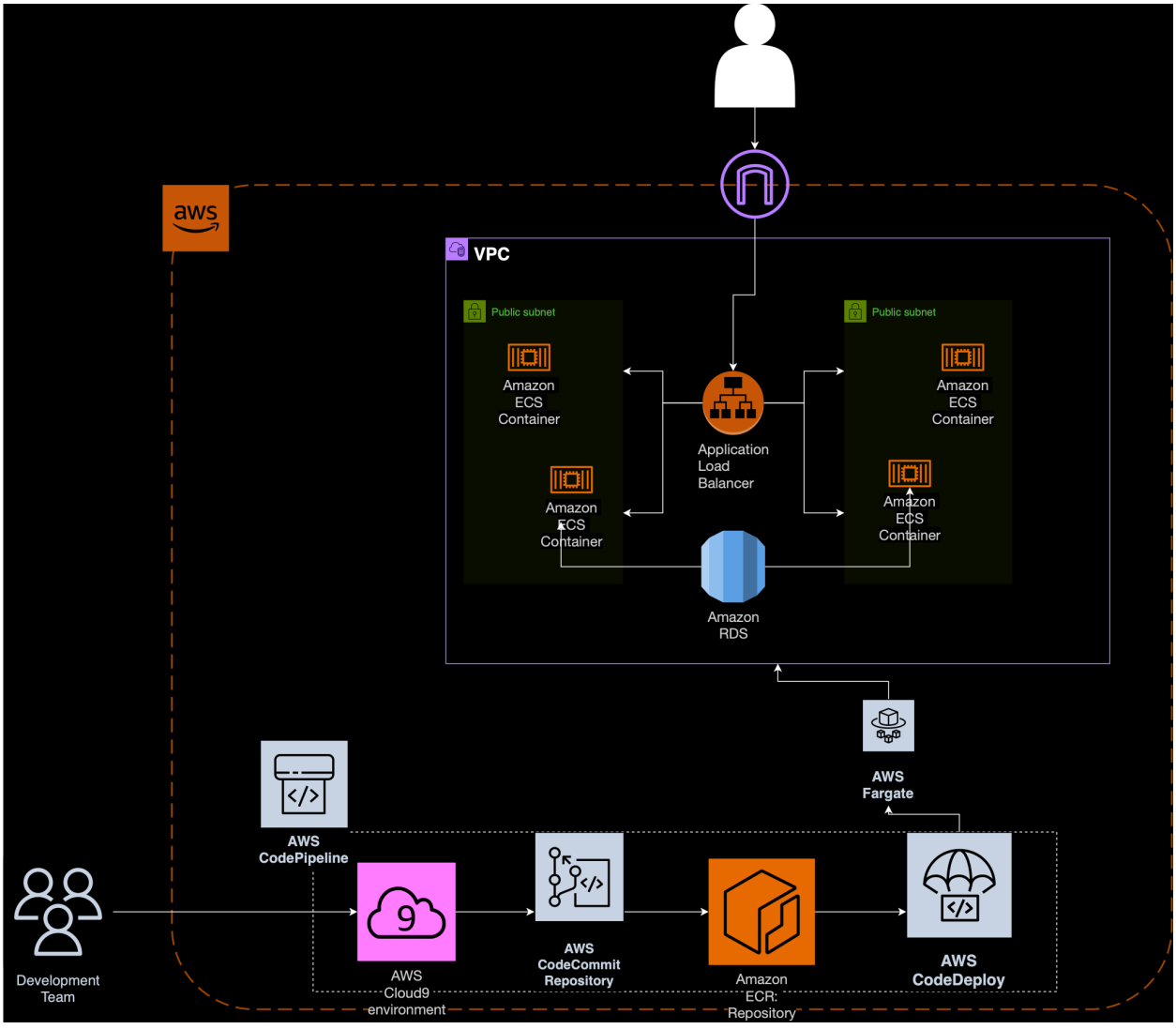
Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu

CS 524 Final Project

Part 1: Project Execution

Phase 1: Planning the design and estimating cost

Task 1.1: Create an architecture diagram



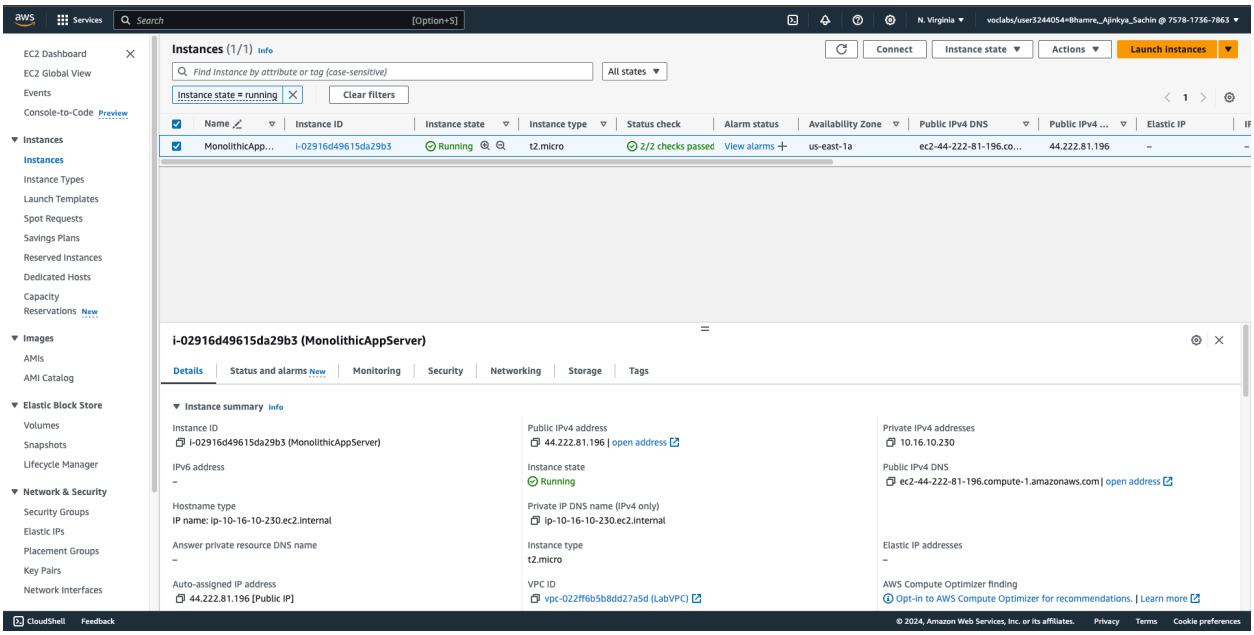
Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu

Phase 2: Analyzing the infrastructure of the monolithic application

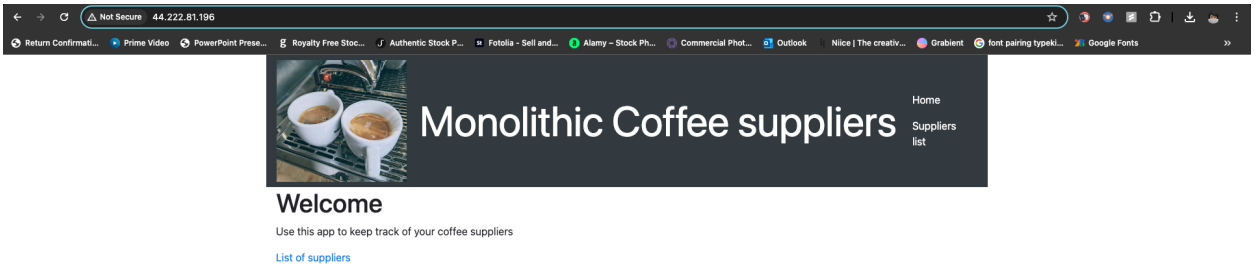
Task 2.1: Verify that the monolithic application is available

1. Verify that the monolithic web application is accessible from the internet.

- Navigate to the Amazon EC2 console.



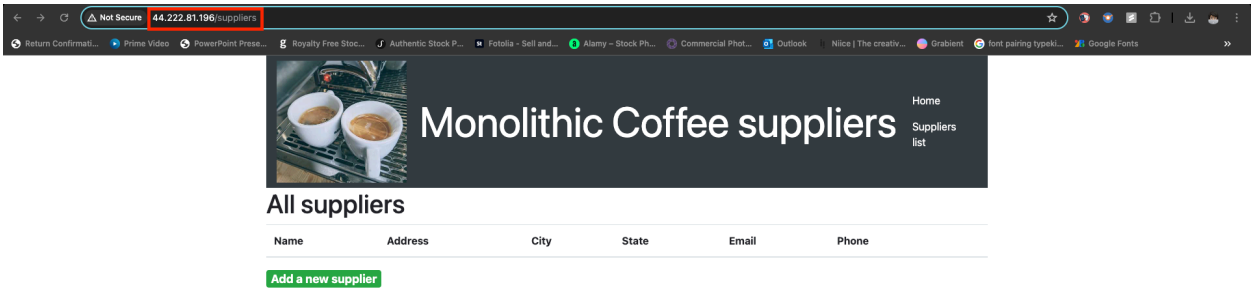
- Copy the Public IPv4 address of the MonolithicAppServer instance, and load it in a new browser tab. The coffee suppliers website displays.



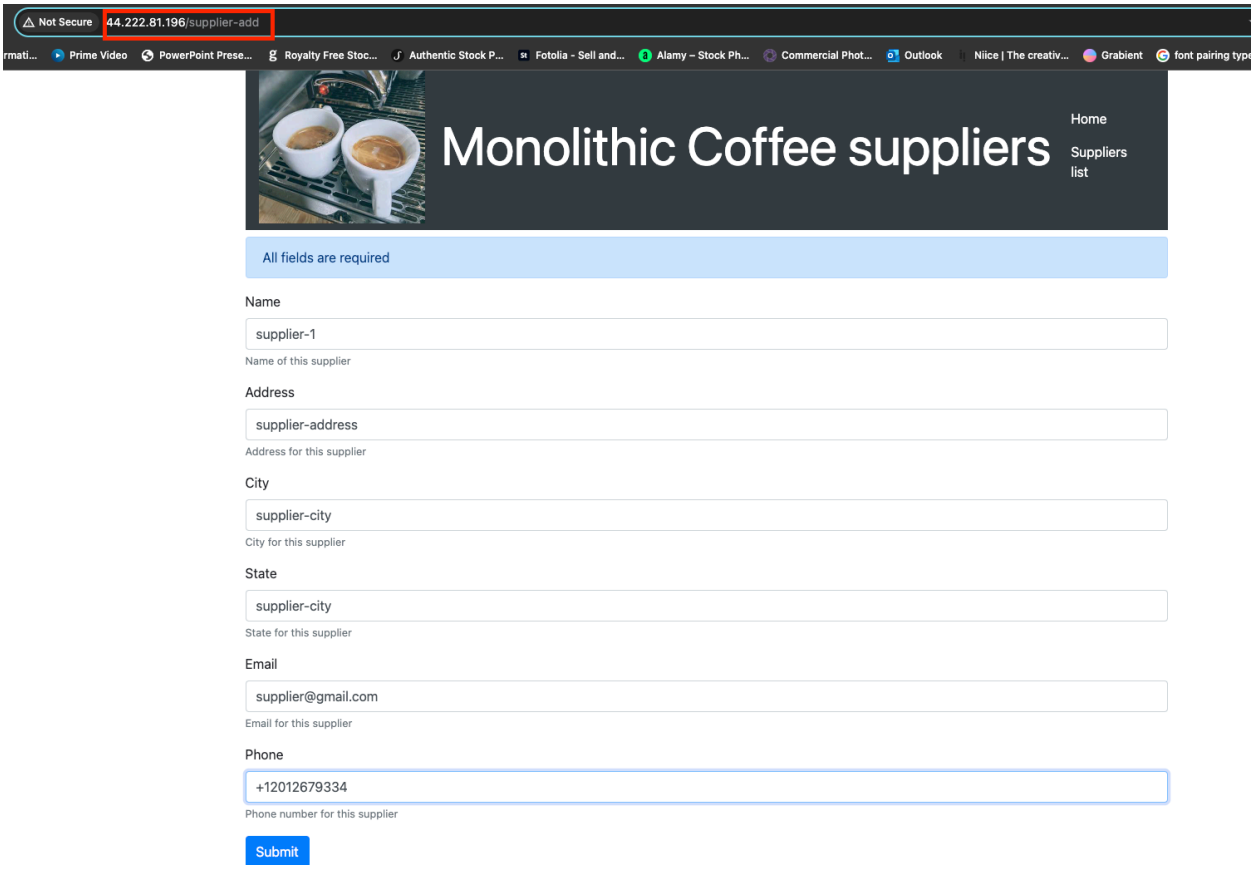
Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu

Task 2.2: Test the monolithic web application

1. Choose List of suppliers. Notice that the URL path includes /Notice that the URL path includes /suppliers.

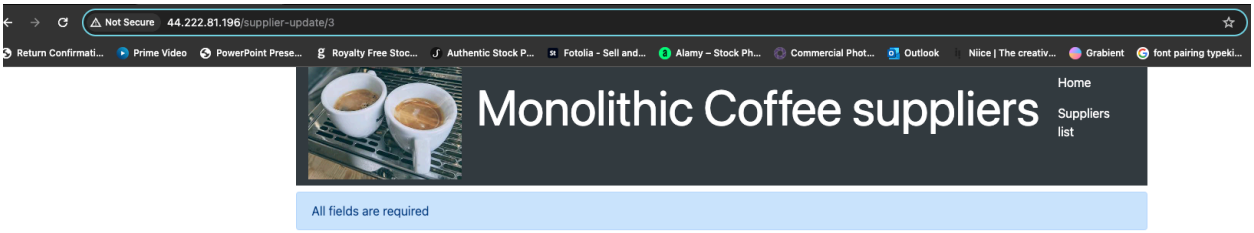


2. Add a new supplier.
- On the page where you add a new supplier, notice that the URL path includes / supplier-add.
- Fill in all of the fields to create a supplier entry.



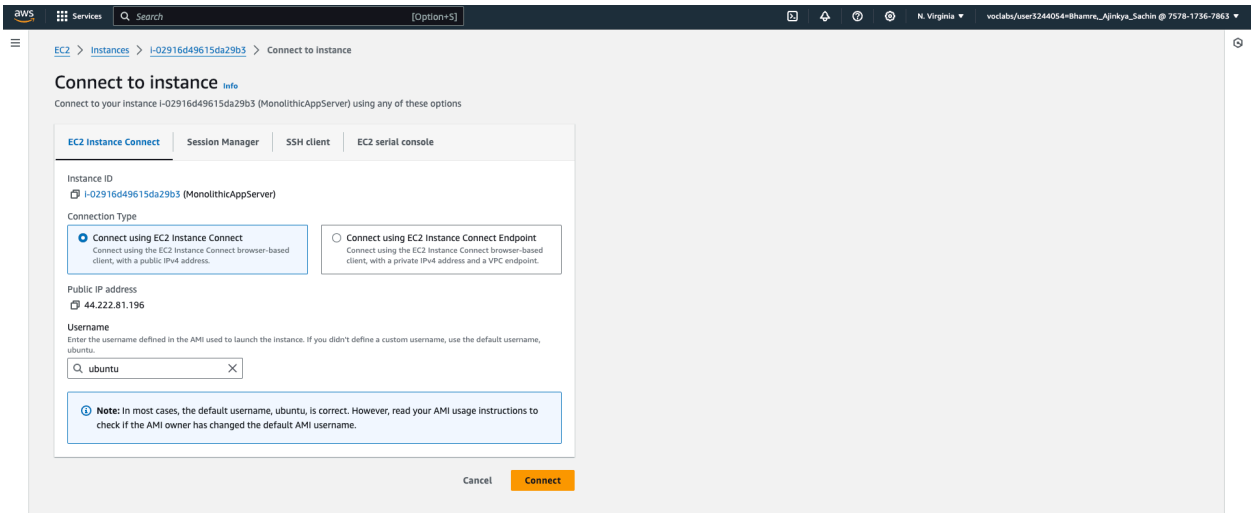
3. Edit an entry.
- On the page where you edit a supplier entry, notice that the URL path now includes supplier-update/1.
- Modify the record in some way and save the change.

Notice that the change was saved in the record.



Task 2.3: Analyze how the monolithic application runs

1. Use EC2 Instance Connect to connect to the MonolithicAppServer instance.



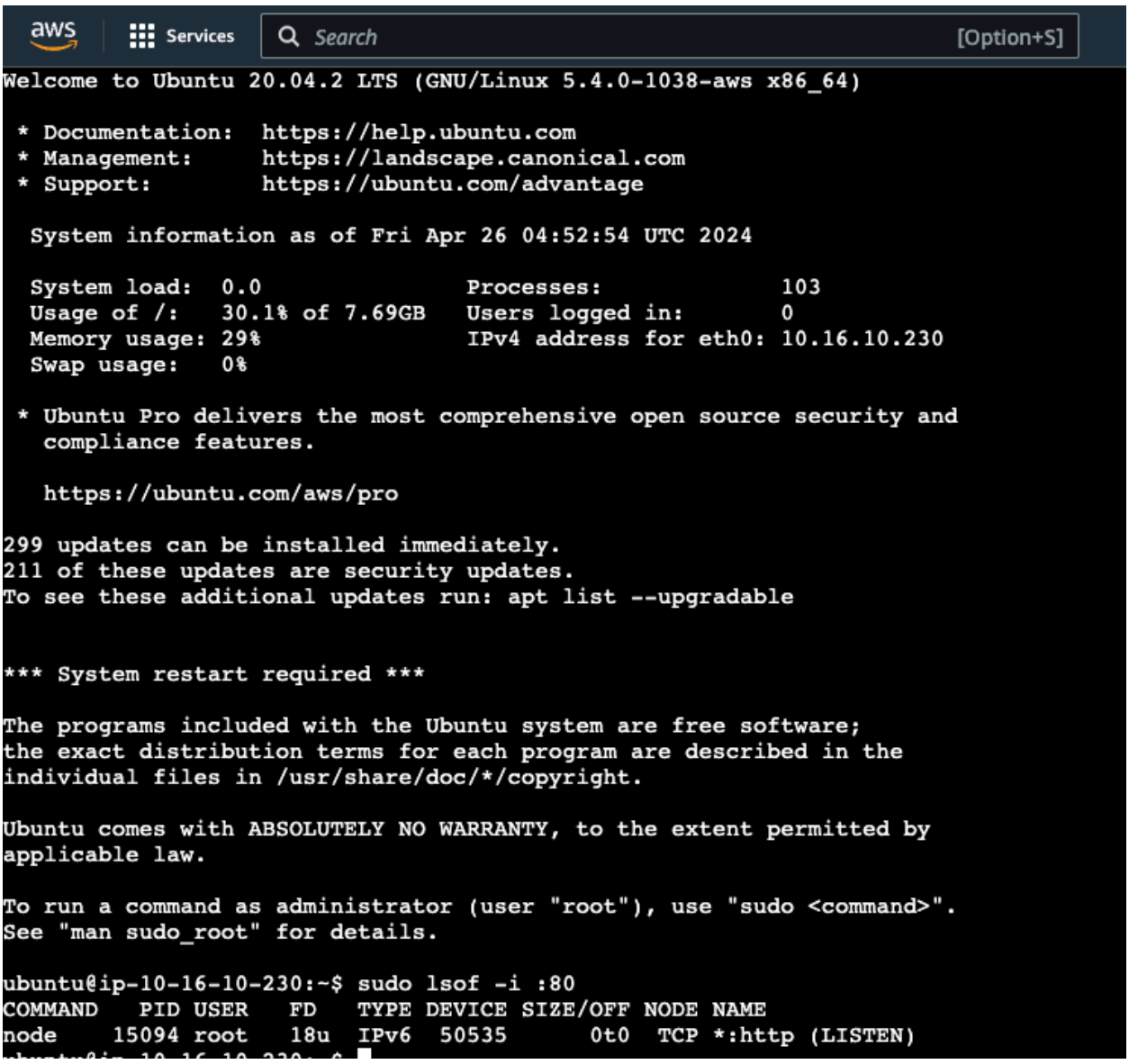
2. Analyze how the application is running.

`sudo lsof -i :80 :`

The command sudo lsof -i :80 is used to list all the processes currently listening on port 80 on the EC2 instance

- COMMAND: Indicates the name of the process listening on port 80. In this case, it's node.
- PID: Represents the process ID (PID) of the node process. It's 15094.
- USER: Specifies the user who owns the process. It's root.

- FD: Stands for File Descriptor and indicates the file descriptor number used by the process. It's 18u.
- TYPE: Describes the type of file or socket. Here, it's a TCP socket.
- DEVICE: Indicates the device number associated with the file. It's IPv6.
- SIZE/OFF: Represents the size of the file or offset within the file. It's 0t0.
- NODE: Specifies the node associated with the file. It's *:http (LISTEN).



From this output, we can see that there is a process with PID 15094 (owned by the root user) listening on port 80. The process name is node, and it's listening on all available IPv6 addresses (*) for incoming HTTP connections (http).

ps -ef | head -1; ps -ef | grep node :

```
ubuntu@ip-10-16-10-230:~$ ps -ef | head -1; ps -ef | grep node
UID          PID    PPID  C STIME TTY          TIME CMD
root       15094       1  0 Apr25 ?        00:00:01 node index.js
ubuntu     16135   16101  0 05:02 pts/0    00:00:00 grep --color=auto node
```

There are two processes listed:

- The first process is the node process with PID 15094. It is owned by the root user and appears to be running index.js.
- The second process is the grep command itself, which is searching for the string "node" in the process list.

So, the node process is being run by the root user on this EC2 instance.

Regarding the PID matching, yes, the PID 15094 from the ps command output matches the PID 15094 from the previous output of the lsof command. Both commands indicate that the node process with PID 15094 is running on the EC2 instance.

3. To analyze the application structure, run the following commands:

```
ubuntu@ip-10-16-10-230:~$ cd ~/resources/codebase_partner
ubuntu@ip-10-16-10-230:~/resources/codebase_partner$ ls
app  index.js  node_modules  package-lock.json  package.json  public  views
```

Questions for thought: Based on what you have observed, what can you determine about how and where this node application is running? How do you think it was installed? What prerequisite libraries, if any, were required to make it run? Where does the application store data?

- The Node.js application is likely running on an Ubuntu server (ubuntu@ip-10-16-10-230) hosted on AWS EC2.
- It appears to be installed in the directory ~/resources/codebase_partner.
- It was likely installed using npm, as indicated by the presence of package-lock.json and package.json.
- Prerequisite libraries were installed via npm and are located in the node_modules directory.
- The application stores data in its directory or possibly in a database, depending on its functionality.

4. Connect a MySQL client to the RDS database that the node application stores data in.

i) Connect to the RDS database:

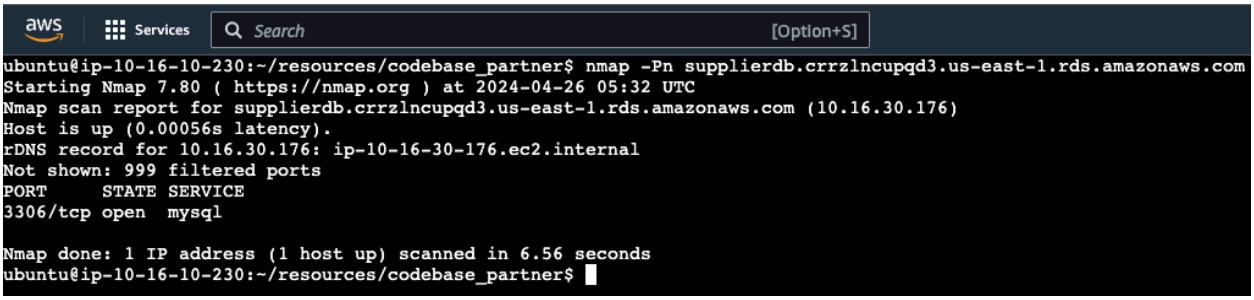
Use the AWS Management Console or AWS CLI to navigate to the RDS dashboard and find the endpoint of the RDS database instance where the node application stores data.

Copy the endpoint URL. (RDSEndpoint supplierdb.crrzlncupqd3.us-east-1.rds.amazonaws.com)

ii) Verify database accessibility:

SSH into the MonolithicAppServer instance.

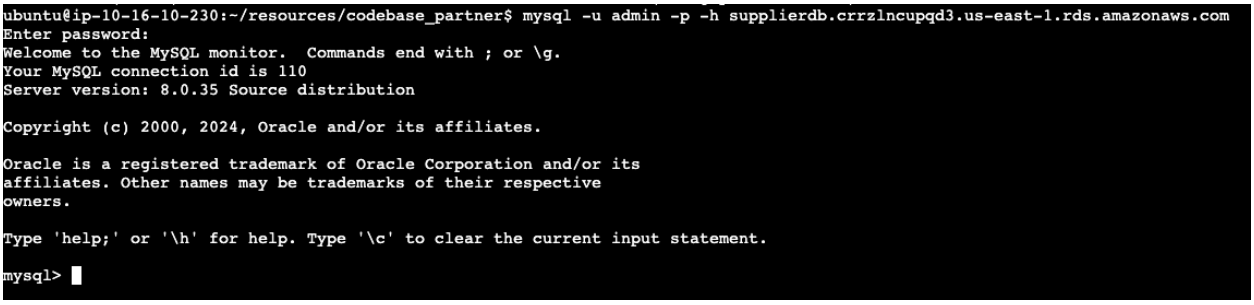
nmap -Pn supplierdb.crrzlncupqd3.us-east-1.rds.amazonaws.com



This command checks if the RDS database is reachable from the MonolithicAppServer instance on the standard MySQL port (usually port 3306).

iii) Connect to the database:

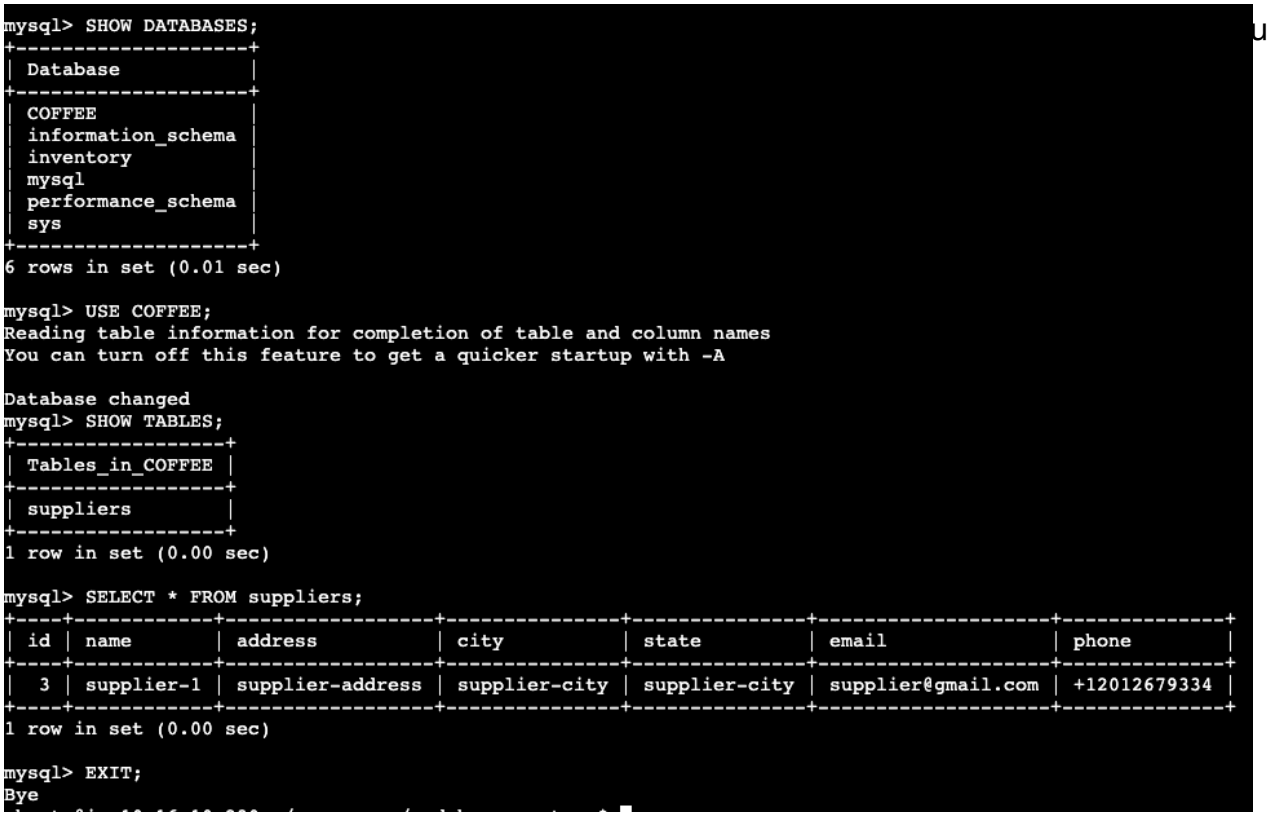
mysql -u admin -p -h supplierdb.crrzlncupqd3.us-east-1.rds.amazonaws.com



5. Observe the data in the database.

i) From the mysql> prompt, run SQL commands as appropriate to see that a database named COFFEE contains a table named suppliers. This table contains the supplier entry or entries that you added earlier when you tested the web application.

ii) Exit the MySQL client and then close the EC2 Instance Connect tab. Also close the coffee suppliers web application tab.



```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| COFFEE             |
| information_schema |
| inventory          |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
6 rows in set (0.01 sec)

mysql> USE COFFEE;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SHOW TABLES;
+------------------+
| Tables_in_COFFEE |
+------------------+
| suppliers        |
+------------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM suppliers;
+----+------------+------------------+---------------+---------------+--------------------+--------------+
| id | name       | address          | city          | state         | email              | phone        |
+----+------------+------------------+---------------+---------------+--------------------+--------------+
|  3 | supplier-1 | supplier-address | supplier-city | supplier-city | supplier@gmail.com | +12012679334 |
+----+------------+------------------+---------------+---------------+--------------------+--------------+
1 row in set (0.00 sec)

mysql> EXIT;
Bye
```

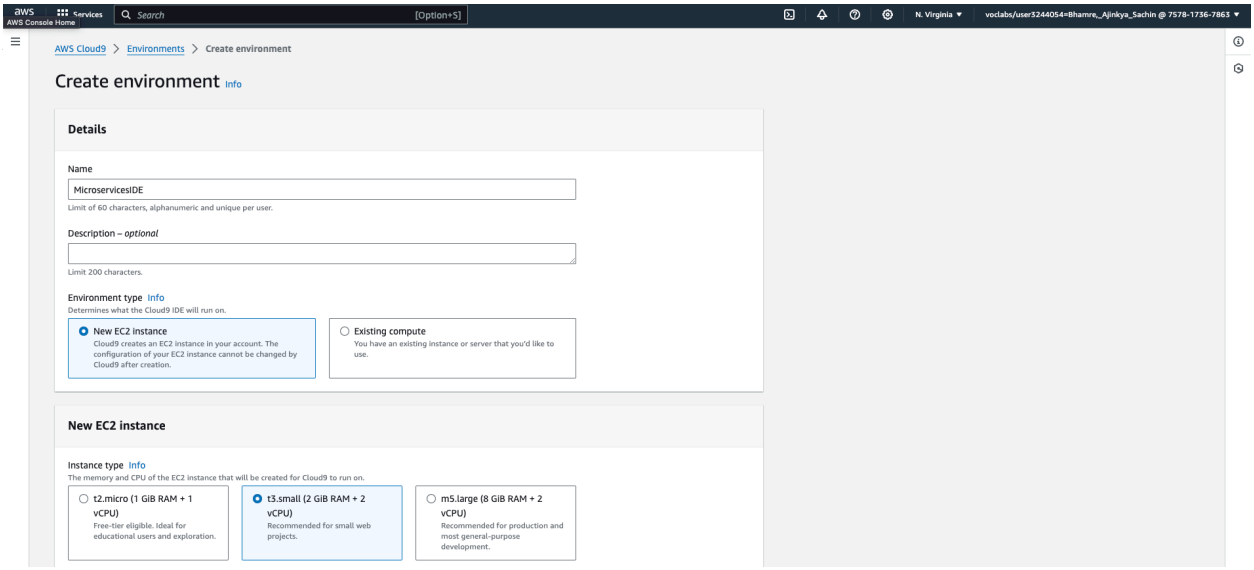
Phase 3: Creating a development environment and checking code into a Git repository

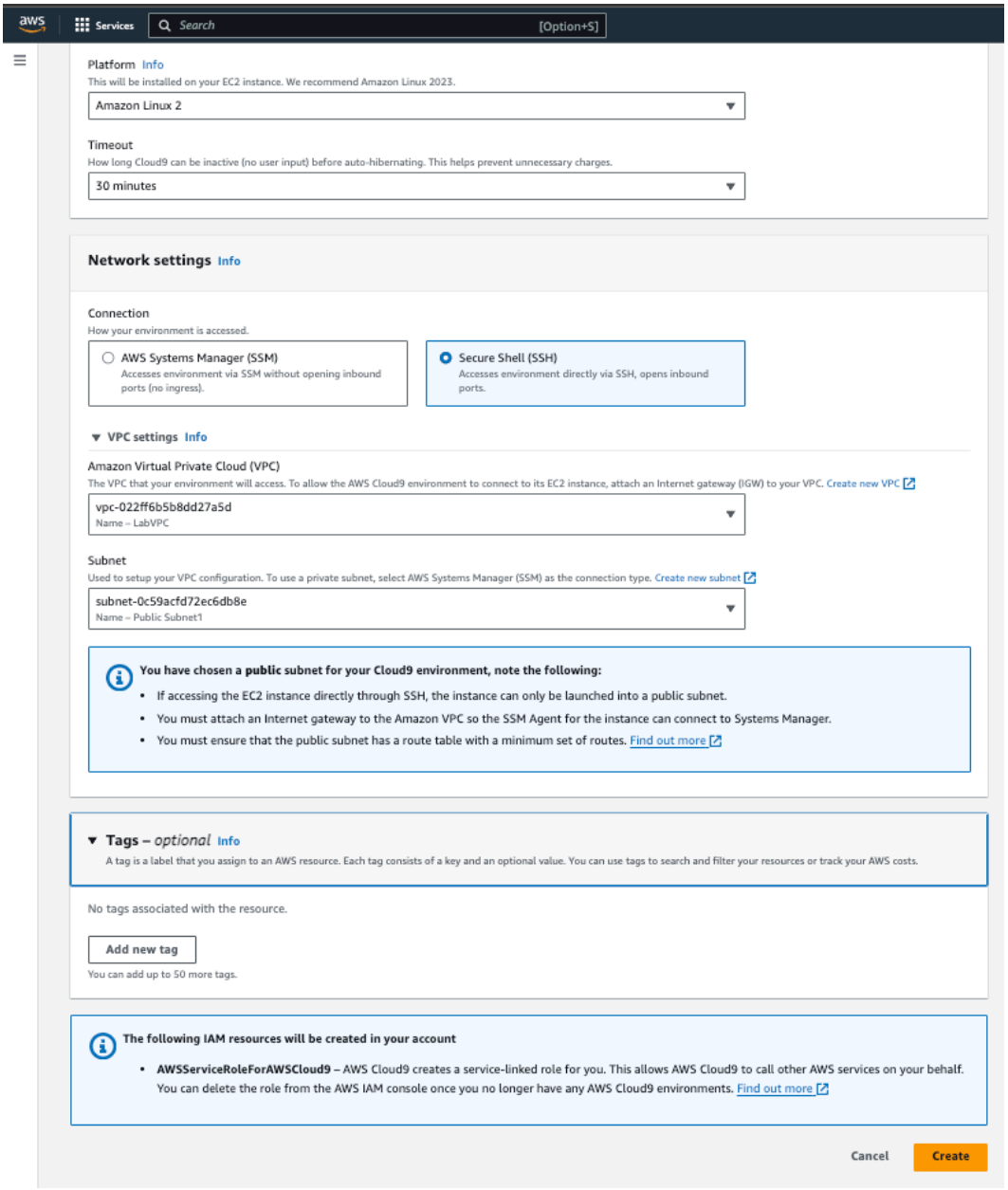
In this phase, you will create a development environment by using AWS Cloud9. You will also check your application code into AWS CodeCommit, which is a Git-compatible repository.

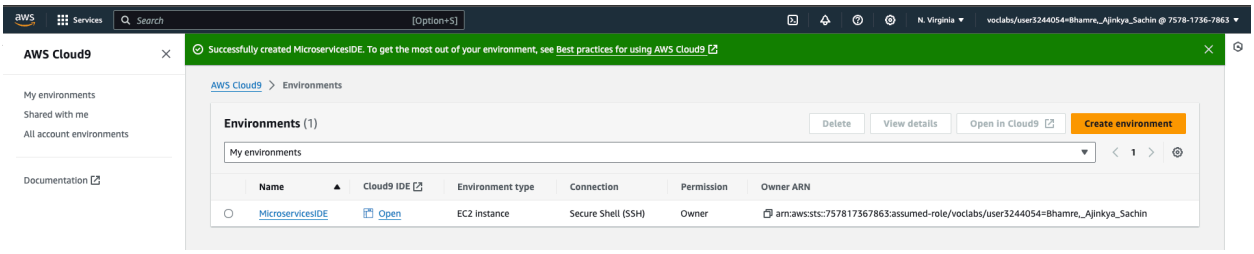
Task 3.1: Create an AWS Cloud9 IDE as your work environment

Create an AWS Cloud9 instance that is named MicroservicesIDE and then open the IDE.

It should run as a new EC2 instance of size t3.small and run Amazon Linux 2. The instance should support SSH connections and run in the LabVPC in Public Subnet1.

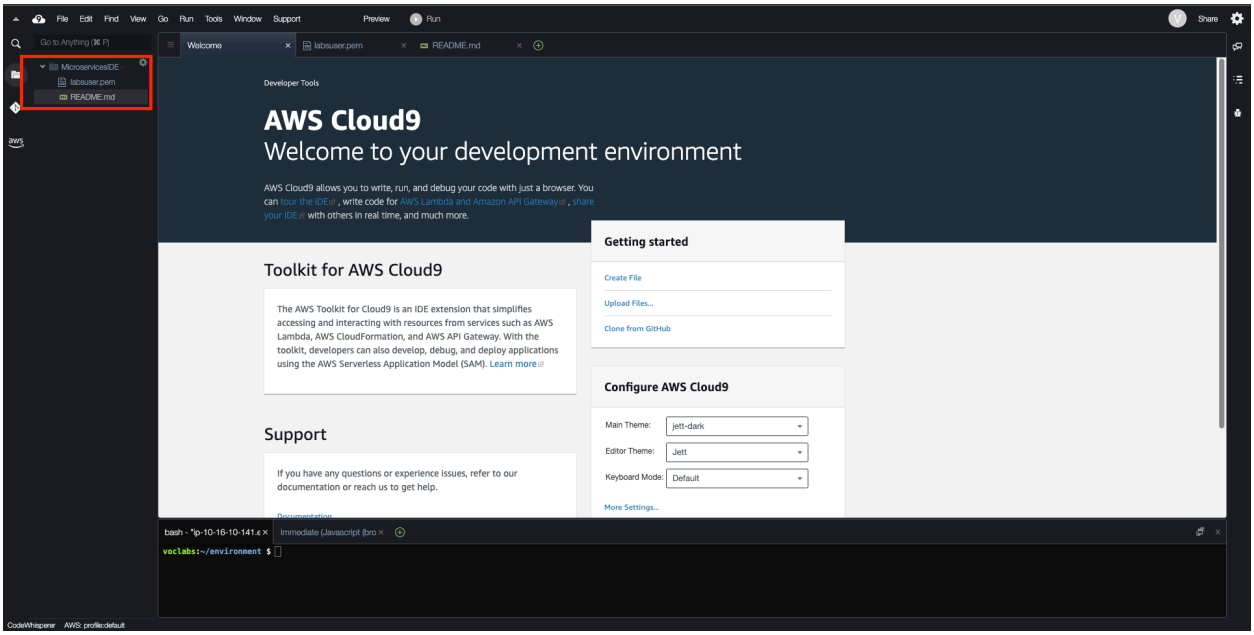

aws
AWS Console Home
Services
Search
[Option+S]
N. Virginia
voclabs/user3244054=Bhamre,_Ajinkya_Sachin @ 7578-1736-7863
AWS Cloud9 > Environments > Create environment
Create environment Info
Details
Name
MicroservicesIDE
Limit of 60 characters, alphanumeric and unique per user.
Description – optional
Limit 200 characters.
Environment type Info
Determines what the Cloud9 IDE will run on.
New EC2 instance
Cloud9 creates an EC2 instance in your account. The configuration of your EC2 instance cannot be changed by Cloud9 after creation.
Existing compute
You have an existing instance or server that you'd like to use.
New EC2 instance
Instance type Info
The memory and CPU of the EC2 instance that will be created for Cloud9 to run on.
t2.micro (1 GiB RAM + 1 vCPU)
Free-tier eligible. Ideal for educational users and exploration.
t3.small (2 GiB RAM + 2 vCPU)
Recommended for small web projects.
m5.large (8 GiB RAM + 2 vCPU)
Recommended for production and most general-purpose development.


aws
Services
Search
[Option+S]
Platform Info
This will be installed on your EC2 instance. We recommend Amazon Linux 2023.
Amazon Linux 2
Timeout
How long Cloud9 can be inactive (no user input) before auto-hibernating. This helps prevent unnecessary charges.
30 minutes
Network settings Info
Connection
How your environment is accessed.
AWS Systems Manager (SSM)
Accesses environment via SSM without opening inbound ports (no ingress).
Secure Shell (SSH)
Accesses environment directly via SSH, opens inbound ports.
VPC settings Info
Amazon Virtual Private Cloud (VPC)
The VPC that your environment will access. To allow the AWS Cloud9 environment to connect to its EC2 instance, attach an Internet gateway (IGW) to your VPC. Create new VPC
vpc-022ff6b5b8dd27a5d
Name – LabVPC
Subnet
Used to setup your VPC configuration. To use a private subnet, select AWS Systems Manager (SSM) as the connection type. Create new subnet
subnet-0c59acfd72ec6db8e
Name – Public Subnet1
You have chosen a public subnet for your Cloud9 environment, note the following:
If accessing the EC2 instance directly through SSH, the instance can only be launched into a public subnet.
You must attach an Internet gateway to the Amazon VPC so the SSM Agent for the instance can connect to Systems Manager.
You must ensure that the public subnet has a route table with a minimum set of routes. Find out more
Tags – optional Info
A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources or track your AWS costs.
No tags associated with the resource.
Add new tag
You can add up to 50 more tags.
The following IAM resources will be created in your account
AWSServiceRoleForAWSCloud9 – AWS Cloud9 creates a service-linked role for you. This allows AWS Cloud9 to call other AWS services on your behalf. You can delete the role from the AWS IAM console once you no longer have any AWS Cloud9 environments. Find out more
Cancel
Create

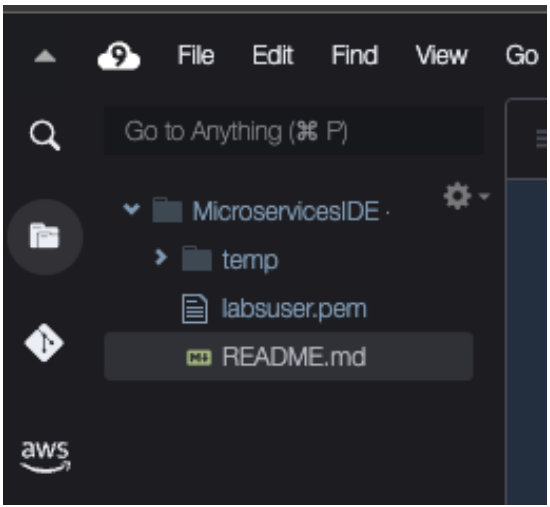


Task 3.2: Copy the application code to your IDE

1. From the **AWS Details** panel on this lab instructions page, download the **labsuser.pem** file to local computer.
2. Upload the .pem file to your AWS Cloud9 IDE, and use the Linux chmod command to set the proper permissions on the file so that you can use it to connect to an EC2 instance.



3. Create a temp directory on the AWS Cloud9 instance at /home/ec2-user/environment/temp.

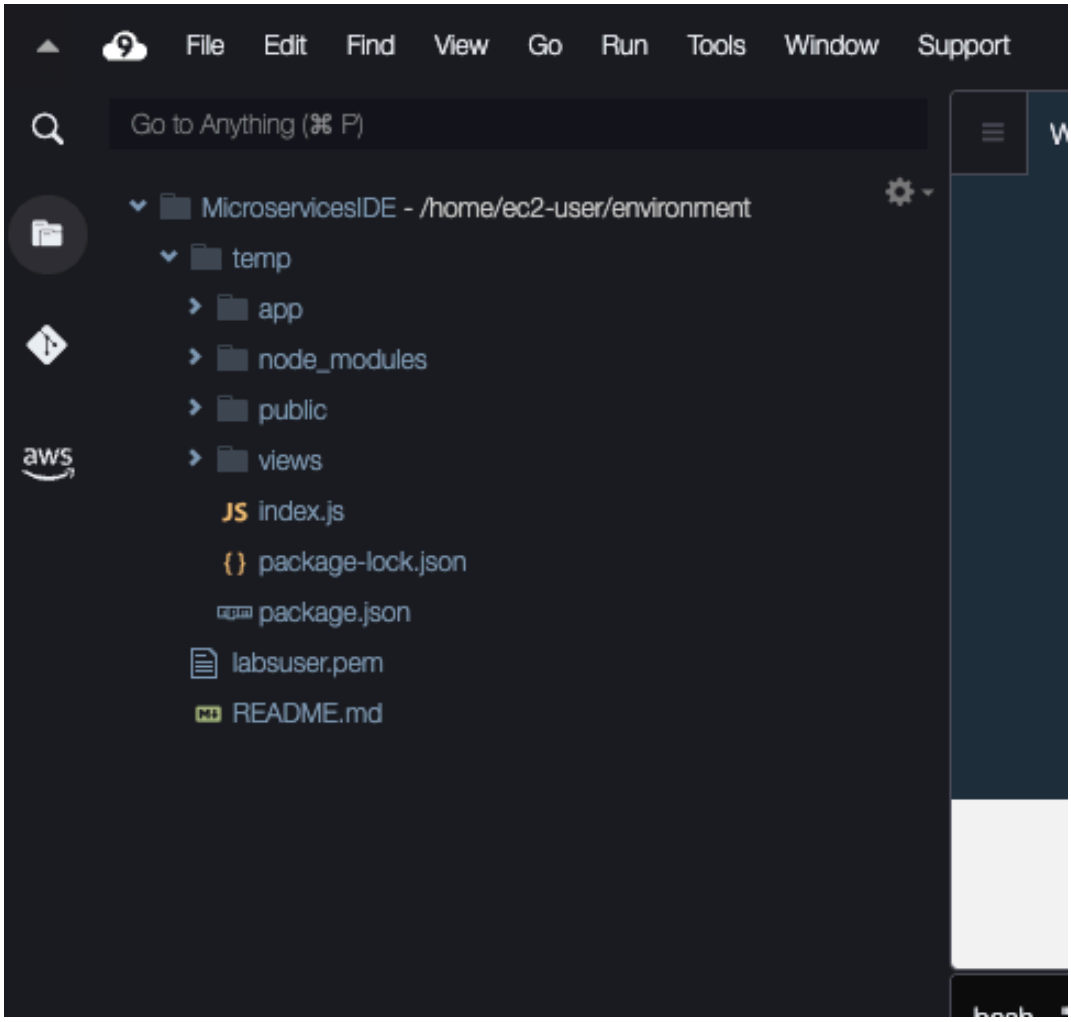


4. From the Amazon EC2 console, retrieve the private IPv4 address of the MonolithicAppServer instance.

Public IPv4 address : 44.222.81.196

5. Use the Linux scp command in the Bash terminal on the AWS Cloud9 instance to copy the source code for the node application from the MonolithicAppServer instance to the temp directory that you created on the AWS Cloud9 instance.

6. In the file browser of the IDE, verify that the source files for the application have been copied to the temp directory on the AWS Cloud9 instance.



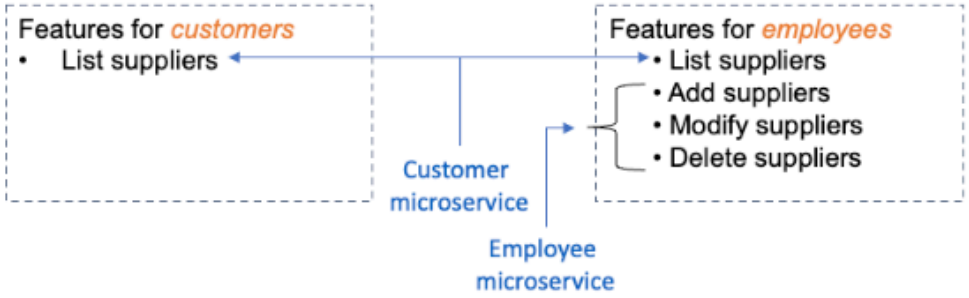
Task 3.3: Create working directories with starter code for the two microservices

In this task, you will create areas in your development environment to support separating the application logic into two different microservices.

Based on the solution requirements of this project, it makes sense to split the monolithic application into two microservices. You will name the microservices *customer* and *employee*.

The following table explains the functionality that is needed for each microservice.

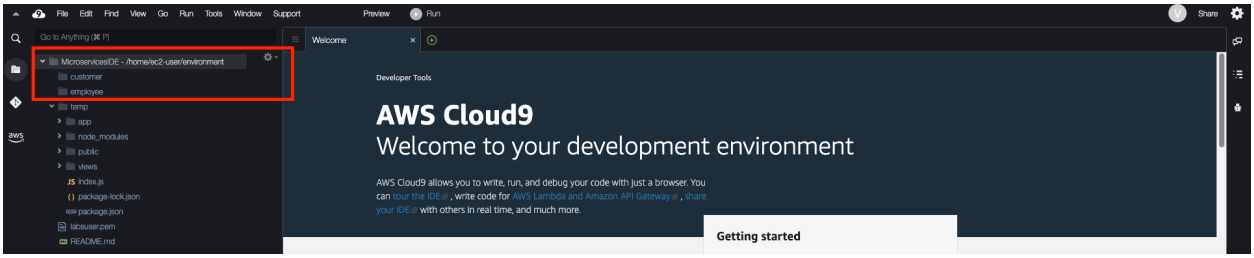
Primary User	Microservice Functionality	Access Level
Customer	The *customer* microservice will provide the functionality that customers (the café franchise location managers who want to buy coffee beans) need. The customers need a read-only view of the contact information for the suppliers to be able to buy coffee beans from them. You can think of the café franchise location managers as *customers* of the application.	Read-only
Employee	The *employee* microservice will provide the functionality that employees (the café corporate office employees) need. Employees need to add, modify, and delete suppliers who are listed in the application. Employees are responsible for keeping the listings accurate and up to date.	Read-write



The employee microservice will eventually be made available only to employees. You will accomplish this by first encapsulating them as a separate microservice (in phases 3 and 4 of the project), and then later in phase 9 of the project, you will limit who can access the employee microservice.

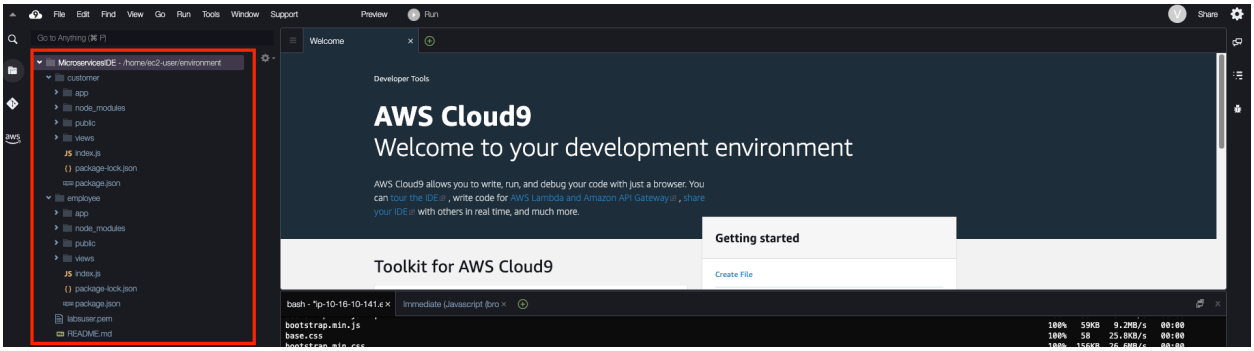
---

1. In the microservices directory, create two new directories that are named customer and employee.



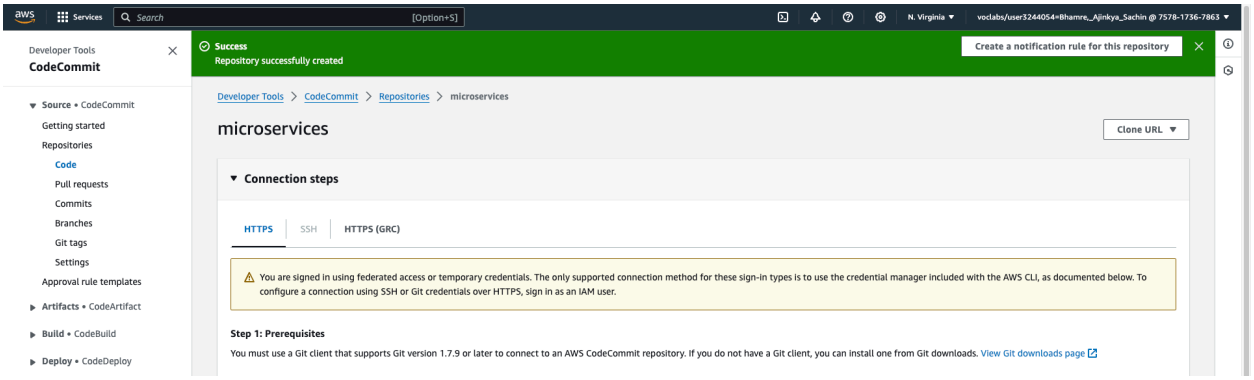
2. Place a copy of the source code for the monolithic application in each new directory, and remove the files from the temp directory.

3. Delete the empty temp directory.



Task 3.4: Create a Git repository for the microservices code and push the code to CodeCommit

1. Create a CodeCommit repository that is named microservices.



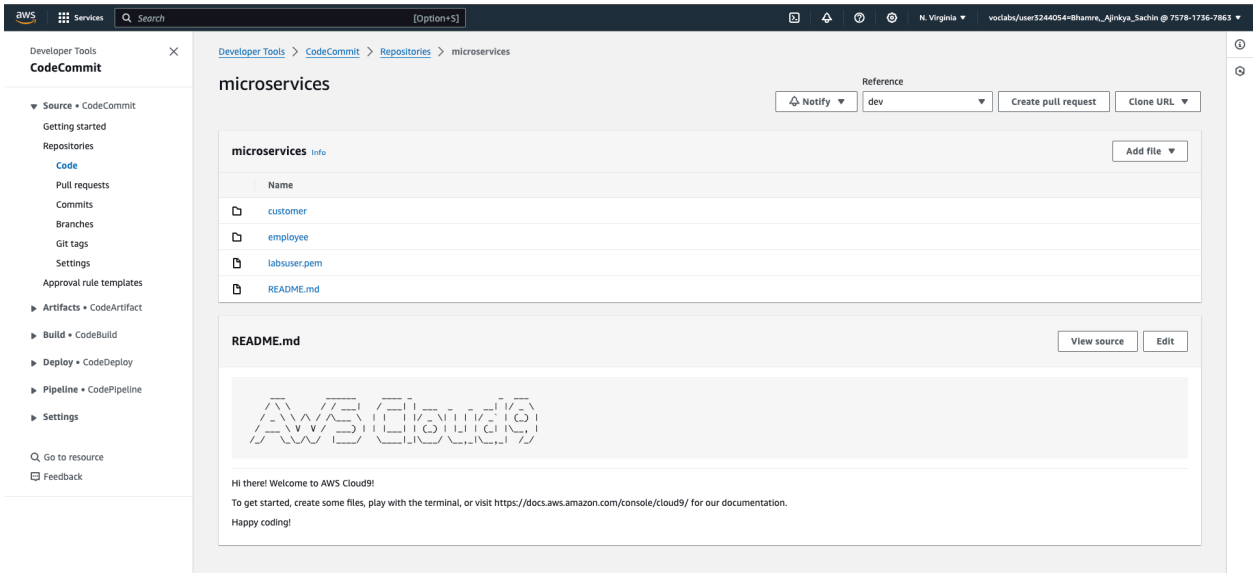
2. To check the unmodified application code into the microservices CodeCommit repository, run the following commands:

```
voclabs:~/environment/microservices (dev) $ git push -u origin dev
Enumerating objects: 2193, done.
Counting objects: 100% (2193/2193), done.
Delta compression using up to 2 threads
Compressing objects: 100% (2085/2085), done.
Writing objects: 100% (2193/2193), 1.94 MiB | 5.28 MiB/s, done.
Total 2193 (delta 550), reused 0 (delta 0), pack-reused 0
remote: Validating objects: 100%
To https://git-codecommit.us-east-1.amazonaws.com/v1/repos/microservices
 * [new branch]      dev -> dev
branch 'dev' set up to track 'origin/dev'.
```

3. Configure your Git client to know your username and email address.

```
voclabs:~/environment/microservices (dev) $ git config --global user.name "Ajinkya Bhamre"
voclabs:~/environment/microservices (dev) $ git config --global user.email "abhamre@stevens.edu"
```

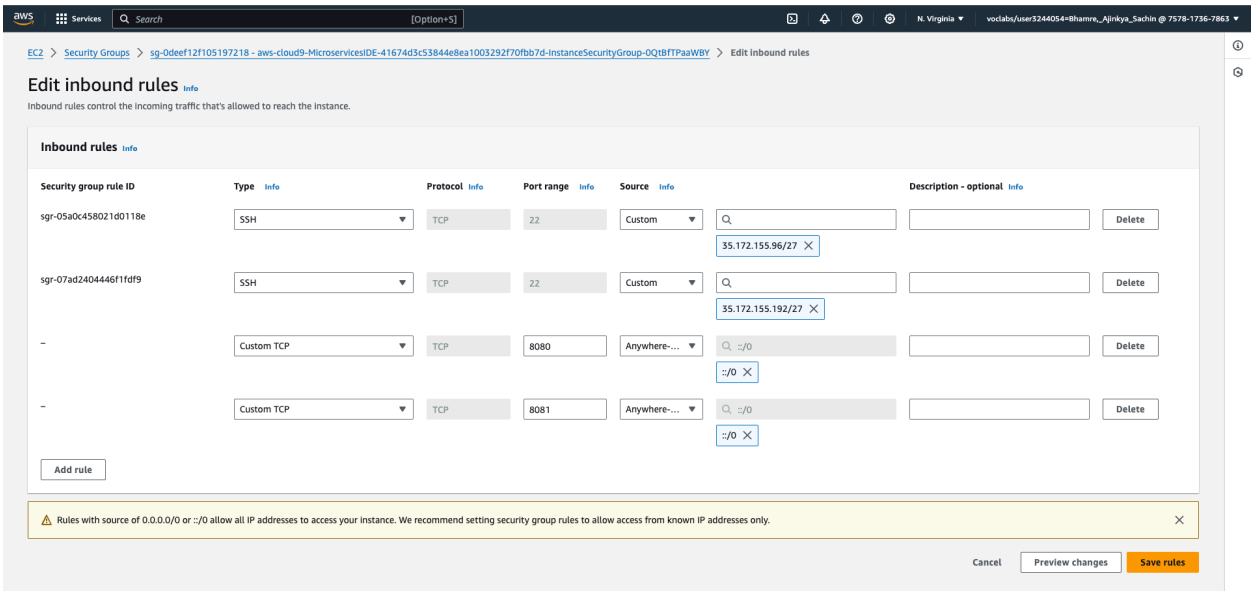
4. In a new browser tab, browse to the CodeCommit console and observe that the code is now checked into your microservices repository.

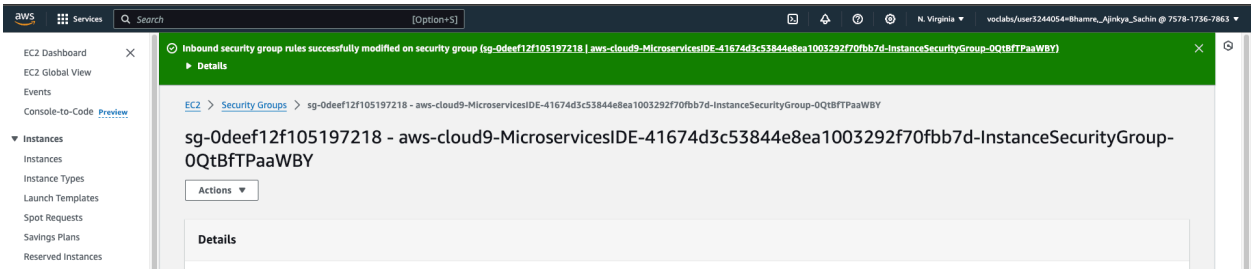


Phase 4: Configuring the application as two microservices and testing them in Docker containers

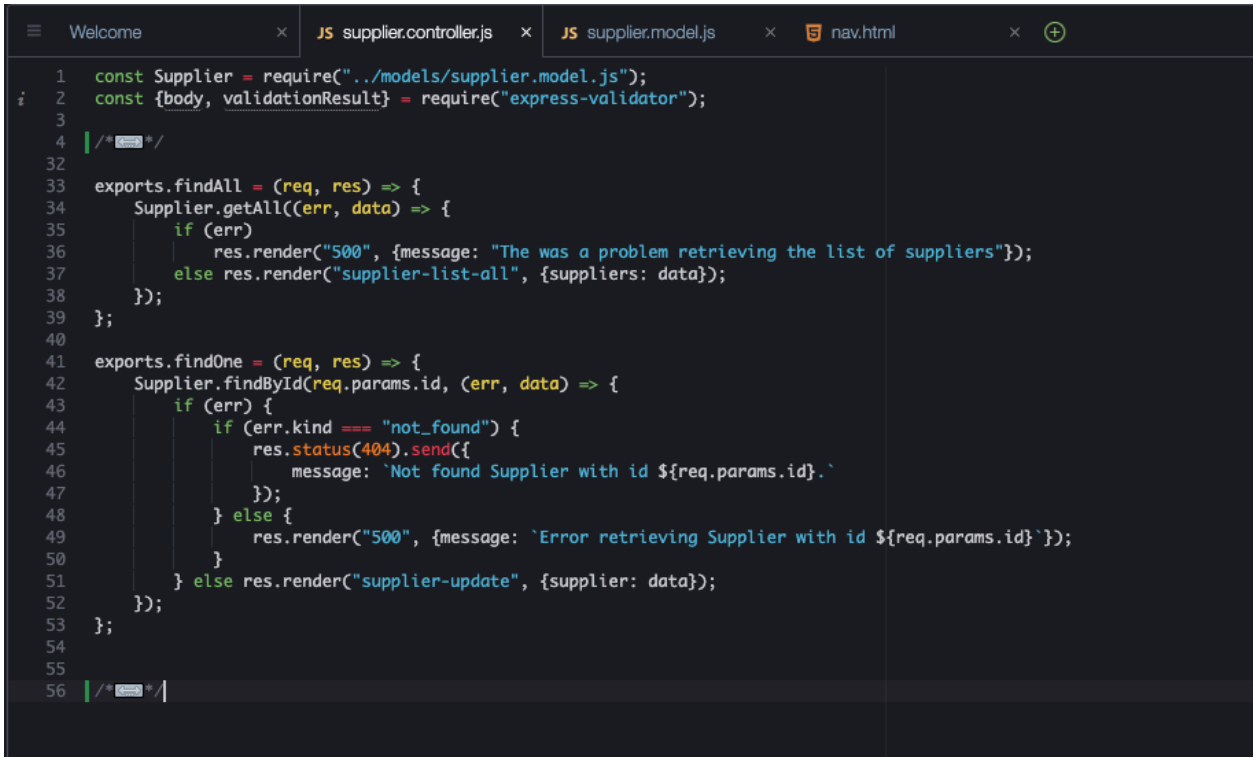
Task 4.1: Adjust the AWS Cloud9 instance security group settings

security group associated with your AWS Cloud9 EC2 instance is configured to allow inbound network traffic on TCP ports 8080 and 8081.





Task 4.2: Modify the source code of the *customer* microservice



Welcome | supplier.controller.js | supplier.model.js | nav.html

```
const mysql = require("mysql2");
const dbConfig = require("../config/config");

const Supplier = function (supplier) {...};

const db_connection = mysql.createPool(...);
/*...*/

Supplier.getAll = result => {
    db_connection.query("SELECT * FROM suppliers", (err, res) => {
        if (err) {
            console.log("error: ", err);
            result(err, null);
            return;
        }
        console.log("suppliers: ", res);
        result(null, res);
    });
};


Supplier.findById = (supplierId, result) => {
    db_connection.query(`SELECT * FROM suppliers WHERE id = ${supplierId}`, (err, res) => {
        if (err) {
            console.log("error: ", err);
            result(err, null);
            return;
        }
        if (res.length) {
            console.log("found supplier: ", res[0]);
            result(null, res[0]);
            return;
        }
        result({kind: "not_found"}, null);
    });
};

/*
Supplier.updateById = (id, supplier, result) => {...};

Supplier.delete = (id, result) => {...};

Supplier.removeAll = result => {...};

*/

module.exports = Supplier;
```

Welcome | supplier.controller.js | supplier.model.js | nav.html

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
    <img src="/img/espresso.jpg" width="200">
    <div><a class="navbar-brand page-title" href="/supplier">Coffee suppliers</a></div>
    <div class="collapse navbar-collapse" id="navbarSupportedContent">
        <ul class="navbar-nav mr-auto">
            <li class="nav-item active">
                <a class="nav-link" href="/">Customer home</a>
                <a class="nav-link" href="/suppliers">Suppliers list</a>
                <a class="nav-link" href="/admin/suppliers">Administrator link</a>
            </li>
        </ul>
    </div>
</nav>
```

Welcome | supplier.controller.js | supplier.model.js | nav.html | supplier-list-all.html

```
{{>header}}
<div class="container">
    {{>nav}}
    <h1>All suppliers</h1>
    <table class="table table-hover">
        <thead>
        <tr>
            <th scope="col">Name</th>
            <th scope="col">Address</th>
            <th scope="col">City</th>
            <th scope="col">State</th>
            <th scope="col">Email</th>
            <th scope="col">Phone</th>
          <!--  <th scope="col"></th> -->
        </tr>
        </thead>
        <tbody>
        {{#suppliers}}
        <tr>
            <th scope="row">{{name}}</th>
            <td>{{address}}</td>
            <td>{{city}}</td>
            <td>{{state}}</td>
            <td>{{email}}</td>
            <td>{{phone}}</td>
            <!--   <td><h4><span class="badge badge-info"><a class="text-light"
                                                            href="/supplier-update/{{id}}">edit</a></span></h4></td>
                -->
        </tr>
        {{/suppliers}}
        </tbody>
    </table>
  <!--  <h4><a class="badge badge-success" href="/supplier-add">Add a new supplier</a></h4> -->

</div>
{{>footer}}

```

Welcome | index.js

```
app.engine("html", mustacheExpress())
app.set("view engine", "html")
app.set("views", __dirname + "/views")
app.use(express.static('public'));
app.use(favicon(__dirname + "/public/img/favicon.ico"));

// list all the suppliers
app.get("/", (req, res) => {
    res.render("home", {});
});
app.get("/suppliers/", supplier.findAll);
// show the add suppler form
//app.get("/supplier-add", (req, res) => {
//    res.render("supplier-add", {});
//});
// receive the add supplier POST
//app.post("/supplier-add", supplier.create);
// show the update form
//app.get("/supplier-update/:id", supplier.findOne);
// receive the update POST
//app.post("/supplier-update", supplier.update);
// receive the POST to delete a supplier
//app.post("/supplier-remove/:id", supplier.remove);
// handle 404
app.use(function (req, res, next) {
    res.status(404).render("404", {});
})


// set port, listen for requests
const app_port = process.env.APP_PORT || 8080
app.listen(app_port, () => {
    console.log(`Server is running on port ${app_port}.`);
});
```

1. Edit the supplier.controller.js file:

- Removed unnecessary functions related to adding, updating, and deleting suppliers.

- Kept only the read-only functions findAll and findOne.

2. Edit the supplier.model.js file:

- Removed unnecessary functions related to adding, updating, and deleting suppliers.

- Kept only the functions getAll and findById.

3. Edit the nav.html file:

- Changed "Monolithic Coffee suppliers" to "Coffee suppliers".

- Changed "Home" to "Customer home".

- Added a new link for administrators: Administrator link.

4. Edit the supplier-list-all.html file:

- Removed the "Add a new supplier" button.

- Removed the edit buttons next to supplier rows.

5. Delete unnecessary HTML files:

- Deleted supplier-add.html, supplier-form-fields.html, and supplier-update.html from the customer/views directory.

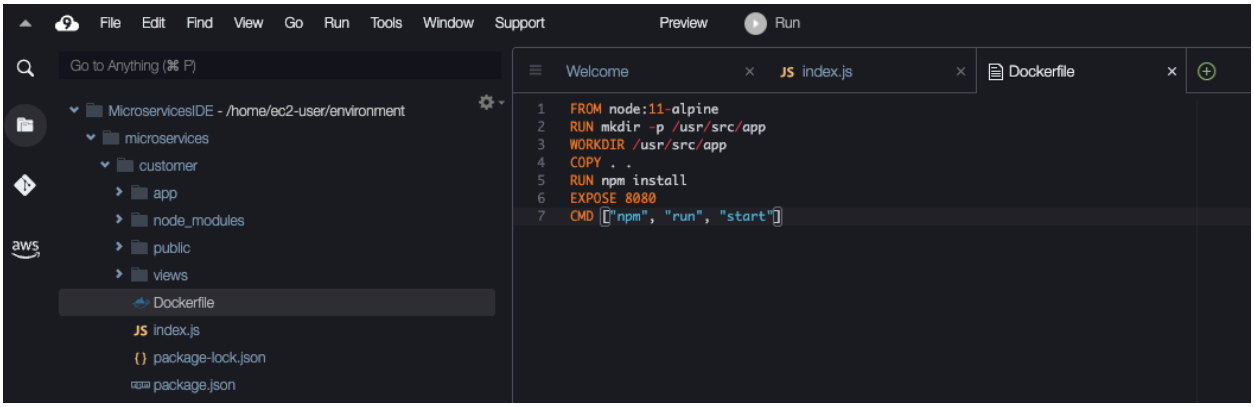
6. Edit the index.js file:

- Commented out unnecessary routes related to adding, updating, and deleting suppliers.

- Changed the port number to 8080 to match the Docker container configuration.

Task 4.3: Create the *customer* microservice Dockerfile and launch a test container

1. In the **customer** directory, create a new file named `Dockerfile` that contains the following code:



Ajin

```
voclabs:~/environment/microservices (dev) $ cd customer
voclabs:~/environment/microservices/customer (dev) $ docker build --tag customer .
Sending build context to Docker daemon  9.008MB
Step 1/7 : FROM node:11-alpine
11-alpine: Pulling from library/node
e7c96db7181b: Pull complete
0119aca44649: Pull complete
40df19605a18: Pull complete
82194b8b4a64: Pull complete
Digest: sha256:8bb56bab197299c8ff820f1a55462890caf08f57ffe3b91f5fa6945a4d505932
Status: Downloaded newer image for node:11-alpine
 ---> f18da2f58c3d
Step 2/7 : RUN mkdir -p /usr/src/app
 ---> Running in dfa84ea13147
Removing intermediate container dfa84ea13147
 ---> 5613e0a99cc6
Step 3/7 : WORKDIR /usr/src/app
 ---> Running in 4d7f50a1ce30
Removing intermediate container 4d7f50a1ce30
 ---> 5977b0cdf719
Step 4/7 : COPY . .
 ---> 7a5300ced89d
Step 5/7 : RUN npm install
 ---> Running in cb2b775cea69
npm WARN coffee_api@1.0.0 No repository field.

audited 78 packages in 0.858s
found 10 vulnerabilities (5 moderate, 3 high, 2 critical)
  run `npm audit fix` to fix them, or `npm audit` for details
Removing intermediate container cb2b775cea69
 ---> 32ff33c2c09b
Step 6/7 : EXPOSE 8080
 ---> Running in 50873c7b933b
Removing intermediate container 50873c7b933b
 ---> 6628e59812cb
Step 7/7 : CMD ["npm", "run", "start"]
 ---> Running in fba762324a8b
Removing intermediate container fba762324a8b
 ---> a3bf4ccb12df
Successfully built a3bf4ccb12df
Successfully tagged customer:latest
```

The Docker build process has completed successfully. Here's a breakdown of what happened:

1. The Docker daemon pulled the Node.js 11 Alpine image from the Docker Hub repository.

2. A directory named /usr/src/app was created inside the container.

3. The working directory was set to /usr/src/app.

4. The current directory contents were copied into the container's working directory.

5. The npm install command was executed to install dependencies for the Node.js application.

6. The container was instructed to expose port 8080.

7. Finally, the container was configured to run the application using the command npm run start. The Docker image with the tag customer:latest has been successfully built.

3. Verify that the customer-labeled Docker image was created.

docker images

```
voclabs:~/environment/microservices/customer (dev) $ docker images
REPOSITORY   TAG          IMAGE ID       CREATED          SIZE
customer     latest       a3bf4ccb12df   15 minutes ago   82.7MB
node         11-alpine    f18da2f58c3d   4 years ago      75.5MB
```

i. Set the dbEndpoint variable using the provided command:

dbEndpoint=$(cat ~/environment/microservices/customer/app/config/config.js | grep 'APP_DB_HOST' | cut -d '"' -f2)

ii. Verify that the dbEndpoint variable is set correctly by running:

```
voclabs:~/environment/microservices/customer (dev) $ dbEndpoint=$(cat ~/environment/microservices/customer/app/config/config.js | grep 'APP_DB_HOST' | cut -d '"' -f2)
voclabs:~/environment/microservices/customer (dev) $ echo $dbEndpoint
supplierdb.crrzlncupqd3.us-east-1.rds.amazonaws.com
```

iii. If the dbEndpoint variable is set correctly, you can launch the Docker container using the following command

docker run -d --name customer_1 -p 8080:8080 -e APP_DB_HOST="$dbEndpoint" customer

```
voclabs:~/environment/microservices/customer (dev) $ docker run -d --name customer_1 -p 8080:8080 -e APP_DB_HOST="$dbEndpoint" customer
45eda30c632ec2395c7eb84123a75542a6c304e079b67b367f176ddf82b6a4d5
voclabs:~/environment/microservices/customer (dev) $
```

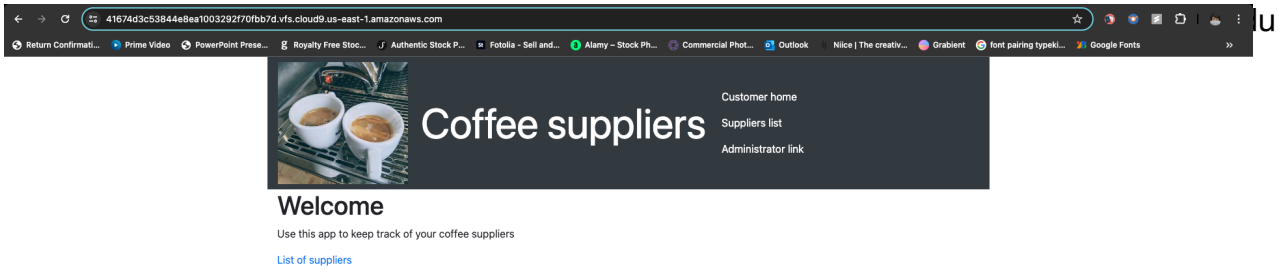
This command will launch a Docker container named customer_1 from the customer image, expose port 8080 on AWS Cloud9 instance, and pass the database host location as an environment variable to the container. The container will run in the background (-d flag).

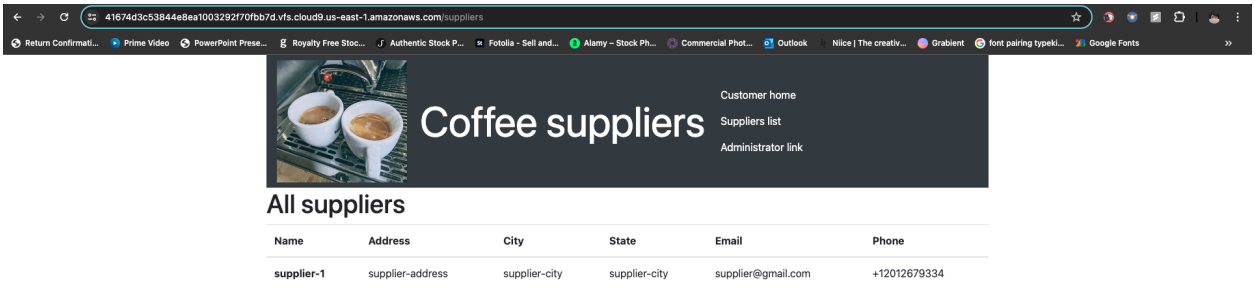
5. Check which Docker containers are currently running on the AWS Cloud9 instance.

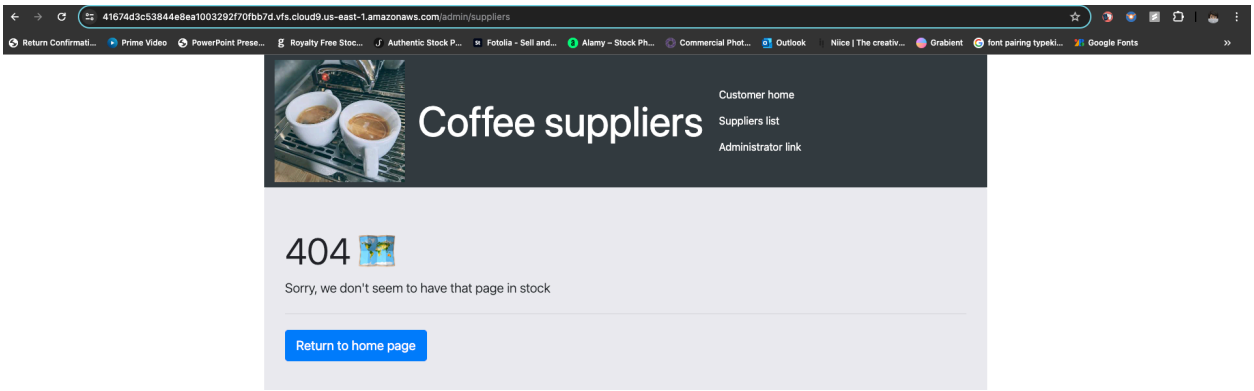
docker ps

```
voclabs:~/environment/microservices/customer (dev) $ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED         STATUS         PORTS                                       NAMES
45eda30c632e   customer   "docker-entrypoint.s…"   5 minutes ago   Up 5 minutes   0.0.0.0:8080->8080/tcp, :::8080->8080/tcp   customer_1
voclabs:~/environment/microservices/customer (dev) $
```

6. Verify that the customer microservice is running in the container and working as intended.







7. Commit and push your source code changes into CodeCommit.

8. Optional: Observe the commit details in the CodeCommit console.

```
voclabs:~/environment/microservices/customer (dev) $ git log
commit 340b8d3f7954f586629466265678437d82a479c8 (HEAD -> dev)
Author: Ajinkya Bhamre <abhamre@stevens.edu>
Date:   Fri Apr 26 22:36:55 2024 +0000

    customer-container-hosted

commit f33356aa82e6bc27741d19f496e5856f7c27d439 (origin/dev)
Author: EC2 Default User <ec2-user@ip-10-16-10-141.ec2.internal>
Date:   Fri Apr 26 07:34:52 2024 +0000

    two unmodified copies of the application code
voclabs:~/environment/microservices/customer (dev) $ git push origin dev
Enumerating objects: 24, done.
Counting objects: 100% (24/24), done.
Delta compression using up to 2 threads
Compressing objects: 100% (12/12), done.
Writing objects: 100% (13/13), 1.35 KiB | 1.35 MiB/s, done.
Total 13 (delta 7), reused 0 (delta 0), pack-reused 0
remote: Validating objects: 100%
To https://git-codecommit.us-east-1.amazonaws.com/v1/repos/microservices
   f33356a..340b8d3  dev -> dev
```

Task 4.4: Modify the source code of the *employee* microservice

1. supplier.controller.js:

Prepend "/admin" to all redirect calls.

2. index.js:

Prepend "/admin" to all app.get and app.post calls.

Change the port number to 8081.

3. supplier-add.html and supplier-update.html:

Prepend "/admin" to the form action paths.

4. supplier-list-all.html and home.html:

Prepend "/admin" to the HTML paths.

5. header.html:

Modify the title to "Manage coffee suppliers”.

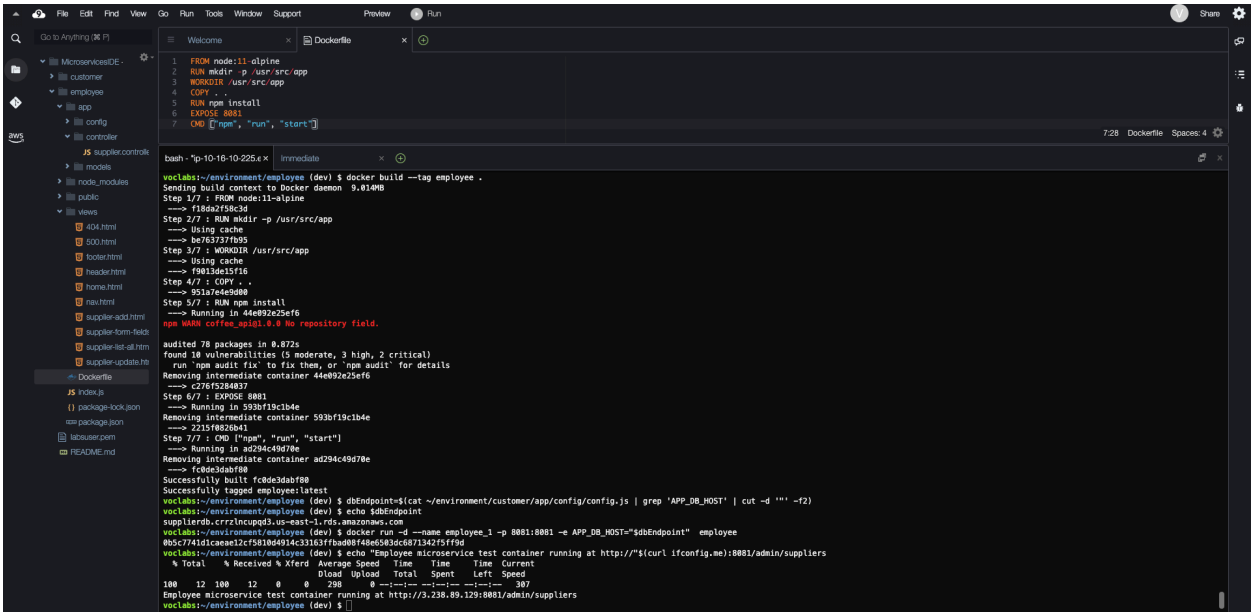
6. nav.html:

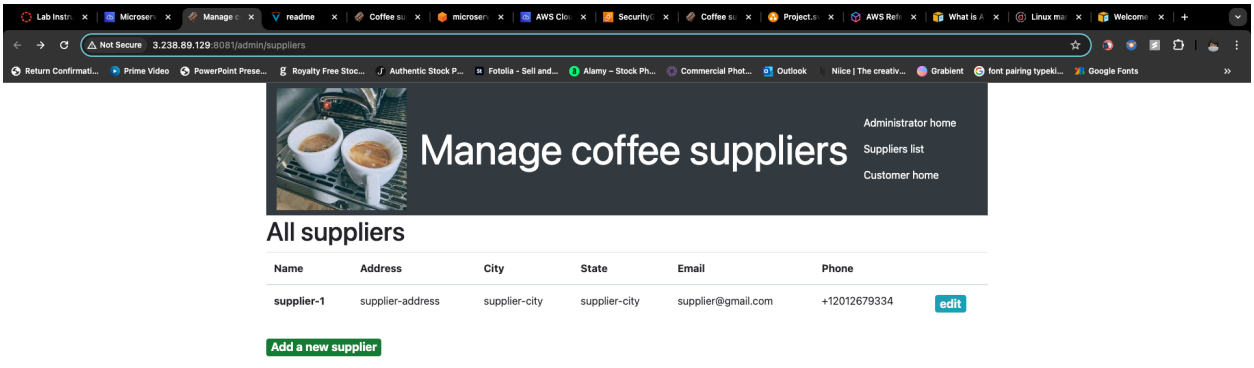
Change "Monolithic Coffee suppliers" to "Manage coffee suppliers" in the navbar title.

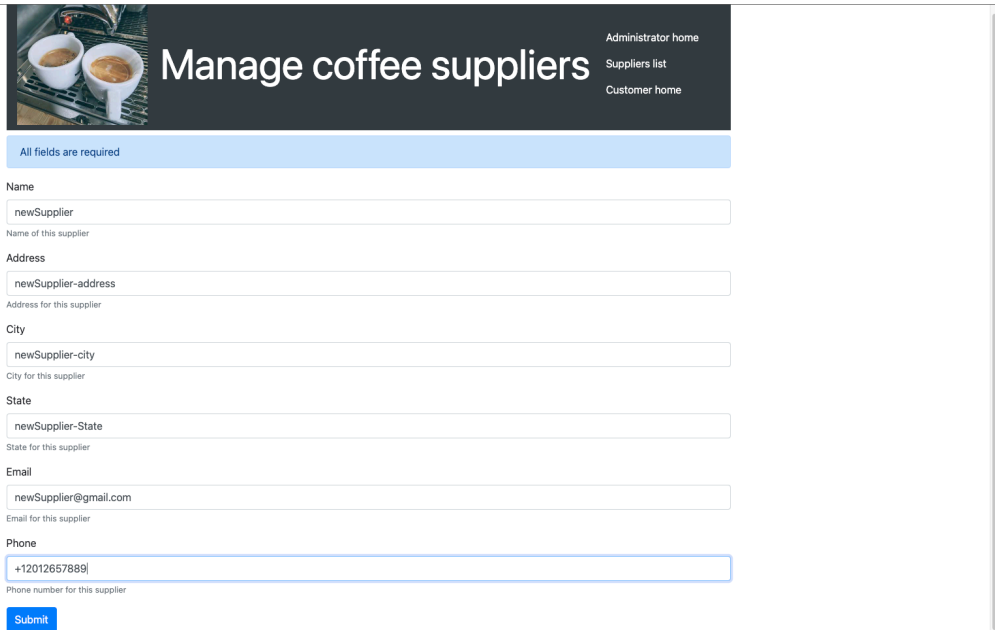
Replace the existing line of code with Administrator home.

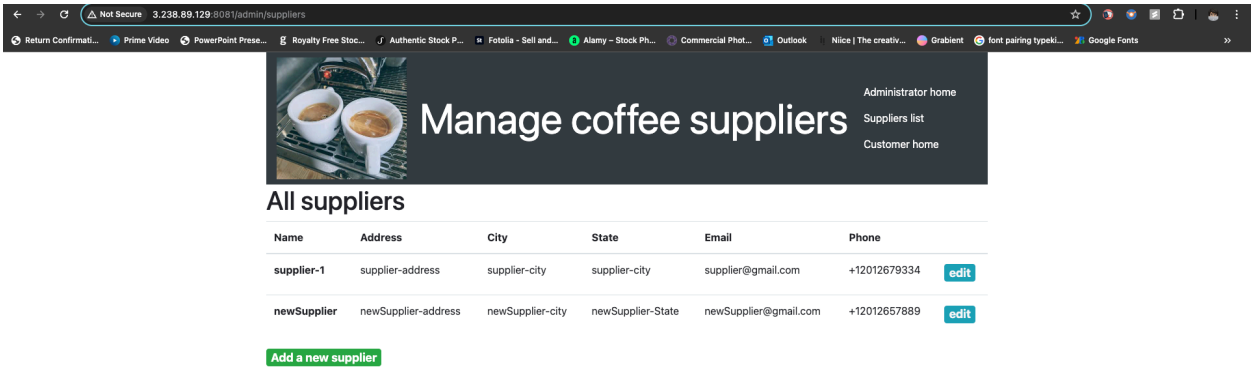
Add a new line with Customer home after line 8.

Task 4.5: Create the *employee* microservice Dockerfile and launch a test container



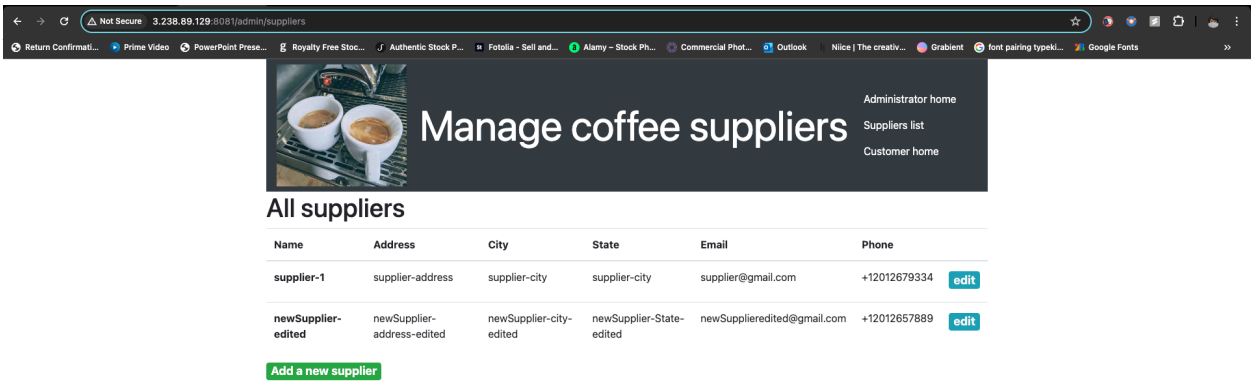




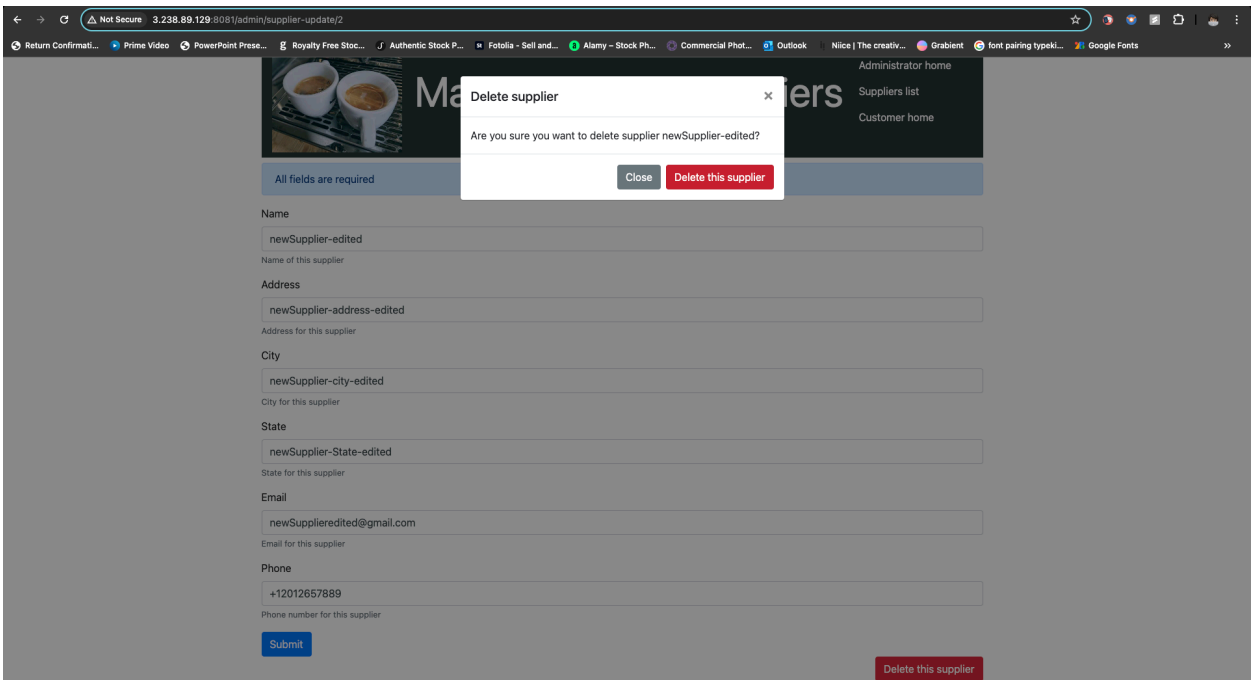


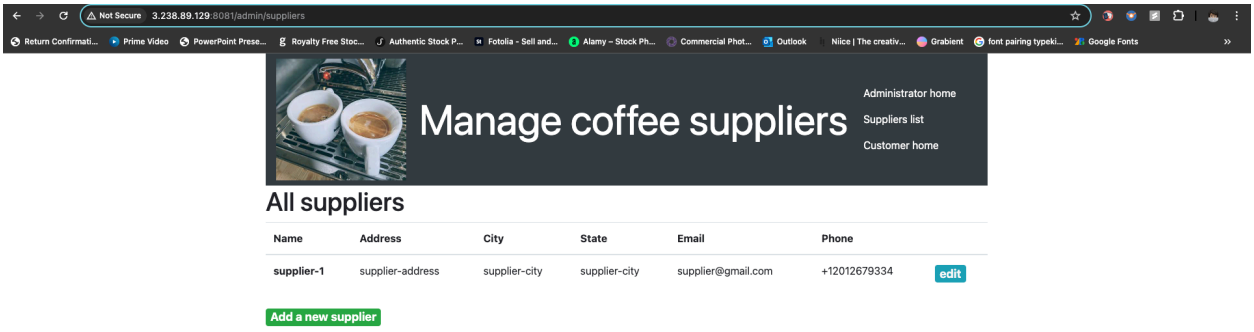
Links that should take to the customer microservice will not work. For example, if you choose Customer home or Suppliers list, the pages won't be found because the link assumes that the customer microservice also runs on port 8081 (but it doesn't). You can ignore this issue—these links should work as intended when you deploy the microservices to Amazon ECS later.

Test editing an existing supplier.



Test deleting an existing supplier:

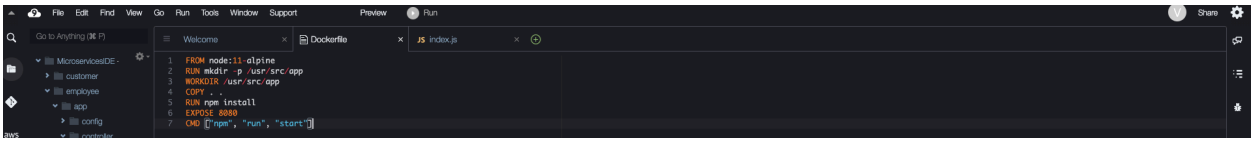


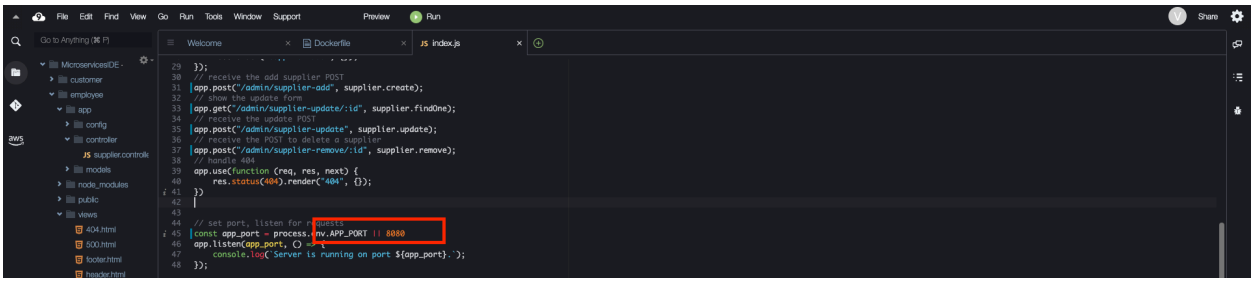


Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu

Task 4.6: Adjust the *employee* microservice port and rebuild the image

1. Edit the employee/index.js and employee/Dockerfile files to change the port from 8081 to 8080
2. Rebuild the Docker image for the employee microservice.






```
voclabs:~/environment/employee (dev) $ docker build --tag employee .
Sending build context to Docker daemon  9.014MB
Step 1/7 : FROM node:11-alpine
 ---> f18da2f58c3d
Step 2/7 : RUN mkdir -p /usr/src/app
 ---> Using cache
 ---> be763737fb95
Step 3/7 : WORKDIR /usr/src/app
 ---> Using cache
 ---> f9013de15f16
Step 4/7 : COPY . .
 ---> fec59d2b0729
Step 5/7 : RUN npm install
 ---> Running in eff62845b51e
npm WARN coffee_api@1.0.0 No repository field.

audited 78 packages in 0.7s
found 10 vulnerabilities (5 moderate, 3 high, 2 critical)
  run `npm audit fix` to fix them, or `npm audit` for details
Removing intermediate container eff62845b51e
 ---> ddd3b9c9446a
Step 6/7 : EXPOSE 8080
 ---> Running in 61758935e2c1
Removing intermediate container 61758935e2c1
 ---> e8b5eb74b089
Step 7/7 : CMD ["npm", "run", "start"]
 ---> Running in 039faeeb1c65
Removing intermediate container 039faeeb1c65
 ---> 06808d35c3e8
Successfully built 06808d35c3e8
Successfully tagged employee:latest
```

```
voclabs:~/environment/employee (dev) $ cd ..
voclabs:~/environment (dev) $ git add .
voclabs:~/environment (dev) $ git commit -m "emp-docker-index-changed"
[dev eb429fa] emp-docker-index-changed
 25 files changed, 44 insertions(+), 21 deletions(-)
 create mode 100644 .c9/metadata/environment/customer/Dockerfile
 create mode 100644 .c9/metadata/environment/customer/app/controller/supplier.controller.js
 create mode 100644 .c9/metadata/environment/customer/app/models/supplier.model.js
 create mode 100644 .c9/metadata/environment/customer/index.js
 create mode 100644 .c9/metadata/environment/customer/views/nav.html
 create mode 100644 .c9/metadata/environment/customer/views/supplier-list-all.html
 create mode 100644 .c9/metadata/environment/employee/Dockerfile
 create mode 100644 .c9/metadata/environment/employee/app/controller/supplier.controller.js
 create mode 100644 .c9/metadata/environment/employee/index.js
 create mode 100644 .c9/metadata/environment/employee/views/header.html
 create mode 100644 .c9/metadata/environment/employee/views/home.html
 create mode 100644 .c9/metadata/environment/employee/views/nav.html
 create mode 100644 .c9/metadata/environment/employee/views/supplier-add.html
 create mode 100644 .c9/metadata/environment/employee/views/supplier-list-all.html
 create mode 100644 .c9/metadata/environment/employee/views/supplier-update.html
 create mode 100644 employee/Dockerfile
voclabs:~/environment (dev) $ git push origin dev
Enumerating objects: 50, done.
Counting objects: 100% (50/50), done.
Delta compression using up to 2 threads
Compressing objects: 100% (41/41), done.
Writing objects: 100% (43/43), 10.35 KiB | 2.07 MiB/s, done.
Total 43 (delta 15), reused 0 (delta 0), pack-reused 0
remote: Validating objects: 100%
To https://git-codecommit.us-east-1.amazonaws.com/v1/repos/microservices
   ffaa69a..eb429fa  dev -> dev
```

Task 4.7: Check code into CodeCommit

Phase 5: Creating ECR repositories, an ECS cluster, task definitions, and AppSpec files

Task 5.1: Create ECR repositories and upload the Docker images

1. To authorize your Docker client to connect to the Amazon ECR service, run the following commands:

```
account_id=$(aws sts get-caller-identity |grep Account|cut -d '"' -f4)
echo $account_id
```

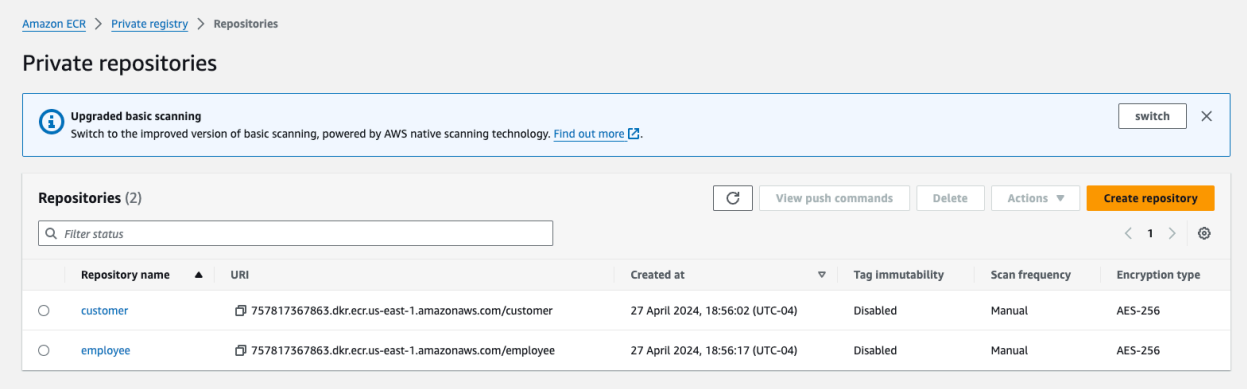
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin $account_id.dkr.ecr.us-east-1.amazonaws.com

2. Create a separate private ECR repository for each microservice.

3. Set permissions on the customer ECR repository.

4. Use the same approach to set the same permissions on the employee ECR repository.

5. Tag the Docker images with your unique registryId (account ID) value to make it



easier to manage and keep track of these images.

```
voclabs:~/environment (dev) $ account_id=$(aws sts get-caller-identity |grep Account|cut -d '"' -f4)
voclabs:~/environment (dev) $ echo $account_id
757817367863
voclabs:~/environment (dev) $ docker tag customer:latest $account_id.dkr.ecr.us-east-1.amazonaws.com/customer:latest
voclabs:~/environment (dev) $ docker tag employee:latest $account_id.dkr.ecr.us-east-1.amazonaws.com/employee:latest
voclabs:~/environment (dev) $ docker images
REPOSITORY                                                TAG         IMAGE ID       CREATED       SIZE
757817367863.dkr.ecr.us-east-1.amazonaws.com/employee     latest      06808d35c3e8   2 hours ago   82.7MB
employee                                                  latest      06808d35c3e8   2 hours ago   82.7MB
757817367863.dkr.ecr.us-east-1.amazonaws.com/customer     latest      dc7b5c34ee66   3 hours ago   82.7MB
customer                                                  latest      dc7b5c34ee66   3 hours ago   82.7MB
node                                                      11-alpine   f18da2f58c3d   4 years ago   75.5MB
```

6. Run the appropriate docker command to push each of the Docker images to Amazon ECR.

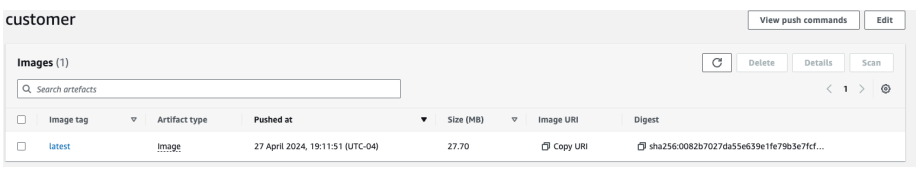
 # Set account_id variable

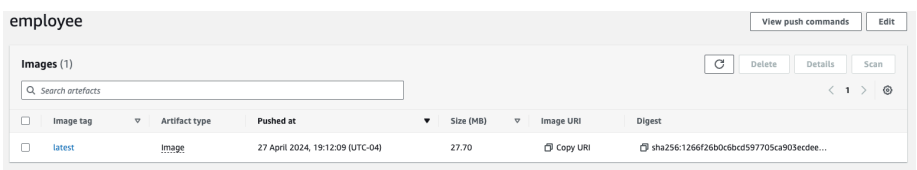
account_id=$(aws sts get-caller-identity | grep Account | cut -d '"' -f4)

 # Push customer image to Amazon ECR

docker push $account_id.dkr.ecr.us-east-1.amazonaws.com/customer:latest

```
voclabs:~/environment (dev) $ account_id=$(aws sts get-caller-identity | grep Account | cut -d '"' -f4)
voclabs:~/environment (dev) $ docker push $account_id.dkr.ecr.us-east-1.amazonaws.com/customer:latest
The push refers to repository [757817367863.dkr.ecr.us-east-1.amazonaws.com/customer]
7dbc3c445ef4: Pushed
5dd1c0aeceb5: Pushed
49777b716811: Pushed
d81d715330b7: Pushed
1dc7f3bb09a4: Pushed
dcaceb729824: Pushed
f1b5933fe4b5: Pushed
latest: digest: sha256:0082b7027da55e639e1fe79b3e7fcf824d29f9669ac4df74af9b9cacbd2c15e5 size: 1783
voclabs:~/environment (dev) $ docker push $account_id.dkr.ecr.us-east-1.amazonaws.com/employee:latest
The push refers to repository [757817367863.dkr.ecr.us-east-1.amazonaws.com/employee]
ebe5c42bc739: Pushed
df7ba9d67f17: Pushed
49777b716811: Pushed
d81d715330b7: Pushed
1dc7f3bb09a4: Pushed
dcaceb729824: Pushed
f1b5933fe4b5: Pushed
latest: digest: sha256:1266f26b0c6bcd597705ca903ecdee863110fc5186bad1cb9de23ab63096ed7b size: 1783
```



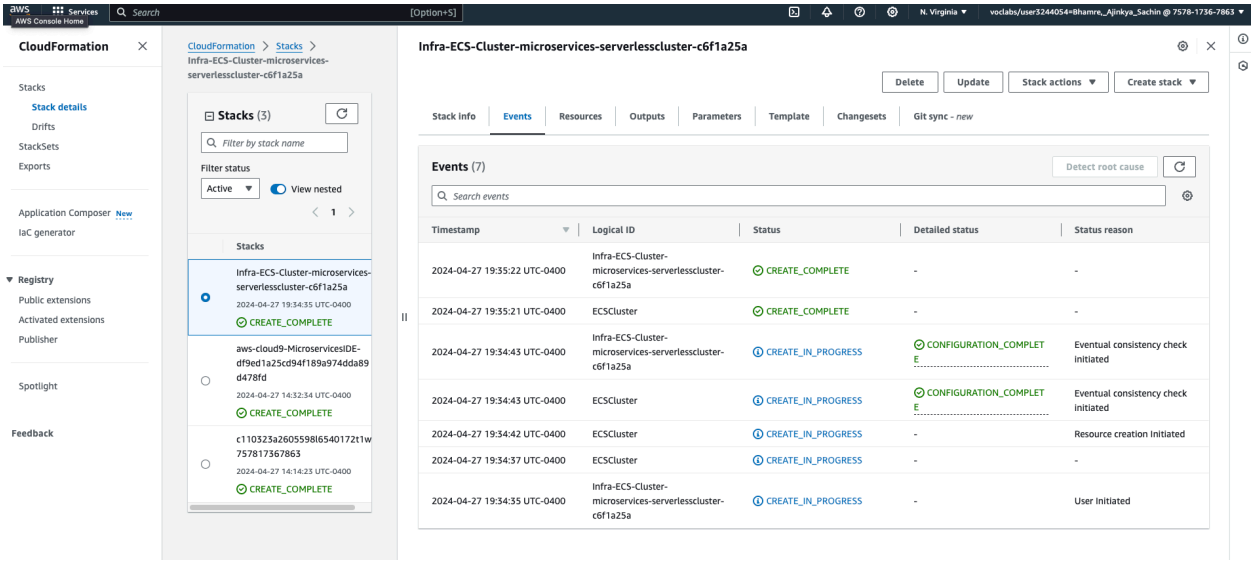


 # Push employee image to Amazon ECR

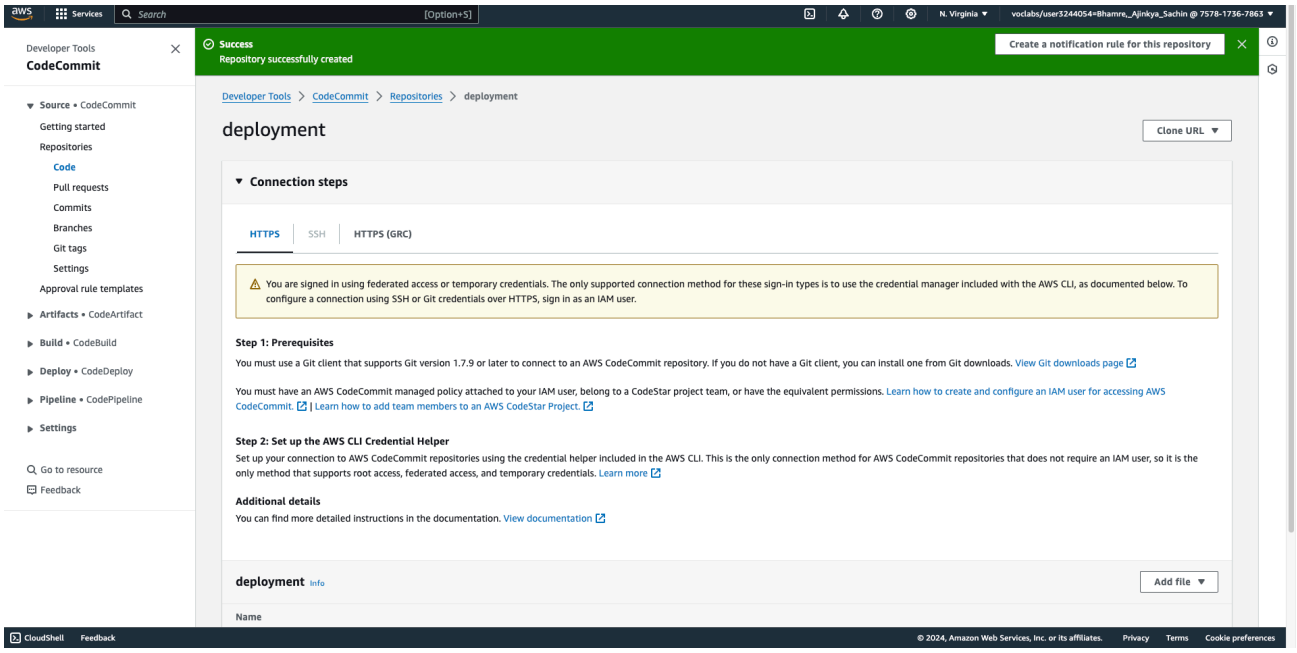
docker push $account_id.dkr.ecr.us-east-1.amazonaws.com/employee:latest

7. Confirm that the two

images are now stored in Amazon ECR and that each has the latest label applied.



Task 5.2: Create an ECS cluster



```
voclabs:~/environment (dev) $ mkdir deployment
voclabs:~/environment (dev) $ cd deployment
voclabs:~/environment/deployment (dev) $ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/ec2-user/environment/deployment/.git/
voclabs:~/environment/deployment (master) $ git checkout -b dev
Switched to a new branch 'dev'
```

Task 5.3: Create a CodeCommit repository to store deployment file

Now you have a CodeCommit repository named "deployment" initialized with a branch named "dev". You can use this repository to store deployment configuration files for Amazon ECS and AppSpec specification files for CodeDeploy.

Task 5.4: Create task definition files for each microservice and register them with Amazon ECS

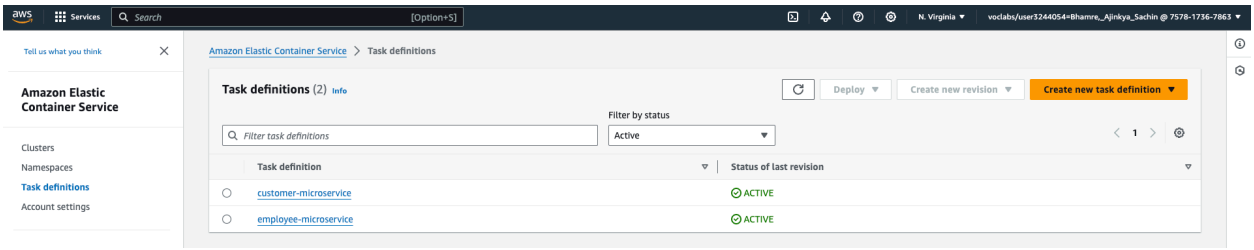
1. In the new deployment directory, create an empty file named taskdef-customer.json

2. Edit the taskdef-customer.json file.

3 .To register the customer microservice task definition in Amazon ECS, run the following command:

$ aws ecs register-task-definition --cli-input-json “file:///home/ec2-user/environment/deployment/taskdef-customer.json"

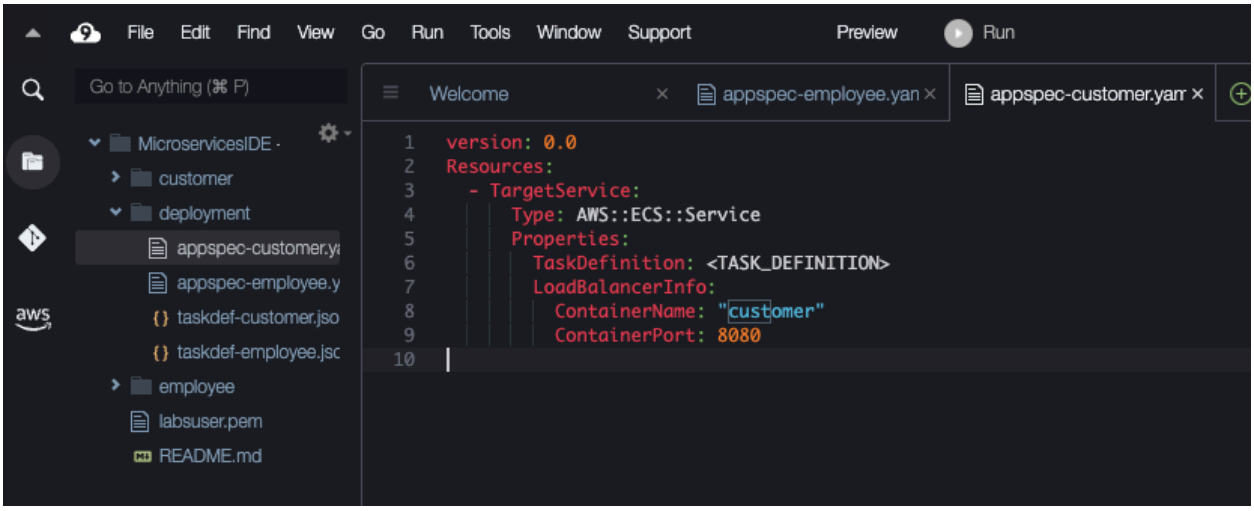

```
{
    "taskDefinition": {
        "taskDefinitionArn": "arn:aws:ecs:us-east-1:757817367863:task-definition/customer-microservice:2",
        "containerDefinitions": [
            {
                "name": "customer",
                "image": "customer",
                "cpu": 0,
                "portMappings": [
                    {
                        "containerPort": 8080,
                        "hostPort": 8080,
                        "protocol": "tcp"
                    }
                ],
                "essential": true,
                "environment": [
                    {
                        "name": "APP_DB_HOST",
                        "value": "supplierdb.crrzlncupqd3.us-east-1.rds.amazonaws.com"
                    }
                ],
                "mountPoints": [],
```

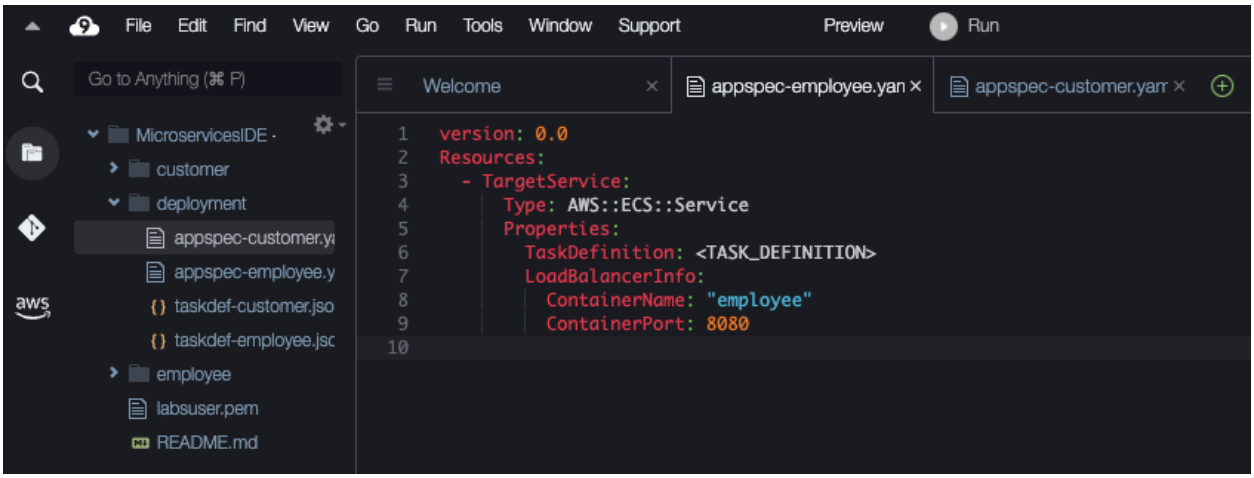


Task 5.5: Create AppSpec files for CodeDeploy for each microservice

1.Create an AppSpec file for the customer microservice.

2. In the same directory, create an AppSpec file for the employee microservice.



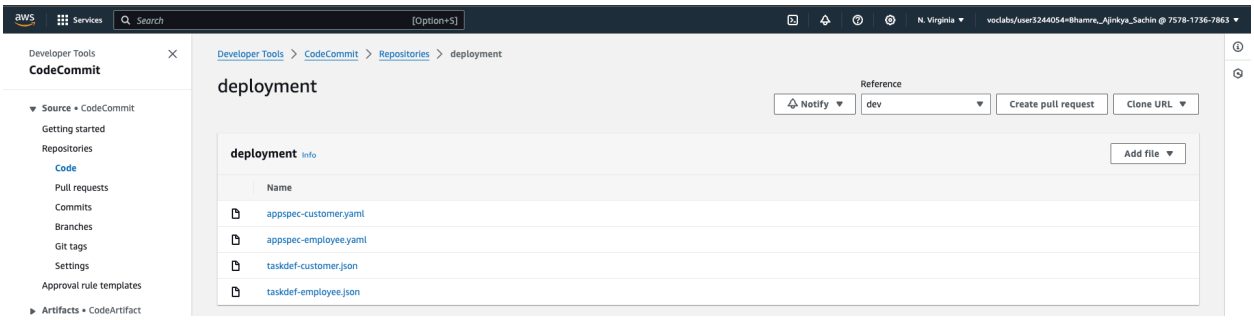


Task 5.6: Update files and check them into CodeCommit

1. Edit the taskdef-customer.json file.
2. Edit the taskdef-employee.json file.
3. Push all four files to CodeCommit.

```
voclabs:~/environment/deployment (dev) $ git add .
voclabs:~/environment/deployment (dev) $ git commit -m "Updated task definition files and added AppSpec files"
[dev (root-commit) 381aff8] Updated task definition files and added AppSpec files
 4 files changed, 97 insertions(+)
 create mode 100644 appspec-customer.yaml
 create mode 100644 appspec-employee.yaml
 create mode 100644 taskdef-customer.json
 create mode 100644 taskdef-employee.json
```

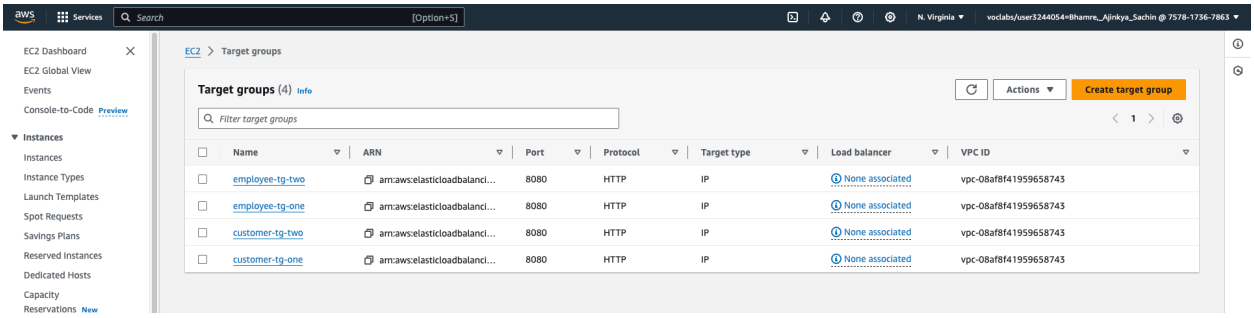
```
voclabs:~/environment/deployment (dev) $ git remote add origin https://git-codecommit.us-east-1.amazonaws.com/v1/repos/deployment
voclabs:~/environment/deployment (dev) $ git push -u origin dev
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 2 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 1.07 KiB | 1.07 MiB/s, done.
Total 6 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Validating objects: 100%
To https://git-codecommit.us-east-1.amazonaws.com/v1/repos/deployment
 * [new branch]      dev -> dev
branch 'dev' set up to track 'origin/dev'.
```



Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu

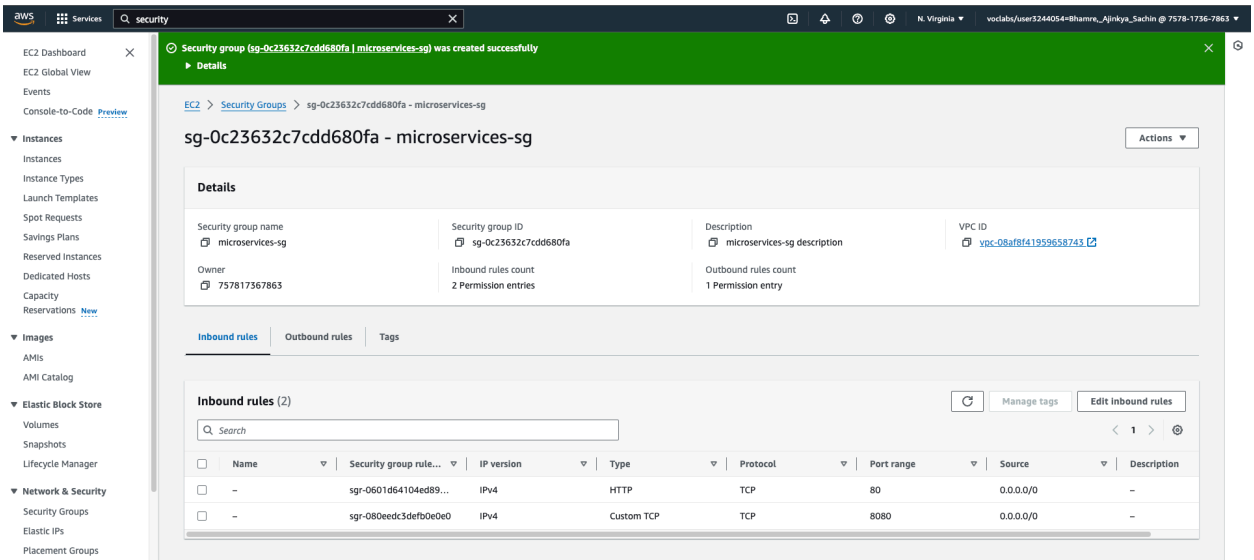
Phase 6: Creating target groups and an Application Load Balancer

1. Create the first target group for the customer microservice.
2. Create a second target group for the customer microservice. Use the same settings as the first target group except use customer-tg-two as the target group name.
3. Create a target group for the employee microservice.
4. Create a second target group for the employee microservice.



Task 6.2: Create a security group and an Application Load Balancer, and configure rules to route traffic

1. Create a new EC2 security group:



Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu

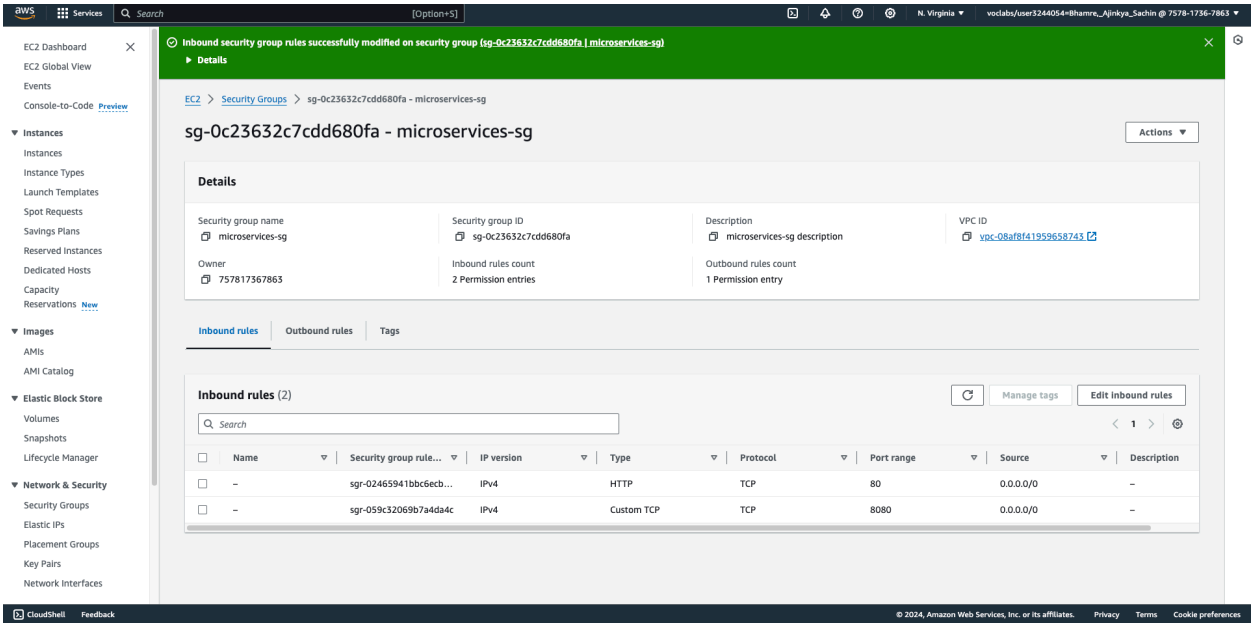
2. Create an Application Load Balancer:

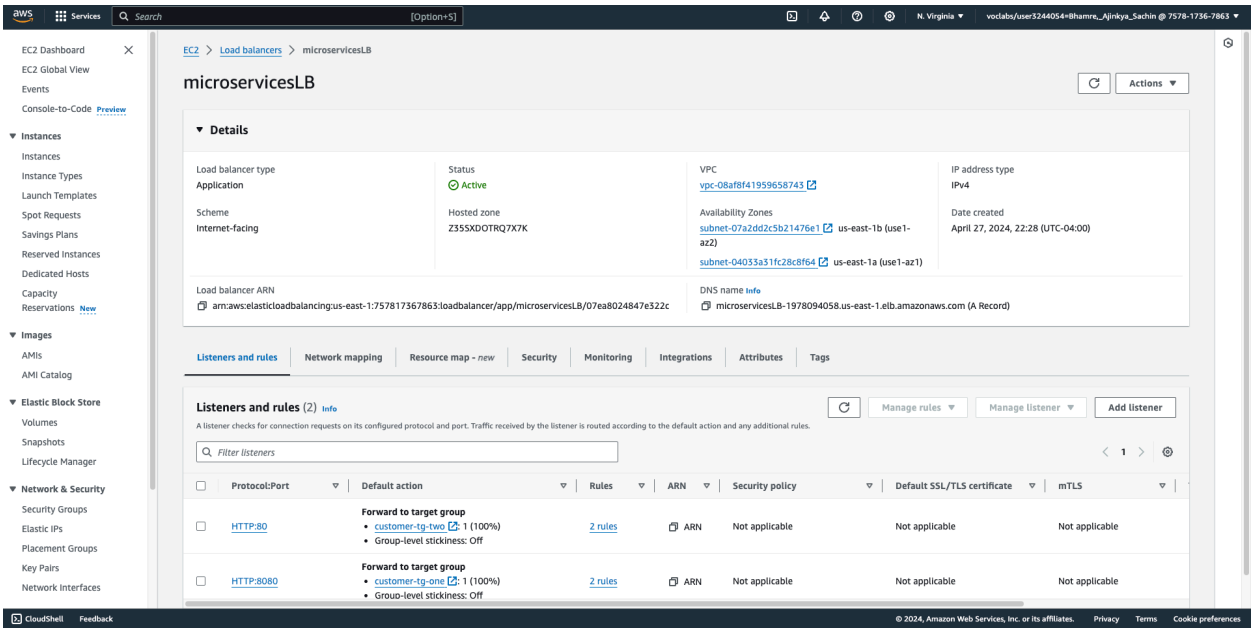
1. Create a new EC2 security group named microservices-sg to use in LabVPC. Add inbound rules that allow TCP traffic from any IPv4 address on ports 80 and 8080.

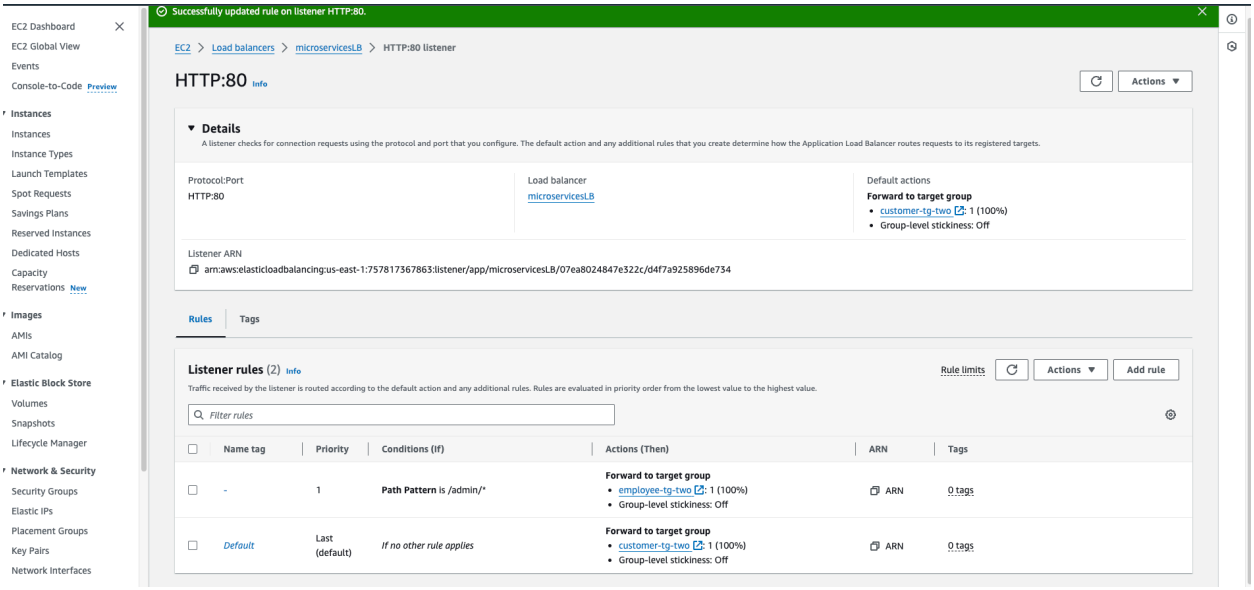
2. In the Amazon EC2 console, create an Application Load Balancer named microservicesLB.

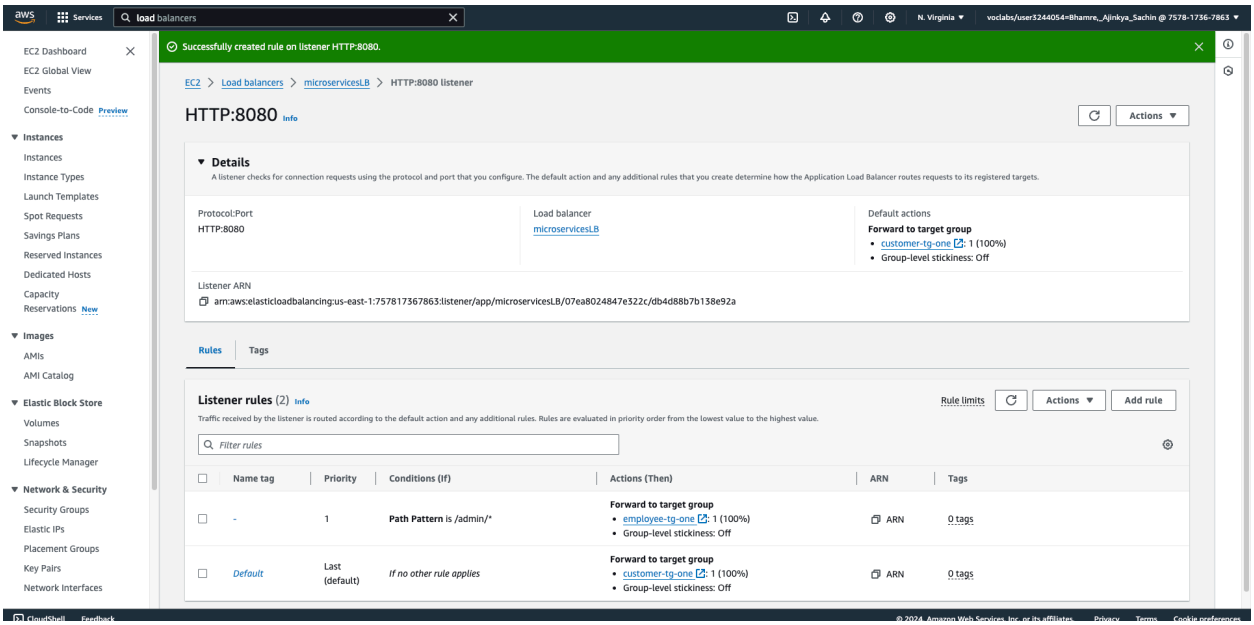
3.Add a second rule for the HTTP:80 listener.

4. Add a second rule for the HTTP:8080 listener.



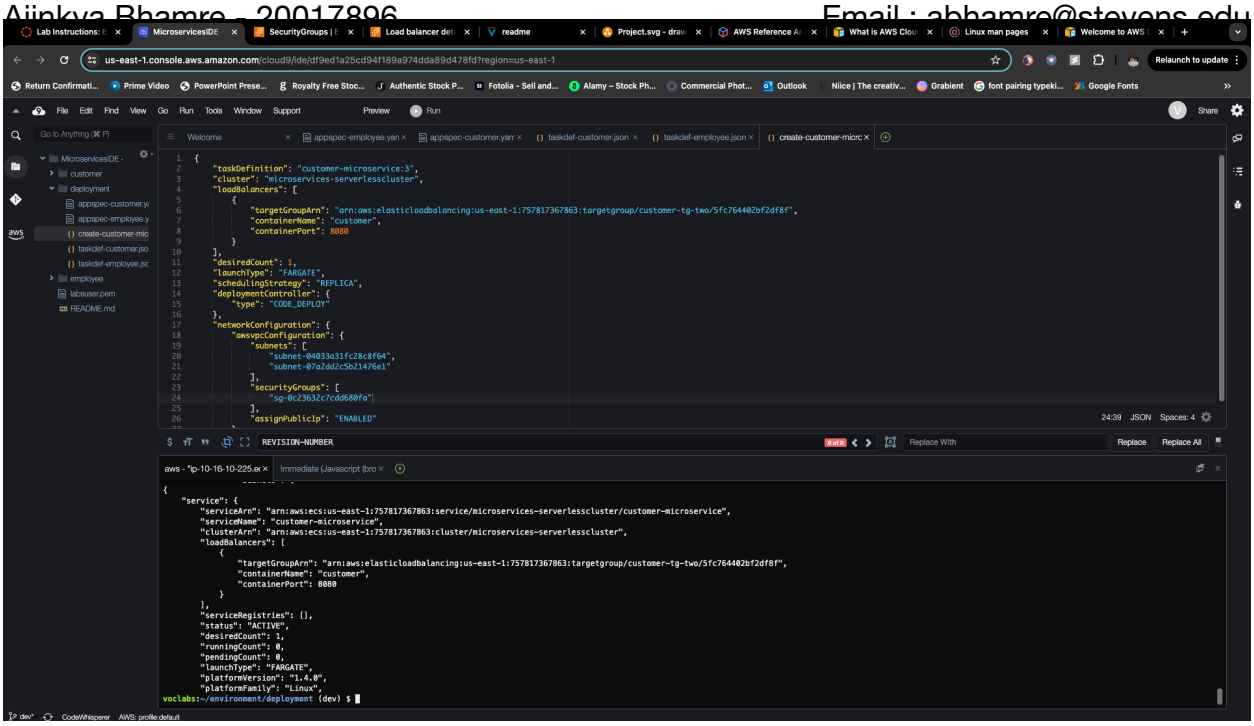






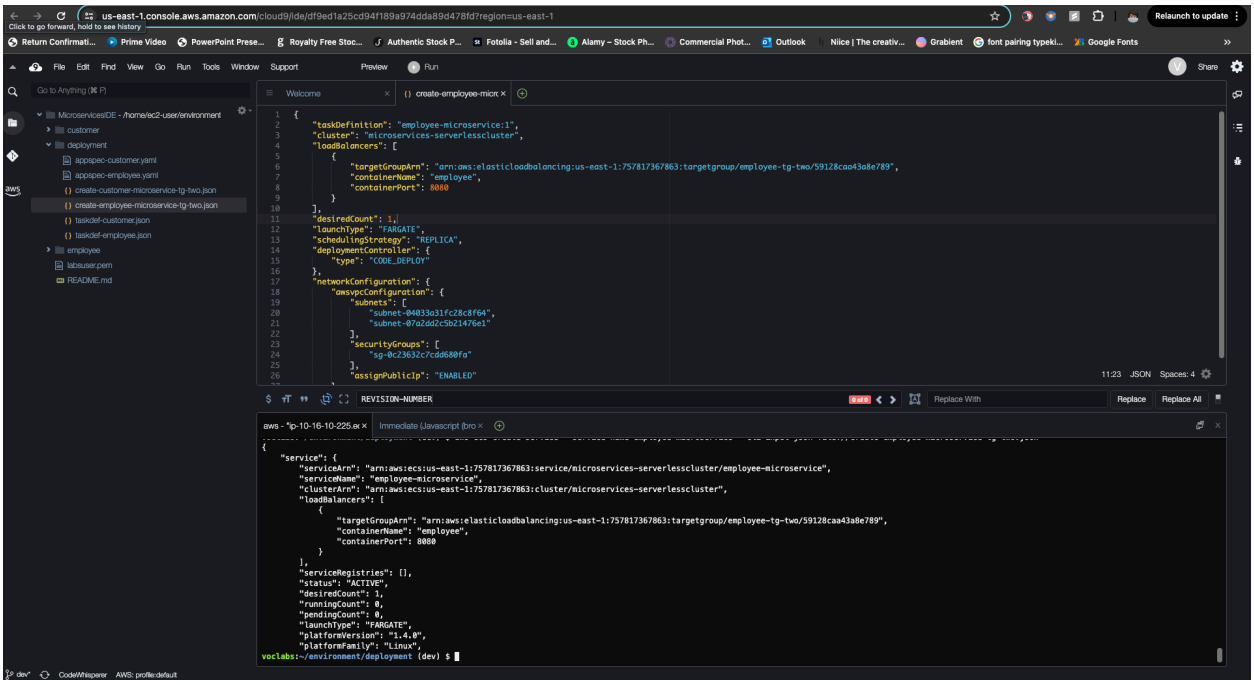
Phase 7: Creating two Amazon ECS services

1. In AWS Cloud9, create a new file named create-customer-microservice-tg-two.json in the deployment directory.
2. Pasted the provided JSON code into the file
3. Edited the create-customer-microservice-tg-two.json file
4. created the Amazon ECS service for the customer microservice

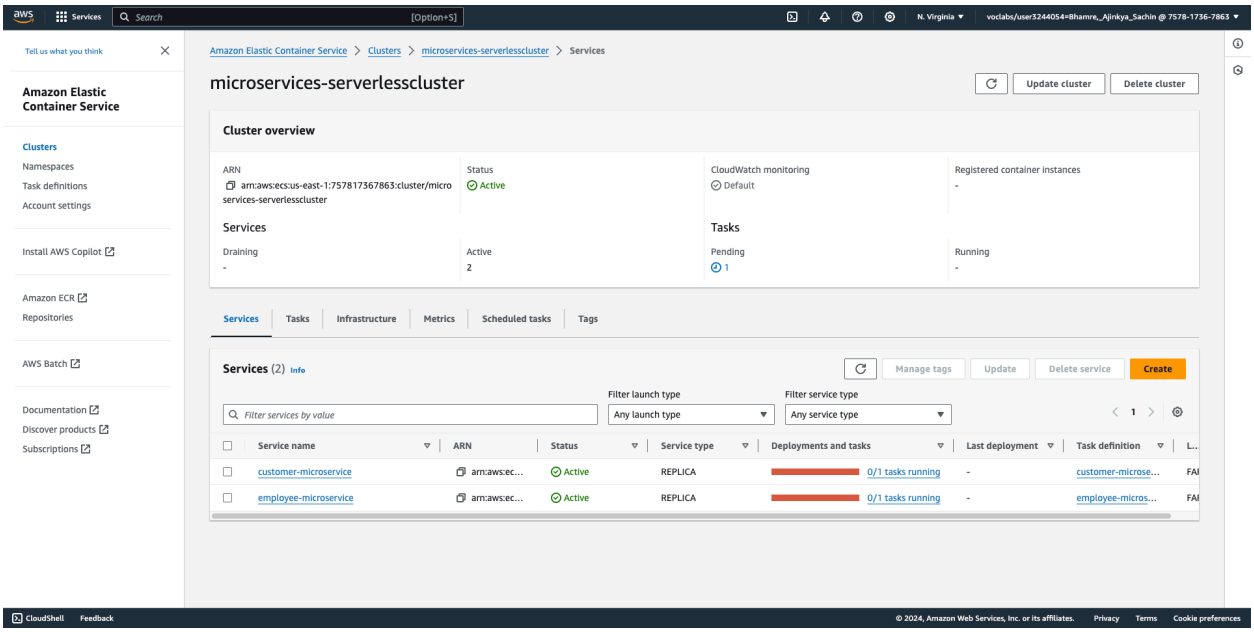


Task 7.2: Create the Amazon ECS service for the *employee* microservice

1. Created an Amazon ECS service for the employee microservice.
2. Ran the appropriate AWS CLI command to create the service in Amazon ECS.

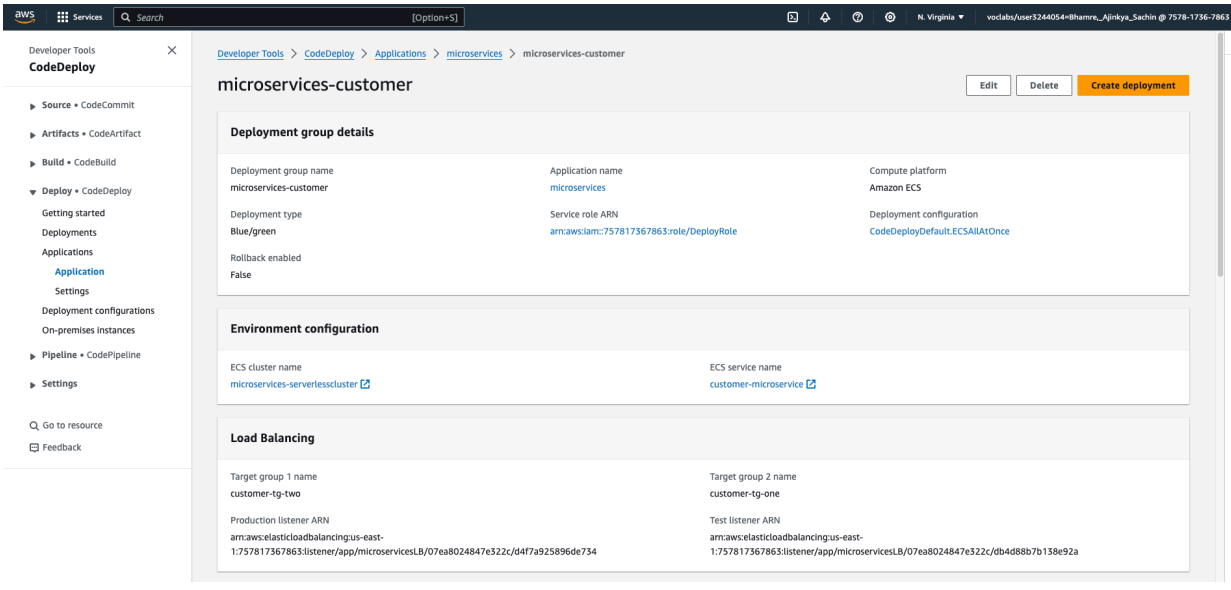


Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu

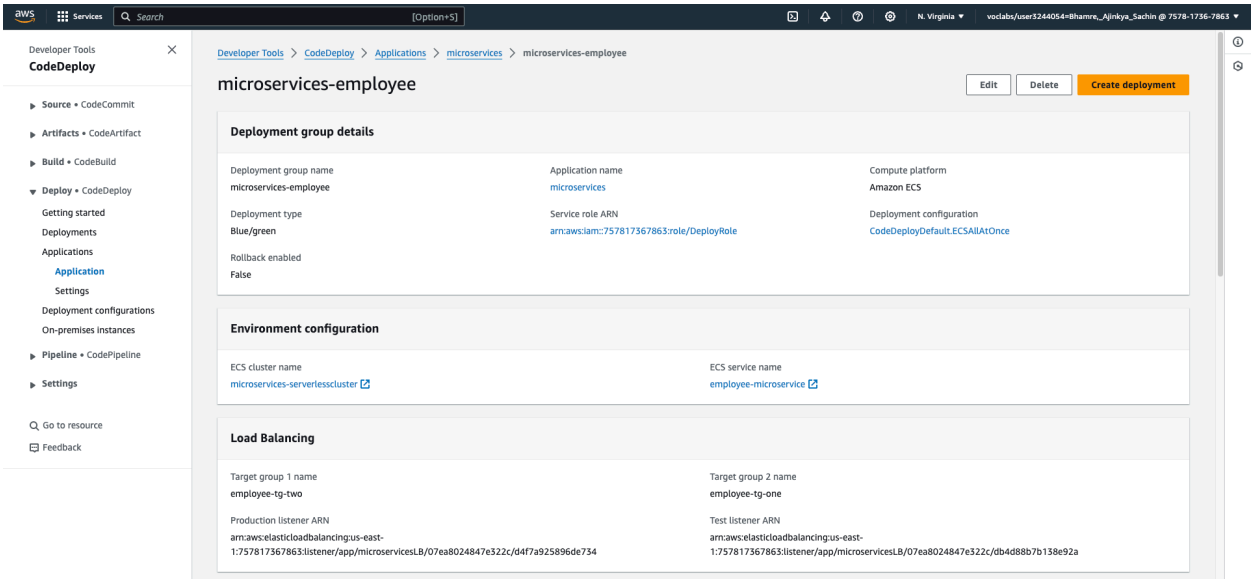


Phase 8: Configuring CodeDeploy and CodePipeline

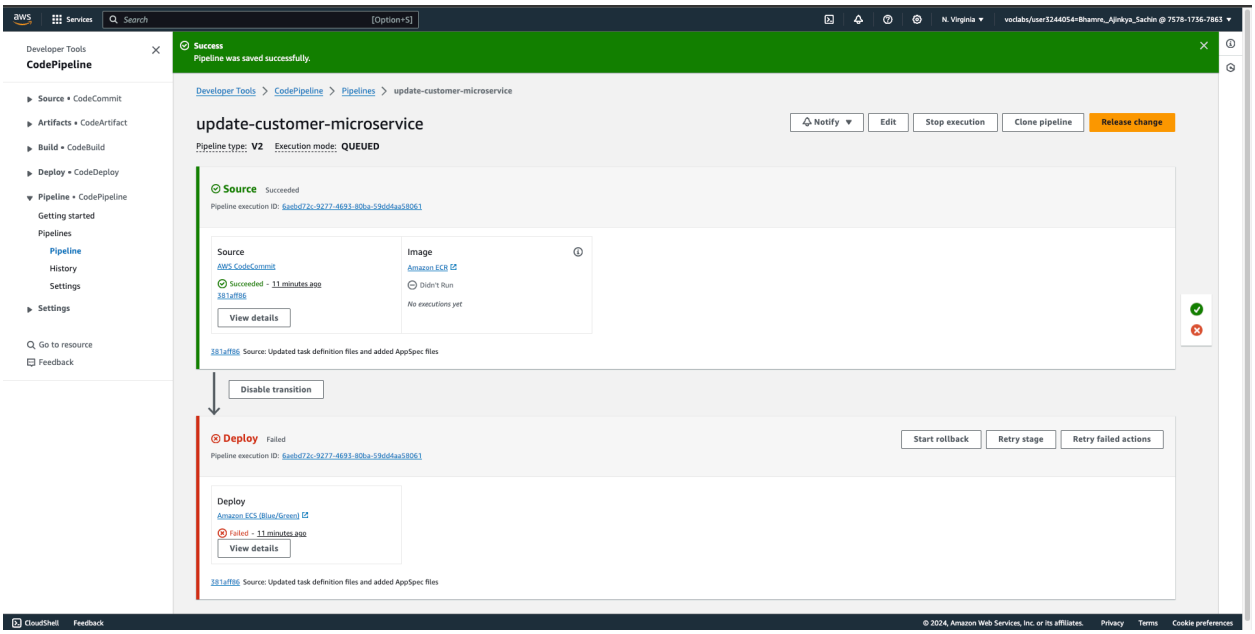
Task 8.1: Create a CodeDeploy application and deployment groups



Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu



Task 8.2: Create a pipeline for the *customer* microservice



Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu

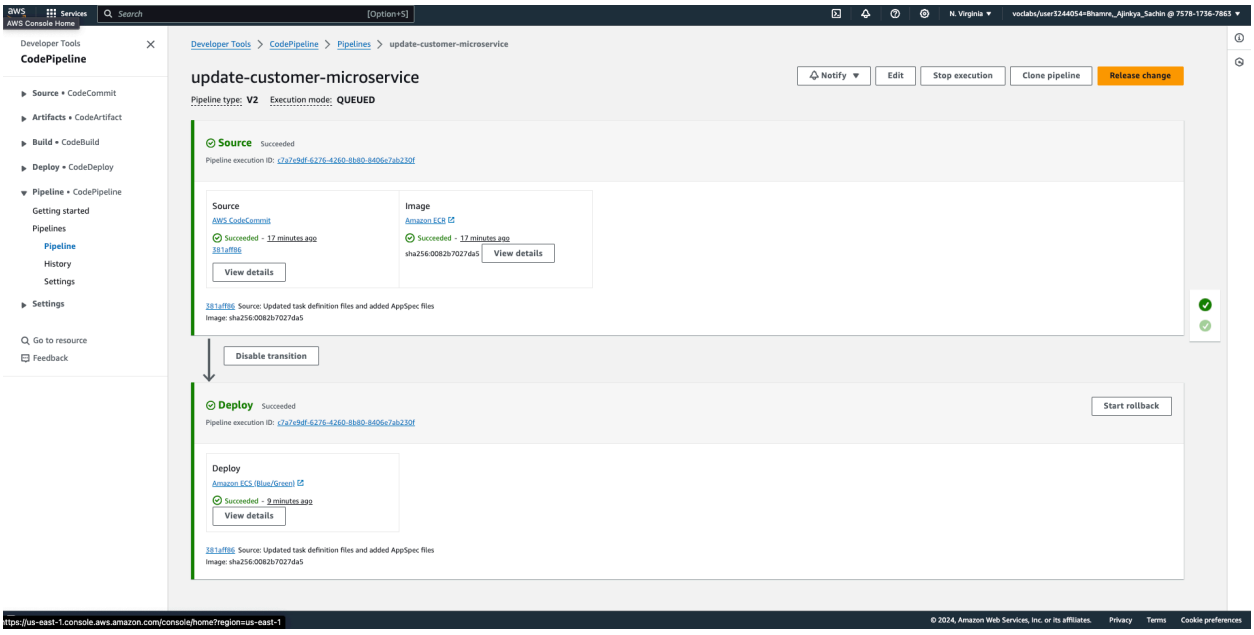
Task 8.3: Test the CI/CD pipeline for the *customer* microservice

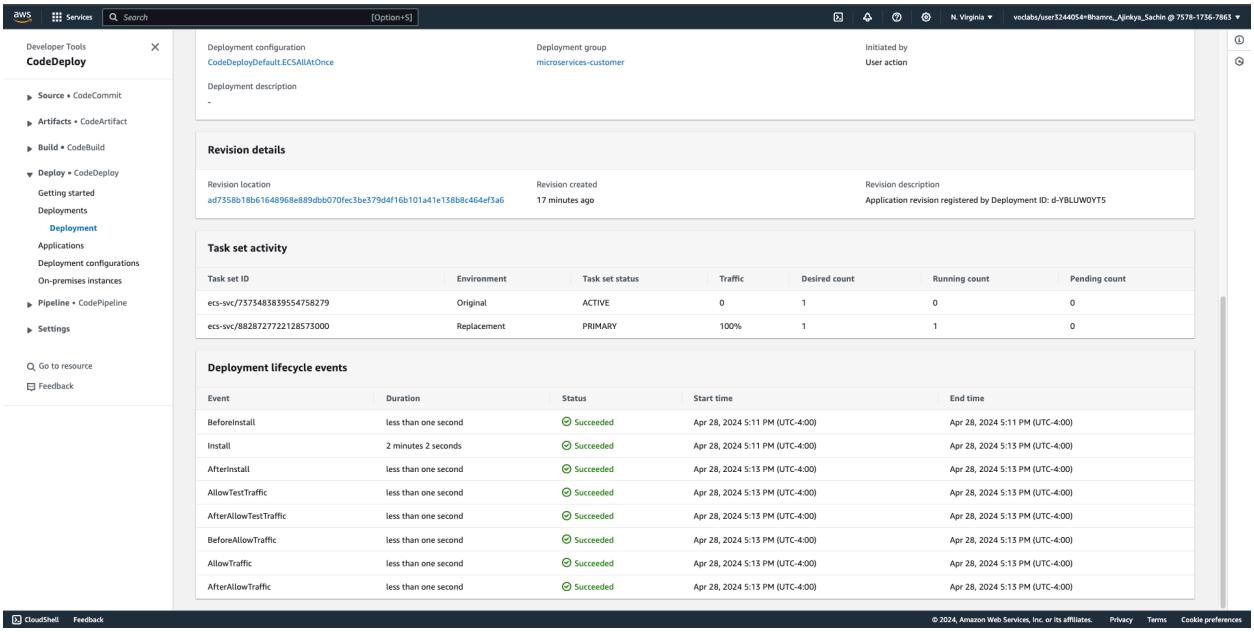
1.Launched a Deployment

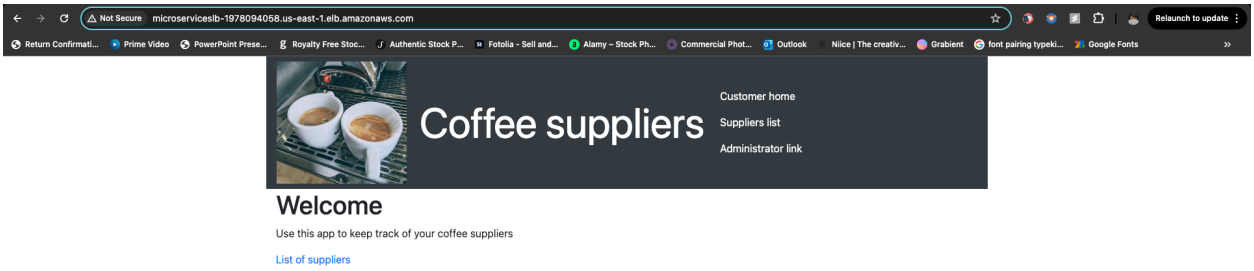
2. Observe Progress in CodeDeploy:

3. Test the Customer Microservice

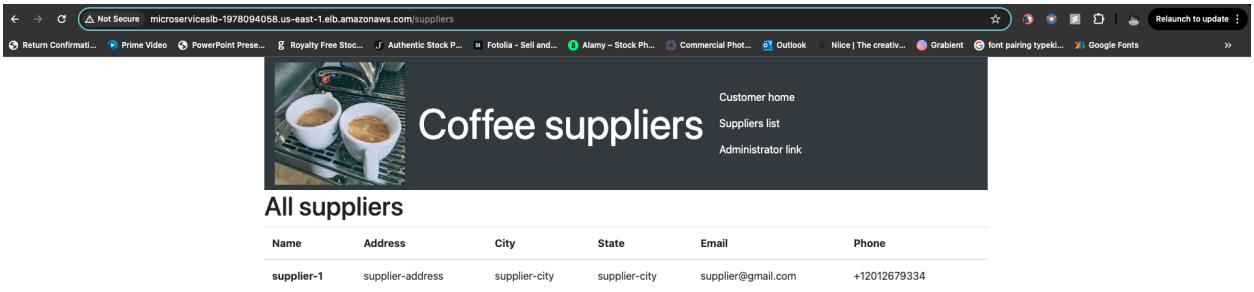
4. Observe Running Tasks in Amazon ECS

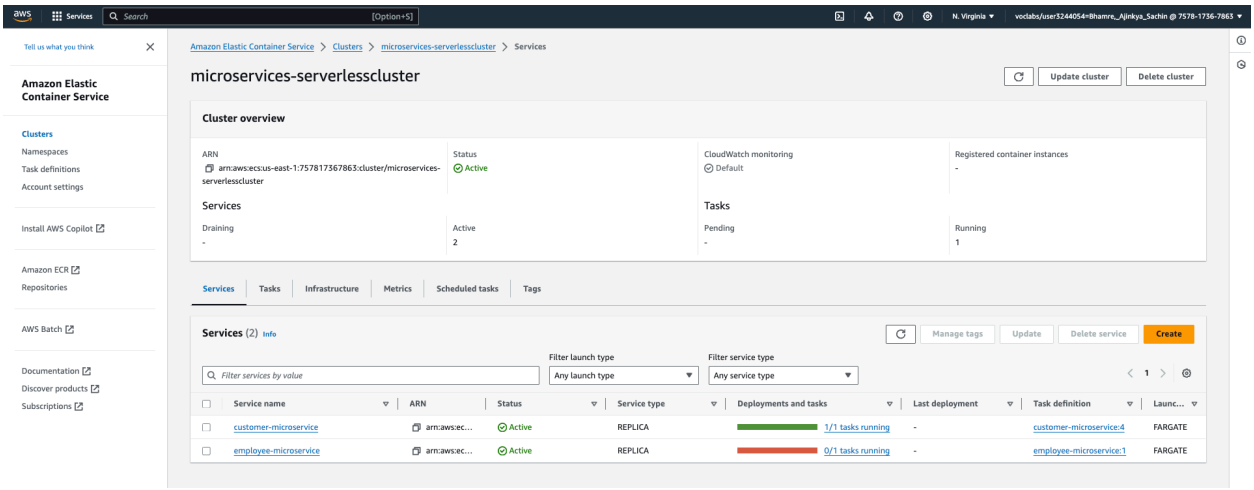


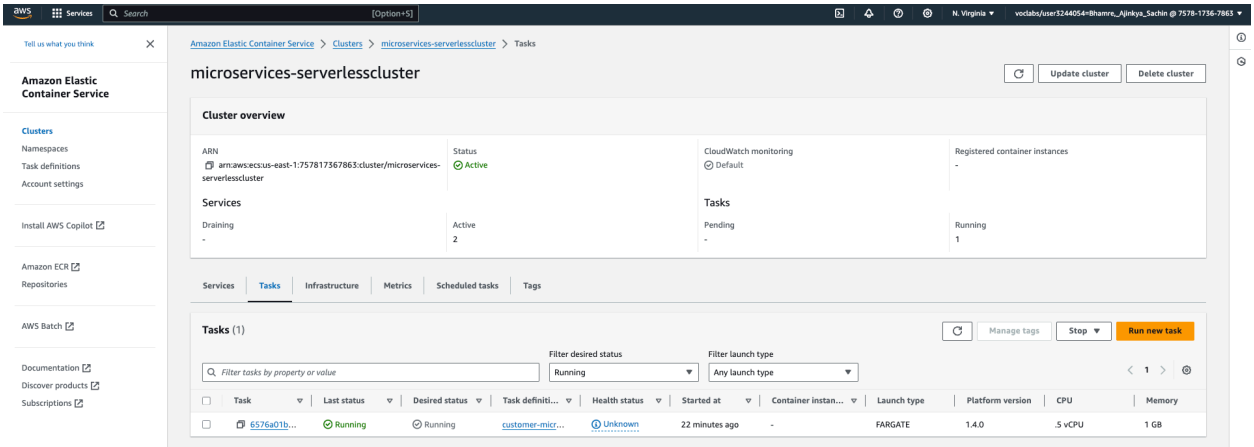


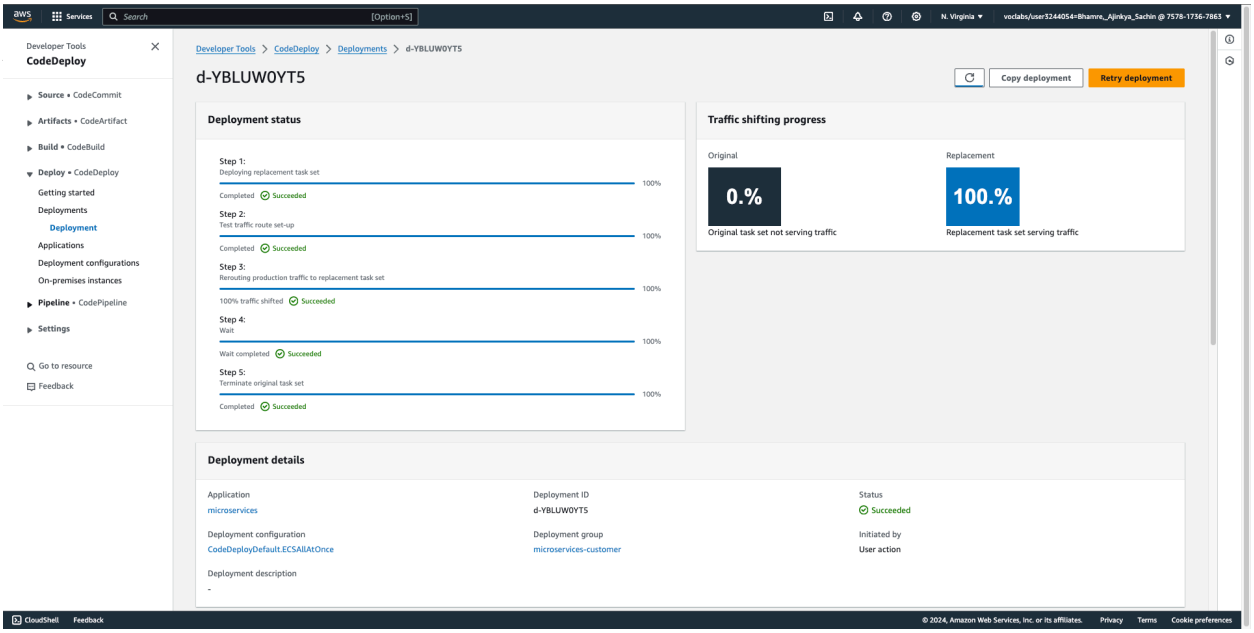


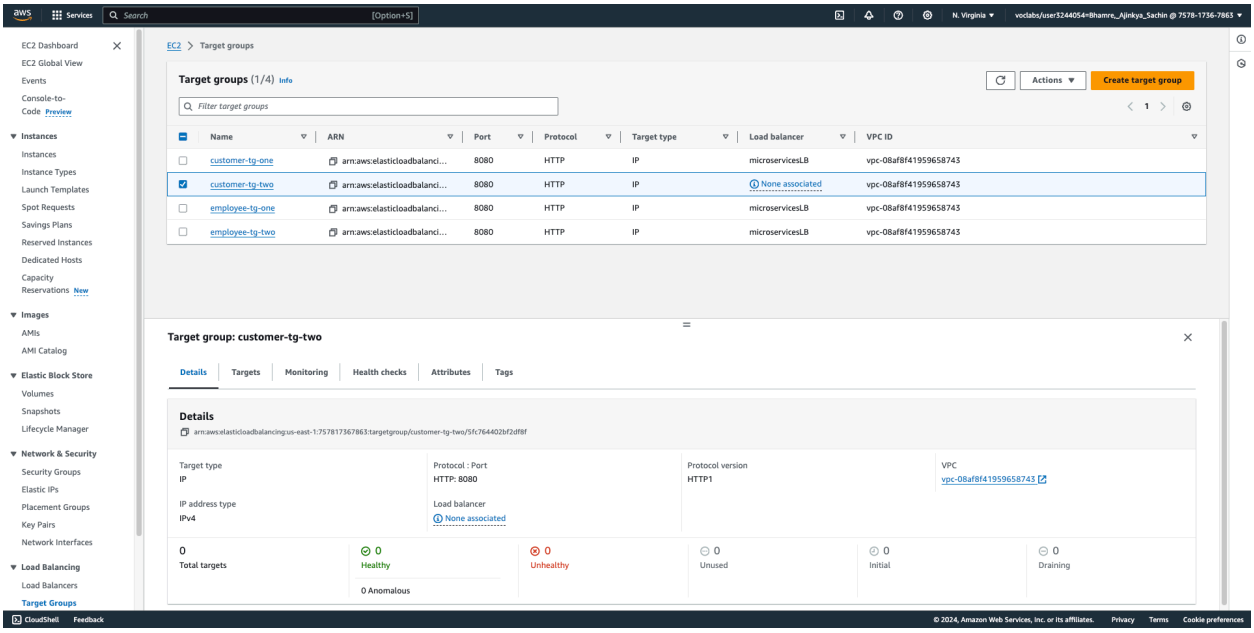
5. Return to CodeDeploy Page





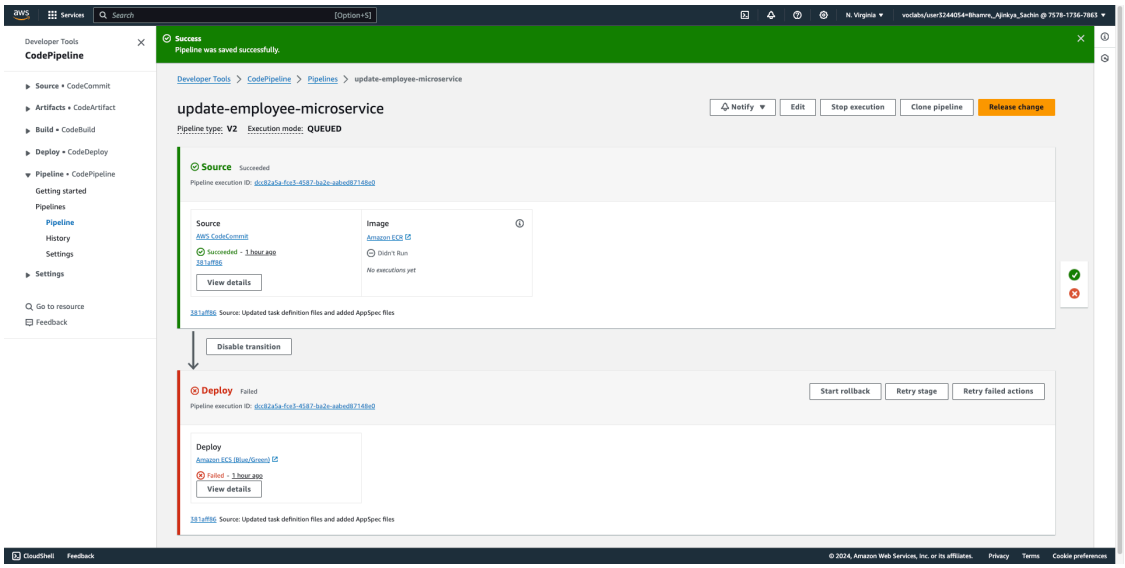






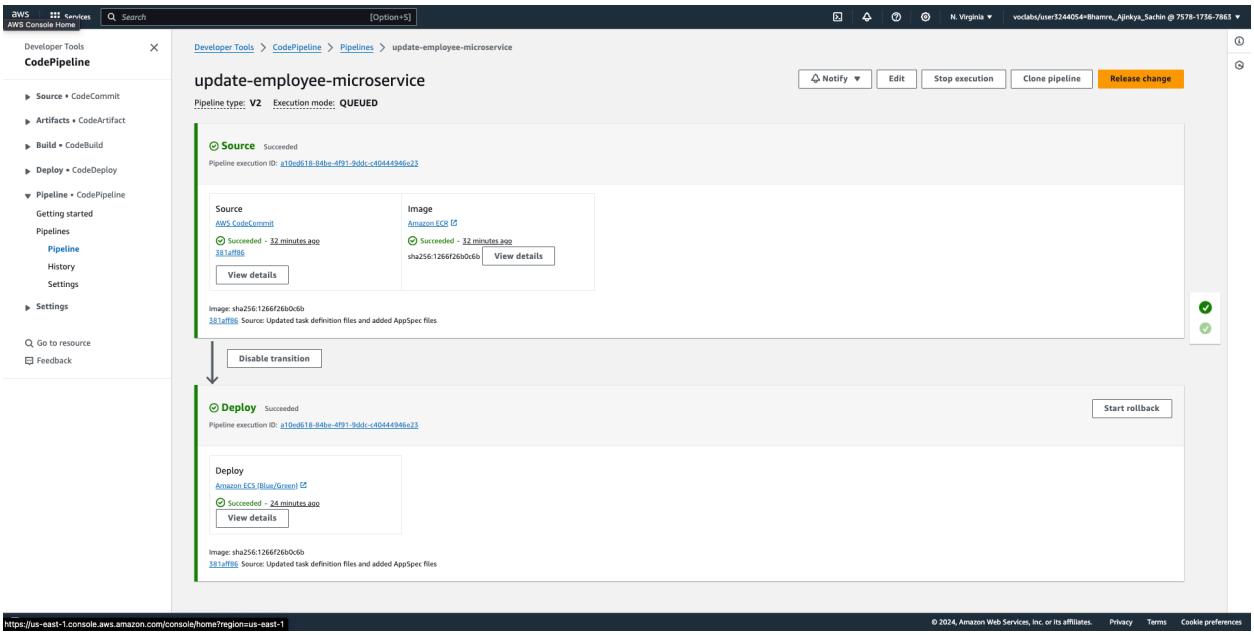
6. Observe Load Balancer and Target Group Settings

Task 8.4: Create a pipeline for the *employee* microservice

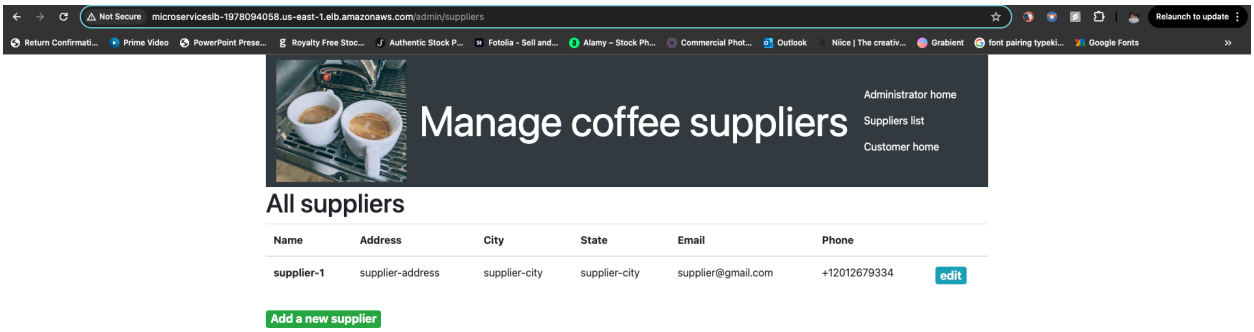


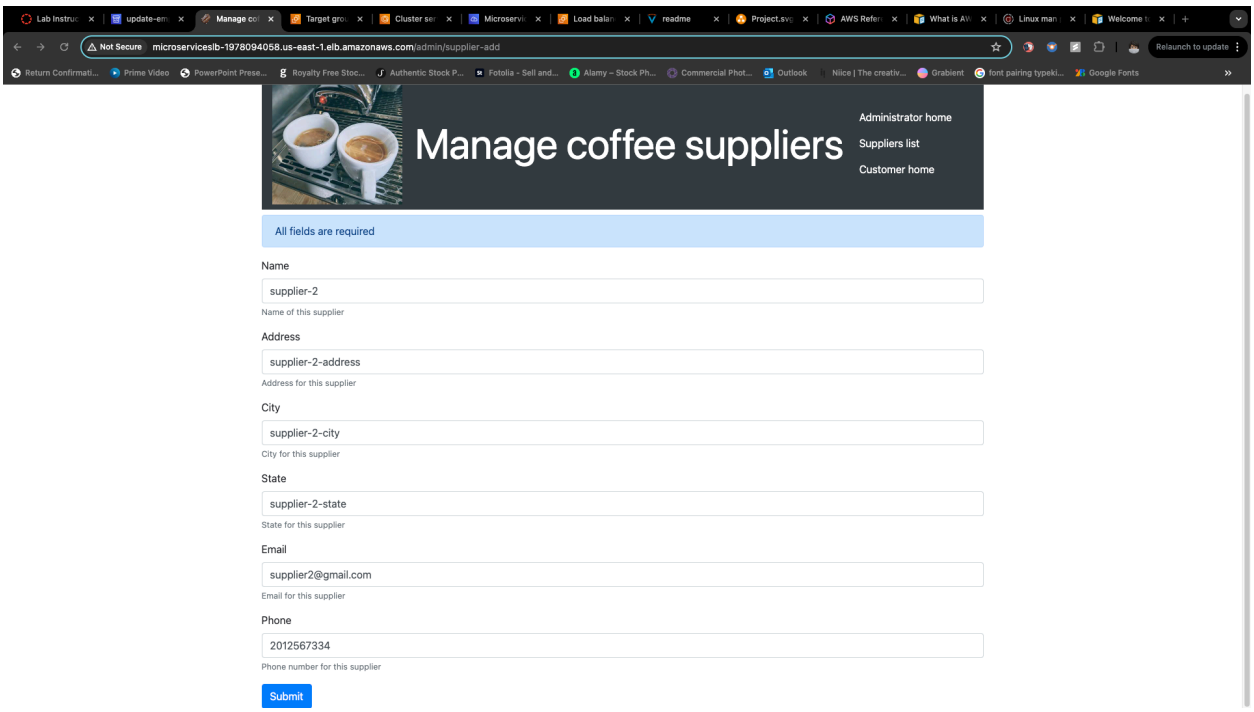
Task 8.5: Test the CI/CD pipeline for the *employee* microservice

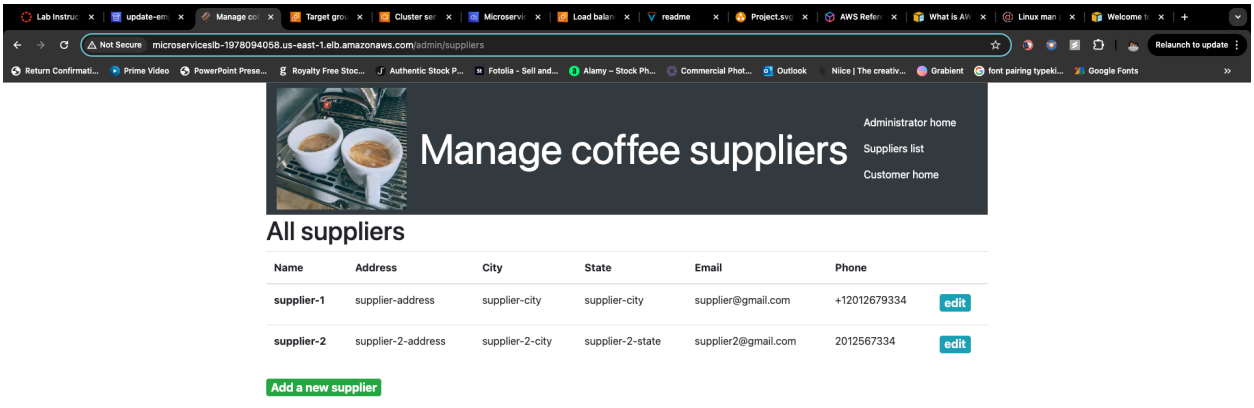
1. Launched Deployment:
2. Monitored CodeDeploy Progress:
3. Loaded Employee Microservice:
4. Tested Employee Microservice:
5. Observed Running Tasks:
6. Confirmed Deployment Success:



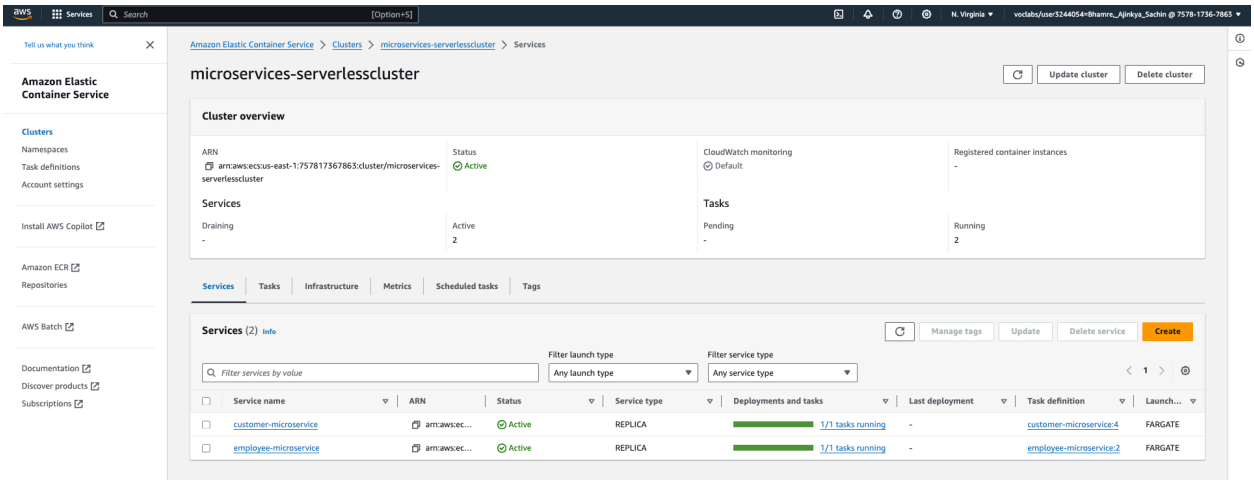
Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu

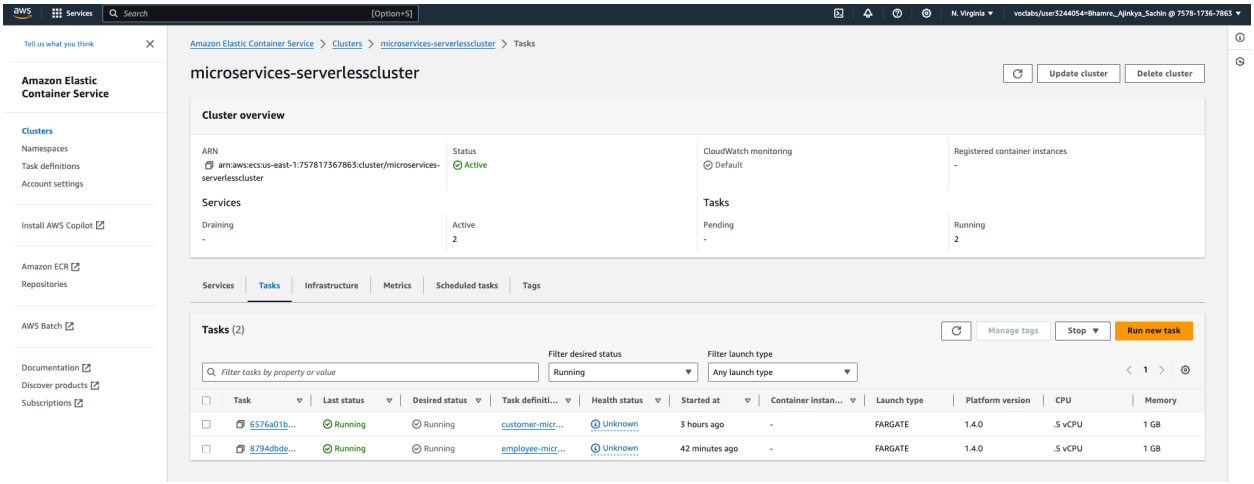


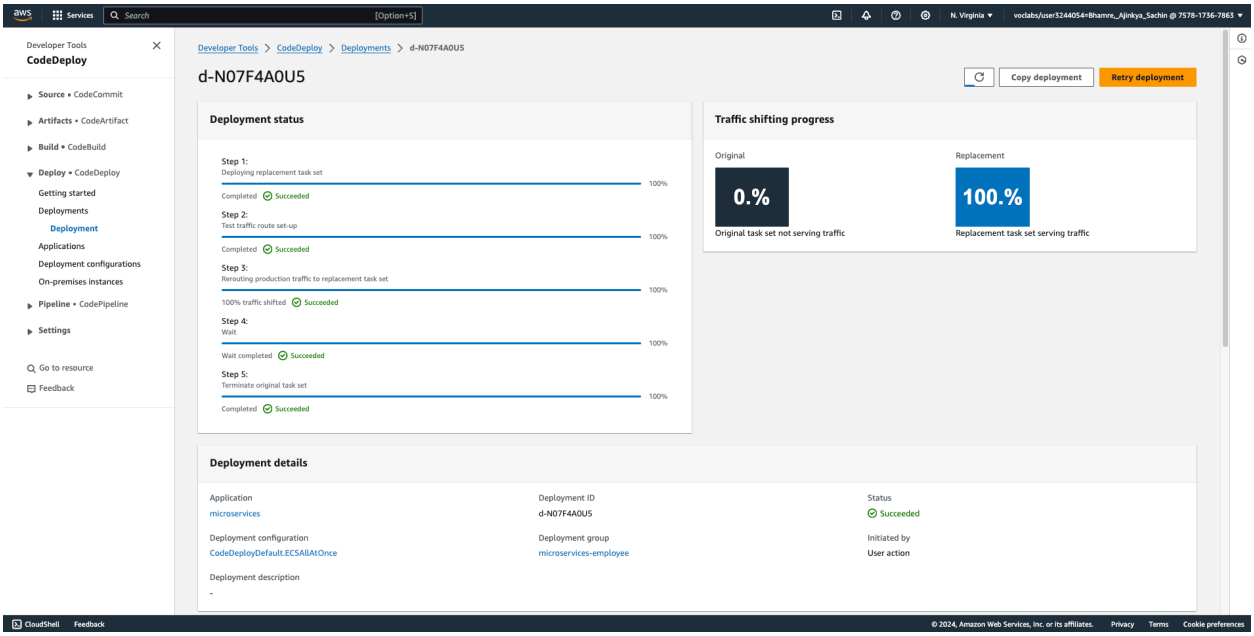




Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu

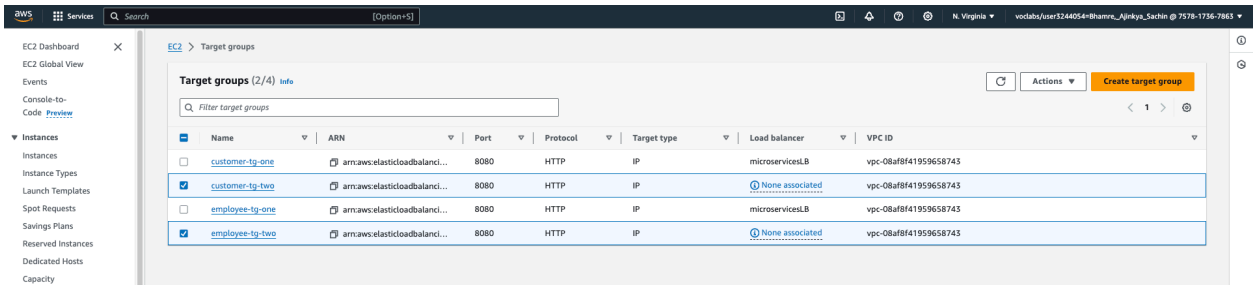






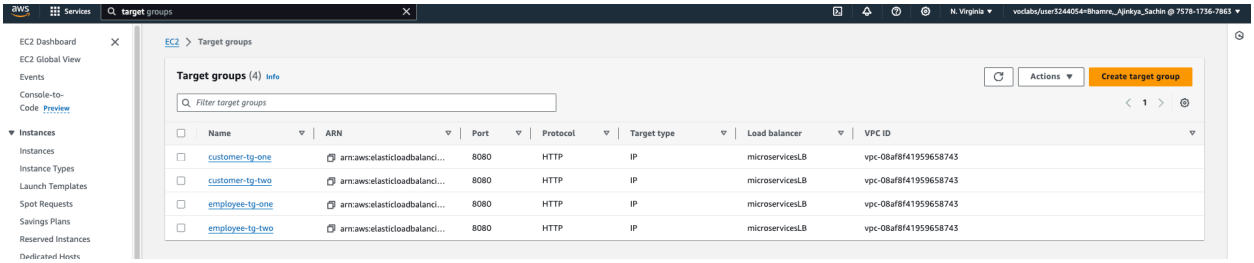
Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu

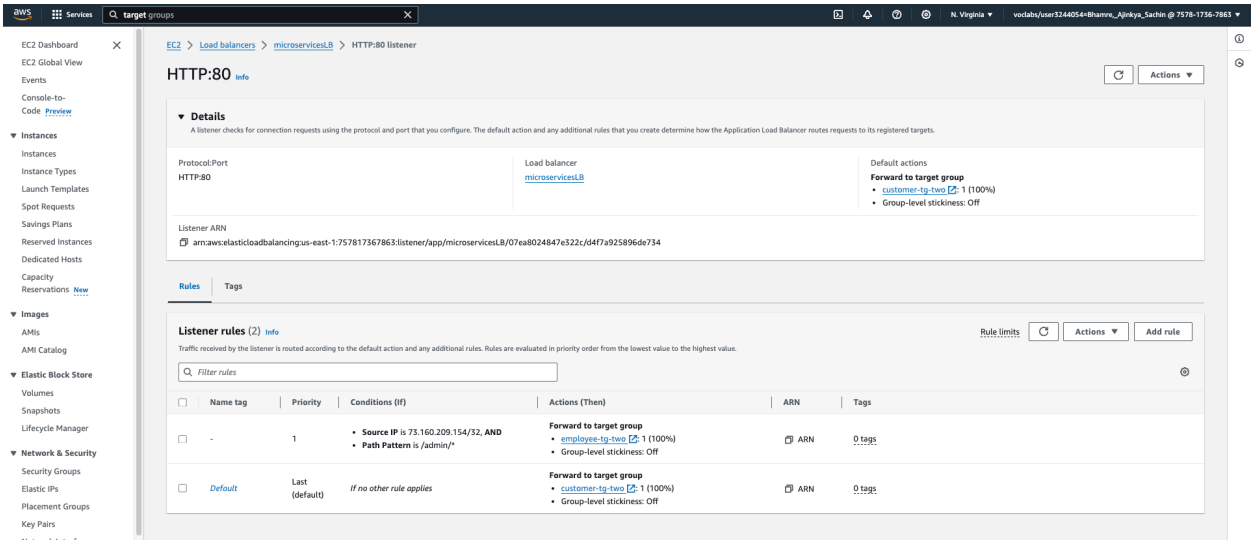
Task 8.6: Observe how CodeDeploy modified the load balancer listener rules



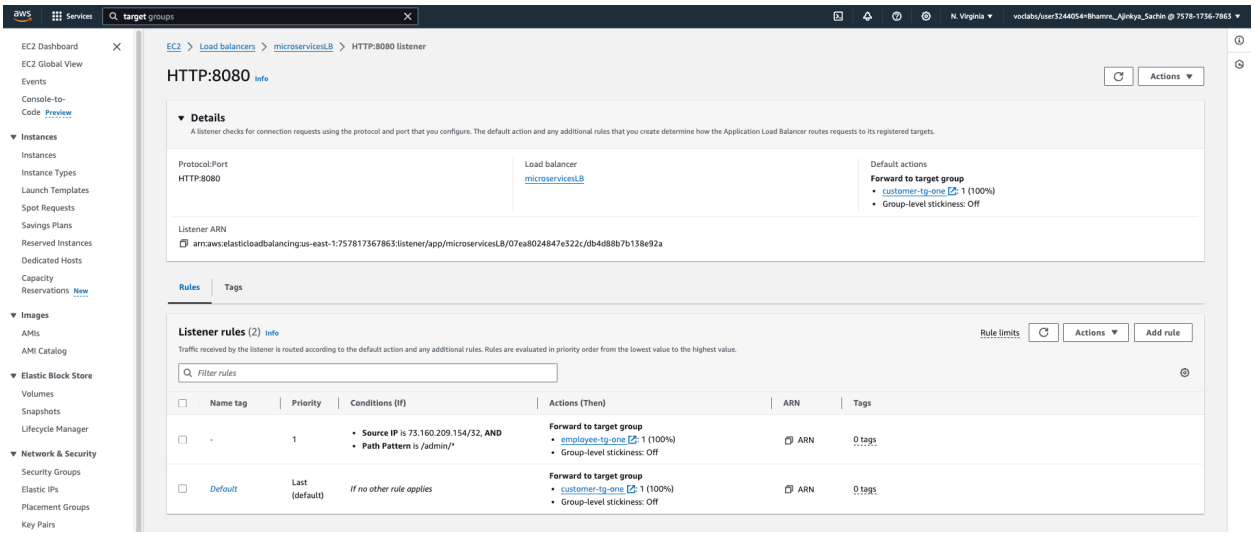
Phase 9: Adjusting the microservice code to cause a pipeline to run again

Task 9.1: Limit access to the *employee* microservice

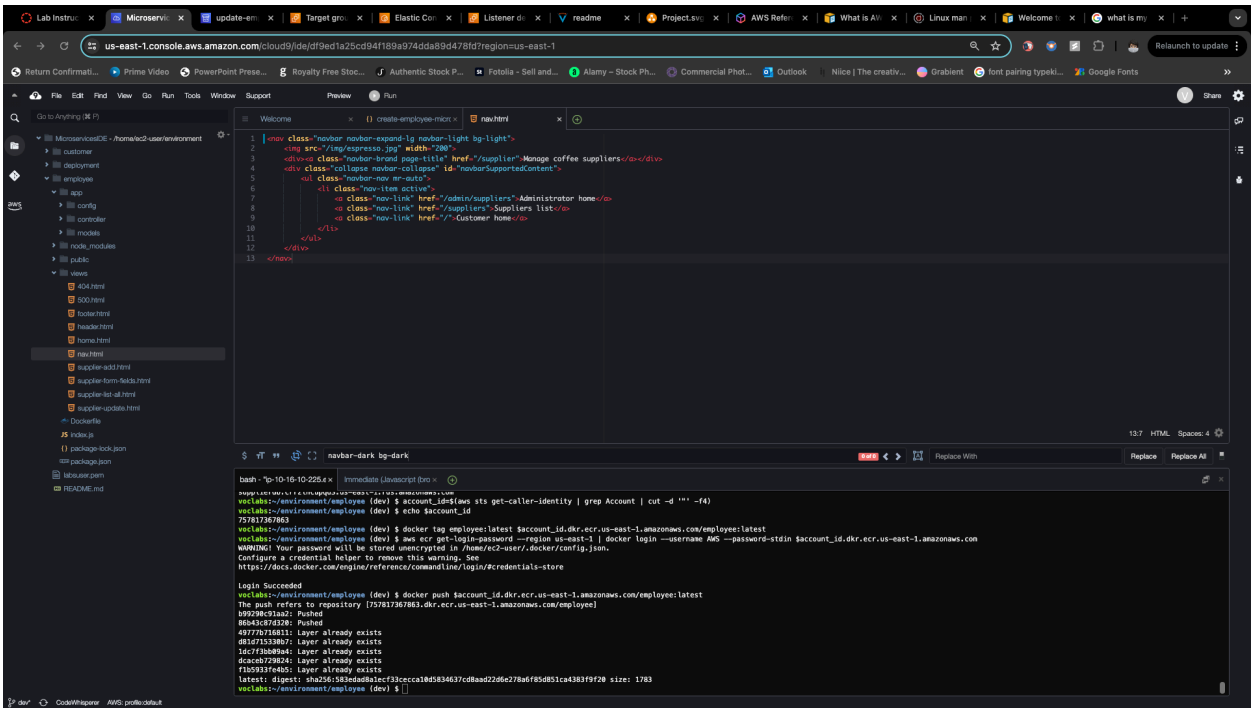




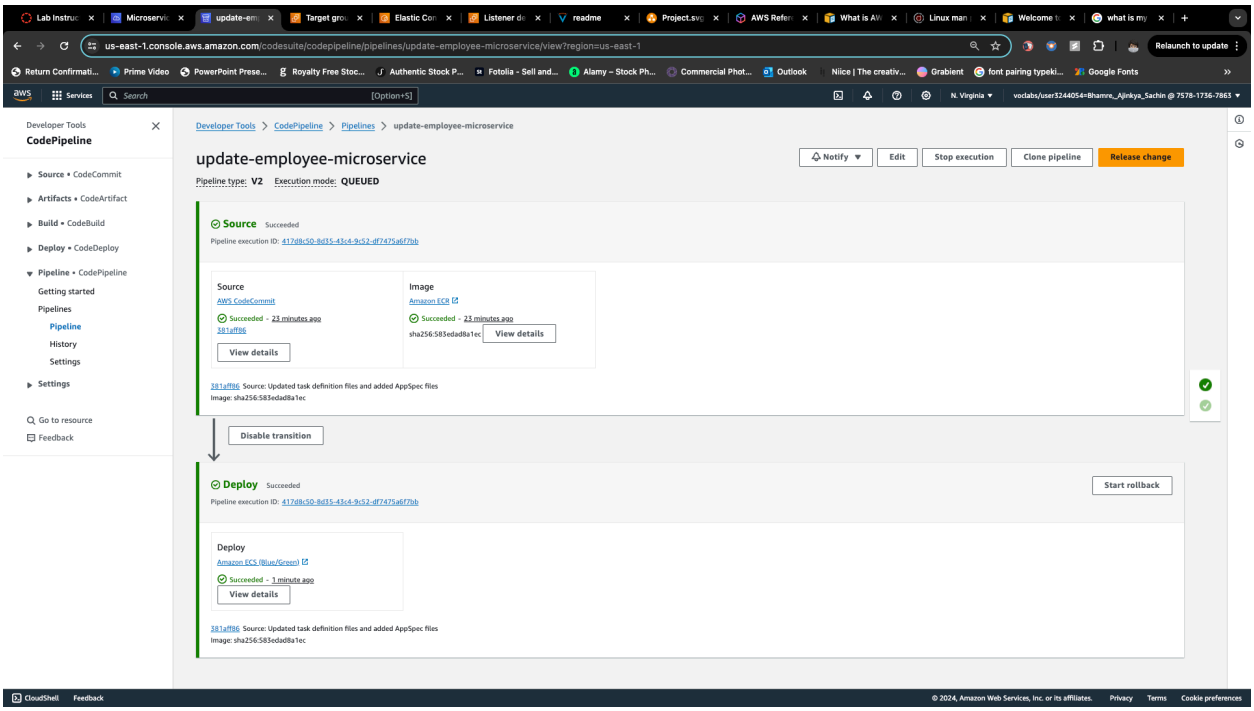
Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu

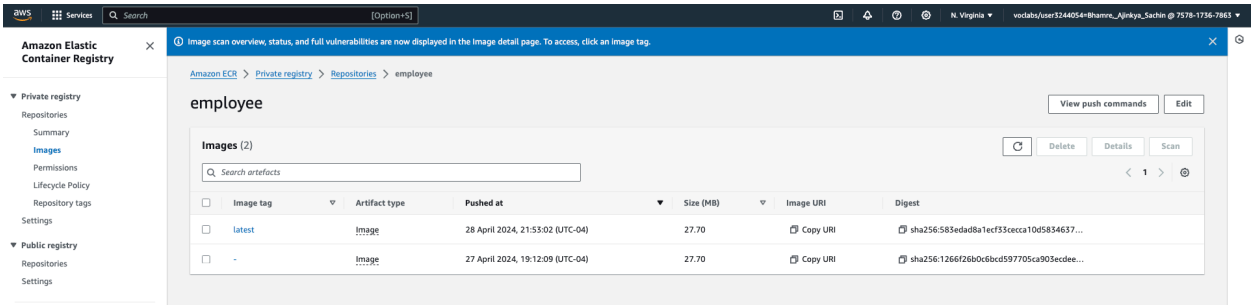


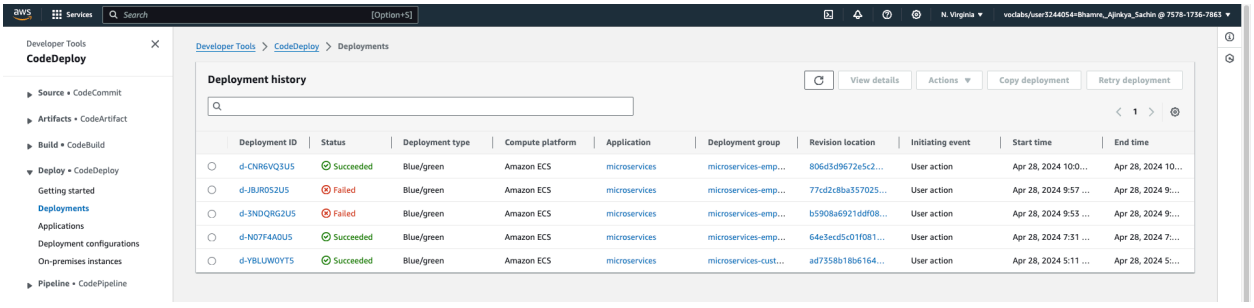
Task 9.2: Adjust the UI for the *employee* microservice and push the updated image to Amazon ECR



Task 9.3: Confirm that the *employee* pipeline ran and the microservice was updated



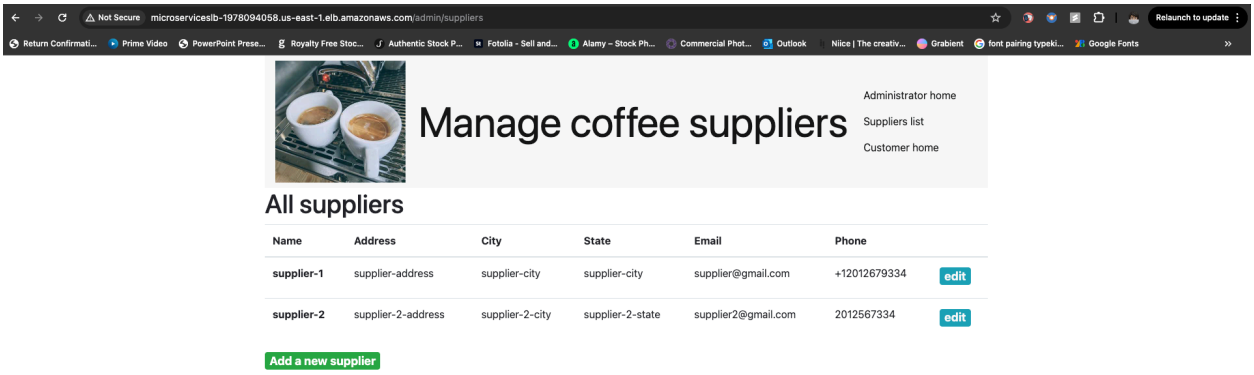




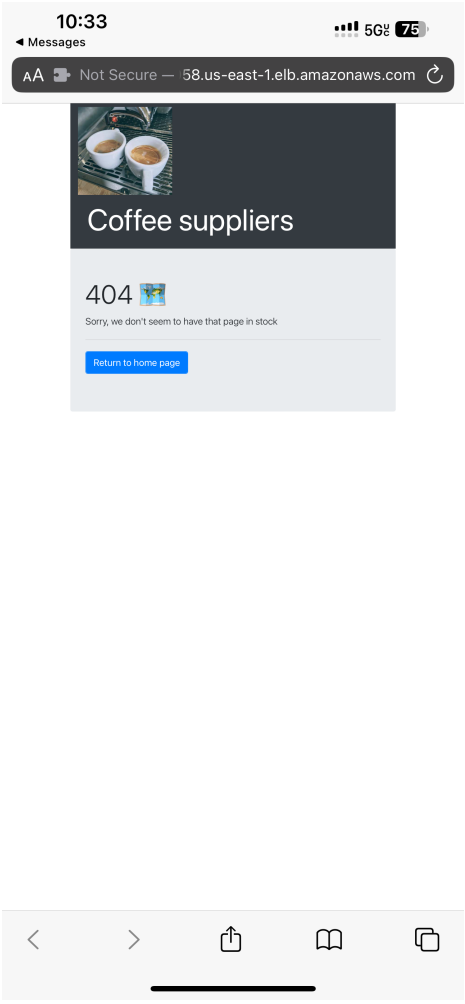
Task 9.4: Test access to the *employee* microservice

Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu

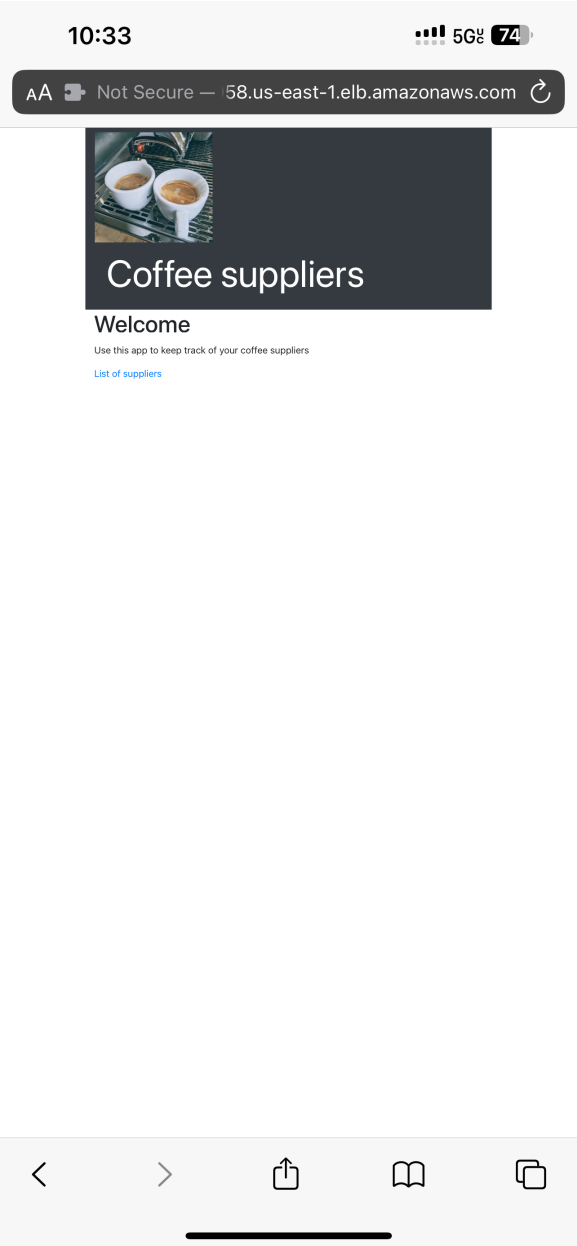
1. Tested from the Same Device: (http://microserviceslb-1978094058.us-east-1.elb.amazonaws.com/admin/suppliers)



2. Verified that the banner with the page title is now a light color due to the change made to the nav.html file.
3. Tested from a Different Device:



Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu



Used a different device (e.g., smartphone) / connected from a different network.

Visited the same URLs (http://<alb-endpoint>/admin/suppliers and http://<alb-endpoint>:8080/admin/suppliers) from this device.

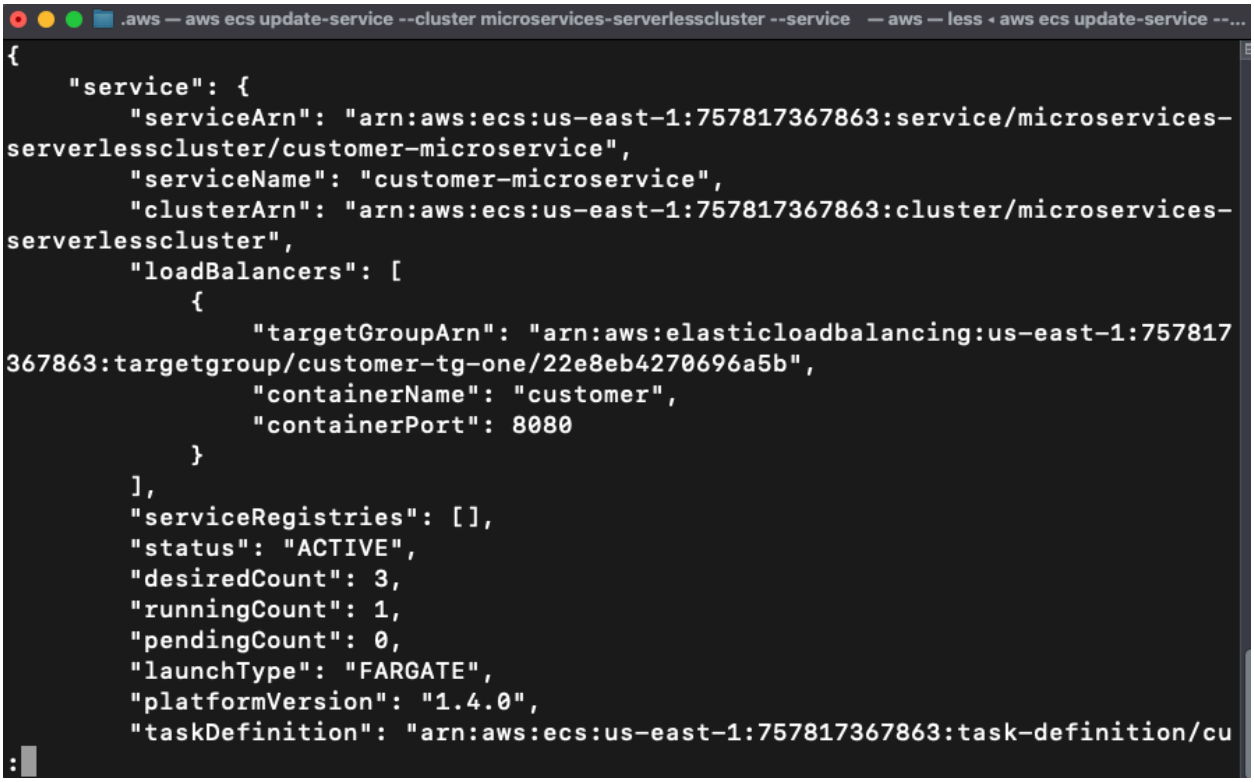
Verified that I receive a 404 error on any page that loads.

The page should display "Coffee suppliers" instead of "Manage coffee suppliers."

This indicates that access to the employee microservice is restricted based on the IP address.

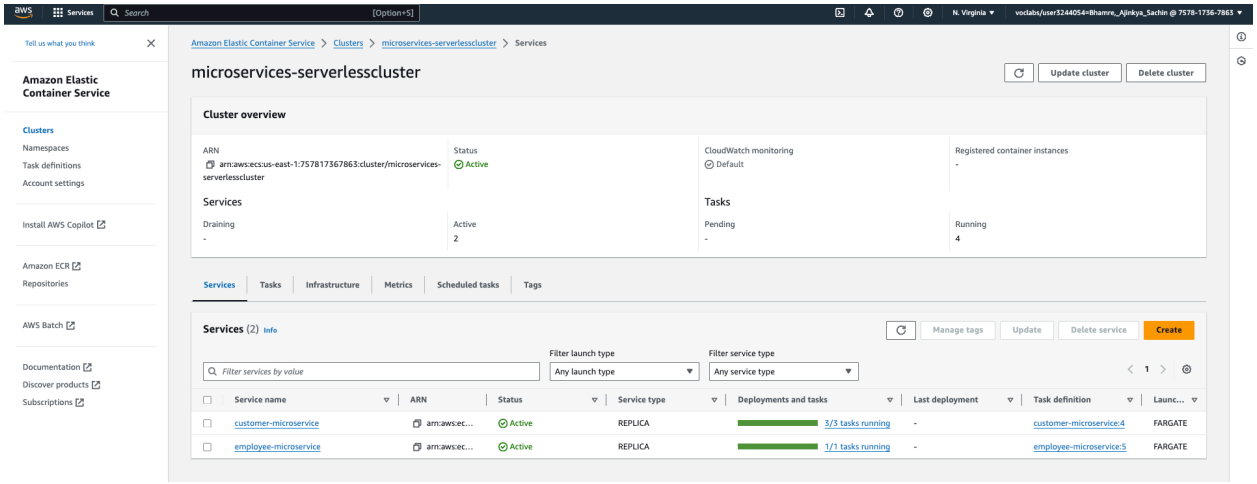
Task 9.5: Scale the *customer* microservice

1. Updated the Customer Service in Amazon ECS: aws ecs update-service --cluster microservices-serverlesscluster --service customer-microservice --desired-count 3



This command increases the desired count of containers for the customer-microservice service to 3.

2. View the Customer Service in Amazon ECS:



Ajinkya Bhamre - 20017896 Email : abhamre@stevens.edu

Part 2: Analysis

In Phase 1, I meticulously planned the solution's design and estimated its cost using the AWS Pricing Calculator. Creating an architectural diagram helped visualize the solution's components and their interactions, laying a solid foundation for implementation. The cost estimate breakdown provided insights into the financial implications, aiding in budgeting for the project's duration.

Phase 2 involved analyzing the existing monolithic application's infrastructure and functionality. By verifying its availability and interacting with its features, I gained a comprehensive understanding of its behavior and structure. Exploring the application's code, running processes, and database connections deepened my insights, paving the way for a smooth transition to a microservices architecture.

In Phase 3, I established a development environment using AWS Cloud9 and integrated the application code into AWS CodeCommit. Setting up the Cloud9 instance with the required configurations and copying the source code facilitated seamless development. Initializing Git repositories, creating branches, and pushing changes to CodeCommit ensured version control and collaboration efficiency.

Transitioning to a microservices architecture in **Phase 4** involved several crucial steps. Adjusting security group configurations, modifying microservices functionalities, creating Docker images, and testing containerized applications laid the groundwork for scalability and deployment automation. Version control with CodeCommit ensured code integrity and traceability, fostering a collaborative development environment.

Phase 5 focused on preparing microservices for deployment on Amazon ECS and integrating them into the CI/CD pipeline. Establishing ECR repositories, creating ECS clusters, defining task definitions, and authoring AppSpec files ensured smooth deployment orchestration. Updates to deployment files and code check-ins streamlined the integration process, setting the stage for continuous delivery.

In Phase 6, I configured an Application Load Balancer to serve as the HTTPS entry point for the application. Setting up target groups and routing rules enabled efficient traffic distribution, supporting scalability and reliability for the microservices.

Phase 7 involved creating ECS services for the microservices, defining task definitions, and utilizing AWS CLI commands for service creation. These services ensure the seamless deployment and management of microservices within the ECS cluster, enhancing scalability and reliability.

Phase 8 focused on configuring CodeDeploy and CodePipeline for microservice deployment. Creating CodeDeploy applications and deployment groups, setting up pipelines, and employing blue/green deployment strategies ensured smooth updates. Testing pipelines and observing CodeDeploy's modifications to load balancer rules confirmed successful deployments.

In Phase 9, I adjusted microservice code, observed CI/CD pipelines in action, and tested access restrictions. Limiting access to the employee microservice, modifying UI, and triggering pipeline runs through Docker image updates demonstrated the agility and automation of the deployment process. Scaling the customer microservice showcased independent microservice management, enhancing scalability and efficiency.