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

16th July 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

TODO

Preface

TODO MOTIVATION FOR RESEARCH TOPIC

TODO ACKNOWLEDGEMENTS

// collect all information about your project at the company in the Preface
Ruiyang Ding

Delft, The Netherlands
16th July 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 new software development method has drawn attention to the industry. The 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 [1] of Agile is to satisfies customer with early and continuous delivery of the software. The Agile, which aims at the improvement of the process within the software development team and the communication between the development team and costumers [2] do makes the software development faster. However, it doesn't emphasis 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 become the barrier for shortening the delivery time of the software project.

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 [3], 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 [4]. By eliminates the barrier we mentioned in the last paragraph, DevOps further fasten software delivery. In conclusion, DevOps means a combination of practices and culture which aims to combine separate departments(software development, quality assurance and the operation and others) in the same team, in order fasten the software delivery, maximizing delivered without risking high software quality [5][6].

In software engineering, the toolchain is a set of tools which combined for performing a specific objective. DevOps toolchain is the integration between tools that specialised in different aspect of the DevOps ecosystem, which support and coordinate the DevOps practices. The DevOps toolchain could assistant business in creating and maintain an efficient software delivery pipeline, simplify the task and further achieve DevOps [7]. On the other hand, DevOps strongly rely on tools. There are specialised tools exist for helping teams adopt different DevOps practices [8].

Traditionally, the DevOps toolchain is to have individual tools which are standalone and from different companies. The tools usually are on-premise. But can be also deployed on cloud virtual machines. We call it **non-integrated toolchain** in this report.

At the same period that the tools for DevOps emerged and developed, the cloud technologies also developed rapidly. This leads to the emigrations of Serverless Computing. The Serverless Computing is a new cloud computing model which all used to build and run the application on the cloud, without thinking about the servers [9]. It also allows developers to build the application with less overhead [9] and more flexibility by eliminates infrastructure management tasks [10]. With serverless computing technologies, many new cloud technologies emerged, which gives developers an alternative way than traditional cloud servers or cloud virtual machines. For examples: Functional computing allows the application to be divided by functions and designed under event-driving paradigm without managing the hardware infrastructures. The on-demand nature of the serverless computing cloud could be used to deploy certain component of a DevOps toolchain to ease the implementation difficulties and reduce the cost. Managed scalable container services in the cloud enable the user to run the container-based application directly on the cloud, which helps the toolchain become more scalable. DevOps tools as a service [11] allow the cloud provider to deliver a DevOps tool directly on its cloud platform.

Helping the customer do the DevOps transformation is one of the main business activities of Eficode, the company which I'm writing my thesis. This is done by the developing and deployment of DevOps toolchain for costumers. 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 in money and time. As part of thesis work at Eficode, We will investigate how could serverless computing help improve the DevOps toolchain.

1.1 Problem Statement

As per the last paragraph, serverless computing gives developers alternative ways to deploy the non-integrated DevOps beside cloud virtual machines. Several cloud providers utilise serverless computing. Among them, Amazon Web Services(AWS)[1] has the largest market share is the first cloud provider which provides the serverless computing services. According to the report from Gartner [12], the market share

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

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 includes for instance, AMS Lambda[3] for function computing, AWS Fargate[4] for managed container services. AWS also gains the most popular among the developers that using serverless technologies. The most recent survey report [13] 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. And the company keeps looking for ways to leverage serverless computing services in AWS to benefit the DevOps toolchains it builds for costumers.

However, despite the wide application of serverless technologies, and an enormous number of research papers about the use-case or benefit of serverless in data analysis [14], for container-based microservices [15], or for IoT applications [16] [17], the benefit of serverless to DevOps is not yet be discussed. There do have paper [18] or book [19] about DevOps toolchain for serverless applications. Nerveless, there still lack researches about how could serverless helps DevOps toolchain itself. Thus, our first research question is to fill the gap by answering this question.

The second area we'd like 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 pain of having to choose, integrate, learn, and maintain a multitude of tools. In other words, the cloud based-integrated DevOps toolchain is to offering 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 trying to build DevOps toolchain on 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, but 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:

1. **RQ1:** How can serverless computing services in Amazon Web Services help

[2]https://aws.amazon.com/serverless/
[3]https://aws.amazon.com/lambda/
[4]https://aws.amazon.com/fargate/

the DevOps toolchain?

2. **RQ2:** How does the newly emerged integrated toolchain in Amazon Web Services compared 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 the Eficode. We first analyze the functionality that each service has and how does these services could be used by our DevOps toolchain that deployed to AWS in Chapter 3.

For 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 developing and deploy the toolchain, we could already partly answer the RQ1 by answering how the Serverless computing services be used in our DevOps toolchain. Second, the implementation is 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. The metrics cover different perspectives includes cost, performance and development difficulties.

To answer RQ2, The non-integrated toolchain is used to compare with the DevOps toolchain build by the DevOps tools provided by AWS as a service. Besides, we conduct a study on a comparison between an AWS based traditional toolchain and the out-of-box integrated DevOps toolchains also provided by AWS as services. The reason that we keep the comparison scope within AWS is that both 2 toolchains are runs on the same hardware setup provided by AWS, this could eliminate the errors caused by the difference between vendors and focuses on the difference between toolchains.

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 2 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 include the concepts in cloud computing which is 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 designed and the implementation of our DevOps toolchains(both non-integrated and integrated). Chapter 5 focuses on the experiments and evaluations, which show how does the serverless computing services introduced in CH3 could benefit DevOps toolchain, and how these 2 kinds of toolchains we mentioned earlier compared with each other. 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, we introduce several main concepts related to our study. Section 2.1-2.3 shows the definition of DevOps and two important concepts that DevOps based on. Section introduced serverless computing.

2.1 Agile software development

The term ”Agile” represents the fast adaptation and response to the changes in the plan[20]. Agile software development is a new 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 costumers 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 shouldn't change the output (document and code) of this phase [21]. In contrast, the 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 the end an iteration, together with a demonstration to stakeholders. After delivery, the team continue to next iteration according to the feedback it gets from stakeholders. In each iteration, the team not aims to add major features to the software, rather their goal [22] is to have a working and deliverable release. In the ideology of agile, the best design the software product comes from the iterative development [1], rather than the tedious planning.

High Software Quality: The rapid development doesn't mean low software development quality. On the contrast, the quality of software design is highly appreciated in the agile software development. The automatic testing is widely used in Agile. The test cases will be defined and implements from the beginning of the development process. The testing goes through the whole development iteration ensure the software has a high enough quality to be released or demonstrate to costumers at any point of an iteration [23].

Collaboration: The agile software development processes include collaboration across different groups, ie. bushiness development team, software development team, test team, and costumers. It values more face to face communications [24] and feedbacks. The goal for these communications is, firstly to let everyone in the multifunctional agile team understand the whole project, secondly, to receive feedback that helps the software in the right development track that 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 [25]. As we 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 [25]. 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 to the main codebase [26]. The continuous integration rely on 2 practices: *Build Automation* and *Test Automation*. The definition of these 2 practices are:

- *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 [27].

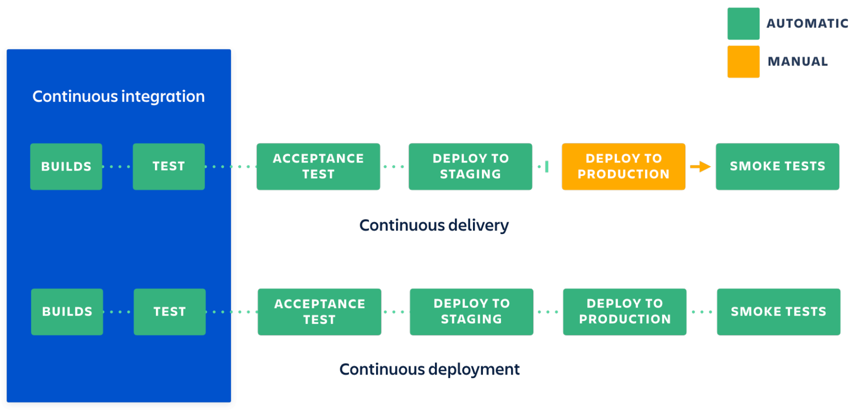


Figure 2.1: The relationship between continuous integration, continuous delivery and continuous deployment [25]

- *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 [28].

With the help of these 2 practices, for each developer in the team, the workflow [26] 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 could also be added to the automated test. After the development is done, automated testing also runs on the code to maintain the code quality and minimize the number of bugs from the beginning. The build automation 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 code base. 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 practices that software development team build a software that can be released at any time of the lifecycle [29].This means the software always maintains a high quality and in a deployable state[30]. It is a subset of agile, which focuses on the software delivery[31]. From the last section, we introduce the concept of continuous integration. The continuous delivery is based on continuous integration but further automate the software deployment pipeline. In the software deployment pipeline, the team divide build into several stages, first build the product and then push the product into the production-like environment for further testing. This ensures that the software could be pushed to production at any time. However, in continuous delivery, the deployment of software into production is done manually. The benefit [30][29] of continuous delivery includes:

- High code quality: The automate and continuous testing ensure the quality of the software.
- Low risk: The software could be related at any time, and it's easier to release and harder to make the mistake
- Short time before going to the market: The iteration of software development is much shorter. The automation in testing, deployment, environment confirmation included in the process, and the always read-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 deploy without manual approval. In the continuous deployment, each change that passed automated build and testing will be deployed directly. The continuous deployment is a relatively new concept that most company not yet put the practice into production [32]. While continuous delivery is the required practice for the company to be DevOps and it is already being widely used.

2.3 DevOps

> The fundamental goal of DevOps is to minimize 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 (eg. 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 [33].

DevOps is the extension and evolution [34][35] of Agile. DevOps and Agile both driven by the collaboration ideology and the adoption of DevOps needs Agile as the key factor [34]. DevOps has a different focus on agile. DevOps focus on the delivery while agile is focused on the development with the requirement and customer [36]. Figure 2.2 shows the workflow and practices of a team working under DevOps.

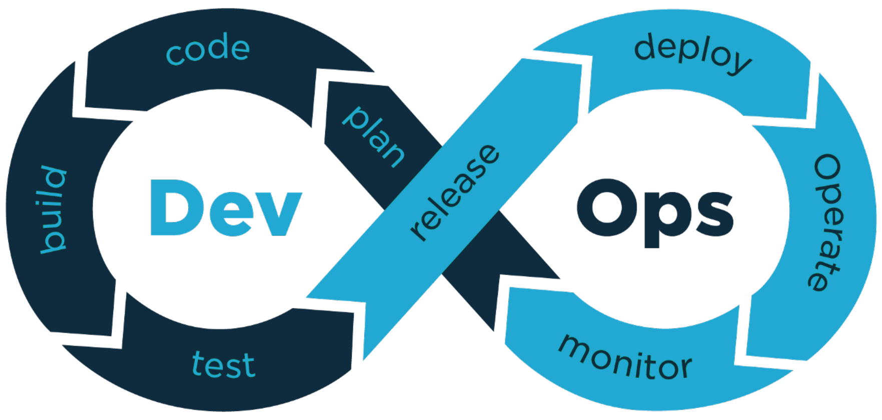


Figure 2.2: DevOps Practices and Workflow [37]

2.3.1 Elements

In this section, we will introduce the necessary elements that an organization need to includes when employing DevOps. 4 necessary elements need to be considered.

Culture

In the pre-DevOps era, the Development and Operation are two different teams with a different goal. The interface between them is based on the ticket system which the operation team do the ticket management. As we mentioned at 2.1, the goal of Agile is to shorten the deliver life cycle and delivery software quickly to the costumers. So when practice agile development method under this scenario, the development try to deliver the code they develop earlier but 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 costumers [35]. 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). According to Walls (2013), this is being done by promoting the culture with 4 characterises: open communication, incentive and responsibility alignment, respect and trust [38].

The open communication means openly discussion and debate. As mentioned above, the traditional communication method is through a very formal and regularized ticket system. In the DevOps, the communication is not limited within the formal ticket system. Instead, the team will keep in the whole lifecycle of a product, from the requirement, schedule, and anything else [38]. The information sharing is also important [39]. The metrics and the project status is available for everyone in the team , so each member could have a clear scope about what the team is doing.

The incentive and responsibility alignment mean the whole teams (combines Dev and Ops) shares the same goals and also takes the same responsibility. The shift from ”Dev” and ”Ops” to DevOps requires people who used charges in only development and operation starting sharing the responsibility from both side [39]. This means individuals or a certain part of the team will be not solely blamed if the product is failed. This ”no blame” culture could help each engineer be willing to take the development responsibility for the whole system [40].

Respect means all employees should respect and recognize 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 [41]. However the people operation team will take development responsibility, and the developers will also put their hands-on operation and management[42]. To make people with different roles works in a team, trust and respect each other is critically important.

Organisation

In the organizational level, the DevOps emphasizes the collaboration between different part of an organization. This is strongly correlated with the ”culture” part of this section. Inside a team, each member should be a generalist who could understand all aspect of a project. There will not be a dedicate QA, operation or security team within a team. Instead, these are the job that belongs to everyone [40][3]. The organization should provide the team member with opportunities to learn all skill needed for building the whole system.

The team size should be small. A small team could help to reduce the inter-team communication. The small team means the scope of the project is small. And it also means less bureaucracy in team management. There are four benefits [3] to have a small team:

- The smaller team allows each team member to easily understand the whole project.
- 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 decentralize 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 doesn’t mean a disaster for the company. This allows the team to fail. Thus each employee could train their headship skill in the team without too much pressure.

Furthermore, another important organizational aspect for DevOps is to have a loosely-coupled architecture. The first benefit of this is the better safety. In the organisation with a tightly-coupled architecture, small changes could result in large failure [3]. The second benefit is productive. In a traditional organisation, the result of each team will be merged, tested together and deploy together. This means it is time-costly to configure and manage the test environment requires dependencies. A loose organisation enable each team to finish the development of lifecycle (from planing to deployment) independently. Each team could update their products independently, which gives the team more flexibly to align the product with the change in the customer requirement. This means the update of each team's product won't affect other teams as well.

Automation

In the DevOps, automation means automation within the whole development and operation process. The organisations which employing DevOps aims for a high degree of automation[43]. 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 [44]. The main practices regarding Automation are the automated testing, continuous delivery and automated operation. Automated testing could be achieved by test automation. We already mentioned the benefit of this at 2.2.1.

The continuous delivery pipeline is the core of the DevOps [45]. As we discussed at 2.3.2. The continuous delivery will ultimately automate all steps between the developer to commit the code to the product in the production.

The automation of the operation part is usually done by using the concept of "Infrastructure as Code" [39]. The Infrastructure as Code (IasC) means to define everything in the software infrastructure level as code [46]. Because it is code, we could use the automation methodology used in the software development to manages and deploy these codes. According to Christof et. (2016), under IasC, infrastructure can be shared, tested, and version-controlled [6]. This could help emphasizes the automation within the operation scope. With the automation in operation, the team could be free from the tedious environment configuration and shorten the product development lifecycle. Automating server configuration means the developers and operation staff can equally know the server configuration [44] which help build the culture of shared responsibility and trust.

Monitoring and Measurement

Monitoring is to continuously collect the matrices from the running system for helping the team find the problems in the system. To do 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 [47]. The result (metrics) should be properly used by the organisation.

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, we cannot test the software any more. So, we need monitoring to keep track of the status of the product [48]. According to State of DevOps report from Google, the good monitoring structure and the wisely usage of the data from monitoring for making bushiness decision could improve the software delivery performance [49]. 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. The monitoring has also enabled the team to collect the data from costumers' usage behaviour. This helps the agile development team to improve in the next iteration of the product [39].

For develop a high-quality monitoring system, the development of monitoring could 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 this, the development team can improve the monitoring system continuously together with the main software system. The parallel development of the monitoring system and the main system helps the team to find the gap in the monitoring earlier [48].

As we 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 we mentioned at 2.3.1 (Culture) that the project status and matrices should available to every team members.

2.3.2 Toolchain

A DevOps toolchain is a set of tools that integrated to aid the software development, deployment and management through the whole software development lifecycle, which helps the software development to fit the DevOps principles [7][50][4]. Each tool in the toolchain supports specific activities in DevOps, for example, version control, build, testing.

According to [4], Google Cloud state of DevOps reports [49][51][52] and our previous definition of the DevOps, we summarize 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

Provided a central platform to manage the configuration across the assets. This is usually done by defining the desired state of the assets in a configure file and automate the configuration process which reaching the assets to the defined status. In 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[53].

Continuous Integration

Continuous integration (in short: CI) is the top practice for improving the Deployment Frequency [51]. It is one of the most important parts of DevOps toolchain. As we introduced at 2.3.2, CI allows the developers to integrate their work more frequently to the production products, 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. So CI is the core part of the whole DevOps toolchain.

Version Control

Version control is the key component of DevOps toolchain. It is a system that could record and track the changes in a set of file overtime. Version control simplifies the collaboration between team members. and allow the simultaneous development on the different part of a software system According to [54] and [51], version control is the top practices when comes to improve the multiple metrics in DevOps. Version control becomes the indicator of the software system performance [54] Infrastructure as code, an important DevOps practise we mentioned at 2.3.1 also relies on the version control.

Monitoring

The monitoring system is one of the basic practices in a DevOps toolchain[52]. It is also one of 4 basic elements of DevOps as we mentioned at 2.3.1. In the DevOps toolchain, the monitoring system detects the failure in the whole system and helps the software team find the problems earlier. The log taking by the monitoring system can also record the system activity history which allows the further analysis.

Automated Testing

The automated testing tool could verify the code before it being built. Due to the common practise of continuous integration which we mentioned at , the automated testing usually integrated into the continuous integration pipeline. The integration of testing in the CI pipeline makes it easy for the organisation to implement the quality gate in the software development [48].

2.4 Serverless Computing

In this section, we focus on the concepts of Serverless Computing. We 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 which the sever and resources allocation is managed by the cloud provider. The popularity of serverless is precipitated by the development of microservices and container technologies [55]. A survey from Cloud Native Computing Foundation (CNCF) shows that, in 2019, 41% of respondents are using serverless technologies in the production, the number was 32% in 2018[13]. The report of this survey also shows that serverless architectures and cloud functions are being used by 3.3 million developers [13] in 2019.

In the traditional cloud computing service, the user rents the fixed number of cloud servers from the cloud providers, and the cloud providers charge user according to the renting length and the server type (pay-as-you-go model). While in the serverless computing services, the developer only pays according to the execution time of the program. Another difference between serverless computing and traditional computing method is that, in serverless computing, although the task is still running on the physical cloud servers, the cloud servers are fully managed by the cloud providers. The means the user leave all server provisioning and administration tasks to the cloud providers [56] when using serverless.

2.4.1 History

In the early days of cloud computing, the consideration behind the design of cloud computing is that the developer simply moves their deployment environment from the local server to the server on the cloud. Therefore, the cloud virtual machine, for example, Amazon Web Service EC2 is the main form of the cloud service providing. 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 year 2008, Google released Google App Engine (GAE) [1][57]. 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 runs 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

[1]https://cloud.google.com/appengine

first public cloud provider that provides serverless computing platform[58]. Since then the serverless computing starts its rapid commercial development. Following AWS, other providers also introduced their serverless computing platforms. Only in year 2016, Google [2], Microsoft [3], and IBM [4] released their serverless computing platform respectively.

2.4.2 Characterises

We summarise 4 main characterises of serverless computing.

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 task such as data processing. A good example is the serverless computing use case of Thomson Reuters in their social media data analysis project[59]. Thomson Reuters uses AWS Lambda to hosting a serverless application that triggered when new data is stored. The application processes the data real-time, extract the hashtag trend data and store it in Amazon DynamoDB, a database solution by AWS, which is also serverless.

Managed Resources Allocation

The managed resources allocation means the developer only need to deploy the code but leaving the operation task to the cloud. As we mentioned before The developer doesn't need provisioning or managing any server besides, the developer is not required to install any software or runtime [60] when deploying his/her application.

The managed resources allocation also means the cloud provider will manage the scaling of the infrastructure which the developers are running code on. In the traditional virtual machine, although some cloud providers, for example, AWS and Azure support auto-scaling, however, the scaling policy has to be defined by the user. And the user needs to set up the cloud infrastructure for using autoscaling. On the contrast, in Serverless computing, the cloud provider will handle everything

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

related to auto-scaling. Furthermore, the availability and security issues 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. This is not economy flexibly for the user since they have to pay even nothing is running on the VM they are using, they still pay as the same as when their VM is fully loaded. On the contrast, in serverless computing, the users don't need to pay the idle time, they only pay for the time that the application is running. In any scenario, such payment mode could lower the cost [56].

Extensive Application Scenarios

Serverless computing has extensive application. The serverless runtime that what we discussed the most above. But beside deploy runtime on the cloud, serverless computing also gives you more possibilities which cover all backend services that we could be possibly used when building a modern application. According to the definition of Amazon Web Service, it's serverless offering not only include serverless functions (AWS Lambda) but also include serverless database, container runtime services, data analysis and Kubernetes cluster, which we already mentioned in Chapter 1. Google cloud also advocates ”full-stack serverless” [61]. Same with AWS, Google Cloud also provides, all kinds of serverless solutions, from compute, DevOps storage, to AI and data analysis. Furthermore, Azure's serverless offering also covers a wide range of the backend component, including computing, storage, ai, monitoring and analysis [62].

2.4.3 Limitations

Serverless computing is not the perfect solution. In some aspects, it still has it's limitations compared with traditional VMs.

Hardware Performance

This is mainly the problem within the compute task that runs in serverless. In the current serverless offering from cloud providers, the computation power of serverless computing is limited. For example, in the virtual machine service (AWS EC2) provided by AWS, the user could select the virtual machine which up to virtual 96 CPUs and 192 GB RAM. While in the AWS serverless computing engine, the maximum allocated RAM size is only 3008MB [63] and no maximum vCpu number is specified in the documentation. This has limited the application scenario of serverless computing by makes it unsuitable for the heavy task. The limitations to the hardware selection also imitating the performance in some cases like model training. The experiment in research by UC Berkley shows that, because AWS Lambda doesn't support GPU computation, when training the deep learning model, it is 21X slower than EC2 instance with GPU [64]. The longer execution time also makes serverless more expensive in such a case.

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 [65]. 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 that the container is already been deactivated by the cloud provider. In such a situation, the cloud has to provision a new container and this will significantly increase the total execution time.

Communication

In current serverless computing offering from cloud providers, there is a lack of network communication between different running serverless instances [64]. For example, in AWS Lambda, the communication between executing cloud functions can only be done through slow cloud storage. 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.

Chapter 3

Analysis of Current Serverless Cloud Services

In this Chapter, we will do an literature review on the new cloud technologies which emerged in recent years.

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 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(server-based and serverless) 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.

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. This means we will 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 is necessary for us to 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 [66], 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 [67] from GitHub shows that Java ranks third most popular programming language in 2019, and it ranks second before 2018. Furthermore, Java has good versatility, which means it can be used in the development of almost every kind of applications. For instance, Java could be used for developing 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 very complete ecosystem. This 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 the part of the DevOps toolchain.

Therefore, 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 [68] 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's being used in many major internet companies including Google, Microsoft and Amazon [69].

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 [70], 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 DevOps toolchain which is non-integrated. Part of the components is still based on the virtual machine. 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.

```
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

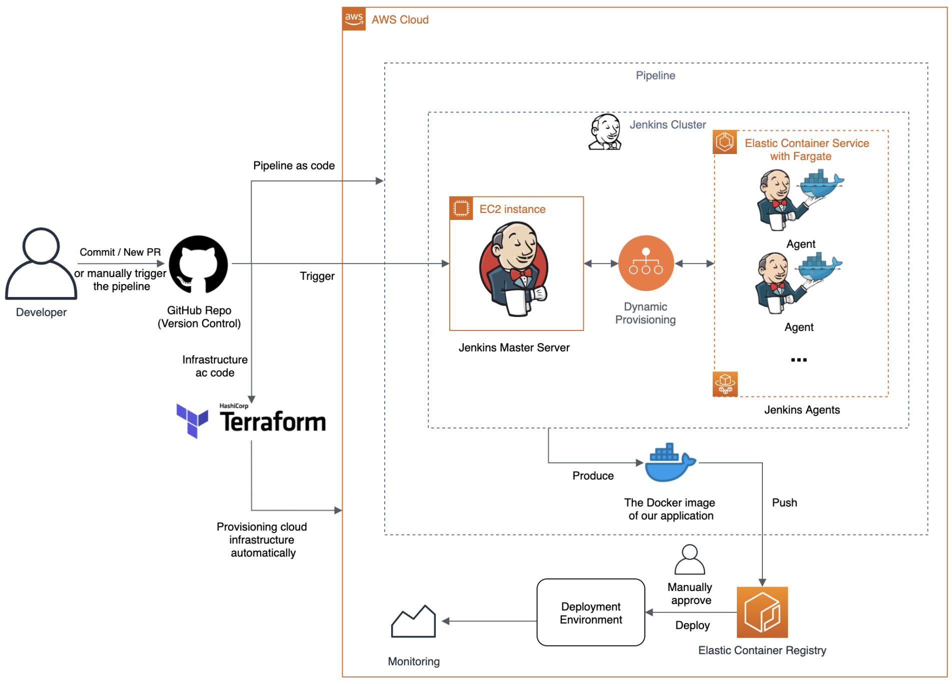


Figure 4.2: Architecture diagram of our DevOps toolchain

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 architec-

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

ture of our DevOps toolchain. In here we only presenting architecture on a more general level. The detailed architecture of each component will be introduced in the following sections, in both text and graph.

When the developer pushes a new commit to the repository in GitHub [2], Github will send an 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, 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.

4.2.2 Tool Selection Considerations

One of the most important 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]. A comparison between these tools is shown in Table 4.1. 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 3 newer tools, Jenkins is more focuses on the ”Build” step within the continuous delivery pipeline. Yet, 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.

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 [71]. Furthermore, Jenkins also attracts software

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

	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

teams with its easy-to-use and high extendibility [71] 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's an aged software created when the term "DevOps" just appeared.

Our continuous delivery pipeline is built 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 [72] 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, etc.) 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, dependencies could be easily 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're using JUnit [15] as the tool for testing. For code analysis, we use SonarQube[16]. Both are one of the most common

[12]https://www.jenkins.io/doc/book/pipeline/

[13]For example, "Build", Test", "Deploy" step in a continuous delivery pipeline.

[14]https://gradle.org/

[15]https://junit.org/junit5/

[16]https://www.sonarqube.org/

used tools in their specialized field in the Java ecosystem. And 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 an isolated unit that includes the application and all its dependencies which allow application runs in the same way regardless of the host environment [73]. A container is the running instance of a Docker image that defined by Dockerfile.

There will be 2 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 certain steps inside ephemeral Docker containers [74]. 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 good 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 simply 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're serverless computing services. This means our Dockerized application could easily be cloud-native and be deployed across a multi-cloud environment.

4.2.3 Infrastructure as Code (IasC)

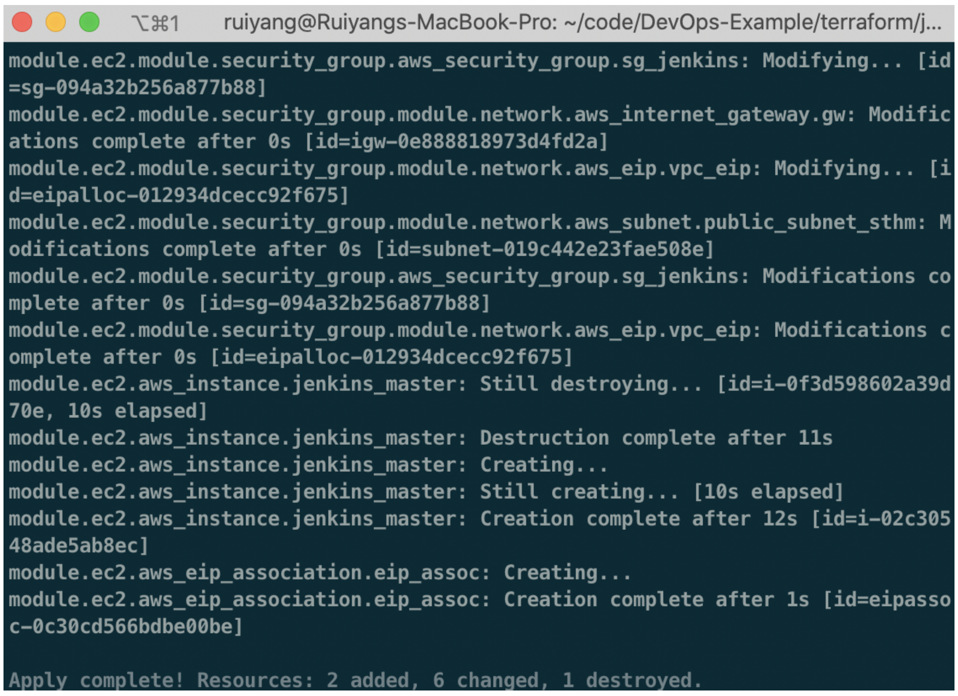
Configuration Management is one of the component of DevOps toolchain that we mentioned in Chapter 2. Infrastructure as code the common practices to implement configuration Management in the cloud-based environment.

Terraform [19] is one of the most popular tools to manage cloud Infrastructure with IasC practice. It has the thorough support of AWS. In our implementation,

[17] https://www.docker.com/

[18] https://www.docker.com/

[19] https://www.terraform.io/



```
ruiyang@Ruiyangs-MacBook-Pro: ~/code/DevOps-Example/terraform/j...
module.ec2.module.security_group.aws_security_group.sg_jenkins: Modifying... [id
=sg-094a32b256a877b88]
module.ec2.module.security_group.module.network.aws_internet_gateway.gw: Modific
ations complete after 0s [id=igw-0e888818973d4fd2a]
module.ec2.module.security_group.module.network.aws_eip.vpc_eip: Modifying... [i
d=eipalloc-012934dcecc92f675]
module.ec2.module.security_group.module.network.aws_subnet.public_subnet_sthm: M
odifications complete after 0s [id=subnet-019c442e23fae508e]
module.ec2.module.security_group.aws_security_group.sg_jenkins: Modifications co
mplete after 0s [id=sg-094a32b256a877b88]
module.ec2.module.security_group.module.network.aws_eip.vpc_eip: Modifications c
omplete after 0s [id=eipalloc-012934dcecc92f675]
module.ec2.aws_instance.jenkins_master: Still destroying... [id=i-0f3d598602a39d
70e, 10s elapsed]
module.ec2.aws_instance.jenkins_master: Destruction complete after 11s
module.ec2.aws_instance.jenkins_master: Creating...
module.ec2.aws_instance.jenkins_master: Still creating... [10s elapsed]
module.ec2.aws_instance.jenkins_master: Creation complete after 12s [id=i-02c305
48ade5ab8ec]
module.ec2.aws_eip_association.eip_assoc: Creating...
module.ec2.aws_eip_association.eip_assoc: Creation complete after 1s [id=eipasso
c-0c30cd566bdbe00be]

Apply complete! Resources: 2 added, 6 changed, 1 destroyed.
```

Figure 4.3: Creating a cloud environment with Terraform CLI

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 [75], and versioning the history of these files. VSC is suitable for track the development progress and manages the goal within a software development team [76]. Among all software for version control, Git is the most popular one nowadays. The survey [77] from Synopsys shows that in 2019, 71% of the project today is using Git as it's 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.

The Git flow [79] proposed in 2010 is a successful workflow for working with Git. Git flow has already widely used and has been approved by the software

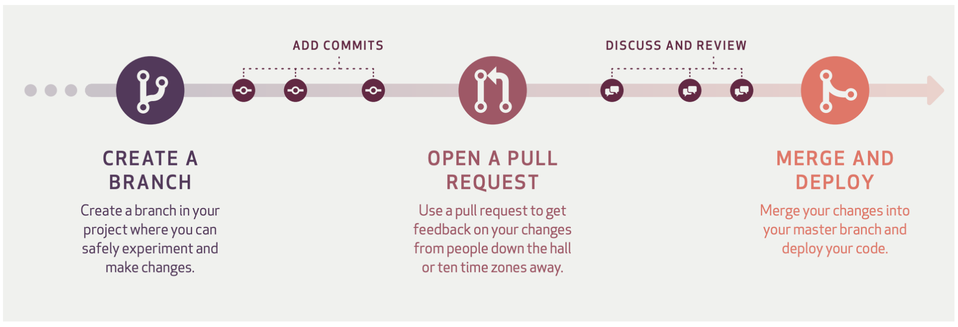

ADD COMMITS
DISCUSS AND REVIEW
CREATE A BRANCH
Create a branch in your project where you can safely experiment and make changes.
OPEN A PULL REQUEST
Use a pull request to get feedback on your changes from people down the hall or ten time zones away.
MERGE AND DEPLOY
Merge your changes into your master branch and deploy your code.

Figure 4.4: GitHub Workflow [78]

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 [80] 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 [80] 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. And there shouldn't 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 code related to this feature this branch and push from this branch to the branch with the same name on the remote server (github.com).

- Open a pull request[20] when the feature is ready to merge, or when you feel that you need help or comments from other team means on this feature. The code review is also done by others in the pull request.

- When the code is already be 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

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

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 with Jenkins

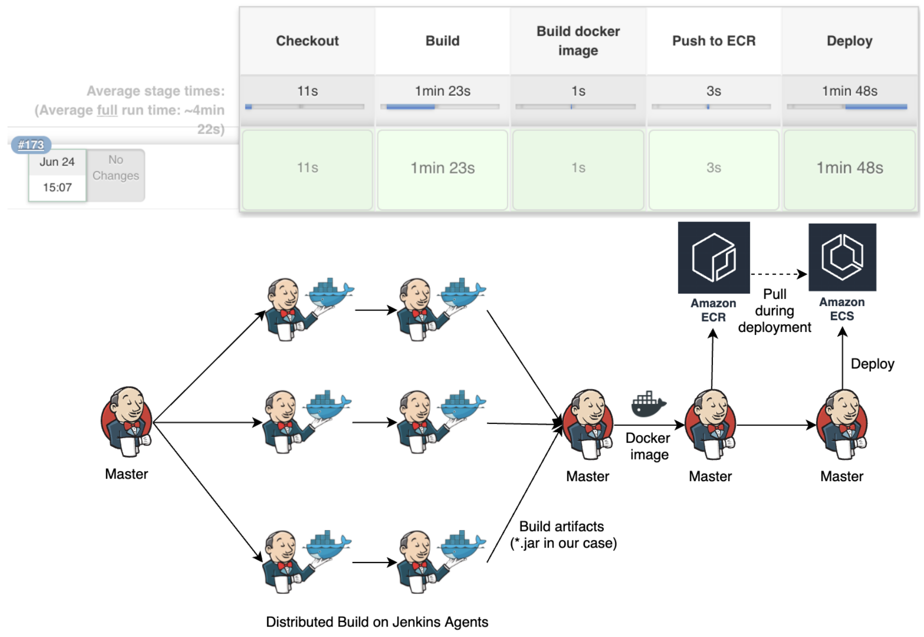


Figure 4.5: The Stages and Distributed Build in Our Pipeline

Figure 4.5 shows the 5 stages in our pipeline that shown in the Jenkins dashboard. 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.

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 executes 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 to first 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 [21] of the Jenkins agent.

[21]You could find the Docker image we developed at https://hub.docker.com/r/dry1995/jnlp

We use the "jenkins/jnlp-slave"[22] 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 build environment within the agent. We add shell script for auto-install all build dependencies to build 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 serverless. 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) / 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 2 steps take most of the time in our pipeline and according to Figure 4.6 runs more frequently than other steps [23]. The running time will be further extended when building a larger project. These 2 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 Jenkins agents 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 can't put the stage that builds a Docker image in Jenkins agents. This is because AWS Fargate does not allow building agent runs in a privileged 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 case project the Docker build only takes a very short time (¡1s on average). Therefore this won't slow down the whole pipeline.

We also notice that the Deploy stage also takes a long time. Still, we don't 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 2 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 the workflow of a project that goes through our continuous delivery pipeline. We can see when the pipeline is triggered by the event on the feature branch, it only runs through the first 2 stages. This is because according to the practices of continuous integration mentioned by

[22]https://hub.docker.com/r/jenkins/jnlp-slave/

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

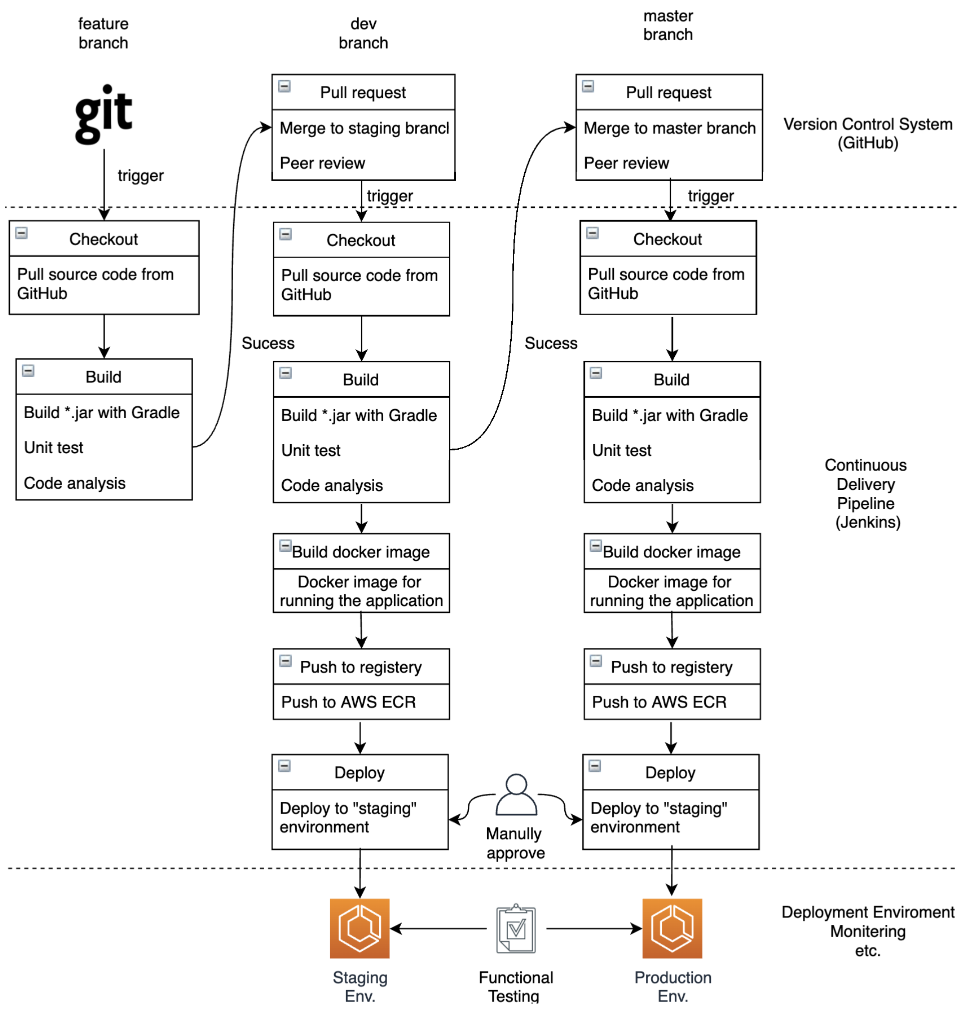


Figure 4.6: The Workflow of Continuous Delivery Pipeline in Our DevOps Toolchain

us in 2.3.2 and by Martin Fowler in [26], 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 2 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 analysed, 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). 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 reducing pipeline running time if there is error exists in the code.

If all the above steps are done and no error returns, 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. This is to make sure that the automated tests don't 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, and then we push the image to the Amazon Elastic Container Registry AWS (AWS ECR) for further use.

The last 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. This means we 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 functional testing. This is for test if our deployed API works and if it works as expected. If the deployed function passes the smoke 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 costumers.

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

Figure 4.7 shows our deployment architecture. The deployment region in Stockholm(EU-north-1). The Fargate instances in ECS cluster are automated scaled

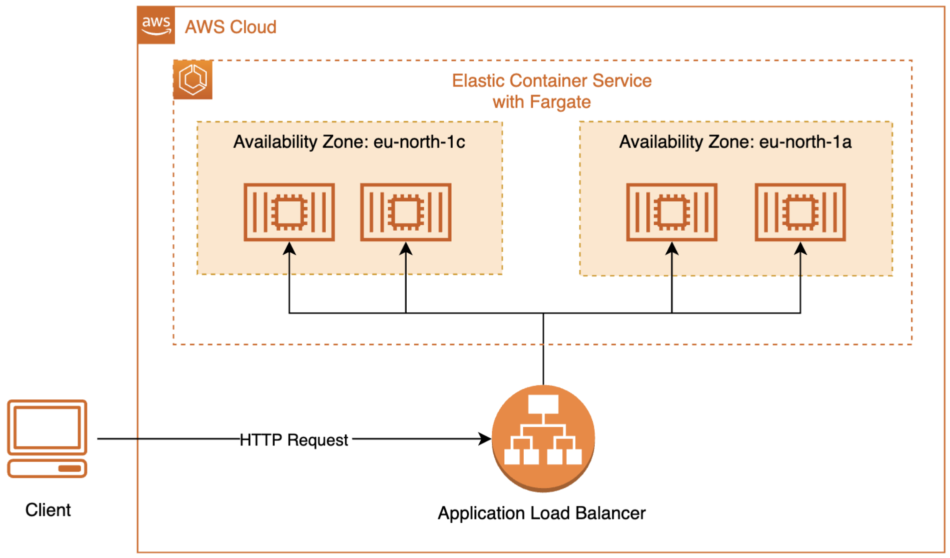

AWS Cloud
Elastic Container Service with Fargate
Availability Zone: eu-north-1c
Availability Zone: eu-north-1a
HTTP Request
Client
Application Load Balancer

Figure 4.7: Deployment Environment

according to the number of incoming requests. To improve the availably of the product, we deploy the case project into 2 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 the Deployment

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 primary tool for monitoring. 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 4.8 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.

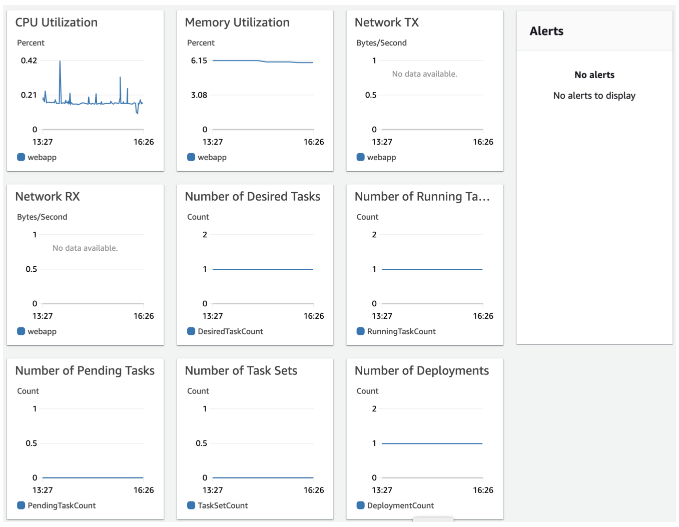


Figure 4.8: Cloudwatch Monitoring Dashboard

Auto-Scaling the ECS Cluster with Custom Alarm in Cloudwatch As we mentioned in 4.7 Deployment Environment, The deployment could be auto-scaled. This is done 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. The scaling process is usually done by this: 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 [81], 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 2 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 threshold and function that aggregating the metrics need to be determined by the DevOps team 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. But, 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 visualisation platform, for example, Grafana [24] to give the management team an overview of the KPIs.

4.3 Design of Serverless DevOps Toolchain

AWS provides a set of serverless DevOps tooling 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 tooling 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 4.9 shows the general workflow of a project delivered by our serverless 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 2.3.2. Instead of Jenkins which is server-based, we build the pipeline with AWS CodePipeline. Figure 4.9 shows the activity within the CodePipeline in a single graph. Different from Jenkins who can do solely the whole continuous delivery process with help of plugins, the CodePipeline just provides a platform which you can configure a workflow with AWS DevOps tooling or other third-party tools.

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 tooling, so it is not necessary for us to change our version control system away from GitHub.

In the next step, we 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 very simple compared with Jenkins. In Jenkins, without the help of the plugin, the pipeline

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

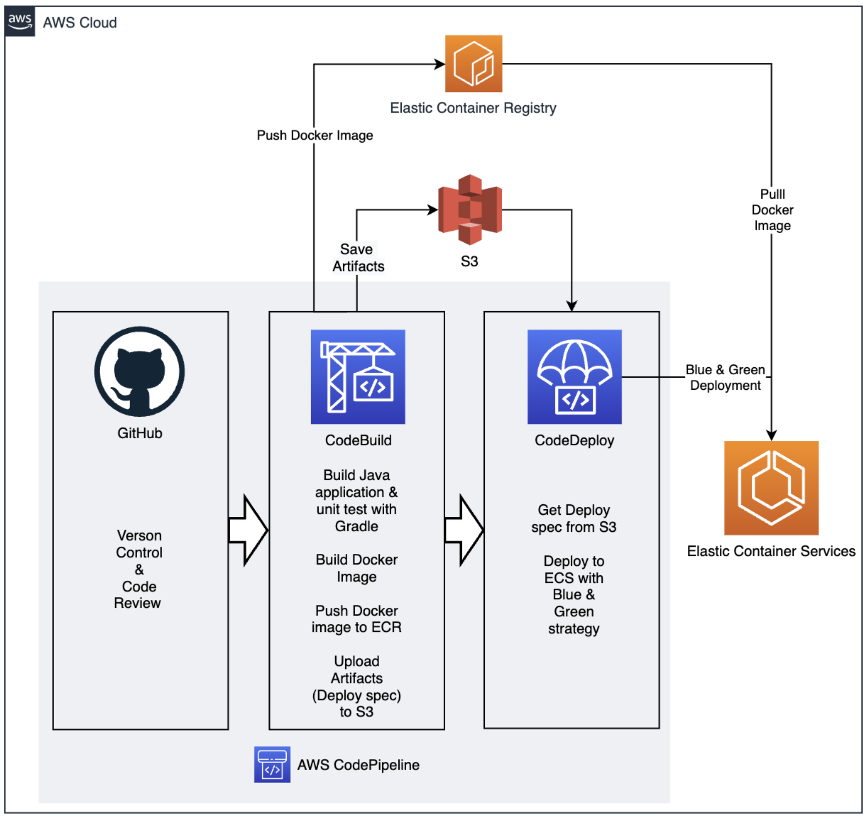

AWS Cloud
Elastic Container Registry
Push Docker Image
Pulll Docker Image
Save Artifacts
S3
GitHub
Verson Control & Code Review
CodeBuild
Build Java application & unit test with Gradle
Build Docker Image
Push Docker image to ECR
Upload Artifacts (Deploy spec) to S3
CodeDeploy
Get Deploy spec from S3
Deploy to ECS with Blue & Green strategy
Blue & Green Deployment
Elastic Container Services
AWS CodePipeline

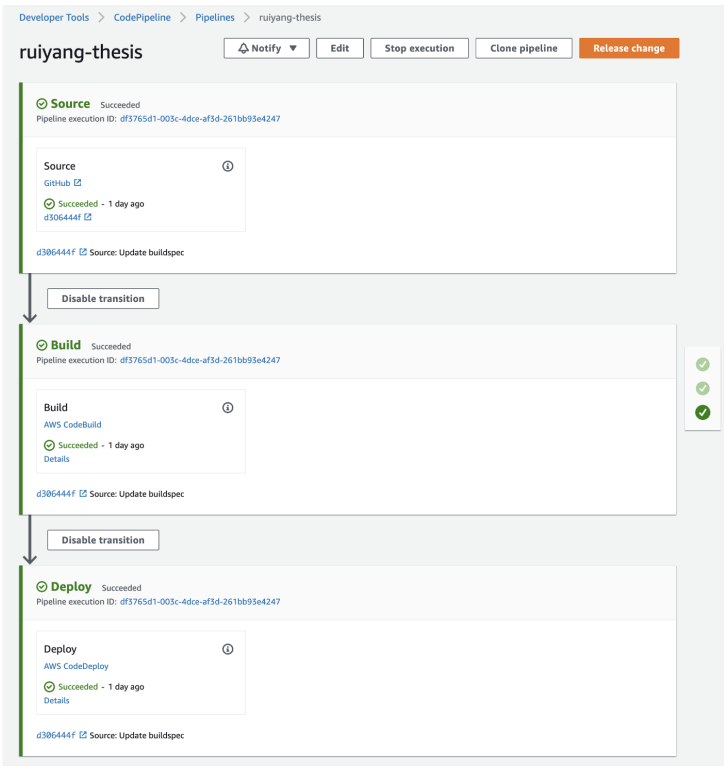
Figure 4.9: Serverless DevOps Toolchain

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 Docker container that provided by AWS already contains environment for build of different programming languages. It also includes Java environment and Gradle which needed by our case project. Therefore we could save time in setting up the pipeline since we don't 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



ruiyang-thesis

Source Succeeded
Pipeline execution ID: df3765d1-003c-4dce-af3d-261bb93e4247

Source
GitHub
Succeeded - 1 day ago
d306444f

d306444f Source: Update buildspec

Build Succeeded
Pipeline execution ID: df3765d1-003c-4dce-af3d-261bb93e4247

Build
AWS CodeBuild
Succeeded - 1 day ago
Details

d306444f Source: Update buildspec

Deploy Succeeded
Pipeline execution ID: df3765d1-003c-4dce-af3d-261bb93e4247

Deploy
AWS CodeDeploy
Succeeded - 1 day ago
Details

d306444f Source: Update buildspec

Figure 4.10: Our Workflow in CodePipeline

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 tooling 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 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.

In real-life production, the team would like to make sure the deployment is reliable with minimized downtime. Thus the safety in 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

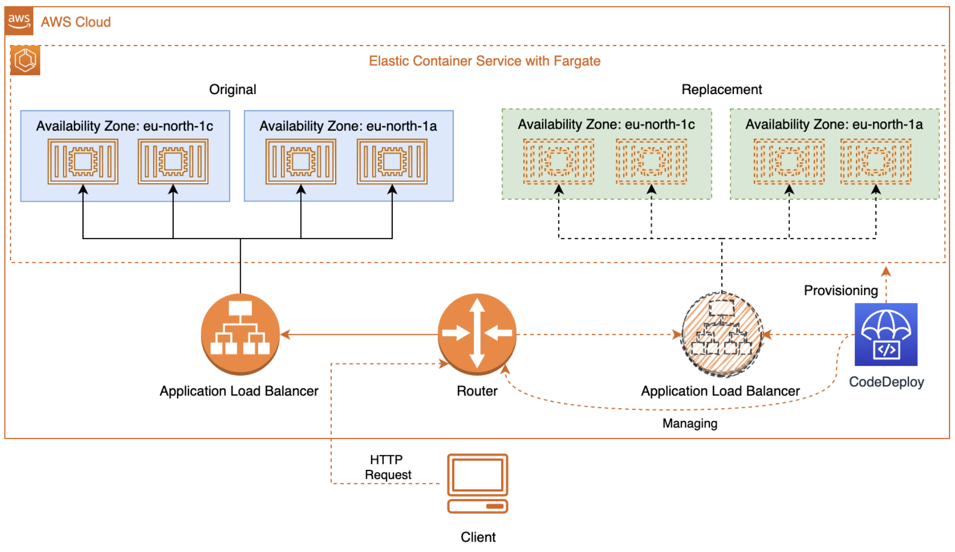

AWS Cloud
Elastic Container Service with Fargate
Original
Replacement
Availability Zone: eu-north-1c
Availability Zone: eu-north-1a
Availability Zone: eu-north-1c
Availability Zone: eu-north-1a
Provisioning
Application Load Balancer
Router
Application Load Balancer
CodeDeploy
Managing
HTTP Request
Client

Figure 4.11: Blue and Green Deployment for Our Deployment Solution

and original version of code respectively, while the router gradually routing more incoming requests to the environment that runs the newer version of code.

Figure 4.11 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 4.7). CodeDeploy controls a router before the load balancer. When new deployment comes, CodeDeploy does the following steps:

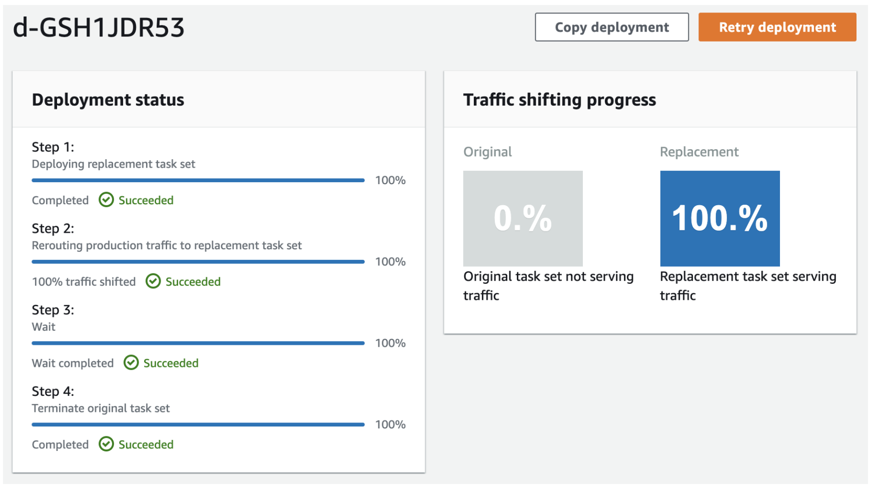
- Provisioning new identical deployment environment (replacement environment) and deploy a newer version of code on it. In ECS, the deployment is called ”task set”.

- Control the router, rerouting incoming traffic gradually to replacement ECS task set. We set the rerouteing rule as 10% per minuets. We don’t rerouting 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 rerouting is done. During the rerouteing and this 5 minutes, the load balancer in replacement deployment environment keeps doing the health check by sending a request to health check API endpoint of our case project. CodeDeploy read the health status from the load balancer. If the replacement tasks set is un-healthy, CodeDeploy does rollback by rerouting 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 error happens with a newer version of code, we can immediately roll back to the last version by switching the rerouteing to Blue [82] 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 back to the previous version. This could cause longer downtime of the server.


d-GSH1JDR53

Deployment status

Step 1:
Deploying replacement task set
100%
Completed ✓ Succeeded

Step 2:
Rerouting production traffic to replacement task set
100%
100% traffic shifted ✓ Succeeded

Step 3:
Wait
100%
Wait completed ✓ Succeeded

Step 4:
Terminate original task set
100%
Completed ✓ Succeeded

Traffic shifting progress

Original: 0.% — Original task set not serving traffic

Replacement: 100.% — Replacement task set serving traffic

Figure 4.12: CodeDeploy Dashboard

The better integration of CodeDeploy with the rest of AWS also be evidenced by 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 rerouting process. Figure 4.12 shows the dashboard of CodeDeploy that shows the status of our case project during the deployment.

4.4 Comparison between Integrated/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. And 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.

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

	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

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[25] 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 tool-

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

chain 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.
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 basically 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's "DevOps Partner Solutions" [26]. If user wants to use anything except these tools, it is not possible.

But, 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.

In a non-integrated toolchain, the software team is allowed to pick any tools for each component as they want, as long as those tools can be integrated with each other. 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 company internal 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 integrated with each other, 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 doesn't react to the build triggering signal from GitHub. This means the further maintaining of the toolchain is needed by the software teams. In integrated toolchain, the toolchains are delivered as a single cloud-based software, each part of naturally coped with each other. This makes integration much easier. The better integrating between each component also makes it easier to monitor the toolchain as a whole.

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 basically 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 prob-

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

lem with all hosted serverless services since the users don't 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 5 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 our there are two plugins that support run Jenkins agent in ECS cluster. However, we find our only one actually works after tried both plugins. Besides, the documentation of Jenkins plugins sometimes is very poor. For example, the documentation of the plugin that we use for Jenkins agent is too brief to tell us how to use the plugin, 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 3 factors, we spent a very long time in selecting and configuring tools. And also 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 2 stages take a very short time (¡5s in total), so this limitation won't slow down the pipeline too much when multiple builds run in parallel.

To solve this problem, a possible solution is to runs these 2 stages in AWS CodeBuild, AWS CodeBuild has support to Jenkins, which allow us to run certain Jenkins stages in CodeBuild. And 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 job start running 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 [27] 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 economic way to solve the challenge with the current setup.

4.5.4 Challenge IV: No Enough Visibility in AWS DevOps tooling

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. Yet, 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.

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

Chapter 5

Performance Comparison and Evaluation

In this chapter, we describe our experiments regarding 2 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 this experiment, we also 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 possibility to further ease the infrastructure management task for the Docker build agents. This experiment is a controlled experiment which examines whether serverless

[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/

container service could improve the continuous delivery pipeline from various perspectives.

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 CH3, and better-answering research question 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 4 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 parallelization 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 Utilization 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

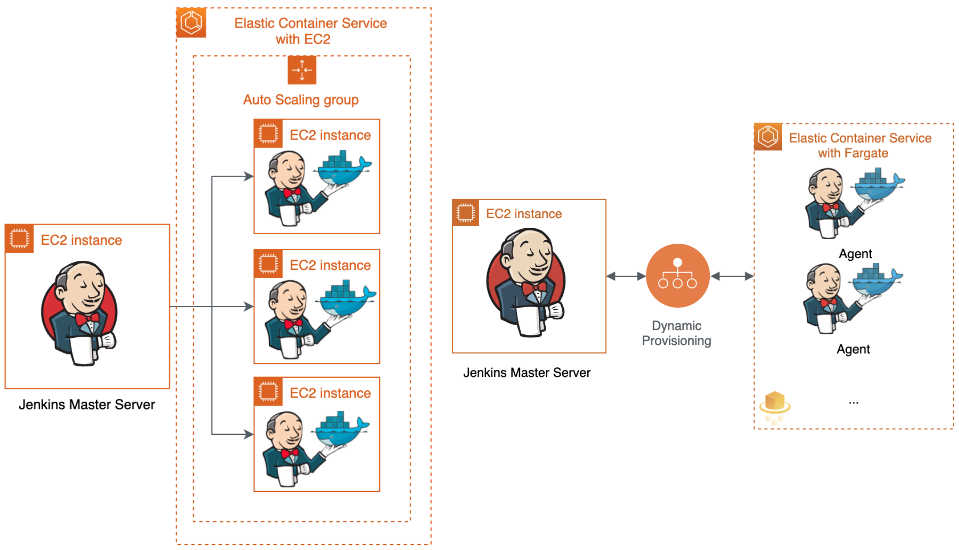


Figure 5.1: Architecture diagram of the test Jenkins cluster with agents running in traditional virtual machines (left) and on ECS with AWS Fargate (right)

instance of type t3.medium with 2 virtual CPU, 4 GB RAM and 30 GB disk. The EC2 instances as worker node are type t3.small, with 2 virtual CPU and 2GB RAM. Each EC2 instance can run 1 container at the same time.

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 2 virtual CPU, and 2 GB of RAM. This 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 you can find at [7]. The docker image includes essential dependencies that will be used to build the Spring Boot application and the base image which allow 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). The automated testing and code analysis is being integrated with this step, and being conducted by the plugin of Gradle.

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

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. This simulates the different team size that and show the scalability when comes to the need for task parallelization in bigger organizations.

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 Utilization* 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. The test result is depicted in Figure 5.2.

The test result shows that when 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, 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 initialization of underlay infrastructure that runs the serverless container. This takes a significantly longer time. This shows one of the limitations of serverless computing (cold-start) that we mentioned in 2.4.3.

When comes to parallel job execution, the performance of Fargate is significantly better. This is because, in Fargate, each container runs on an independent

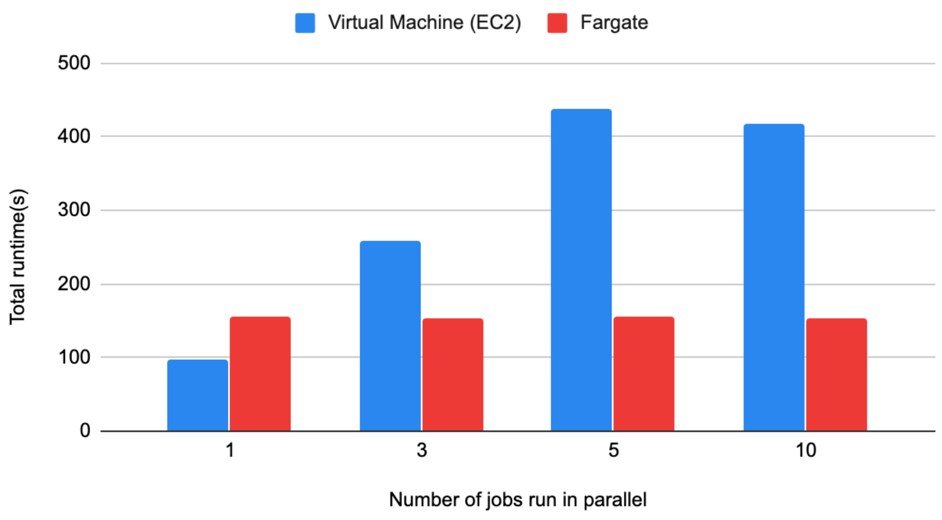


Figure 5.2: Runtime of Pipeline with Different Jenkins Agents

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 runs 3 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 parallel task reaches 10, the EC2 runtime becomes shorter. This 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. But even with 3 EC2 instances, only 3 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 is heavier to create a VM than create a new Fargate instance. The other reason is the auto-scaling of EC2 VM is based on reaching certain resource utilization 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 utilization within containers that runs Jenkins agent in two setups. The data from AWS cloud watch shows the resource utilization rate is similar in these 2 setups. This is because the "run in same way regardless of

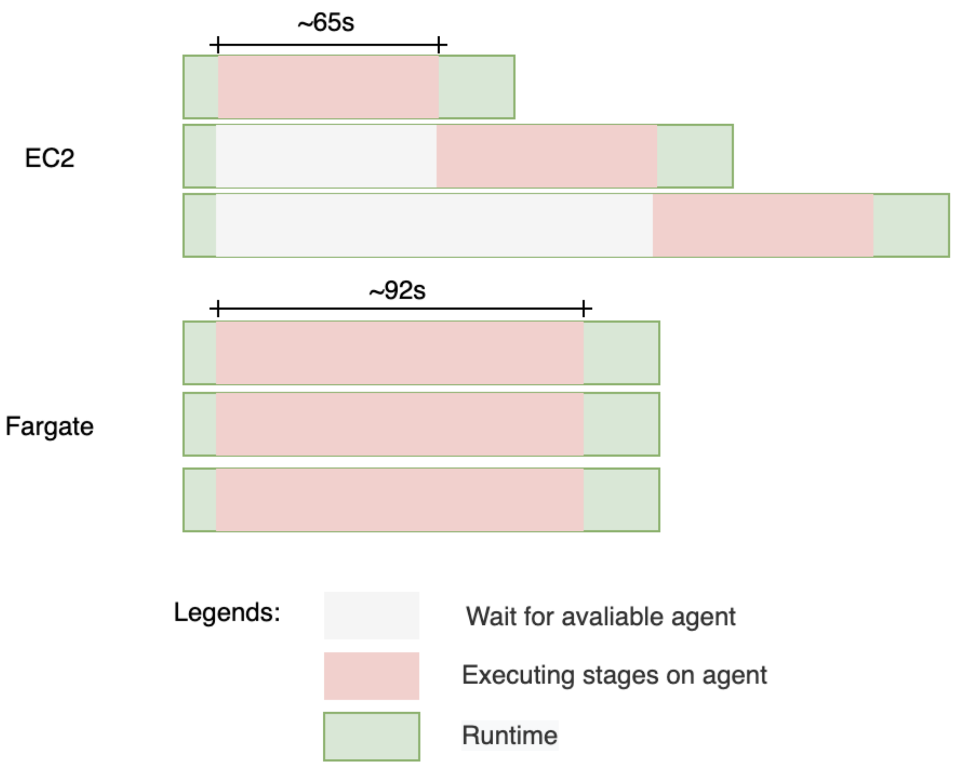
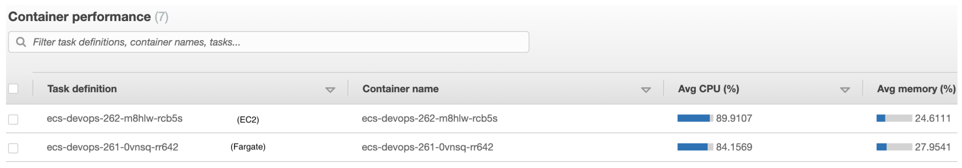


Figure 5.3: Execution Mode of Pipelines with Different Jenkins Agents

host environment" [73] 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 set-up (2vCPU, 2 GB RAM), the price should be \$0.01813 per running agent for one hour's runtime. In

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

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 the job is distributed by Jenkins master. On the job finished, the Fargate instance will be terminated immediately. However, EC2 not so flexible due to the resource-utilization-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 comes to single job execution, this 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 utilization 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, we deliver our case project through two DevOps toolchains. And we compare the quantitative result from the experiment. 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 2 virtual CPU and 3 GB RAM. The Hardware configuration for Jenkins build agent is 2 virtual CPU and 4 GB RAM. We cannot set the RAM to 3 GB since AWS Fargate is

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

not allowing [10] RAM to below 4GM when using 2 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 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.

The resource utilization comparison is not available in the experiment since we cannot get the resources utilization 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 the 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 [11]. 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.

The Figure 5.5 shows the result of this experiment. From the graph,, we can see that during execution of single job, tow toolchains has similar runtime. This is because the build stage, which takes most of runtime, is runs in the similar environment in both toolchains. Both execution environment is within a Docker container

[10]https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-cpu-memory-error.html

[11]Workload means different number of jobs runs in parallel in both pipelines

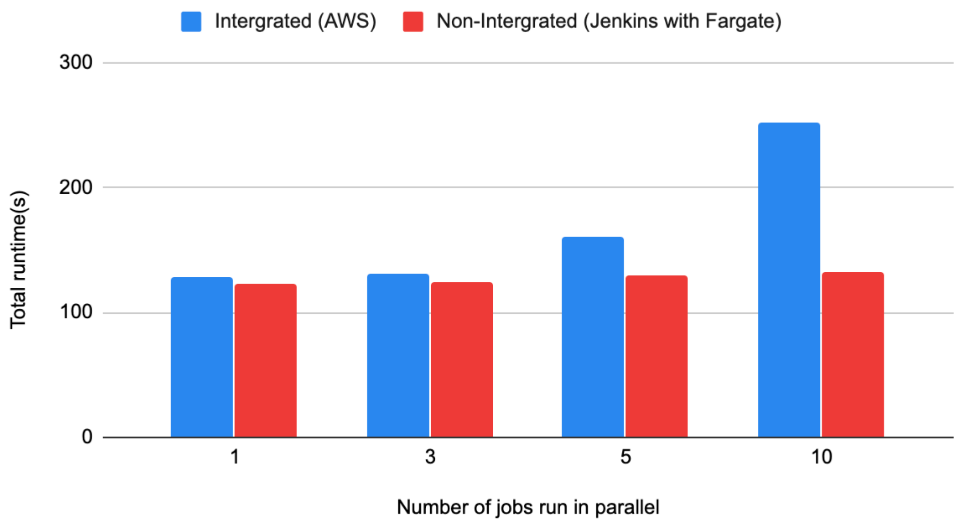


Figure 5.5: Runtime of Pipeline with on Different Toolchain

runs with same hardware in AWS. Although, the Docker image for 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. But still, the performance is not so different in both toolchain.

We further observe the time allocation between stages within the runtime in both toolchains. Figure shows our observation. To get the runtime in Figure 5.6, we run pipeline in each toolchains for 5 times and get the average runtime of each stage. The first difference shows in the Figure 5.6 is that the Git Checkout stage in AWS integrated toolchain does not runs on the container build agent, instead, it is runs 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 second, while the AWS toolchain does not do anything in the first 5 second. We presume that AWS is provisioning environment for runs the Git checkout stage in this 5 second, so this stage is also runs in a serverless environment within AWS, although we are not allowed configure anything in this environment. The second difference is that in AWS integrated toolchain, the provisioning of build environment is much faster. We presume this is due to both build agent and this toolchain are managed by AWS, the provision process of the build environment is being optimized by AWS. Furthermore, the cloud instance that runs build environment in CodeBuild might already started (used for the build task of other users) before our build job is sent to CodeBuild, on the other hand, instances (Fargate) that runs Jenkins agents is cold start, which means it is not running before we send build job there. But, 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.

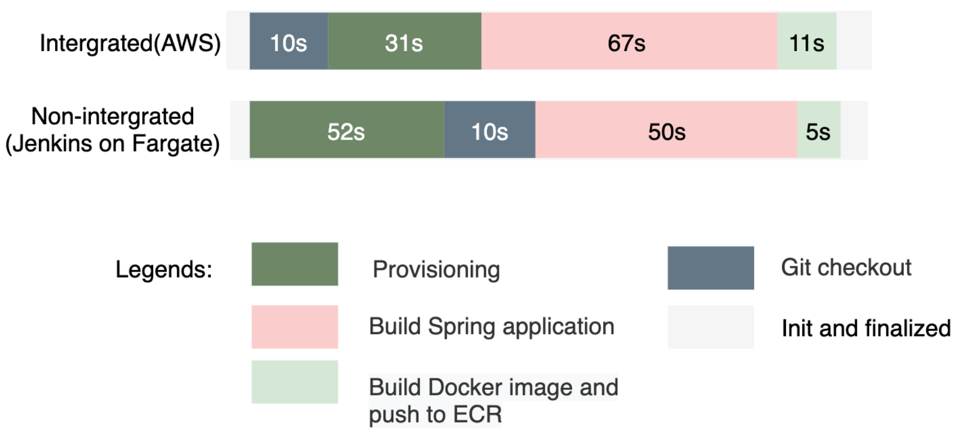


Figure 5.6: Observation of Runtime of Each Stage on Two Toolchains

In addition, 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 changes overtime 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 job runs in parallel, essential if the parallel jobs' number goes over 5. AWS will starts limiting the resource allocation to limits the number of job that can be runs at the same time. As the result, part of our running jobs have to be putted in "Queued" status before entering the "Provisioning" stage. In 10 jobs we runs in parallel for experiment, the time spent for queuing for resource various from 1 seconds (get resource allocated directly) to 130 seconds. Among these 10 jobs, 4 jobs was putted into queue before resource are available to them. Figure 5.7 shows the distribution of queued time among jobs.

Cost Analysis

As a serverless tools, AWS DevOps tooling charge 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 2 vCPU) that we used in experiment is \$0.3 per hour. The CodeDeploy is free of charge under condition that we are not deploy to on-premises servers. The cost of S3 which store artifacts (¡1 MB in our case) is negligible according to AWS[12]. In conclusion, the price should be \$0.30014 per running job for one hour's runtime.

In our non-integrated toolchain, our experiment set-up (2 vCPU, 4 GB RAM) is different with what we have in 5.1, the price should be \$0.02646 per running

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

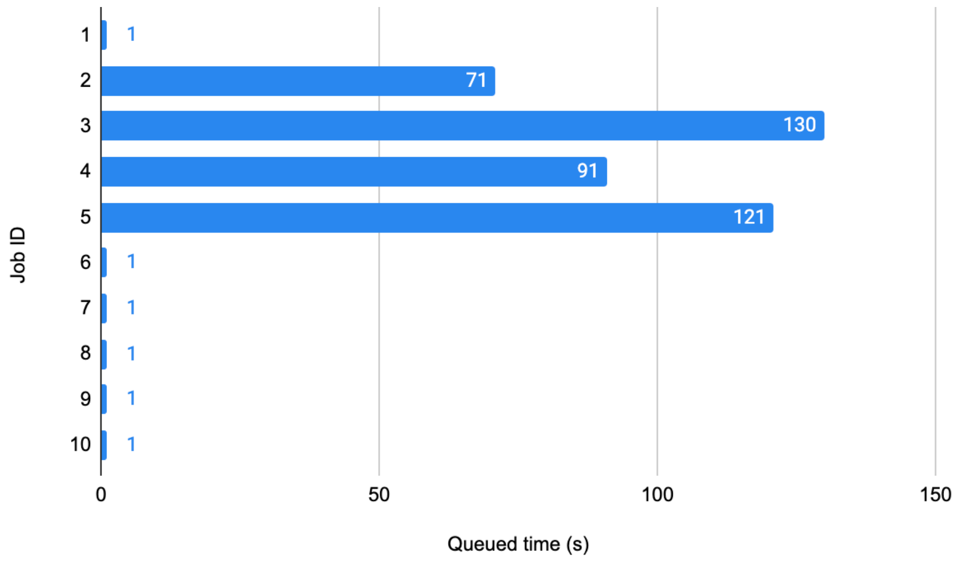


Figure 5.7: Observation of Queued Time in AWS Integrated Toolchain During The runtime Experiment, When 10 Jobs Run in Parallel

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 remains constant when multiple jobs runs in parallel. In general, the cost of non-integrated toolchain is 0.06966 when only one agent is running. From the calculation we can see that the AWS integrated toolchain is much more expensive under similar performance.

5.2.4 Conclusion

Chapter 6

Conclusions and Future Work

6.1 Conclusions

TODO CONCLUSIONS

6.2 Future Work

TODO FUTURE WORK

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] 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.

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

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

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

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

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

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

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

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

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

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

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

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

[16] 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.

[17] 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.

[18] 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.

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

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

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

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

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

[24] 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.

[25] 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.

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

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

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

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

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

[31] Continuous delivery vs. traditional agile - dzone devops. `https://dzone.com/articles/continuous-delivery-vs`. (Accessed on 03/24/2020).

[32] 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.

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

[34] 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.

[35] 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.

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

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

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

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

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

[41] 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).

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

[43] 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.

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

[45] Asif Qumer Gill, Abhishek Loumish, Isha Riyat, and Sungyoup Han. Devops for information management systems. *VINE Journal of Information and Knowledge Management Systems*, 2018.

[46] 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.

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

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

[49] 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.

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

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

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

[53] 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.

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

[55] 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.

[56] 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.

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

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

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

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

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

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

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

[64] 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.

[65] 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).

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

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

[68] 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).

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

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

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

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

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

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

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

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

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

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

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

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

[81] 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).

[82] 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).