A DevOps Toolchain for Efficient Software Development

Ruiyang Ding

Delft University of Technology

A DevOps Toolchain for Efficient Software Development

Master’s Thesis in Cloud Computing and Services

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

Ruiyang Ding

28th June 2020

Author
Ruiyang Ding

Title
A DevOps Toolchain for Efficient Software Development

MSc presentation
TODO GRADUATION DATE

Graduation Committee
TODO GRADUATION COMMITTEE   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
28th June 2020

Contents

Preface **v**

1 Introduction **1**
1.1 Problem Statement 2
1.2 Research Approach 4
1.3 Thesis Structure and Main Contributions 4

2 Background and Concepts **7**
2.1 Agile software development 7
2.2 Continuous Integration & Continuous Delivery 8
2.2.1 Continuous Integration 8
2.2.2 Continuous Delivery and Continuous Deployment 9
2.3 DevOps 10
2.3.1 Elements 11
2.3.2 Toolchain 14
2.4 Serverless Computing 16
2.4.1 History 16
2.4.2 Characterises 17
2.4.3 Limitations 18

3 Analysis of Current Serverless Cloud Services **21**

4 Design of DevOps Toolchain **23**
4.1 Case Project 23
4.1.1 Programming Language and Framework Considerations . 23
4.1.2 Project Description 24
4.2 Design of Non-integrated DevOps Toolchain 25
4.2.1 Architecture 25
4.2.2 Version Control 26
4.2.3 Continuous Delivery Pipeline 27
4.2.4 Monitoring 32
4.3 Design of Serverless DevOps Toolchain 32

5 Experiments and Evaluation **35**
5.1 Experiment on Serverless Container Services 35
5.1.1 Test task and System Description 36
5.1.2 Performance Properties and Evaluation 38
5.1.3 Result 38

6 Conclusions and Future Work **43**
6.1 Conclusions 43
6.2 Future Work 43

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.

Thus, in answering to how to solve the gaps and flaws when applying Agile into the 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].

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 user to build and run application on the cloud, without thinking about the servers [9]. It also allow developers to build 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 with out 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 cloud, which help the toolchain become more scalable. DevOps tools as a service [11] allow the cloud provider deliver a DevOps tool directly on it's 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 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 could help improve the DevOps toolchain.

1.1 Problem Statement

As per last paragraph, serverless computing gives developers alternative ways to deploy DevOps tools with the new cloud technologies other than traditional cloud virtual machines. There are several cloud providers that utilise the serverless computing. Among them, Amazon Web Services(AWS)[1] has the largest market share is the first cloud provide which provides the serverless computing services. According to the report from Gartner [12], the market share of AWS was 47.8% in 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 include new AWS cloud products that leverage the serverless computing

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

technologies. These products includes for instance, AMS Lambda[3] for function computing, AWS Fargate[4] for managed container services and AWS CodePipeline provided a managed continuous delivery pipeline as service[5] etc. AWS also gains the most popularity 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 is keep looking for ways to leverage serverless computing services in AWS in order to benefit the DevOps toolchains it builds for costumers.

However, despite the widely application of serverless technologies, and 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 of researches about how could serverless helps DevOps toolchain itself. Thus, our first research question is fill the gap by answering this question.

The second area we'd like to investigate in the project is related to the integrated toolchain which is powered by the tool-as-a-services in AWS. The integrated DevOps toolchain is delivered as a 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. While the traditional non-integrated toolchain is to have individual tools which are stand-alone and from different companies.

This newly emerged type of 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 of third party researches about the comparisons between these two. Our second research question is to giving audience a better

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

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

2. **RQ2:** How does the newly emerged integrated toolchain compared with the stand-alone toolchain in Amazon Web Services?

[3] https://aws.amazon.com/lambda/
[4] https://aws.amazon.com/fargate/
[5] https://aws.amazon.com/codepipeline/

1.2 Research Approach

We will implement our toolchain based on the DevOps practices and tools used by Eficode. In the design and implementation of teh toolchain, we will focus on the following DevOps practices: Version Control, Continuous Delivery and Monitoring. To answer the RQ 1, we will first investigate the current serverless offering in Amazon Web Services (AWS) which is one of the cloud services mainly used in the Eficode. We will first analysis the functionality that each service have, and how does these services could be used by our DevOps toolchain that deployed in AWS. Thus in the process of developing and deploy the toolchain, we could already partly answer the RQ1 doing case study on how the Serverless computing services be used in our DevOps toolchain. In addition how the serverless nature of these services could benefit the toolchain.

To further facilitate the result from last step, our next step of the study will be the case study on how serverless services in AWS helps our toolchain and the experiments. The experiments will be 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. We will also have the demo implementations which shows the answer to this research question.

To answer RQ2, The standalone toolchain will be used to compare with the DevOps toolchain build by the DevOps tools provided by AWS as a services. We will conduct a case 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 toolchain will be 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 comparison, we will simulate the same DevOps lifecycle of a demo Spring Boot web app on both toolchains. We will again measure the metrics in these 2 toolchains, the process will be similar to what we will 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 will introduce concepts within the scope of DevOps. We will 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. 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 will finally summarise our research and answer the research questions in Chapter 6.

The main contributions of this paper 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 team understand which toolchain cloud be selected based on the needs.

Chapter 2

Background and Concepts

In this chapter, we will introduce several main concepts related to our study.

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].

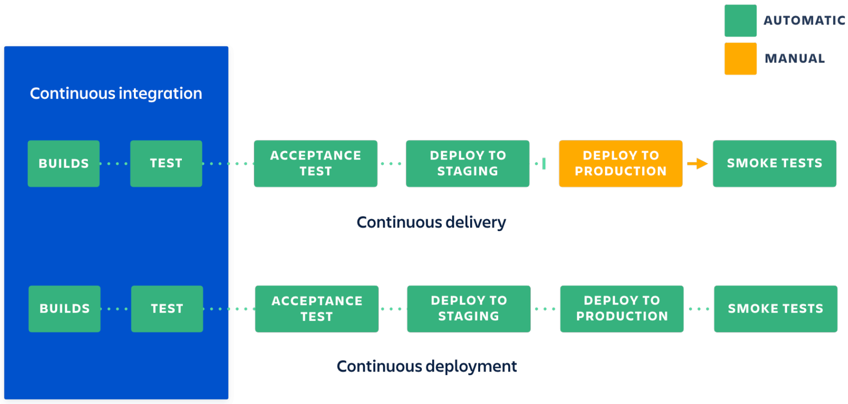

Figure 2.1: The relationship between continuous integration, continuous delivery and continuous deployment [25]

- *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.

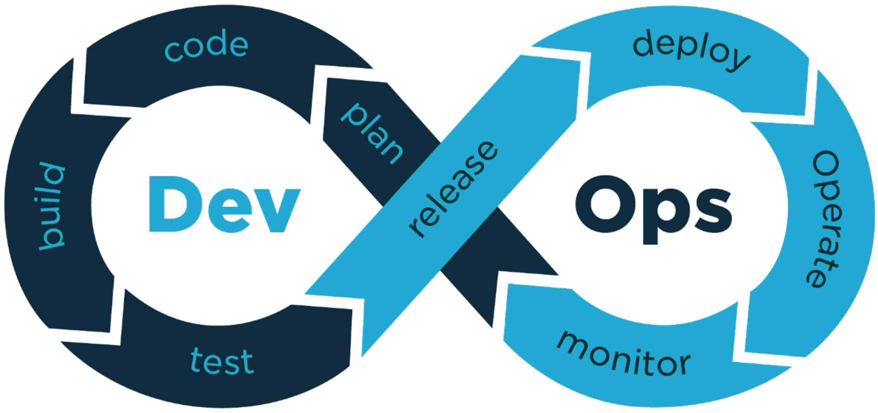

Figure 2.2: DevOps Practices and Workflow [37]

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.
Tools: Slack, Jira, Trello, Asana

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.
Tools: Puppet, Chef, Ansible

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.
Tools: Jenkins, Drone CI, Teamcity, GitLab CI/CD

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 [53] 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 [53] Infrastructure as code, an important DevOps practise we mentioned at 2.3.1 also relies on the version control.
Tools: GitHub, Gitlab, Bitbucket

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.
Tools: Zabbix, Promethus

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].
Tools: Robot Framework, Selenium, JMeter

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 [54]. 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 [55] 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][56]. The platform allows developers to run their code without managing the cloud virtual

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

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 first public cloud provider that provides serverless computing platform[57]. 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[58]. 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 [59] when deploying his/her application.

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

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 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 [55].

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" [60]. 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 [61].

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 [62] 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 [63]. 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 [64]. 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 [63]. 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 Toolchain

In the chapter, we will first present the case software project that will be built, tested, and deployed by our DevOps toolchain in the experiment. Then we introduce the design and implementation of our DevOps toolchain which acts as the basis of our experiments in CH5. Note that for the experiment that answering RQ 2, 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.

4.1 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 [65], 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 [66] 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 [67] 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 [68].

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 [69], by simplifying the configuration of Spring framework.

4.1.2 Project Description

```
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

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

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.

4.2.1 Architecture

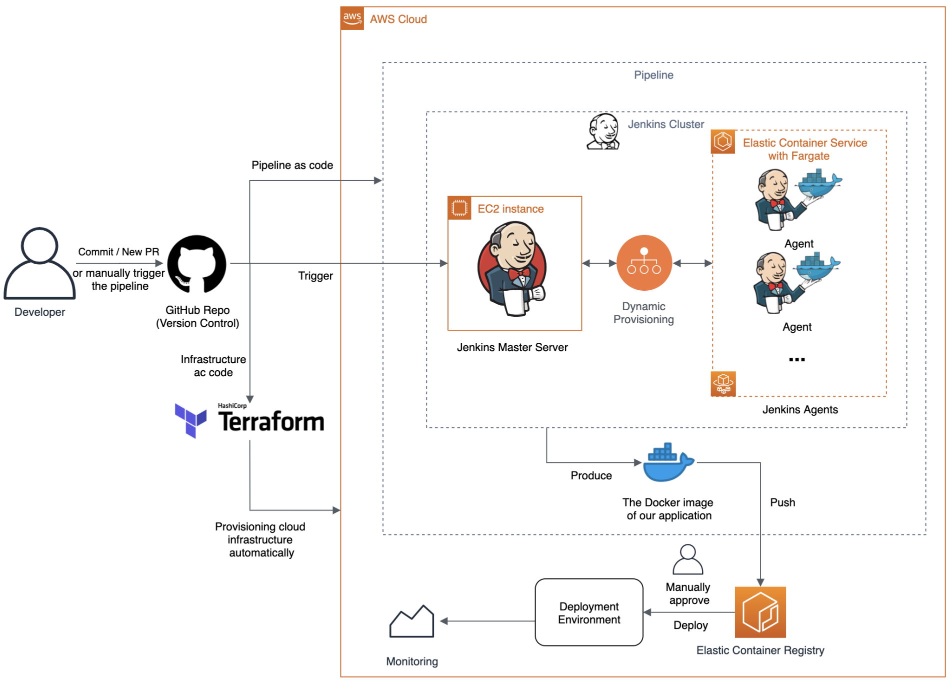

Figure 4.2: Architecture diagram of our DevOps toolchain

The toolchain implementation is based on the DevOps elements we presented in Chapter 2, and the DevOps practises from Eficode. Figure 4.2 shows the architecture of our DevOps toolchain. In here we 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 Version Control

Version Control System (VCS) is the process that record the changes in files set over time [70], and versioning the history of these files. VSC is suitable for track the development progress and manages the goal within a software development team [71]. Among all software for version control, Git is the most popular one nowadays. The survey [72] 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.

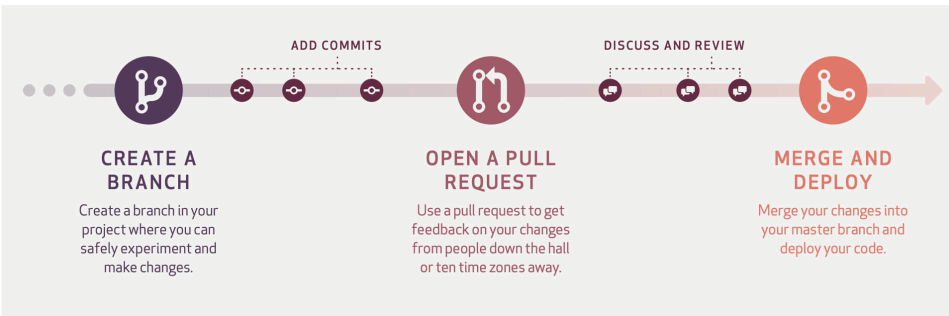

Figure 4.3: GitHub Workflow [73]

The Git flow [74] proposed in 2010 is a successful workflow for working with Git. Git flow has already widely used and has been approved by the software

[2] https://github.com/

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 [75] is being chosen as our workflow in the version control. The simplified version of this workflow is shown as in Figure 4.3

Several general principals followed by us when adapting GitHub flow, we refer the principals in [75] 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[3] 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 pipeline will also vary with the time point within the version control workflow. We will introduce this in detail on 4.2.3.

4.2.3 Continuous Delivery Pipeline

Tool Selection Considerations

In this section, we describe about our consideration when select tools used in the continuous deliver pipeline.

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

Continuous Delivery Pipeline The most popular server-based tools for build continuous delivery pipeline are Jenkins[4], Drone[5], GoCD[6] and Circle CI[7]. 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. However, 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.

	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 [8]	93 [9]	110 [10]	88 [11]
Price of self-hosted solution	Free	Free	$35 user/month	Free
Number of companies use it in the tech stack[12]	2634	82	1368	42

Table 4.1: Comparison of continuous delivery tools

Created by Kohsuke Kawaguchi in 2001, Jenkins is an open-source continuous integrating tool write with Java. It is suitable for a team of all sizes and varies of languages and technologies [76]. Furthermore, Jenkins also attracts software teams with it's easy-to-use and high extendibility [76] with 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[13] 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 [77] is defined as a "stage"[14] and each task within a step is

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

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[15] 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 of the build configurations of a software project in a single file.

For unit testing within the build stage, we using JUnit [16] as the tool for testing. For code analysis, we use SonarQube[17]. Both are one of the most common 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 [18] 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 [78]. 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. 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. Therefore we can configure the built environment (operating system version, dependencies) runs within 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

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

makes sure that the docker image could runs 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.

Pipeline OverView

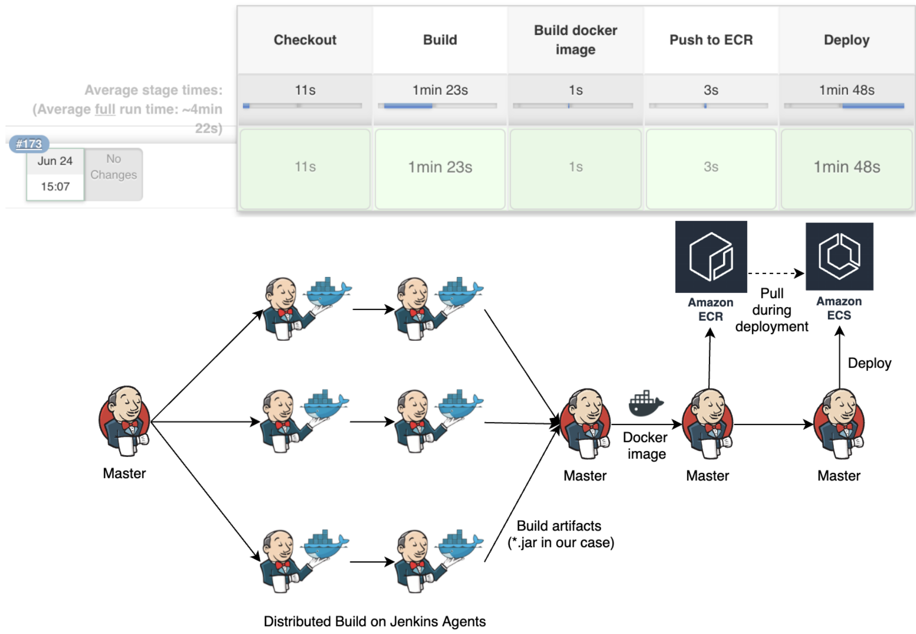

Figure 4.4: The Stages and Distributed Build in Our Pipeline

Figure 4.4 shows the 5 stages in our pipeline that shown in the Jenkins dashboard. The bottom part of this Figure shows the task distribution between master node and agent nodes. As we can see from the figure, when the master node start a job, it will create Docker container in AWS Fargate as agent. The agent will pull codes from VCS, build the code, and then send the build artifacts back to master node. After this the container will be terminated. The master node will continue executes the rest steps.

The considerations behind to our design is that, the first 2 steps takes most of time in our pipeline and according to Figure 4.5 runs more frequently than other steps [19]. The running time will be further extended when building larger project. These 2 stage 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 4.2.3, the build environment inside Jenkins agents that runs in Docker container is easier

[19] The reason will be discussed in next section "Pipeline Workflow"

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 different build environment for different projects, they eliminates 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.

We also notice that the Deploy stage also takes long time. However, we don't have it in the distributed build because: first, it is on the end of a pipeline so it will not block te further steps, second, the pipeline runs ths stage less frequently than first 2 stages as shows in Figure 4.5, thus there will be less possibility that there are many jobs runs at "Deploy" stage in parallel.

Pipeline Workflow

Figure 4.5 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 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 our case, we using AWS Elastic Container Service (ECR) on AWS Fargate to host our application. As we mentioned in the CH3, it allow us to run our containerized application without having to manage servers. So it is easier for us to make an functionality complete DevOps toolchain implementation compared with server based deployment environment, for example AWS Elastic Kubernetes Service (EKS). During deployment we use Martin Fowler's blue and green deployment strategy [79] which is natively supported by ECS. This means when a new deployment comes, the older version will continue serving until the newer version reach the stable status. This could significantly reduces the downtime in the deployment.

In the dev branch, we deploy the application to the staging environment. The deployment to staging environment should be automate, this is because the staging environment is only for testing, and only visible within the team. In the staging environment, we will conduct API smoke testing [80]. 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 an 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.

4.2.4 Monitoring

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

In CH3 we introduced AWS cloudwatch as one of the serverless service in AWS. In our toolchain, we will use it as the primary tool for monitoring. Wih 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 be seem at figure 4.6 Another service we introduced in CH3 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 cloud watch, and we use it for 2 cases.

Auto-Scaling the ECS Cluster with Alarm in Cloudwatch

Custom Project-Specific Metrics

4.3 Design of Serverless DevOps Toolchain

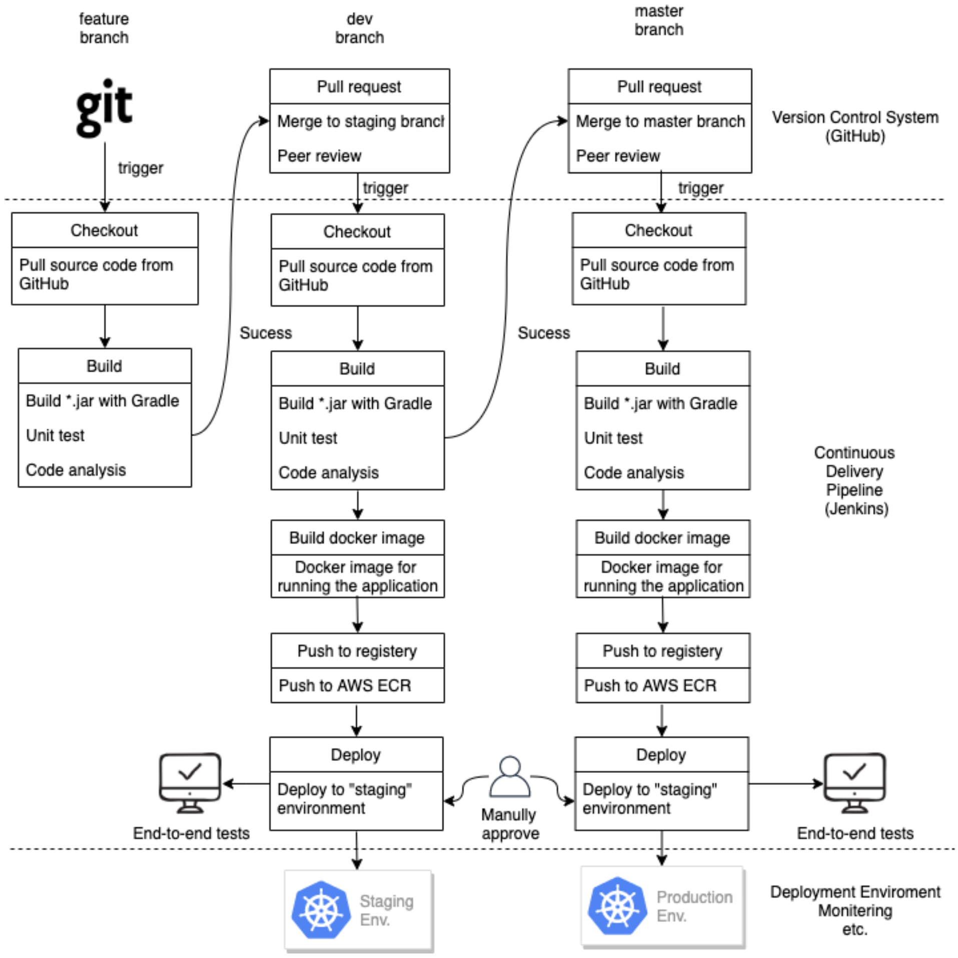

Figure 4.5: The Workflow of Continuous Delivery Pipeline in Our DevOps Tool-chain

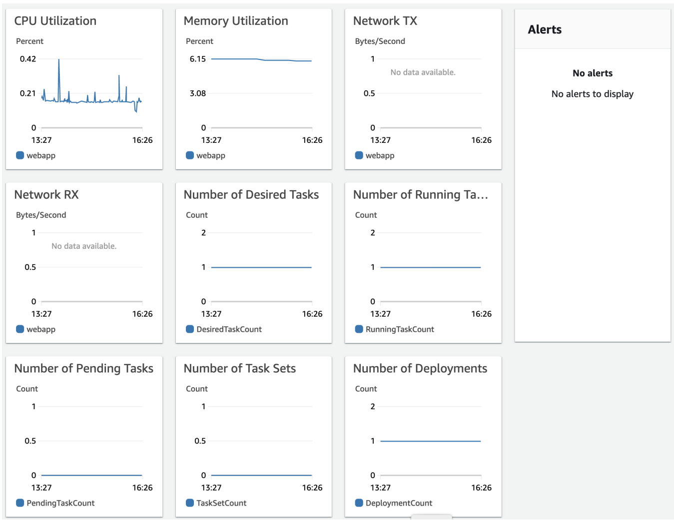

Figure 4.6: Cloudwatch Monitoring Dashboard

Chapter 5

Experiments and Evaluation

In this chapter, we describe our experiments regarding 2 research questions we proposed in chapter 1. The experiment based on the DevOps toolchain we implemented in Chapter 4. In the experiments, we compared the implementation of serverless with the implementation of traditional VM based cloud infrastructure. Thus in experiments, we also implement the solution with a different type of cloud environment (with/without serverless) as a comparison group.

In the first experiment, we will examine how does the serverless compute engine for containers (Amazon ECS on AWS Fargate) could be used in the continuous delivery pipeline(in our case, Jenkins) – the core element of our DevOps toolchain. The second experiment shows how does serverless functions (AWS lambda) could be used in DevOps toolchain. The last experiment focuses on answering research question 2, in which we will using compared 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 on Serverless Container Services

Nowadays, Docker[1] is being widely used as build agents in the continuous integration and continuous delivery (CI/CD) pipelines. This means the pipeline will execute certain steps inside ephemeral Docker containers [81]. It is easier to manage build dependencies in Docker container. Besides, container based agent requires less effort to maintain.

(This paragraph might be moved to CH4 later, to justify our design) The Docker agent has already been supported by many CI/CD tools, for example container job[2] in Azure DevOps[3], Docker agent in TeamCity [4], Docker agent[5] in Jenkins and

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

[2] https://docs.microsoft.com/en-us/azure/devops/pipelines/process/container-phases

[3] https://azure.microsoft.com/en-us/services/devops/

[4] https://www.jetbrains.com/help/teamcity/build-agent.html

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

docker runner[6] in Drone[7]

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 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 belongs to a auto scaling group that will scale up when CPU Utilization rate reach 70%. The initial size for 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.

[6] https://docs.drone.io/runner/docker/overview/

[7] https://drone.io/

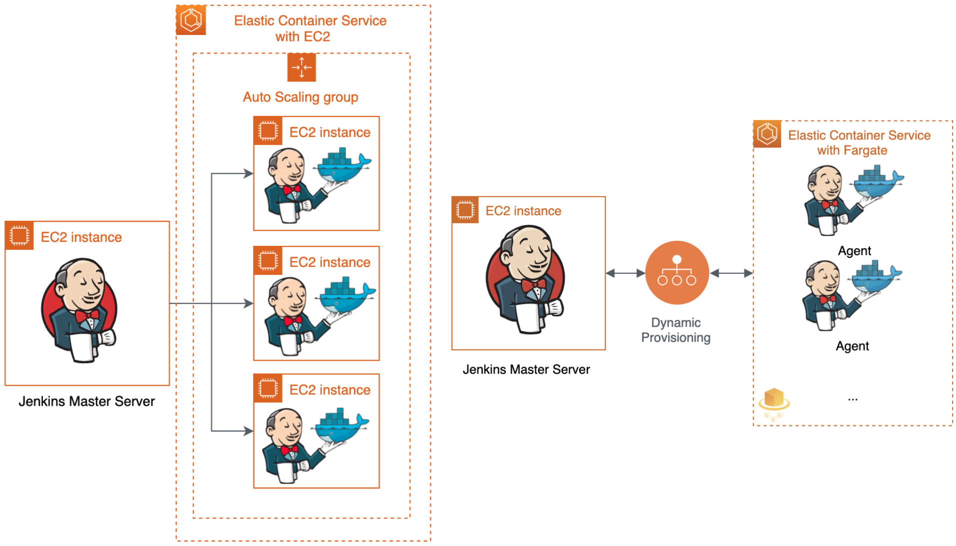

Figure 5.1: Architecture diagram of the test Jenkins cluster with agents running in traditional virtual machines (left) and on ECS with AWS Fargate (right)

Hardware

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

The experiments are conducted on Amazon Web Services (AWS). The hardware of Jenkins master node in both experiment groups is the same, which is EC2 instance of type t3.medium with 2 virtual CPU, 4 GB RAM and 30 GB disk. The EC2 instances as worker node is 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 operation 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 [8]. 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, the automated testing and code analysis is being integrated with this step, and being conducted by plugin of Gradle.

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

Here shows the result of this experiment.

Runtime

We first compare the runtime of these 2 setups. Except 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 single task. The traditional VM has 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 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 agent once AWS finishes the initialization of underlay infrastructure

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

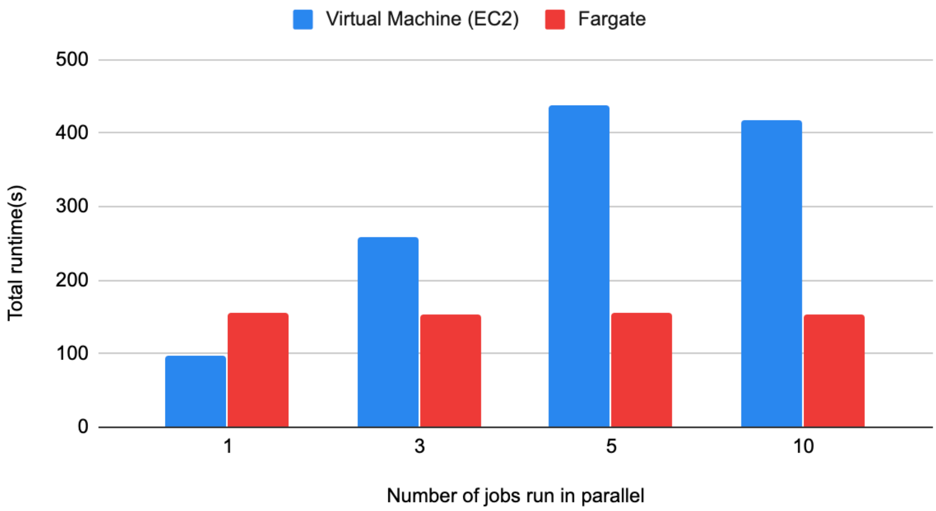

Figure 5.2: Runtime of Pipeline with Different Jenkins Agents

that runs the serverless container. This takes significantly longer time. This shows the one of the limitation 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 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 resource. On the other hand, when we runs multiple agent 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 easy 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 reach 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 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 patten, the AWS still more "hesitate" to create new instances compared with in the Fargate.

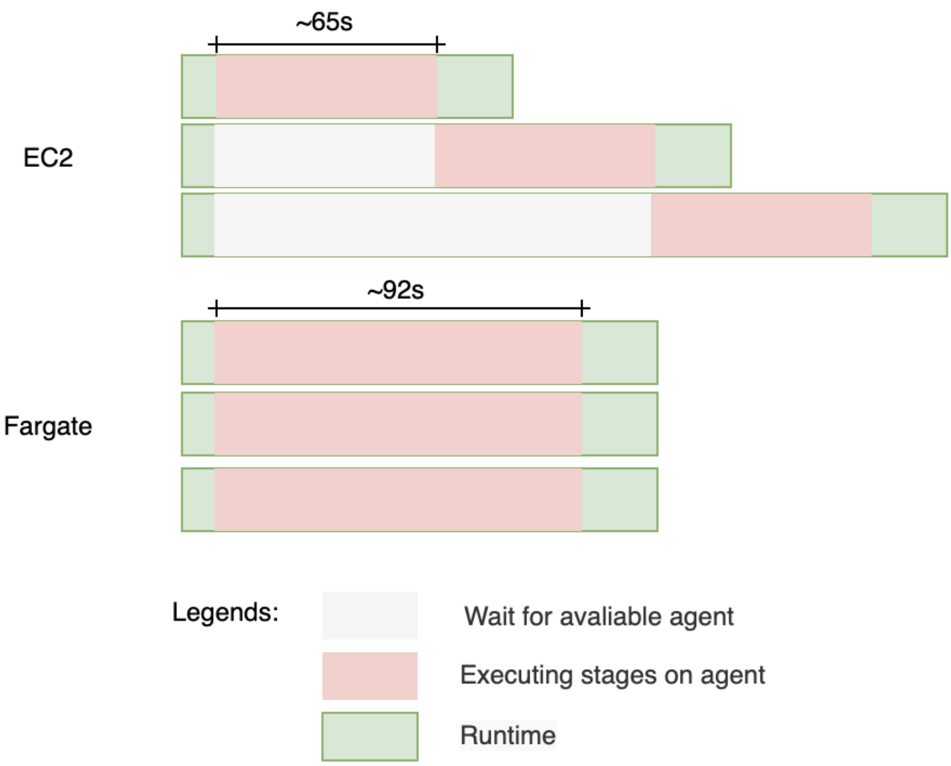

Figure 5.3: Execution Mode of Pipelines with Different Jenkins Agents

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 host environment" [78] feature of Docker container that we mentioned in 4.2.3.

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 [9]. Thus in our experiment set-up (2vCPU, 2GB RAM) the price should be \$0.01813 per running agent for one hour's runtime. In AWS EC2 our cost depends on 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 [10]

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

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

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

[54] 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.

[55] 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.

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

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

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

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

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

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

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

[63] 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.

[64] 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).

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

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

[67] 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).

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

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

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

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

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

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

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

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

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

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

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

[79] Martin Fowler. Bluegreendeployment, 2010. *http://martinfowler.com/bliki/BlueGreenDeployment.html*, 2010.

[80] Smoke testing (software) - wikipedia. `https://en.wikipedia.org/wiki/Smoke_testing_(software)`. (Accessed on 06/22/2020).

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