KARELIA UNIVERSITY OF APPLIED SCIENCES
Degree Program in Information and Communication Technologies

Jesse Heiskanen

COMPARING DEVOPS PROCESSES IN DIFFERENT CLOUD PLATFORMS FROM A SOFTWARE DEVELOPERS PERSPECTIVE

Thesis
May 2020

<table>
<tr><td colspan="2"> </td><td colspan="2">OPINNÄYTETYÖ
Toukokuu 2020
Tieto- ja viestintätekniikan koulutus
Tikkarinne 9
80200 JOENSUU
+358 13 260 600</td></tr>
<tr><td colspan="4">Tekijä
Jesse Heiskanen</td></tr>
<tr><td colspan="4">Nimeke
DevOps-prosessien vertailu eri pilvipalvelujen välillä ohjelmistokehittäjän näkökulmasta

Toimeksiantaja
Valamis Group</td></tr>
<tr><td colspan="4">Tiivistelmä

Tässä opinnäytetyössä vertailtiin kahta Valamis Groupissa käytettyä DevOps-alustaa sekä niihin liittyvien prosessointiaikojen tuloksia. Vertailun kohteena oli automatisoitujen prosessointiaikojen lisäksi, työnkulku ja alustojen käyttöliittymät. DevOps-alustojen samankaltaisuudesta huolimatta prosessointiajoissa paljastui huomattavia eroja.

Opinnäytetyössä esitellään molempien DevOps-alustojen rakentamisprosessi. Työssä esitellään myös palveluihin luotavan yksinkertaisen DevOps-pipelinen rakentaminen ja siihen liittyvän automatiikan luominen. Pipelinet luotiin molempiin palveluihin käyttäen niiden tarjoamia omia työkalujaan. Työssä vertailtiin työnkulkua pipelineä käyttäen, sekä automatisoidun pipelinen prosessiaikoja testattiin kahden eri testin avulla.

Opinnäytetyö esittelee DevOpsin tarkoituksen ohjelmistokehityksessä, sekä sen alkuperää ja kehitystä jatkuvasti kasvavana kulttuurina ohjelmistokehittäjien keskuudessa. Opinnäytetyö käsittelee myös ohjelmistokehittäjien käyttämää työnkulkua DevOps prosessin sisällä.</td></tr>
<tr><td rowspan="3">Kieli

englanti</td><td>Sivuja</td><td>41</td></tr>
<tr><td>Liitteet</td><td>1</td></tr>
<tr><td>Liitesivumäärä</td><td>2</td></tr>
<tr><td colspan="4">Asiasanat

DevOps, Azure DevOps, Amazon Web Services</td></tr>
</table>

THESIS
May 2020
Degree Program in Information and Communication Technologies

Tikkarinne 9
80200 JOENSUU
FINLAND
+ 358 13 260 600

Author
Jesse Heiskanen

Title
Comparing DevOps Processes in Different Cloud Platforms From a Software Developer's Perspective

Commissioned by:
Valamis Group

Abstract

In this thesis a comparison between two DevOps platforms and the process times in those platforms is created. The comparison includes differences in the automated process times, differences in workflows and differences in user interface. Even if the DevOps platforms seem similar, there were notable differences found in the processing times.

In this thesis, the setting up of each DevOps platforms are presented. Also, the creation of a simple DevOps pipeline is show and how the automatic functions work inside the pipeline is explained. The pipelines are created separately inside both compared DevOps platforms. The workflow and the process times in the automated pipelines are then compared. Data for the process time is composed of two different sets of tests.

This thesis also introduces the meaning of DevOps, how DevOps was created and what it means for the software developer. The workflow in the DevOps processes is also displayed. Both of these platforms are used at Valamis Group.

Language	Pages	41
English	Appendices	1
	Pages of Appendices	2

Keywords

DevOps, Azure DevOps, Amazon Web Services

CONTENTS

1 INTRODUCTION 6
2 DEVOPS PROCESS 7
3 DEVOPS CLOUD PLATFORMS..12
3.1 Microsoft Azure DevOps...12
3.2 Amazon Web Services (AWS) 13
3.3 Google Cloud Platform 14
4 DEVOPS PROCESS TESTING 14
4.1 Setting up the environments 16
4.1.1 Setting up the Azure DevOps 16
4.1.2 Setting up the Amazon Web Services 21
4.2 DevOps Workflow Process 28
4.2.1 Workflow in Azure...28
4.2.2 Workflow in AWS 29
4.3 User interface comparison 31
4.4 Process Time Measurement 32
4.4.1 Process Time in Azure...32
4.4.2 Process Time in AWS 33
4.5 Process time in failed test 34
4.5.1 Bad deploy in Azure...35
4.5.2 Bad deploy in AWS 36
5 SUMMARY OF RESULTS 36
6 REFLECTIONS...39
REFERENCES 41

Appendices

Appendix 1 azure-pipeline.yaml

Abbreviations

DevOps	Software Development (dev) and IT Operations (ops), is a combination of practices striving to bring continuous value to customers
AWS	Amazon Web Services, cloud service provided by Amazon
GCP	Google Cloud Platform, cloud service provided by Google
CI/CD	Continuous Integration/Continuous Delivery is a method used in DevOps to ensure a frequent delivery to customer by using automated tools
IaC	Infrastructure as Code is a way to modify infrastructural services by configuring system files
SCM	Source Control Management is a tool and practice used to track and manage code changes.
AI	Artificial Intelligence is a software that can-do different functions and possibly learning from those functions to make choices on its own.
IDE	Integrated Development Environment is graphical application used to edit and debug code.
UI	User interface is the display of site/software that user interacts with.
IAM	Identity and Access Management is used to manage access for users to different services and resources provided in AWS.

1 INTRODUCTION

Nowadays, the DevOps developing process is the go-to way to give customer great value by giving them continuously smaller updates instead of creating a single big update. This way of working helps other developers that are just jumping into a project but also the customer. By having these constant small updates helps the customer to be as close as possible with development and increases their understanding about where development is heading. This enhances their possibilities to give feedback about development. As the name DevOps suggests it is striving to combine development and operations processes so that the two teams would work closer. Because DevOps is not only a software platform but more like a combination of cultures, practises and the implementation of DevOps into to a workflow will not be an easy task. When applying the DevOps culture to an old way of working within a project it is important to understand that this change will take extra resources. Good way to start implementing the new workflow would be by first pinpointing the process that needs most attention rather than trying to transform every workflow to be using DevOps as a solution.

This documentation will present more about what DevOps can mean and compare differences between the two most used DevOps cloud solutions; Microsoft Azure DevOps and Amazon Web Services (AWS). This thesis is part of an assignment from Valamis Group Oy, which is the company I work for. As a base for this thesis Valamis needed the documentation for new employees about the workflow of Azure DevOps in customer projects. Valamis and I decided to expand the thesis' area, and we chose to include the other DevOps platform that had been used in few customer projects at Valamis Group. We planned to compare the differences between the two platforms. I will be trying to find if there are any notable differences in workflow or process time.

I would like to note that previously before this project I had only used Azure DevOps as the DevOps service cloud platform in my daily work and I had no earlier experience when using AWS. In Valamis Group Azure DevOps is the more

common DevOps platform but there are few projects that use Amazon Web Services.

2 DEVOPS PROCESS

Introduction of agile development methodology was a huge leap towards more organized and ideal development cycle. But where agile tends to solve the communication problems with developer and the client DevOps strives to also solve similar problem between developer and operations. Therefore, many view DevOps as the logical continuation of agile method. This can also be noticed when looking into DevOps culture since DevOps shares multiple similarities with agile development. DevOps culture got introduced because it was noticed that the developers and IT operators rarely worked together as a team toward the same goal, but they worked as separate units. [1.]

DevOps is not supposed to make developers do operational work or operations doing development work so that another unit could be released. It is to make developers and system operators work together with as minimal barrier between them as possible. Also one big reason to introduce the DevOps culture to a workflow is to make updates more smaller and frequent and automate some parts of the software workflow so that developers and operations will be able to work more efficiently rather than having to do for example multiple build tests for every small update by themselves. [2, p. 14]

Since DevOps is a constantly growing trend, more and more companies are starting to implement the DevOps into their development cycle and only a fraction has not heard of DevOps as shown by search done by Dimensional Research in 2018 (Figure 1).

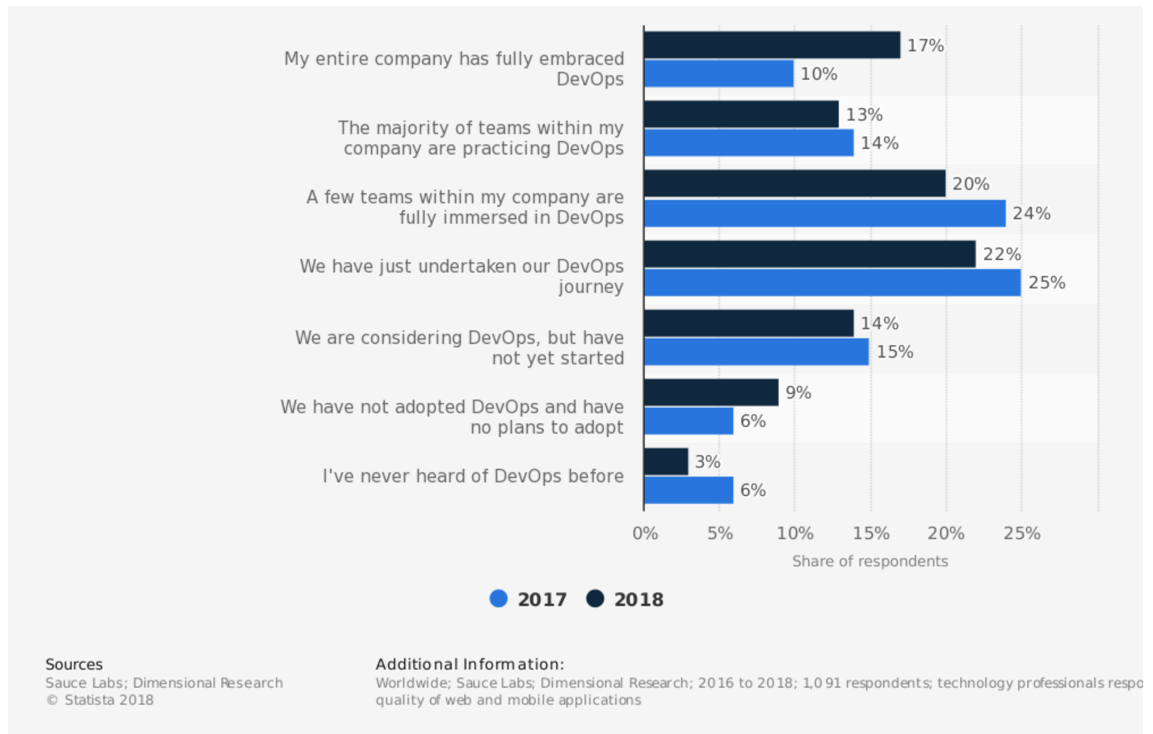


Figure 1. Growth of DevOps [3].

In the ideal scenario of DevOps, the developers receive constant feedback on their work which allows the developers tackle possible errors faster and implement, validate and deploy new code changes to the production environment at a constant pace. This way the time required to deploy code changes to the environment is shortened and the company and the customer will gain more value from the increased quality of work. [4.] This constant process of code validation and deployment is called continuous integration/continuous delivery or CI/CD in the DevOps environment.

Continuous integration ensures that issues are found early and all parts that are essential for the process platform are kept on the same page. Continuous delivery on the other hands strives to ensure that customer gains functional software, updated in small bits as frequent as possible. [2, p. 55.]

Because big part of DevOps services happens in cloud also system's activities and administrative tasks can be managed as a software code. This is called Infrastructure as Code, IaC for short. In DevOps environment this means using

different configuration files to handle system operations and automate certain tasks for example restarting virtual or physical machines if needed. [5.]

DevOps process can easily be confused only as a continuous integration/continuous delivery (CI/CD) but should not be, because DevOps and CI/CD are related but are not the same thing. DevOps is more than just CI/CD and usually CI and CD are included in DevOps. The big part of the CI/CD process is automated by using something called pipelines. Multiple pipelines with different stages and jobs inside those stages can be made for specific environment, for example one pipeline for development environment, one for staging environment and one for production environment. Since developer may not need to have same tools on development and production environment, customizing pipelines for different environments is a possibility. [2, p. 85-87]

The most basic pipeline usually uses at least some these 5 tools. These tools include the Framework for CI/CD, the Source Control Management (SCM) tool, the tool for Build Automation, the Web Application Server, and the tool to help with Automated Code Testing. Framework for CI/CD is a tool used to guide the CI/CD process by working with other services and tools in the pipeline. This is the brain of CI/CD process. As an example, Jenkins is one of the most popular CI/CD tools. SCM is used to control code in repositories which makes versioning the code easier. SCM can also be used for code backups and the most common SCM tool is Git. To make code into a deployable format or to make it executable a tool for building the code is needed. These Automated building tools compile, test and deploy the code. Most common building tools include Maven, Gradle and Ant. Web Application Server is the endpoint where to deploy the code. Common tools include Tomcat and JBoss. Lastly to ensure the quality of the deployed code some testing tools can be integrated to CI/CD process. JUnit is quite common tool used for testing. These pipelines can be customized with using different plugins and extensions to help user further automate building, testing, deploying, adding new integrations and many other functions. There are many community created extensions for multiple purposes. This makes developing and configuring the pipelines integral part of DevOps culture. These pipelines change a workflow a bit compared to agile development. [6.]

Here is an example of one type of workflow that uses these pipelines and CI/CD integration. The workflow can be divided to four different stages as presented in the figure 2. These stages are Planning, Development, Delivery and Monitoring. These stages can run parallel to each other, which ensures continuity.

First stage is “Planning”, in this stage developers may have sprint meetings where the goal of the upcoming sprint with the customer is decided. In this stage it is important for developers to visualize the goal and possible obstacles. Different tools that are used include for example Kanban boards that are used also in agile development. After the goals are set, it is time to start the development process. Second stage is “Development” where code changes are constantly committed to code repositories and merged to make a deployable package. This is the continuous integration phase of DevOps. Before deploying a build, unit and code quality are tested automatically. These tests should ensure stability and possibly collect some metric data that can be compared to earlier builds. Further some manual testing may be done on test environments before delivering the changes to customer environments.

In the third stage, “Delivery”, when earlier tests succeed, and the project team is ready to deploy the update to customer is time to move to the continuous delivery phase. This phase will need some help from the IT operations team to ensure that deployment is a success. Also, some parts of this phase can be made automatically depending on how the pipeline is configured. When changes are deployed in the production environment the team currently working on a project can start moving to the fourth stage “Monitoring” or sometimes called “Operating”. As the name suggests during this stage new deployed build is further monitored when used by a customer to aim for high availability and least amount of downtime as possible. Here the possible issues are tracked and if any issues are found they are logged as tickets into an issue tracking service for example. Given the importance of issue, fix should be made as soon as possible. During the delivery and the monitoring stages the operations team is working closely with the development team. This is what the DevOps is all about. The DevOps workflow can be

similar to the development with agile way, but with DevOps culture, the IT operations and automation are mixed in.

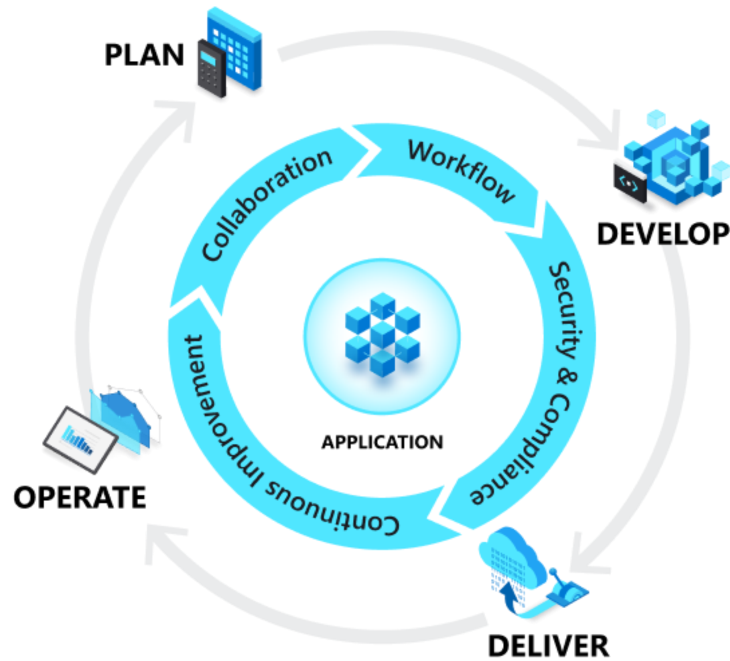


Figure 2. Application lifecycle [7].

During all these stages it is important to maintain communications inside the teams between the developers and operations, but also maintain the communication with the customer. This ensures that the customer receives the product they are paying for and in the deadline set for the update.

When IT operations are working as closely as possible with the development team, the quality of the service can be maxed out, and the communication barrier is shrunk. This means it is easier to see IT Operations and Development team as one functional team, rather than separate entities. Automation on the other hand gives more time for developers to work on actual solutions rather than doing tedious and monotonous tasks time after time.

3 DEVOPS CLOUD PLATFORMS

The DevOps trend has also caused the competition between different cloud service providers to intensify because a big part of the DevOps process and automation happen in the cloud. The most notable of these cloud solutions are Amazon Web Service (AWS), Microsoft Azure and Google Cloud Platform (GCP). AWS is the oldest and the biggest platform currently at the time of writing this thesis. AWS was launched in 2006, Azure in 2010 and GCP in 2011. Because AWS was the first cloud domain it has had time to expand its network all around the world, and it has also had time to develop its platform and different tools for the longer time compared to other providers. In 2019, AWS held around 30% of cloud market shares, followed by Azure with 16% and GCP with 10%. [8.]

Workflow in AWS and Microsoft Azure seems be quite similar to each other because they offer comparable functions regarding integration, building & testing, automation, deployment and monitoring but they offer their own version of these services. Both cloud services provide support for 3rd party tools.

3.1 Microsoft Azure DevOps

Microsoft Azure DevOps is the Microsoft's take on DevOps solutions. This solution is the evolved form of the former Visual Studio Team Services. Azure DevOps includes five different services: Azure Boards, Azure Repos, Azure Pipelines, Azure Test Plans and Azure Artifacts. These services are extensible and flexible, and they can be used with different platforms and clouds so the user may choose not to use the default Azure cloud solution when using these Azure tools. Azure DevOps also offers a variety of extensions and support for user created extensions. There are extensions for example for Docker, Slack, GitHub, SonarQube and AWS tools. Many of these extensions are offered free, but some may require paid subscription.

Azure Boards is the planning tool used to track the work and backlogs of the projects. Azure Repos is the Git hosting service with pull requests, reviews and unlimited repositories. Azure Test Plans is used for manual and investigating testing, and Azure Artifacts is used to manage the public and private packages of the project. Azure Repos, Test Plans and Artifacts can be integrated with Azure Pipeline's CI/CD. [9.]

Azure DevOps Services' pricing model will mostly depend on how big of a team is working with Azure Cloud. Even with the free version users gain access to Azure Boards, Azure Repos, Azure Artifacts, Azure Test Plans and Azure Pipelines. These CI/CD pipelines costs change how much traffic and how many multiple parallel jobs can be done at the same time. Azure Portal is also needed to host virtual machines when deploying the test apps. Costs for Azure Portal can be changed with different subscriptions. For the testing in this study I will be using the free subscriptions for both Azure DevOps and Azure Portal.

3.2 Amazon Web Services (AWS)

Amazon Web Services or AWS for short is the biggest cloud solution when comparing the market shares. Because AWS is much older than other competition it has had time to develop multiple different developer tools to use in the DevOps process. These tools include AWS CodeBuild, AWS CodePipeline, AWS CodeDeploy, AWS CodeStar, AWS Cloud Development Kit, AWS X-Ray, AWS CloudWatch, AWS CodeCommit and AWS Device Farm to name a few. [10.]

In AWS, the CodePipeline is built by first defining the CodeCommit or the source, like GitHub. After that the building and deploying stages can be configured. The workflow can then be configured to use also other services and extensions in the build and deploy stages. AWS CodeBuild is a build service that can process multiple builds concurrently. AWS CodeDeploy is used to make code deployments to environments. [11.] For project's monitoring AWS provides AWS X-Ray and Amazon CloudWatch. The user can choose which of these services are used so

every tool mentioned comes out of the box with AWS subscription, but they are not necessary to use.

In AWS the user only pays for the individual services, so if one does not use something it will not cost extra. Some services are tiered; this means user might get more value by choosing a more expensive plan. For this thesis I will be working with quite simple version and focusing to use the basic repository, build, deploy and pipeline tools that AWS provides.

3.3 Google Cloud Platform

GCP is Google's DevOps platform and it was made available by the end of 2011. GCP uses the same infrastructure as Google's search engine and YouTube, which makes it a respectable contender in the DevOps competition. GCP has also focused on using artificial intelligence (AI) and machine learning to help with the DevOps process. Like other competitors GCP is flexible and can be used alongside other cloud providers. Different services inside GCP range from computing, storage and databases to big data, AI and security services. [12.]

We decided during project meetings that I will leave Google Cloud Platform out from comparison because we do not use GCP in our work at Valamis Group.

4 DEVOPS PROCESS TESTING

The plan is to find if there are any notable differences between the two cloud services used in Valamis. The tests include the workflow process and time measurement with multiple deployments. Setting up the environments will also be documented. With workflow testing I wanted to know that if the different tools provided by these two services would alter the process between these two services. I am also interested to know if this would cause differences in process time, because at least I think there are differences in processing strength between the

two when using the cheapest subscriptions. With these tests I will also keep an eye out for the UI if there happens to be something notable, so usability is tested.

The plan is to create a branch in the DevOps service’s own repository, to pull the branch to local environment, to make changes, to commit, to push the test app back to DevOps remote repository, to have the app be built in the pipeline, to see it pass the JUnit test and to deploy it to a virtual machine. Another test is similar but involves making the JUnit test fail to see if there are any different reactions or if there are differences in the build time.

For the tests I am going to be using my custom home computer with Windows 10 Home 64-bit, Intel Core i5 3.80GHz and 16Gt of RAM (DDR4). The app I used for the tests is a simple “Hello World” Java app with JUnit test build in. The JUnit test looks for the two integers that have been set inside a function. The test succeeds only if the integer 2 is bigger than integer 1. The repository is made in my personal Git which is then cloned to the DevOps services. I will try to make the pipelines as similar as possible using tools the different platforms provide so that tests would be comparable. I will be using the “Free” or the most basic subscription/version for both cloud services. The Integrated Development Environment, IDE, I will be using is Visual Studio Code.

I chose to use as simple data as a test since I wanted to focus more on the differences in workflows and if I could find any differences in the pipeline process between the two compared services. More complicated test data could possibly make the differences in the process times more noticeable, but unfortunately that would have cost more time and resources, which I did not have. Hopefully when using the free subscriptions for the platforms, even the simple test app can provide some differences in the process times.

During the environment set up I will also keep notes of any notable differences in the user interface if some errors appears. In the UI comparison I will also compare some of the differences in the functionality of the platforms that I had noticed. I strive to give objective comparison between the two UIs, but because I have used Azure DevOps earlier, I will not compare which platforms UI is easier to use.

4.1 Setting up the environments

In this segment I will provide info about how I set up the environments as close as possible. The difficulty of setting up these platforms is not compared because it is more related to personal taste and because I have more experience of using Azure, the comparison would not be fair. For both services I used a GitHub repository that I created just for these projects, because both supports GitHub integration. In this segment I will not provide steps on how to create GitHub repository because the goal is to set up the environment and creating a CI/CD pipeline for the DevOps services.

4.1.1 Setting up the Azure DevOps

First of all, I need to create new account for Azure DevOps which can be created in https://azure.microsoft.com/en-gb/services/devops/. After the account creation, dev.azure.com is available to use and should take the user to the main page of the Azure DevOps. Here I am going to build the CI/CD pipeline. I created new private project called Azure_T and was then greeted by the home page for Azure. From the sidebar I can find all services Azure provides (Figure 3).

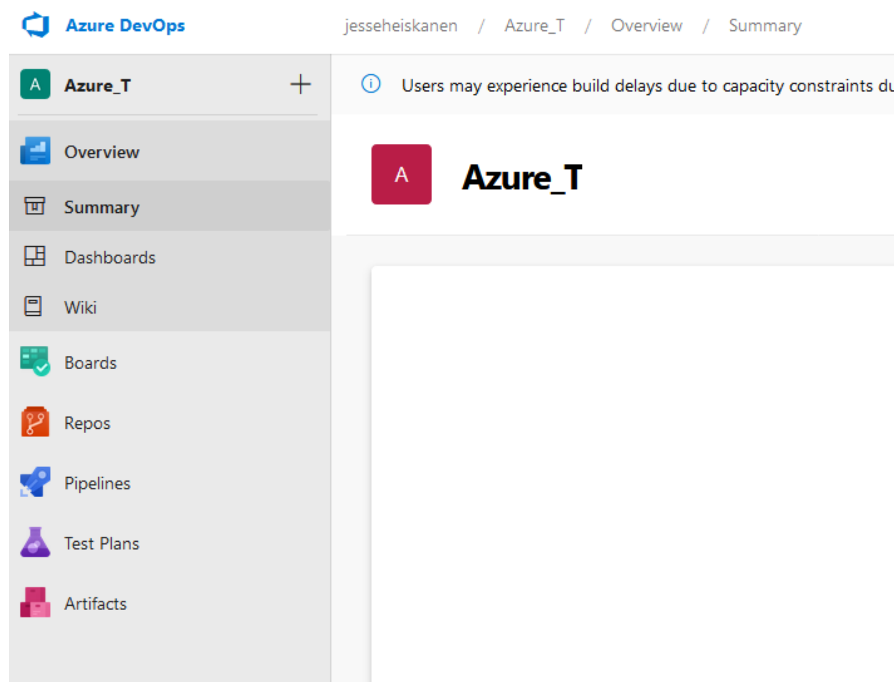


Figure 3. Azure DevOps’ homepage.

Now I will need to import my GitHub repository to Azure. It can be done from the Repos tab in the sidebar. From there click “Import Repository” and select repository type as Git, users will need to paste their Git repository URL to the option. Note that if a repository is set up as private, users will need to select “Requires Authentication” option and they will then be asked for authentication. After options are set, they should select “Import” to start importing process.

Now in the Repos tab userd can browse files, look at commits and pushes, modify and create branches for this repository, create tags for commits and view and create pull requests. Pull requests are used for teams to review and either deny or accept the changes others have made. I will also clone the Azure repository to my computer, so all future commits are sent to Azure. This is done by clicking Repos and from the right-side selecting Clone, (Figure 4). I used the “Clone in VS Code” since that is the IDE I am using during these tests. When clicked, a prompt appears that suggests opening these files in VS Code. From VS Code I saved the project in a new folder.

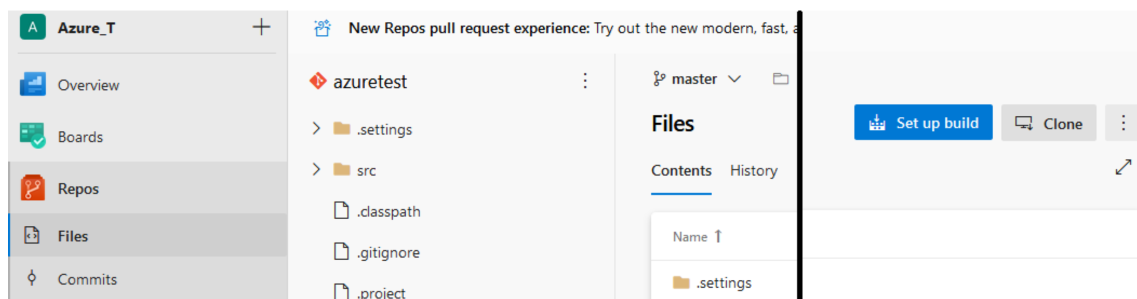


Figure 4. Cloning repo to a local machine in Azure.

Next, I can start creating the barebone version of pipeline that can build our test app. Pipelines are managed from the sidebar by selecting the Pipelines tab. Here I opened the Azure Pipeline wizard by selecting “Create Pipeline”.

This opens up a new prompt that wants to select our source location. Because I am using GitHub as a source, I need to authenticate my access to GitHub. After connecting and selecting our source I am asked to select a template for this pipeline. Our test app is going to use Java components so I will select Maven as a

template for this pipeline. This will create a yaml file that is pushed to our repository and can be viewed/modified if necessary, before saving. Pipeline can be modified at any time by just editing this azure-pipeline.yml file, I used the template without modifications for this test build (Figure 5).

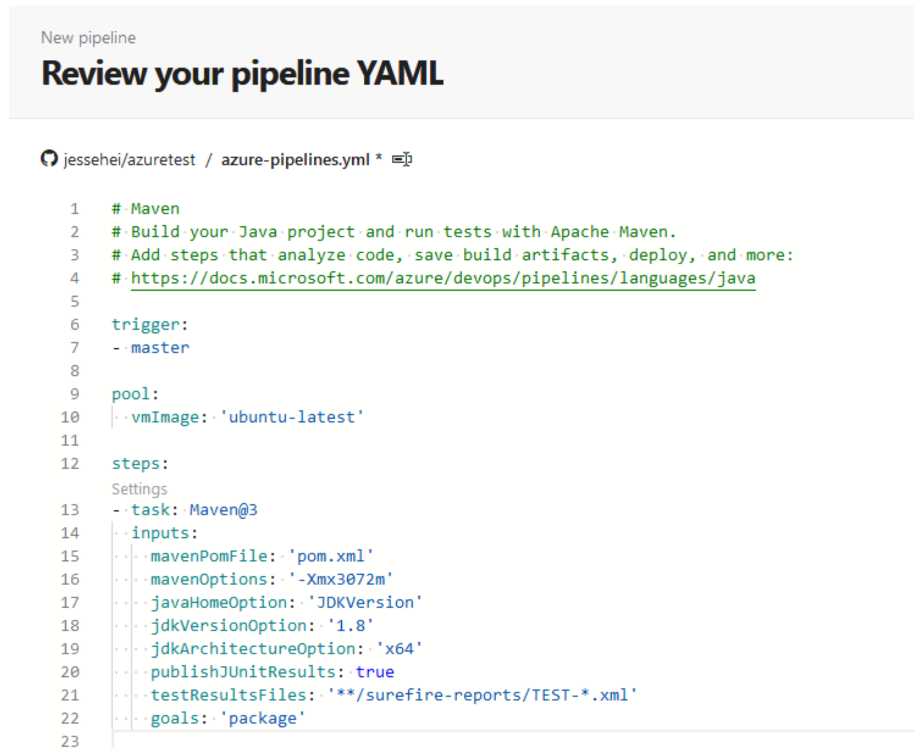


Figure 5. Maven template.

Next, I saved this file. Because this is a whole new file it must be committed and pushed to my repo. This will also automatically activate and run the pipeline I just created, because the pipeline is triggered when there are any changes to a master branch. This pipeline will just build our app using Maven. Everything seems to be working without errors, so I am ready to start creating the deployment environment and then I will be creating a different pipeline that also deploys to the said environment.

The deployment environment is in portal.azure.com. I logged on to the site with the same Microsoft account I used when creating Azure DevOps project. I started by clicking “Create resource” and selecting a web app. I created a new Web App that runs in Tomcat 8.5 and the operating system used was Linux. For subscription I chose to use Free Trial. I also needed to fill in resource group, instance

name (name of the web page) and created new app service plan which controls the computing resources, such as memory for our app. For this also I will use Free F1 plan that has 1GB of memory. Details can be seen in the figure 6.

Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * Free Trial

Resource Group * (New) azuretest

Create new

Instance Details

Name * azure-t

.azurewebsites.net

Publish * Code Docker Container

Runtime stack * Tomcat 8.5

Operating System * Linux Windows

Region * North Europe

Not finding your App Service Plan? Try a different region.

App Service Plan

App Service plan pricing tier determines the location, features, cost and compute resources associated with your app.

Learn more

Linux Plan (North Europe) * (New) ASP-azuretest-bfe7

Create new

Sku and size * **Free F1**

1 GB memory

Change size

Figure 6. Web app details.

Before creation, the web app can be reviewed. After reviewing, everything seemed fine, so I clicked “Create”. After creation more details about the web app can be looked up from the main page. By clicking the web app I noticed that it is currently running, and by clicking on the URL which is the web address composed of the web app instance name and *.azurewebsites.net*, and by checking the site I was greeted by the default message (Figure 7).

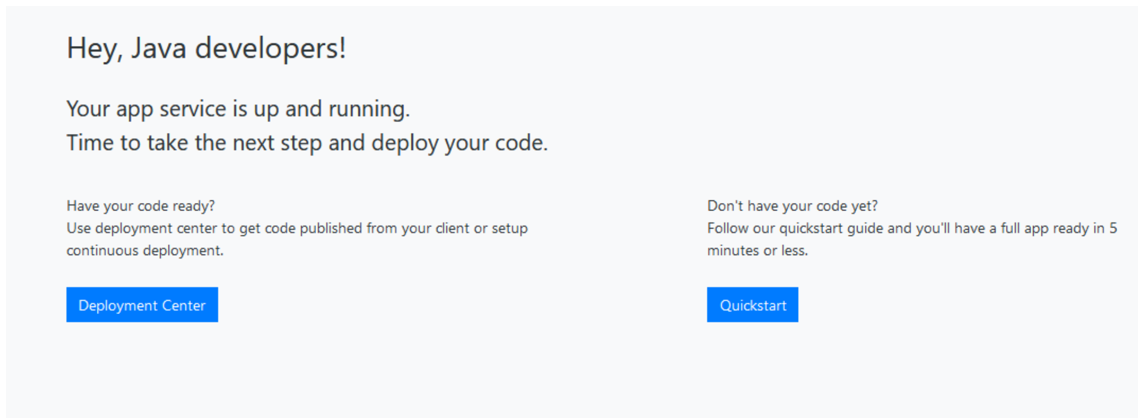
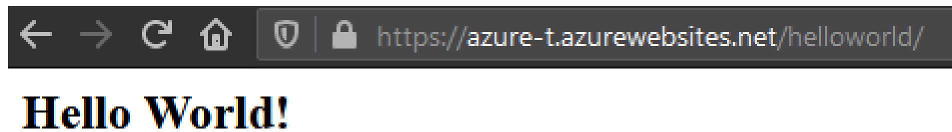


Figure 7. Azure website.

Now that the environment was up, it was time to create a pipeline that deployed the test web app to this site. For that I created new pipeline from Azure DevOps, in the same way I created the first build pipeline. For this pipeline I will be using "Azure Repos Git" as source because I pushed all tests to Azure Repos. For the deployment there is also a pipeline template, "Maven package Java project Web App to Linux on Azure". After selecting this template, a user is asked to select subscription. This was the same subscription which was selected when account was created in Azure Portal. Authentication might also be asked about once more. Next, I selected the web app I just created. Users can once more review the pipeline if there happens to be something wrong. In this case I did not change anything from the template, the YAML file for pipeline is set as Appendix 1. The user can then just click save and run the new pipeline. If the pipeline runs clean, the test app should have been deployed to the new environment. This can also be confirmed by looking at overview of the web app in Azure Portal that there has been some activity.

But by refreshing the azurewebsites.net I was still greeted by same "Hey, Java developer" message. With further inspection I noticed that the default .jsp file in the *wwwroot/webapps/ROOT* hadn't for some reason been overridden. This is where that Azure default page is located, but the deployment had been pushed

to a new folder inside *wwwroot/webapps/helloworld*. And when going to the directory inside the site I was greeted with my own “Hello World!” customization (Figure 8).



Hello World!

Figure 8. Deployment complete.

The complete pipeline now builds our test app that has JUnit test put inside it and deploys it to remote environment. After the build stage is complete, the build is stored as a .war file as an artifact. Next, in the deploy stage this artifact was deployed to the virtual environment hosted in Azure Portal.

4.1.2 Setting up the Amazon Web Services

To start setting up the Amazon Web Services cloud platform, first an account for AWS is needed. Account can be created in *aws.amazon.com*. Because AWS will charge users by the usage amount of their services a credit card is asked for. This will cause a small payment to happen in the account, but that money will be given back after some time. After an account is created user may, now access the AWS management console from the main page as the root user (Figure 9). The user can access everything AWS has to offer from this management console.

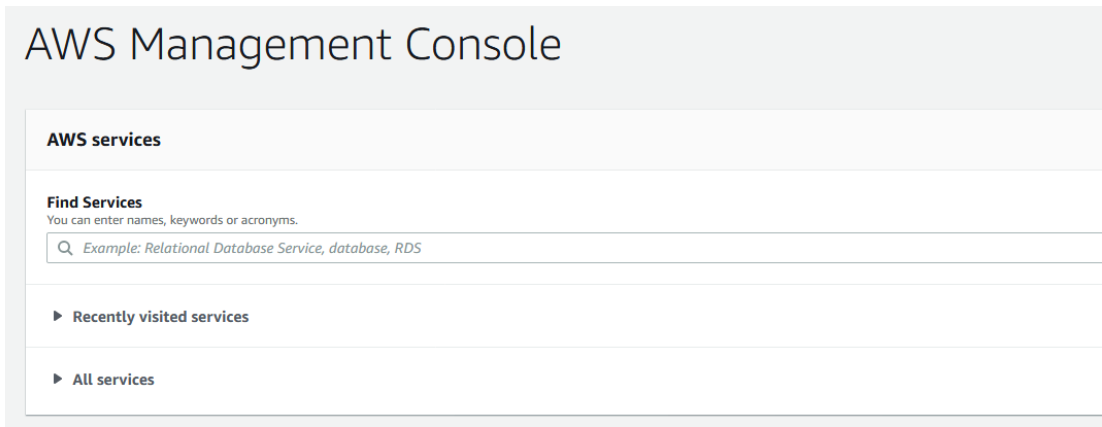


Figure 9. AWS Management Console homepage.

For security reasons it is suggested to not use the root user for these administrative tasks, so I will create an IAM, Identity and Access Management user, that I will use for the rest of this project. Users can find any services AWS provides from the search inside the management console, by typing in IAM and selecting the IAM dashboard opens. Here individual IAM users can be managed and created. I created a new user with all accesses and custom passwords then gave it an AdministratorAccess policy. Before creation, one can view the accesses given (Figure 10).

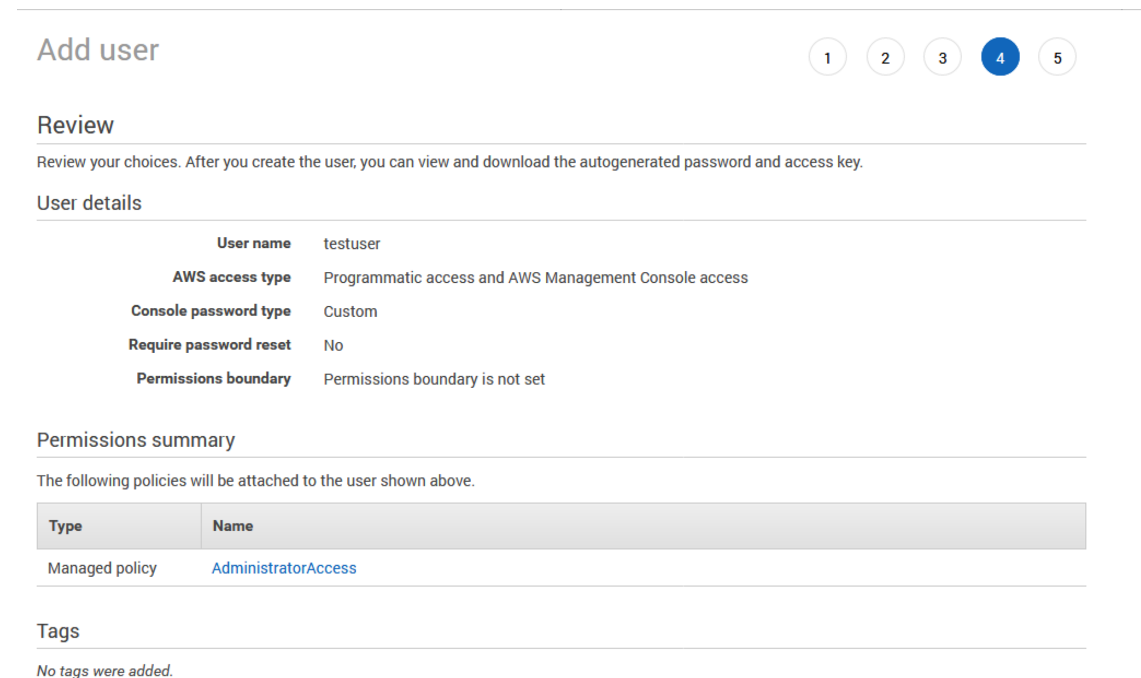


Figure 10. IAM user details.

When user is created it can now be used to sign into AWS platform with the custom login address AWS provides. Before creating a repository inside the AWS, I needed to add Git credentials for our IAM user. This can be done from IAM service by selecting “Users” from the side menu, then selecting the user that was just created. Then from the user menu, one should select “Security Credentials”, and from this menu scroll down to a “HTTPS Git credentials for AWS CodeCommit” topic and clicking on “Generate credentials”. Then, one should save the credentials somewhere for later use.

Next, it was time to get my code to the remote repository inside AWS. For this, I used AWS CodeCommit service. From the management console, one can search for CodeCommit. They were then taken to CodeCommit dashboard, where new repositories are made. They can then provide the name and optional description for the repository. At this point, we had an empty repository inside the CodeCommit, so it was time to add the test app in the repo. To add data, I first copied the repository using the git clone command repository provided (Figure 11) and pasted it in my local command-line. It is important to note that at this point the credentials that were created were needed. I made a mistake when writing credentials, so my access was denied. When I tried copying the repository again, I was instantly denied access without asking any credentials. I found out that Windows stored the wrong credentials inside its own credential manager, so I had to modify those credentials manually from inside the Windows’ settings.

Figure 11. Copying repository to local machine in AWS.

Now that I had cloned the repository to a local environment. I copied all the test app files to this local folder. Then I created a commit and pushed the changes to the master branch. Now I had my test app inside the CodeCommit.

Next, I created a build stage for the test app. To make a build stage I used CodeBuild service. This can also be accessed from the Management Console. From the CodeBuild dashboard, users should select “Create project”. Here they can customise how the build stage works by filling in the name of the build stage, then choose the CodeCommit as a source. The repository should be the one that was just created. As a branch I used the master branch, so build would use the latest version of master branch every time it was run. I chose Ubuntu with standard 2.0 image as the environment that is used when building (Figure 12). I chose to use the buildspec.yml that stores yaml commands for the build stage. When all options seemed good, I clicked on “Create build project” to finish the setup wizard.

Figure 12. AWS Build environment.

Before going any further, it was time to create the buildspec.yml file. Because there were not any templates for this file structure, I used azure-pipelines.yml as a reference for YAML file used by AWS. This file provided information on building the Java app with Maven and then packaging it as .war so that it could be sent to the deployment environment (Figure 13). After creating this file, it had to be pushed to the repository inside the AWS.

```
buildspec.yml
1  version: 0.2
2
3  phases:
4    install:
5      runtime-versions:
6        java: corretto11
7    build:
8      commands:
9      - echo Start build
10     - mvn package
11   post_build:
12     commands:
13       - echo build finished
14 artifacts:
15   files:
16     - target/helloworld*/*
17   discard-paths: yes
```

Figure 13. buildspec.yml

Next, I created the deployment instance. I used Elastic Beanstalk as the environment. Like other services, Elastic Beanstalk can be accessed from the Management Console's search function. From the Elastic Beanstalk, users should click the "Create Application" in the same manner that the CodeBuild stage was created. Here one needs to fill in the basic information, such as the name for the environment. Next, for the platform I used "Tomcat 8.5 with Java 8 running on 64bit Amazon Linux" with recommended version of 3.3.6. The sample application can be used to deploy to the environment, because it will be overridden by the test app later. Configuration for the environment can be seen in Figure 14.

Figure 14. Elastic Beanstalk configurations.

After clicking “Create application” the environment was created and started. The creation process takes few minutes, but after it is done it can be accessed. The user is then taken into a dashboard for the environment where the health status, running version and the platform used can be seen. This dashboard also provides a view for recent events that have happened in the environment. In the top left the URL for the environment can be seen and by clicking it, user is taken to a site where currently the AWS created sample app is located (Figure 15).

Figure 15. Sample app inside new Elastic Beanstalk.

Now that source repository (CodeCommit), build stage (CodeBuild) and the deployment environment (Elastic Beanstalk) had been created it was time to create

a pipeline that uses all these services. Pipeline can be created from the Management Console using CodePipeline. The configuration of the pipeline can be started from the CodePipeline's "Getting Started" tab. For the first step in setting up the pipeline, the user is asked to enter the name for the pipeline. In the second step, the user is asked to fill in the details for the source code, so in this case the provider was CodeCommit, repository was the "aws-test" I created earlier. As a branch I selected to use the master branch. To detect the changes in this branch I used "Amazon CloudWatch Events". This meant that the pipeline may be automatically started with every change to a master branch.

Next step was to add the build stage, and for this stage I filled in the details about the build stage that was created earlier. Final step for the pipeline was to add the deploy stage, so user is asked to fill in the details about the Elastic Beanstalk service that was created earlier. Then the pipeline can be reviewed before creation. After the creation, the whole pipeline process can be viewed (Figure 16).

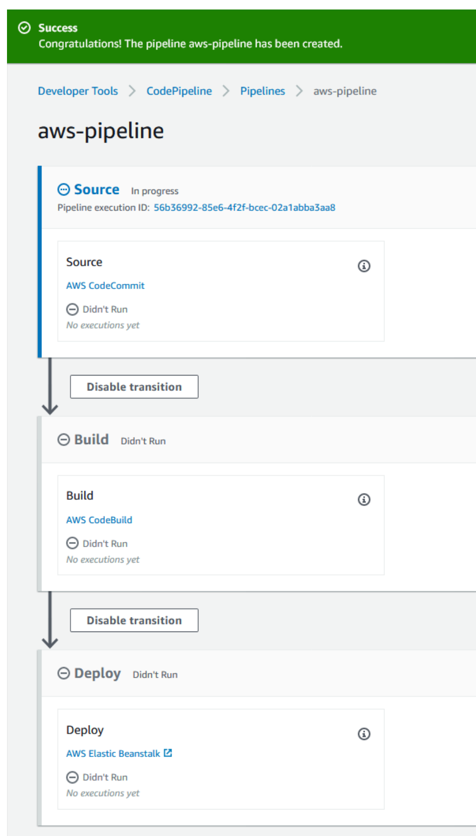


Figure 16. AWS CodePipeline

The pipeline should start running automatically the first time it is created. If everything went well it should run without any errors. After the deployment stage has run successfully, the changes can be noticed inside the Elastic Beanstalk environment, and the sample application from before should be overridden as can be seen from the Figure 17.

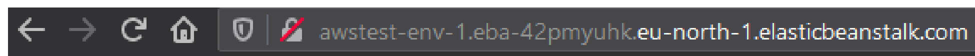


Figure 17. Successful deployment to Elastic Beanstalk.

To recap, the pipeline checks for the changes in the master branch, and when the changes are noticed it will start automatically to build the master branch with new changes. Build stage creates a .war file that is then deployed to Elastic Beanstalk.

4.2 DevOps Workflow Process

Because both environments are now set up, I will introduce how the development workflow proceeds in each environment with the pipeline I have created for both platforms. This type of workflow will be used during the tests. I will also compare the differences in UI I found during the setting up process.

4.2.1 Workflow in Azure

Workflow with Azure is quite simple, but there are few things to be noted. This is simplest workflow with the pipeline I have created. First user creates a branch inside Azure DevOps and then either clones the Azure repository if this is the first time working with project, or, if user already has repository set in the local environment they can just pull the new branch and start working on it. After the changes are made and user is ready to push the changes to repository, a commit is needed to be made that describes the changes. After the commit, the user can

push changes to Azure DevOps. Then from inside the Azure DevOps site, when clicking on the “Repos” tab there is notification that states there are new changes in the branch. It also proposes making a pull request out of it.

Pull requests are used by development team to review the changes to the code. The changes can be approved, rejected and other team members can comment on changes and suggest alternative fixes for the code. Pull requests can be managed from under the “Repos” tab and selecting “Pull requests”. Here, users can create a new pull request for the branch they have been working on. If someone is set as a reviewer for code changes made by others, they will see those requests here. When the right number of reviewers have accepted the changes the pull request can be created. This will merge the changed branch to a master branch.

Next is the continuous pipeline’s turn to start doing its part. This pipeline can be modified to automatically start the deployment process when the master branch is changed. This means that it builds the code and does the tests included in it. There is also an artifact created after the build process. The artifact is then deployed to the virtual environment by the deploy stage. After the pipeline has run its stages, the changes can be seen in the deployment environment.

4.2.2 Workflow in AWS

The workflow in AWS is quite straightforward, as it is in other DevOps platforms. First, new branch is needed to make out of the master branch, this is done in CodeCommit, where the repository is located. In the CodeCommit users can select the repository they are using, and from the side menu the ”Branches” tab provides information about every branch in the repository. Here, new a branch can also be created. When clicking ”Create Branch” a new prompt appears where users can give name to a branch and then choose what existing branch will act as basis for the new branch. When a new branch is created it can be pulled to the local repository, provided the user has copied the repository earlier

to the local machine. Next, the user makes their changes locally and then makes a commit and pushes the changes back to the remote repository.

Then, before the automatic deployment pipeline can be started, the code must be merged to the master branch. From the CodeCommit repository in the side menu there is ”Pull requests” tab. If there are new pull requests made oneself or if the user is set as a reviewer, active pull requests show up here. A new pull request can be created by clicking ”Create pull request”. When clicked, a new window opens where source branch and destination branches can be chosen. Here a new branch will be used as a source and as a destination branch the master branch is set. New pull request window opens, and there is notification made if merge issues are or are not detected. The next step is to give the pull request a title and optional description. Code changes and commits are also visible in this page. After necessary information is given, active pull request is now made. This request can be viewed by any project team members and they can give feedback and approvals before merging to the master branch.

When all seems to be good with the pull request, merging it to a master branch can be done by clicking on the ”Merge” button. Before merging, a user is asked how the merge should be done with three different strategies. Also, source branch can be deleted if wanted. When ”Merge pull request” is clicked the master branch will be updated.

I built the pipeline to automatically start when there are changes made to a master branch. Now when changes are noticed the pipeline starts building the app and runs the JUnit test within. Successful build is packed inside the .war file and sent forward in the pipeline. The deploy stage then sends the .war file further to Elastic Beanstalk service, which is our deployment service, where the pushed changes can be verified.

4.3 User interface comparison

During the creation of the pipelines and when testing the workflow, I noticed three key differences in UI between the two platforms. Azure has a clean interface, and all the necessary tools are accessible from the side menu. Because AWS has a lot more services and components, they all cannot be fitted to the similar side menu. AWS has solved this problem with the search function in its management console. AWS also updates its management console page, giving users quick access to the tools that were recently used. This also helps AWS to avoid crowding in the number of services. There were no weirdly placed elements in the UI for both platforms, so it boils down to user preferences which user thinks had more likeable UI.

Azure seemed to be more beginner friendly with its UI and services. One great example is how different templates were used when creating the pipeline. This meant that users did not need as much prior experience working with pipelines. AWS on the other hand seemed to be aimed more so at professional developers, as it had many different configuration options appearing during the pipeline creation. During the pipeline creation process in AWS, I did not find that it provided any templates in the pipeline set up tool that could be used in the pipeline.

Last notable difference I found was related to the deployment environment that was used to display the app. In Azure this was created in the Azure Portal, which is a separate page from Azure DevOps. I had to create a separate free subscription in there and configure the deployment environment completely inside that portal page and later connect it to the DevOps. Because AWS had huge pool of different services, the creation of this deployment environment could be done from inside the AWS management console. This made some of the necessary tasks easier.

4.4 Process Time Measurement

The process time will be measured from the pipeline's start of the build to the end of the deployment stage where changes can be seen in a test environment. This test will be done for ten times with a fresh build every time to see if there are any notable differences in the average time of the pipeline process. I will create graphs out of the test results to better show the results. The tests are done as follows: first for every new test a new branch is created and pulled to the local environment. In local environment I will modify the paragraph from the *index.jsp* file so that changes can be previewed in the deployment environment. Figure 18 shows an example.

```
src > main > webapp > index.jsp > ...
1  <html>
2      <body>
3          <h2>Hello World!</h2>
4          <p>test #1</p>
5      </body>
6  </html>
7
```

Figure 18. index.jsp.

4.4.1 Process Time in Azure

Azure DevOps has its own timer that is set off when the pipeline starts working with the first stage and is stopped when the last stage has finished (Figure 19). I will be using this data to compare the times between different builds.

Time started and elapsed
Today at 15.19
1m 2s

Figure 19. Example process time from Azure.

For every test I created new branch and modified the *index.jsp* file that is displayed on the web page. This allowed verifying that the changes had been made.

I then pushed changes to Azure, where I would make a pull request for the changes. During testing there was one error that happened with test number 7. This error happened in deploy stage but still it was deployed successfully to the test environment. I could not be sure if this had affected the process time, so I decided to run the test again so that all data would be comparable. After the tests it is noticeable that the first test took the longest time, 87 seconds and the average after 10 tests was 63.6 seconds. The shortest process time was with test number 9, with a duration of only 54 seconds. Visualised data can be viewed for the results in Figure 20.

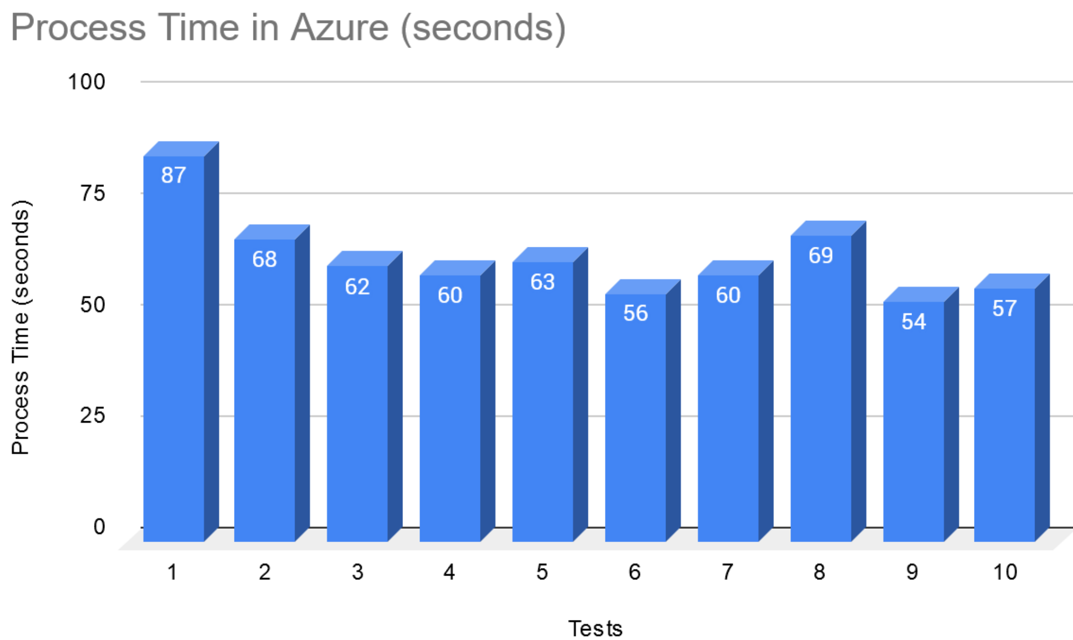
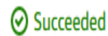


Figure 20. Visualised process time in Azure.

4.4.2 Process Time in AWS

The process time was tested in an identical way compared to the tests in Azure. AWS also provides the duration of the pipeline process and that is the data I will compare with the tests (Figure 21).



Source – 98490278: process-time-1 1 minute 45 seconds

Figure 21. Example process time in AWS

For every test I created a new branch and made local changes *index.jsp*. Then I pushed the changes to the AWS repository and merged the changes with master branch to trigger the pipeline. After the pipeline had deployed the changes successfully, I manually checked the Elastic Beanstalk environment that the changes to the .jsp could be seen. After the ten test builds, I noticed that the pipeline's processing time was consistent with only a few changes here and there. The longest time was 105 seconds; the shortest was 103 seconds, which happened with 5 out of 10 tests. The average was 103.6 seconds, which also shows the consistency between test times. Test results have been visualised in Figure 22.

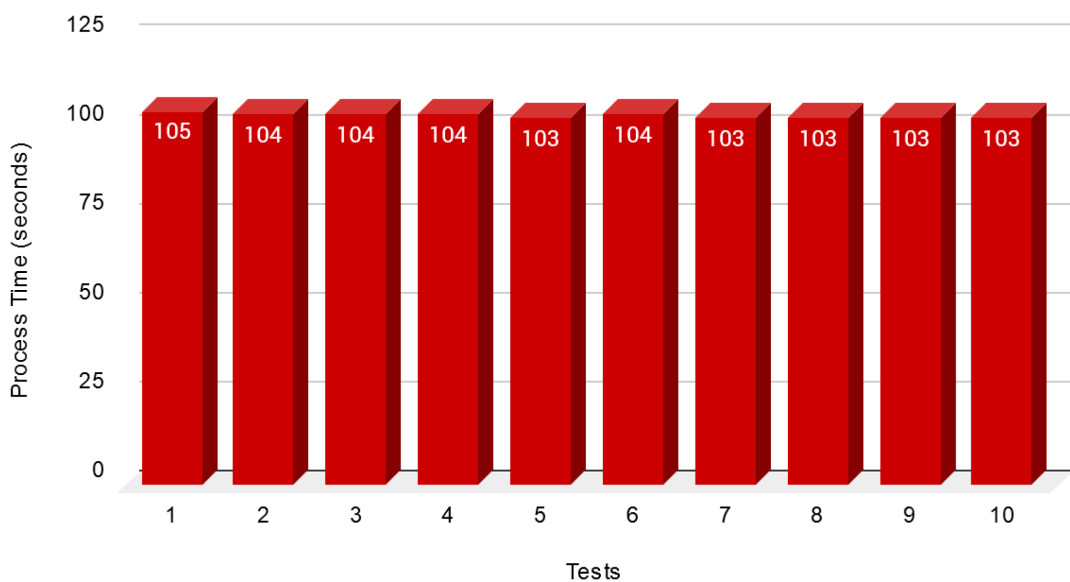


Figure 22. Visualised process time in AWS.

4.5 Process time in failed test

This test will find out if there are any differences in time when the JUnit test fails. This test is also done ten times with fresh build to see if there were differences in

time of the building stages. Also the possible differences in the output message of the error was checked. The results will be documented same way as I did with the last tests.

4.5.1 Bad deploy in Azure

This test was done in same principle as the test with the whole process time, but this time I made the test fail so it would not get past the building stage. Again, I did all ten tests with new branch and new commit every time and would use the pull request for the merge to start the automatic pipeline. Every time the JUnit test failed with the error message saying "*Failed tests: isGreaterTest(junit.JunitMeasureTest): Num1 is greater than Num 2*". Once more, I would compare the times Azure provides in the pipelines. After the tests it seems that the build times move a lot by random. The longest time for build stage was 34 seconds, and the shortest time was only 17 seconds, or half of the longest time. The average time for the tests was 26.7 seconds. Figure 23 visualizes the data.

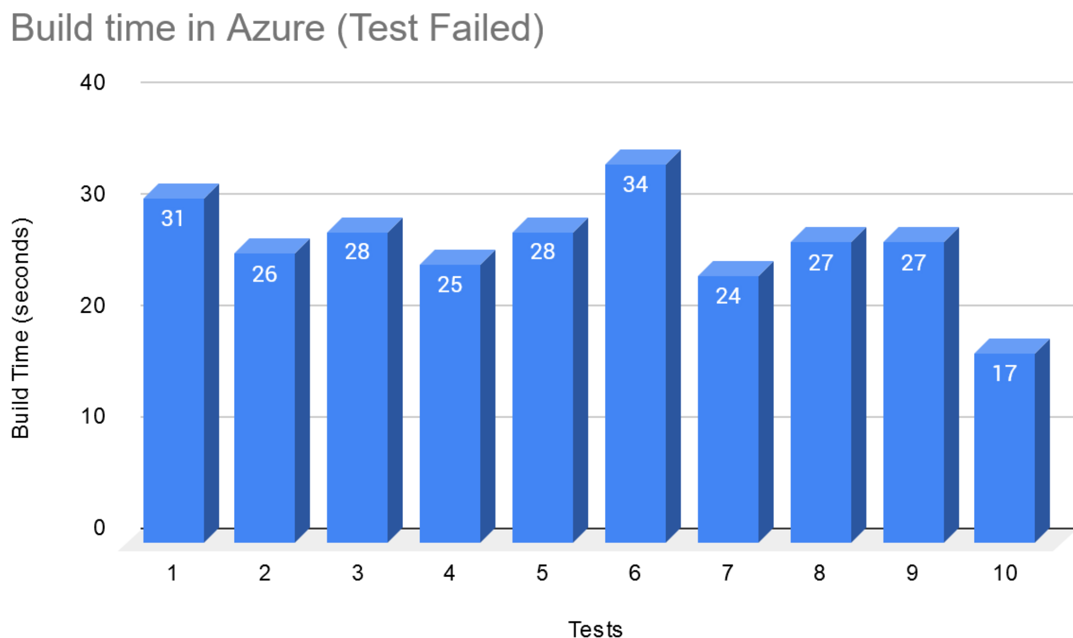


Figure 23. Build time in Azure for failed tests visualised.

4.5.2 Bad deploy in AWS

This test was done same way as the last test runs. The difference this time was to make the built in JUnit test fail and to see if there was anything notable in the error logs or in the build time. For these tests, a new branch was created and merged to master after changes to trigger a fresh pipeline.

Like the last AWS test there, was not much disparity between the build times. Test failed every time with message: ”*Failed tests: isGreaterTest(junit.JUnitMeasureTest): Num 1 is greater than Num 2*”. In Figure 24 the test data is visualised for clarity.

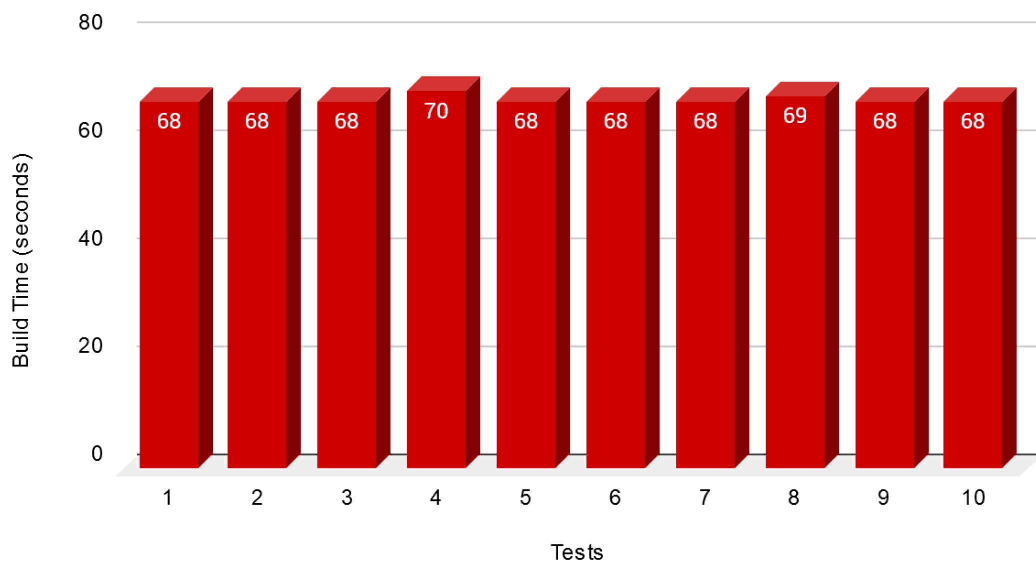


Figure 24. Build time for failed tests in AWS visualised.

5 SUMMARY OF RESULTS

Both cloud platforms were built using similar toolsets, and the pipelines were created to be as similar as possible. The focus was to create the pipelines using the own tools each platform had created, so I did not want to make same pipeline to both services using third party tools.

The development workflow between the two platforms did not vary at all when looking at the entirety of the workflow. There were minor user interface differences, but nothing major in the big picture of the workflow. For both platforms the workflow goes as follows: user creates a branch, pulls branch to local environment, makes some changes to the code, pushes changes back to DevOps repository, creates a pull request and merges it to a master branch. Then the pipeline automatically deploys changes to deployment environment.

After the process time tests, it was easy to notice that in Azure DevOps both process times, in the test with complete deployment pipeline and the test where JUnit test failed, were much faster compared to the Amazon Web Services. Process times can be seen compared in Figures 25 and 26.

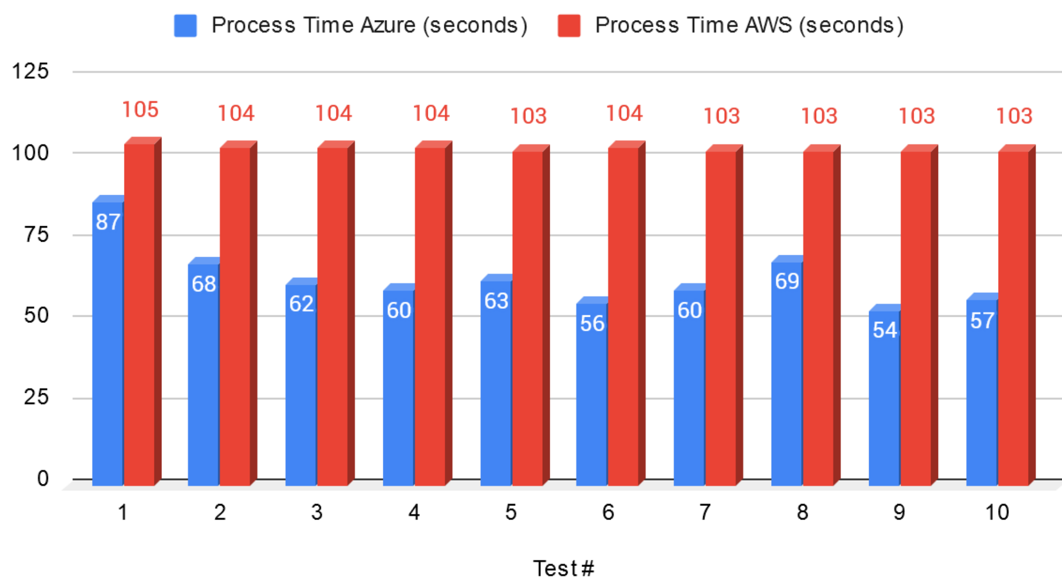


Figure 25. Process time comparison.

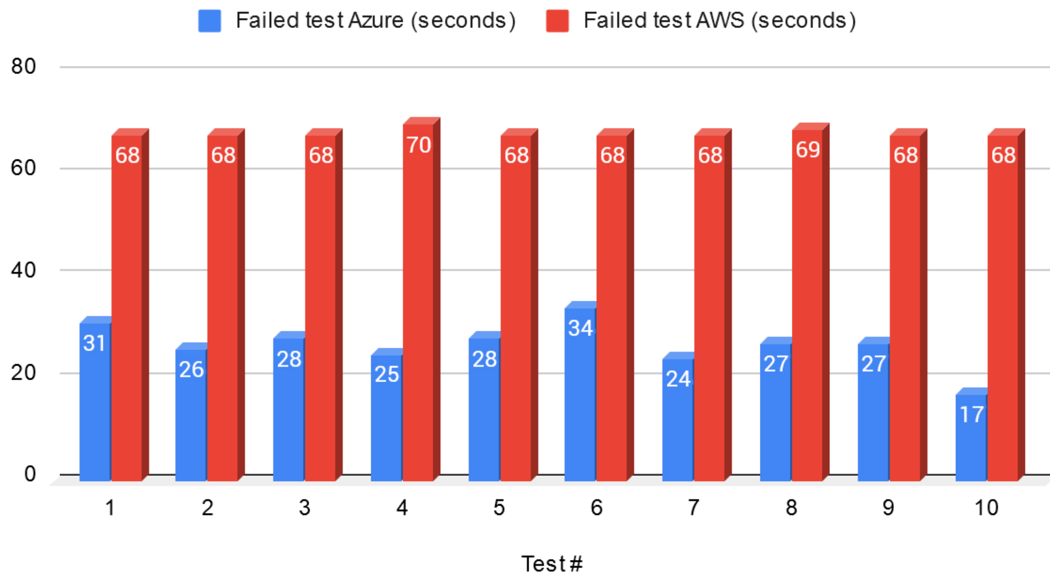


Figure 26. Process time comparison in failed test.

The average process time in Azure was 40 seconds faster in the first test than in AWS. Azure showed even more notable differences in average times with the build time in the failed JUnit test. In failed test, Azure's average was 41.6 seconds faster than AWS. Because the build pipeline was almost identical, the reason for this big difference might be related to the processing power of the AWS. In AWS I had used the free model, which was using 3 GB memory, 2 vCPUs. Unfortunately, I could not find the exact information about what Microsoft-hosted agents in Azure were used in the pipeline during the tests. I was therefore not able to conduct any hardware comparisons. The agent used in Azure environment varies using three different processors: Intel Xeon 8171M 2.1GHz, Intel Xeon E5-2673 v4 2.3 GHz or Intel Xeon E5-2673 v3 2.4 GHz. This might explain the wide variety of process times in the tests with Azure.

Difference between the slowest and fastest time in Azure environment was 33 seconds in the first set of tests and 17 seconds in the second set of tests. These same numbers in AWS were only 2 seconds of difference in both set of tests. Consistency between build times seem to be on AWS' side based on test results.

Both platforms did not have any notable differences in the error message from the failed JUnit tests. When the JUnit tests failed also the pipelines stopped, so nothing broken was deployed to remote environments. On one occasion, deployment in Azure failed once for unknown reason during the first process tests. Even though the pipeline notified that deployment had failed, the changes were still applied in the environment. This made me redo the test for one extra time because it might have affected the pipeline's speed. Otherwise there were no other problems during the process time tests, and I am confident that I gained comparable data for these two platforms.

When the user interface comparison is taken to account, the Azure DevOps' more beginner friendly UI with the usage of ready-made templates is even with AWS' bit more in-depth approach to the interface with vast amount of services at hand. There were no bigger problems found when using either of the interfaces.

The two cloud services are almost identical when looking purely at the workflow. The differences start to arise when checking the process time of the automated pipeline. From the numbers it is easy to see that Azure DevOps was much faster in the pipeline process. On the other hand, the inconsistency of the process times is something to keep in mind. That is where the Amazon Web Services were shining, because there were only few alternations in process time between the tests. But when looking purely at processing speed of the pipeline, which was the focus point of the tests, the average time for the Azure was around 40 seconds faster than the AWS after both tests. This means that Azure DevOps takes the upper hand from this comparison.

6 REFLECTIONS

Before this project I only had experiences in using the Azure DevOps platform. But now that the tests and the comparison is done, I have not only gained experience on the AWS platform, but I understand the DevOps process a bit better

with new view for the whole area. As I see it, the entire DevOps culture is the next step for us developers to be able to produce good products for the customers.

Comparing the two big DevOps platforms was a good experience and I would not mind using either of these services in my daily work. No matter what the results would have been, I still would have to use Azure in my work. It was still insightful to look how DevOps process works in other platforms. It feels like during this project I only scratched the surface of the AWS, but there are also functions that were not used during the tests on the Azure environment.

Unfortunately, with the budget and time I had for this project I had to use the free subscriptions and the pipelines in the environments were simple. Still, I think because of the free subscriptions, the test data varied so much between the platforms. To improve the tests conducted, I would use paid subscriptions and have multiple and more complicated tests so that the pipeline created would have to have more processing power. With these changes the results would have more impact on the company level.

REFERENCES

1. Lucidchart Content Team. 2020 Understanding the DevOps process flow. https://www.lucidchart.com/blog/devops-process-flow [Used 3rd March 2020]
2. Swartout, P. 2012. Continuous Delivery and DevOps: A Quickstart guide. Packt Publishing.
3. Belagetti, P. 2018. 8 reasons why DevOps gets more exciting in 2019. JAXenter. https://jaxenter.com/top-8-devops-predictions-2019-152914.html [Used 5th March 2020]
4. Humble, J., Willis J., Kim G., Debois P. 2016. The DevOps Handbook. IT Revolution Press. https://learning.oreilly.com/library/view/the-devops-handbook/9781457191381/DOHB-pt_04_text.xhtml
5. Vadapalli, S. 2018. DevOps: Continuous Delivery, Integration, and Deployment with DevOps. Packt Publishing. https://learning.oreilly.com/library/view/devops-continuous-delivery/9781789132991/ch02s02.html
6. Son, B. 08.04.2019. A beginner's guide to building DevOps pipelines with open source tools. https://opensource.com/article/19/4/devops-pipeline [Used 14th April 2020]
7. Microsoft Corporation. 2020. https://azure.microsoft.com/en-us/overview/what-is-devops [Used 3rd May 2020]
8. Intellipaat. 2019. https://intellipaat.com/blog/aws-vs-azure-vs-google-cloud/ [Used February 16th 2020]
9. Microsoft Corporation. 19.05.2020. https://docs.microsoft.com/en-us/azure/devops/user-guide/services?view=azure-devops [Used 28th May 2020]
10. Amazon Web Services. 2020. https://aws.amazon.com/ [Used 28th May 2020]
11. Amazon Web Services. 2020. DevOps Pipeline Example. https://docs.aws.amazon.com/codepipeline/latest/userguide/concepts-devops-example.html [Used 28th May 2020]
12. Google. 2020. https://cloud.google.com/why-google-cloud [Used 28th May 2020]

azure-pipeline.yaml

```yaml
# Maven package Java project Web App to Linux on Azure
# Build your Java project and deploy it to Azure as a Linux web app
# Add steps that analyze code, save build artifacts, deploy, and more:
# https://docs.microsoft.com/azure/devops/pipelines/languages/java

trigger:
- master

variables:

  # Azure Resource Manager connection created during pipeline creation
  azureSubscription: 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx'

  # Web app name
  webAppName: 'azure-t'

  # Environment name
  environmentName: 'azure-t'

  # Agent VM image name
  vmImageName: 'ubuntu-latest'

stages:
- stage: Build
  displayName: Build stage
  jobs:
  - job: MavenPackageAndPublishArtifacts
    displayName: Maven Package and Publish Artifacts
    pool:
      vmImage: $(vmImageName)

    steps:
    - task: Maven@3
      displayName: 'Maven Package'
      inputs:
        mavenPomFile: 'pom.xml'

    - task: CopyFiles@2
      displayName: 'Copy Files to artifact staging directory'
      inputs:
        SourceFolder: '$(System.DefaultWorkingDirectory)'
        Contents: '**/target/*.?(war|jar)'
        TargetFolder: $(Build.ArtifactStagingDirectory)

    - publish: $(Build.ArtifactStagingDirectory)
      artifact: drop
```

azure-pipeline.yaml

```yaml
- stage: Deploy
  displayName: Deploy stage
  dependsOn: Build
  condition: succeeded()
  jobs:
  - deployment: DeployLinuxWebApp
    displayName: Deploy Linux Web App
    environment: $(environmentName)
    pool:
      vmImage: $(vmImageName)
    strategy:
      runOnce:
        deploy:
          steps:
          - task: AzureWebApp@1
            displayName: 'Azure Web App Deploy: azure-t'
            inputs:
              azureSubscription: $(azureSubscription)
              appType: webAppLinux
              appName: $(webAppName)
              package: '$(Pipeline.Workspace)/drop/**/target/*.war'
```