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.

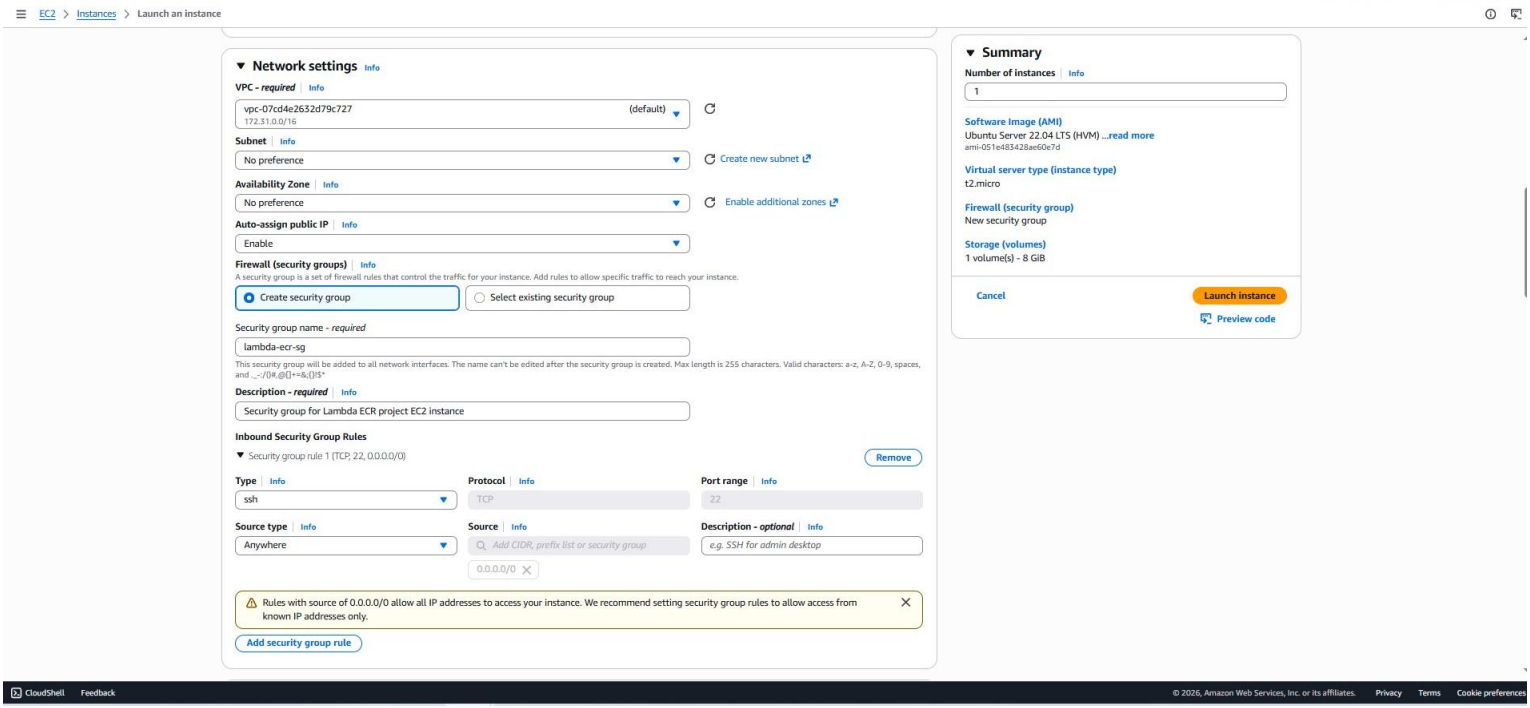
Architecture diagram

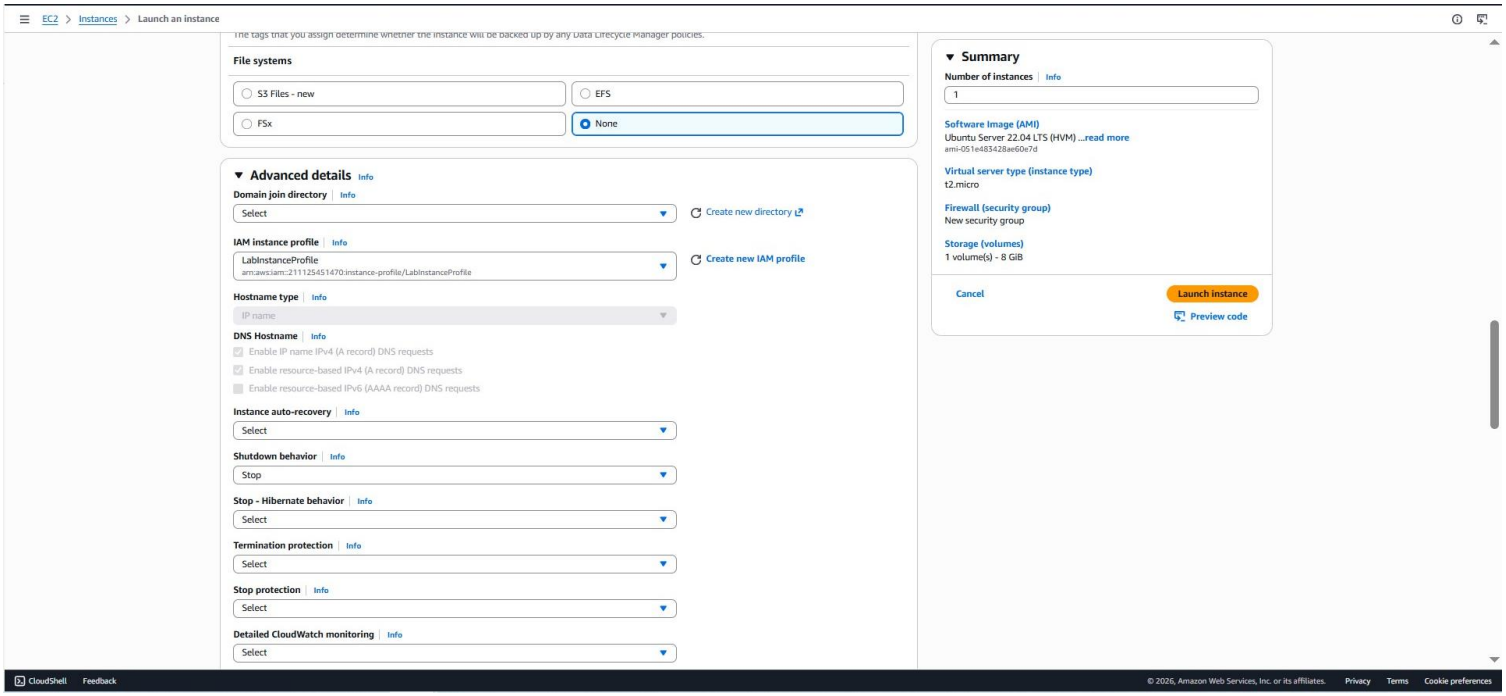
Architecture Implementation	
1	Download the Dockerfile and the app code folder provided with this workbook
2	Package the web application as a Docker image running on Alpine with Python
3	Create an ECR repository and login to it.
4	Build the image with the downloaded dockerfile and the support files
5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

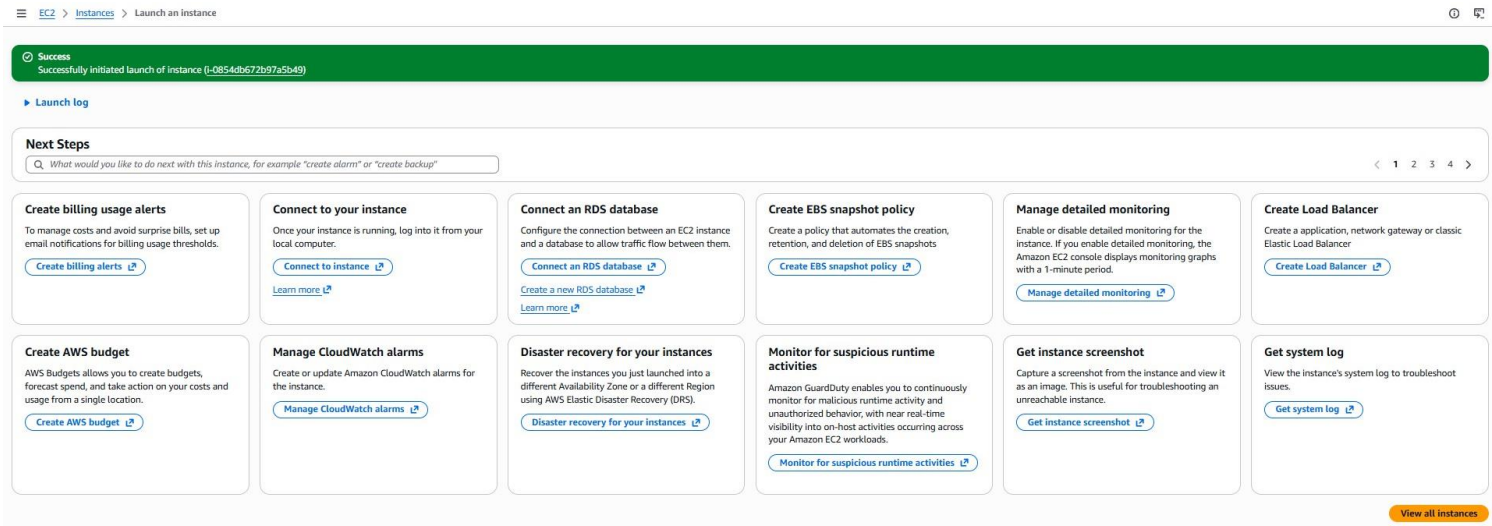
Step 1 : Docker Image creation

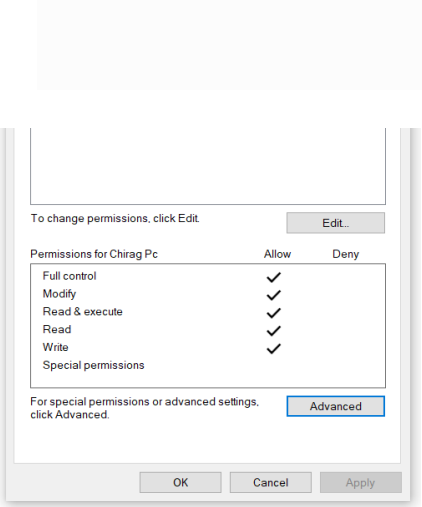
Step number	a
Step name	Creation of Docker image
Instructions	1) Create an EC2 instance using the Ubuntu 22.04 in the default VPC. 2) Attach the role “LabInstanceProfile” to the instance created above. This role is already present on your AWS Academy account. Note : For personal AWS accounts, create a new IAM role for EC2 and add the policy “EC2ContainerRegistryFullAccess” to it. 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 ubuntu@<public IP of instance>:/home/ubuntu (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 wget https://d6opu47qoi4ee.cloudfront.net/dockerinstallscript.sh bash ./dockerinstallscript.sh sudo apt install unzip unzip OCI.zip 5) **(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 apt 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** 6) Run the below command to create the Docker image *docker build -t lambda_ecr .* 7) Run the below command to verify the creation of the image *docker images*
Expected screenshots	1) Building the Docker image 2) List of the created images

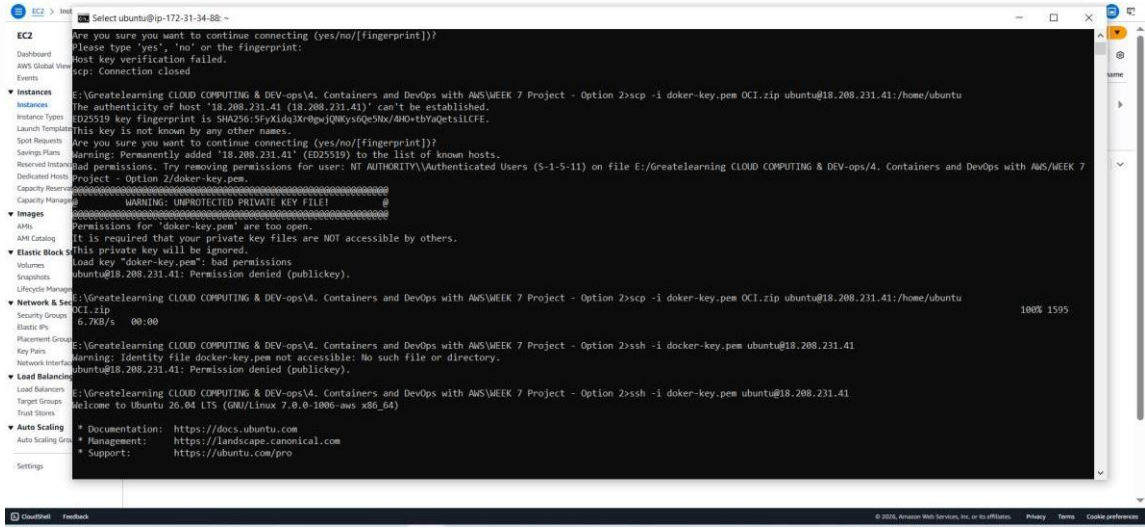
<Insert screenshot for a(1) here>

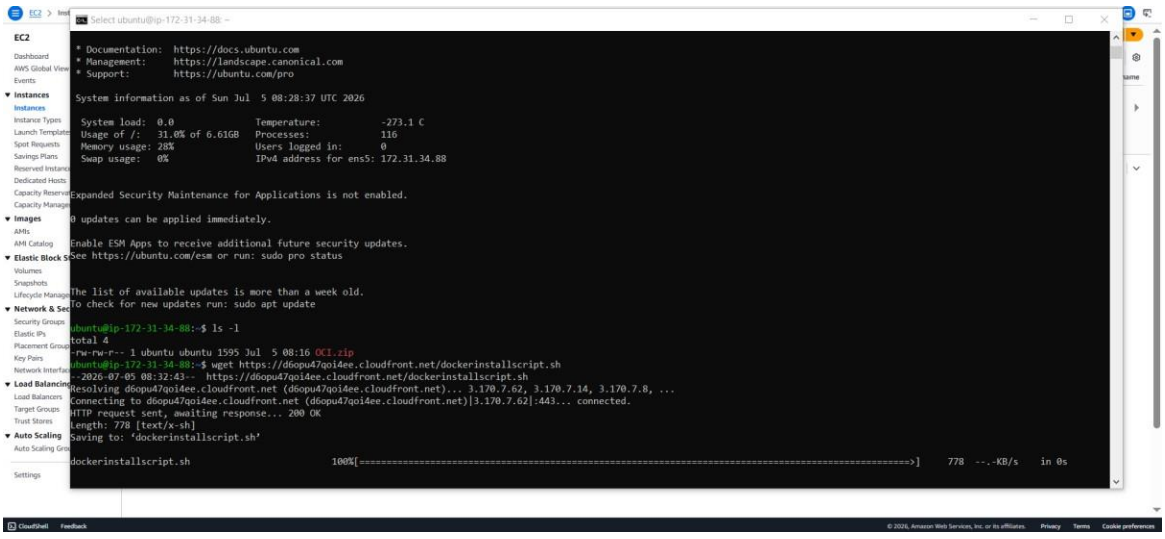


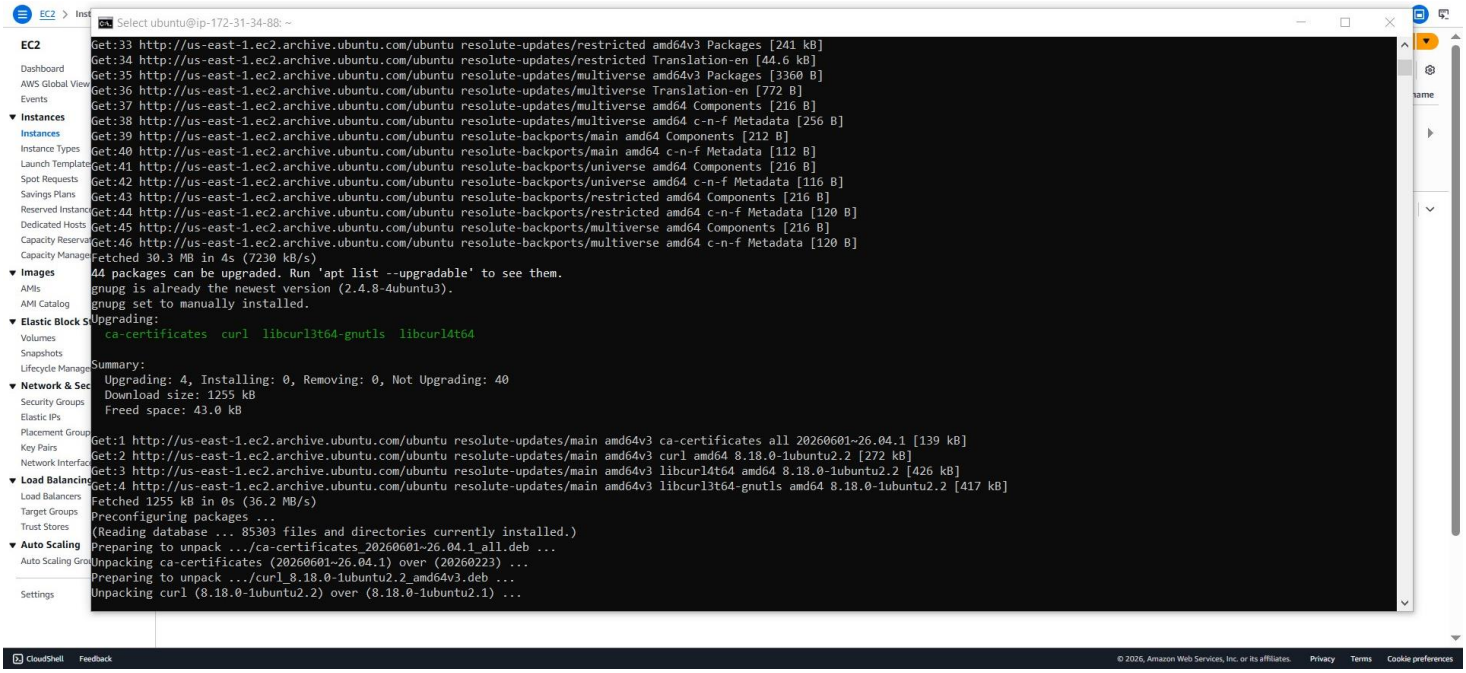


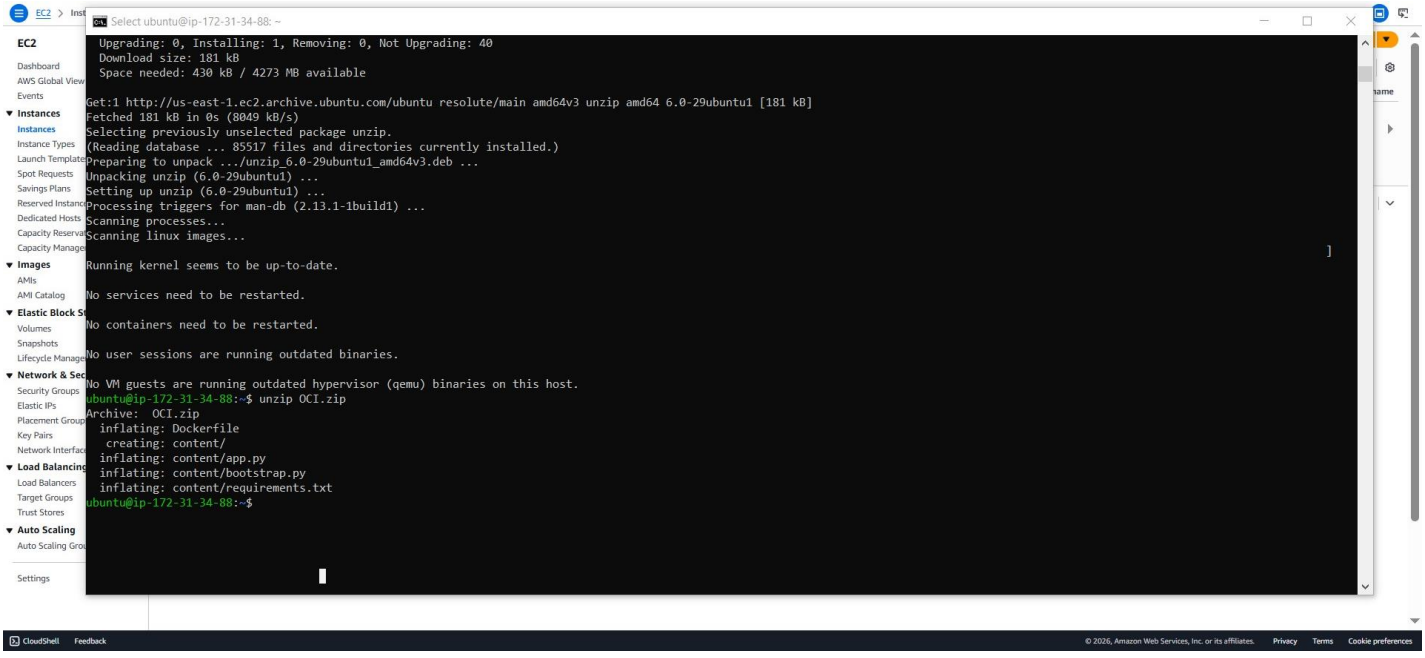

EC2 > Instances > Launch an instance
Success
Successfully initiated launch of instance (i-0854db672b97a5b49)
Launch log
Next Steps
What would you like to do next with this instance, for example "create alarm" or "create backup"
Create billing usage alerts
To manage costs and avoid surprise bills, set up email notifications for billing usage thresholds.
Create billing alerts
Connect to your instance
Once your instance is running, log into it from your local computer.
Connect to instance
Learn more
Connect an RDS database
Configure the connection between an EC2 instance and a database to allow traffic flow between them.
Connect an RDS database
Create a new RDS database
Learn more
Create EBS snapshot policy
Create a policy that automates the creation, retention, and deletion of EBS snapshots
Create EBS snapshot policy
Manage detailed monitoring
Enable or disable detailed monitoring for the instance. If you enable detailed monitoring, the Amazon EC2 console displays monitoring graphs with a 1-minute period.
Manage detailed monitoring
Create Load Balancer
Create a application, network gateway or classic Elastic Load Balancer
Create Load Balancer
Create AWS budget
AWS Budgets allows you to create budgets, forecast spend, and take action on your costs and usage from a single location.
Create AWS budget
Manage CloudWatch alarms
Create or update Amazon CloudWatch alarms for the instance.
Manage CloudWatch alarms
Disaster recovery for your instances
Recover the instances you just launched into a different Availability Zone or a different Region using AWS Elastic Disaster Recovery (DRS).
Disaster recovery for your instances
Monitor for suspicious runtime activities
Amazon GuardDuty enables you to continuously monitor for malicious runtime activity and unauthorized behavior, with near real-time visibility into on-host activities occurring across your Amazon EC2 workloads.
Monitor for suspicious runtime activities
Get instance screenshot
Capture a screenshot from the instance and view it as an image. This is useful for troubleshooting an unreachable instance.
Get instance screenshot
Get system log
View the instance's system log to troubleshoot issues.
Get system log
View all instances

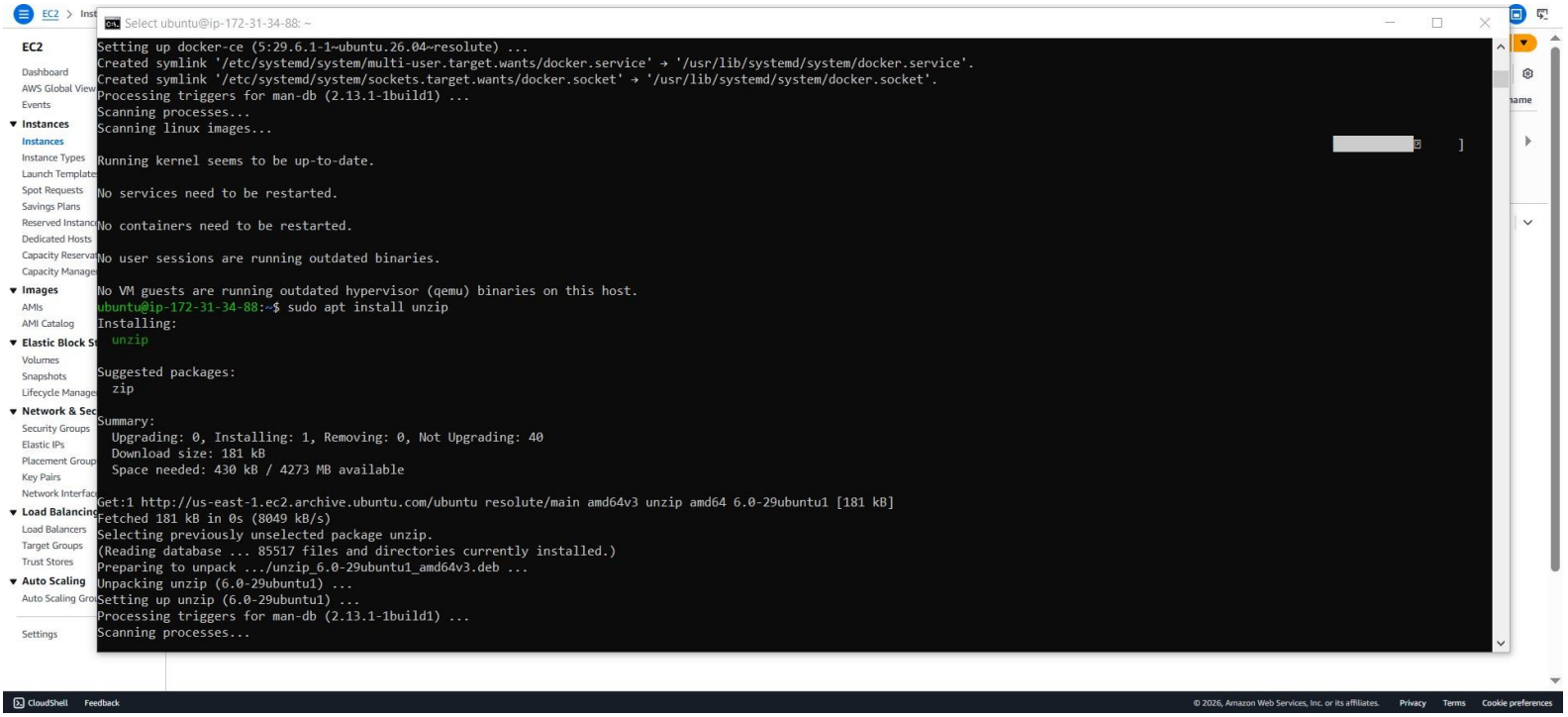

To change permissions, click Edit.
Edit...
Permissions for Chirag Pc
Allow
Deny
Full control
Modify
Read & execute
Read
Write
Special permissions
For special permissions or advanced settings, click Advanced.
Advanced
OK
Cancel
Apply


Select ubuntu@ip-172-31-34-88: ~
Are you sure you want to continue connecting (yes/no/[fingerprint])?
Please type 'yes', 'no' or the fingerprint:
Host key verification failed.
lost connection
E:\Greatelearning CLOUD COMPUTING & DEV-ops\4. Containers and DevOps with AWS\WEEK 7 Project - Option 2>scp -i doker-key.pem OCI.zip ubuntu@18.208.231.41:/home/ubuntu
The authenticity of host '18.208.231.41 (18.208.231.41)' can't be established.
ED25519 key fingerprint is SHA256:5FyXidq3Xr0gwjQNKys6Qe5Nx/4HO+tbYaQetsiLCFE.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
Warning: Permanently added '18.208.231.41' (ED25519) to the list of known hosts.
Bad permissions. Try removing permissions for user: NT AUTHORITY\\Authenticated Users (S-1-5-11) on file E:/Greatelearning CLOUD COMPUTING & DEV-ops/4. Containers and DevOps with AWS/WEEK 7 Project - Option 2/doker-key.pem.
@ WARNING: UNPROTECTED PRIVATE KEY FILE! @
Permissions for 'doker-key.pem' are too open.
It is required that your private key files are NOT accessible by others.
This private key will be ignored.
Load key "doker-key.pem": bad permissions
ubuntu@18.208.231.41: Permission denied (publickey).
E:\Greatelearning CLOUD COMPUTING & DEV-ops\4. Containers and DevOps with AWS\WEEK 7 Project - Option 2>scp -i doker-key.pem OCI.zip ubuntu@18.208.231.41:/home/ubuntu
OCI.zip 100% 1595 6.7KB/s 00:00
E:\Greatelearning CLOUD COMPUTING & DEV-ops\4. Containers and DevOps with AWS\WEEK 7 Project - Option 2>ssh -i docker-key.pem ubuntu@18.208.231.41
Warning: Identity file docker-key.pem not accessible: No such file or directory.
ubuntu@18.208.231.41: Permission denied (publickey).
E:\Greatelearning CLOUD COMPUTING & DEV-ops\4. Containers and DevOps with AWS\WEEK 7 Project - Option 2>ssh -i doker-key.pem ubuntu@18.208.231.41
Welcome to Ubuntu 26.04 LTS (GNU/Linux 7.0.0-1006-aws x86_64)
* Documentation: https://docs.ubuntu.com
* Management: https://landscape.canonical.com
* Support: https://ubuntu.com/pro

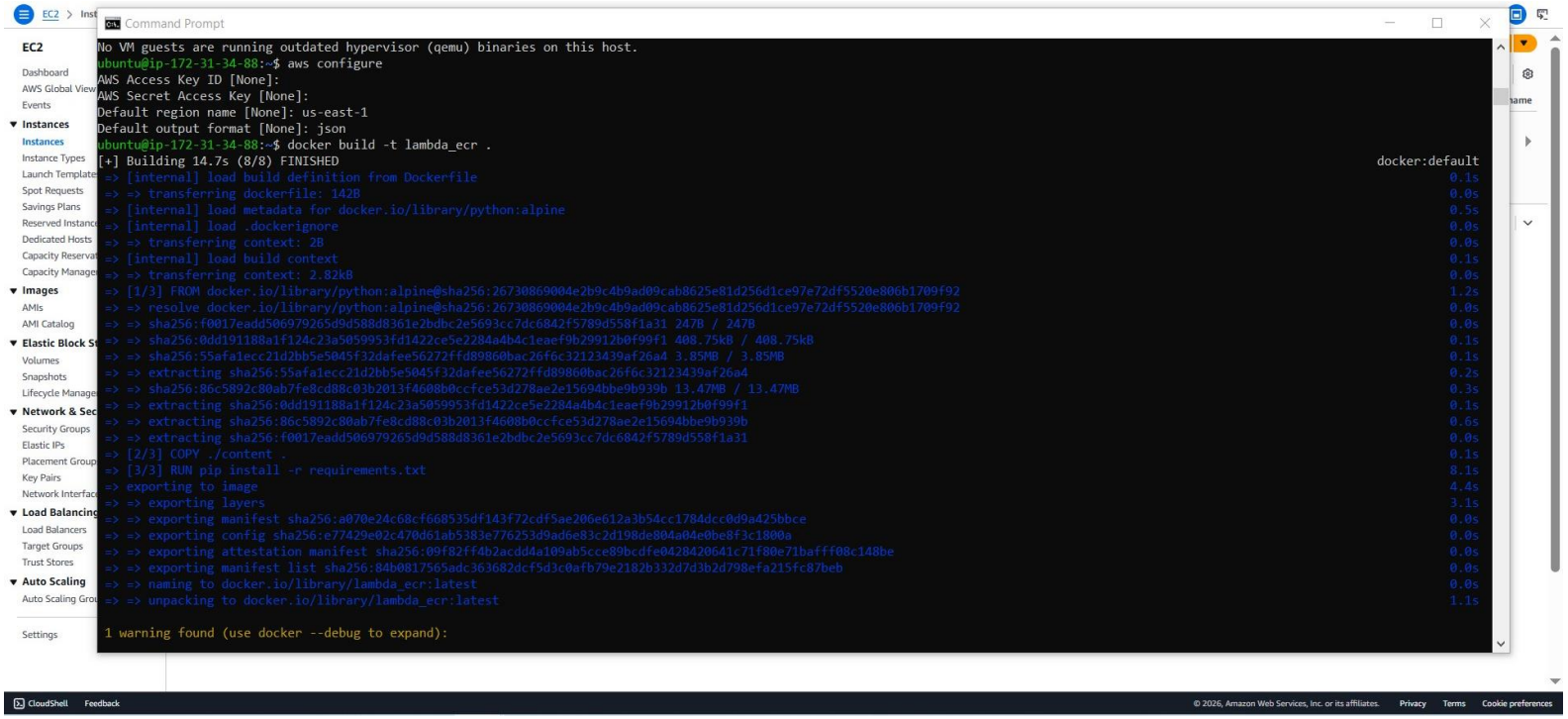

Select ubuntu@ip-172-31-34-88: ~
* Documentation: https://docs.ubuntu.com
* Management: https://landscape.canonical.com
* Support: https://ubuntu.com/pro
System information as of Sun Jul 5 08:28:37 UTC 2026
System load: 0.0 Temperature: -273.1 C
Usage of /: 31.0% of 6.61GB Processes: 116
Memory usage: 28% Users logged in: 0
Swap usage: 0% IPv4 address for ens5: 172.31.34.88
Expanded Security Maintenance for Applications is not enabled.
0 updates can be applied immediately.
Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status
The list of available updates is more than a week old.
To check for new updates run: sudo apt update
ubuntu@ip-172-31-34-88:~$ ls -l
total 4
-rw-rw-r-- 1 ubuntu ubuntu 1595 Jul 5 08:16 OCI.zip
ubuntu@ip-172-31-34-88:~$ wget https://d6opu47qoi4ee.cloudfront.net/dockerinstallscript.sh
--2026-07-05 08:32:43-- https://d6opu47qoi4ee.cloudfront.net/dockerinstallscript.sh
Resolving d6opu47qoi4ee.cloudfront.net (d6opu47qoi4ee.cloudfront.net)... 3.170.7.62, 3.170.7.14, 3.170.7.8, ...
Connecting to d6opu47qoi4ee.cloudfront.net (d6opu47qoi4ee.cloudfront.net)|3.170.7.62|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 778 [text/x-sh]
Saving to: 'dockerinstallscript.sh'
dockerinstallscript.sh 100%[==========>] 778 --.-KB/s in 0s


Select ubuntu@ip-172-31-34-88: ~
Get:33 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-updates/restricted amd64v3 Packages [241 kB]
Get:34 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-updates/restricted Translation-en [44.6 kB]
Get:35 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-updates/multiverse amd64v3 Packages [3360 B]
Get:36 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-updates/multiverse Translation-en [772 B]
Get:37 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-updates/multiverse amd64 Components [216 B]
Get:38 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-updates/multiverse amd64 c-n-f Metadata [256 B]
Get:39 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-backports/main amd64 Components [212 B]
Get:40 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-backports/main amd64 c-n-f Metadata [112 B]
Get:41 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-backports/universe amd64 Components [216 B]
Get:42 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-backports/universe amd64 c-n-f Metadata [116 B]
Get:43 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-backports/restricted amd64 Components [216 B]
Get:44 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-backports/restricted amd64 c-n-f Metadata [120 B]
Get:45 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-backports/multiverse amd64 Components [216 B]
Get:46 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-backports/multiverse amd64 c-n-f Metadata [120 B]
Fetched 30.3 MB in 4s (7230 kB/s)
44 packages can be upgraded. Run 'apt list --upgradable' to see them.
gnupg is already the newest version (2.4.8-4ubuntu3).
gnupg set to manually installed.
Upgrading:
ca-certificates curl libcurl3t64-gnutls libcurl4t64
Summary:
Upgrading: 4, Installing: 0, Removing: 0, Not Upgrading: 40
Download size: 1255 kB
Freed space: 43.0 kB
Get:1 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-updates/main amd64v3 ca-certificates all 20260601~26.04.1 [139 kB]
Get:2 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-updates/main amd64v3 curl amd64 8.18.0-1ubuntu2.2 [272 kB]
Get:3 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-updates/main amd64v3 libcurl4t64 amd64 8.18.0-1ubuntu2.2 [426 kB]
Get:4 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute-updates/main amd64v3 libcurl3t64-gnutls amd64 8.18.0-1ubuntu2.2 [417 kB]
Fetched 1255 kB in 0s (36.2 MB/s)
Preconfiguring packages ...
(Reading database ... 85303 files and directories currently installed.)
Preparing to unpack .../ca-certificates_20260601~26.04.1_all.deb ...
Unpacking ca-certificates (20260601~26.04.1) over (20260223) ...
Preparing to unpack .../curl_8.18.0-1ubuntu2.2_amd64v3.deb ...
Unpacking curl (8.18.0-1ubuntu2.2) over (8.18.0-1ubuntu2.1) ...
EC2
Dashboard
AWS Global View
Events
Instances
Instances
Instance Types
Launch Templates
Spot Requests
Savings Plans
Reserved Instances
Dedicated Hosts
Capacity Reservations
Capacity Manager
Images
AMIs
AMI Catalog
Elastic Block Store
Volumes
Snapshots
Lifecycle Manager
Network & Security
Security Groups
Elastic IPs
Placement Groups
Key Pairs
Network Interfaces
Load Balancing
Load Balancers
Target Groups
Trust Stores
Auto Scaling
Auto Scaling Groups
Settings
CloudShell
Feedback
© 2026, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

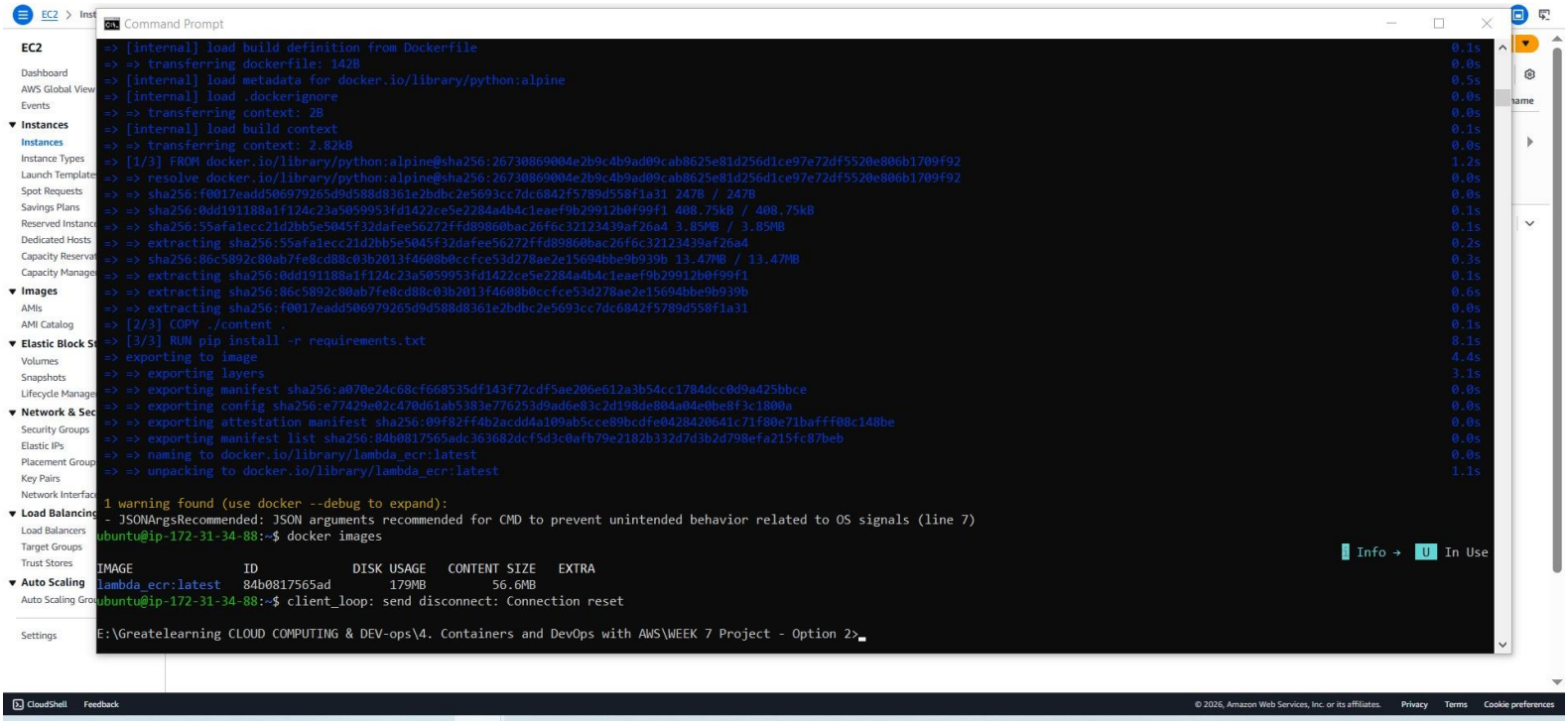

Select ubuntu@ip-172-31-34-88: ~
Upgrading: 0, Installing: 1, Removing: 0, Not Upgrading: 40
Download size: 181 kB
Space needed: 430 kB / 4273 MB available
Get:1 http://us-east-1.ec2.archive.ubuntu.com/ubuntu resolute/main amd64v3 unzip amd64 6.0-29ubuntu1 [181 kB]
Fetched 181 kB in 0s (8049 kB/s)
Selecting previously unselected package unzip.
(Reading database ... 85517 files and directories currently installed.)
Preparing to unpack .../unzip_6.0-29ubuntu1_amd64v3.deb ...
Unpacking unzip (6.0-29ubuntu1) ...
Setting up unzip (6.0-29ubuntu1) ...
Processing triggers for man-db (2.13.1-1build1) ...
Scanning processes...
Scanning linux images...
Running kernel seems to be up-to-date.
No services need to be restarted.
No containers need to be restarted.
No user sessions are running outdated binaries.
No VM guests are running outdated hypervisor (qemu) binaries on this host.
ubuntu@ip-172-31-34-88:~$ unzip OCI.zip
Archive: OCI.zip
inflating: Dockerfile
creating: content/
inflating: content/app.py
inflating: content/bootstrap.py
inflating: content/requirements.txt
ubuntu@ip-172-31-34-88:~$
EC2
Dashboard
AWS Global View
Events
Instances
Instances
Instance Types
Launch Templates
Spot Requests
Savings Plans
Reserved Instances
Dedicated Hosts
Capacity Reservations
Capacity Manager
Images
AMIs
AMI Catalog
Elastic Block Store
Volumes
Snapshots
Lifecycle Manager
Network & Security
Security Groups
Elastic IPs
Placement Groups
Key Pairs
Network Interfaces
Load Balancing
Load Balancers
Target Groups
Trust Stores
Auto Scaling
Auto Scaling Groups
Settings
CloudShell
Feedback
© 2026, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences



<Insert screenshot for a(1) here>



<Insert screenshot for a(2) here>

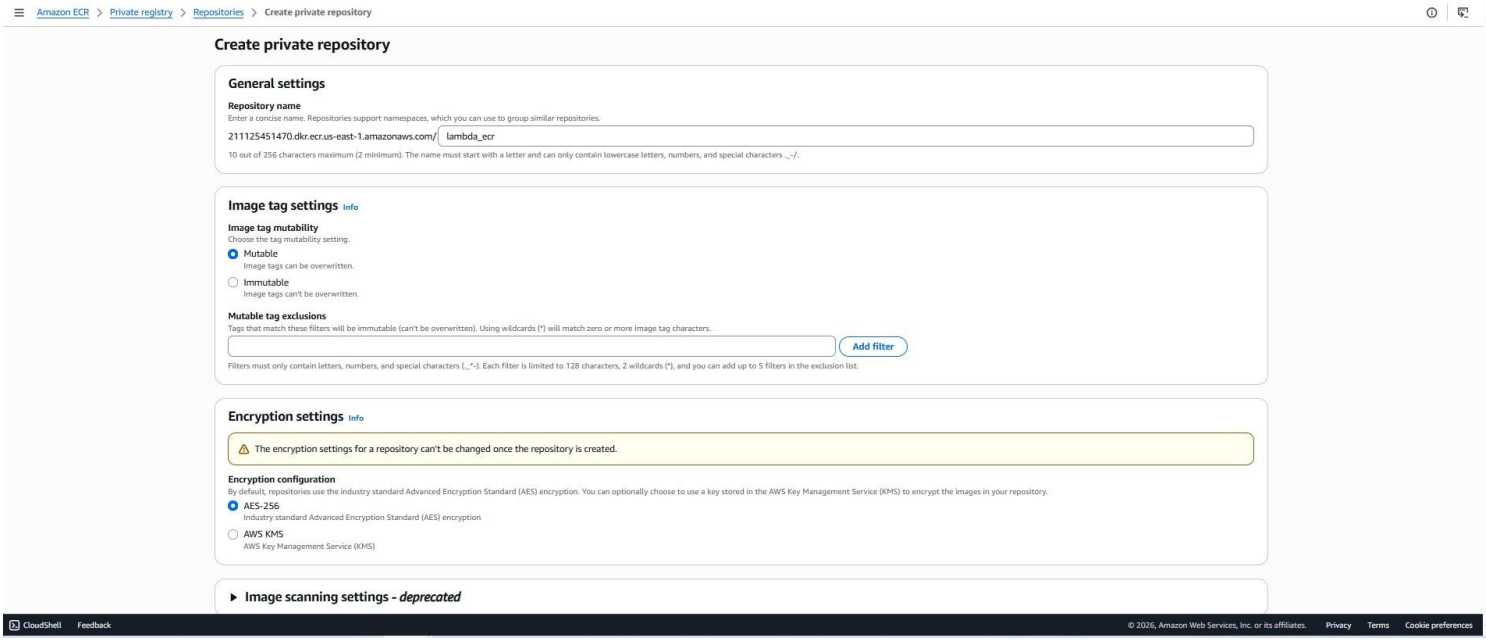


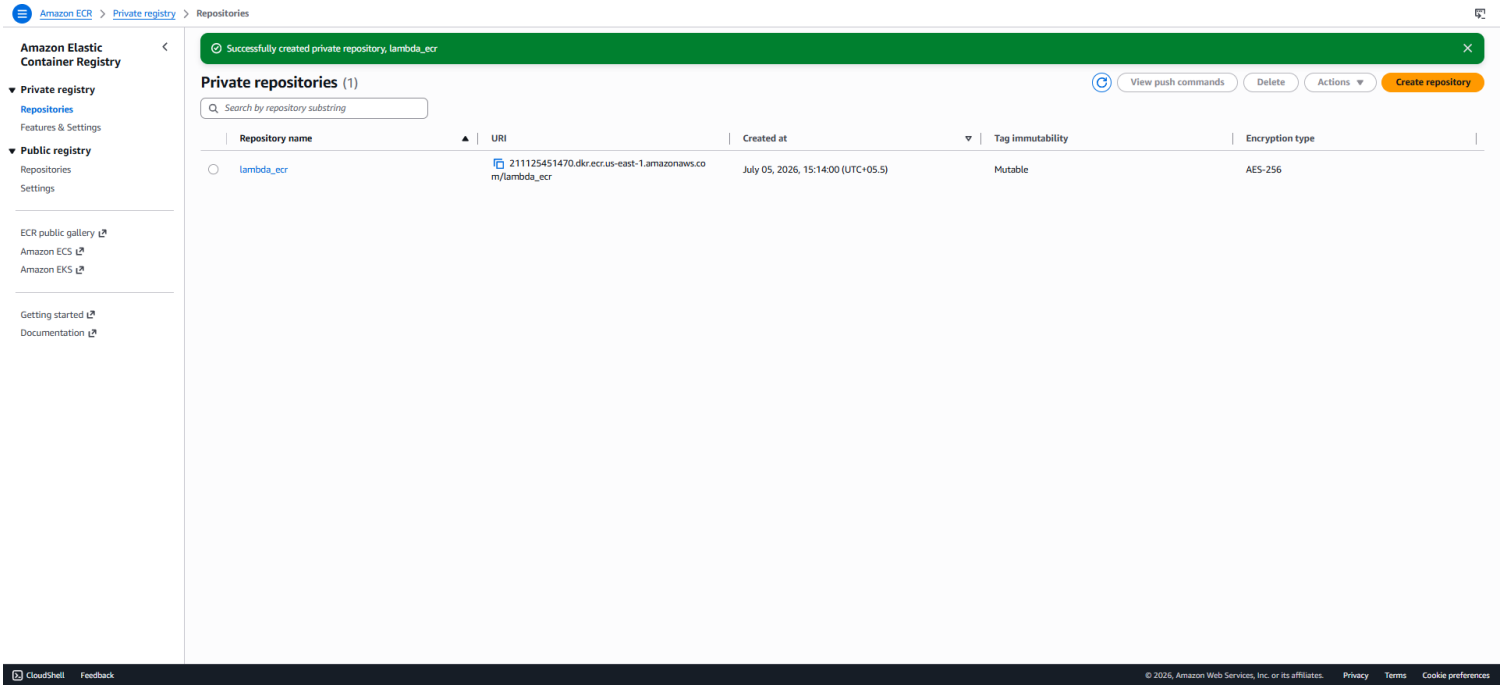
Step 2: Create ECR repository and upload image to ECR

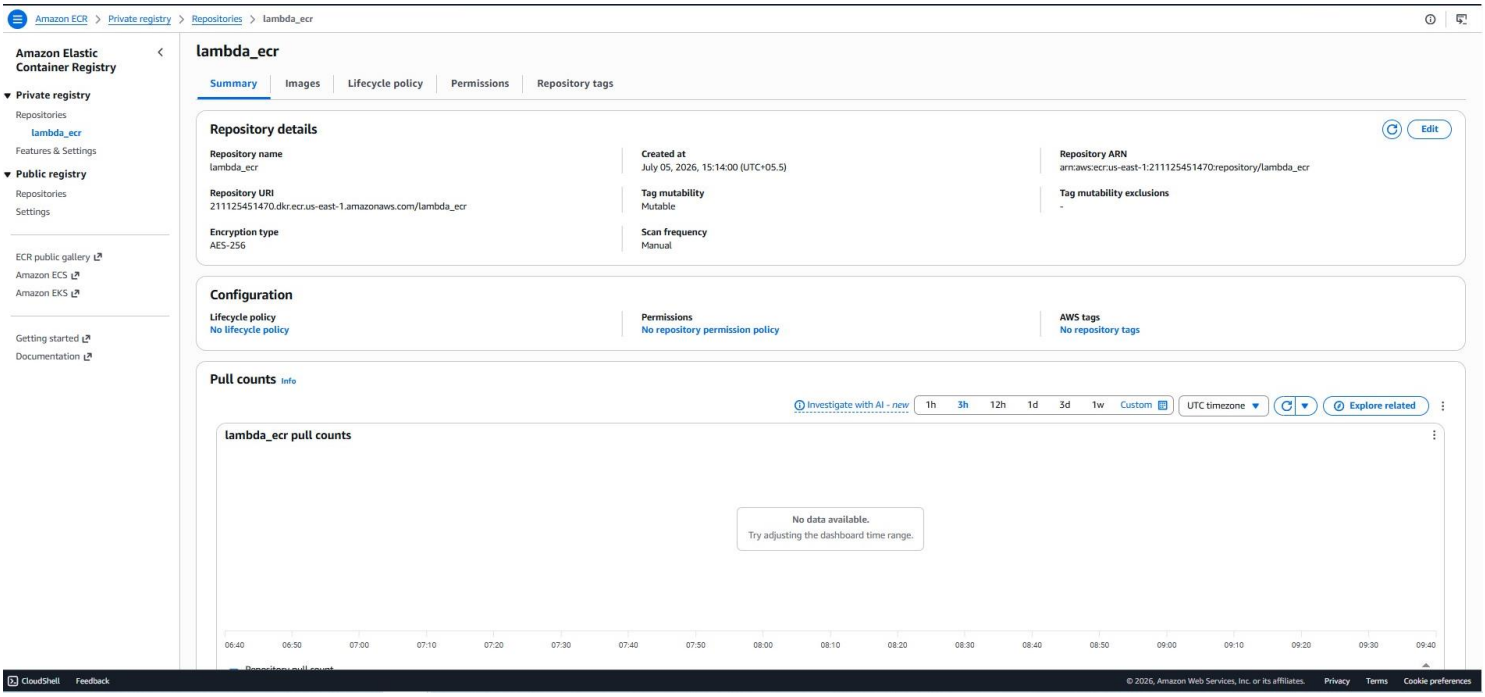
Step number	a
Step name	Creating the ECR repository
Instructions	1) Go to the ECR service on the AWS console 2) Select the Repositories from the left pane 3) Create a new private repository named **lambda_ecr** with the default settings
Step number	b
Step name	Image upload to ECR
Instructions	1) Once 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

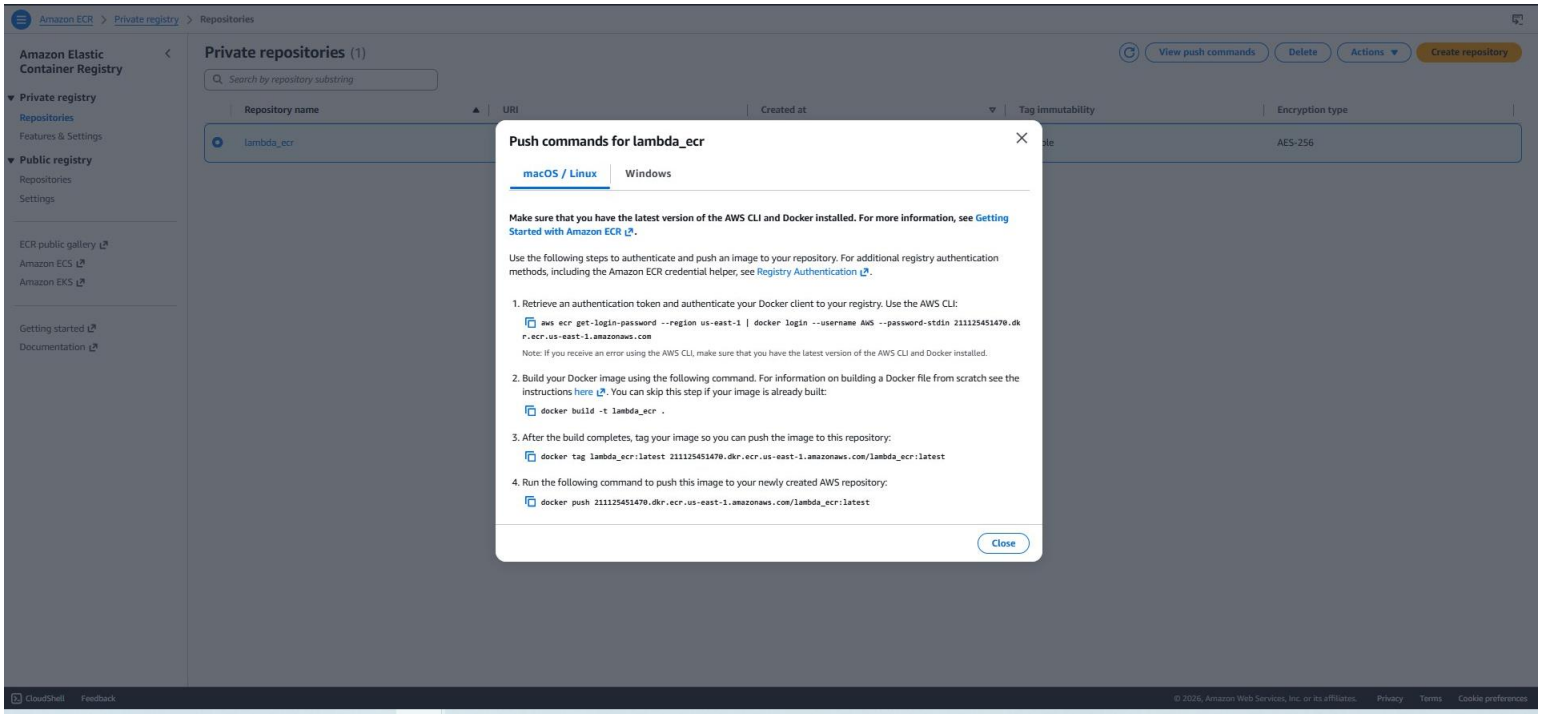
<Insert screenshot for a(1) here>



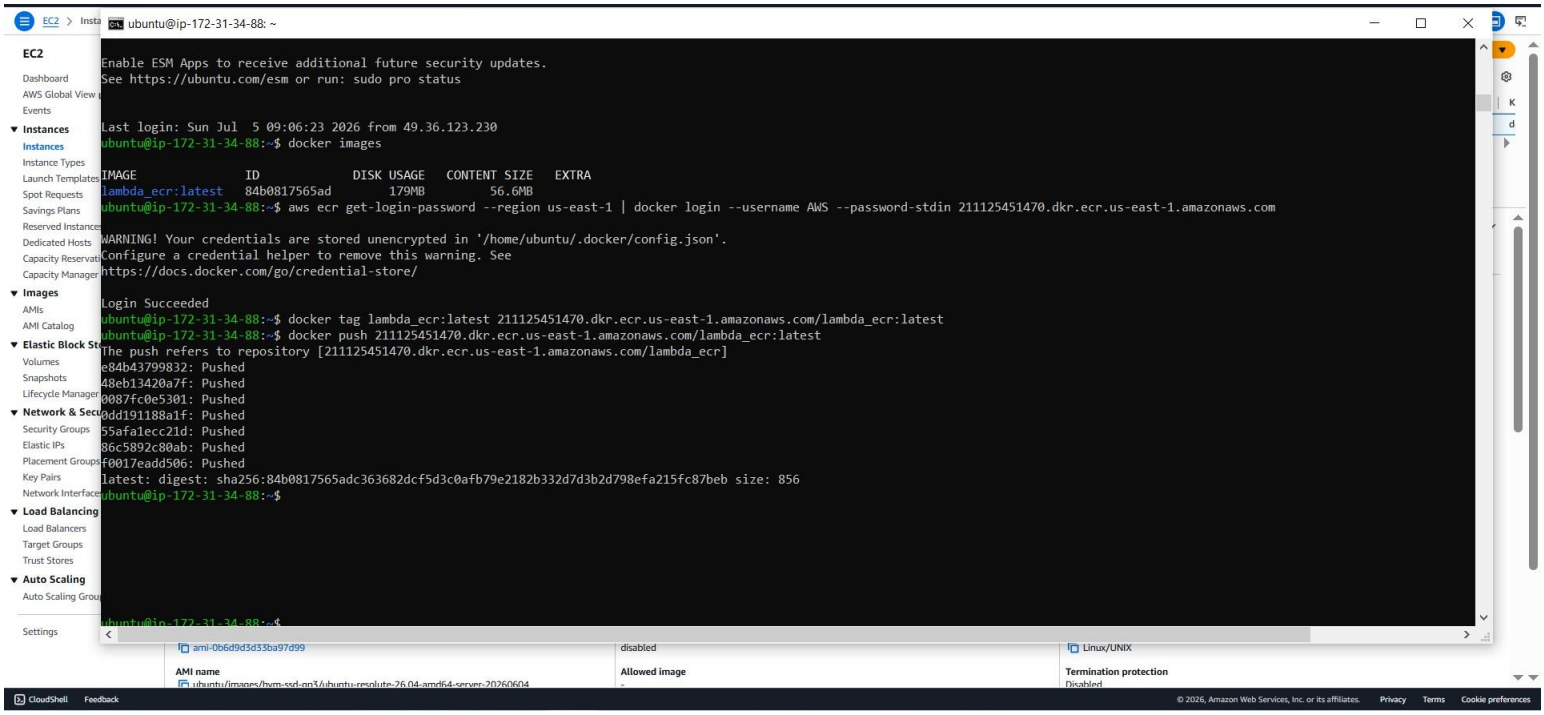

Amazon ECR > Private registry > Repositories
Amazon Elastic Container Registry
Private registry
Repositories
Features & Settings
Public registry
Repositories
Settings
ECR public gallery
Amazon ECS
Amazon EKS
Getting started
Documentation
Successfully created private repository, lambda_ecr
Private repositories (1)
View push commands
Delete
Actions
Create repository
Search by repository substring
Repository name
URI
Created at
Tag immutability
Encryption type
lambda_ecr
211125451470.dkr.ecr.us-east-1.amazonaws.com/lambda_ecr
July 05, 2026, 15:14:00 (UTC+05.5)
Mutable
AES-256
CloudShell
Feedback
© 2026, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


Amazon ECR > Private registry > Repositories > lambda_ecr
Amazon Elastic Container Registry
Private registry
Repositories
lambda_ecr
Features & Settings
Public registry
Repositories
Settings
ECR public gallery
Amazon ECS
Amazon EKS
Getting started
Documentation
lambda_ecr
Summary
Images
Lifecycle policy
Permissions
Repository tags
Repository details
Edit
Repository name
lambda_ecr
Created at
July 05, 2026, 15:14:00 (UTC+05.5)
Repository ARN
arn:aws:ecr:us-east-1:211125451470:repository/lambda_ecr
Repository URI
211125451470.dkr.ecr.us-east-1.amazonaws.com/lambda_ecr
Tag mutability
Mutable
Tag mutability exclusions
-
Encryption type
AES-256
Scan frequency
Manual
Configuration
Lifecycle policy
No lifecycle policy
Permissions
No repository permission policy
AWS tags
No repository tags
Pull counts
Info
Investigate with AI - new
1h
3h
12h
1d
3d
1w
Custom
UTC timezone
Explore related
lambda_ecr pull counts
No data available.
Try adjusting the dashboard time range.
06:40
06:50
07:00
07:10
07:20
07:30
07:40
07:50
08:00
08:10
08:20
08:30
08:40
08:50
09:00
09:10
09:20
09:30
09:40
CloudShell
Feedback
© 2026, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

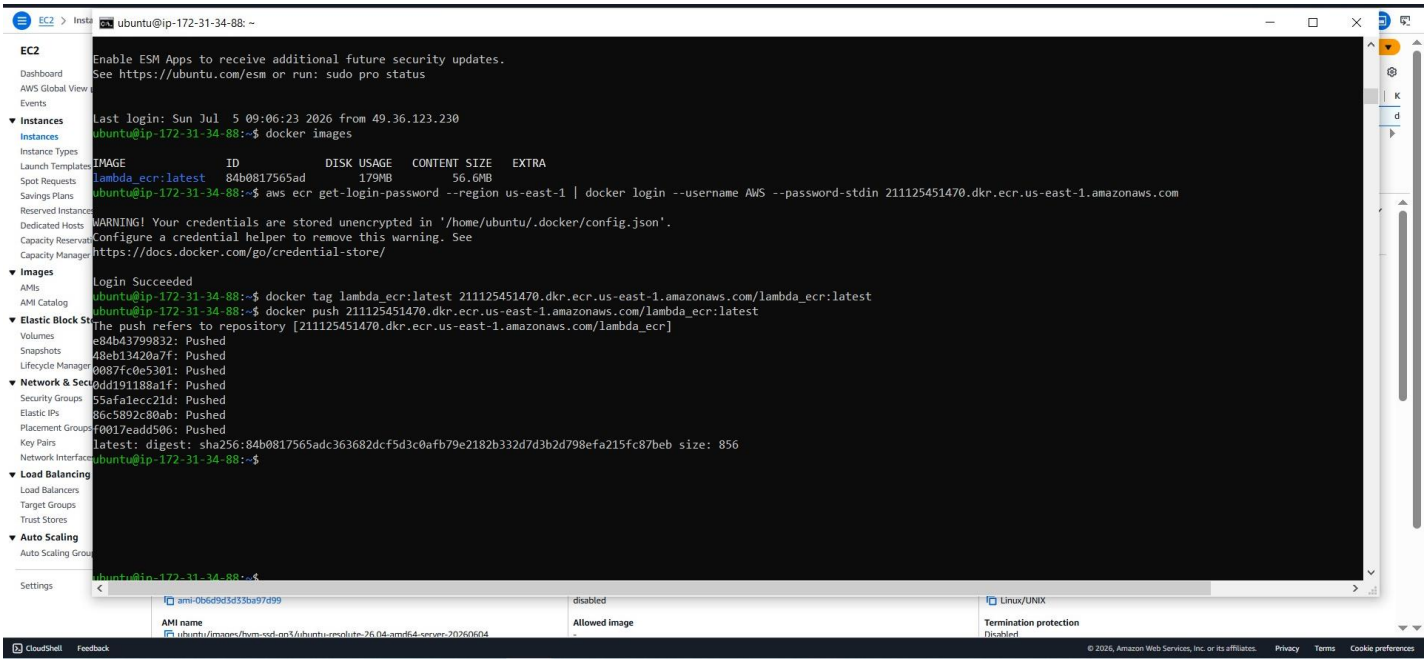
<Insert screenshot for b(1) here>



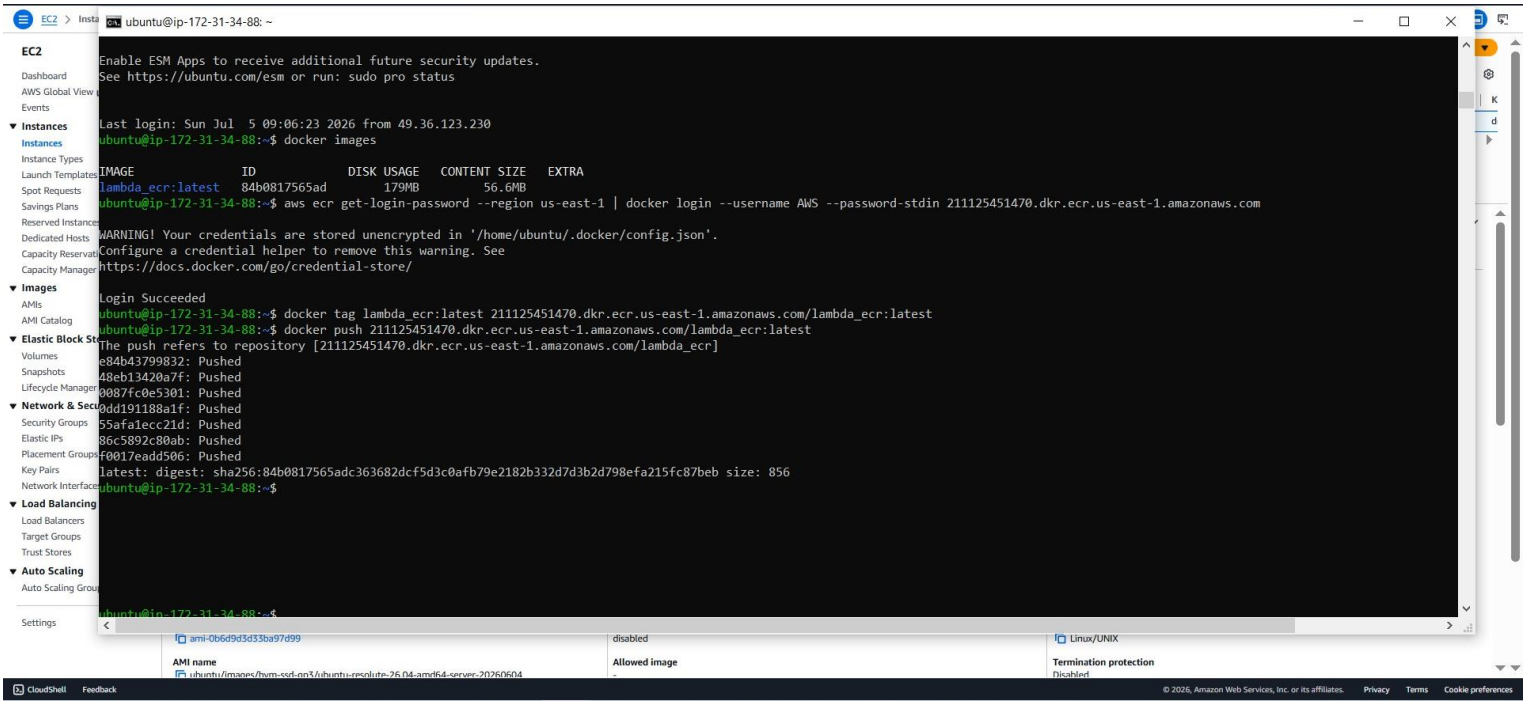
<Insert screenshot for b(2) here>



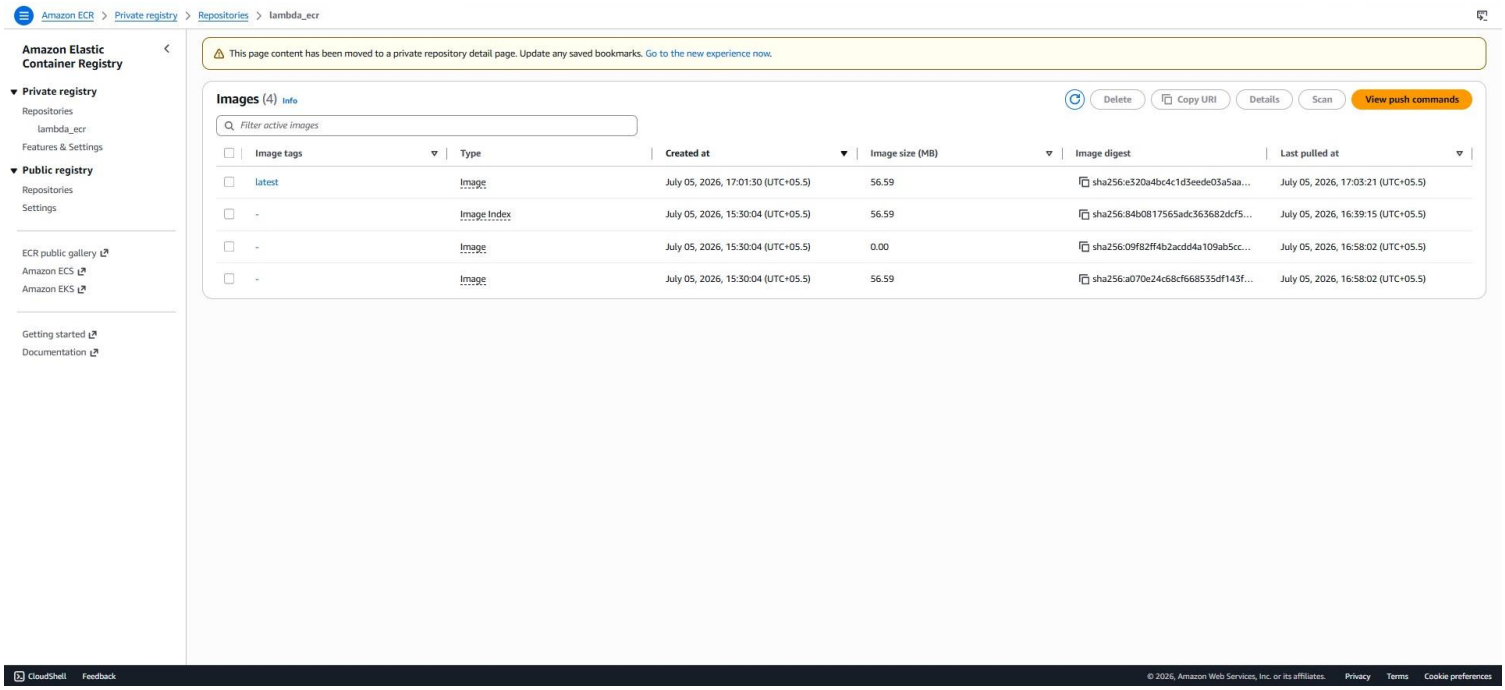
<Insert screenshot for b(3) here>

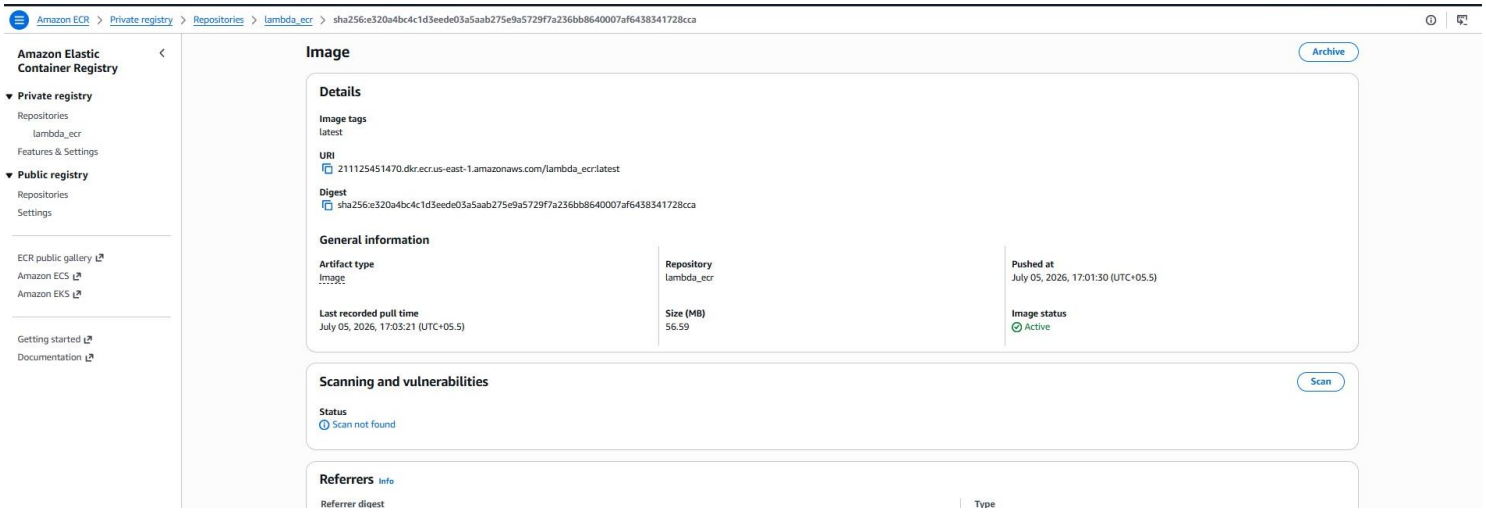


<Insert screenshot for b(4) here>



<Insert screenshot for b(5) here>

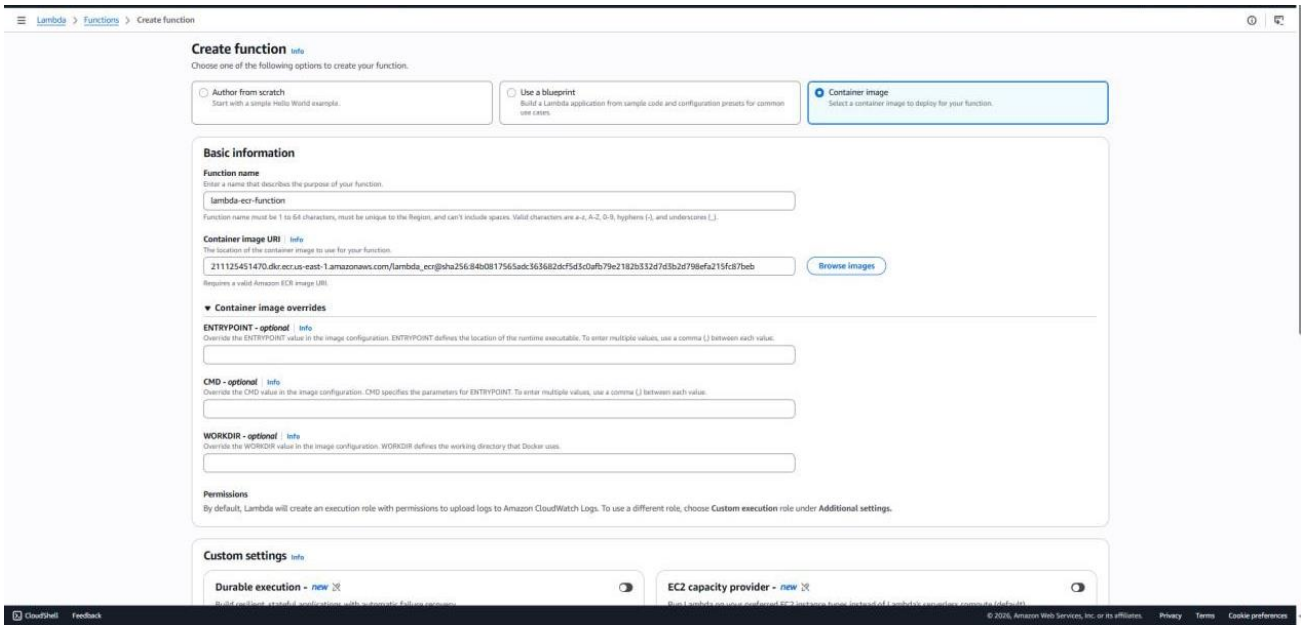


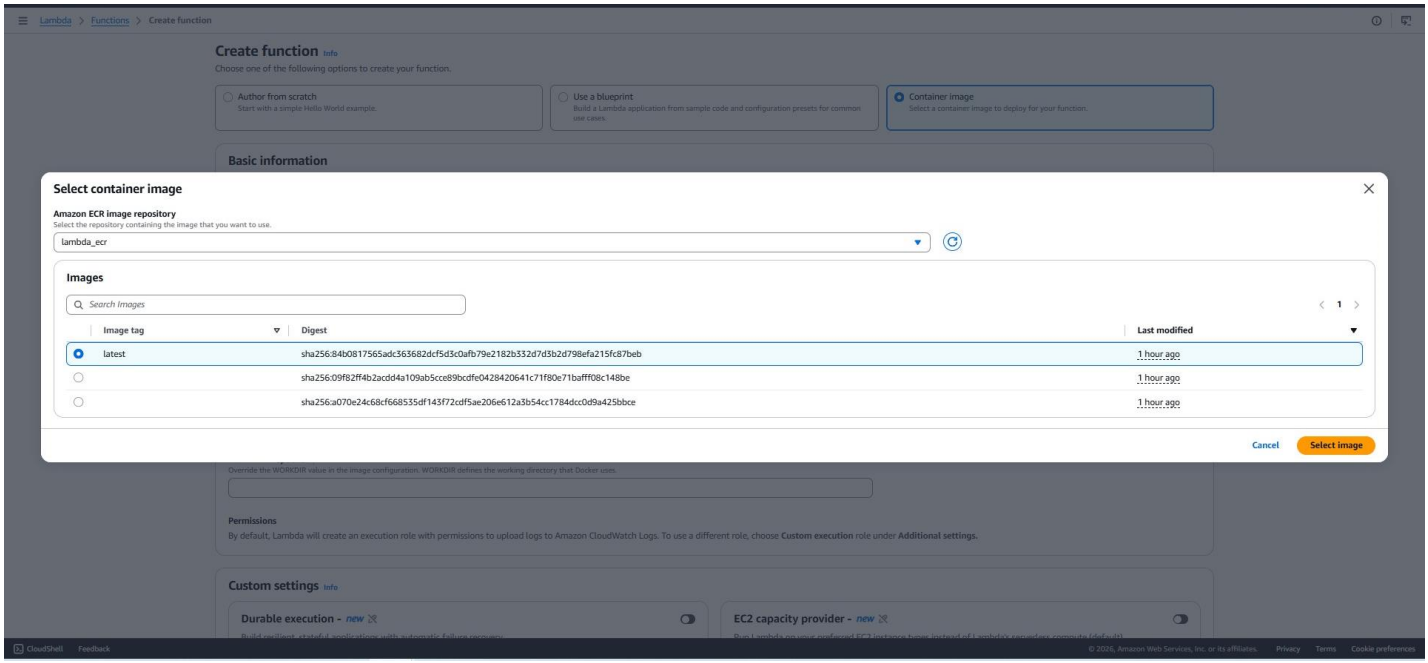


Step 3: Creation of Lambda function to test the image

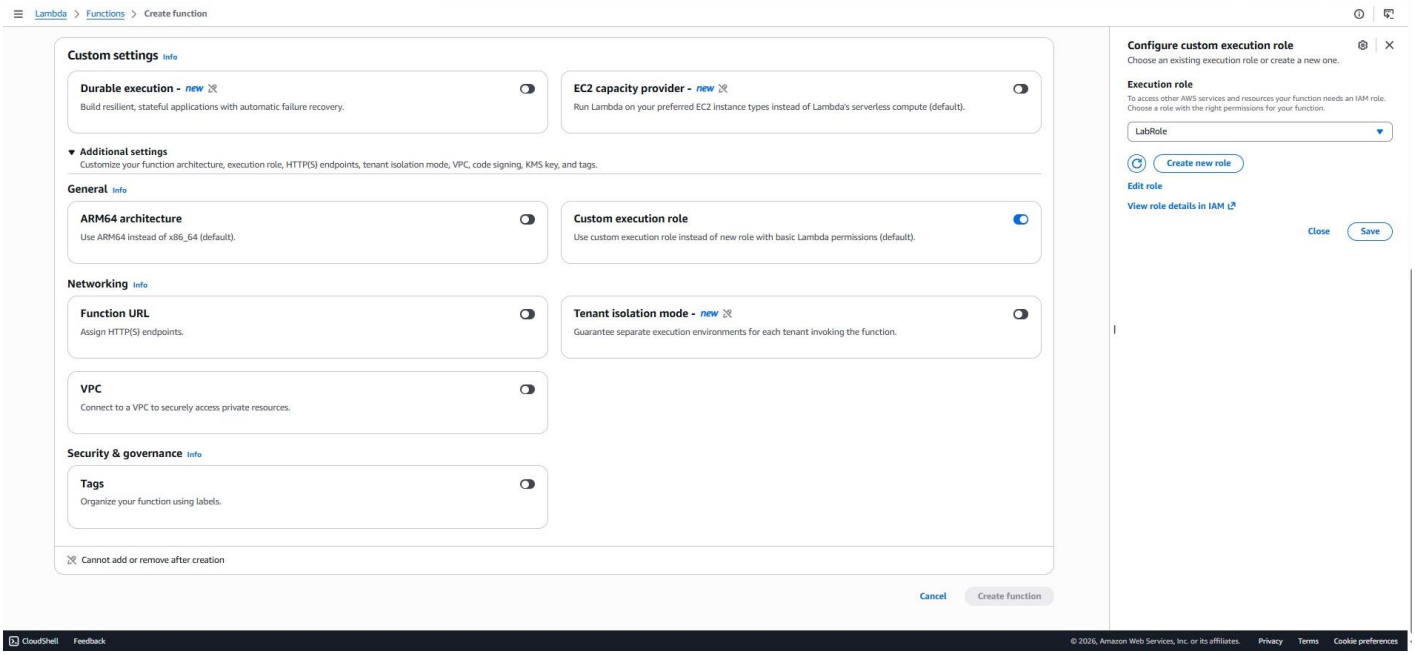
Step number	a
Step name	Create the Lambda function and test the image
Instructions	1) Navigate to the AWS Lambda service using the AWS Console 2) Click on **Create Function** 3) Under Create Function page select the 'Container image' option and enter a function name of your choice 4) For 'Container image URI' Click on "Browse Images" and select the repository and the image 5) Use the existing IAM role – LabRole. This role is already present on your AWS Academy accounts. Note : For personal AWS accounts, create a new IAM role for EC2 and add the policy "EC2ContainerRegistryFullAccess" to it. 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

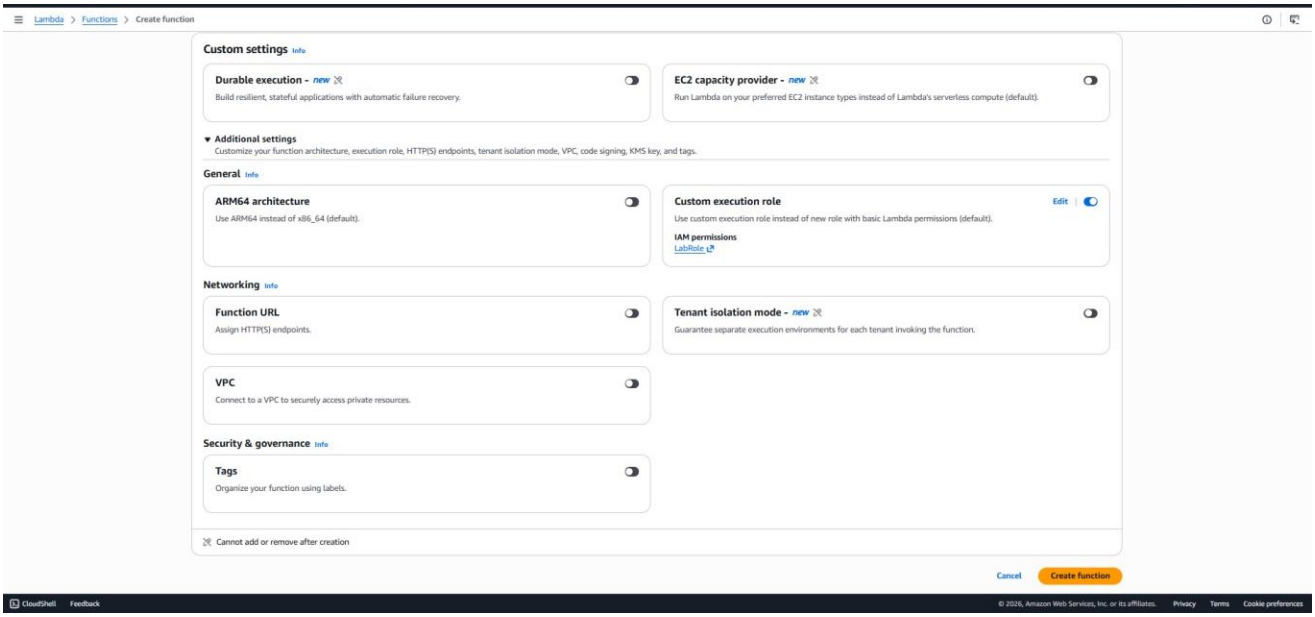
<Insert screenshot for a(1) here>



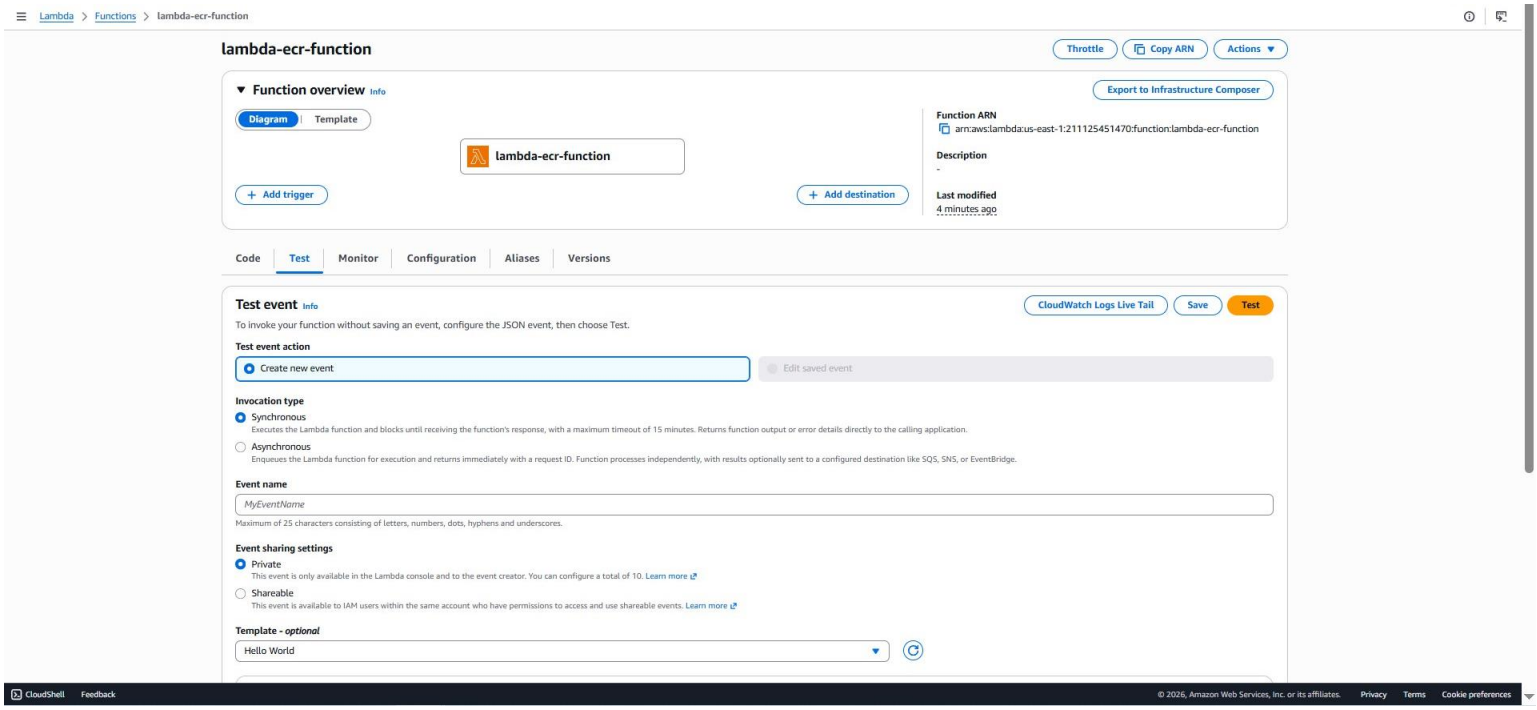


<Insert screenshot for a(2) here>

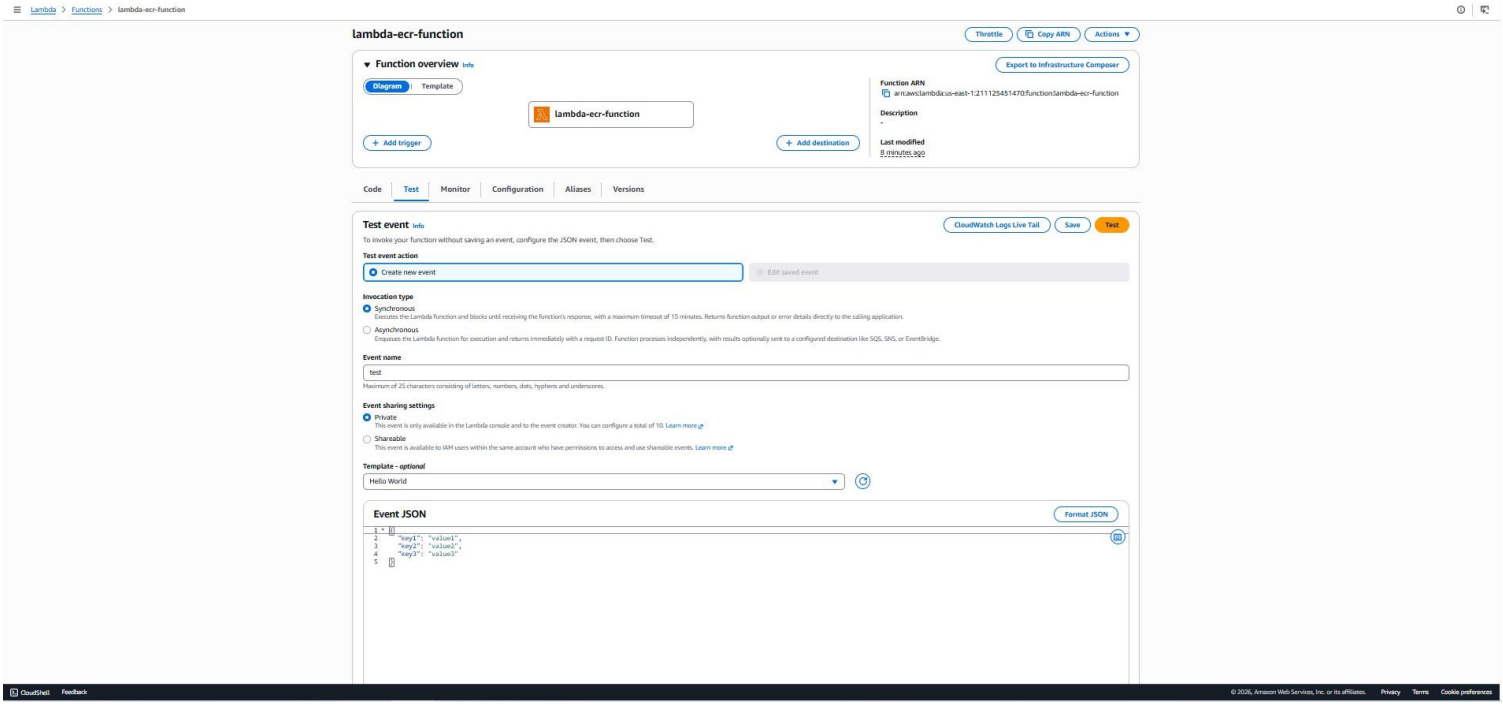


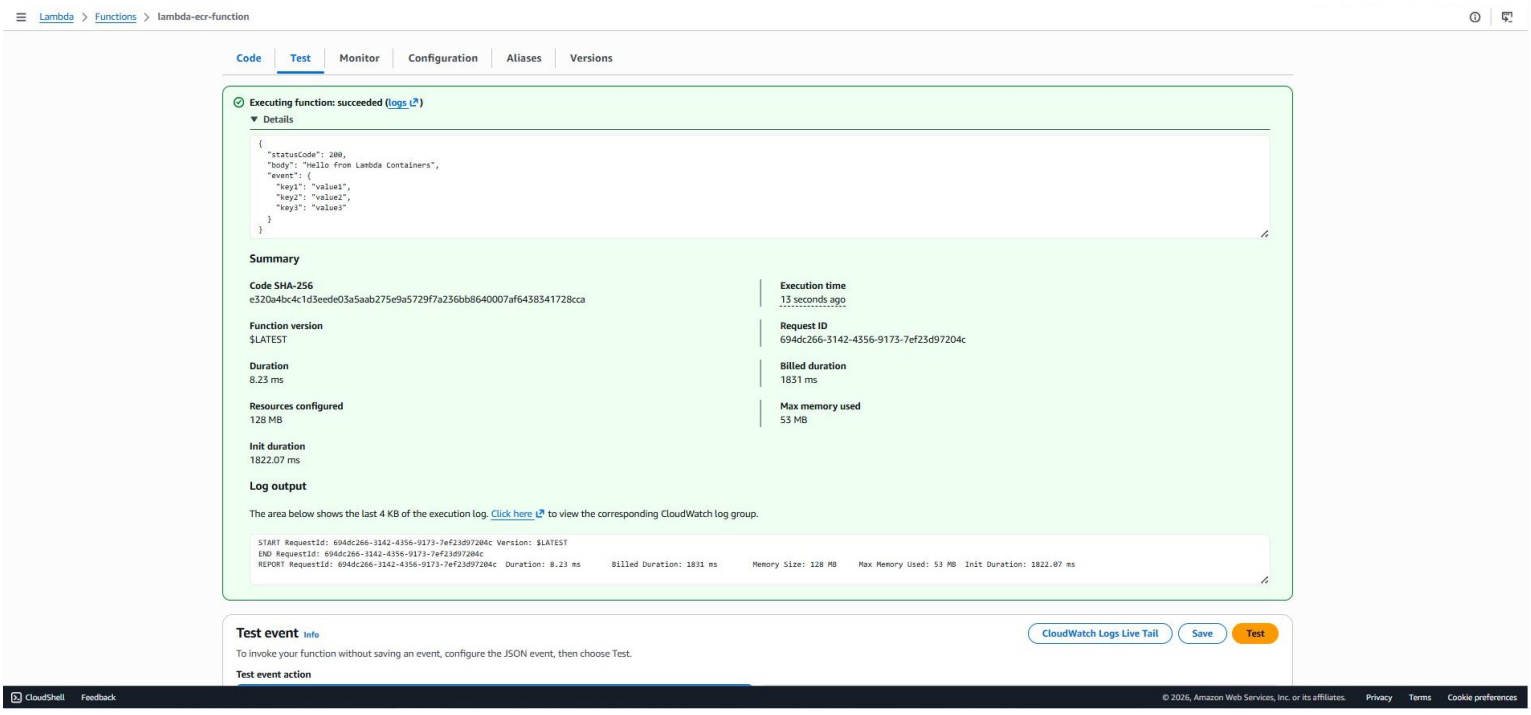


<Insert screenshot for a(3) here>



<Insert screenshot for a(4) here>





Answer the following questions

Q1 How long does a container stay in the running state if it is not manually halted?

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?

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?

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 if the same Docker image is pushed to Docker Hub multiple times with different tags?

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 | b) The layers in the first image (if unchanged) will be reused in subsequent pushes

Q5 Which of the following will run a Docker container in interactive mode?

a) -v

b) -it

c) -b

d) -u

Enter your answer here	b) -it

Q6 How should data persistence be handled in a container environment configured for autoscaling?

In an autoscaling container environment (e.g., Kubernetes HPA, Docker Swarm, or AWS ECS with scaling policies), containers are **ephemeral**—they are frequently created, destroyed, and rescheduled across different nodes. **Persisting data inside the container's filesystem is not an option**, as it will be lost upon termination.
Here's how to handle data persistence properly in such environments:
1. The Golden Rule: Externalize State
Treat containers as **stateless** wherever possible. If your application must persist data, that data should live **outside** the container lifecycle.

2. Use External Managed Services (Best Practice)
For databases, caches, and message queues, **avoid running them inside autoscaled containers**. Instead, use cloud-managed services:

- **Relational/NoSQL DBs**: AWS RDS, Aurora, GCP Cloud SQL, Azure SQL, or DynamoDB/MongoDB Atlas.
- **Caches**: ElastiCache (Redis/Memcached), Cloud Memorystore.
- **Object Storage**: AWS S3, GCS, Azure Blob Storage (ideal for files, logs, assets, and static content).

These services are inherently persistent, highly available, and scale independently of your container fleet.

3. Use Persistent Volumes (for stateful workloads)
If you must run stateful containers (e.g., a Kafka broker, Elasticsearch node, or a custom application that writes to disk), use **Persistent Volumes** (PVs) with dynamic provisioning:

- **In Kubernetes**: Use PersistentVolumeClaims (PVCs) backed by **CSI (Container Storage Interface)** drivers that connect to underlying block or file storage.
- **Recommended storage types**:
  - **Block storage** (AWS EBS, GCP Persistent Disk, Azure Managed Disks): Good for low-latency, single-writer workloads. However, note that these are zone-bound—when a pod moves to a different availability zone, it cannot attach to the same disk unless you use **regional persistent disks** or snapshot/restore.
  - **Shared file storage** (AWS EFS, GCP Filestore, Azure Files, NFS): Supports **ReadWriteMany** access, allowing multiple replicas to read/write simultaneously. This works well with autoscaling because pods can be rescheduled on any node and still access the same shared data.
- **Dynamic Provisioning**: Ensure your storage class includes allowVolumeExpansion:

true and proper reclaim Policy to handle scaling events.

4. Handle Node Affinity / Topology (Critical for Block Storage)
If using zonal block storage (like EBS), you **must** configure your workload to schedule on the same zone as the volume. Use:

- **StatefulSets** with volume claims (each pod gets its own PV that follows it).
- **Pod Topology Spread Constraints** or **nodeAffinity** rules to pin the pod to the zone where the volume resides.
- Alternatively, use **volume snapshot controllers** to restore data to a new volume if a pod is rescheduled to a different zone.

5. For Autoscaling Stateless Apps with Ephemeral Data
If the data is **temporary** (e.g., cache, session state, or processing temp files) and can be recreated, you can use:

- **EmptyDir** volumes (local node storage) – but be aware that data disappears when the pod is evicted or moved.
- **HostPath** (mounts a node directory) – **not recommended** for autoscaling, as it ties the pod to a specific node and breaks across scaling events.

6. Backup and Lifecycle Policies
Since autoscaling can be aggressive, implement:

- **Automated snapshots** of your PVs (e.g., EBS snapshots, Velero for Kubernetes).
- **Retention policies** to avoid orphaned volumes and cost creep.
- **Storage class parameters** for IOPS/throughput to match the scaling demands (e.g., gp3 with provisioned IOPS).

Summary Decision Matrix

Workload Type	Recommended Persistence Approach
Databases / Caches	Use **external managed services** (RDS, DynamoDB, ElastiCa
Shared files / logs	Use **shared file storage** (EFS, NFS, GCS Filestore)
Stateful apps (Kafka, Elasticsearch)	Use **StatefulSets + PersistentVolumeClaims** with dynamic p aware scheduling
Temporary / scratch space	Use emptyDir (memory or disk-backed)
Static assets / uploads	Use **object storage** (S3, GCS)

Key takeaway: In an autoscaled environment, **persistence must be decoupled from compute**. Design your storage layer to be independently scalable, highly available, and resilient to container restarts or node failures.

Q7 Why is this statement false? "Docker is the only popular choice for microservices deployment".

This statement is **false**. While Docker was once the undisputed leader in containerization, today it is **neither the only choice nor even the default or best option** in many microservices deployment scenarios.

The core idea of microservices—breaking a large application into independent, individually deployable services—does not strictly require Docker. Docker's perceived irreplaceability is being challenged on multiple fronts:

Container Runtime Layer: Docker is NOT the Only Engine

Many people equate "containers" with "Docker," but container technology is built on OCI (Open Container Initiative) standards. Docker itself runs on lower-level runtimes like containerd and runc.

- **Kubernetes officially removed Docker as a runtime**: A landmark event—Kubernetes deprecated Docker as a container runtime starting with version 1.20. This clearly proves that Docker is not indispensable for the most important container orchestration platform.
- **Lighter-weight runtimes**: containerd (donated by Docker to the CNCF) and runc are now more popular lightweight runtime choices—they are more focused and have lower resource overhead. There is also CRI-O, specifically built for Kubernetes.
- **Rise of "Dockerless" architectures**: Using containerd directly with tools like nerdctl (which is Docker-compatible) eliminates the overhead and potential single point of failure posed by the Docker daemon.

Orchestration Layer: Kubernetes is the De Facto Standard

At the orchestration level for managing large fleets of microservices, Docker Swarm is just *one* option—and not the mainstream one.

- **Kubernetes (K8s) dominates**: Kubernetes has become the de facto standard for container orchestration, with an adoption rate of over **82%** in production environments. Its ecosystem is far superior to Docker Swarm.
- **Other mature orchestration tools**:
  - **Docker Swarm**: Docker's native orchestration tool. Good for small-scale, simple setups, but its market share (**~24%**) and scalability fall far short of Kubernetes.
  - **HashiCorp Nomad**: A highly flexible tool that orchestrates not only containers but also heterogeneous workloads like virtual machines (VMs) and standalone applications.
  - **Amazon ECS**: A cloud-native orchestration service on AWS, deeply integrated with the AWS ecosystem.

Deployment Methodology Layer: Microservices DON'T Require Containers

This is the most overlooked point: **The microservices architecture itself does not mandate containerization.**

- **Deploy as native processes**: You can run each microservice as a direct system process (e.g., a Java, Python, or Go application) directly on the operating system.
- **Deploy on Virtual Machines (VMs)**: Package each microservice as a VM image (e.g., using Packer) and deploy it on cloud VMs like AWS EC2. This provides decent isolation, though with higher resource overhead than containers.

Higher-Level Abstractions: Serverless and PaaS Platforms

Some platforms fully abstract away the underlying infrastructure (whether containers or VMs), allowing developers to just deploy code.

- **Serverless (Function-as-a-Service)**: Like **AWS Lambda**—you just upload your code, and the platform automatically handles execution and scaling.

- **Serverless container platforms**: Like **AWS Fargate** or **Azure Container Apps**—you don't manage servers or clusters; you just define the container image and run it.
- **Application Hosting Platforms (PaaS)**: Like **Heroku** or **Google App Engine**—they hide most of the underlying details, letting developers focus purely on application logic.

Specific Alternatives at the Container Engine Level

Even if you decide to use containers, Docker faces strong competition.

- **Podman**: Led by Red Hat. Its core advantages are **daemonless** architecture (no background daemon) and **native rootless mode**, offering better security and lower resource consumption than Docker.
- **Other tools**: Tools for local development like **ServBay**, and container management platforms like **Rancher** and **Portainer**, offer alternatives to the standard Docker ecosystem.

In conclusion, Docker was once synonymous with containerization, but today's microservices deployment landscape is highly diverse and feature-rich. Calling Docker the *only* popular choice is simply out of touch with current technological reality.