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

7th 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
7th June 2020

Contents

Preface v

1 Introduction 1
1.1 Problem Statement 2
1.2 Research Method 3
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 Design of Server-based DevOps Toolchain 23
4.1.1 Architecture 23
4.1.2 Version Control 24
4.1.3 Continuous Delivery Pipeline 24
4.1.4 Monitoring 24
4.2 Design of Serverless DevOps Toolchain 24

5 Experiments and Evaluation 25
5.1 Experiment on Managed Container Services 25
5.1.1 Test task and System Description 25
5.1.2 Performance Properties and Evaluation 27

6 Conclusions and Future Work **29**
6.1 Conclusions 29
6.2 Future Work 29

Chapter 1

Introduction

The Agile Manifesto [1] drafted by Kent Beck etc. in 2001 created the Agile software development method. Since then, this new software development method has draw attention to the industry and more and more companies started to apply Agile in the production. 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 technologies. These products includes for instance, AMS Lambda[3] for function

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

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.

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 cloud technologies bring, 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, It is still unclear if it is better than the traditional standalone toolchain.

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?

1.2 Research Method

To answer the RQ 1 we will first build a DevOps toolchain with the popular tools used in the industry. The toolchain will be deployed on Amazon Web Services (AWS) which is the cloud services used by Eficode. So in the process of developing and deploy the toolchain, we will answer the RQ by research how could serverless computing in AWS benefit our toolchain. We will first conduct a literature review on new cloud technologies in chapter 3, in which we also introduce the implementation of these technologies by different cloud vendors. In this Chapter, we will also discuses which cloud technologies can benefit the DevOps toolchain.

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

In the next step (chapter 4), we will design and conduct experiments which evaluating the benefit that each cloud technologies we researched in chapter 3 can bring.

The evaluation will be done by comparing the metrics measured from the toolchain with and without using certain cloud technologies 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 eliminates 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 [14]. 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 [15]. 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 [16] 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 [17].

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 [18] 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 [19]. 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 interaction is the base practice of all practices within CI/CD, and continuous delivery/deployment is based on the continuous interaction [19]. 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 [20]. The continuous interaction 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 [21].

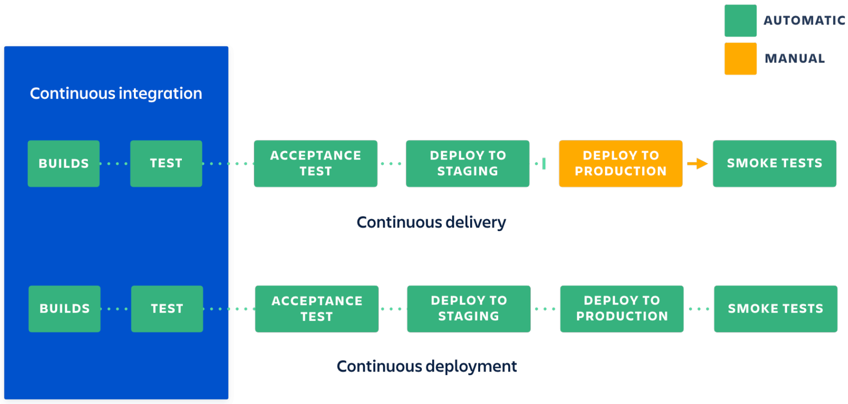

Figure 2.1: The relationship between continuous integration, continuous delivery and continuous deployment [19]

- *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 [22].

With the help of these 2 practices, for each developer in the team, the workflow [20] in continuous interaction 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.[23]This means the software always maintains a high quality and in a deployable state[24]. It is a subset of agile, which focuses on the software delivery[25]. From the last section, we introduce the concept of continuous interaction. The continuous delivery is based on continuous interaction 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 [24][23] 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 [26]. 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 [27].

DevOps is the extension and evolution [28][29] of Agile. DevOps and Agile both driven by the collaboration ideology and the adoption of DevOps needs Agile as the key factor [28]. DevOps has a different focus on agile. DevOps focus on the delivery while agile is focused on the development with the requirement and customer. Figure 2.2 shows the workflow and practices of a team working under DevOps.

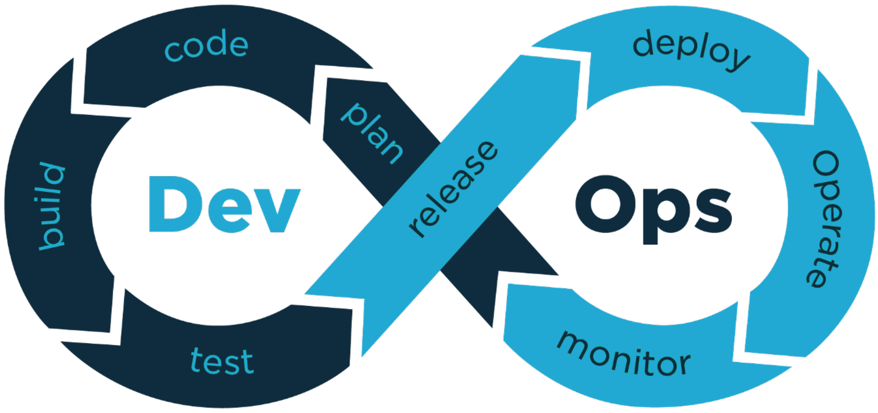

Figure 2.2: DevOps Practices and Workflow [30]

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 [29]. 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 [31].

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. [31] The information sharing is also important [32]. 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 [32]. 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 [33].

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 [34]. However the people operation team will take development responsibility, and the developers will also put their hands-on operation and management[35]. 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 [33][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[36]. 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 [37]. 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 [38]. 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" [32]. The Infrastructure as Code (IasC) means to define everything in the software infrastructure level as code [39]. 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 [37] 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 [40]. 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 [41]. 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 [42]. 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 [32].

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 [41].

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][43][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 [42][44][45] 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 [44]. 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 [46] and [44], 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 [46] 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[45]. 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 finds 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 [41].
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 [47]. 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 [48] 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][49]. 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[50]. 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[51]. 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 [52] 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 [48].

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" [53]. 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 [54].

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 [55] 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 [56]. 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 [57]. 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 [56]. 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 introduce the design and implementation of our DevOps toolchain which acts as the environment of our experiments in CH5. The pipeline implementation is based on the DevOps elements we presented in Chapter 2. For conducting the experiment that answering RQ 2, we implement two different continuous delivery pipelines design with two set of tools respectively, one with tradition server based tool while another one with the serverless DevOps tools from AWS. We will introduce both design in this chapter as well.

4.1 Design of Server-based DevOps Toolchain

In section, we present our design of DevOps toolchain which based on virtual machine.

4.1.1 Architecture

Figure 4.1 shows the architecture of our DevOps toolchain. In here we only presenting architecture in a more general level. The detail architecture of each competent will Introduced in following sections, in both text and graph.

When the developer pushed an new commit to the repository in GitHub [1], Github will send an HTTP POST request that contains 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 git repository, then runs the docker containers with required build environment and build the project. In the end, an docker image for running the project will be created and be pushed to the container registry of AWS. Depends on the git branch that developer committed to, the project will be deployed to different development environment.

As shows in Figure 4.1, except version control, the whole environment is runs in Amazon Web Services,

[1] https://github.com/

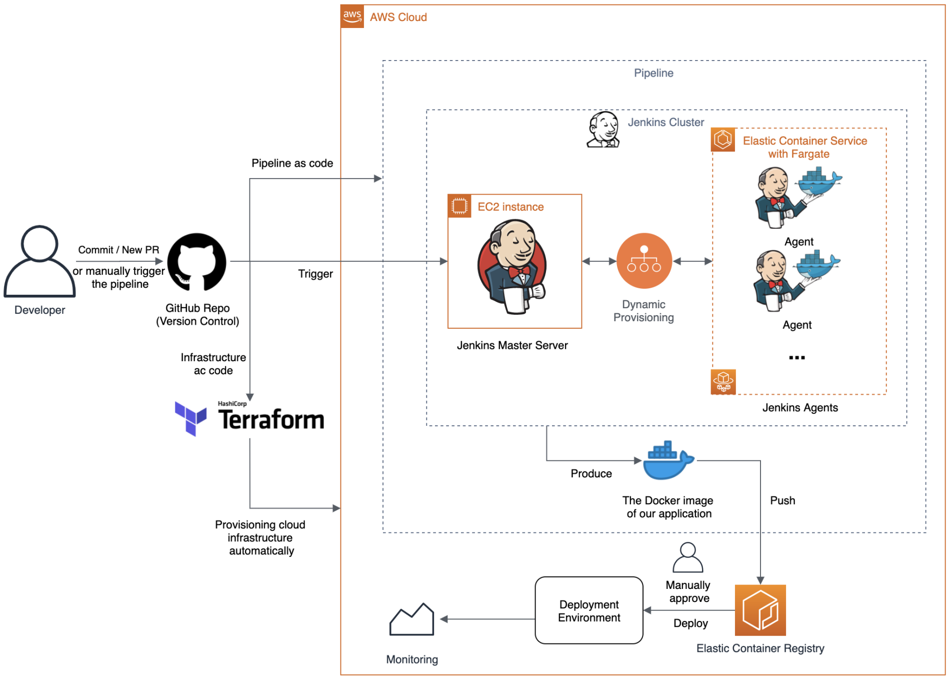

Figure 4.1: Architecture diagram of our DevOps toolchain

4.1.2 Version Control

4.1.3 Continuous Delivery Pipeline

4.1.4 Monitoring

4.2 Design of Serverless DevOps Toolchain

Chapter 5

Experiments and Evaluation

In this chapter, we describe our experiments regarding to 2 research questions we proposed on chapter 1. The experiment based on the DevOps toolchain we implemented in the Chapter 4. In the experiments we compared the implementation on serverless with the implementation on traditional VM based cloud infrastructure. Thus in experiments, we also implement the solution with 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 continuous delivery pipeline(in our case, Jenkins) – the core element of our DevOps toolchain. The second experiment shows how does serverless functions (AWS lambada) could be used in DevOps toolchain. The last experiment focus on answering research question 2, in which we will using compared continuous delivery pipeline composed with fully-managed serverless DevOps tools in AWS with our Jenkins-based pipeline that runs on the virtual machine.

5.1 Experiment on Managed Container Services

This experiment is a controlled experiment which examine how serverless container service could affect 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 following steps:

1. *Checkout*: Pull the most recent change from Github repository

2. *Build*: Build the application with Gradle, with automate testing with JUnit integrated in 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 last step to the AWS elastic cloud registry (ECR) for the further deployment.

In these 4 steps, the step ”Build” and ”Checkout” is being done in parallel with in the ECS cluster. As we mentioned in CH4, when new job started in the Jenkins master server, jenkins will provisioning 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 experimental group we replace AWS Fargate with traditional VM, which is EC2 in the Amazon Web Services. We manually create EC2 virtual machines and let Jenkins runs the same continuous delivery pipeline on it. The parallelization pattern remain the same, this means as in the controlled group, only first two steps are being run distributively in the Jenkins nodes.

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

Hardware

The hardware of 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.small with 2 virtual cpu, 2 GB RAM and 30 GB disk. Each EC2 instance has 2 Jenkins executer according to the Jenkins default setting. This means 2 container could run in parallel on each EC2 virtual machine.

In the control group, which is the implementation we presented in CH4, the Jenkins agents runs on AWS ECS powered by AWS Fargate. The virtual hardware resources that are allocated to each serverless container is 1 virtual cpu, and 1 GB of RAM. This make 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 own developed docker image which you can find at [1]. The docker image includes essential dependencies that will be used for build the Spring Boot application, and the base image which allow container connects Jenkins master as an agent.

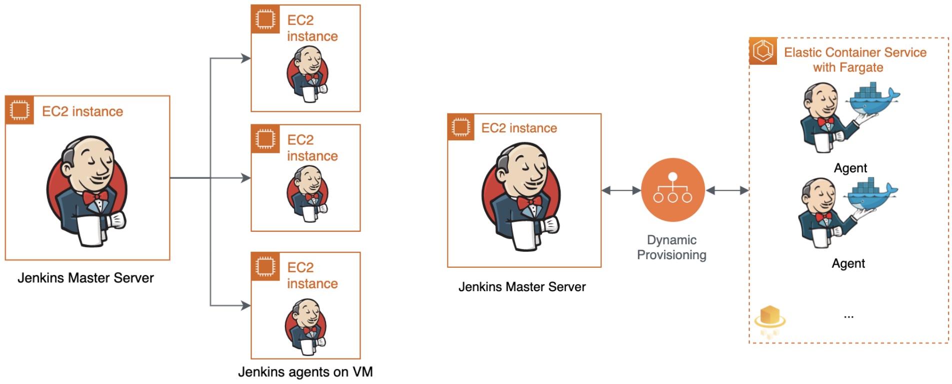

Figure 5.1: Architecture diagram of the test Jenkins cluster with agents running in traditional virtual machine (left) and on ECS with AWS Fargate (right)

5.1.2 Performance Properties and Evaluation

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

- *Runtime* describes the total time for finishing all the tasks.
- *Cost Structure* describes the daily cost of 2 setups under the same workload
- *Resource Utilization* describes the average CPU/RAM usage for each instance during a single run of the pipeline.

To shows how does the 2 setups performance within the teams with different sizes, we run by run different number of tasks parallel through the pipeline. This simulates the different team size, besides, it could also shows the scalability when comes to the need of task parallelization in bigger organizations.

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

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. https://www.gartner.com/en/newsroom/press-releases/2019-07-29-gartner-says-worldwide-iaas-public-cloud-services-market-g (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] Jim Highsmith. What is agile software development? *crosstalk*, 15(10):4–10, 2002.

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

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

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

[18] 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.

[19] 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.

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

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

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

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

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

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

[26] 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.

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

[28] 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.

[29] 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.

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

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

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

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

[34] 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).

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

[36] 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.

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

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

[39] 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.

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

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

[42] 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.

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

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

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

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

[47] 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.

[48] 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.

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

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

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

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

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

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

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

[56] 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.

[57] 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).