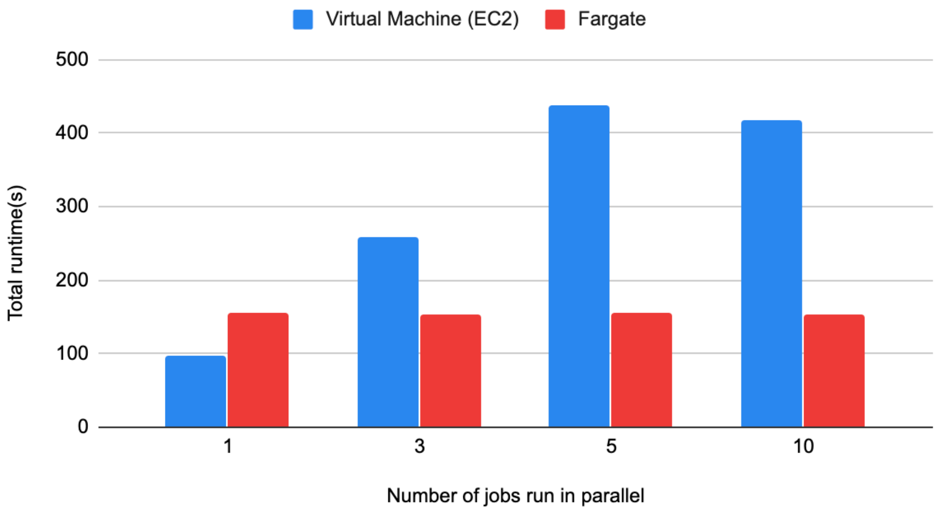


Figure 5.2: Runtime of Pipeline with Different Jenkins Agents

on an independent instance on AWS's infrastructure. Therefore, AWS provisions one Fargate instance for each running job. The independence between Fargate instance ensures the agent will not compete for the resource. On the other hand, when we run multiple agents in our EC2 instance, due to the limitation of resource, part of running jobs has to wait until the resource on EC2 instance available until they can start the execution. To further investigate the reason for the result, we observe the parallel execution mode when it runs three jobs in parallel. Figure 5.3 shows the execution modes. We find that the easily scalable character (mentioned in 2.4.3) helps the serverless suites better with the parallel task. The long wait time is the reason that makes the total runtime in EC2 much longer.

We also notice that in Figure 5.2, when the parallel task reaches 10, the EC2 runtime becomes shorter. The shorter runtime of EC2 is because we set auto-scaling for our EC2 instance. So in the later part of our experiment, the EC2 scaled from 1 to 2 and then to 3. Nevertheless, even with 3 EC2 instances, only three jobs are allowed to run in parallel, while in Fargate is easy to have 10 jobs runs in the complete parallel method. This because the scaling of EC2 VMs is much slower because it is heavier to create a VM than create a new Fargate instance. The other reason is the auto-scaling of EC2 VM is triggered by reaching certain resource utilisation threshold, while in Fargate is based on one instance per container(Jenkins agent). Even we set the scaling policy of EC2 to a more aggressive pattern, the AWS still more "hesitate" to create new instances compared within the Fargate.

Resource Utilization

We compared the average resource utilisation within containers that runs Jenkins agent in two setups. The data from the AWS cloud watch shows that the resource utilisation rate is similar in these 2 setups. The similarity is because the "run in

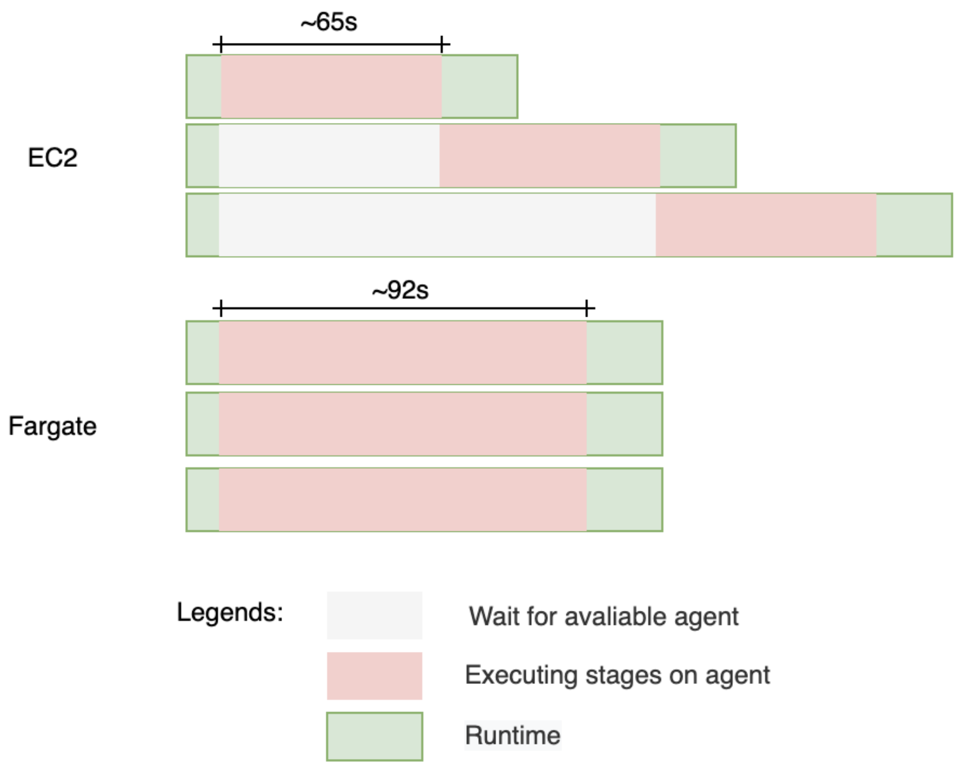
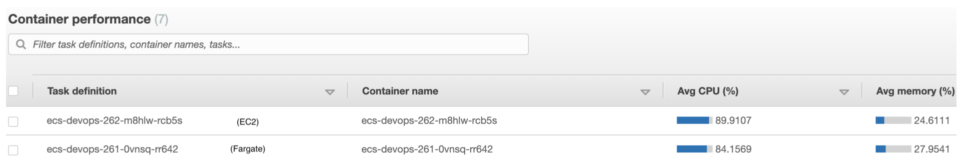


Figure 5.3: Execution Mode of Pipelines with Different Jenkins Agents

the same way regardless of the host environment" [82] feature of Docker container that we mentioned in 4.2.2.


Container performance (7)

Filter task definitions, container names, tasks...

Task definition		Container name	Avg CPU (%)	Avg memory (%)
ecs-devops-262-m8hlw-rcb5s	(EC2)	ecs-devops-262-m8hlw-rcb5s	89.9107	24.6111
ecs-devops-261-0vnsq-rr642	(Fargate)	ecs-devops-261-0vnsq-rr642	84.1569	27.9541

Figure 5.4: The comparison of Resource Utilization

Cost Analysis

The container orchestration service ECS itself is free of charge. We only pay for the resources we are using, which is Fargate or EC2 virtual machine.

In AWS Fargate we pay only by resource we are using and the runtime. The price for Fargate service in EU(Stockholm) is $0.004165 per GB RAM per hour plus $0.0049 per vCPU per hour [8]. Thus, in our experiment setup (2vCPU, 2 GB

[8] https://aws.amazon.com/fargate/pricing/

RAM), the price should be \$0.01813 per running agent for one hour's runtime. In AWS EC2 our cost depends on the type of VM we are using. The type of VM we use for running Jenkins agents is t3.small that costs \$0.0216 per hour [9]. The Fargate-based Jenkins agent is cheaper than EC2-based agent.

The pay-per-use characteristic of serverless service makes Fargate even more competitive in terms of cost. The instance only up and run when Jenkins master distributes the job. On the job finished, the AWS will terminate Fargate instance immediately when it finds no container is running on it. However, EC2 not so flexible due to the resource-utilisation-based scaling policy. The EC2 instance is not scaled in immediately after job finished, and the user has to pay for the instance runtime before it gets terminated – even there is no job running in EC2 instance any more.

5.1.4 Conclusion

In this experiment, we compare the serverless (AWS Fargate) and non-serverless solution (EC2) for hosting distributed continuous delivery pipeline. The experiment shows that the performance of the serverless solution is worse when it comes to single job execution, which is because of the cold-start issue with serverless computing. However, in terms of the parallel job executing, the serverless solution has better performance over traditional VM. The cost of the serverless solution is cheaper in AWS, and the resource utilisation is similar in both solutions.

5.2 Experiment 2: Experiment on Integrated DevOps Toolchain

For solving the RQ2, we compare the implementation of our design of two different the DevOps toolchain – the non-integrated Jenkins based toolchain and the AWS DevOps toolchain implementation with AWS DevOps tooling.

5.2.1 Test Task and System Description

This second experiment is similar to the first experiment, and we deliver our case project through two DevOps toolchainst. The first toolchain is our Jenkins-based toolchain in 4.2, with agents runs on AWS Fargate. The second toolchain is the toolchain with AWS DevOps tooling that we described in 4.3.

During the experiment, we have set up as follows:

Hardware AWS DevOps tools allow us to select the hardware configuration for underlying computing resources. The configuration we chose is two virtual CPU and 3 GB RAM. The Hardware configuration for Jenkins build agent is two virtual CPU and 4 GB RAM. We cannot set the RAM to 3 GB since AWS Fargate is not

[9] https://aws.amazon.com/ec2/pricing/on-demand/

allowing [10] RAM to below 4GM when using two virtual CPU. However, we still allowed to set an additional software RAM limit to the build agent runs in Fargate agent. Thus, we limit the RAM that a running build agent to 3 GB, which ensure both setups have identical hardware configuration.

Software The version of Jenkins that runs on the server is 2.222.3. We use Jenkins plugin" Amazon Elastic Container Service (ECS) / Fargate", version 1.34 for connecting ECS and Fargate which works as the Jenkins agents. The Jenkins cluster has the same configuration as the last experiment. The built environment in both setups is the same, with Gradle (version:6.2.1) and JVM version is OpenJDK 1.8.0.252.

5.2.2 Performance Properties and Evaluation Criteria

We run the pipeline on these two different setups, we will get the result of the following properties:

- *Runtime* describes the total time for finishing all the jobs. If the jobs run in parallel, the runtime is from the start of jobs until the end of the last finished job. AWS monitions it's parallel execution ability in the document[11][12] of both CodePipeline and CodeBuild, thus we will also validate this ability by analysing the parallel executing pattern of AWS integrated toolchain.
- *Cost Structure* describes the daily cost of 2 setups under the same workload, within the same period.

The resource utilisation comparison is not available in the experiment since we cannot get the resources utilisation in underlying hardware resource when running AWS DevOps tools.

5.2.3 Quantitative Experiment Result and Evaluation

This section shows the result of our experiment. We also give an evaluation and analysis of the reason behind the result.

Runtime

As in Experiment 1, we compare the runtime of each toolchain under a different workload [13]. Due to the reason that different deployment mode is being used by two toolchains which has a significant difference in terms of the speed, we only compare the runtime without the final deploy stage.

Figure 5.5 shows the result of this experiment. From the graph 5.6, we can

[10] https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-cpu-memory-error.html
[11] https://aws.amazon.com/codepipeline/features/
[12] https://aws.amazon.com/codebuild/features/
[13] workload means the different number of jobs runs in parallel in both pipelines

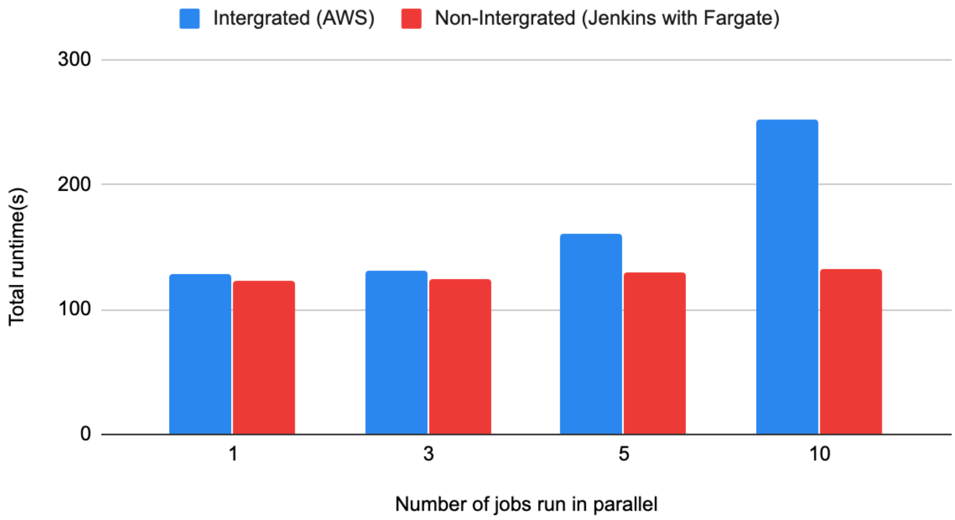


Figure 5.5: Runtime of the Pipeline with on Different Toolchain

see that during the execution of a single job, tow toolchains have similar runtime. The similar runtime is because the build stage, which takes most of the runtime, is running in a similar environment in both toolchains. Both execution environment is within a Docker container runs with the same hardware in AWS. Although, the Docker image for the container is different in two toolchains, and we are not sure what if the actual hardware is the same since we have no visibility to the hardware in both toolchains. However, still, the performance is not so different in both toolchains.

We further observe the time allocation between stages within the runtime in both toolchains. The figure shows our observation. To get the runtime in Figure 5.6, we run pipeline in each toolchain for five times and get the average runtime of each stage. The first difference shows in Figure 5.6 is that the Git Checkout stage in AWS integrated toolchain does not run on the container build agent. Instead, it is running in the unknown environment fully managed by AWS.

We observed that, during this 10 seconds of Git Checkout stage in our integrated toolchain, the actual Git checkout only takes around 5 seconds, while the AWS toolchain does not do anything in the first 5 seconds. We presume that AWS provisions environment for runs the Git checkout stage in this 5 second, so this stage is also running in a serverless environment within AWS. However, we are not allowed to configure anything in this environment. The second difference is that in AWS integrated toolchain, the provisioning of the build environment is much faster. We presume this is due to AWS manages both build agent and this toolchain, and AWS is optimising the provisioning process of the build environment. Furthermore, the cloud instance that runs build environment in CodeBuild might already be started (used for the build task of other users) before CodePipeline sends our build job to CodeBuild. On the other hand, instances (Fargate) that run Jenkins agents is a cold start, which means it is not running before we send build job there. Nevertheless, we can also notice that the build time in the AWS integrated toolchain takes a longer time. It is hard for us to find the reason since the hardware configuration used by AWS is unknown except the size of RAM and the number of virtual CPU.

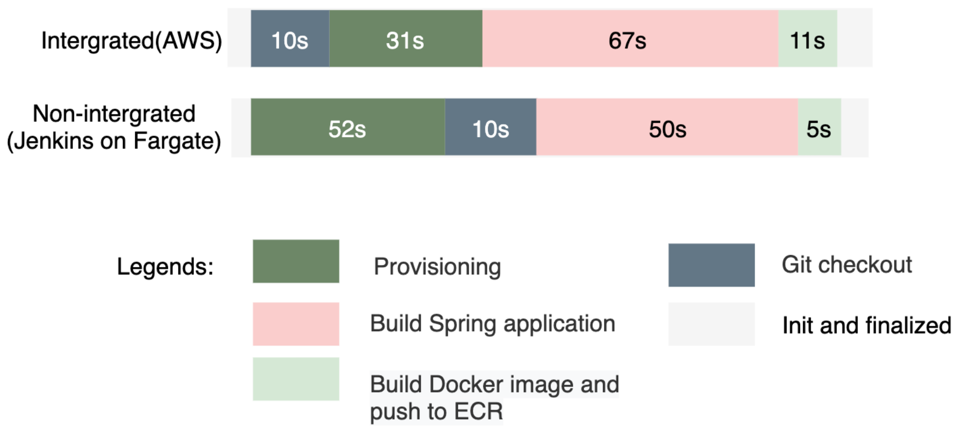


Figure 5.6: Observation of Runtime of Each Stage on Two Toolchains

Also, we observe that with the workload goes up, the runtime of AWS integrated toolchain increases with it. We already answered why the runtime of the pipeline in our non-integrated toolchain with agent runs in AWS Fargate does not change over time in 5.1.3. To explain this, we also check the runtime of each stage when runs several jobs in parallel. We notice that when it has multiple jobs runs in parallel, essential if the parallel jobs' number goes over 5. AWS will start limiting the resource allocation, which limits the number of jobs that runs at the same time. As a result, part of our running jobs has to enter the "Queued" status before entering the "Provisioning" stage. In 10 jobs we run in parallel for the experiment, the time spent for queuing for resource various from 1 second (get resource allocated directly) to 130 seconds. Among these ten jobs, four jobs were put into the queue before resource are available to them. Figure 5.7 shows the distribution of queued time among jobs. This validates the claim from AWS about parallel execution ability, which we mentioned in 5.2.2. However, the parallel is not fully in parallel but with some limitations on available resources.

The runtime of the non-integrated toolchain is also slightly going up because of the runtime of build and push container to ECR increases. The reason is obvious, we have these two stages runs on the master node, so the increased data transfer between agent and master and limited computing resource on the Jenkins master increase the runtime. However, as we observed in Experiment 1, the build stage is always fully paralleled, and the runtime of this stage remains unchanged despite the increasing workload. The queuing time for resource in each job is negligible.

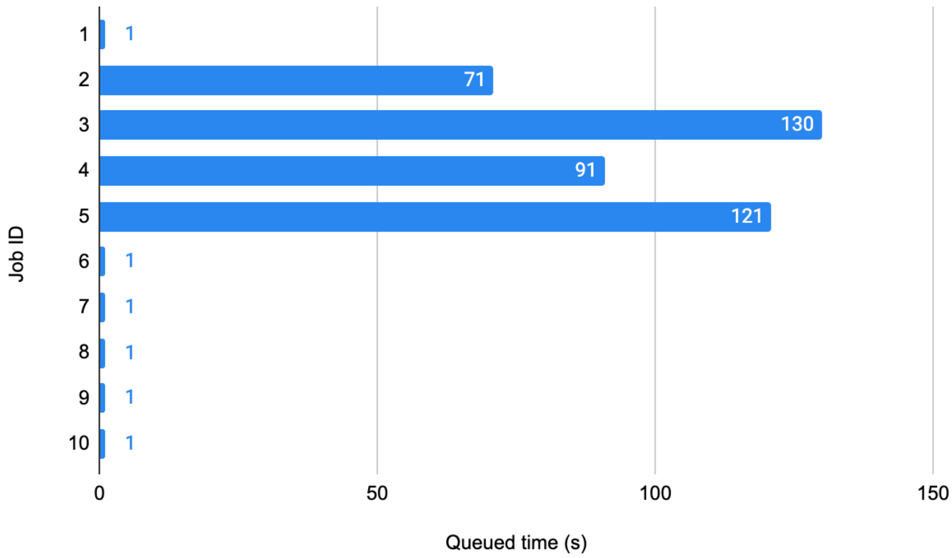


Figure 5.7: Observation of Queued Time in AWS Integrated Toolchain During The runtime Experiment, When 10 Jobs Run in Parallel

Cost Analysis

As a serverless tool, AWS DevOps tooling charges us according to the time and type of hardware configuration (in CodeBuild) we are using. Each pipeline in CodePipeline cost $1 per month, which is a negligible amount of $0.0014 per hour, price of with the build agent type general1.small (Linux instance with 3GB Memory and two vCPU) that we used in the experiment is $0.3 per hour. The CodeDeploy is free of charge under the condition that we are not deploying to on-premises servers. The cost of S3 which store artifacts (¡1 MB in our case) is negligible according to AWS[14]. In conclusion, the price should be $0.30014 per running job for one hour's runtime.

In our non-integrated toolchain, our experiment setup (2 vCPU, 4 GB RAM) is different from what we have in 5.1. The price should be $0.02646 per running agent for one hour's runtime. While the cost of EC2 instance that hosts the Jenkins master node costs $0.0432 per hour, this price will remain constant when multiple jobs run in parallel. In general, the cost of the non-integrated toolchain is $0.06966 per hour when only one agent is running. From the calculation, we can see that the AWS integrated toolchain is more expensive under similar performance.

5.2.4 Conclusion

By analysing and explain the results, we could see that two toolchains have similar performance when it runs our case project. However, by further observe the

[14]https://aws.amazon.com/getting-started/projects/set-up-ci-cd-pipeline/services-costs

runtime in each stage, we notice that the AWS integrated toolchain is faster in provisioning the build environment, but, slower in running the build and testing task itself. Such runtime distribution means it might be suitable for light but more frequent build task. From the parallel execution part, we conclude that with our AWS integrated toolchain, the AWS CodeBuild, which takes most of the runtime, do have a limitation on the resource that we can use at the same time. Thus, before the system provisioning any resource, some tasks have to wait for the resource allocation in the ”Queued” status,. The ”Queued” status largely prolongs the total runtime.

Chapter 6

Conclusions and Future Work

The efficiency and the agility of DevOps are driving more and more software development teams to perform the transition to DevOps. This master thesis project is done in Eficode Oy, a Finnish multinational company that one of its main tasks is to help customer with the DevOps transition. The transition to DevOps needs the team to precede the change in process, culture and tools. In this master thesis project, we focus on the tools aspect within the DevOps, which is the DevOps toolchain that helps the software team to implement DevOps practises.

The development of cloud computing technologies, especially serverless computing, has brought many changes in the development and deployment of DevOps toolchains. Serverless computing services in cloud could either act as the runner of automated build and automated testing, monitoring tools, or even replace the whole DevOps toolchain with a hosted integrated DevOps toolchain. This master thesis project explore the possibility that the serverless computing could improve the DevOps toolchain in term of performance and functionality. This is done by, first, implemented a traditional non-integrated toolchain and deploy to AWS, and find what improvement could AWS serverless computing services bring to this toolchain for answering RQ1. Second, implement a integrated toolchain with AWS serverless DevOps tools, and compare with the traditional toolchains, which answer our RQ2.

In this chapter, we first make conclusions by summarise our findings in chapter 3 and chapter 4. The conclusion is presented by give answers to our research questions in section 6.1. In section 6.2, we discuses the possible future work that could be done in the future.

6.1 Conclusions

In this section we summarise our main findings during the implementation and evaluation as the answers to two research questions.

RQ1: *How can serverless computing services in Amazon Web Services helps the DevOps toolchain?*

We identified there are following serverless computing services that could improving the DevOps toolchain within AWS.

The first improvement is done by the managed serverless container service (AWS ECS with Fargate) which could works as the build agent which host the build and testing process within the CD pipeline. According to our experiments and analysis, the feature of serverless computing, combined with the beneficial brings by Docker container, could improve the parallel execution performance and save cost. The high performance in parallel pipeline execution could be helpful in, first improving the performance of continuous delivery for microservices project, second, improving the performance of pipeline of a large origination with the frequent delivery. Furthermore, the serverless container service reduces the effort when set up the environment for running the agents. Docker build agent is widely used by different CI tools, and Elastic Container Services has extensive API which could be used for deploying the Docker build agent, therefore, this benefit is not only limited to the tools that we used in our implementation. Our experiments also shows that, Pay-per-use mode makes Fargate cheap to use for hosting the build agent.

Moreover, the serverless function (AWS Lambda) could be used within the monitoring part of the toolchain. The event-driving computing model of serverless functional is perfect for processing with the logs, alarms and realtime performance metrics within monitoring. In our toolchain we use AWS Lambda for customizing metrics and do auto-scaling upon alarms. In addition to these, AWS Lambda could be helpful to in build notification system that keeps the team aware the status within the pipeline.

The third improvement is from the serverless managed tools (AWS CloudWatch e.t.). Those tools could be used directly as tool in the toolchain. For example, we use CloudWatch as the monitoring solution in the toolchain. This type of tools free the development team from developing their own solution for certain part of the toolchain. If the tools is provided by the cloud provider where the DevOps toolchain is being deployed, the good integration between the tool and cloud infrastructure may allow a more detailed monitoring than any self-build or third party tools.

There are still limitation within serverless computing services. One most significant limitation is the Fargate does not support runs privileged container. This largely limiting what we operation can we do within the running Docker build agent, for example, we cannot build Docker image for the project that is being built in the build agent. Moreover, the performance of build agent runs in the serverless computing service is lower. This is due to the cold-start problem of the serverless computing. In addition, the average time of build stage of the same software project is longer in the serverless build agent (92s) than in EC2 based build agent(65s). This difference shows that serverless computing engine has lower performance when runs the automated build and automated testing within a continuous delivery pipeline.

RQ2: *How does the newly emerged integrated toolchain in Amazon Web Services compared with the traditional non-integrated toolchain?*

The integrated toolchain is a new type of DevOps toolchain that delivered as a single application. It is usually composed by serverless tools from a single vendor (in our case: AWS) but could also integrated with a limited range of third-party tools depends on the vendor. Compared with non-integrated toolchain, The first advantage for integrated toolchain is it is a out-of-box solution that needs much less time and effort to develop and setup, while a lot of works are needed when build the non-integrated toolchain, namely, tool selection, cloud infrastructure design and management, manual configuration and even software development. In addition, an integrated provides a better sight on the DevOps toolchain as a whole, this is easier to achieve because the integrated toolchain is delivered as a single application. In our experiment on performance, we find that the integrated toolchain is faster in provisioning the computing resource for continuous delivery. We believe due to the computing resources are also managed by AWS, it it easier for AWS to optimize the provisioning process.

We also find that, under the same hardware configuration, it takes the integrated toolchain longer time to run the automated build and automated testing. The gap of the runtime between integrated and non-integrated toolchain increased with the number of the parallel executing jobs. Our research shows this is because the AWS is imitating the maximum hardware resources that we could use. The restriction shows the first disadvantage of the integrated toolchain – the development team do not has fully control on the underlying hardware. A similar problem with integrated toolchain is that the team also has low visibility on the status of underlying hardware, which brought some challenges in our implementation. Moreover, our analysis shows the cost of integrated toolchain is higher in non+integrated toolchain. The high cost is because, first, not possible to manly use open-source solution, second, the AWS charges much more on computing resource when use it from it's DevOps toolchain. Furthermore, the integrated toolchains limits the tool selection, which on the one hand reduce the time needed for tool selection, on the other hand, it limiting the use of some special tools that is not supported by the toolchain.

6.2 Future Work

Based on our current implement and findings, we propose following further works could be done.

- Due to the tight schedule in finishing the thesis, our case project is rather simple, and the test cased are rather small. With the increasing popularity of microservices architecture, it could be interesting if the case project could be expended to a microservices which include different type of services. This could better simulate the software project in the real-life development.

- Current our non-integrated toolchain does not supporting run Docker within Jenkins build agent. As a result the Docker build task has to be executed on master node. The Docker build is luckily quite light in our case project, however if the Docker build is suppose takes longer time with a heavier case project, the non-parallel execution of this step could caused serious delay. A solution is needed for solving this problem.

- The monitoring solution in both solution could be extended, currently the monitoring is only based on the solution from AWS. The extension could be done by integrating CloudWatch with third-party tools, for example Grafana.

Bibliography

[1] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.

[2] Marco Miglierina. Application deployment and management in the cloud. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 422–428. IEEE, 2014.

[3] Ramtin Jabbari, Nauman bin Ali, Kai Petersen, and Binish Tanveer. What is devops? a systematic mapping study on definitions and practices. In *Proceedings of the Scientific Workshop Proceedings of XP2016*, pages 1–11, 2016.

[4] Gene Kim, Jez Humble, Patrick Debois, and John Willis. *The DevOps Handbook:: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution, 2016.

[5] What is a devops toolchain? – bmc blogs. `https://www.bmc.com/blogs/devops-toolchain/`. (Accessed on 03/13/2020).

[6] Devops - wikipedia. `https://en.wikipedia.org/wiki/DevOps`. (Accessed on 02/24/2020).

[7] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016.

[8] Devops toolchain - wikipedia. `https://en.wikipedia.org/wiki/DevOps_toolchain`. (Accessed on 03/11/2020).

[9] Liming Zhu, Len Bass, and George Champlin-Scharff. Devops and its practices. *IEEE Software*, 33(3):32–34, 2016.

[10] Serverless computing – amazon web services. `https://aws.amazon.com/serverless/`. (Accessed on 05/25/2020).

[11] Serverless computing vs. containers how to choose — cloudflare. `https://www.cloudflare.com/learning/serverless/serverless-vs-containers/`. (Accessed on 05/25/2020).

[12] Devops as a service: Automation in the cloud — sumo logic. `https://www.sumologic.com/insight/devops-as-a-service/`. (Accessed on 05/25/2020).

[13] Gartner says worldwide iaas public cloud services market grew 31.3% in 2018, 07 2019. (Accessed on 05/21/2020).

[14] Kim McMahon. The state of cloud native development. *KEY INSIGHTS FOR THE CLOUD NATIVE COMPUTING FOUNDATION STATE OF DEVELOPER NATION Q2 2019*, 05 2020.

[15] Y. Kim and J. Lin. Serverless data analytics with flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 451–455, 2018.

[16] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. Serverless computing for container-based architectures. *Future Generation Computer Systems*, 83:50–59, 2018.

[17] Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Sasko Ristov, and Radu Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21(4):64–71, 2017.

[18] Alex Glikson, Stefan Nastic, and Schahram Dustdar. Deviceless edge computing: extending serverless computing to the edge of the network. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–1, 2017.

[19] Vitalii Ivanov and Kari Smolander. Implementation of a devops pipeline for serverless applications. In *International Conference on Product-Focused Software Process Improvement*, pages 48–64. Springer, 2018.

[20] Shashikant Bangera. *DevOps for Serverless Applications: Design, deploy, and monitor your serverless applications using DevOps practices*. Packt Publishing Ltd, 2018.

[21] Jim Highsmith. What is agile software development? *crosstalk*, 15(10):4–10, 2002.

[22] Michael A Cusumano and Stanley A Smith. Beyond the waterfall: Software development at microsoft. 1995.

[23] Kent Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.

[24] Agile software development - wikipedia. `https://en.wikipedia.org/wiki/Agile_software_development#Iterative,_incremental_and_evolutionary`. (Accessed on 03/18/2020).

[25] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Principles behind the agile manifesto. *Agile Alliance*, pages 1–2, 2001.

[26] Sten Pittet. Continuous integration vs. continuous delivery vs. continuous deployment. *Web-article. Atlassian.¡ https://www.atlassian.com/continuous-delivery/ci-vs-ci-vs-cd¿. Fetched*, 24:2018, 2018.

[27] Martin Fowler and Matthew Foemmel. Continuous integration, 2006.

[28] Test automation in a ci/cd pipeline — spritecloud. `https://www.spritecloud.com/test-automation-with-ci-cd-pipeline/`. (Accessed on 03/19/2020).

[29] Build automation - wikipedia. `https://en.wikipedia.org/wiki/Build_automation`. (Accessed on 03/19/2020).

[30] Paul E Ceruzzi, E Paul, William Aspray, et al. *A history of modern computing*. MIT press, 2003.

[31] M Fowler. Continuous delivery. may 30, 2013, 2013.

[32] What is continuous delivery? - continuous delivery. `https://continuousdelivery.com/`. (Accessed on 03/23/2020).

[33] Why agile isn't agile without continuous delivery. `https://www.atlassian.com/continuous-delivery/principles/why-agile-development-needs-continuous-delivery`. (Accessed on 08/10/2020).

[34] Is continuous delivery replacing agile? `https://www.tempo.io/blog/2016/continuous-delivery-replacing-agile`. (Accessed on 08/10/2020).

[35] Marko Leppänen, Simo Mäkinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika V Mäntylä, and Tomi Männistö. The highways and country roads to continuous deployment. *Ieee software*, 32(2):64–72, 2015.

[36] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[37] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A survey of devops concepts and challenges. *ACM Computing Surveys (CSUR)*, 52(6):1–35, 2019.

[38] Devops in a scaling environment - tajawal - medium. https://medium.com/tech-tajawal/devops-in-a-scaling-environment-9d5416ecb928. (Accessed on 03/27/2020).

[39] Lucy Ellen Lwakatare, Pasi Kuvaja, and Markku Oivo. Relationship of devops to agile, lean and continuous deployment. In *International conference on product-focused software process improvement*, pages 399–415. Springer, 2016.

[40] Ian Buchanan. Agile and devops: Friends or foes. *Atlassian Agile Coach*, 2015.

[41] Mandi Walls. *Building a DevOps culture*. " O'Reilly Media, Inc.", 2013.

[42] Lucy Ellen Lwakatare, Pasi Kuvaja, and Markku Oivo. Dimensions of devops. In *International conference on agile software development*, pages 212–217. Springer, 2015.

[43] Dror G Feitelson, Eitan Frachtenberg, and Kent L Beck. Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.

[44] There's no such thing as a "devops team" - continuous delivery. https://continuousdelivery.com/2012/10/theres-no-such-thing-as-a-devops-team/. (Accessed on 03/26/2020).

[45] Jordan Shropshire, Philip Menard, and Bob Sweeney. Uncertainty, personality, and attitudes toward devops. 2017.

[46] FMA Erich, Chintan Amrit, and Maya Daneva. A qualitative study of devops usage in practice. *Journal of software: Evolution and Process*, 29(6):e1885, 2017.

[47] Devopsculture. https://martinfowler.com/bliki/DevOpsCulture.html. (Accessed on 03/27/2020).

[48] Matej Artac, Tadej Borovssak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. Devops: introducing infrastructure-as-code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 497–498. IEEE, 2017.

[49] M HERING, D DeGrandis, and N Forsgren. Measure efficiency effectiveness, and culture to optimize devops transformation. devops enterprise forum, 2015.

[50] Michael Hüttermann. *DevOps for developers*. Apress, 2012.

[51] N Forsgren, J Humble, and G Kim. Accelerate: state of devops report: Strategies for a new economy. dora (devops research and assessment) and google cloud, 2018.

[52] Toolchain - wikipedia. `https://en.wikipedia.org/wiki/Toolchain`. (Accessed on 03/11/2020).

[53] Nicole Forsgren Velasquez, Gene Kim, Nigel Kersten, and Jez Humble. State of devops report, 2014.

[54] Nicole Forsgren, Dustin Smith, Jez Humble, and Jessie Frazelle. 2019 accelerate state of devops report. 2019.

[55] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri. Devops: Introducing infrastructure-as-code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 497–498, 2017.

[56] Source and version control in devops – bmc blogs. `https://www.bmc.com/blogs/devops-source-version-control/`. (Accessed on 05/09/2020).

[57] Ben Chacon, Scott; Straub. *Getting Started – About Version Contro*. 2014. (Accessed on 08/11/2020).

[58] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[59] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[60] Alexander Zahariev. Google app engine. *Helsinki University of Technology*, pages 1–5, 2009.

[61] Serverless computing - wikipedia. `https://en.wikipedia.org/wiki/Serverless_computing`. (Accessed on 06/01/2020).

[62] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, 2019.

[63] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 442–450. IEEE, 2018.

[64] Thomson reuters case study. https://aws.amazon.com/solutions/case-studies/thomson-reuters/. (Accessed on 06/01/2020).

[65] Serverless computing – amazon web services. https://aws.amazon.com/serverless/#Serverless_application_use_cases. (Accessed on 06/02/2020).

[66] Tim Wagner. Serverlessconf 2018 keynote - debunking serverless myths. https://www.slideshare.net/TimWagner/serverlessconf-2018-keynote-debunking-serverless-myths, ServerlessConf 2018 2018. (Accessed on 08/11/2020).

[67] Serverless computing — google cloud. https://cloud.google.com/serverless. (Accessed on 06/02/2020).

[68] Azure serverless — microsoft azure. https://azure.microsoft.com/en-us/solutions/serverless/#solutions. (Accessed on 06/02/2020).

[69] Aws lambda limits - aws lambda. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html. (Accessed on 06/03/2020).

[70] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[71] Keeping functions warm - how to fix aws lambda cold start issues. https://www.serverless.com/blog/keep-your-lambdas-warm/. (Accessed on 06/03/2020).

[72] Tim Wagner. Serverless networking is the next step in the evolution of serverless — by tim wagner — a cloud guru. https://read.acloud.guru/https-medium-com-timawagner-serverless-networking-the-next-step-in-se 10 2019. (Accessed on 08/11/2020).

[73] What is amazon elastic container service? - amazon elastic container service. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html. (Accessed on 07/20/2020).

[74] Amazon Cloud Services. Amazon cloudwatch - application and infrastructure monitoring. https://aws.amazon.com/cloudwatch/. (Accessed on 08/12/2020).

[75] TIOBE. index — tiobe - the software quality company. https://www.tiobe.com/tiobe-index/. (Accessed on 06/11/2020).

[76] GitHub. The state of the octoverse. https://octoverse.github.com/, 2019. (Accessed on 06/11/2020).

[77] Programming languages used in most popular websites - wikipedia. https://en.wikipedia.org/wiki/Programming_languages_used_in_most_popular_websites. (Accessed on 06/11/2020).

[78] Spring — why spring? https://spring.io/why-spring. (Accessed on 06/11/2020).

[79] Spring boot. https://spring.io/projects/spring-boot. (Accessed on 06/11/2020).

[80] John Ferguson Smart. *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. " O'Reilly Media, Inc.", 2011.

[81] Jenkins. Pipeline-jenkins user documentation. https://www.jenkins.io/doc/book/pipeline/. (Accessed on 06/16/2020).

[82] What is a container? — app containerization — docker. https://www.docker.com/resources/what-container. (Accessed on 06/22/2020).

[83] Overview — drone. https://docs.drone.io/runner/docker/overview/. (Accessed on 06/10/2020).

[84] Overview — prometheus. https://prometheus.io/docs/introduction/overview/. (Accessed on 08/05/2020).

[85] Git - about version control. https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control. (Accessed on 06/12/2020).

[86] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. " O'Reilly Media, Inc.", 2012.

[87] Compare repositories - open hub. https://www.openhub.net/repositories/compare. (Accessed on 06/12/2020).

[88] GitHub Guides. Understanding the github flow, 2013.

[89] Vincent Driessen. A successful git branching model. *URL http://nvie.com/posts/a-successful-git-branching-model*, 2010.

[90] Scott Chacon. Github flow. 2011, 2011.

[91] Luca Tiozzo. Aws ecs host auto-scaling with custom cloudwatch metrics and aws lambda. https://medium.com/thron-tech/aws-ecs-host-auto-scaling-with-custom-cloudwatch-metrics-and-aws-lamb 04 2019. (Accessed on 07/05/2020).

[92] Cloud Foundry Documentation. Using blue-green deployment to reduce downtime and risk — cloud foundry docs. https://docs.cloudfoundry.org/devguide/deploy-apps/blue-green.html, 04 2019. (Accessed on 07/07/2020).