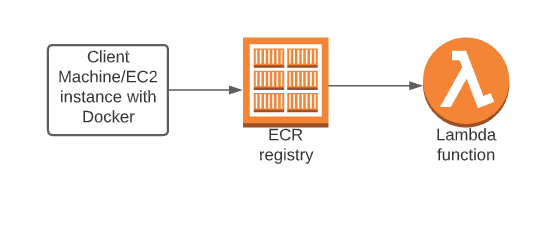
Dr. Shubhangi Kharche

**AWS Project 4**

**Scenario**

The introduction of Lambda support for OCI container images provides customers with more choices when it comes to packaging formats. Developers can now choose to take advantage of the event-driven runtime model and cost-savings advantages of AWS Lambda, while taking advantage of the predictability and control offered by a container-based development and deployment cycle.

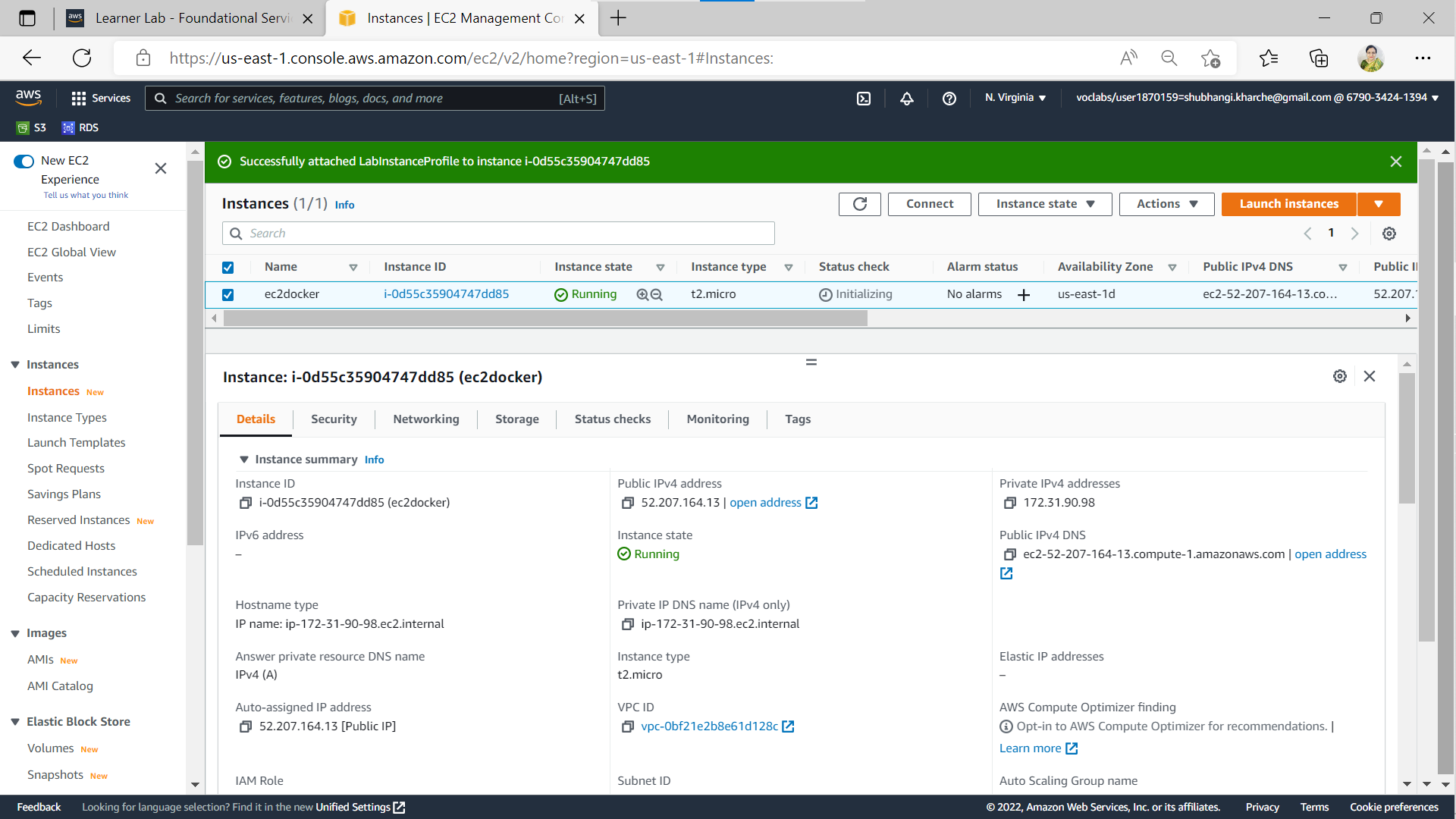
**Architecture diagram**

****

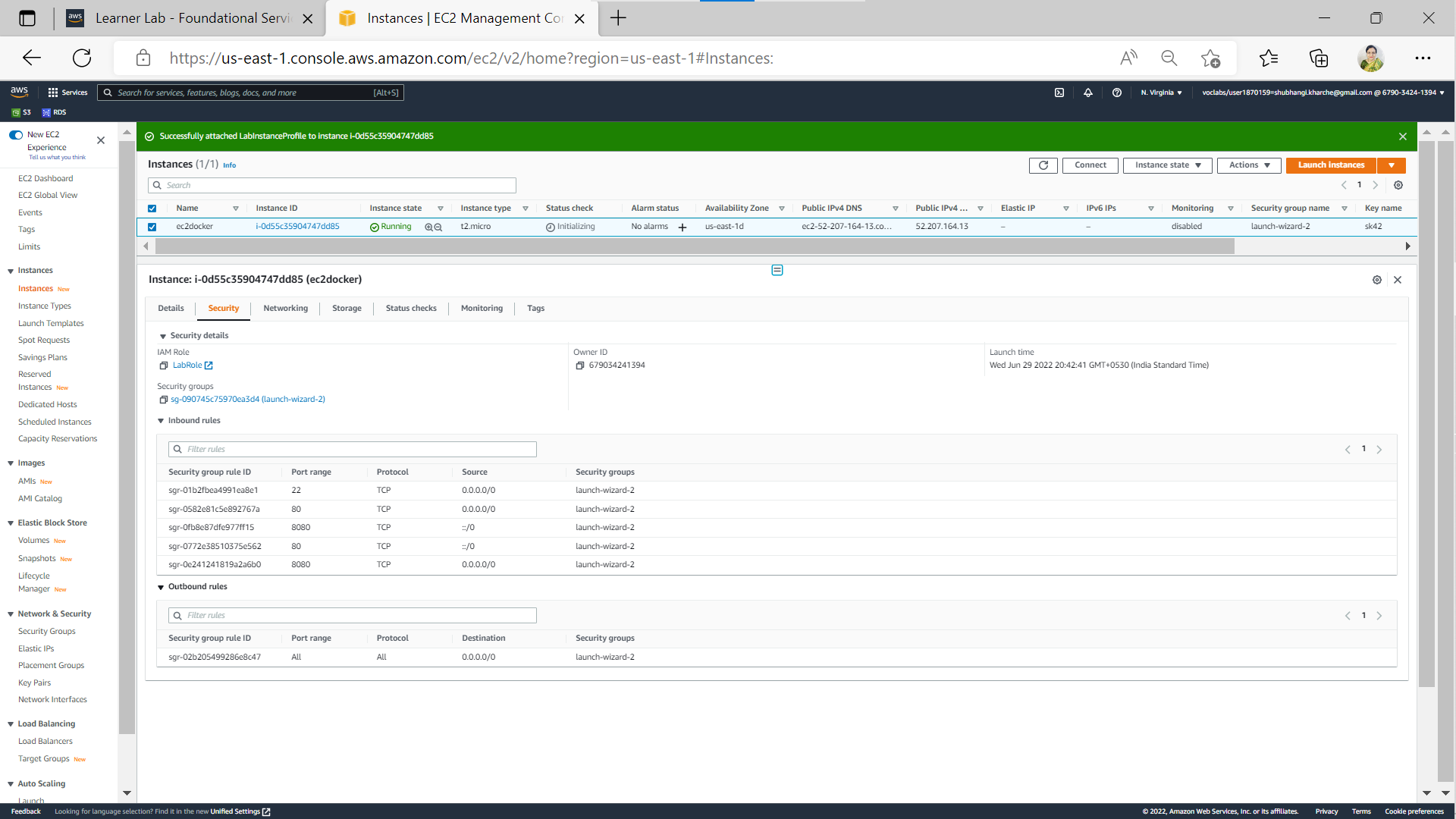
|  |  |
| --- | --- |
| **Architecture Implementation** | |
| 1 | Download the Dockerfile and the app code folder provided with this workbook |
| 2 | Package the web application as a Docker image running on Alpine with Python |
| 3 | Create an ECR repository and login to it. |
| 4 | Build the image with the downloaded dockerfile and the support files |
| 5 | Tag the image appropriately and push it into the ECR repository. |
| 6 | Create a Lambda function with the image in ECR. |

**Step 1 : Docker Image creation**

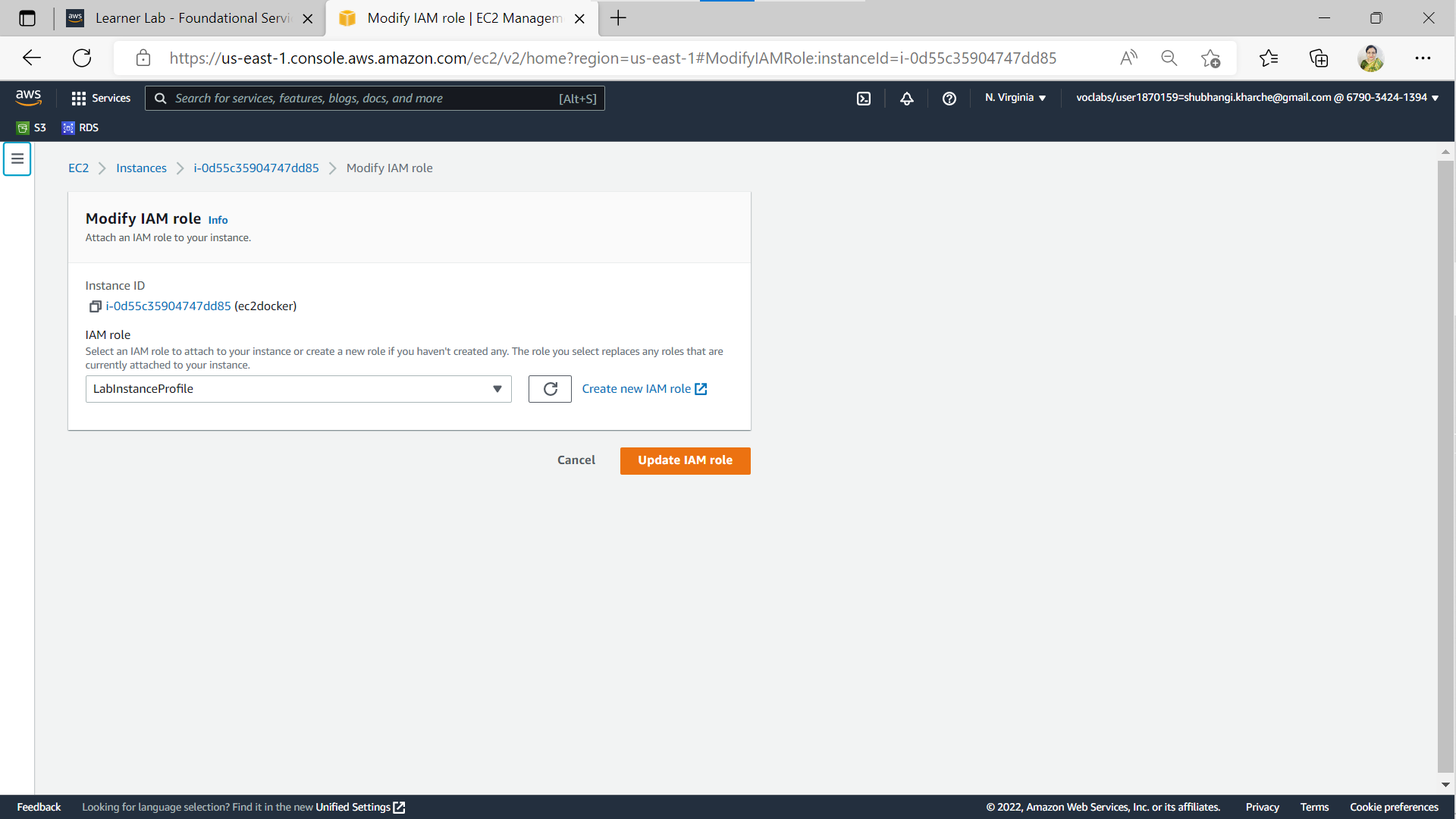
|  |  |
| --- | --- |
| Step number | a |
| Step name | Creation of Docker image |
| Instructions | 1. Create an EC2 instance using the Amazon Linux 2 AMI in the default VPC. 2. Attach the role “LabInstanceProfile” to the instance created above 3. Download the file OCI.zip provided with this workbook and copy it to the EC2 instance using the scp command scp -i <pem file name> ./OCI.zip ec2-user@<public IP of instance>:/home/ec2-user (Ensure that the file OCI.zip and the pem file are in the same folder before running this command) 4. Login to the instance using SSH and run the following commands to set up the environment *sudo yum update sudo yum install unzip sudo unzip OCI.zip sudo amazon-linux-extras install docker sudo service docker start sudo usermod -a -G docker ec2-user* **(At this point, log out of the instance and log in again to ensure that the above command works. Then continue with the rest of the commands)** *sudo yum install awscli -y aws configure*   **Skip the access key and secret access key fields by pressing the Enter key. Enter the region as**  **us-east-1 and format as json**   1. Run the below command to create the Docker image *docker build -t lambda\_ecr .* 2. Run the below command to verify the creation of the image *docker images* |
| Expected screenshots | 1 )Building the Docker image 3) List of the created image |

****

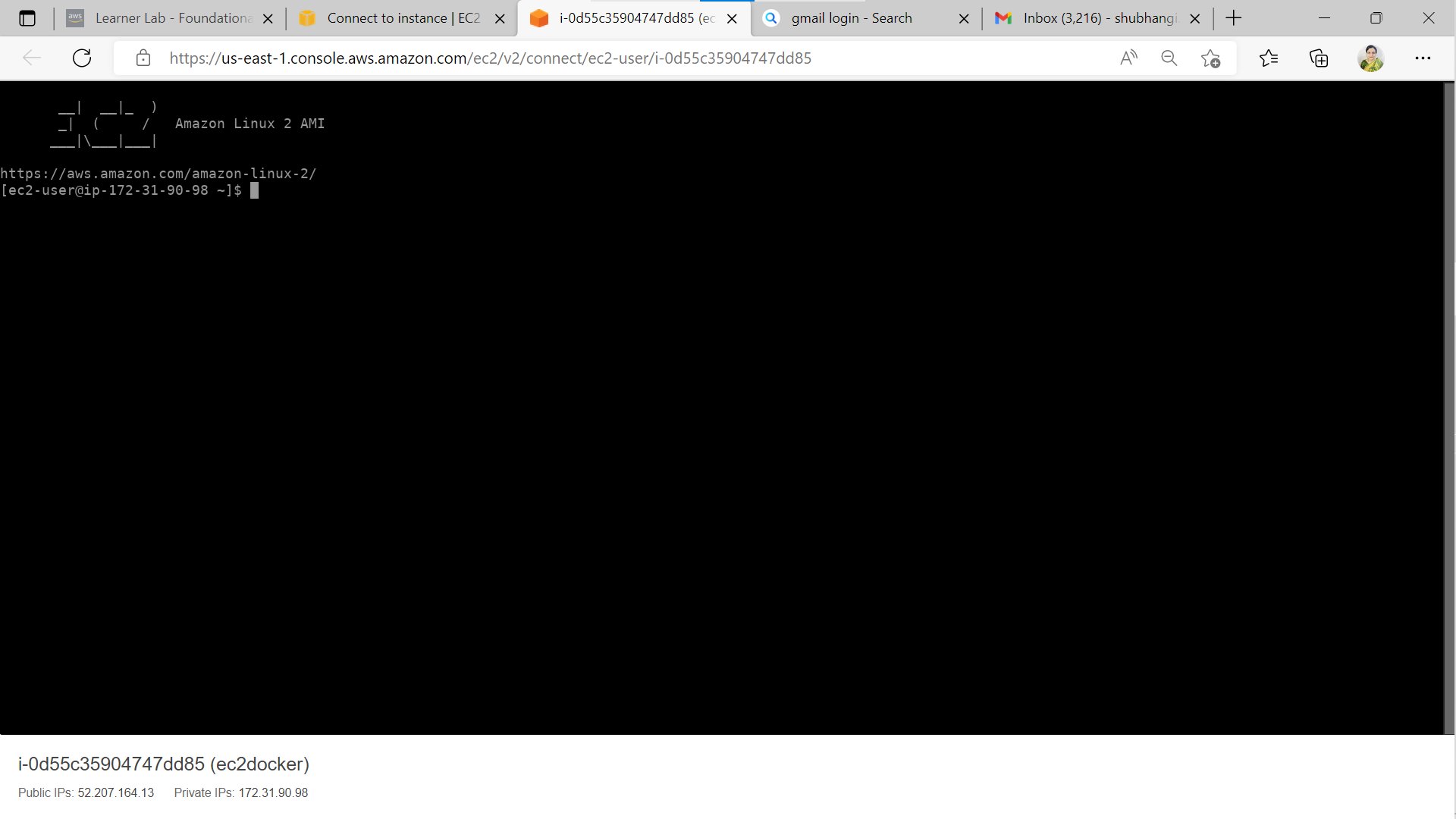
**Fig 1.a Creation of EC2 instance (optional screenshot)**

****

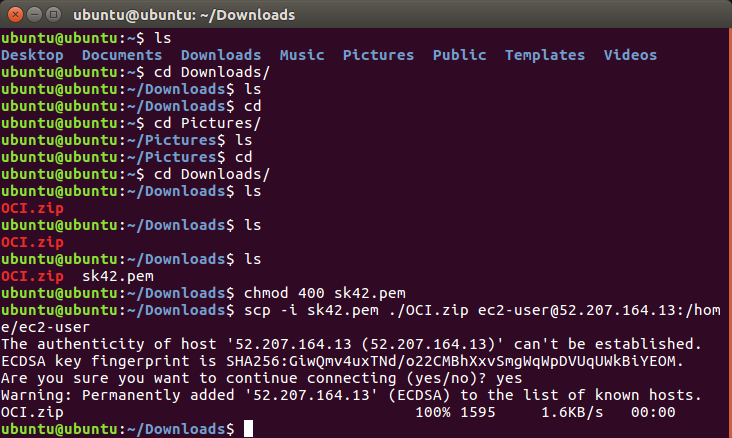
**Fig 1.b Defining security groups [SSH port 22 (from anywhere using IPv4), HTTP port 80 (from anywhere using both IPv4 and IPv6), docker host port 8080 (from anywhere using both IPv4 and IPv6)] (optional screenshot)**

****

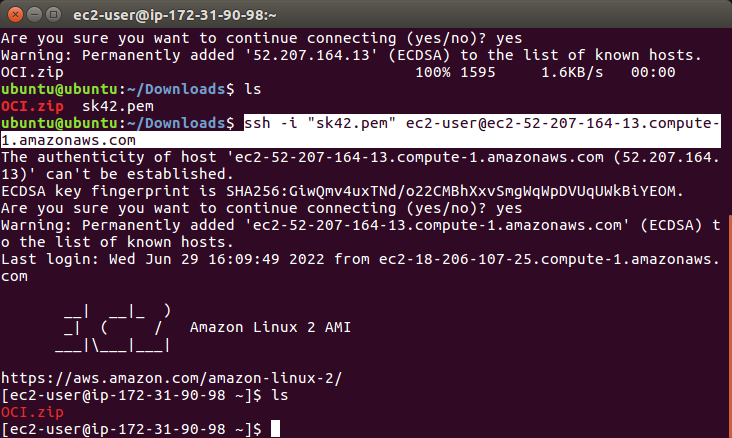
**Fig 1.c Updation of IAM Role (optional screenshot)**

****

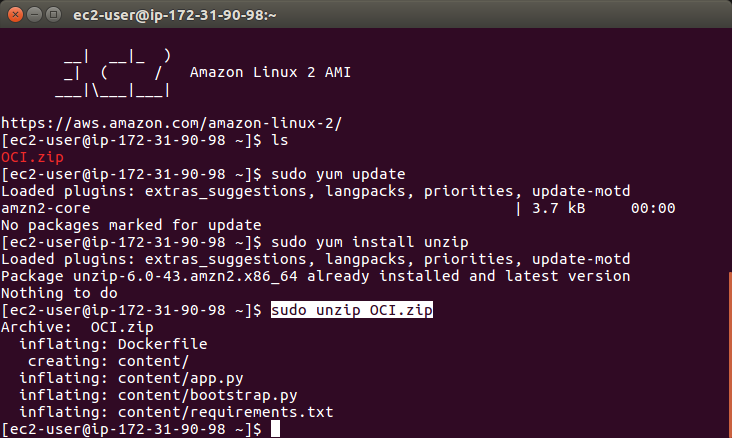
**Fig 1.d Successful login to created EC2 instance (optional screenshot)**



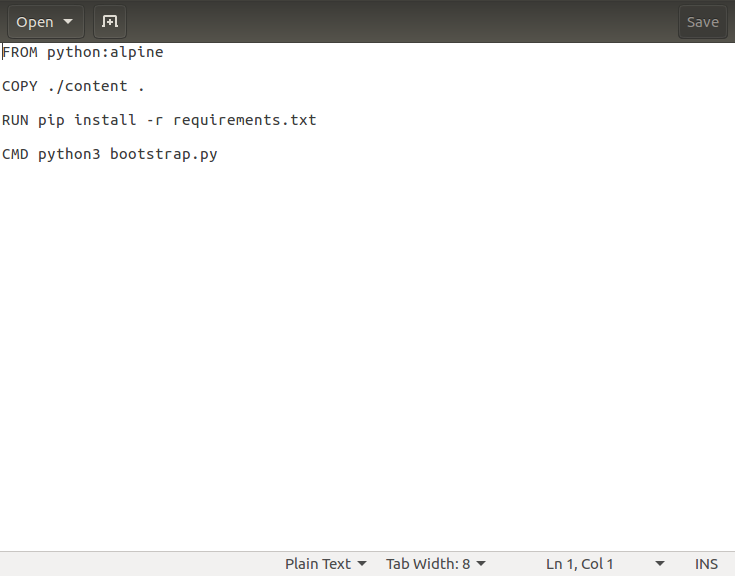
**Fig 1.e Securely copying OCI.zip to created EC2 instance (optional screenshot) (assigned read permission to .pem file)**



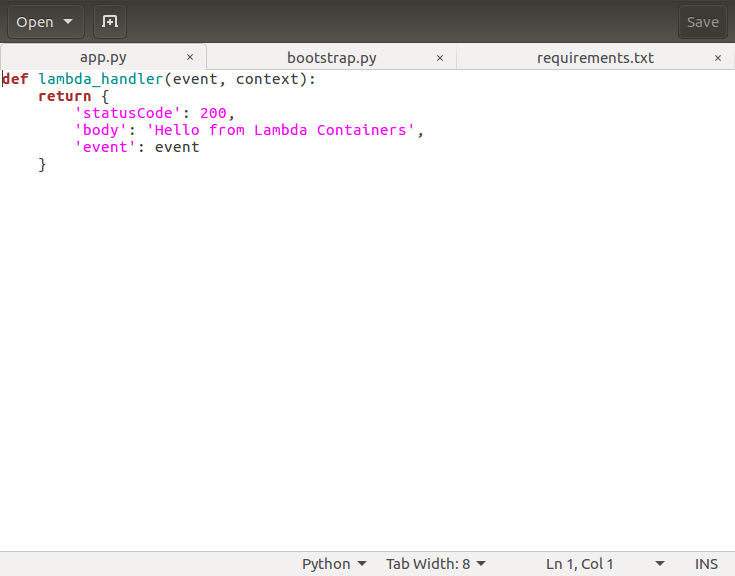
**Fig 1.f Successful SSH login to created EC2 instance from ubuntu terminal (optional screenshot)**



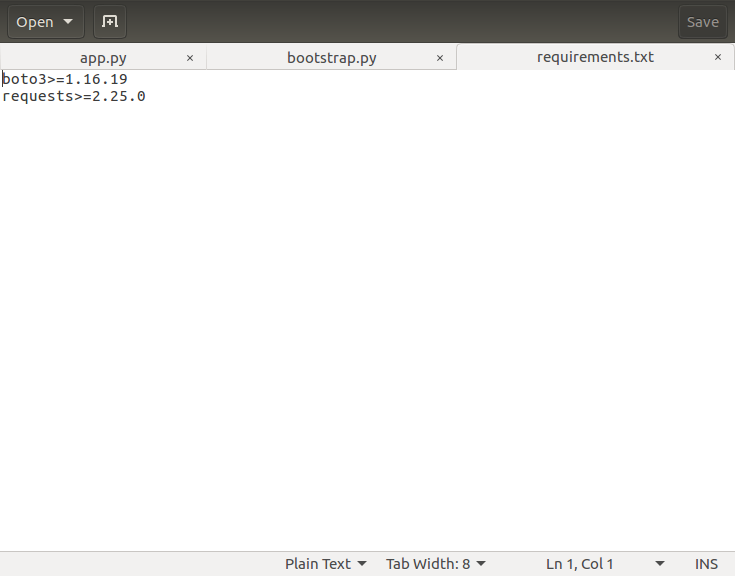
**Fig 1.g Unzipping OCI.zip (optional screenshot)**

****

**Fig 1.h Contents of Dockerfile (optional screenshot)**

****

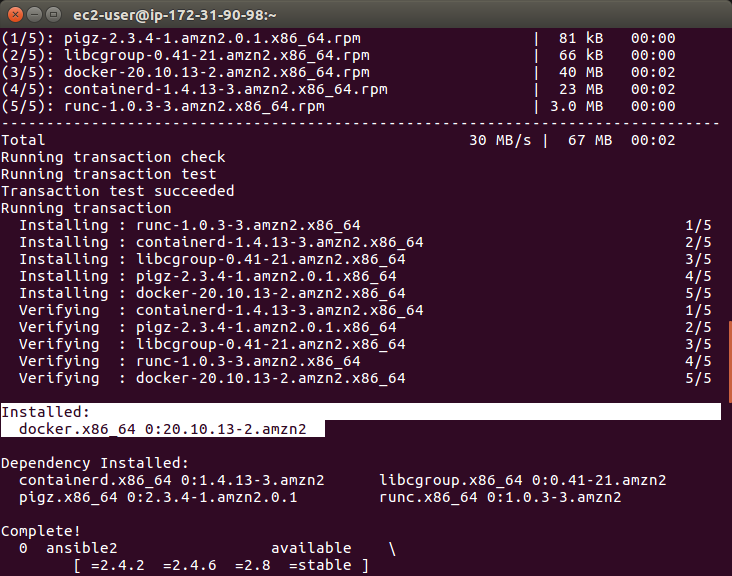
**Fig 1.i Contents of app.py file (optional screenshot)**

****

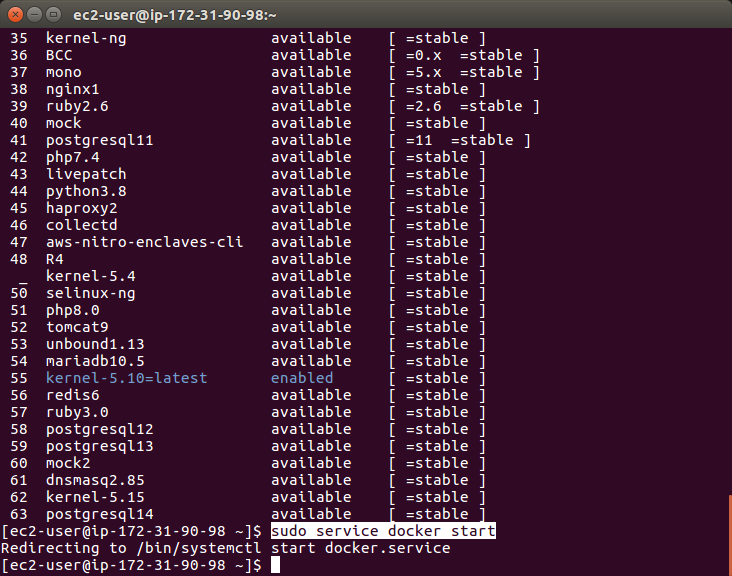
**Fig 1.j Contents of requirements.txt file (optional screenshot)**

import os  
import requests  
import sys  
import traceback  
  
def run\_loop():  
       aws\_lambda\_runtime\_api = os.environ['AWS\_LAMBDA\_RUNTIME\_API']  
         
       import app  
         
       while True:  
              request\_id = None  
              try:  
                     invocation\_response = requests.get(f'http://{aws\_lambda\_runtime\_api}/2018-06-01/runtime/invocation/next')  
  
                     request\_id = invocation\_response.headers['Lambda-Runtime-Aws-Request-Id']  
                     invoked\_function\_arn = invocation\_response.headers['Lambda-Runtime-Invoked-Function-Arn']  
                     trace\_id = invocation\_response.headers['Lambda-Runtime-Trace-Id']  
                     os.environ['\_X\_AMZN\_TRACE\_ID'] = trace\_id  
                       
                     context = {  
                            'request\_id': request\_id,  
                            'invoked\_function\_arn': invoked\_function\_arn,  
                            'trace\_id': trace\_id  
                     }  
                       
                     event = invocation\_response.json()  
                       
                     response\_url = f'http://{aws\_lambda\_runtime\_api}/2018-06-01/runtime/invocation/{request\_id}/response'  
                       
                     result = app.lambda\_handler(event, context)  
                       
                     sys.stdout.flush()  
  
                     [requests.post](http://requests.post/" \t "_blank)(response\_url, json=result)  
               
              except:  
                      if request\_id != None:  
                            try:  
                                   exc\_type, exc\_value, exc\_traceback = sys.exc\_info()  
                                   exception\_message = {  
                                        'errorType': exc\_type.\_\_name\_\_,  
                                        'errorMessage': str(exc\_value),  
                                        'stackTrace': traceback.format\_exception(exc\_type,exc\_value, exc\_traceback)  
                                   }  
                                     
                                   error\_url = f'http://{aws\_lambda\_runtime\_api}/2018-06-01/runtime/invocation/{request\_id}/error'  
                                   sys.stdout.flush()  
                                     
                                   [requests.post](http://requests.post/" \t "_blank)(error\_url, json=exception\_message)  
                            except:  
                                   pass  
  
run\_loop()

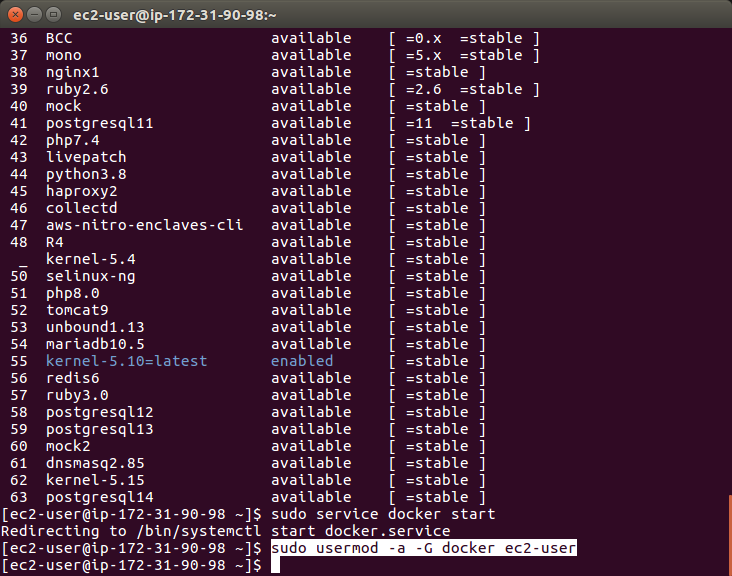
**Fig 1.k Contents of bootstrap.py file (optional screenshot)**

****

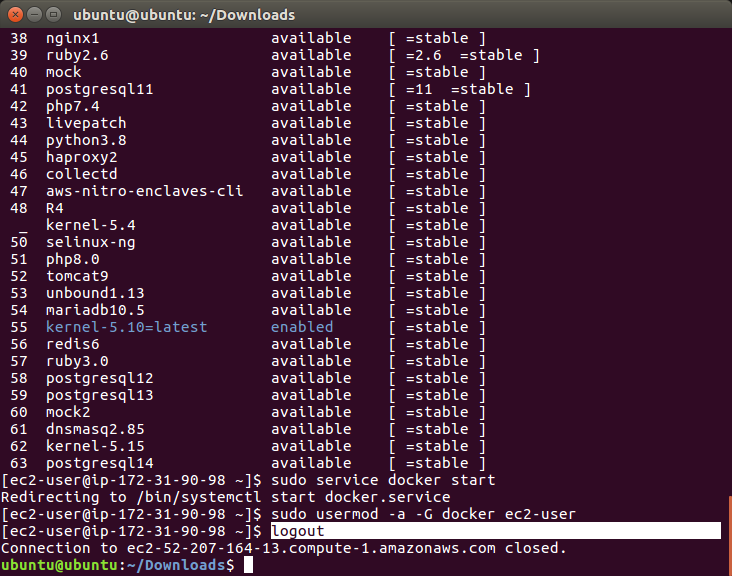
**Fig 1.l Successful installation of docker (optional screenshot)**

****

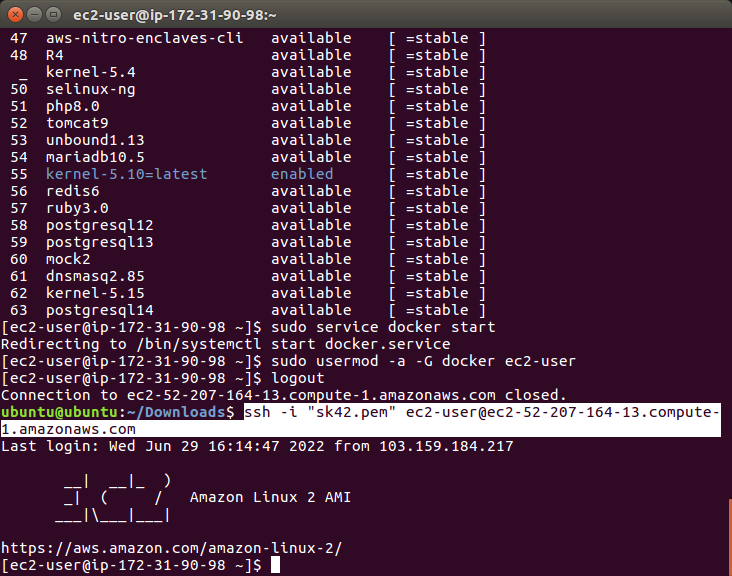
**Fig 1.m docker service start (optional screenshot)**

****

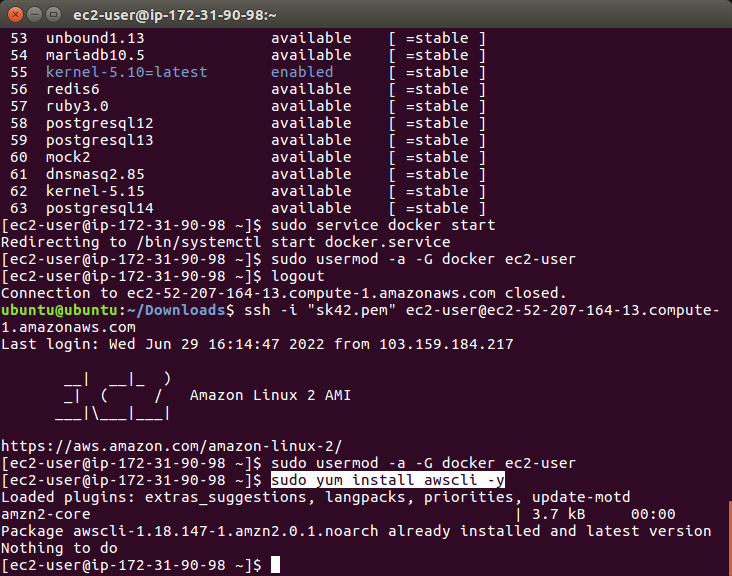
**Fig 1.n Adding ec2-user to docker group(optional screenshot)**

****

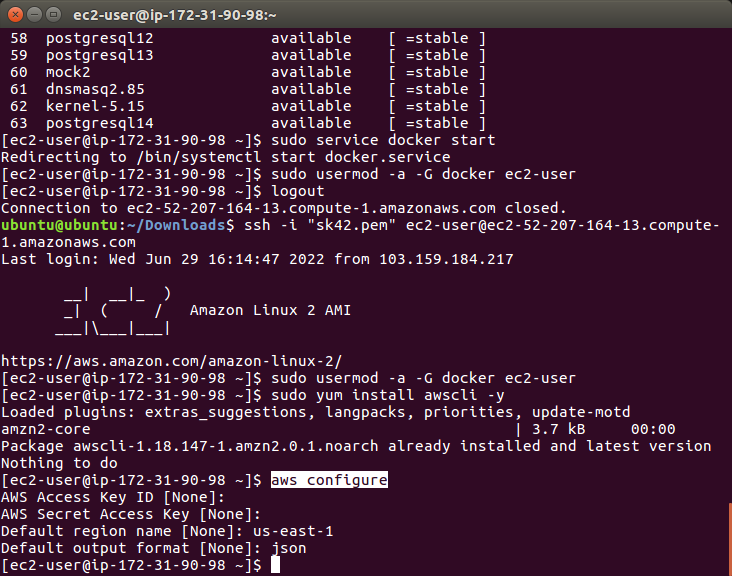
**Fig 1.o logout from ec2 instance to check whether ec2 user is added to docker group or not (optional screenshot)**

****

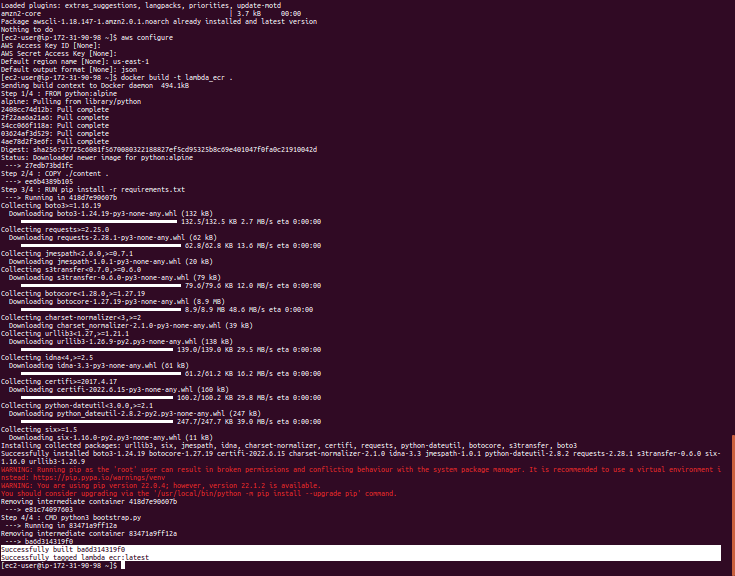
**Fig 1.p Login again to ec2 instance (optional screenshot)**

****

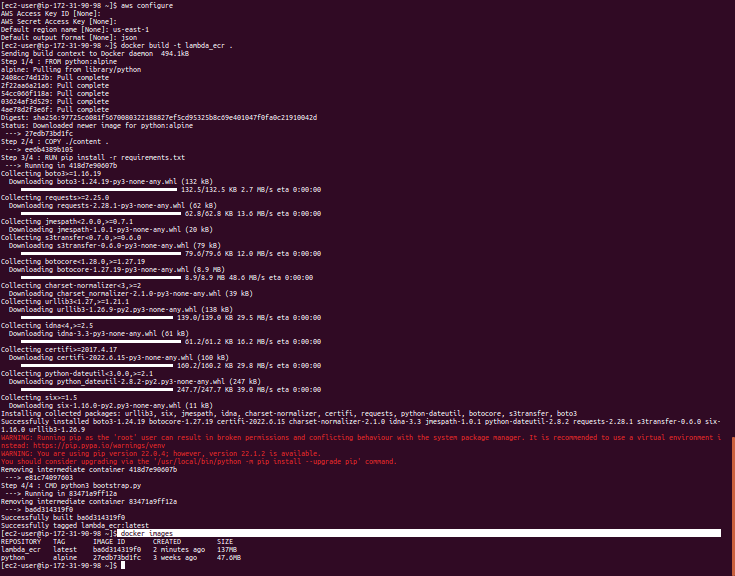
**Fig 1.q Installing awscli for building docker image via CLI(optional screenshot)**

****

**Fig 1.r Configure AWS (fastest way to set up your AWS CLI installation) (optional screenshot)**



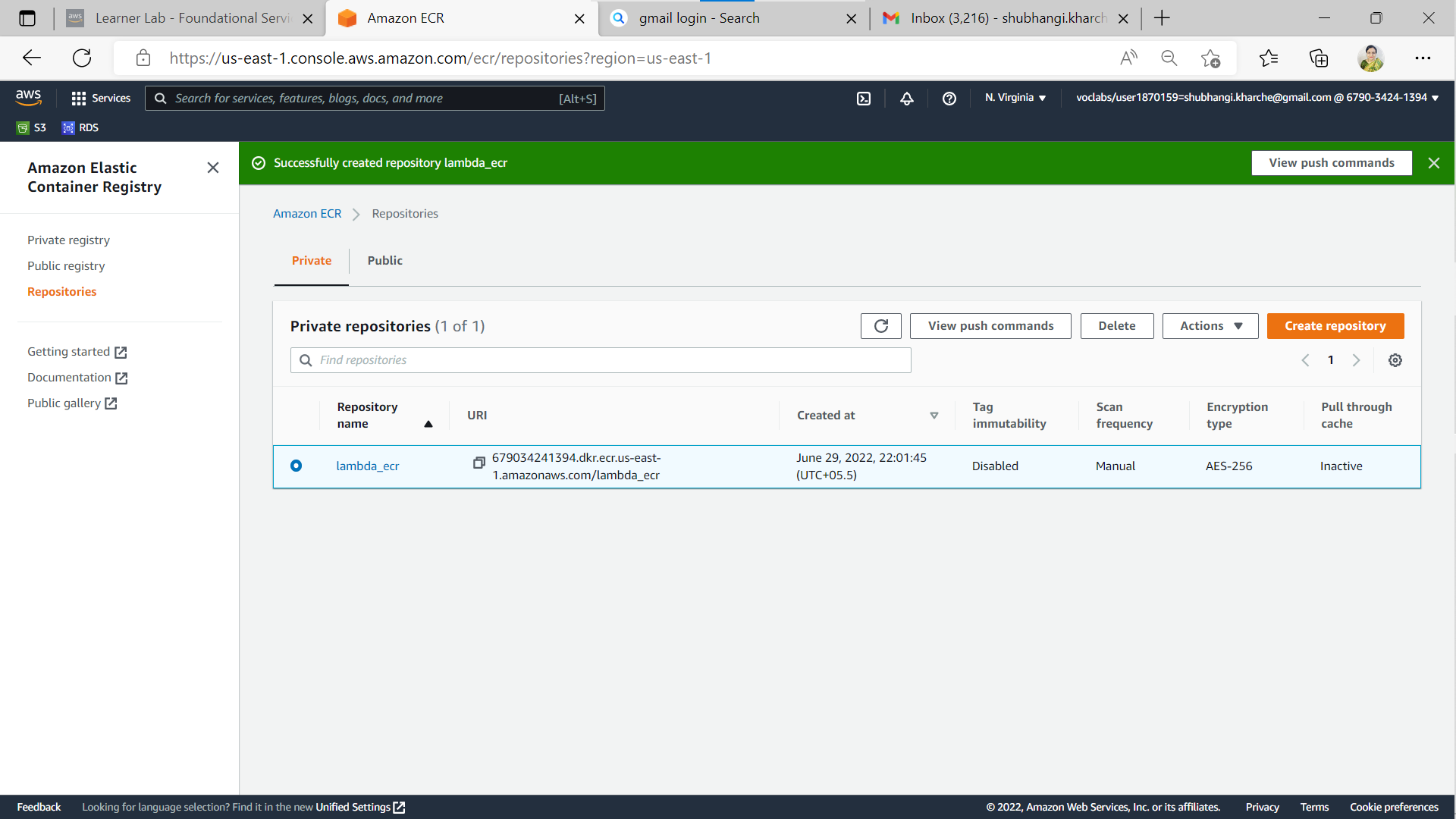
**Fig 1.s Building docker image**



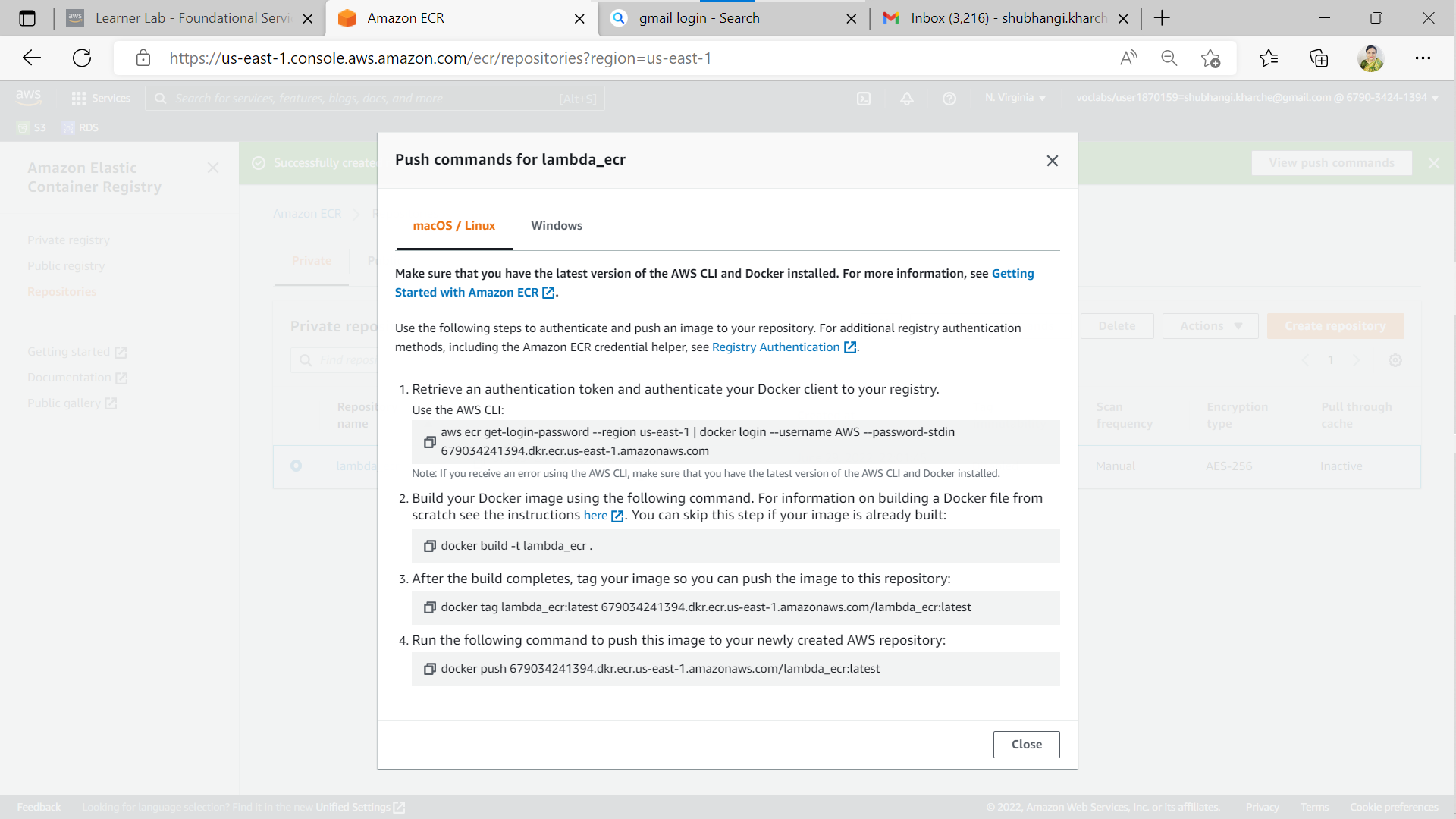
**Fig 1.t List of created images**

**Step 2: Create ECR repository and upload image to ECR**

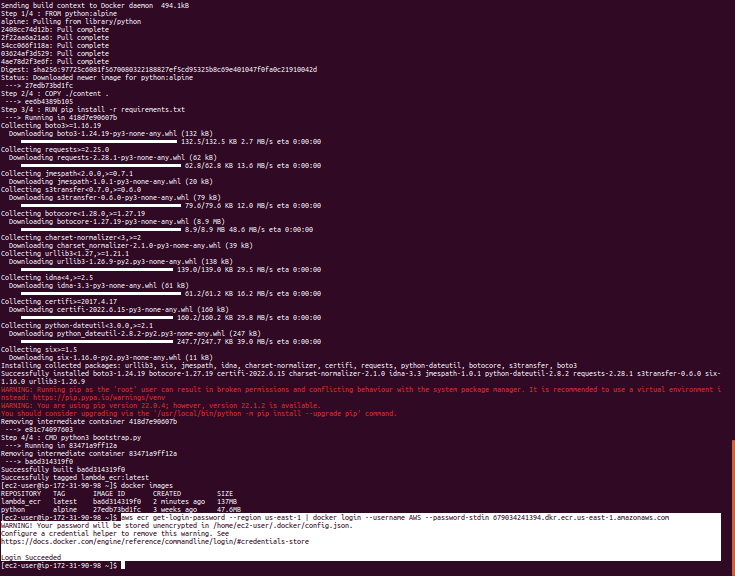
|  |  |  |
| --- | --- | --- |
| Step number | a |  |
| Step name | Creating the ECR repository |  |
| Instructions | 1. Go to the ECR service on the AWS console 2. Select the Repositories from the left pane 3. Create a new private repository named **lambda\_ecr** with the default settings |  |
| Step number | b |  |
| Step name | Image upload to ECR |  |
| Instructions | 1. Once the repository is created, select the repository and then click  on “View push commands“ on the top right 2. From the pop up screen which appears, run commands 1, 3 and 4 after logging into the EC2 instance created above. Note that command 2 was already executed in the previous step when the image was created. For reference, the commands will be in the format shown below:   **aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin <xxxxxxx.dkr.ecr.us-east-1.amazonaws.com>**  **docker tag lambda\_ecr\_image:latest <xxxxxxx.dkr.ecr.us-east-1.amazonaws.com/lambda\_ecr>:latest**  **docker push <xxxxxxx.dkr.ecr.us-east-1.amazonaws.com/lambda\_ecr>:latest** |  |
| Expected screenshots | 1) Creation of Repository 2) View push commands  3) Login Succeeded 4) Tagging of the image 5) Pushing of image to ECR 6) Image uploaded on the ECR repo |  |

****

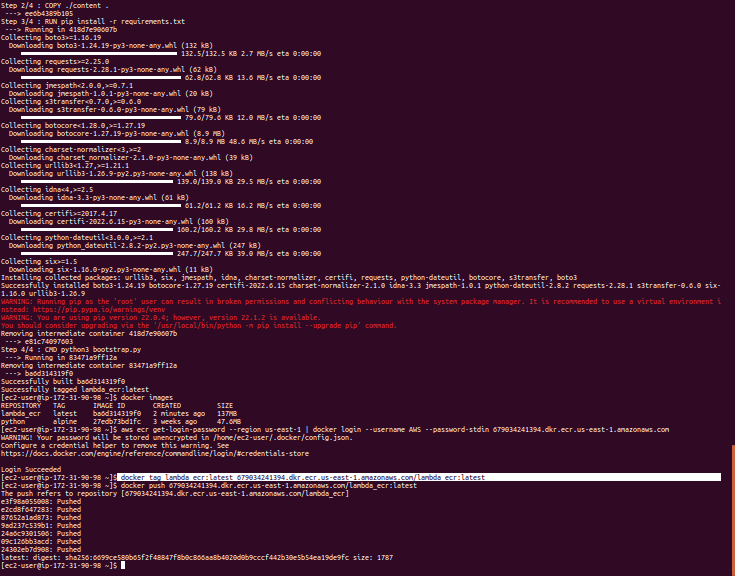
**Fig. 2 a) Creation of Repository on ECR (lambda\_ecr)**

****

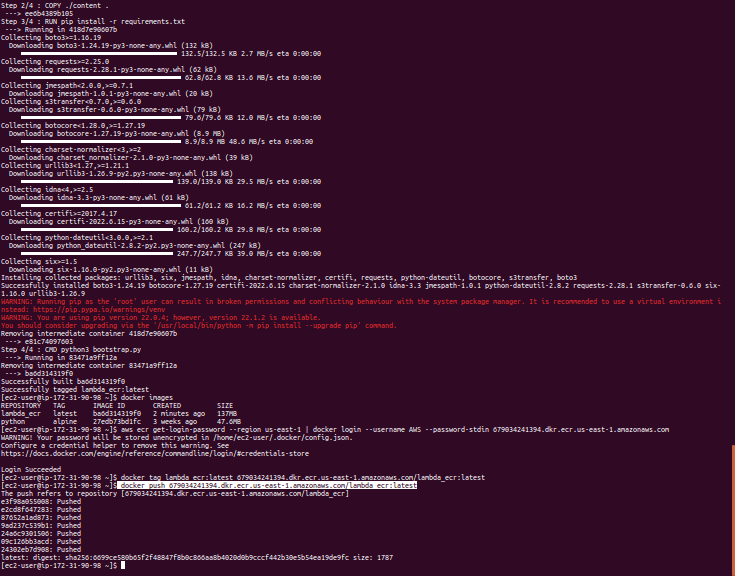
**Fig. 2 b) View push commands**

****

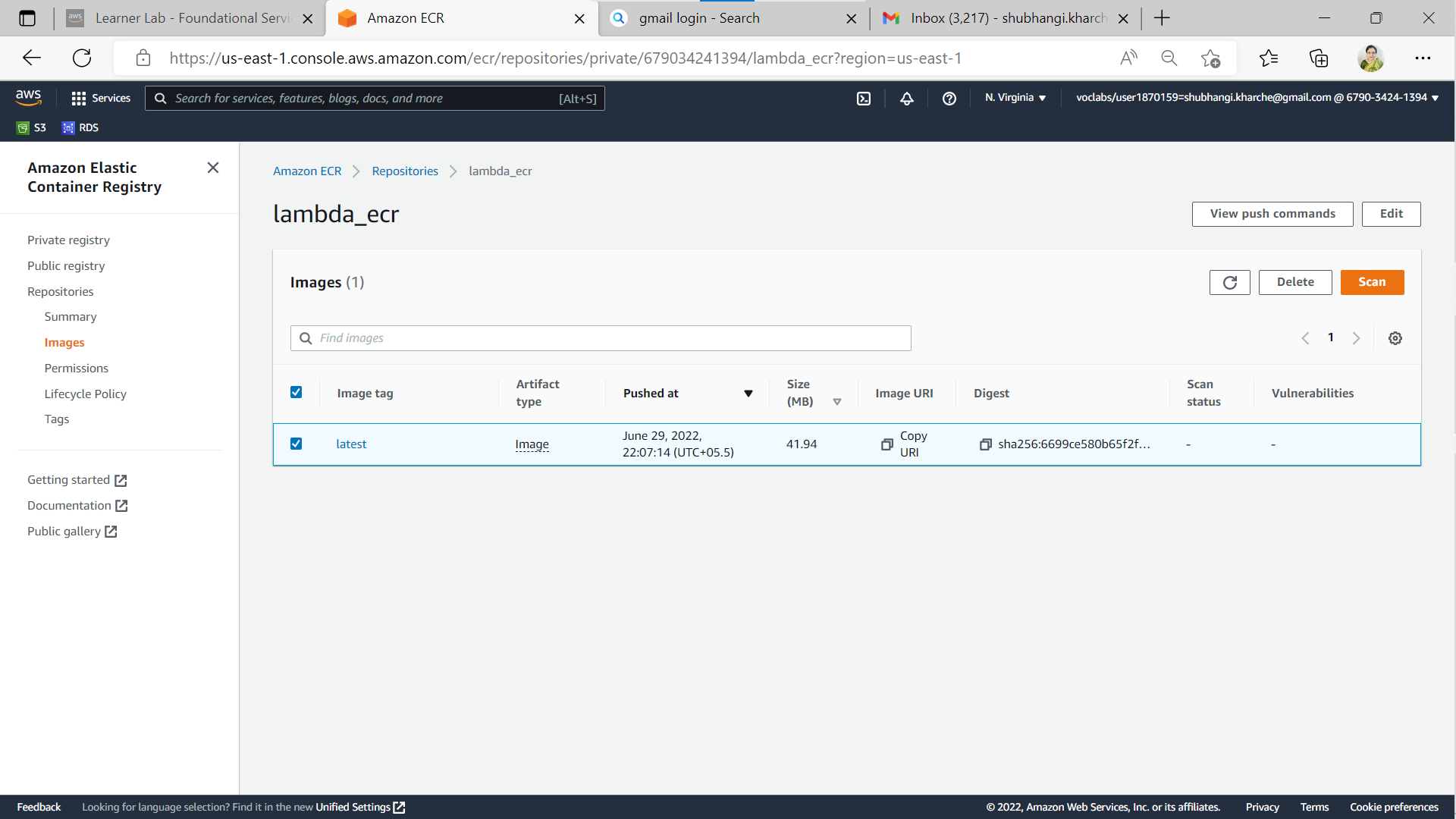
**Fig. 2 c) Login Succeeded**

****

**Fig. 2 d) Tagging of the image**

****

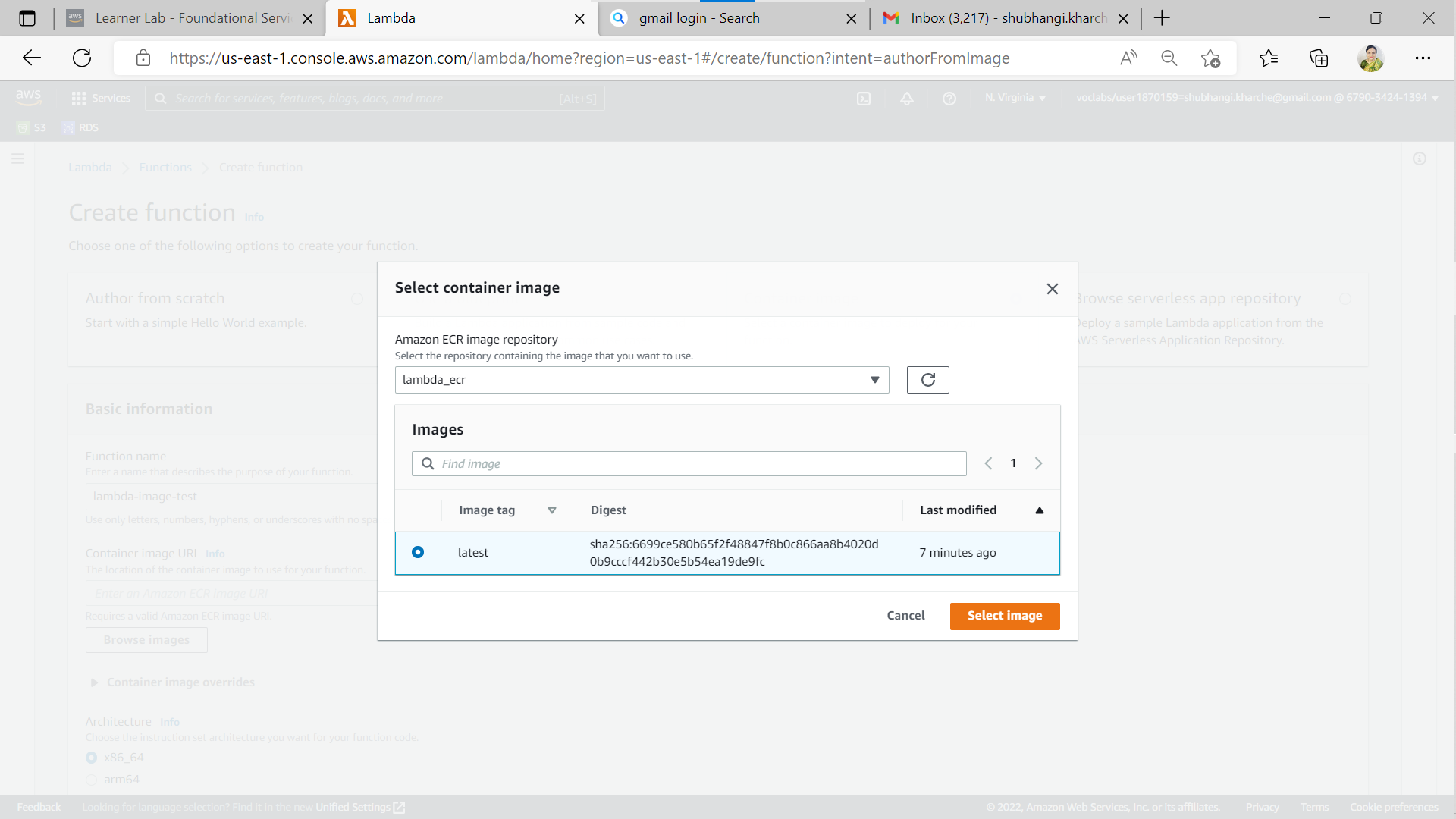
**Fig. 2 e) Pushing of image to ECR**

****

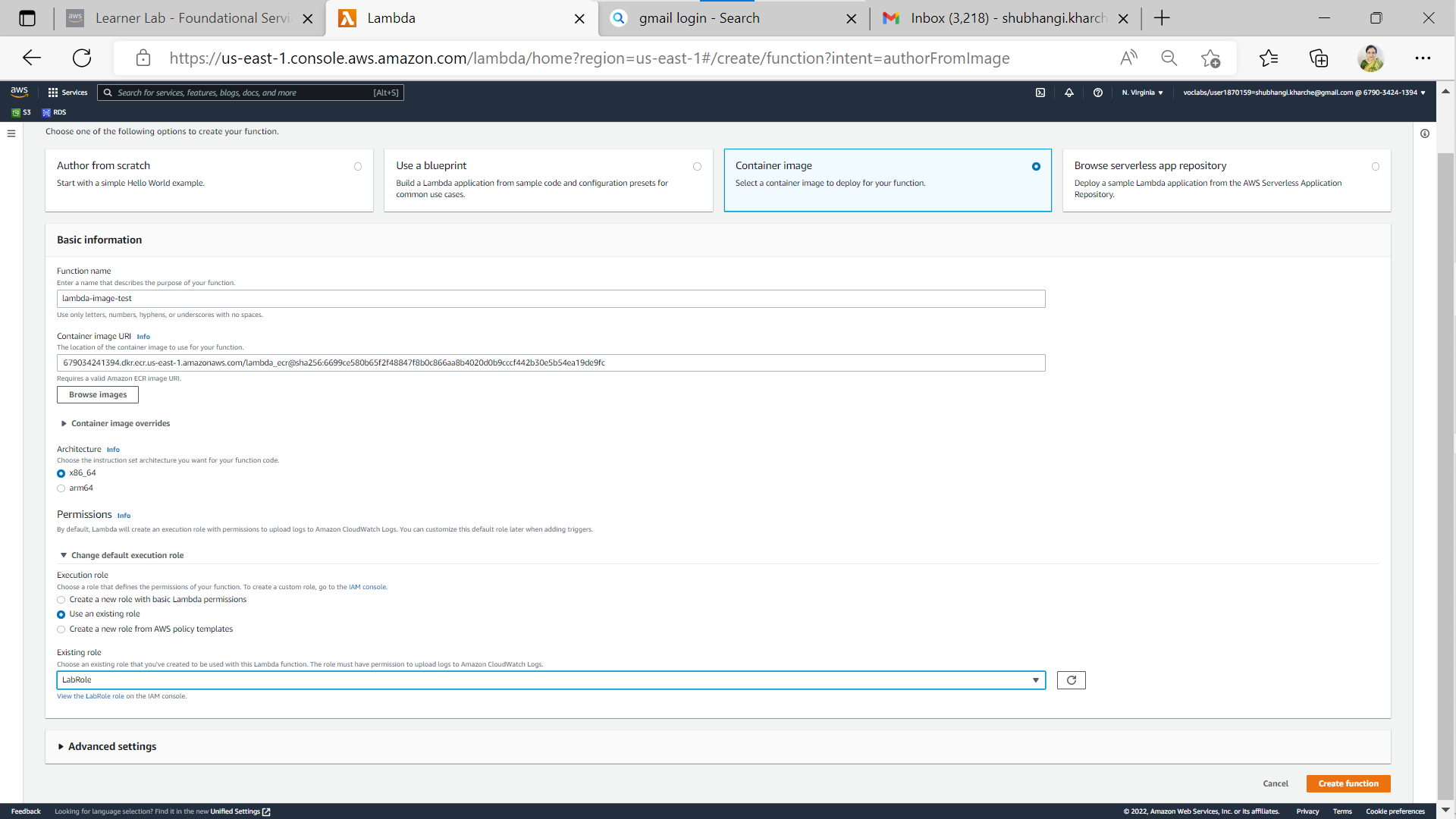
**Fig. 2 f) Image uploaded on the ECR repo**

**Step 3: Creation of Lambda function to test the image**

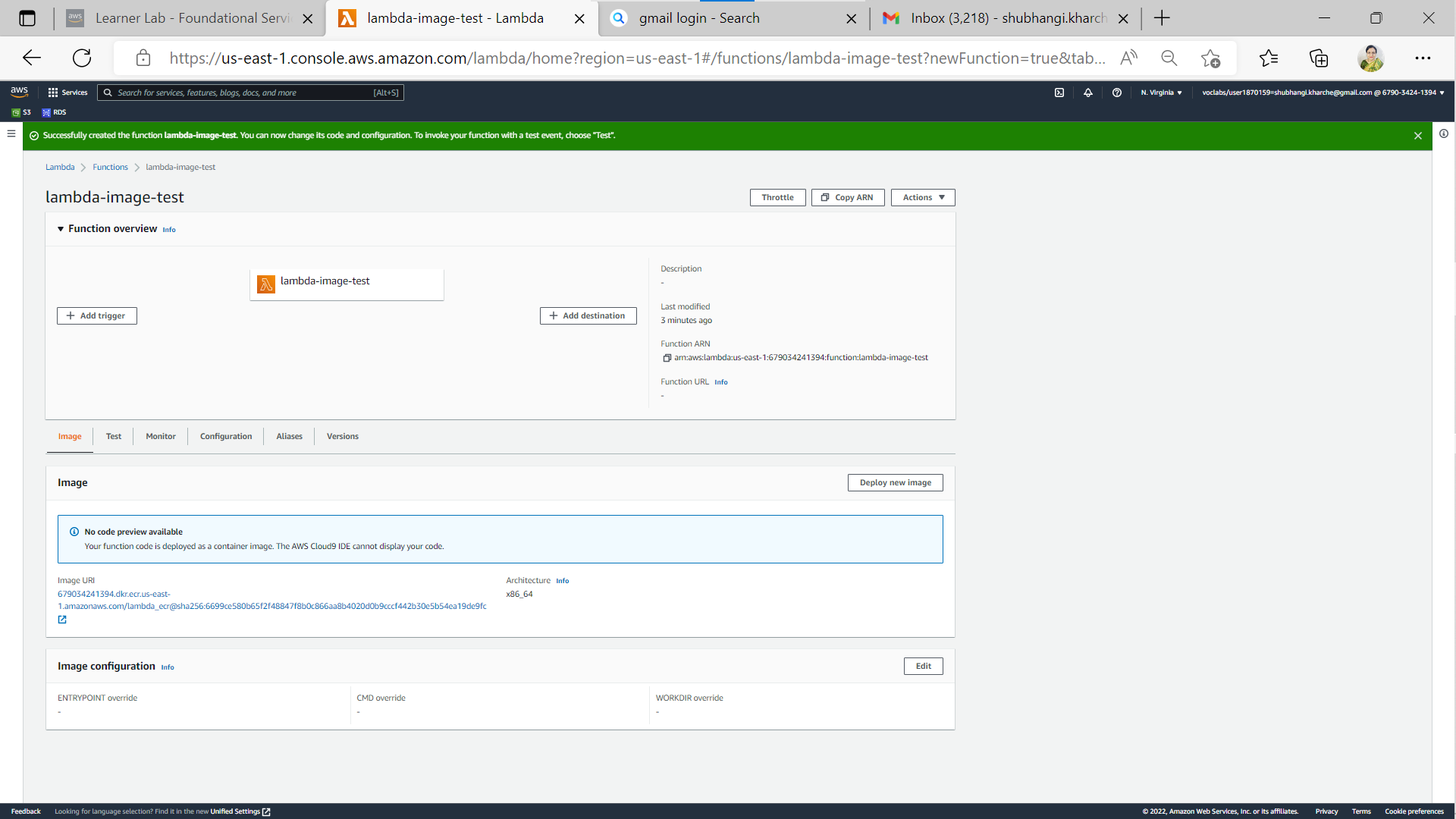
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Step number | a |  |  |  |
| Step name | Create the Lambda function and test the image |  |  |  |
| Instructions | 1. Navigate to the AWS Lambda service using the AWS Console 2. Click on **Create Function** 3. Under Create Function page select the ‘Container image’ option and enter a function name of your choice 4. For ‘Container image URI’ Click on “Browse Images” and select the repository and the image 5. Use the existing IAM role – LabRole. 6. Click on Create 7. Wait a few minutes for the function to be created 8. Test the function with the default “Hello World” test to see the result. | | | |
| Expected screenshots | 1) Container image selection 2) Execution role selection 3) Created function 4)Test result of function | | | |

****

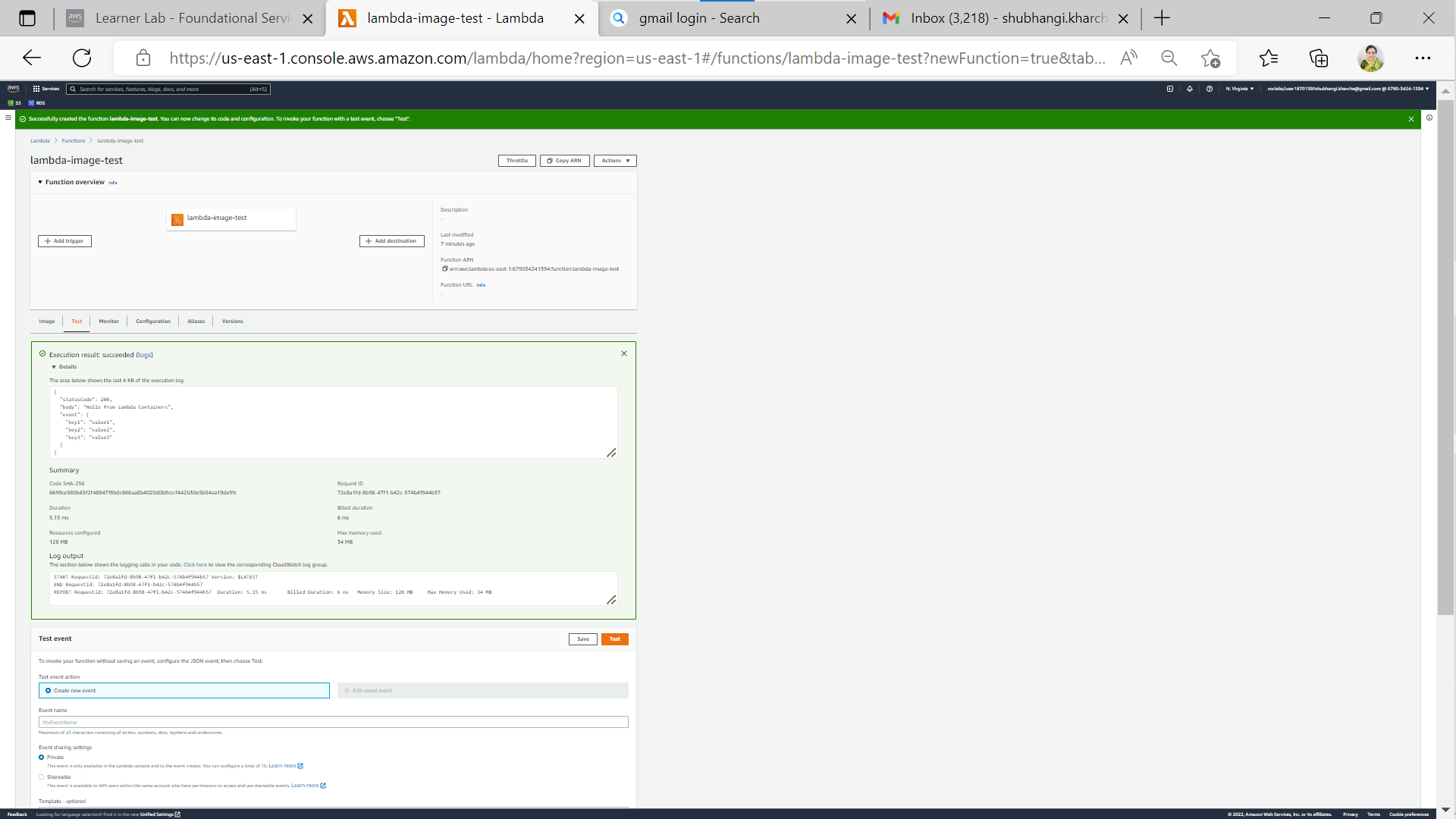
**Fig 3. a) Container image selection (latest)**

****

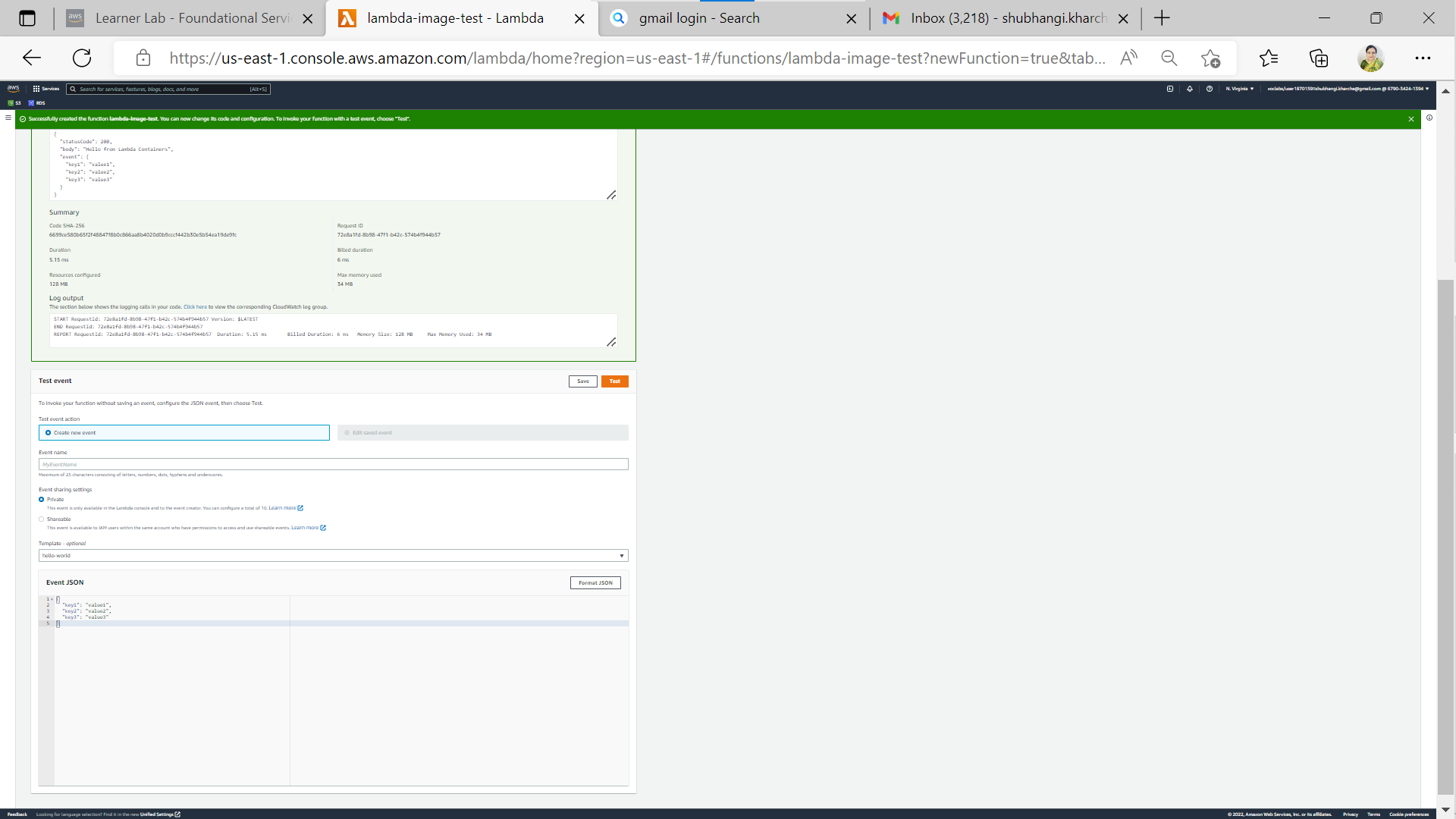
**Fig 3. b) Execution role selection (use an existing role)**

****

**Fig 3. c) Created function (lambda-image-test)**

****

**Fig 3. d)-1 Test result of function**

****

**Fig 3. d)-2 Test result of function**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Answer the following questions** | | | | **Points** |
| Q1 | How long does a container stay in the running state if it is not manually halted? | | | 1 |
|  | a) As long as the container’s PID 1 is running |  |  |  |
|  | b) Has a set timeout after which it pauses |  |  |  |
|  | c) Until its container is expunged |  |  |  |
|  | d) Docker daemon process scheduler decides on load |  |  |  |
|  | Enter your answer here | a) As long as the container’s PID 1 is running |  |  |
|  |  |  |  |  |
| Q2 | Which of the following best illustrates the relationship between an image and a container? | | | 1 |
|  | a) Executable and its hard link |  |  |  |
|  | b) Executable and process |  |  |  |
|  | c) Parent and child process |  |  |  |
|  | d) Many to one |  |  |  |
|  | Enter your answer here | b) Executable and process |  |  |
|  |  |  |  |  |
| Q3 | What is the maximum amount of RAM a container can consume if the memory flag is not used? | | | 1 |
|  | a) 8GiB |  |  |  |
|  | b) 32GiB |  |  |  |
|  | c) None of these |  |  |  |
|  | d) As much as the host instance has free |  |  |  |
|  | Enter your answer here | d) As much as the host instance has free |  |  |
|  |  |  |  |  |
| Q4 | Which of the following will happen in the same Docker image is pushed to Docker Hub multiple times with different tags | | | 1 |
|  | a) Dockerhub will refuse to upload the image |  |  |  |
|  | b) The layers in the first image (if unchanged) will be reused in subsequent pushes |  |  |  |
|  | c) Dockerhub will merge the images |  |  |  |
|  | d) The same image cannot have multiple tags |  |  |  |
|  | Enter your answer here | c) Dockerhub will merge the images |  |  |
|  |  |  |  |  |
| Q5 | Which of the following will run a Docker container in interactive mode? | | | 1 |
|  | a) -v |  |  |  |
|  | b) -it |  |  |  |
|  | c) -b |  |  |  |
|  | d) -u |  |  |  |
|  | Enter your answer here | b) -it |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| Q6 | How would data persistence be handled in a container environment set up for autoscaling? | | | 4 |
|  |  | | |  |
|  | **Data persistence in containers?**   * A container can be in running state as long as its process is running. * Data persists if it outlasts the process that created it. * **By default, containers don't persist the data they produce.**   **How to provide data persistence in containers?**   * Containers are suitable for stateless applications because when they are terminated the data as well is destroyed. * Data persistence can be handled in containerized applications with the storage backend that isn’t destroyed when container terminates. * Containerized autoscaling applications requiring stateful containers can be developed using AWS elastic container service (ECS) that can use the aws storage services to provide data persistence to inherently ephemeral containers. * The suitable aws storage services to provide data persistence are EBS, EFS or FSx for windows file server.   **Now, how aws ECS can provide data persistence?**   * Data persistence in AWS is achieved by coupling compute and storage services. * Like EC2, ECS can be used the decouple the lifecycle of containerized applications from the data they consume and produce. * Using AWS storage services, ECS tasks can persist data even after tasks terminate. | | |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| Q7 | Why is this statement false? "Docker is the only popular choice for microservices deployment". | | | 4 |
|  |  |
|  | Containers are a means to deploy microservice architectures and Docker is a well-known containerization platform.  Though Docker is a popular technology to deploy microservices but it not the most popular or only one due to its following drawbacks:  • Containers don't run at bare-metal speeds. Containers consume resources more efficiently than virtual machines.  • The container ecosystem is fractured.  • Persistent data storage is complicated.  • Graphical applications don't work well.  • Not all applications benefit from containers.  Moreover, Microservices are about logical separation, not physical.  • it is not mandatory to use Docker in microservices architecture.  • We can design our system/Application and use microservices architecture and the final deployment can be pure hardware.  At the end, a microservice can be treated as a process that needs a host to run.  There are more other ways to deploy microservices:  REST is an architectural design pattern for building RESTful APIs. REST allows services to communicate directly via HTTP. Requests and responses are handled in standard formats like XML, HTML, or JSON. REST is a natural choice for most microservices, since many of them are Web Applications.  Redis, Prometheus and Consul with their associated set of advantages are some other tools to deploy microservices | | |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |