CI/CD JENKINS PIPELINE

A Project

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Computer Science

By

Long Lin

2020
This project is a design and implementation of continuous integration, continuous delivery, and continuous deployment (CI/CD) for a modern Dev-Ops environment. The pipeline bridges the gap between development and deployment teams by automating the building, testing, and deployment of applications. The entirety of the pipeline includes shared repositories on Github with different branches for each environment, namely, feature branch for adding new features, develop branch for pre-production environment testing, and master branch for live consumer used production environment, and Amazon EC2 instances for running the pipelines automatically upon changes from repositories and automatically updated servers. The Github repositories architecture is designed and implemented based on the Git Flow architecture with some variations. The CI/CD Jenkins pipelines gets triggered from designated Github repositories for different branches, pull requests, merge, and release. With the CI/CD Jenkins pipelines, developers can push their changes or add new features to an application without worrying about updating current versions of pre-production and production servers with chosen tested features for applications.
<table>
<thead>
<tr>
<th>TABLE OF CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGNATURE PAGE ................................................................. II</td>
</tr>
<tr>
<td>ABSTRACT ........................................................................................ II</td>
</tr>
<tr>
<td>TABLE OF CONTENTS ........................................................................ III</td>
</tr>
<tr>
<td>LIST OF FIGURES ............................................................................ V</td>
</tr>
<tr>
<td>CHAPTER 1: CI/CD ............................................................................. 1</td>
</tr>
<tr>
<td>Modern DevOps operations .............................................................. 1</td>
</tr>
<tr>
<td>Continuous Integration and Continuous Deployment .................. 1</td>
</tr>
<tr>
<td>CHAPTER 2: INTEGRATING CI/CD PIPELINES TO APPLICATION ........... 3</td>
</tr>
<tr>
<td>Application overview ........................................................................ 3</td>
</tr>
<tr>
<td>Application structure and workflow ........................................ 4</td>
</tr>
<tr>
<td>CHAPTER 3: CI/CD PIPELINE DESIGN .............................................. 5</td>
</tr>
<tr>
<td>Choosing tools ................................................................................. 5</td>
</tr>
<tr>
<td>Designing structure ........................................................................ 6</td>
</tr>
<tr>
<td>Designing Github workflow .......................................................... 7</td>
</tr>
<tr>
<td>Designing Servers .......................................................................... 9</td>
</tr>
<tr>
<td>Designing Jenkins pipeline .......................................................... 10</td>
</tr>
<tr>
<td>CHAPTER 4: IMPLEMENTING CI/CD PIPELINES ................................ 11</td>
</tr>
<tr>
<td>Implementing Github repositories ................................................. 11</td>
</tr>
<tr>
<td>Implementing Amazon EC2 servers .......................................... 11</td>
</tr>
<tr>
<td>Implementing Jenkins pipelines .................................................. 12</td>
</tr>
<tr>
<td>CHAPTER 5: RESULT ........................................................................ 16</td>
</tr>
<tr>
<td>Github result .................................................................................. 16</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1 Bitsplanet.io .......................................................................................................... 3
Figure 2 Jenkins Structure .................................................................................................. 6
Figure 3 Github Workflow ................................................................................................. 8
Figure 4 Jenkins Pipeline Architecture ............................................................................. 14
Figure 5 Github Organizations ......................................................................................... 16
Figure 6 User Web Application Organization ................................................................. 16
Figure 7 Administrator Application Organization ............................................................ 17
Figure 8 User Frontend Application Repository ............................................................ 17
Figure 9 Github Webhooks ............................................................................................... 18
Figure 10 Amazon EC2 Instances .................................................................................. 18
Figure 11 Production Instance Control Panel ................................................................. 19
Figure 12 Amazon EC2 Production Instance Security Group ......................................... 19
Figure 13 Amazon EC2 Jenkins Instance Security Group .............................................. 20
Figure 14 Jenkins Pipelines .............................................................................................. 21
Figure 15 Jenkins User Frontend Pipeline Overview ...................................................... 21
Figure 16 Jenkins User Frontend Develop Branch Pipeline Steps .................................. 22
Chapter 1: CI/CD

Continuous integration (Wikipedia, Continuous integration, n.d.) and continuous deployment (Wikipedia, Continuous deployment, n.d.), short for CI/CD, is a combined practice in software engineering. CI/CD forms the backbone of modern day DevOps operations. This chapter talks about how modern day development operations are impacted by using the combined practices of CI/CD pipelines.

**Modern DevOps operations**

DevOps is the combination of using practices and tools that increases a team’s or organization’s ability to deliver applications and services at high velocity. Modern day DevOps practices include continuous development, continuous testing, continuous integration, continuous deployment and continuous monitoring of software applications throughout its development life cycles.

**Continuous Integration and Continuous Deployment**

Continuous integration (CI) and continuous deployment (CD) embody a culture, set of operating principles, and collection of practices that enable application development teams to deliver code changes more frequently and reliably. The implementation is also known as the CI/CD pipeline. Continuous integration and continuous deployment (CI/CD) is a combined practice in software engineering. By enforcing automation in the building, testing and deployment of applications, CI/CD bridges the gaps between development, operation activities, i.e. adding features to an existing software application, and teams. The CI/CD pipeline forms the backbone of modern day DevOps operations. Having CI/CD pipelines could automate the operations needed as mentioned previously.
for adding, changing, and deleting features of a software applications for different types of environments, including developing, testing, and deployment settings.

The technical goal of CI is to form a consistent and automated way to build, package, and test applications. With such consistency in the integration process in a software application development cycle, teams are more likely to commit code changes more frequently, which can lead to better collaboration and software quality.

Continuous deployment picks up where continuous integration ends, it automates the deployment of applications to selected infrastructure environments. Most teams work with different environments other than the production, such as development and testing environments, and the CD pipeline ensures there is an automated way to push good quality code changes to production environment.

The CI/CD pipelines help store parameters needed for different environments that must be packaged with each deployment. The pipelines performs any service calls to web servers, databases, and other services needed to be restarted or some other procedures defined when applications are deployed.

Implementing CI/CD pipelines is one of the DevOps best practice because it addresses the misalignment between developers who want to push changes frequently, with operations that want stable applications. With automation in place, developers can push changes more frequently with quality and stable new or edition of existing features of applications. Overall with CI/CD pipelines in place, the operation teams could see greater stability because of the standard configurations set in pipelines, there is continuous testing in the integration and deployment process, environment variables are separated and kept secure, and rollback operations are done automatically.
Chapter 2: Integrating CI/CD pipelines to application

Every application would have different CI/CD pipeline needs than others. Let’s consider implementing CI/CD pipelines into an existing application and its development workflow. This chapter shows an overview of an existing application and its development architecture.

Application overview

The application that I work on is called Bitsplanet (Bitsplanet, n.d.), it is an application for researching learning resources.

![Bitsplanet.io](image)

The image above shows the front page of the web application. The application requires CI/CD pipelines is a web application consist of servers that runs frontend, backend, and database. The team has three people with four different sections, frontend application for customer usage, backend application for customer frontend, fronted application for administrator usage, and backend application for administrator frontend. First person has the responsibility for customer usage frontend, second person has the...
responsibility for backend for customer frontend and administrator usage frontend, the last person has the responsibility for backend for administrator frontend.

**Application structure and workflow**

The team uses individual Github (Tom Preston-Werner, 2008) repositories for each section mentioned above. The team also setup Amazon EC2 instances (Amazon.com, 2006) to run different types of services and a server on Atlas to run the database. The members commit code changes on local machines then push the desired changes to the Github repositories. Then use the newly edited repository to create a Docker (Hykes, 2013) image and upload it to Dockerhub. Afterwards connect to different servers to pull the new Docker image, stop the old services, and run the updated Docker image on servers to push new or edited features to the existing application. Overall, the entire process for each service has to be done manually.
Chapter 3: CI/CD pipeline design

The design of any CI/CD pipeline is crucial, the needs of the team and the application differs from others. Most CI/CD pipelines designs are different due to the requirements. This chapter shows the process of the designing phase based on the knowledge of the application’s workflow architecture and the team members’ requests.

Choosing tools

Selecting the right tools is a crucial step for designing CI/CD pipelines. The tools chosen also need to satisfy the team’s requirements and automating process. First the team needs to keep using the existing servers for the services which is Amazon EC2 instances, Atlas server for the MongoDB (MongoDB Inc., 2009) service, and Dockerhub service because they have already paid for it. Secondly, the team needs the CI/CD pipeline to add minimal or no cost. With the team’s requirements, I also have to keep in note that from the point of committing and pushing code changes to Github repositories, testing and building the changes, and updating the services to the newest one on servers all needs to be automated with the CI/CD pipelines. Based on the requirements, I have decided to keep using the Github repositories and use Jenkins (CloudBees, 2011) to host CI/CD pipelines because they are free to use with massive amount of features needed. Also, by using Jenkins, the pipelines can be triggered each time when changes are committed and pushed to Github repositories, running pipeline process would be automated. The Jenkins tool also provides the team a user interface, and a way to store secret credentials, certificated and parameters needed to access Amazon EC2 instances without exposing them to others.
Designing structure

![Diagram of Jenkins Structure]

**Figure 2 Jenkins Structure**

The overall structure of the CI/CD pipeline can be described as shown above, it consists of two main sections, Github and Jenkins section. The Dockerhub and Amazon EC2 server sections are part of the Jenkins pipeline. In more details, the CI/CD pipeline consists of a team member committing changes to a Github repository, then the Jenkins pipeline which would pack and upload a new Docker image to Dockerhub, and access the corresponding Amazon EC2 server to start new service. From this process the only manual section is the committing and pushing code changes to Github repositories.

In the Github section, a team member commits changes to Github manually, then the repository would send out a request to corresponding Jenkins pipeline automatically.

Then in the Jenkins pipeline section if no error found in any steps, it would pull the entire Github repository to build and test the changes, then it would pack the repository into a Docker image based on the environments needed specified in a Dockerfile inside of the repository and upload it to the Dockerhub account, afterwards it
would access the Amazon EC2 instance to stop the old service, pull and run the new
Docker image as the new service on the server.

**Designing Github workflow**

For frontend and backend of the user and admin application, the team also
expressed the needs for testing and production environments for both the development
team and the customers. By adding these environments, it would be easier to add, test,
and update the services based on the needs of each team member and the goal of the
team. Based on the workflow before having the CI/CD pipelines, the team uses Github
repositories as mentioned above. However each repository only had one branch, master
branch. This would make the team to separate the development phases, such as
development and production, extremely difficult, so the Github workflow needed to be
changed to adapt the automating DevOps process.
The image above shows the general design for the Github workflow. The feature, develop, and master are branches in a Github repository. A team member could use the feature branch to add and edit features when working on a local machine. Then after however many features added, the team member could decide to merge the feature branch into the develop branch for testing the features’ functionalities and overall quality. After confirming the develop branch has all the features added and edited, the team member could merge the develop branch into the master branch for a new or updated production version of the software service. By using this Github workflow, the environments for adding new features, testing a set of added and edited features, and production level
service are separated into individual nodes, it is much easier for the team members to collaborate, track, and update the services based on the needs of the team. Also, by using this work flow, it is extremely easy to rollback production versions if there is anything incorrect with it.

**Designing Servers**

From what has been designed, the local environments are based on local machines to modify features, therefore it does not need to be hosted on a remote server. The testing environment and production environments need to be hosted servers to an in depth integration testing purposes and live customer usage.

There are several ways to design the servers, for example, there can be four servers running at all times for each service, or two servers, one for customer usage, and the other for administrator usage. However, the requirements of the team need to be considered, minimal cost with highest quality, therefore I have decided to host two different servers on Amazon EC2 instances for the services and one server for running the Jenkins pipeline which is discussed in the next section. One server being used for development environment. Another being used for customer usage production environment.

By designing it this way, the two servers will not be affected by one another. If development server fails, production server would not be affected. Same way around, if the production server fails, the team can use the development server to debug the faults of current production server or rollback the production server to a previous version to keep alive. Also, instead of having a server for each service which there would be four servers,
having two slightly larger server would decrease the cost dramatically and would be able to handle more data transmitted.

**Designing Jenkins pipeline**

For designing the Jenkins pipeline, it is more straightforward compared the others since the tool is free to use. The Jenkins pipeline is also hosted on an Amazon EC2 server as a service. Based on the design of Github workflow, there will be three branches/environments for each repository. There also can be a couple of ways to design this, for example have a pipeline for each repository branch. This way could result in twelve pipelines that have similar functions. The more pipelines would result in more storage used, so it would result in using a larger server which could increase the cost. Also, it would be difficult for the team members to find the correct pipeline and check on the status or error outputs. So I decided to have four different pipelines, one for each service. In each pipeline, it would be able to detect Github branches that are being changed, and based on different branch to perform different procedures.

By designing it this way, each pipeline would be able to detect which Github branches that were being changed, and based on different branch to perform different procedures for simply adding more features, updating develop environment, or updating customer production environment. It is much easier for the team members to debug and see if one branch goes wrong, and to choose which branch needs to rollback. Overall, the team members’ productivity would increase dramatically.
Chapter 4: Implementing CI/CD pipelines

With the design phase finished, implementing the design could be troublesome due to the software using may not suit the needs of design. This chapter goes through each design step by step with problems encountered and how it is resolved.

Implementing Github repositories

From the design I talked about previously in Chapter 3, I made changes to all Github repositories accordingly. After modification, each repository has three different branches. I moved all non-production additional features to the feature branches for local environment, the features that soon will be in customer use production environment to develop branch for in depth integration testing, and only customer used features to master branch for production environment. The end result being team members would be using feature branch when working on local environment to add or edit features, create pull requests to merge from feature to develop branch for almost ready customer usage features, and merge from develop to master branch to update the current customer usage service. Since the modifications did not encounter any issues, the end result are exactly the same as predicted in the design phase.

Implementing Amazon EC2 servers

From the design for Amazon EC2 servers, I migrated all original servers into total of three new servers under a new account instead of using the original account, because new accounts are given a free usage for a period of time, this would cut the cost for the team by a short period. During the migrating phase, I had to install some dependencies and libraries on the servers to make sure the services would run properly, including Docker, Java 8 (Corporation, 2014), Python 3.6, and Pip wheel. Then I moved all original
services under one server for customer usage services, and used the new Github repository develop branches to run the testing usage services.

**Implementing Jenkins pipelines**

Due to the design of Github workflow and repositories, I implemented the third Amazon EC2 server designated to run Jenkins pipeline only. The reason behind it is because each time the pipeline runs it would require at least one point three gigabyte of storage to pack a service into a Docker image, therefore if multiple members of the team are making code changes in all branches, it would require at least thirteen gigabyte of storage available just for packing images and enough processing power to run each pipeline. During creating the Jenkins tool on the new server, I had to install some additional packages to make sure Jenkins runs properly, including Java 8, and nginx (Robertson, 2004) for running web servers. In the end, I chose the Amazon EC2 t2 small for this server that has one virtual CPU, two gigabytes of random access memory (RAM), and with the cost of two point three cents per hour from running the server.

Since Jenkins is an extremely popular tool for building pipelines, there are tons of plugins available to suit the teams’ needs and the work flow design in place. Other than the default packages in the tool, I installed the Amazon EC2 plugin for ease of access of existing EC2 servers, the Blue Ocean plugin for better visualization for the team, the CloudBees Docker Build and the Publish plugin for building and testing using the Dockerfiles in Github repositories, the Docker plugin to upload and download images with Docker in the pipelines, the File Operations plugin for storing and using the Amazon EC2 key pair files to interact with the servers, the Matrix Authorization Strategy plugin for better security settings for the team members, pipeline plugin to access all branches in
a Github repository, the SSH pipeline steps plugin to use SSH commands inside of the Amazon EC2 servers to cleanup, stop, and run corresponding services, and a workspace cleanup plugin for better memory management for the Jenkins environment. There were a lot of trial and error in the choosing additional plugin in Jenkins tool phase, many plugins only suit part of the teams’ needs and the work flow design, but in the end there were plugins available to suit the needs of using SSH commands by accessing Amazon EC2 remote servers with detection of multiple branches in Github.

After installing the Jenkins tool and the plugins, I only had the actual pipelines to implement for it to run different tasks from each code change in the Github repositories. There are two styles that I could write the Jenkinsfile for running a pipeline, one is a declarative pipeline, and the other is a scripted pipeline. Declarative pipeline is rather newer compared to the other, and has more functionalities and dependencies available. I tried to use the declarative approach to write the Jenkinsfiles for each repository, however, some installed plugins would only work in a scripted pipeline to make the pipelines to do the desired tasks, therefore a lot more plugins were installed due to this, and that also explains the reason that I had a lot of plugins installed in the Jenkins tool.
The Jenkinsfile logic is shown in image above, a corresponding pipeline would run when there are code changes in any branches from the repositories, there are total of eight steps in each pipeline. Four steps if code changes were on feature branch, eight steps if code changes were on develop or master branch of any repository. The pipeline
starts out with checking out the Github repository that made code changes. Then clean up old Docker images for memory management in case of if any were not cleaned up properly previously. Afterwards build and test using the current version of Dockerfile from the checked out repository, if the code changes are on feature branches, the pipeline would end, if the code changes are on develop or master branches it will move on to the next steps. Then the pipeline would pack the newly built container from previous step into a Docker image and upload it to Dockerhub, for example a frontend for customer usage Docker image would be upload as user_frontend_develop or user_frontend_master. Then the pipeline would connect to an Amazon EC2 remote server, for example the pipeline would access the testing Amazon EC2 server if there are code changes on the develop branch. After connecting to the corresponding remote server, it would stop the old service, remove the old service Docker container and image, download the new Docker image from Dockerhub, run a new container based on the image as the new service, cleanup all unused Docker images and containers for memory management, and end. Each step in the Jenkins pipeline would be run if there weren’t any error encountered in previous steps, which means if any steps fails, the pipeline would output the errors and come to an end. And the member of the team could check the error on Jenkins tool to debug any issues with the code changes. In the end, the implementation of the Jenkins tool and pipeline has been done exactly like the design.
Chapter 5: Result

With every phase finished, from understanding and come up requirements, design CI/CD pipeline architecture, to implementing it. This chapter will go through the results of the entire application’s development operation architecture and how the pipeline has impacted the team’s productivities.

Github result

<table>
<thead>
<tr>
<th>8 teams in the bitsplanet organization</th>
<th>Visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>admin_app</td>
<td>2 members</td>
</tr>
<tr>
<td>dev-ops</td>
<td>1 member</td>
</tr>
<tr>
<td>super-members</td>
<td>3 members</td>
</tr>
<tr>
<td>user-app</td>
<td>3 members</td>
</tr>
</tbody>
</table>

Figure 5 Github Organizations

As the image shown above, currently, after the design and implementation phases, the Github account consist of two main organizations, one for web application for administrator usage, the other for web application for customer usage.

Figure 6 User Web Application Organization

As the image shown above, the user web application organization consists of two repositories, one for web application frontend service, the other for web application backend service.
As the image shown above, the administrator web application organization also consists of two repositories, one for frontend service and the other for backend service.

The image shown above shows the repository and branches in the frontend for customer usage. In all four repositories shown earlier, each of them has three branches, a master branch for production environment, a develop branch for testing environment, and
a feature branch for local environment.

![Figure 9 Github Webhooks](image)

The image shown above shows the web-hooks in each repository, the web-hooks are for sending out requests to designated Jenkins pipeline each time there are code changes, so the Jenkins pipeline would know which repository is making code changes.

The workflow has increased the productivity of the team members, they do not need to worry about breaking existing production level service from adding more features into other branches. And it also gives the team more flexibility of choosing which code changes or features they would like to include in the next version of the production environment web application by using the testing environment.

**Amazon EC2 result**

![Figure 10 Amazon EC2 Instances](image)

The image above shows the three servers that are running on Amazon EC2. The Prod server is the production environment server, PPD, short for pre-production, is the
testing environment server, and the Jenkins is the server that runs the Jenkins tool and pipelines.

![Production Instance Control Panel](image1)

*Figure 11 Production Instance Control Panel*

The image above shows the details of the production environments server, it includes the IP addresses, the security settings, network monitoring, storage status and some others for the team members to use based on the needs.

![Amazon EC2 Production Instance Security Group](image2)

*Figure 12 Amazon EC2 Production Instance Security Group*

The image above shows the security settings for the production environment server. In the security settings for the servers, the team could set available ports needed to access the web application frontend and backend. During setting up the servers, I chose 22 to be the port to connect to the server, port 3000 for user application frontend service,
port 3001 for user application backend service, port 5000 for administrator application frontend service, and port 5001 for administrator application backend service. The source to be 0.0.0.0/0, which means that any IP address can access these ports. Both production and testing environment servers have the same security settings at the moment. The team could limit the source IP addresses to custom ones for tighter security, however the team currently do not have the needs for it.

![Amazon EC2 Jenkins Instance Security Group](image)

*Figure 13 Amazon EC2 Jenkins Instance Security Group*

The image above shows the Jenkins server security settings, the server only has two ports open, port 22 for accessing the server and port 8080 for accessing the Jenkins tool.

Overall, the Amazon EC2 servers have been running for the past few months for the team’s usage and have not encountered any problems.
Jenkins pipelines result

The image above shows the overview of the Jenkins pipeline. There are four projects in total, admin backend, admin frontend, user backend, and user frontend. They are all set up to listen requests from the Github web hooks, so when changes are made in the Github repositories, corresponding pipeline would be triggered.

The image above shows the details of the user frontend project. It consists of three branches, like the Github repository, when certain branch were updated by code changes, the pipeline would run under corresponding branch.
Figure 16 Jenkins User Frontend Develop Branch Pipeline Steps

The image above shows the steps of each pipeline being run under the develop environment. As mentioned in the implementation section, it consists of eight steps, the picture and the tool does not show the first step, the checking out repository step. Green blocks mean passing the step without errors, red blocks mean the step had error outputs. As shown, if any step fails, the pipeline would exist. The team members could go to this page and click on the step that failed and the tool would show the error message of that specific step. It easy to track which step failed with what kind of errors.

Conclusion

With the new continuous integration and continuous deployment pipeline, the team has increased its productivity. Now the team can make smaller code changes, and the changes are simpler and easier to handle than huge chunks of code and would have fewer issues that may come up later. The team now can isolate faulty code changes as well from the separation of service environments. The CI/CD pipelines help the team reduce the Faster Mean Time to Resolution (MTTR) because the code changes are
smaller and fault isolations are easier to detect. With the pipelines, the reliability of the web application improves due to the smaller size and specific changes that are being introduced to the applications. It also helped with increasing the team members’ transparency and collaboration from the feedbacks from other members of the team. And most importantly, from the old way of uploading images to Dockerhub, connecting to a remote server, and updating the service manually, to the new CI/CD pipeline which does all of the manual labor fast and automatically has increased the team’s productivity and confidence.
References

https://aws.amazon.com/ec2/


https://www.jenkins.io/

https://www.oracle.com/java/technologies/java8.html


https://www.mongodb.com/


https://github.com/

Wikipedia. (n.d.). Continuous deployment. Retrieved from Wikipedia:
https://en.wikipedia.org/wiki/Continuous_deployment

Wikipedia. (n.d.). Continuous integration. Retrieved from Wikipedia:
https://en.wikipedia.org/wiki/Continuous_integaration