READ ME PAGE:

Contents.

1. Brief

An application was given to us, and we were expected to work within guidelines to create a Continuous Integration pipeline that includes all of the technologies learned in class. The CI pipeline must be able to automatically build, test and redeploy the application whilst staying live (or with minimal downtime). We also must create and include our research, risk assessment and any extra planning we used to complete the project.

Below is a list of the technologies we must use to complete the task:

- Project Management - Jira (Scrum Board)
- Version Control - Git
- CI Server - Jenkins
- Cloud Server - Amazon Web Services (AWS) EC2
- Database Server - AWS RDS
- Containerisation - DOCKER
- Reverse Proxy - NGINX

2. MoSCoW prioritisation lists:

Must have:

- When files are adjusted in **github**, or via the **instance**, **Jenkins**(being the **CI server**) will pull from github the information as soon as they go live using webhooks and will build the changes into the currently running service without stopping the service and then test and deploy the changes in the application.
- We must also have a full **Jira Board** with full expansion on tasks needed to complete the project.
- We must have a **risk assessment**.
- The application must be deployed using **containerisation** and **orchestration tools**.
- The application must be tested through the **CI pipeline**.
- The project must make use of a managed **database server**.
- Our project must make use of a **reverse proxy**.

Should have:

- Deployed using docker swarm

Could have:

- Increase security in the network as a whole.
- Move the database to the private subnet so it is not accessible publicly.

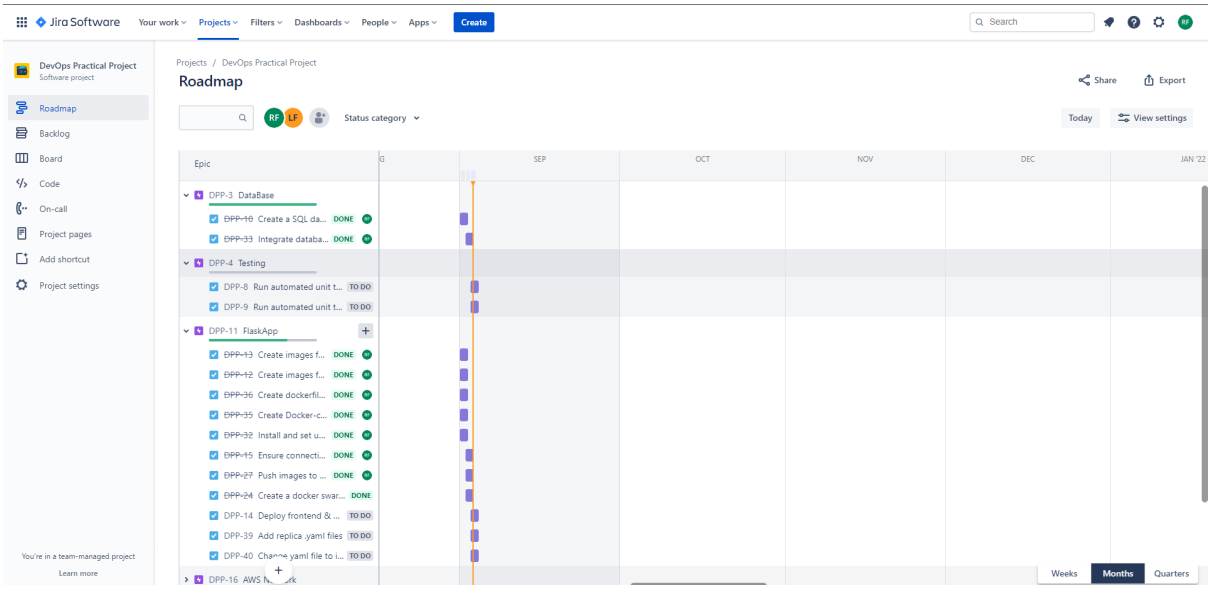
Won't have:

- Any functionality aside from the CI pipeline functions and Cloud accessibility.

3. Jira Board

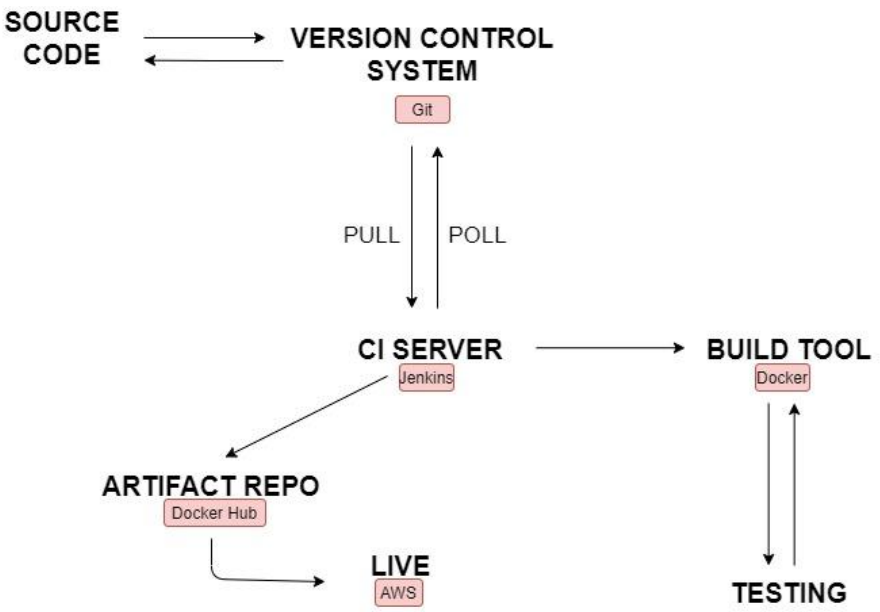
You can find the Jira Board here:
https://lilfinn.atlassian.net/jira/software/projects/DPP/boards/5/roadmap
We used this to create a detailed plan on how we will go about creating and implementing the project. We set epics: DataBase, Testing, FlaskApp, AWS network, Jenkins, GitHub and Documentation
We had 7 epics for each topic of the project, each Epic had assigned to it multiple tasks. Each story was given a priority assessment and a story point estimate.

The stories were then placed into a sprint which we would create for each day, moving across completed tasks.

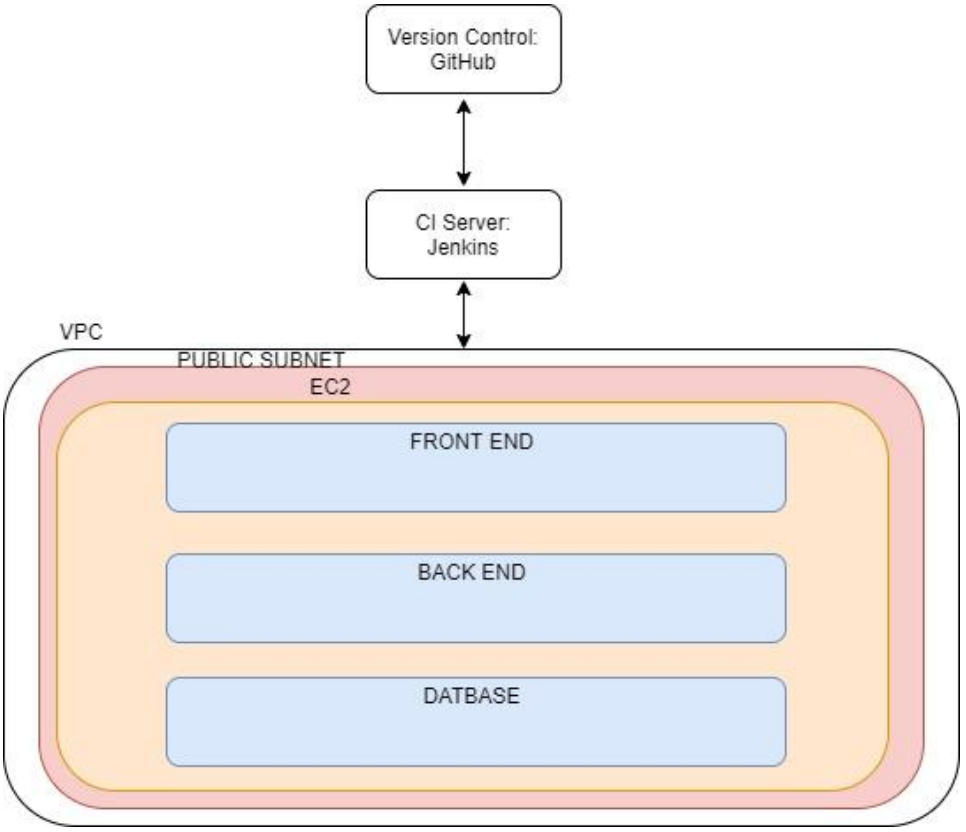


Diagrams used for planning:

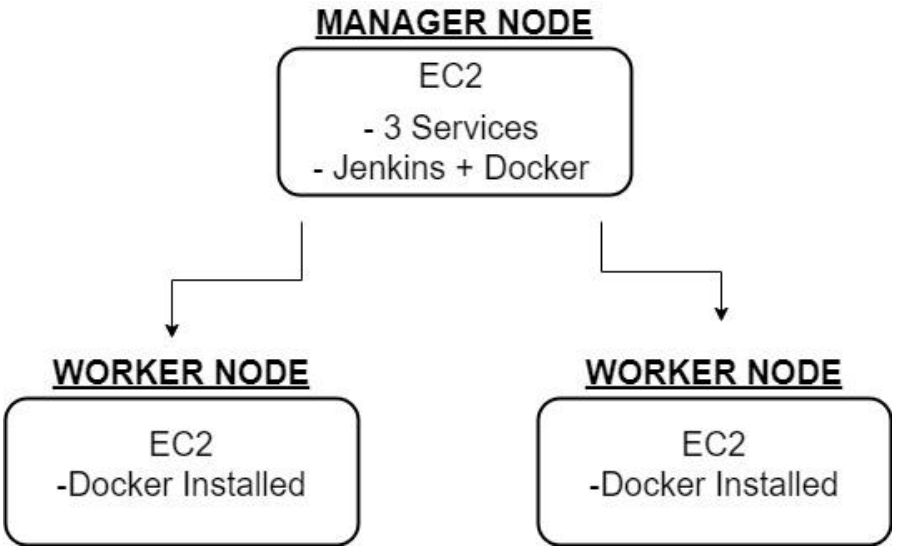
CI Pipeline



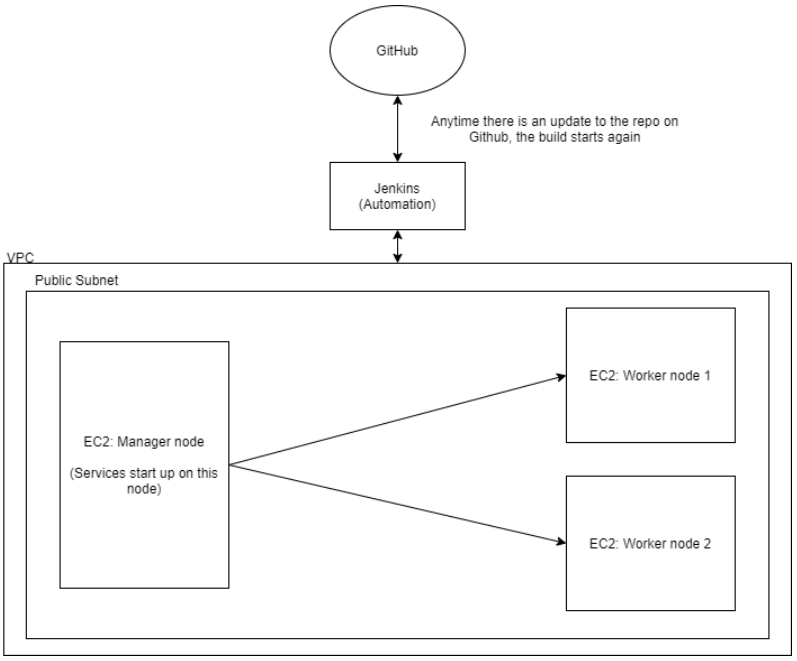
CI pipeline (using Docker Compose)



Docker Swarm Integration.



Full Network



4. Risk Assessment

Description	Evaluation	Likelihood	Impact	Responsibility	Response	Control Measures
AWS service costs	Using the wrong instance type/size	Not very likely	Potentially very high	Us as developers	Would result in potentially very high costs to QA	Put a spending cap in place, use "free tier" where available, take extra care to make sure the right services are being created/used.
Service set-up	Setting up the VPC with the wrong CIDR	Likely	Effect on project timeline: high	Us as developers	Would result in the swarm not being able to deploy, meaning that all services would have to be made from scratch	In case that needs to be done, back up all images to Docker Hub and any files to GitHub.
Data loss	Losing any created files and images	Not very likely	Potentially very high	Us as developers	Would result in having to completely restart the project	Complete regular pushes to GitHub and Docker Hub to prevent loss.
Gaps in Knowledge	Struggling in areas of the project where it's new to us.	Likely	Medium	Us as developers	Can cause us to slow down and potentially not reach project deadline.	Ask for help from tutor and research topics where needed to ensure there isn't an impact on the project.
Illness	One of the two team members becoming ill	Not very likely	Potentially high	Us as developers	Could cause us to slow down dramatically and potentially not reach project deadline.	Take extra care of yourself throughout the duration of the project. Eat healthy, exercise and get plenty of rest.

5. **Creation of AWS structure**

Create: VPC, Subnets and Security Groups

Set up: EC2 and RDS instances

- **VPC:** The VPC allows us to set security rules, thus adding a level of security from the beginning. We used all the default settings, and set the class to BAE12 so we can keep track of resources, and name so it's easy to locate.
- Our Original CIDR had to be changed because Docker Swarm used the same address range and it didn't allow Docker Swarm to work, because of this we had to rebuild our VPC.

- **Set up EC2 and RDS:**

  When we SSH into our EC2 we installed these software packages: Docker, Jenkins, Python 3 and Docker Compose.

  RDS mysql created and assigned a security group.
  We created a RDS database called FlaskApp-db in AWS. We chose a standard create, and chose a secure password. We almost charged QA with a total of $932.88, this is a risk we managed to avoid by checking over everything before creating the RDS database. We originally did not select "free tier".

6. **Microservice setup**

SSH into EC2 instance, install Docker and Docker-Compose

Basic Services set up:

Nginx: nginx.conf file set up to run as a reverse proxy, redirecting http traffic to the correct port to the EC2's, where the Flask app frontend is set to listen.

Frontend: This section required python, and app.py notified us what port needed to be exposed.

Backend: Not that much unlike the front end, the back end also needed python but we needed a different port to be exposed.

Service set up:

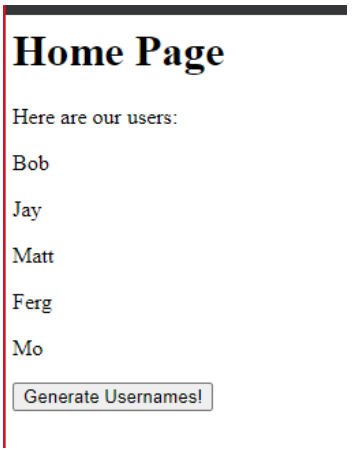
We ran *sudo apt install mysql-client-core-5.7 -y* into the terminal to install mysql into the EC2. The *Create.sql* file was copied to the RDS instance to make it usable with the app by creating the database on the RDS and pre-populating the fields.

For the Nginx, Frontend and Backend containers, we created a *docker-compose.yaml* file to allow for docker compose to create these containers on the EC2 instance.

By using *curl localhost* in the console let us see the HTML format of the data found in the database, and by entering the EC2's public IP address in the browser, we could then confirm so far everything is functional. At this point, the app is running on this

single EC2 instance and everything is manual without any real automation.

```
Creating bae-12-practical-project_nginx_1     ... done
Creating bae-12-practical-project_flask-app_1 ... done
Creating bae-12-practical-project_backend_1   ... done
[redacted]~/BAE-12-Practical-Project$ curl localhost

    <h1>Home Page</h1>
    <p>Here are our users:</p>
    <form action="/" method="get">

        <p>Bob</p>

        <p>Jay</p>

        <p>Matt</p>

        <p>Ferg</p>

        <p>Mo</p>

        <button onClick="window.location.reload();">Generate Usernames!</button>
    </form>
[redacted]/BAE-12-Practical-Project$
```



This confirmed that we had set everything up correctly and that it was functional. Then uploaded the images to Docker hub.

7. Docker Swarm Stack

This is where we created our other 2 EC2's, called Project2-worker1 and Project2-worker2. We assigned them to the same security groups as the EC2 Project2-manager.

On the two worker EC2s we installed docker, it had to be using the correct IP to be able to join swarm by allowing it in the security group.

Docker swarm init was used on the manager node to create the swarm, and the other EC2's are connected by the use of a token.

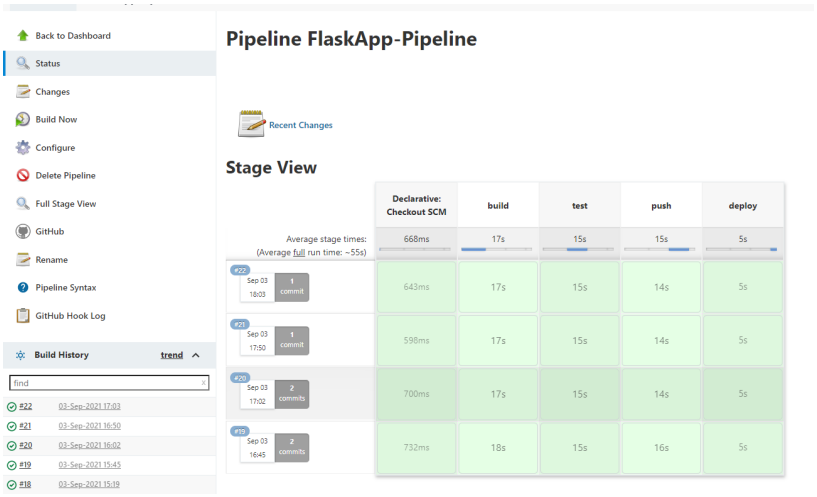
We then deployed the stack by using: *docker stack deploy --compose-file docker-compose.yaml project-stack.* This takes the docker-compose.yaml file we made to produce the docker swarm stack and deploy the services across the 3 EC2s.

```
ID             NAME                  MODE         REPLICAS   IMAGE                                          PORTS
lvoap4l93v6k   flask-stack_backend   replicated   1/1        robbiefletcher2000/flaskback-image:latest      *:5001->5001/tcp
u69aoxx2oye8   flask-stack_flask-app replicated   1/1        robbiefletcher2000/flask-image:latest          *:5000->5000/tcp
ewsu9o5r85r9   flask-stack_nginx     replicated   1/1        robbiefletcher2000/nginx:latest                *:80->80/tcp
[redacted]/BAE-12-Practical-Project$ curl localhost

    <h1>Home Page</h1>
    <p>Here are our users:</p>
    <form action="/" method="get">

        <p>Bob</p>

        <p>Jay</p>

        <p>Matt</p>

        <p>Ferg</p>

        <p>Mo</p>

        <button onClick="window.location.reload();">Generate Usernames!</button>
    </form>
```

8. Jenkins

- We then created the pipeline and associated it to our Github repo, and wrote a Jenkinsfile

- Jenkinsfile on Github setup to run Docker on EC2.
- Set Environment variables for the Database URI, Secret Key and Docker Password.
- Also added a test section to the Jenkins file allowing for unit tests on the frontend and the backend services.



9. Adding Webhooks

A webhook automatically triggers the build of Jenkins project upon a commit pushed in a Git repository. In order for builds to be triggered by PUSH and PULL requests, a Jenkins Web Hook needed to be added to the GitHub repository. This is very useful to the continuous integration set up with Jenkins because this tells Jenkins to attempt a new build only when a change is detected.

10. Summary

<u>What went well?</u>

- We have a completed CI pipeline that meets the project MVP requirements.
- We managed to integrate Docker Swarm as part of the network allowing for the application to be deployed across three EC2s.
- We completed all MVP requirements within the timeframe without too many setbacks/issues.

<u>What didn't go so well for the project?</u>

- After having created the Network to the point where Docker Compose was integrated, we reached a point where we had to rebuild our network from scratch, luckily we had backed up the majority of our project so we were able to pull it back from github and docker hub. This was an issue that was experienced by all teams in the Cohort. We only had two known ways to fix this. One would have us giving the database a publicly accessible endpoint (which would mean we would have no security for our database) OR recreate the entire VPC and network on a different CIDR block.
  We decided to recreate the whole thing as this kept the database from being publicly accessible.

<u>What could we have done differently?</u>

- We would have created a more secure network, including a private subnet for the Database to be in. This would keep the information in the Database from being as accessible.
- We could have increased the number of replicas to allow for any failures or to help manage higher levels of traffic.

Best things we have learned?

- We have learned how to use these services and how to build a network using all of the software listed above. This was good practice, and we will now be able to build a functioning CI pipeline.
- The use and importance of AWS and its products.