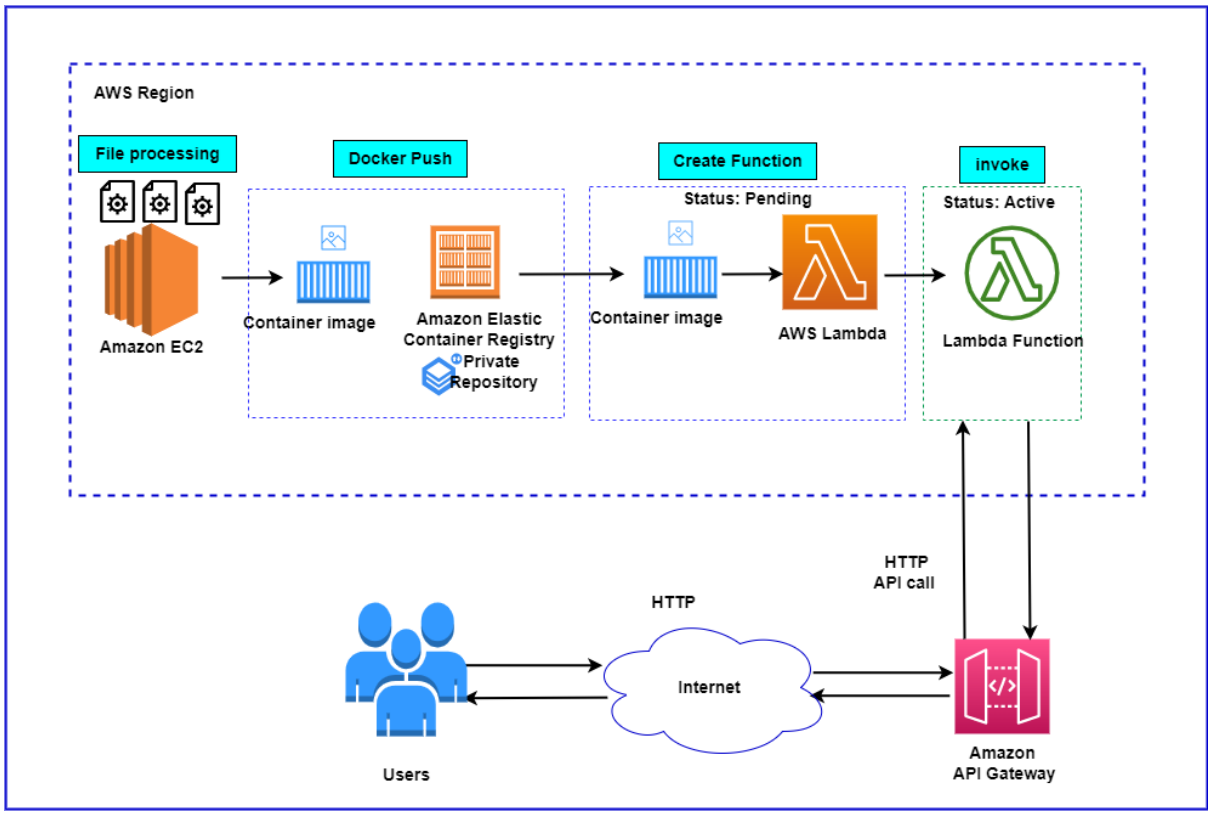
AWS Capstone Use Case 12

Containerizing Lambda deployments using OCI container images

Each use case report should have following sections:

Sections	Points
Implemented Solution Architecture diagram with detailed explanation for each component / service used.	8
Implementation Screenshot with brief description for every screenshot. Code / Script files (if applicable) *Screen shots must have your AWS / Azure account ID clearly visible along with your laptop's date and time*	8
Cost Analysis of the implemented Solution – Assume your solution is used by 1000 users for a month, and give monthly billing estimates.	2
Lessons & Observations	2
TOTAL (for 1 use case)	**20**

A. Implemented Solution Architecture diagram with detailed explanation for each component / service used.



Implemented Solution Architecture Diagram: Lambda Containerization

Steps for Lambda Containerization using OCI Images:

- Configure EC2
- Use SSH to securely login to EC2
- Securely copy files required to build docker image on EC2
- Configure ECR with a private repository to store docker image
- Push the Docker image from EC2 to a private repository in ECR
- Create Lambda function in ECR: Build and tag the lambda function
- Invoke the Lambda Function
- Access Lambda function using API Gateway

Components/Services Used:

- **Amazon EC2:** Compute service from AWS. Used as a local machine to process files required to build container image.
- **Amazon ECR:** used to store the container image in private repository
- **Amazon IAM:** Used for permissions
- **Amazon Lambda:** Serverless compute service from AWS. Used to deploy container image. There are 3 ways to create lambda function;1) author from scratch 2) use a blueprint and 3) use container image.
- **API Gateway:** To make deployed image accessible
- **Python and Text files required to build container image:**
    - **Requirements.txt file** contains all packages that are required in app.py file
    - **app.py file** imports all packages required to build lambda function. It defines a function lambda_handler. The handler function accepts two arguments; event and context. The event argument retrieves the data that is passed to execute the lambda function. For e.g. if the lambda function is triggered using HTTP API call via API Gateway then the event calls for all parameters required for HTTP API call (resource, path, method, resource path, headers etc). The context argument provides the data about the execution environment of the lambda function. The lambda_handler function must have a return value that depends on the invocation type. For example statusCode 200 is returned when lambda function is invoked using HTTP API call indicating that the request for invoking lambda is successful. StatusCode 200 is success response code. The meaning of a success depends on the HTTP request method: GET : The resource has been fetched and is transmitted in the message body (Hello from Lambda Containers).
    - **Dockerfile** contains **python 3.8** lambda base image which is picked up from public.ecr.aws. The Dockerfile installs all packages and dependencies included in requirements.txt file. It copies the app.py to the container and sets the command for app.handler.

Commands for Lambda Containerization:

user@user-HP-Pavilion-14-Notebook-PC:~/Downloads$ chmod 400 sk33.pem; providing read access to the owner of sk33.pem
user@user-HP-Pavilion-14-Notebook-PC:~/Downloads$ ssh -i "sk33.pem" ec2-user@ec2-3-82-206-232.compute-1.amazonaws.com; secure login to EC2

Building Docker Image on EC2:

```
[ec2-user@ip-172-31-88-107 ~]$ mkdir lambda-docker
[ec2-user@ip-172-31-88-107 ~]$ cd lambda-docker/
[ec2-user@ip-172-31-88-107 lambda-docker]$ ls

[ec2-user@ip-172-31-88-107 lambda-docker]$ vim app.py
[ec2-user@ip-172-31-88-107 lambda-docker]$ ls
app.py
[ec2-user@ip-172-31-88-107 lambda-docker]$ vim Dockerfile
[ec2-user@ip-172-31-88-107 lambda-docker]$ ls
app.py  Dockerfile
[ec2-user@ip-172-31-88-107 lambda-docker]$ vim requirements.txt
[ec2-user@ip-172-31-88-107 lambda-docker]$ ls
app.py  Dockerfile  requirements.txt

[ec2-user@ip-172-31-88-107 home]$ sudo apt-get update

[ec2-user@ip-172-31-88-107 home]$  sudo amazon-linux-extras install docker

[ec2-user@ip-172-31-88-107 home]$ sudo service docker start

[ec2-user@ip-172-31-88-107 home]$ sudo chkconfig  docker on

[ec2-user@ip-172-31-88-107 home]$ sudo yum install -y git

[ec2-user@ip-172-31-88-107 home]$ sudo curl -i -L
https://github.com/docker/compose/release/latest/download/docker-compose-
$(uname -s)-$(uname -m) -o /usr/local/bin/docker-compose

[ec2-user@ip-172-31-88-107 ~]$ sudo chmod +x /usr/local/bin/docker-compose
[ec2-user@ip-172-31-88-107 ~]$ docker-compose versionDocker Compose version
v2.11.0
[ec2-user@ip-172-31-88-107 ~]$ docker images
```

Commands to push build Docker image to ECR:

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 204298662877.dkr.ecr.us-east-1.amazonaws.com
```

Commands to run above aws ecr get-login-password command successfuly:

```
[ec2-user@ip-172-31-88-107 ~]$ sudo apt-get install gnupg2 pass
[ec2-user@ip-172-31-88-107 lambda-docker]$ sudo amazon-linux-extras install epel -y
[ec2-user@ip-172-31-88-107 ~]$ sudo apt install awscli -y

[ec2-user@ip-172-31-88-107 ~]$ pwd
/home/ec2-user
[ec2-user@ip-172-31-88-107 ~]$ cd .aws
[ec2-user@ip-172-31-88-107 .aws]$ ls
config  credentials
```

```
[ec2-user@ip-172-31-88-107 .aws]$ cd cerdentials
-bash: cd: cerdentials: No such file or directory
[ec2-user@ip-172-31-88-107 .aws]$ cat credentials
[default]
aws_access_key_id =
[[ec2-user@ip-172-31-88-107 .aws]$ vi credentials
[ec2-user@ip-172-31-88-107 .aws]$ cat credentials
[default]
aws_access_key_id = AKIAS7EJGGPOY4PQ63GB
aws_secret_access_key = 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
[ec2-user@ip-172-31-88-107 .aws]$ cd ..
[ec2-user@ip-172-31-88-107 ~]$ pwd
/home/ec2-user
```

Commands to build and tag the docker image:

```
docker build -t docker-lambda .

docker tag docker-lambda:latest 204298662877.dkr.ecr.us-east-
1.amazonaws.com/docker-lambda:latest
```

Command to push the built docker image to private repository in ECR:

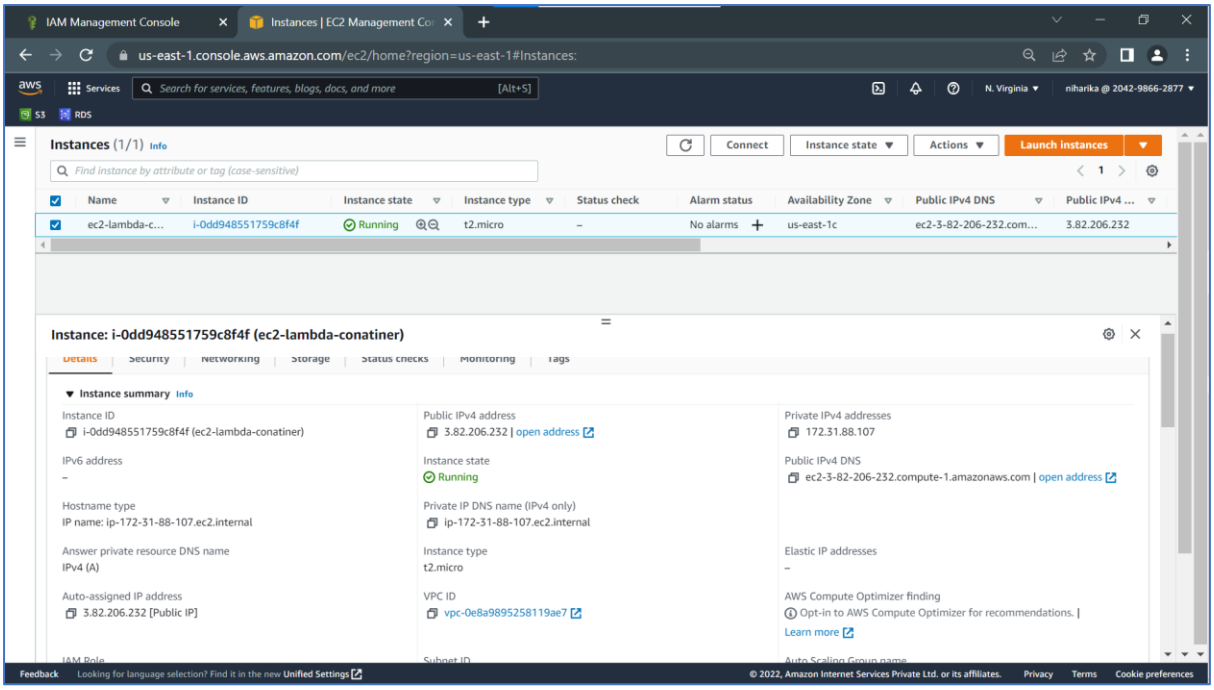
```
docker push 204298662877.dkr.ecr.us-east-1.amazonaws.com/docker-lambda:latest
```

Successful implementation of Docker image Push commands:

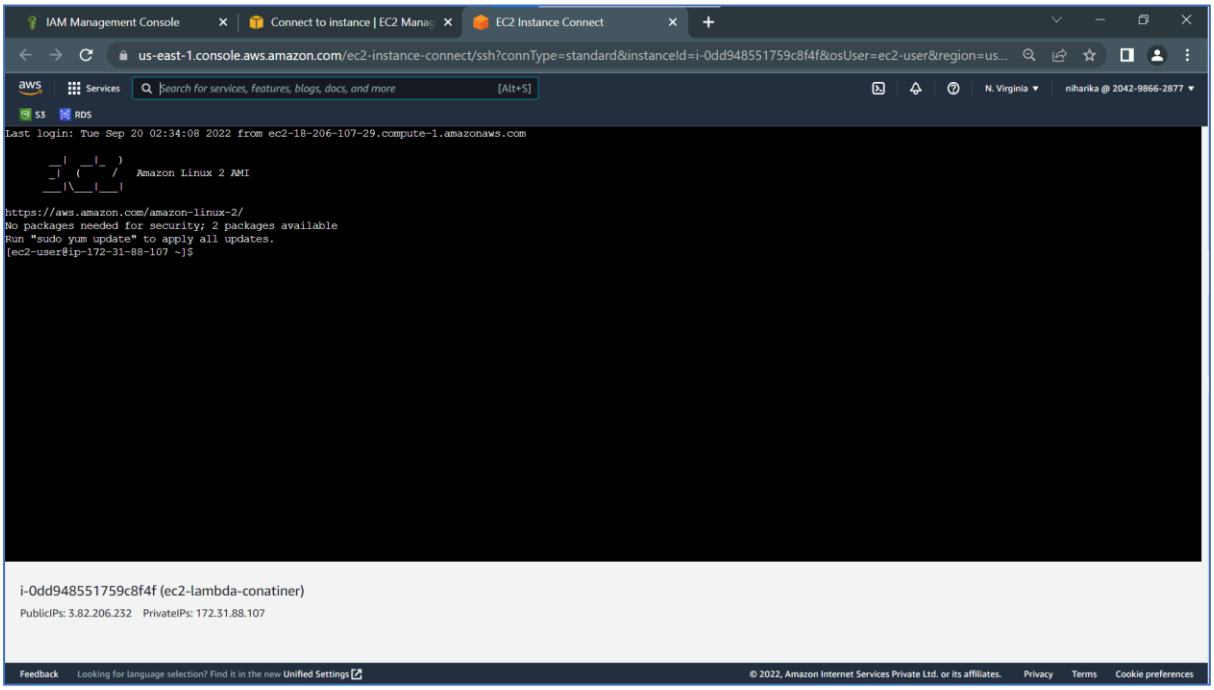
```
[ec2-user@ip-172-31-88-107 lambda-docker]$

[ec2-user@ip-172-31-88-107 ~]$ pwd
/home/ec2-user
[ec2-user@ip-172-31-88-107 ~]$ cd .aws
[ec2-user@ip-172-31-88-107 .aws]$ ls
config  credentials

[ec2-user@ip-172-31-88-107 .aws]$ cat credentials
[default]
aws_access_key_id =
[[ec2-user@ip-172-31-88-107 .aws]$ vi credentials
[ec2-user@ip-172-31-88-107 .aws]$ cat credentials
[default]
aws_access_key_id = AKIAS7EJGGPOY4PQ63GB
aws_secret_access_key = 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
[ec2-user@ip-172-31-88-107 .aws]$ cd ..
[ec2-user@ip-172-31-88-107 ~]$ pwd
/home/ec2-user
[ec2-user@ip-172-31-88-107 lambda-docker]$ aws configure
AWS Access Key ID [None]: AKIAS7EJGGPOY4PQ63GB
AWS Secret Access Key [None]: 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
Default region name [None]: us-east-1
Default output format [None]: JSON
```

```
[ec2-user@ip-172-31-88-107 ~]$ aws ecr get-login-password --region us-east-1 | docker
login --username AWS --password-stdin 204298662877.dkr.ecr.us-east-
1.amazonaws.com
WARNING! Your password will be stored unencrypted in /home/ec2-
user/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
[ec2-user@ip-172-31-88-107 ~]$ docker build -t docker-lambda .
Sending build context to Docker daemon  5.602MB
Step 1/4 : FROM python:alpine
 ---> 4da4c1dc8c72
Step 2/4 : COPY ./content .
 ---> Using cache
 ---> d98f96cdbeac
Step 3/4 : RUN pip install -r requirements.txt
 ---> Using cache
 ---> 205c0cc68ce1
Step 4/4 : CMD python3 bootstrap.py
 ---> Using cache
 ---> a04ad5caa96a
Successfully built a04ad5caa96a
Successfully tagged docker-lambda:latest
[ec2-user@ip-172-31-88-107 ~]$ docker tag docker-lambda:latest
204298662877.dkr.ecr.us-east-1.amazonaws.com/docker-lambda:latest
[ec2-user@ip-172-31-88-107 ~]$ docker push 204298662877.dkr.ecr.us-east-
1.amazonaws.com/docker-lambda:latest
The push refers to repository [204298662877.dkr.ecr.us-east-1.amazonaws.com/docker-
lambda]
67678fa66cbe: Pushed
e2cd8f647283: Pushed
804e16ced35b: Pushed
71aefba622a6: Pushed
765276be336a: Pushed
bdd2dbc0f630: Pushed
994393dc58e7: Pushed
latest: digest:
sha256:f58270bbcb6a6c97c2c28d6fc399f49e9c38715c7387ea03c0f8afa88c5b7ae6 size:
1787
[ec2-user@ip-172-31-88-107 ~]$
```

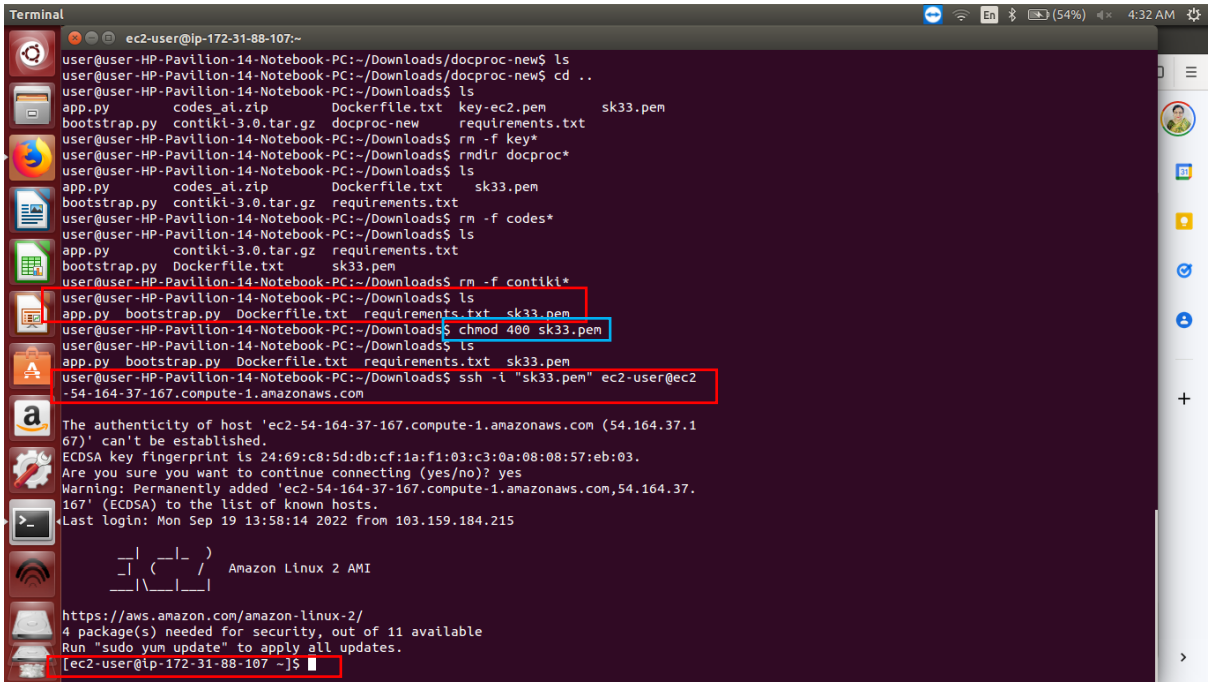
B. Implementation Screenshot with brief description for every screenshot.
Code/ Script files (if applicable) Screen shots must have your AWS / Azure account ID clearly visible along with your laptop's date and time



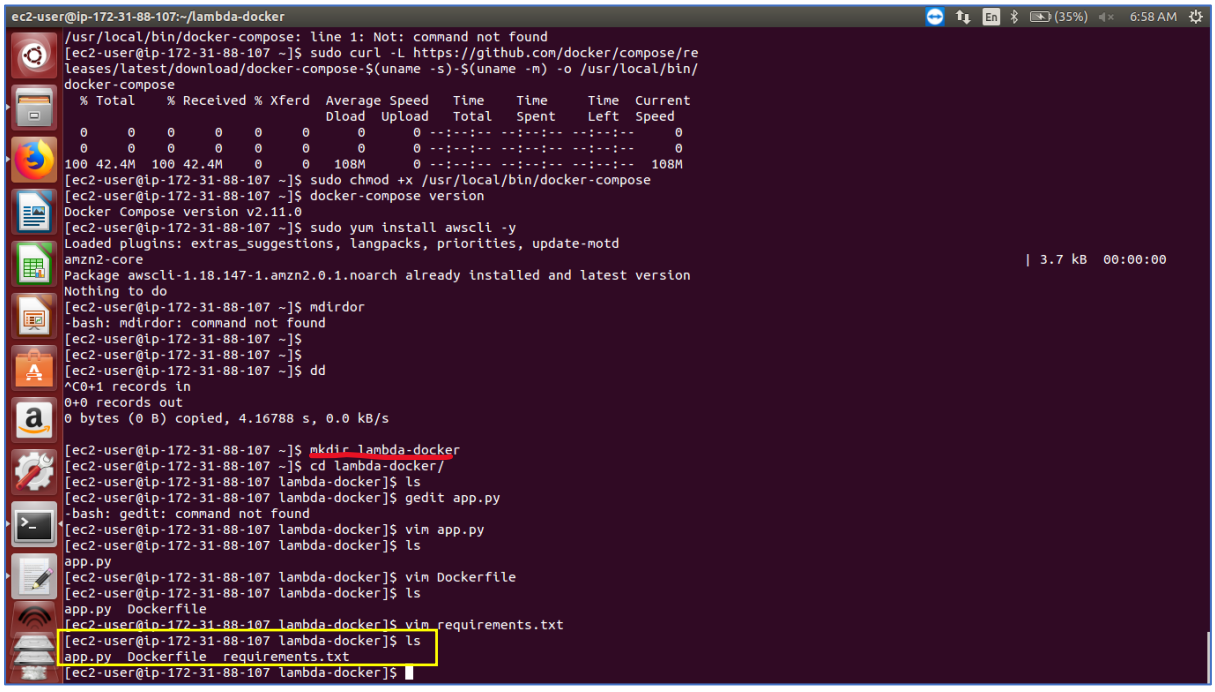
1. Successful creation of EC2 instance: Amazon linux, t2.micro, public IP 3.82.206.232 and sk33.pem (instance state running)



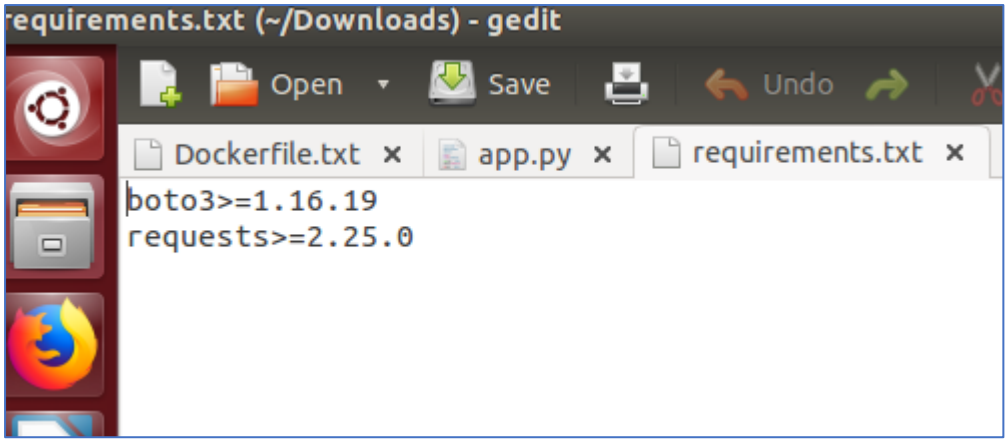
2. Successfully logged into EC2 from AWS console



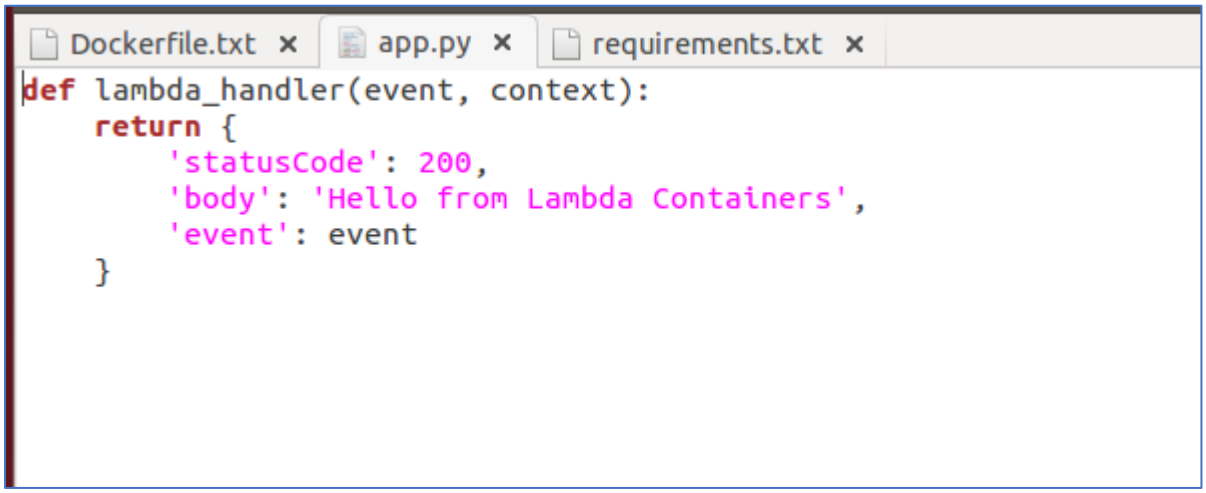
3. Kept all files required for project in downloads folder. Assigned read permission to sk33.pem file. Successfully logged into EC2 using Ubuntu terminal.



4. Created lambda-docker directory in EC2 and copied all required files (app.py, Dockerfile and reuirements.txt) in the directory

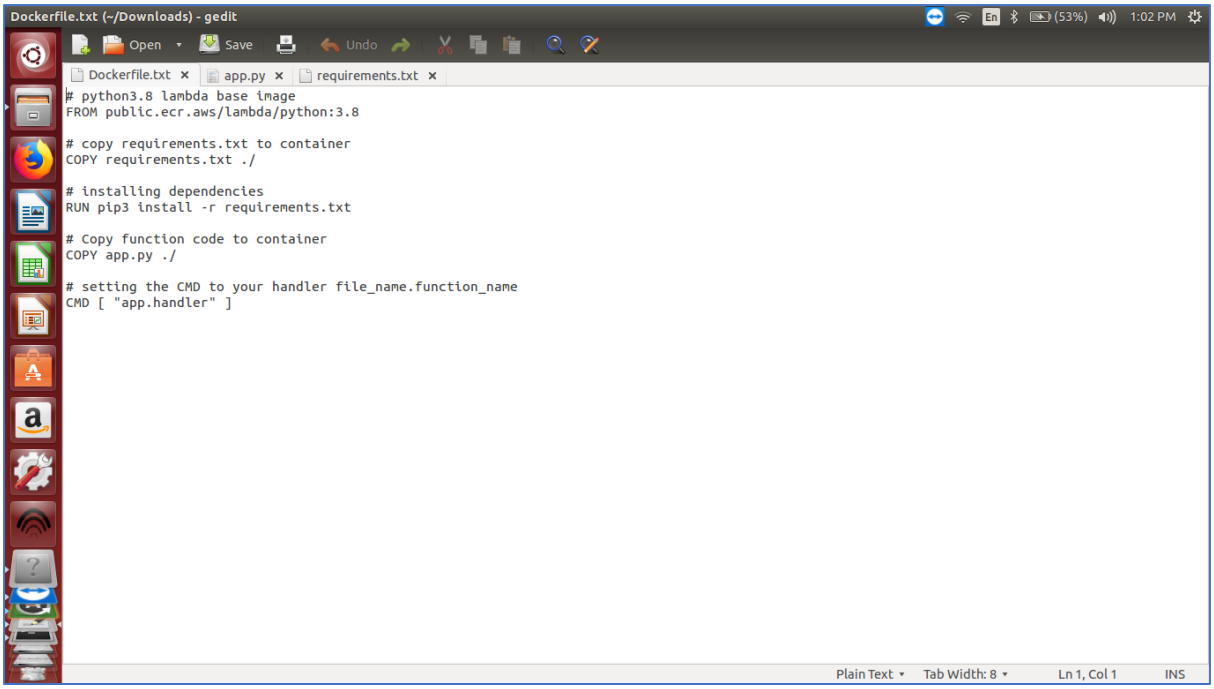


5. Requirements.txt file contains all packages that are required in app.py file



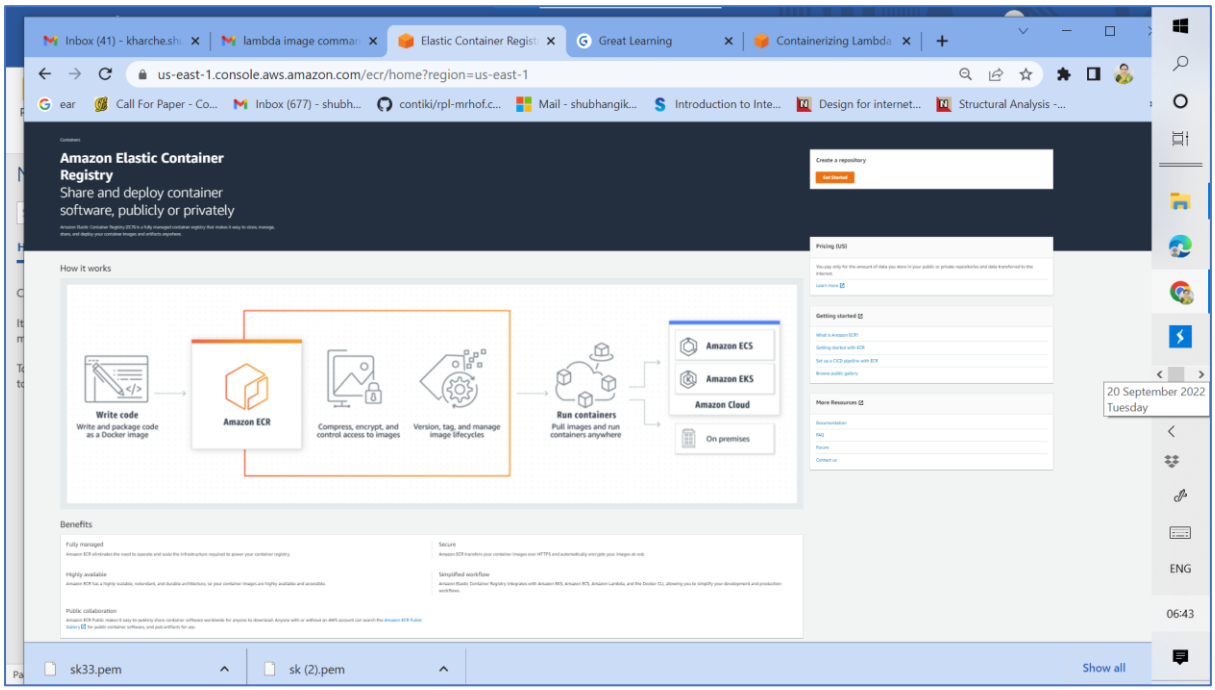
6. App.py file

app.py file imports all packages required to build lambda function. It defines a function lambda_handler. The handler function accepts two arguments; event and context. The event argument retrieves the data that is passed to execute the lambda function. For e.g. if the lambda function is triggered using HTTP API call via API Gateway then the event calls for all parameters required for HTTP API call (resource, path, method, resource path, headers etc). The context argument provides the data about the execution environment of the lambda function. The lambda_handler function must have a return value that depends on the invocation type. For example statusCode 200 is returned when lambda function is invoked using HTTP API call indicating that the request for invoking lambda is successful. StatusCode 200 is success respove code. The meaning of a success depends on the HTTP request method: GET : The resource has been fetched and is transmitted in the message body (Hello from Lambda Containers).

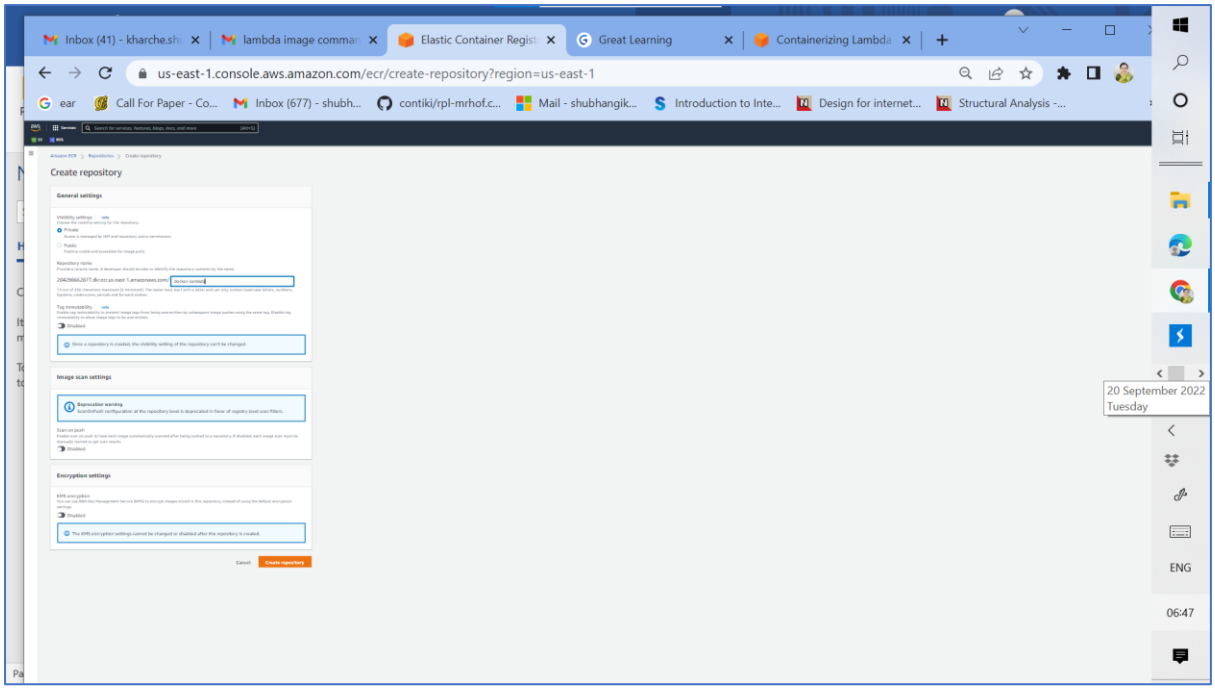


7. Dockerfile

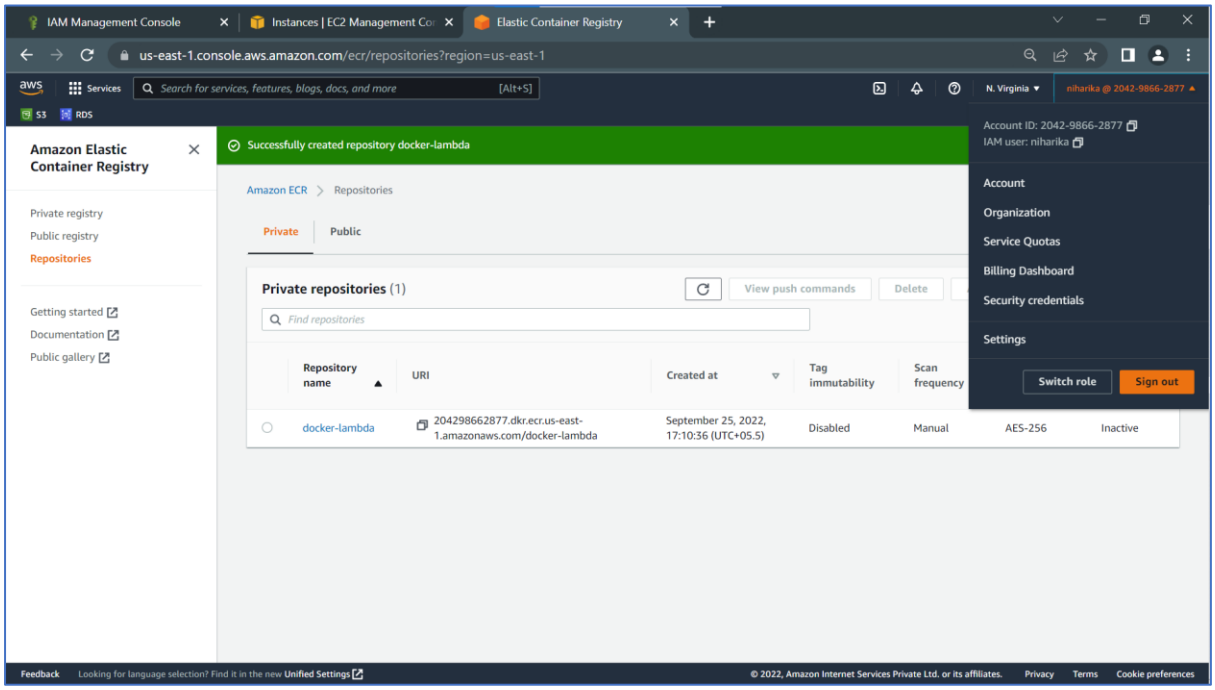
Dockerfile contains python 3.8 lambda base image which is picked up from public.ecr.aws. The Dockerfile installs all packages and dependencies included in requirements.txt file. It copies the app.py to the container and sets the command for app.handler.



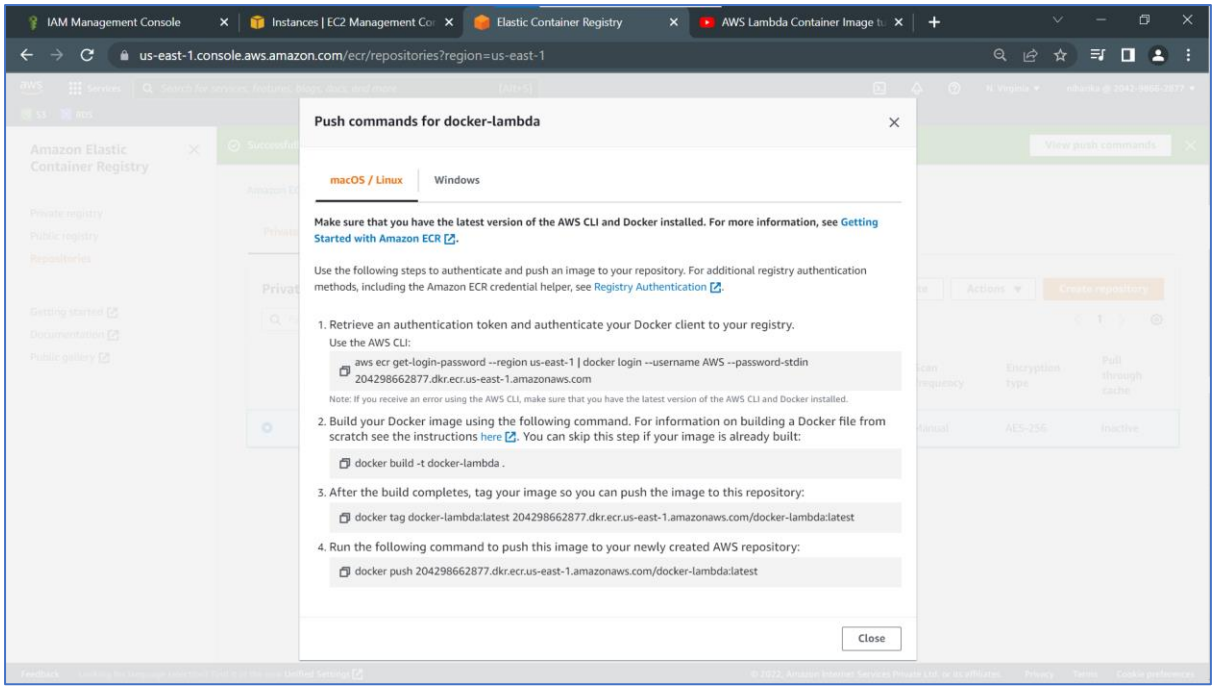
8. Browse Amazon ECR in AWS console



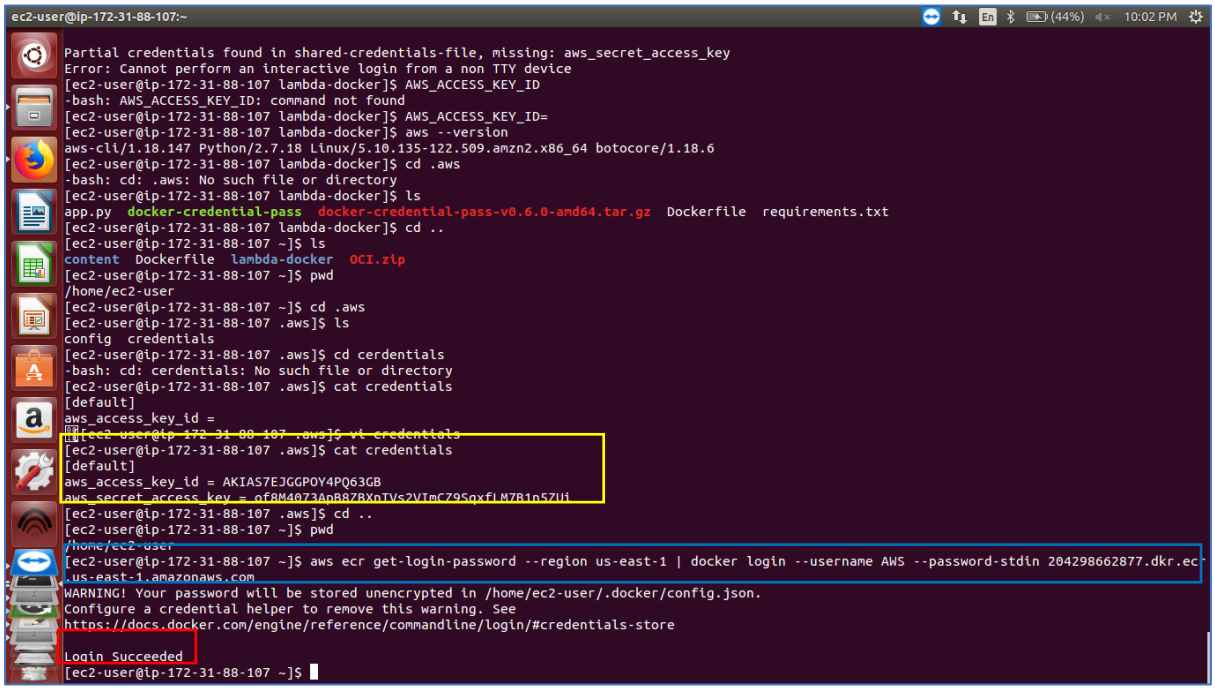
9. Create private repository named docker-lambda in Amazon ECR



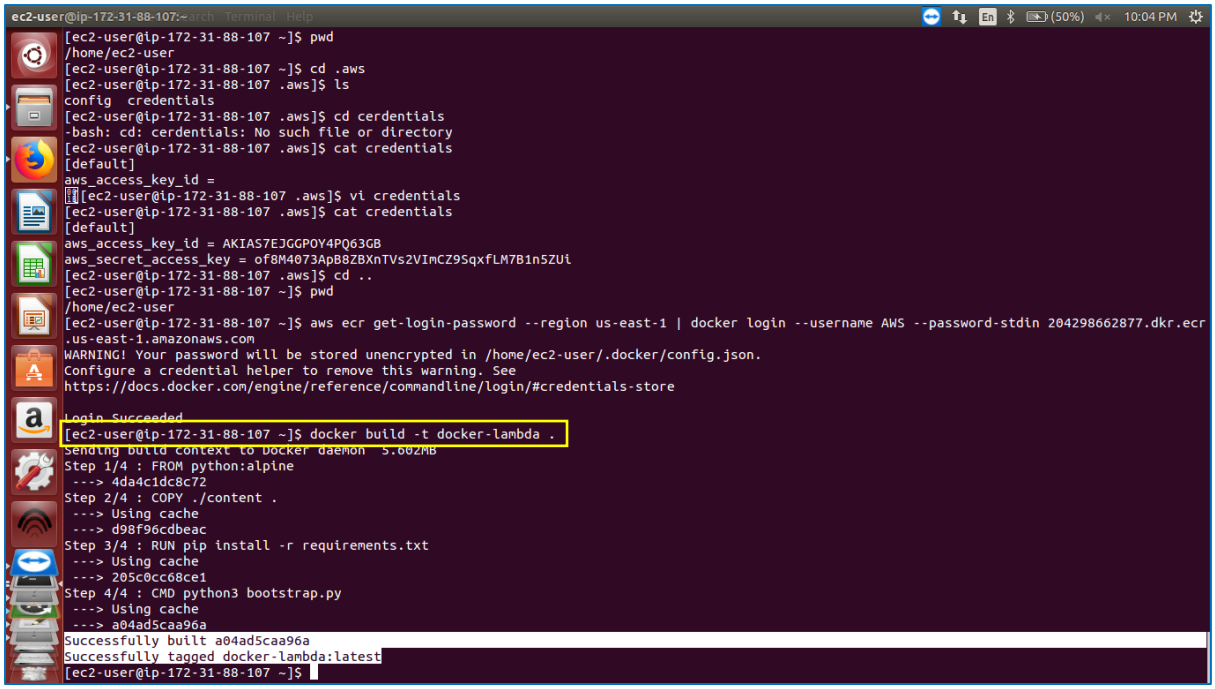
10. Successful creation of private repository named docker-lambda



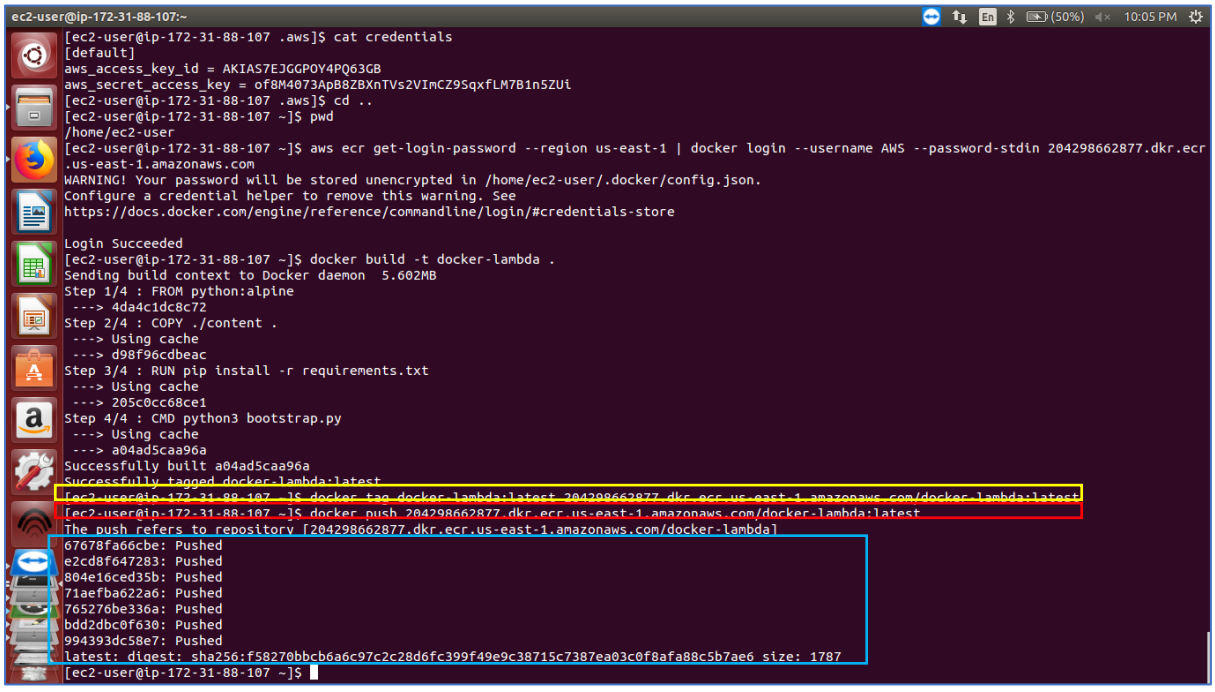
11. Push Commands for docker-lambda repository



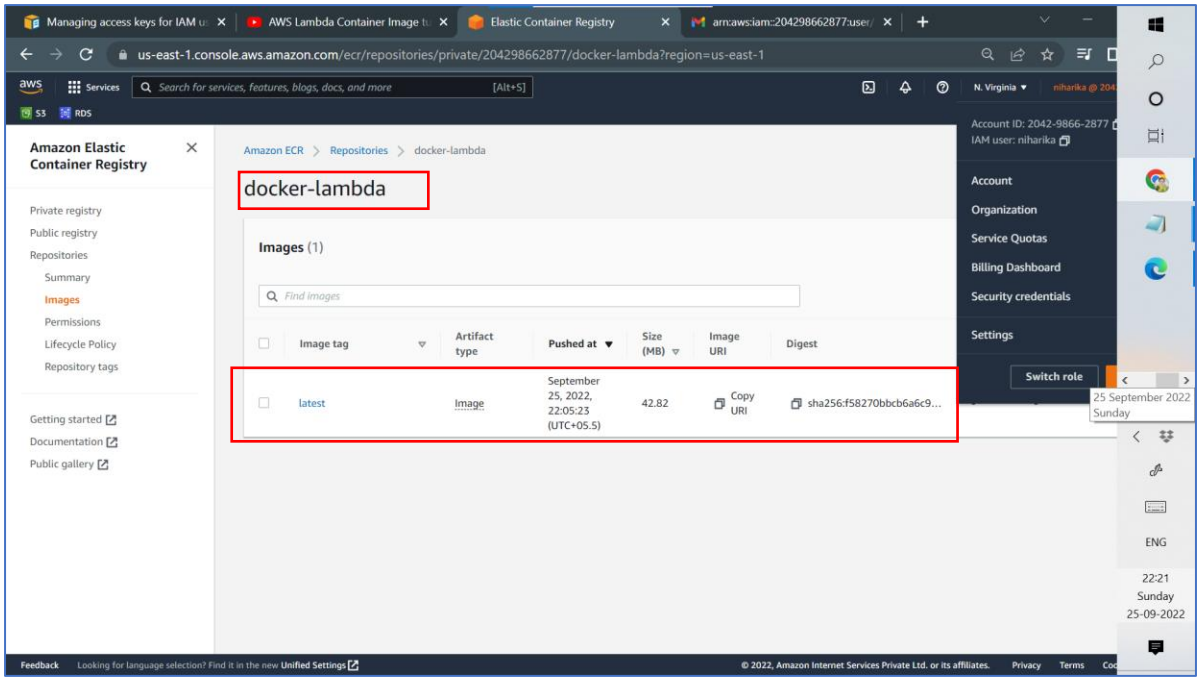
12. Authenticating docker client to docker registry. Authentication successful. Login succeeded



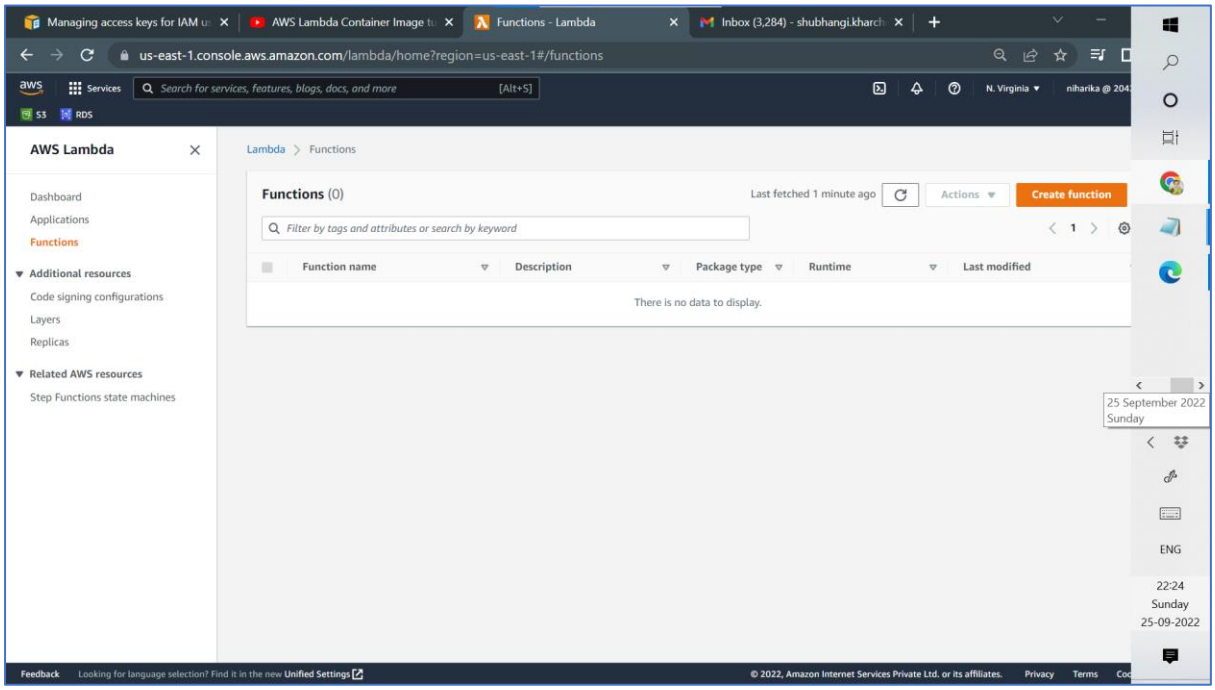
13. Successfully built and tagged docker image

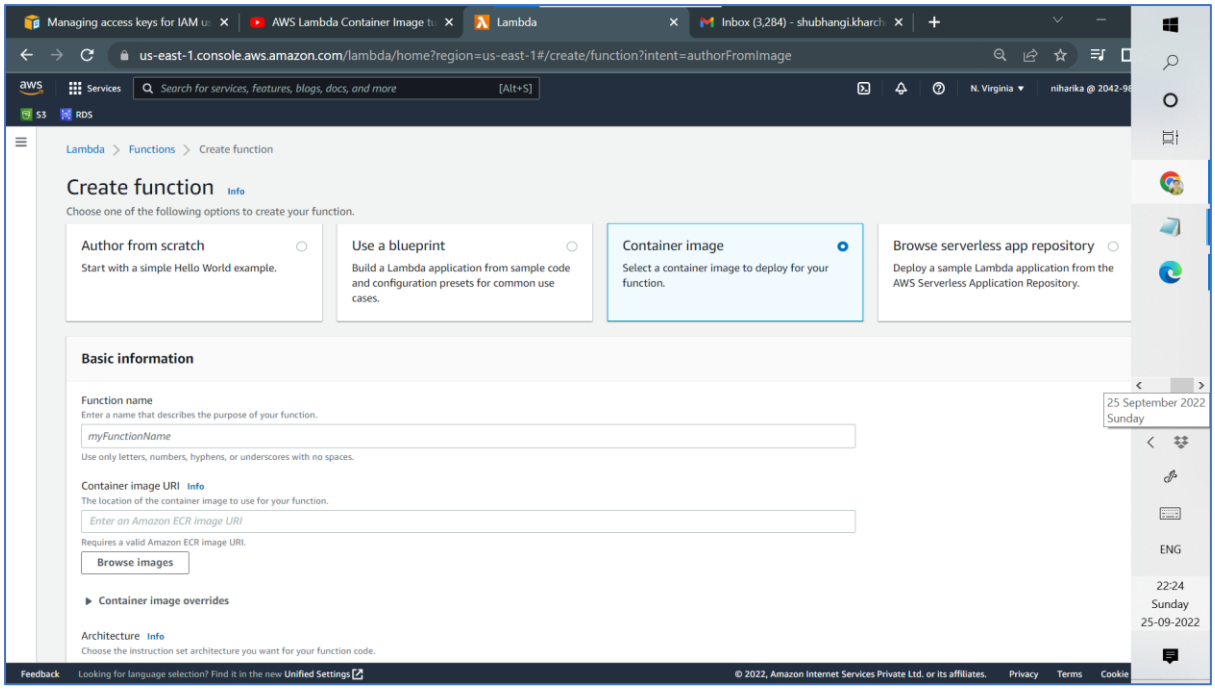


14. Successfully tagged and pushed docker image to docker-lambda repository

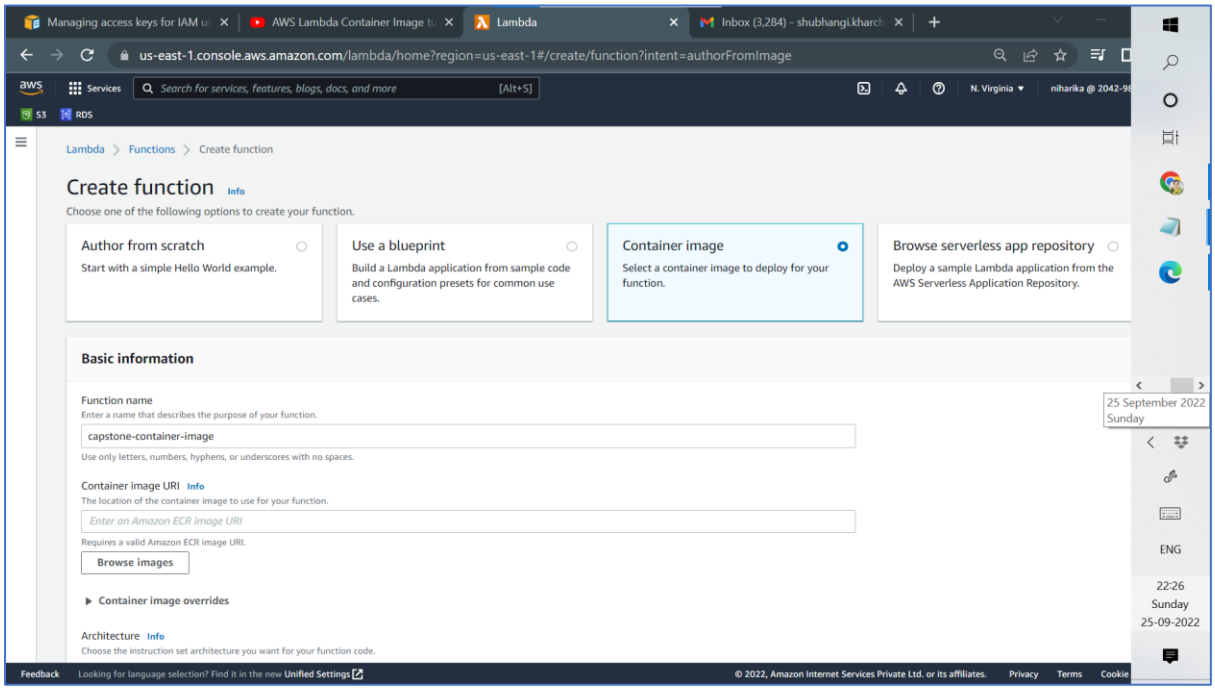


15. Verified that the docker image tagged latest is available in docker-lambda repository

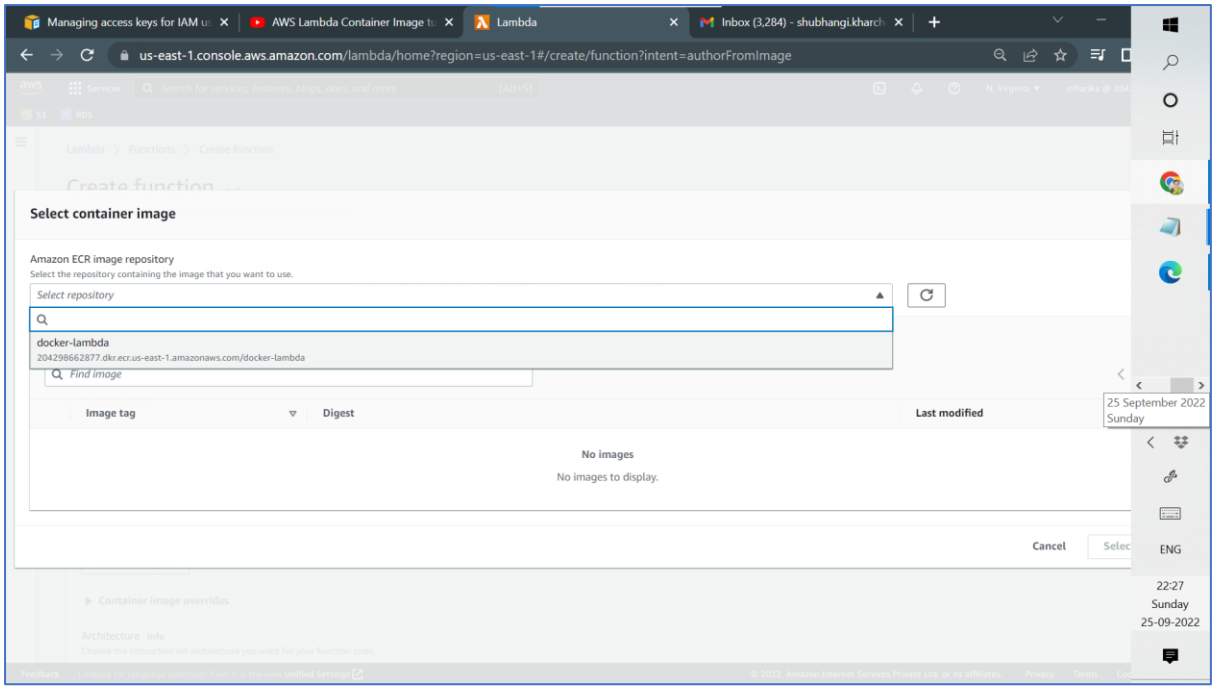




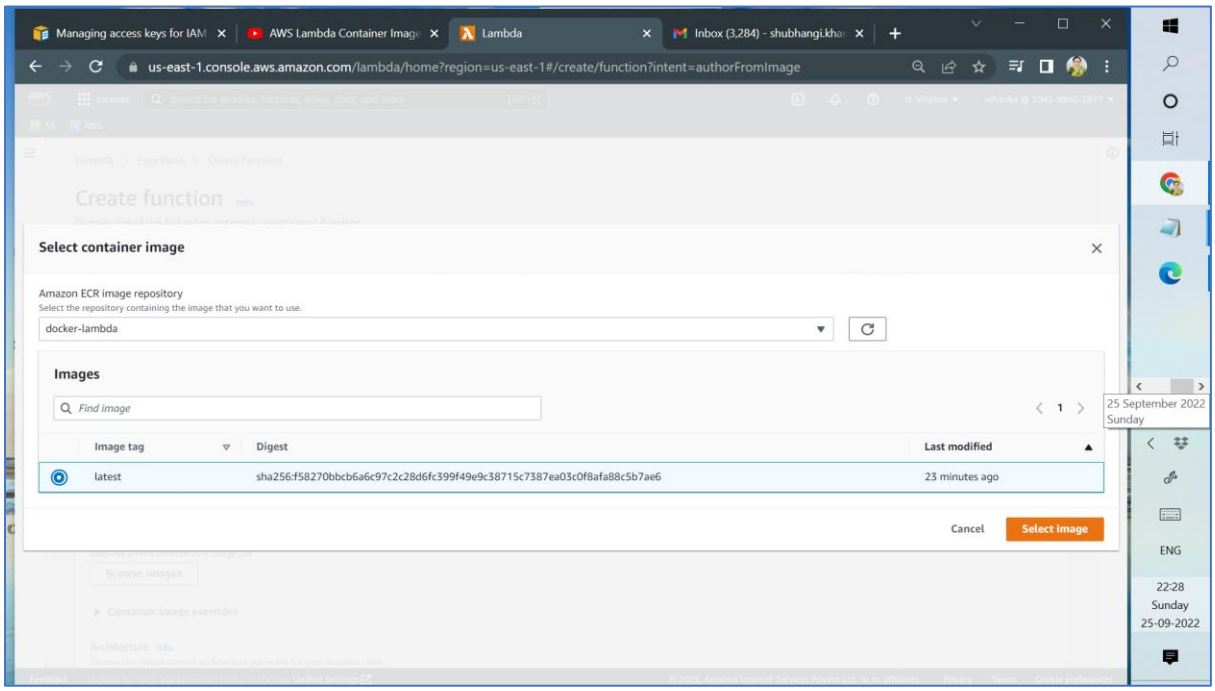
16. Browse lambda from AWS console and create lambda function



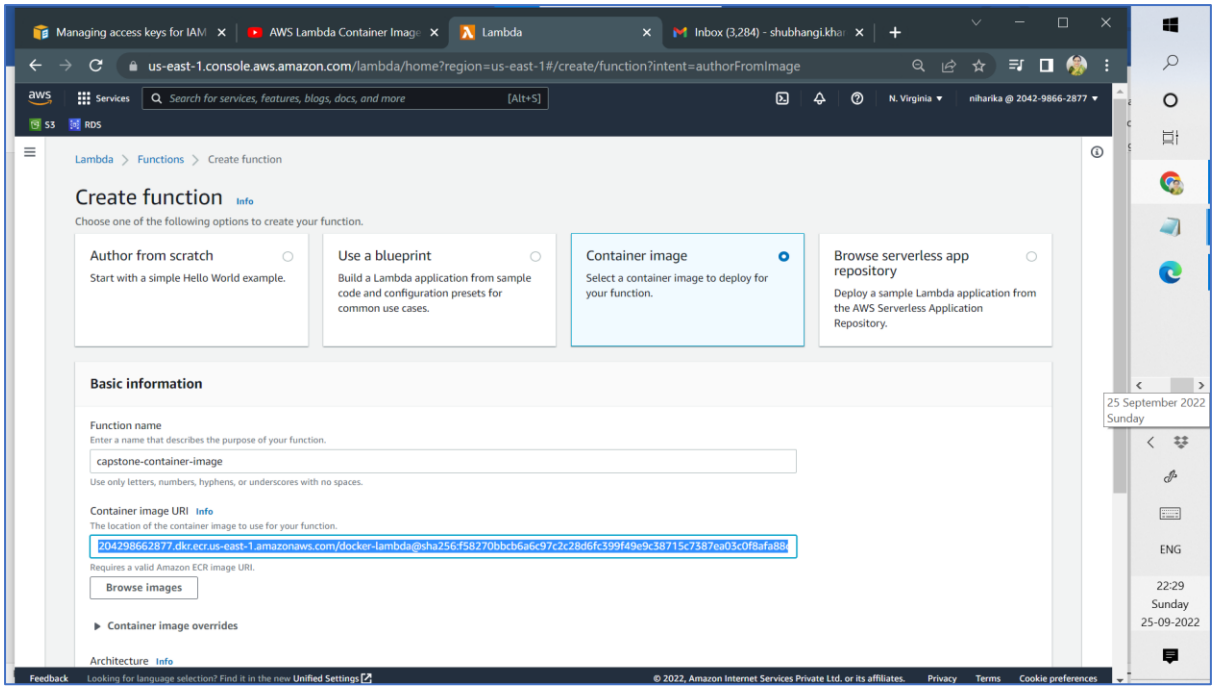
17. Create lambda function using container image



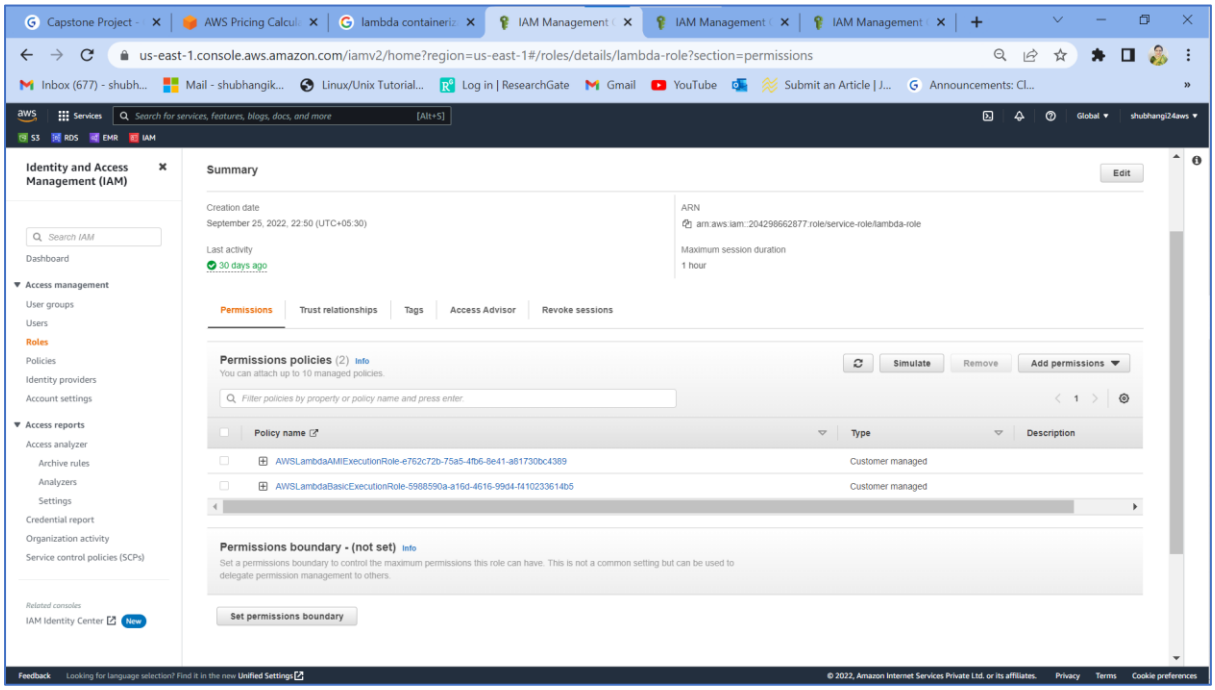
18. Browse the container image from docker-lambda repository



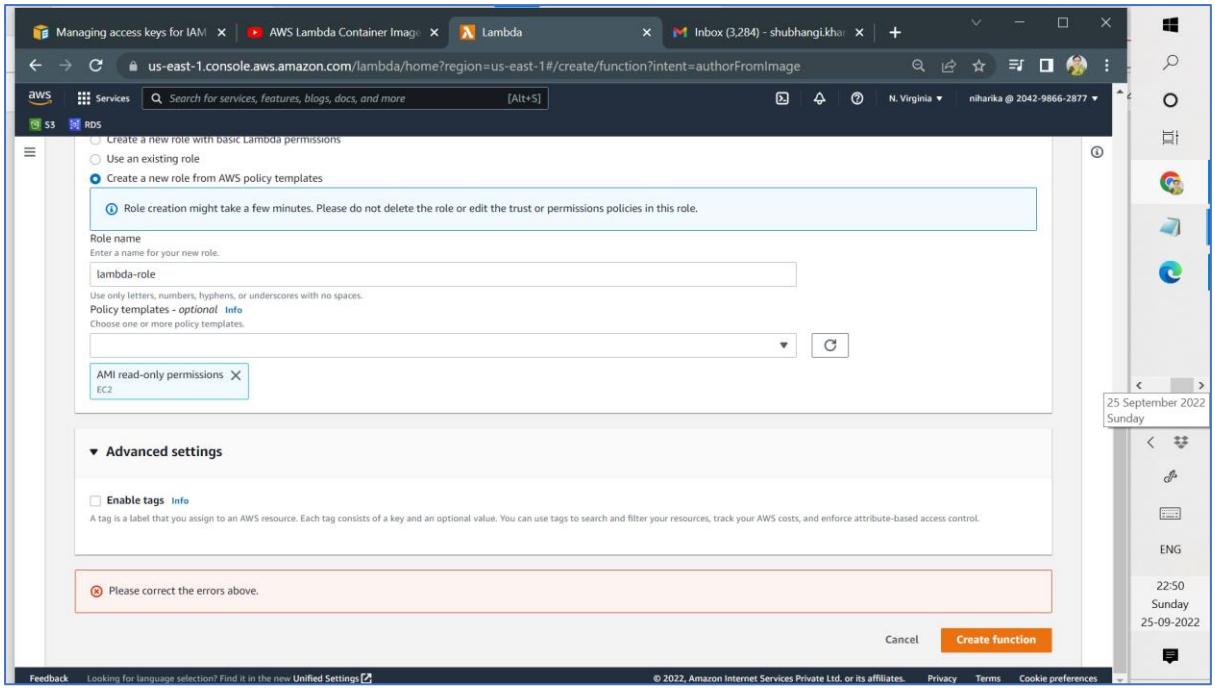
19. Select the container image from docker-lambda repository



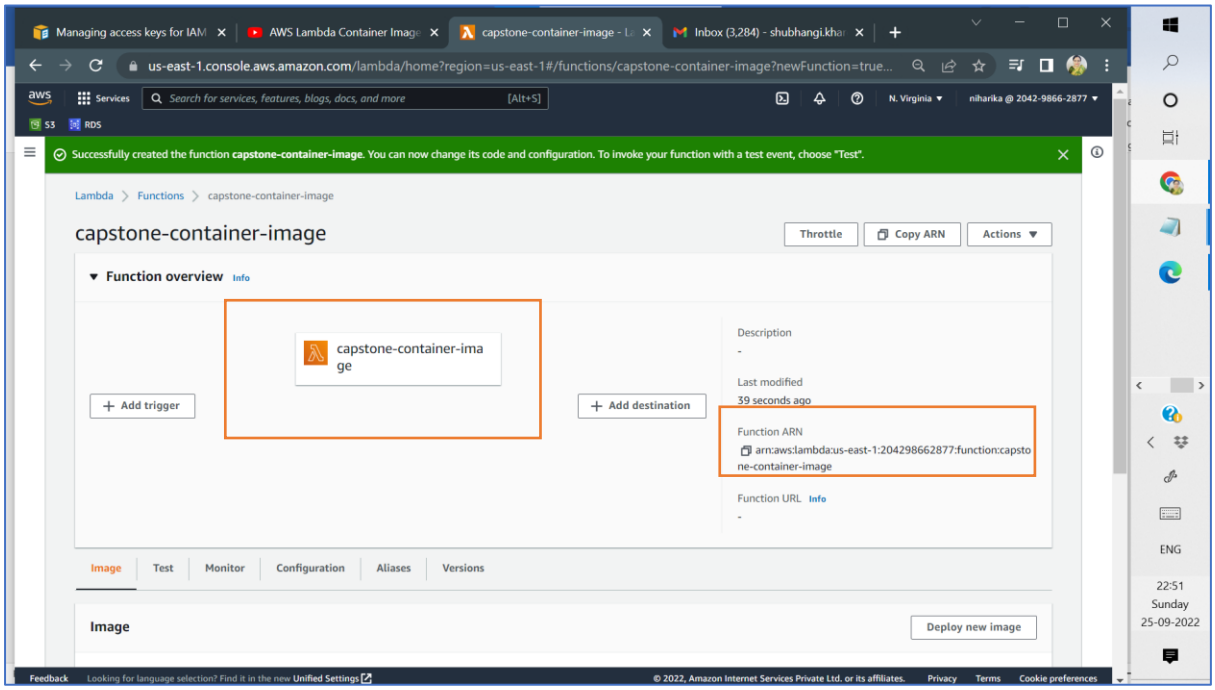
20. Provide container image URI



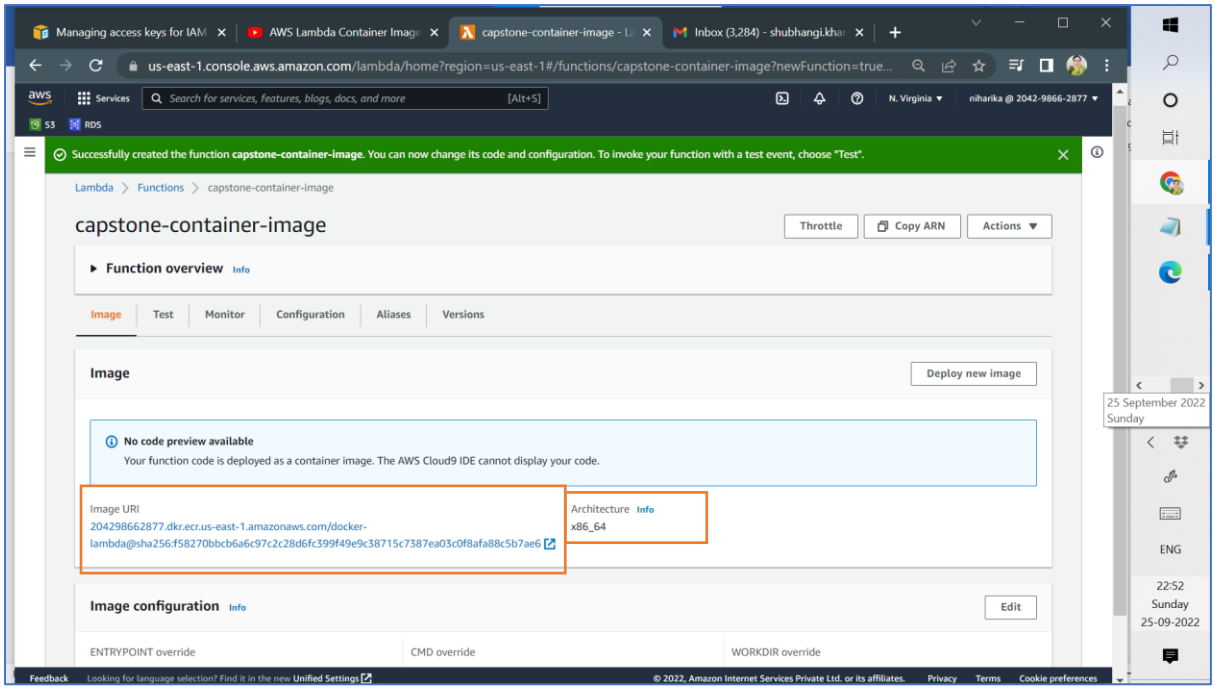
21. Separately create IAM role to provide read permissions from EC2 to lambda function



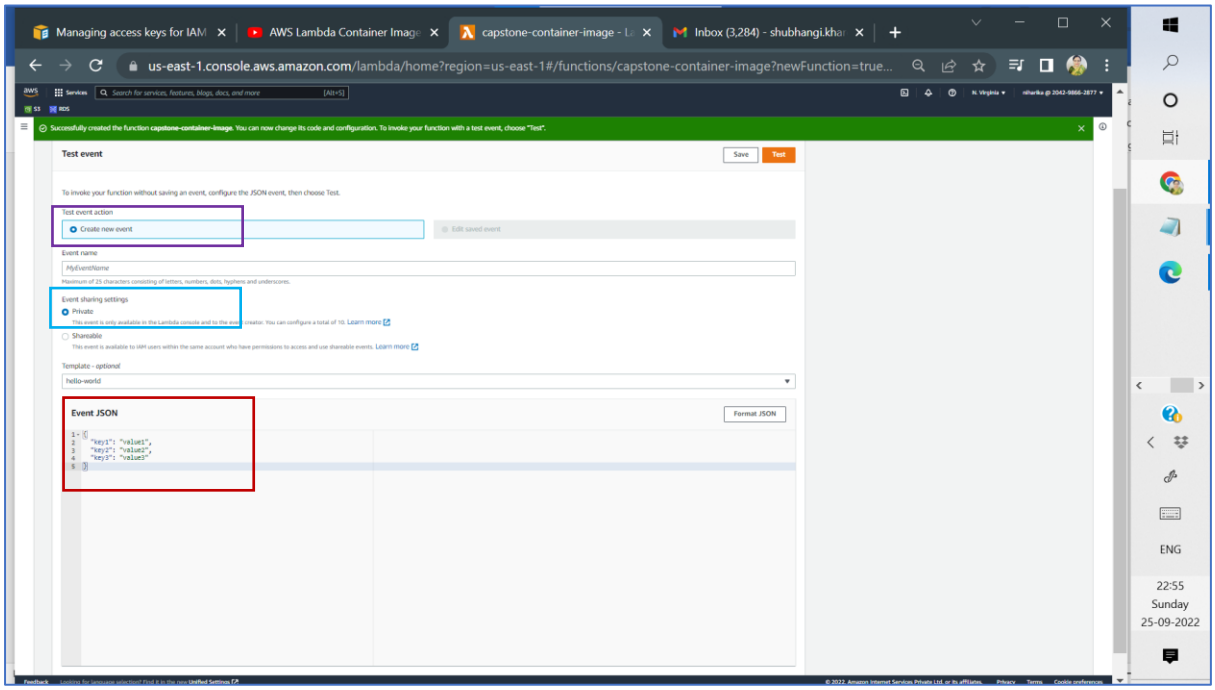
22. Enter created role name and hit create function



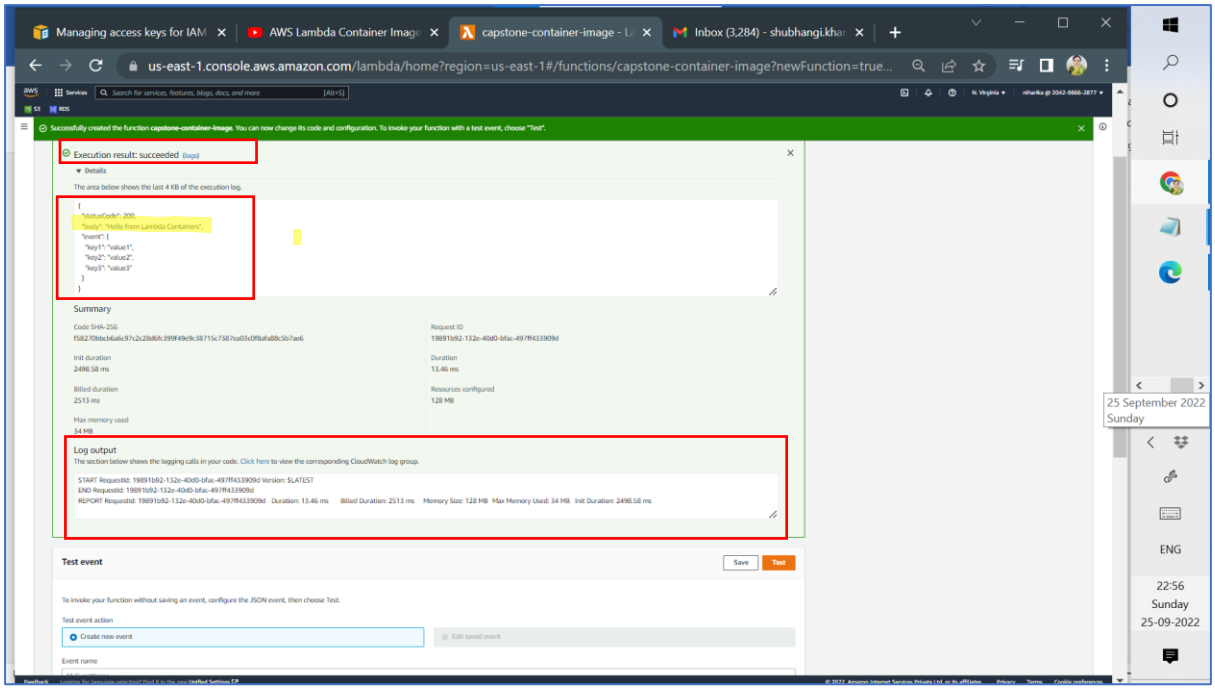
23. Successfully created lambda function named capstone-container-image



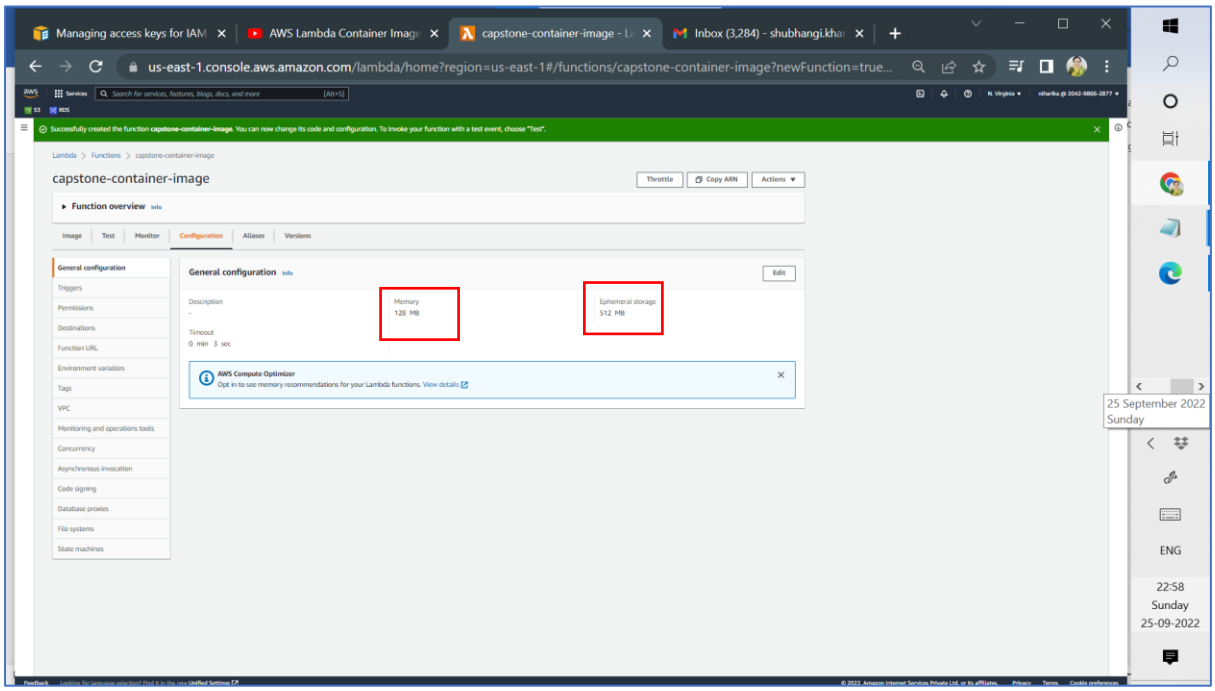
24. Lamba function capstone-container-image is created using x86_64 architecture



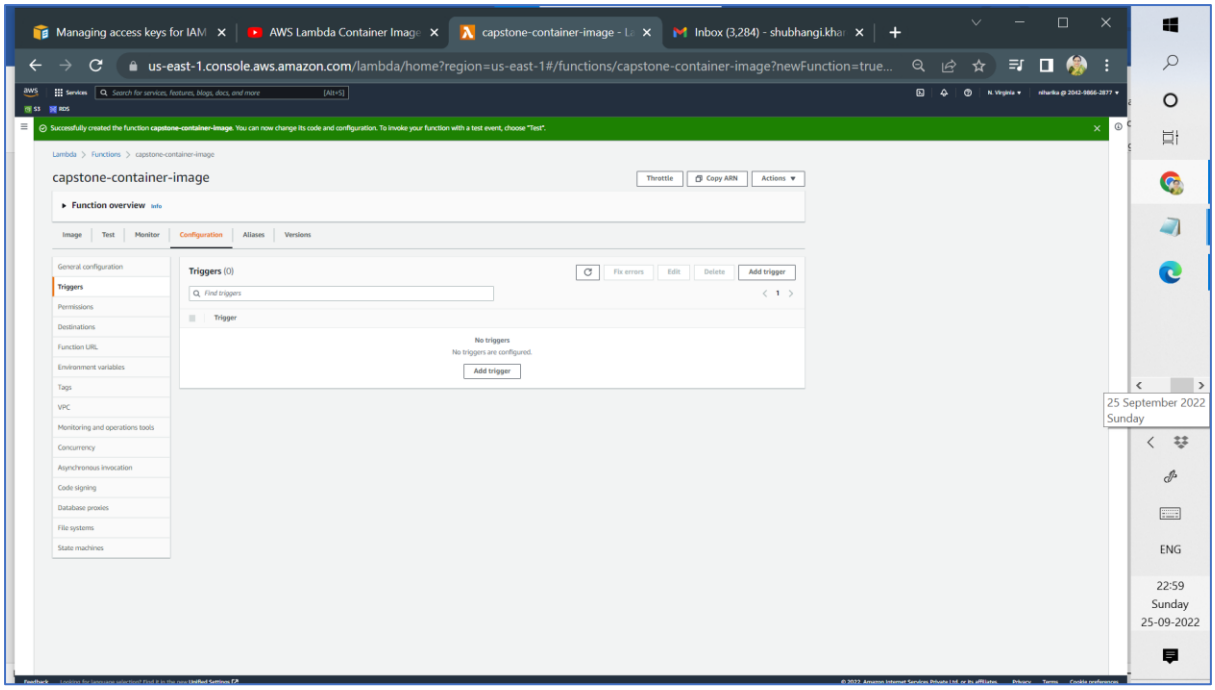
25. Invoking lambda function using JSON event. Test event provides JSON input to the lambda function



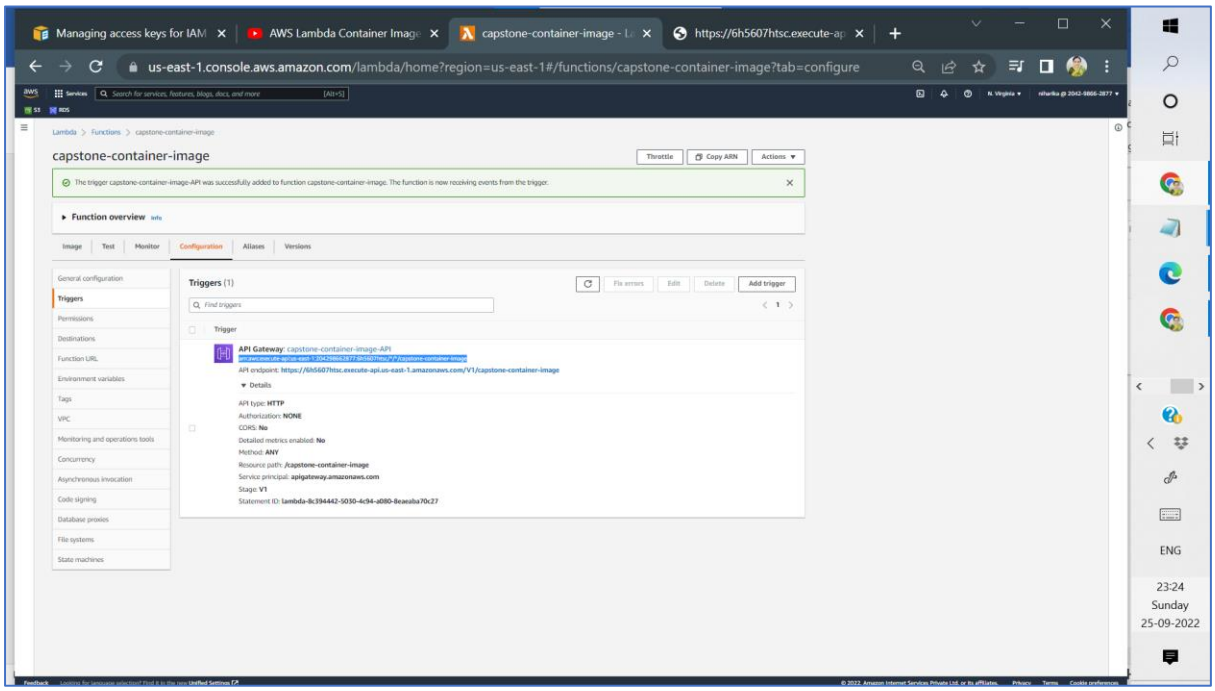
26. Test event execution is successful. Status code 200 indicates that the response has been fetched successfully and added to the message body, Lambda function is invoked successfully. Note the message body includes the message “Hello from lambda containers” which is part of lambda_handler function event attribute in app.py file.



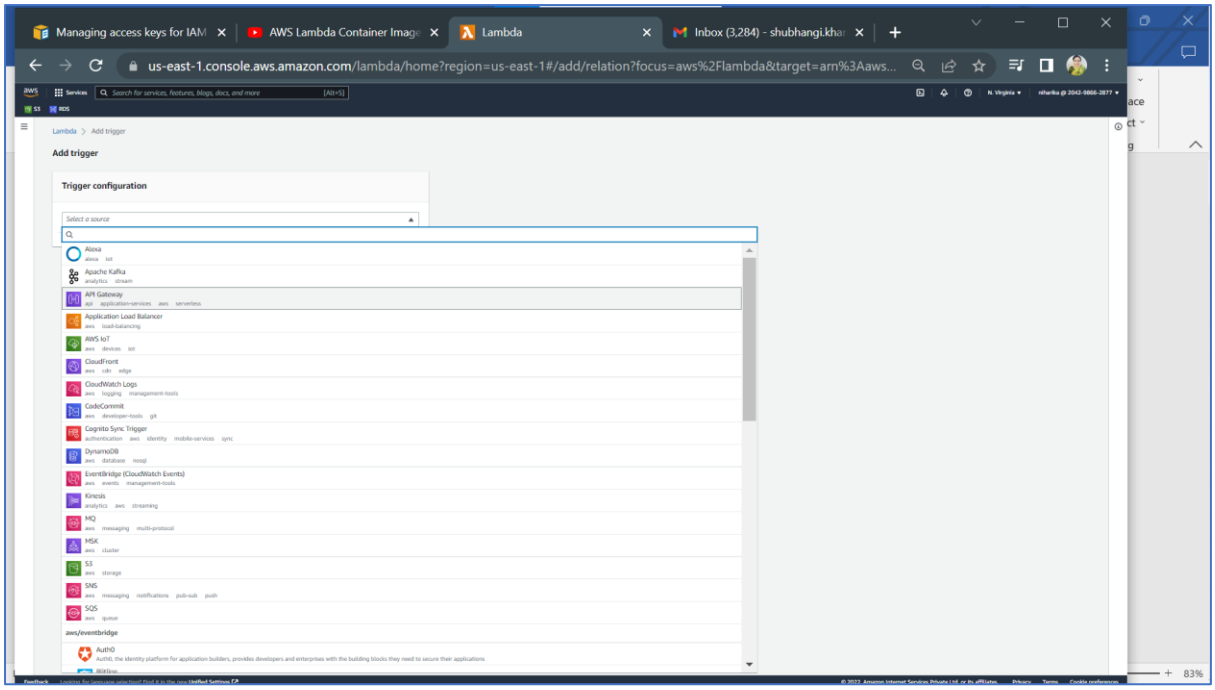
27. Lambda function capstone-container-image is configured to 128MB memory and 512 MB ephemeral storage



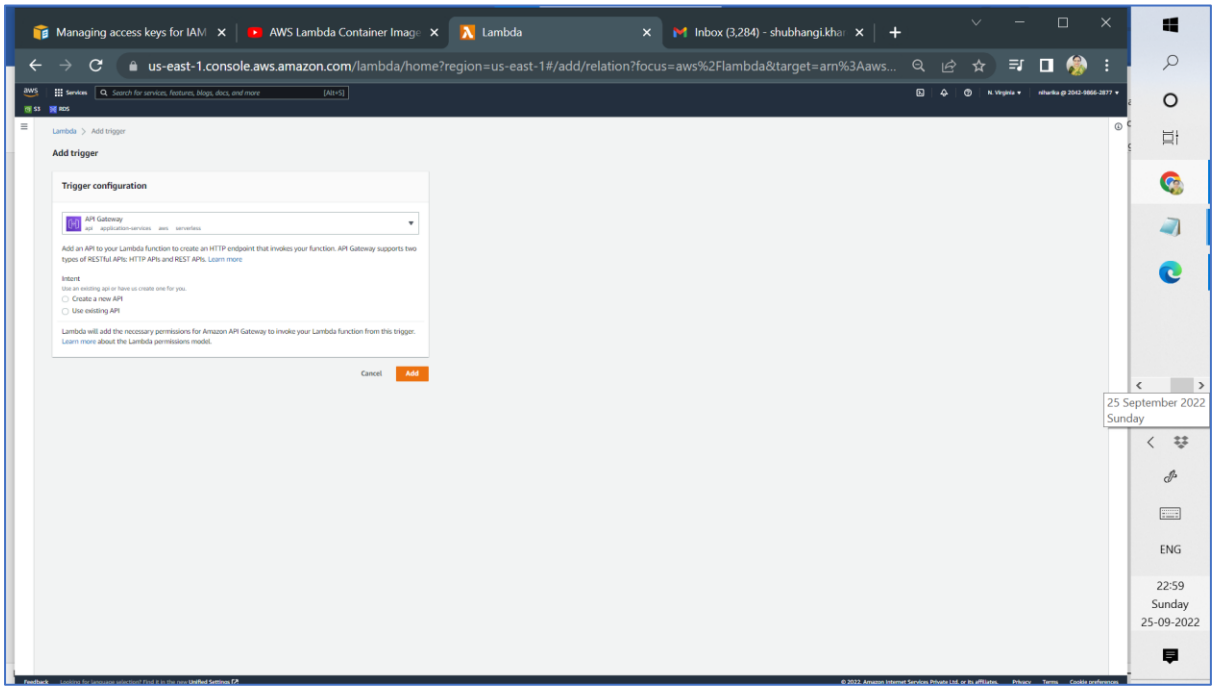
28. Adding trigger to lambda function



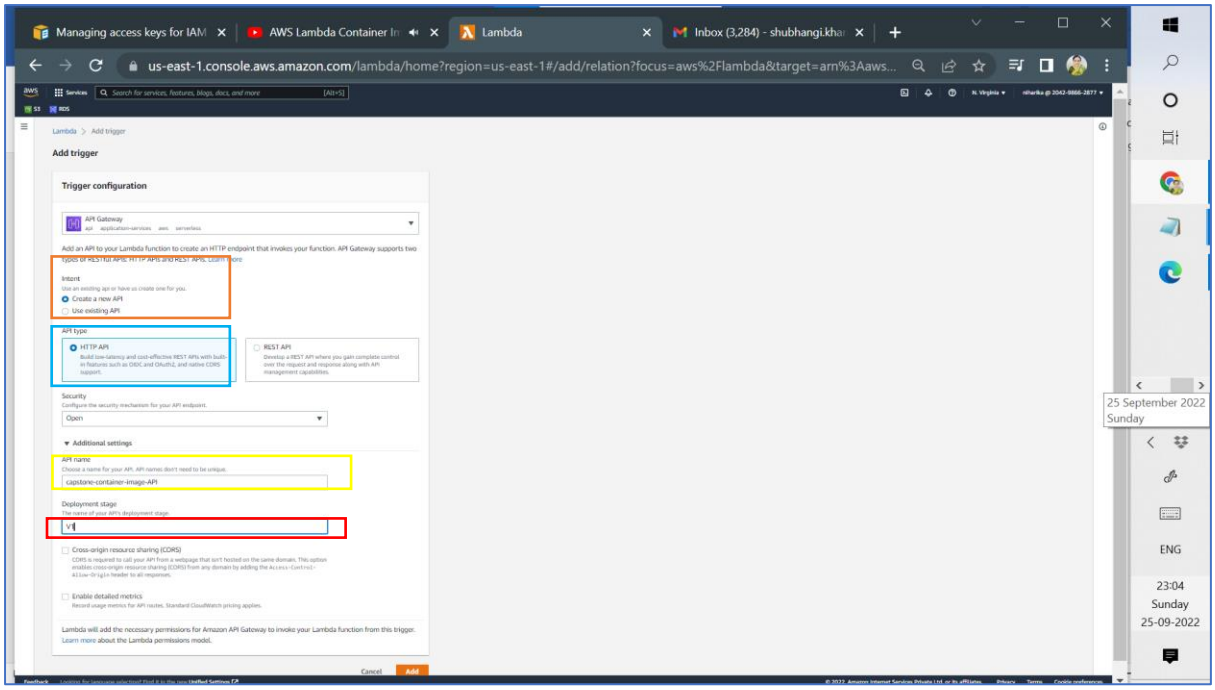
29. Adding trigger to lambda function using API Gateway



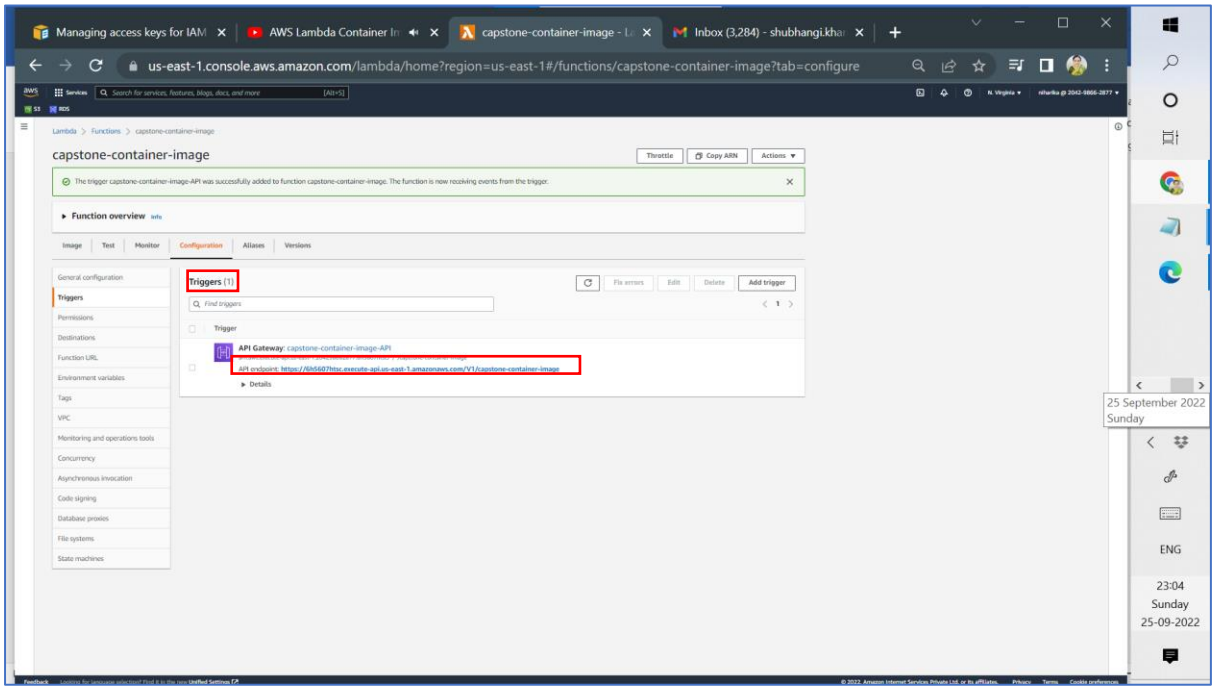
30. Selecting trigger configuration as API Gateway



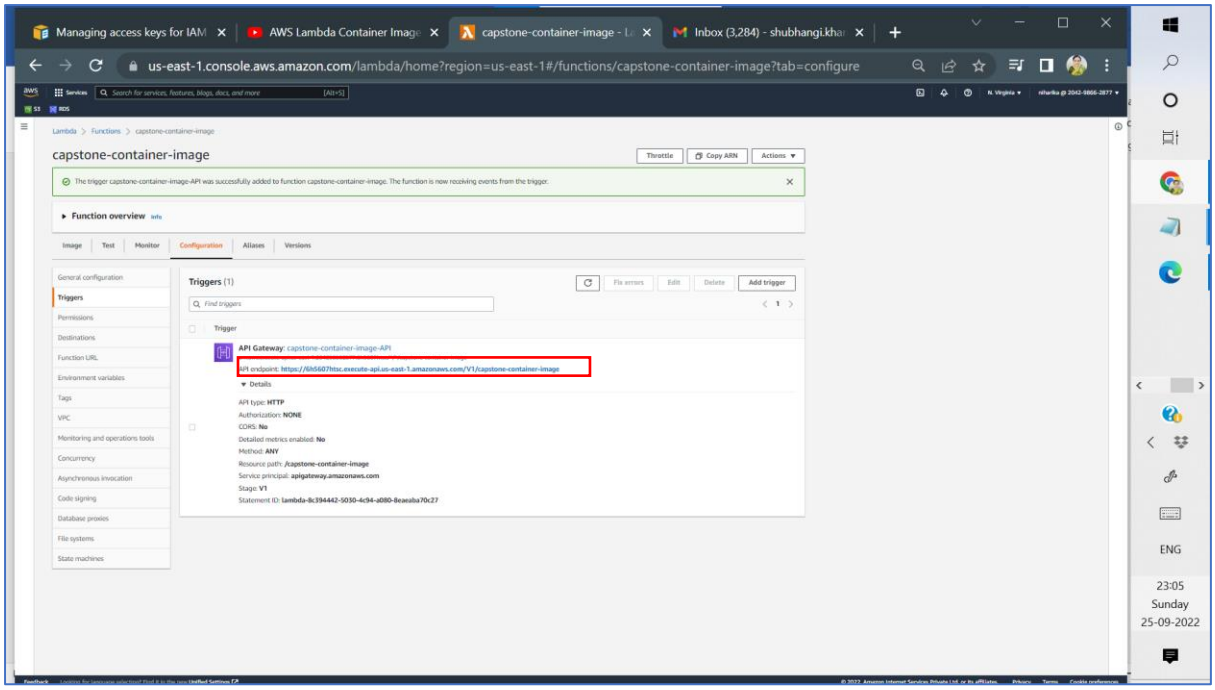
31. Selected API Gateway as trigger configuration



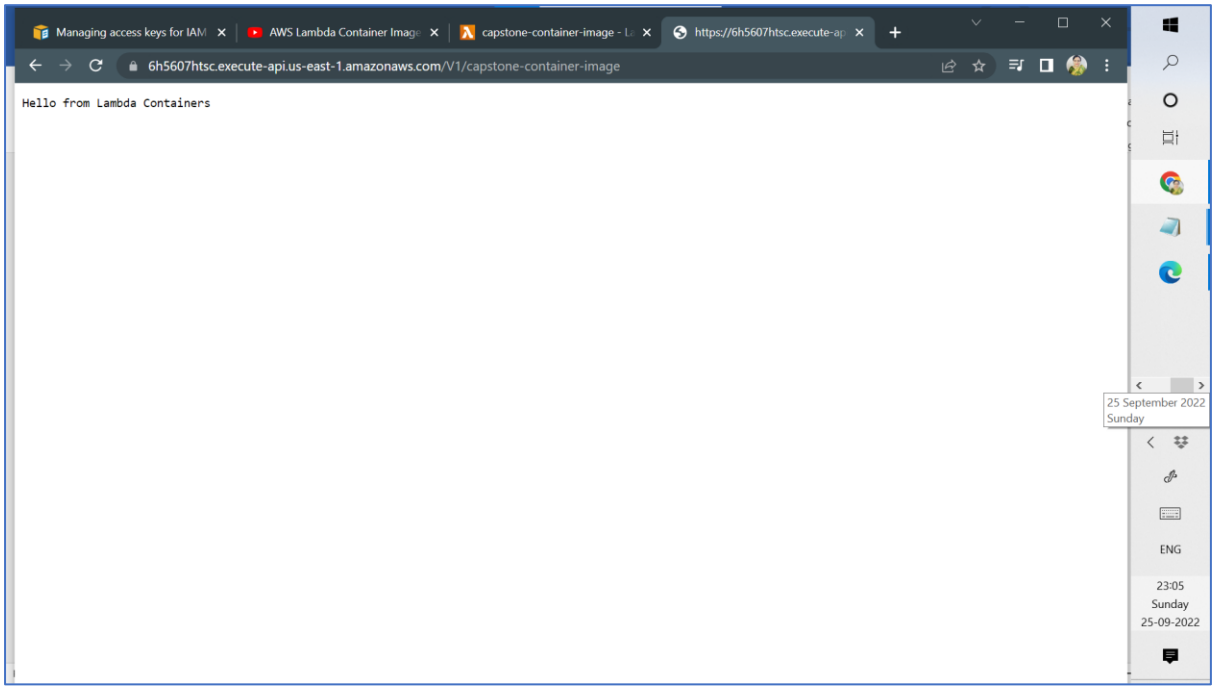
32. Creating new API with HTTP API Type and deployment stage name as V1



33. Trigger API Gateway is successfully added to lambda function capstone container image. The function will now receive events from the trigger. Note API endpoint URL for accessing lambda function



34. Details of API Gateway trigger. Type: HTTP, authorization: None, CORS: No, detailed metrics enabled: No, HTTP method: ANY, deployment stage V1



35. Lambda function capstone-container-image accessed successfully using API endpoint URL. “Hello from lambda containers” message received in lambda function is same as the one sent using JSON event.

C. Cost Analysis of the implemented Solution –Assume your solution is used by 1000 users for a month, and give monthly billing estimates.

Sr. No	Region	Services	Upfront	monthly	For 1000 users	First 12 months	currency	Configuration summary	Calculations
1	US East (N virginia)	Amazon EC2	0.00	8.26	8260	99120	USD	Operating system (Linux), Quantity (1), Pricing strategy (EC2 Instance Savings Plans 1 Year No Upfront), Storage amount (30 GB), Instance type (t2.micro)	EC2 Instance Savings Plans rate for t2.micro in the US East (N. Virginia) for 1 Year term and No Upfront is 0.0072 USD Hours in the commitment: 365 days * 24 hours * 1 year = 8760.0000 hours Total Commitment: 0.0072 USD * 8760 hours = 63.0720 USD Upfront: No Upfront (0% of 63.072) = 0.0000 USD Hourly cost for EC2 Instance Savings Plans = (Total Commitment - Upfront cost)/Hours in the term: (63.072 - 0)/8760 = 0.0072 USD *Please note that you will pay an hourly commitment for Savings Plans and your usage will be accrued at a discounted rate against this commitment. Pricing calculations 1 instances x 0.0072 USD x 730 hours in month = 5.26 USD (monthly instance savings cost) Amazon EC2 Instance Savings Plans instances (monthly): 5.26 USD
2	US East (N virginia)	Amazon ECR	0.00	0.19	190	2280	USD	Amount of data stored (1 GB per month)	1 GB per month x 0.10 USD = 0.10 USD Elastic Container Registry pricing (monthly): 0.10 USD Inbound: Internet: 1 GB x 0 USD per GB = 0.00 USD Outbound: Internet: 1 GB x 0.09 USD per GB = 0.09 USD Data Transfer cost (monthly): 0.09 USD Elastic Container Registry pricing (Monthly): 0.10 USD Data Transfer cost (Monthly): 0.09 USD
3	US East (N virginia)	AWS Lambda	0.00	0.00	0.00	0.00	USD	Lambda function with 1000 requests per month	Data transfer cost between SNS and lambda in same region is 0.00 USD
4	US East (N virginia)	API Gateway	0.00	0.00	0.00	0.00	USD	1000 requests per month	34 KB per request / 512 KB request increment = 0.06640625 request(s) RoundUp (0.06640625) = 1 billable request(s) 1 requests per month x 1,000 unit multiplier x 1 billable request(s) = 1,000 total billable request(s) Tiered price for: 1000 requests 1000 requests x 0.0000010000 USD = 0.00 USD Total tier cost = 0.0010 USD (HTTP API requests) HTTP API request cost (monthly): 0.00 USD.
5	US East (N virginia)	IAM role	0.00	0.00	0.00	0.00	USD	IAM is offered at no additional charge.	IAM role for EC2 to provide full access to aws lambda
		Total	**0.00**	**8.45**	**8450**	**101400**			

D. Lessons & Observations

- Learnt that there are 3 ways to create lambda function;1) author from scratch 2) use a blueprint and 3) use container image.
- Learnt to create lambda function from container image in details
- **Learnt to use API Gateway for accessing** deployed image
- **Learnt to build container image using following 3 files:**
    a. **Requirements.txt file** contains all packages that are required in app.py file
    b. **app.py file** imports all packages required to build lambda function. It defines a function lambda_handler. The handler function accepts two arguments; event and context. The event argument retrieves the data that is passed to execute the lambda function. For e.g. if the lambda function is triggered using HTTP API call via API Gateway then the event calls for all parameters required for HTTP API call (resource, path, method, resource path, headers etc). The context argument provides the data about the execution environment of the lambda function. The lambda_handler function must have a return value that depends on the invocation type. For example statusCode 200 is returned when lambda function is invoked using HTTP API call indicating that the request for invoking lambda is successful. StatusCode 200 is success response code. The meaning of a success depends on the HTTP request method: GET : The resource has been fetched and is transmitted in the message body (Hello from Lambda Containers).
    c. **Dockerfile** contains python 3.8 lambda base image which is picked up from public.ecr.aws. The Dockerfile installs all packages and dependencies included in requirements.txt file. It copies the app.py to the container and sets the command for app.handler.
- Learnt Docker push commands to push built docker container image from EC2 to ECR
- Learnt to create private ECR
- Learnt to troubleshoot errors while using push commands
- Learnt to create IAM role for EC2 to provide full access to AWS Lambda
- Learnt to estimate cost for each service
- Lambda is a serverless compute service from AWS