

1. Draw the cloud diagram that shows the network infrastructure, as well as all resources that needs to be created (you are free to work with AWS, GCP, or Azure)

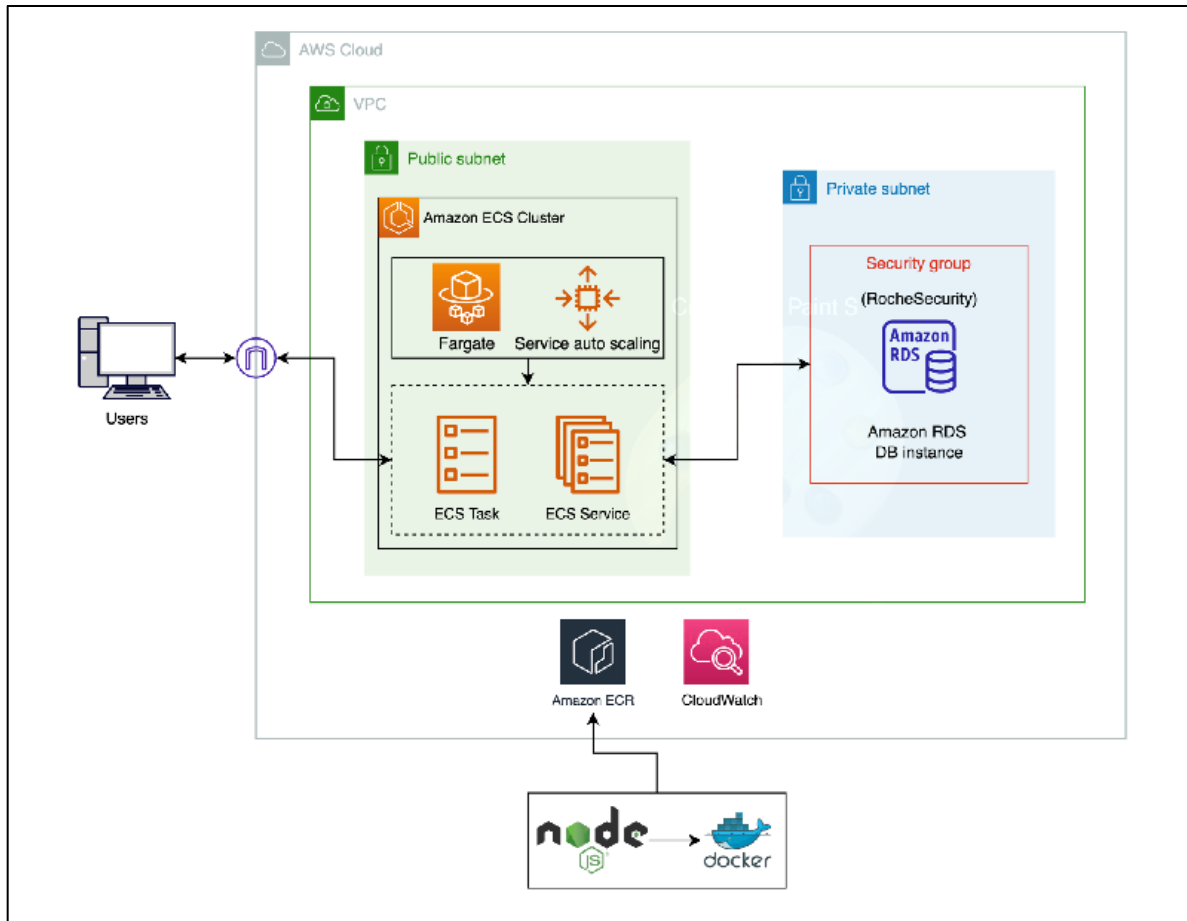


Figure 1: Cloud diagram using Amazon Web Service (AWS)

The diagram shows the following resources:

- Amazon Elastic Container Service (Amazon ECS) to run, stop, and manage Docker containers on a cluster of ECS Fargate.
- AWS ECS Fargate as the serverless compute engine to run containers.
- AWS ECS Task is a unit of work that can run a Docker container on an Amazon ECS Cluster.
- AWS ECS Service is a logical grouping of tasks that defines how tasks are deployed and scaled within an Amazon ECS Cluster.
- Service auto scaling to automatically launch and stop ECS Tasks based on demand.
- Elastic Container Registry (ECR) to store Docker images.
- VPC for network isolation and security groups for network access control.
- RDS Aurora MySQL database to manage data.
- CloudWatch for monitoring logs and metrics.

I have used various AWS cloud services as I mentioned above to create scalable, secure, and reliable microservices. I used AWS ECS and ECR to deploy and run the docker image of the microservices which is created using nodejs. I setup service auto scaling with AWS Fargate to automatically scale the AWS ECS Tasks to provide high performance and high availability of microservices during peak times. I used AWS

RDS service to design system database and store the data inside the Aurora MySQL. I put instance of AWS RDS inside the private subnet and created security group associated with it to prevent the unauthorized access to backend database. This database instance can only be accessed by using AWS ECS Tasks which are located inside the public subnet. Finally, the performance and errors of the microservices will be tracked by the AWS CloudWatch.

2. Draw the data model for the microservice.

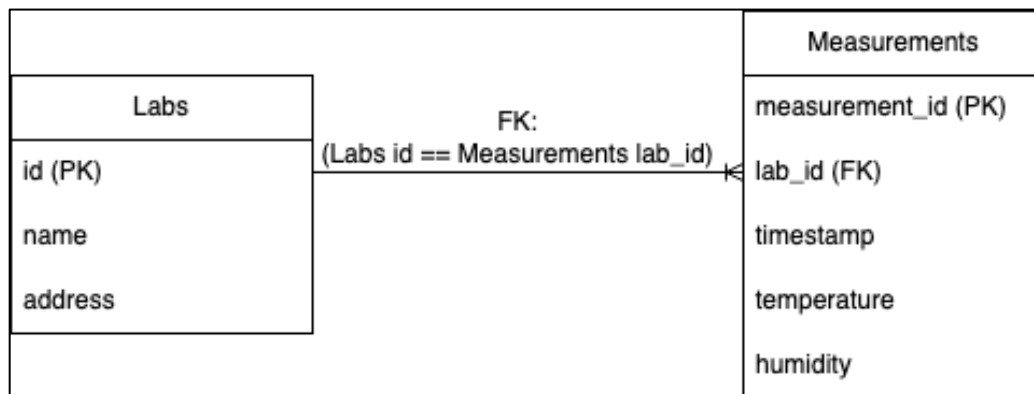


Figure 2: Diagram of data model with tables, and relationships

- This data model consists of two tables: Labs and Measurements.
- The Labs table stores information about each lab, including its unique id, name, and address.
- The Measurements table stores the measurements collected from the lab devices. It has a foreign key relationship with the Labs table, which ensures that each measurement is associated with a specific lab.
- Each Measurement record has a unique measurement_id, a timestamp indicating when the measurement was taken, and the temperature and humidity values captured at that time. Also, the lab_id field is used as a foreign key to link the measurements to the corresponding lab.
- Using this data model, we can easily query the average temperature and humidity values for each lab over the past 24 hours. We can group the measurements by lab_id and calculate the average temperature and humidity for each group.

3. You are free to choose the values and the units posted in the message sent to the API. Explain what solutions could be possible (at least 3 different options). Which one do you select, and why do you make this choice?

- There are 3 possible options for the values and units posted in the message sent to the API:

Option 1: Temperature in Celsius and Humidity in percentage

- Temperature: °C
- Humidity: %

Option 2: Temperature in Fahrenheit and Humidity in percentage

- Temperature: °F
- Humidity: %

Option 3: Temperature in Kelvin and Humidity in grams of water vapor per cubic meter

- Temperature: K
- Humidity: g/m³

For this use case, I would select Option 1, with temperature in Celsius and humidity in percentage.

Celsius is a commonly used unit of temperature measurement in scientific and industrial settings and is also widely understood by the public. Also, Celsius is easily convertible to Kelvin, which is another commonly used unit of temperature measurement.

Humidity is often expressed as a percentage, which represents the ratio of the amount of water vapor in the air to the maximum amount of water vapor that the air can hold at that temperature. This is a widely used unit for measuring humidity, as it is easy to understand and interpret.

Overall, Option 1 offers a clear and widely understood representation of temperature and humidity values, making it the most practical choice for this use case.

4. You are free to choose the database to use. Explain which one do you choose, and what motivates that choice.

- For the use case and data model we have selected, I choose a relational database management system (RDBMS) – Amazon RDS Aurora MySQL.

The main reason for this choice is that Aurora MySQL is designed to deliver high performance, availability, and durability for traditional relational database workloads. It is built on top of MySQL and provides compatibility with MySQL, which makes it easy to use and widely supported by existing applications and tools.

In addition, Aurora MySQL integrates well with other AWS services such as Amazon EC2, Amazon ECS, and AWS Lambda, which we are using in our cloud infrastructure. It is a cost-effective solution compared to other managed database services available in the market. It offers a pay-as-you-go pricing model that allows users to pay only for what they use.

Overall, AWS RDS Aurora MySQL is a reliable, scalable, and performant database solution that meets the requirements of our use case and is well-suited for our cloud infrastructure on AWS.

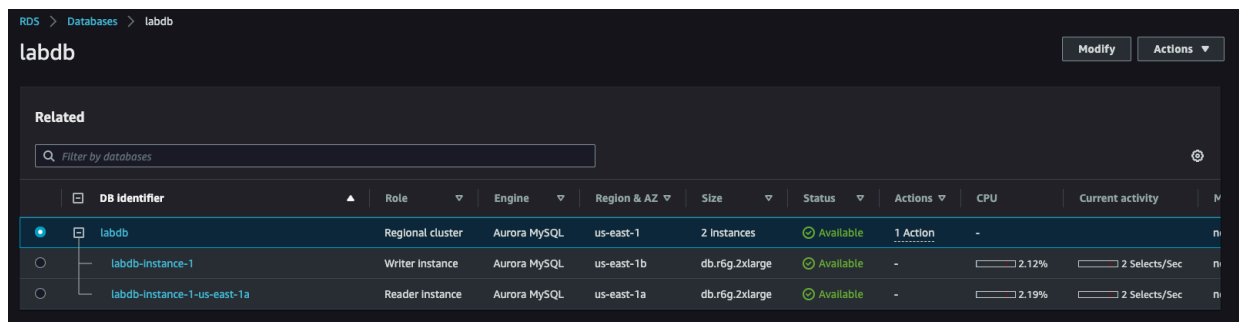


Figure 3: Screenshot of RDS database instance

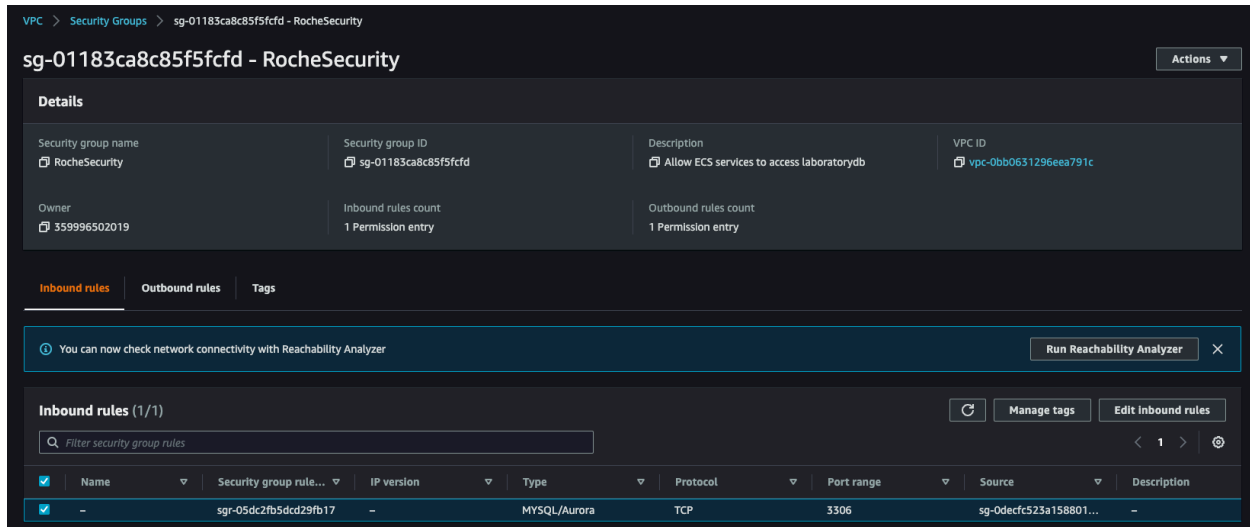


Figure 4: Screenshot of Security group named “RocheSecurity” to prevent public access

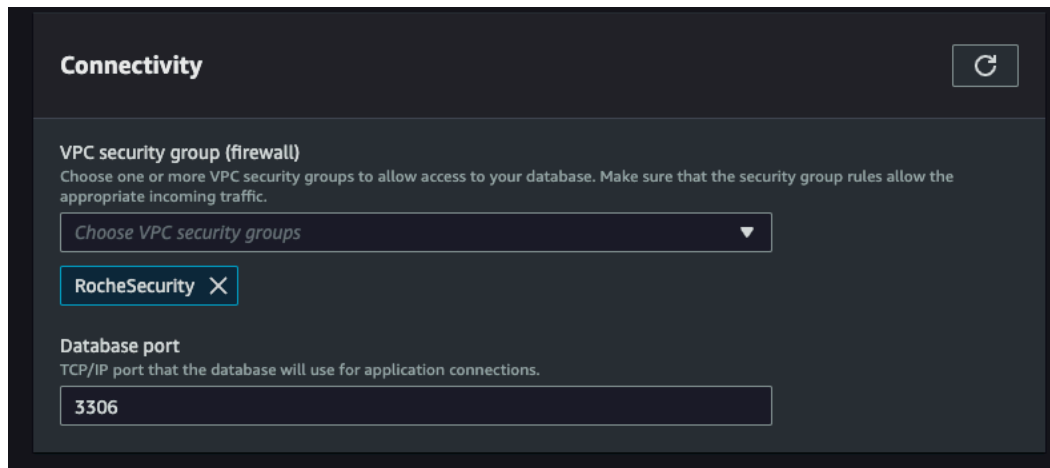


Figure 5: Screenshot of attaching security group to AWS RDS database instance

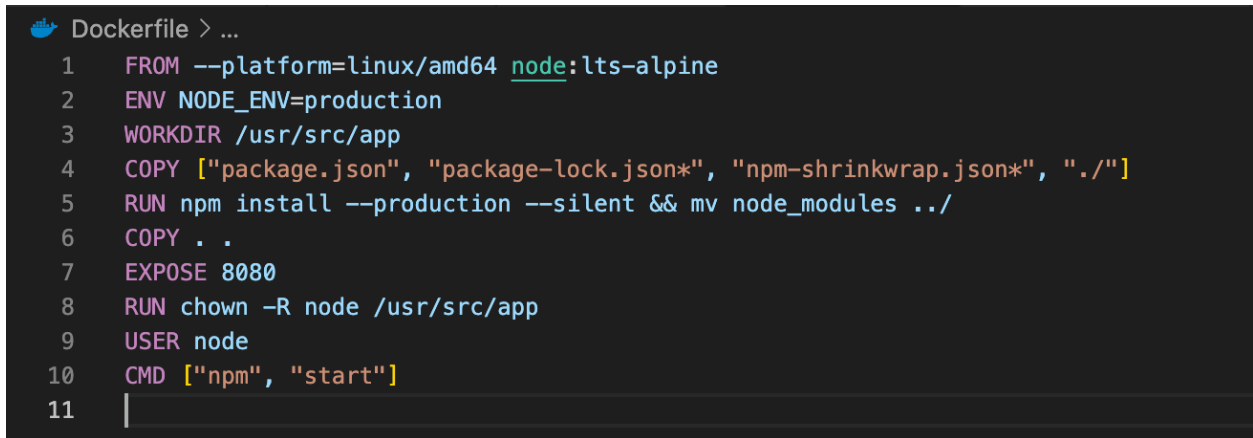
So, it will prevent the unauthorized use of backend database. That means if we try to access it using other IP address then 54.242.210.215(AWS ECS Public IP address) this, then error message will be displayed.

5. Create a github repo and implement the microservice in the language of your choice.

- Github repository link: <https://github.com/pritsorathiya5181/roche-coding-exercise.git>

6. Create the docker file, and document how to build the image and run it.

- The following is the docker file that runs with the base image node:lts-alpine image. Sets the working directory, copies the package files to the working directory first and install all the dependencies before we copy other files. This is because we don't want to install node_modules every time we change anything and build the image. It then copies the entire application code, exposes port 8080 and runs the command "npm start" to start the application.



```
Dockerfile > ...
1 FROM --platform=linux/amd64 node:lts-alpine
2 ENV NODE_ENV=production
3 WORKDIR /usr/src/app
4 COPY ["package.json", "package-lock.json*", "npm-shrinkwrap.json*", "."]
5 RUN npm install --production --silent && mv node_modules ../
6 COPY . .
7 EXPOSE 8080
8 RUN chown -R node /usr/src/app
9 USER node
10 CMD ["npm", "start"]
11 |
```

Figure 6: Screenshot of the code of Dockerfile

Command for build the image:

```
docker build -t node-service .
```

Once the image is built, you can run it using the following command:

```
docker run -p 8080:8080 node-service .
```

This will start the container and map port 8080 of the container to port 8080 of my host machine, so I can access the application at <http://localhost:8080>.

7. Deploy the docker image in your cloud provider account (you can create a 12-months free tier account with AWS)

- Here I have used Amazon Elastic Container Registry (ECR) to store Docker images. Amazon Elastic Container Registry (ECR) is a fully managed Docker container registry that makes it easy for developers to store, manage, and deploy Docker container images. Amazon ECR is integrated with Amazon Elastic Container Service (ECS), simplifying your development to production workflow.

a) What service do you choose for deployment, and why?

- Amazon ECS makes it easy to deploy, manage, and scale Docker containers running applications, services, and batch processes. I created a cluster named “backend-api” based on AWS Fargate. AWS Fargate is a technology that we can use with Amazon ECS to run containers without having to manage servers or clusters of Amazon EC2 instances. With AWS Fargate, we no longer have to provision, configure, or scale clusters of virtual machines to run containers. This removes the need to choose server types, decide when to scale the clusters, or optimize cluster packing.

8. Please provide a URL for reaching that API, so we can test it with Postman.

- GET API: <http://54.242.210.215:8080/labs/>
- POST API: <http://54.242.210.215:8080/labs/upload>

Posting data example:

```
[
  {
    "lab_id": "1",
    "timestamp": "2023-04-24T04:38:11.422Z",
    "temperature": "119.41°C",
    "humidity": "10.6%"
  },
  {
    "lab_id": "2",
    "timestamp": "2023-04-24T04:38:11.422Z",
    "temperature": "119.41°C",
    "humidity": "50.6%"
  },
]
```

9. Explain how do you deploy (deployment script should be provided in repo).

- I have created the dockerfile and built the image using the above-mentioned commands. Then I performed following steps to push the image into AWS ECR.

Step 1: Created private repository with named “node-service” that store the docker image.

Step 2: After the build completes using the build commands, tag the image to push it to this repository using the following commands.

```
docker tag node-service:latest 359996502019.dkr.ecr.us-east-1.amazonaws.com/node-service:latest
```

Step 3: Run the following command to push this image to the created AWS repository:

```
docker push 359996502019.dkr.ecr.us-east-1.amazonaws.com/node-service:latest
```

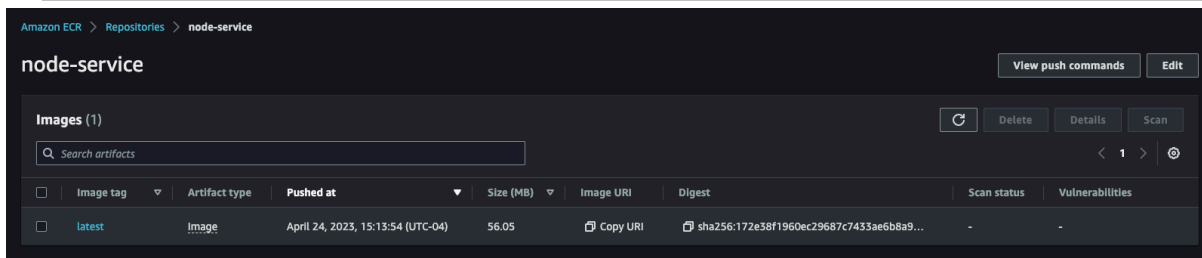


Figure 7: Screenshot of the docker image inside the node-service repository

- Create an ECS cluster named “backend-api” in your AWS account using the ECS console.
- Create an ECS task definition named “backend-task-definition” for the container. This contains ECR image URI, Port mappings, environment variables, which all are necessary to define how container runs.

- Create an ECS service named “backend-microservice” that runs the created task definition. This ensures that the container is always running and can scale as needed. This service is configured for Service auto scaling functionality by setting desired tasks, creating scaling policy named “Roche Policy” and it’s target value which indicates when new tasks should automatically create.

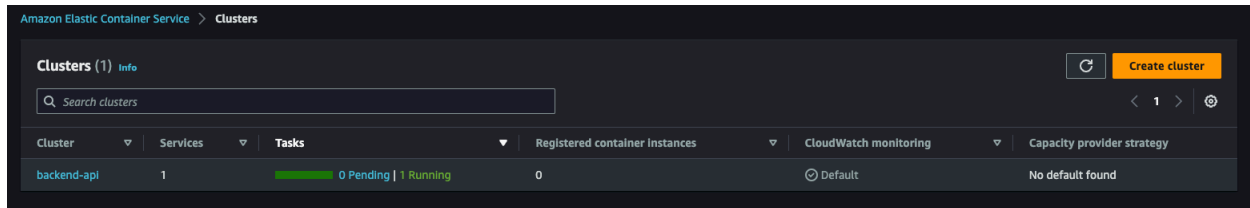


Figure 8: Screenshot of AWS ECS Cluster

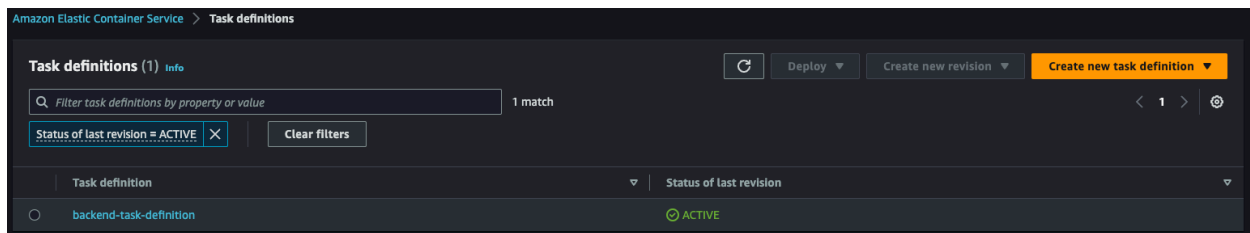


Figure 9: Screenshot of Amazon ECS Task Definition

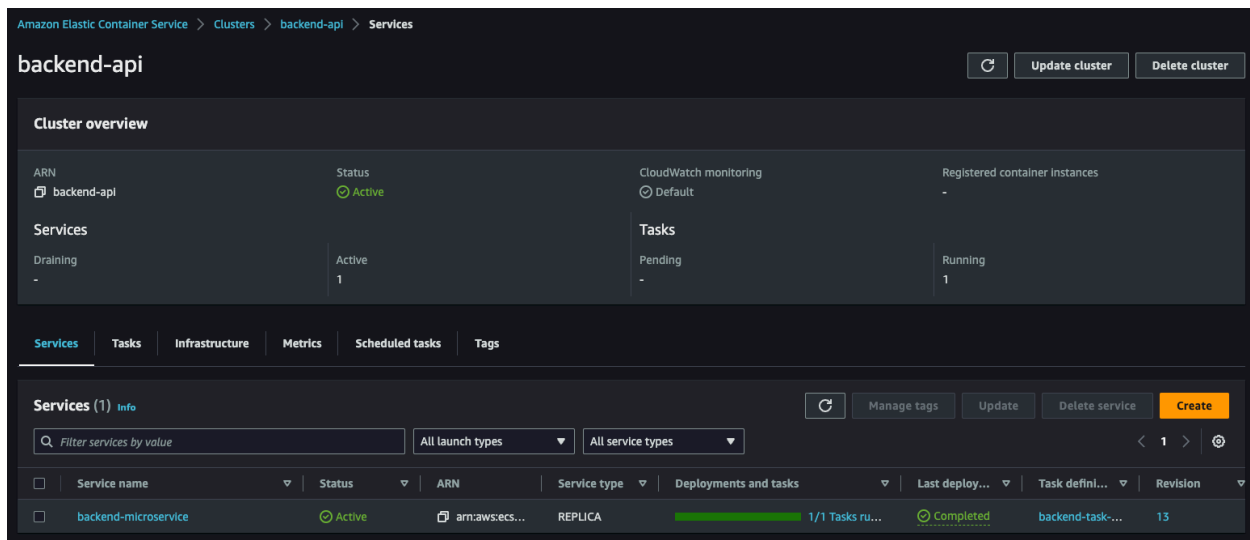


Figure 10: Screenshot of Amazon ECS service

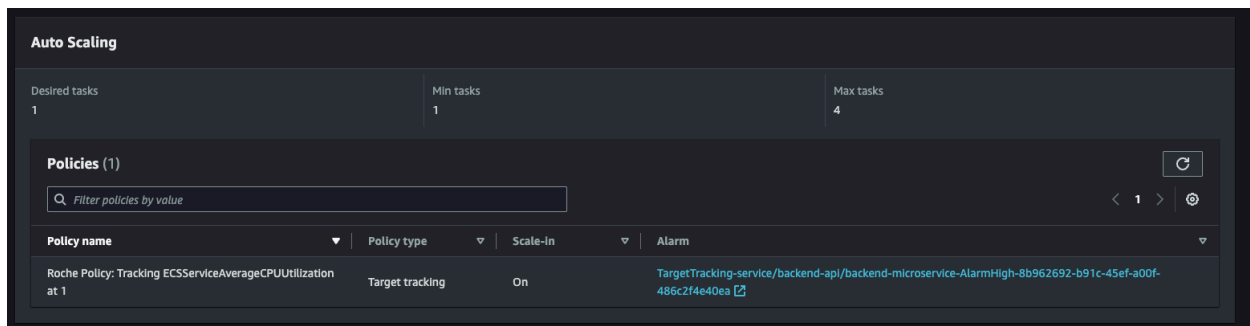


Figure 11: Screenshot of customized policy inside service auto scaling group

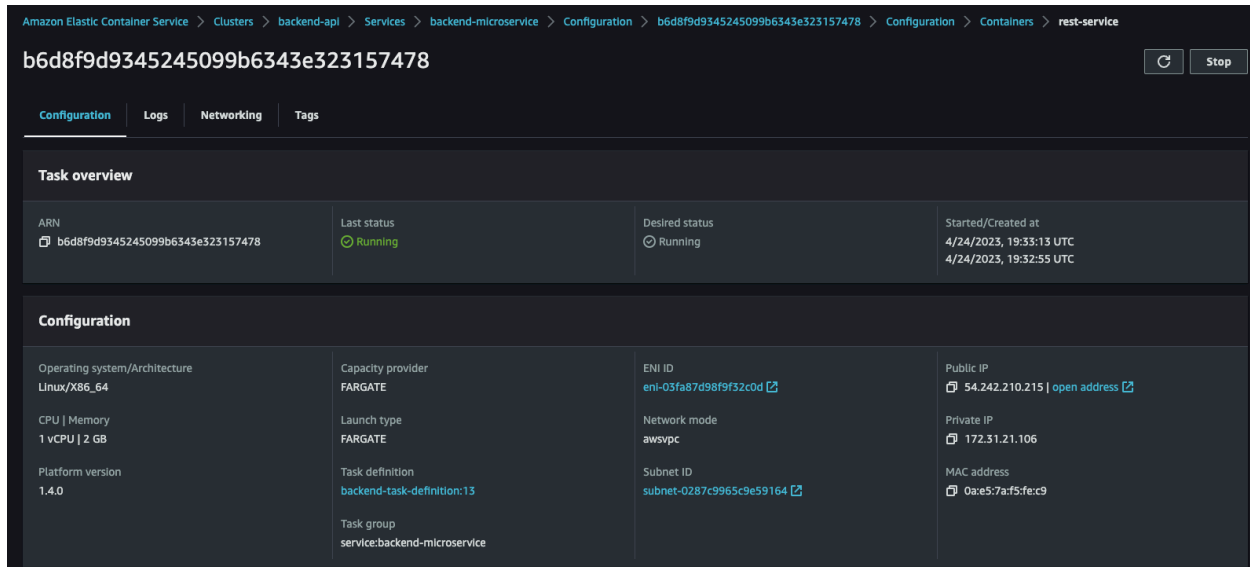


Figure 12: Screenshot of “backend-microservice” configuration indicating public IP to reach APIs

10. How would you monitor the micro-service? (explain what indicators to monitor regarding the business case, and how to get alerted if something is going wrong)

- Monitoring a micro-service is critical to ensure its reliability, performance, and availability. Here are some indicators that could be monitored.
 1. CPU usage: Monitoring the CPU usage of the application container can help detect potential bottlenecks or performance issues. If the CPU usage is consistently high, it may indicate a need to scale the application horizontally or vertically.
 2. Memory usage: Monitoring the memory usage of the application container can help detect potential memory leaks or inefficient memory usage patterns. If the memory usage is consistently high, it may indicate a need to optimize the application's memory usage or increase the container's allocated memory.
- Amazon ECS service provides the health and metrics dashboard to track the CPU utilization and Memory utilization.

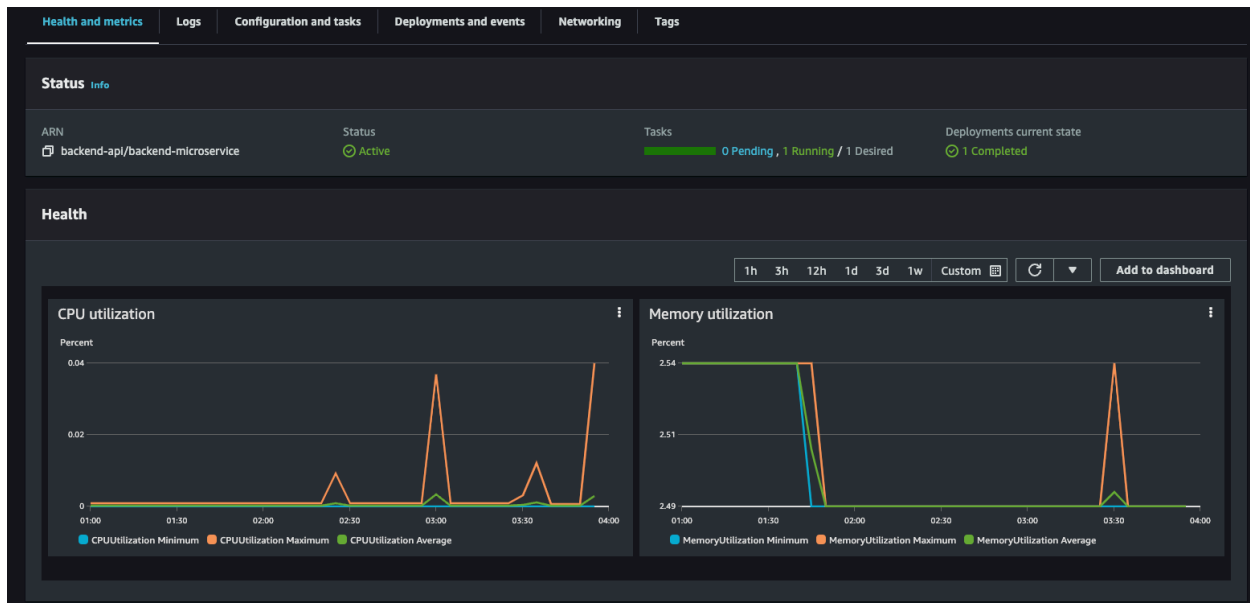


Figure 13: Screenshot of “backend-microservice” health and metrics dashboard

- I have also configured this service with AWS CloudWatch. It can be used to collect and visualize metrics related to the microservice, set up alerts for specific metrics, and monitor logs to identify any errors or issues.
- I have set the policy (Roche Policy) in backend-microservice on 70% CPU utilization with desired task 1. So, if service hits this limits then it will be alarmed by the AWS CloudWatch.

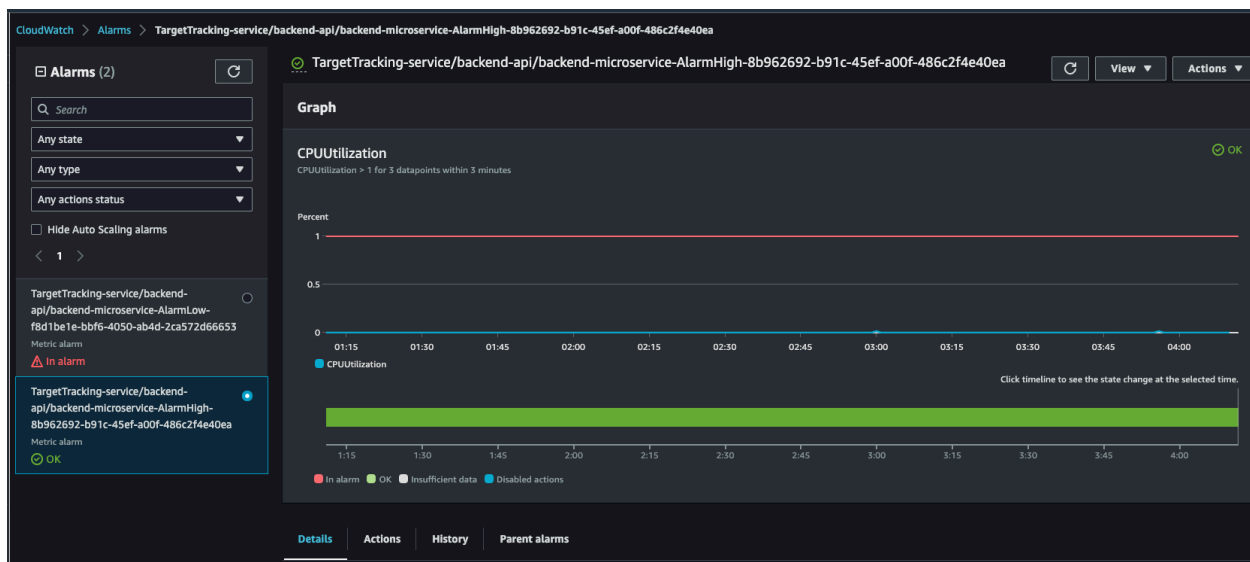


Figure 14: Screenshot of backend-microservice alarm in AWS CloudWatch