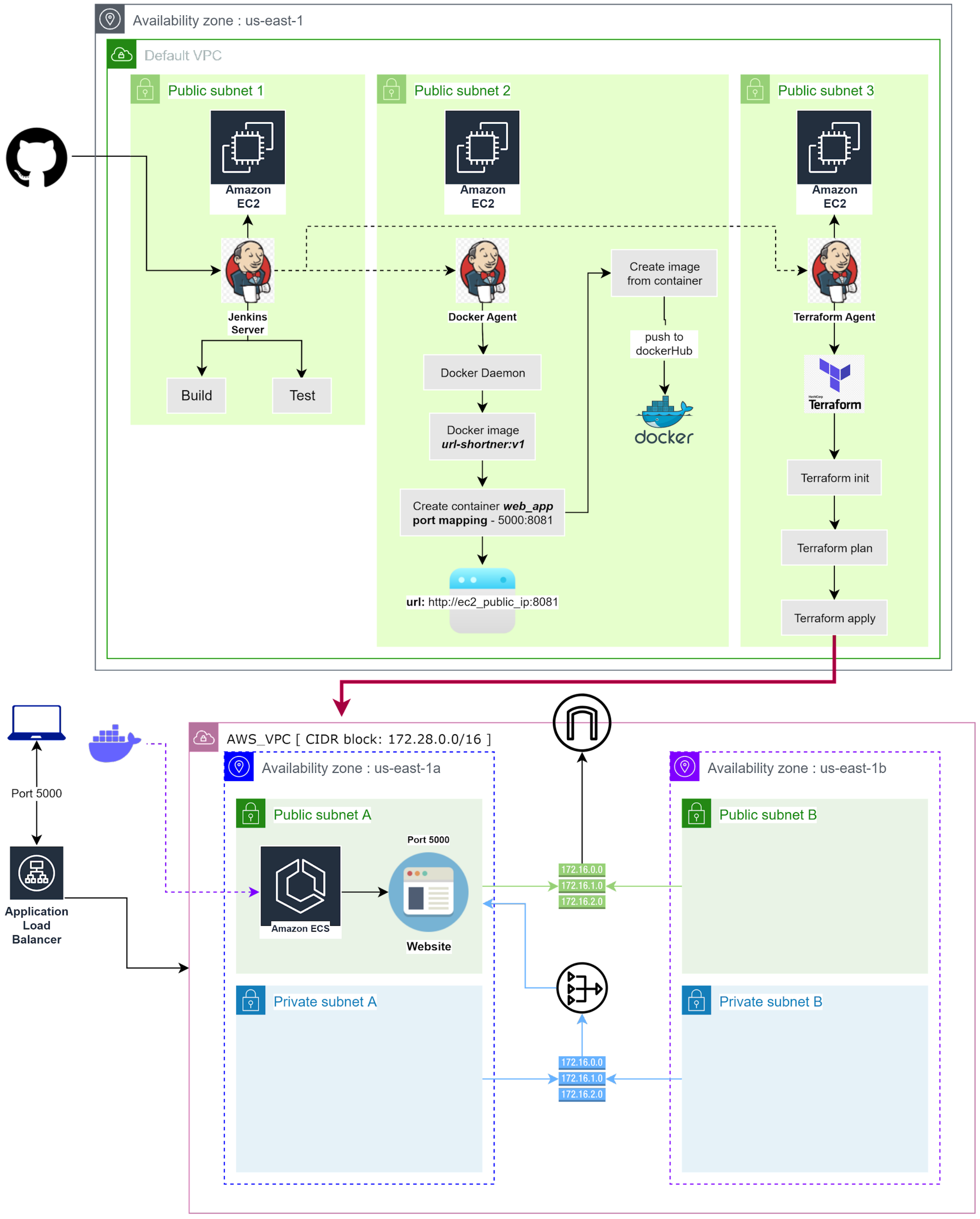
[Deployment 5 Documentation]

Author: Suborna Nath
Date: November 15, 2022

Description: In this deployment, we dockerized a Python application and deployed it with AWS ECS and Terraform.

Diagram



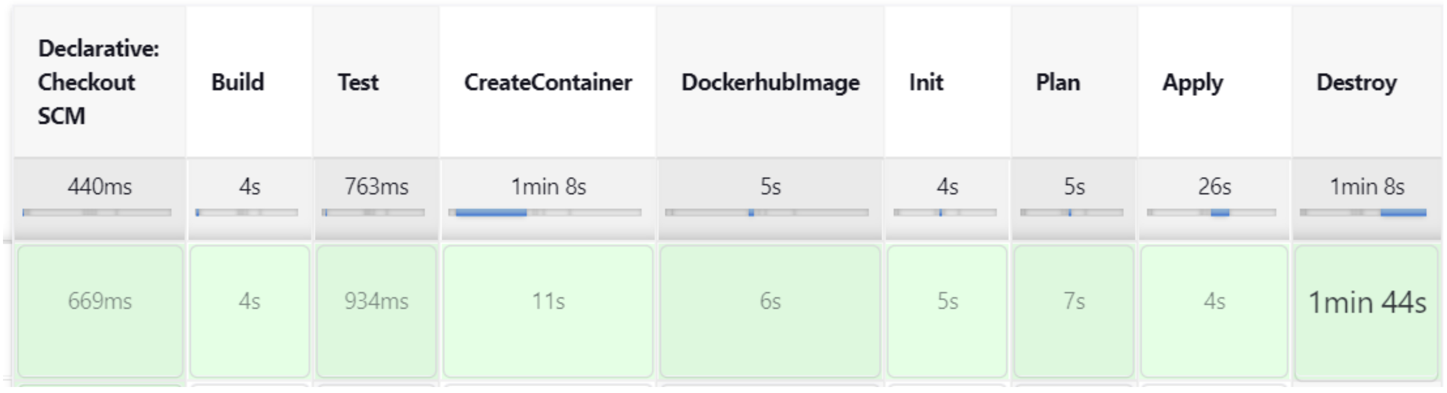
Initial Setup: Setup EC2, Jenkins Agent & dependencies

At first, we need to set up 3 EC2. One of them will be used for the Jenkins server and two of them will be used as Jenkins agents. All of these EC2 are in the same VPC, availability zone, and public subnets and they need Jenkins installed and running. The build and test stage will happen in the first EC2 and needs all the dependencies for running the Python application. The second EC2 needs Docker installed and the last EC2 needs Terraform.

Before running the pipeline, we prepared a *dockerfile* to pull the latest Python image and clone the deployment 5 repository to get all the files related to the app. We added steps to download all the dependencies for running the app referencing *requirements.txt*. Then, we expose port 5000 and run the Python app. The file is then pushed to GitHub for use with the pipeline.

In the Jenkins server, we will need to add credentials for AWS (access key and secret key) so that Terraform has proper permissions to create the resources.

Pipeline Walkthrough



Declarative: Checkout SCM	Build	Test	CreateContainer	DockerhubImage	Init	Plan	Apply	Destroy
440ms	4s	763ms	1min 8s	5s	4s	5s	26s	1min 8s
669ms	4s	934ms	11s	6s	5s	7s	4s	1min 44s

1. Declarative: Checkout SCM

- In this initial stage, Jenkins will access the GitHub repo and pull the latest version of the source code. It will add the project files to the Jenkins workspace.

2. Build

- Python environment is getting created and activated
- Latest version of pip is installed
- Installing all the required packages from requirements.txt
- Setting the application name "application.py" as environment variable
- Running the Flask application

3. Test

- Activating the python environment
- Running the py tests written in "test_app.py" and saving the results as results.xml
    - Testing the homepage to make sure we can access it
    - Testing an invalid URL to make sure it gives a 404 error
    - Testing a saved URL to make sure it redirects the request with a 302 status code

Troubleshoot: The repository did not have the saved URLs which led to the last test case to fail. A file named *urls.json* was added, which includes key-value pairs of the URLs in JSON format.

4. CreateContainer {Docker Agent}

- We used *Docker build* to create an image using the *dockerfile* named *url-shortner:v1*
- We then run a container of that image which maps the port 5000 to 8081. The container is named *web_app*. We can access the app with the EC2_public_ip at port 8081.

5. DockerhubImage {Docker Agent}

- We commit any changes made to the container *web_app into the image*
- We tag our local image (web_app:v1) to the repository we want to send the image to (suborna/web_app:v1)
- We use command *docker login* to login to dockerhub before sending the image
- After that, we push the image into dockerhub

6. Init {Terraform Agent}

- In the terraform directory *intTerraform*, we are initializing the terraform environment and downloading all the dependencies. We are creating a terraform environment.

7. Plan {Terraform Agent}

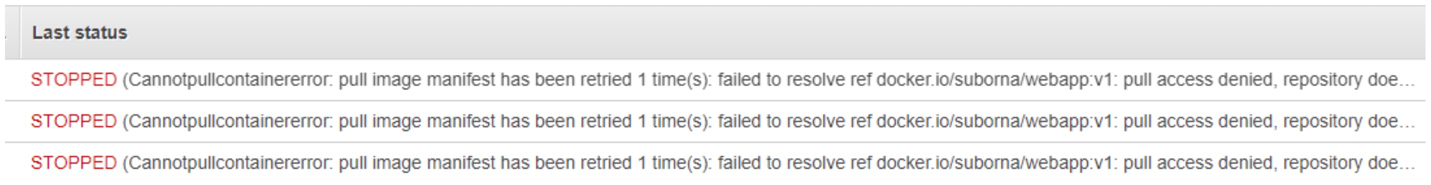
- In the same directory *intTerraform*, terraform is planning out all the steps that need to be taken to create the infrastructure for the application by referencing all the *.tf* files. It checks the current state of the infrastructure and compares it to the desired state and makes an execution plan.

Infrastructure

In our case, we are creating a custom VPC with CIDR block *172.28.0.0/16*. There will be 2 availability zones us-east-1a and us-east-1b. Each zone will have one public and one private subnet. There will also be an application load balancer with port 5000 open. It will redirect all the requests to one of the public subnets.

To access the private subnets, we will need to use public subnet A to go through the NAT gateway to the private routing table. The public subnets have a public routing table that communicates with the internet gateway. The default environment for the application will be public subnet A.

We will be using AWS ECS to create a cluster and a task definition. The task will pull the dockerhub image that we created earlier (step 5) and run a container at port 5000. We will also use Fargate as the compute engine instead of an EC2 so that we only pay for the resources being used.

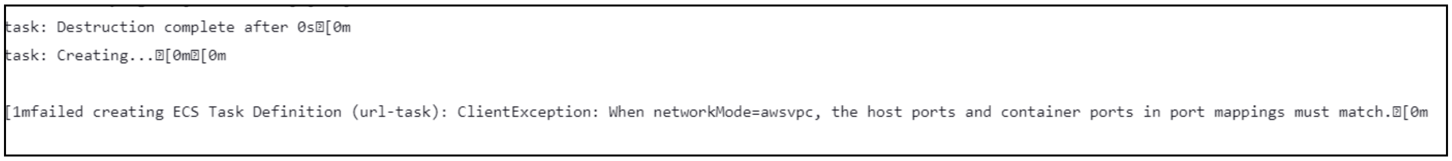


Last status
STOPPED (Cannotpullcontainererror: pull image manifest has been retried 1 time(s): failed to resolve ref docker.io/suborna/webapp:v1: pull access denied, repository doe…
STOPPED (Cannotpullcontainererror: pull image manifest has been retried 1 time(s): failed to resolve ref docker.io/suborna/webapp:v1: pull access denied, repository doe…
STOPPED (Cannotpullcontainererror: pull image manifest has been retried 1 time(s): failed to resolve ref docker.io/suborna/webapp:v1: pull access denied, repository doe…

Troubleshoot: After creating the ECS cluster and task definition, my task was not running. After assessing the log message for the task, I realized that my repository name had a typo. I needed to fix the image name in the *main.tf* file.

8. Apply {Terraform Agent}

- We will be creating all the resources described in the previous step.
- The application can be accessed from the URL created by the application load balancer



```
task: Destruction complete after 0s⍰[0m
task: Creating...⍰[0m⍰[0m

[1mfailed creating ECS Task Definition (url-task): ClientException: When networkMode=awsvpc, the host ports and container ports in port mappings must match.⍰[0m
```

<u>Troubleshoot:</u> Initially, I had the application port as 8081. Even though my application was running, I could not access it. I tried adding the port in the *main.tf* file but later got an error indicating that my container and application need to have the same port open. After I changed the port, I was able to access the webpage.

<u>9. Destroy {Terraform Agent}</u>

- Terraform will tear down all the resources that were created in the previous step including VPC, subnets, route tables, internet gateway, and NAT gateway.

<u>What can be improved?</u>

We can add a backend for the application and have it set up in the private subnets. I would like to set up alerts for the pipeline for successful deployment and for if there was any error. We can also try accessing the webpage with a get request right after the *apply* stage with the pipeline and if there is any issue, we can set up an alert for investigation. With the current setup, we cannot tell by looking at the pipeline if the webpage cannot be accessed.