8byte Intern Assignment – DevOps Deployment Project

Public Application Access

The deployed Node.js application is accessible at:
http://98.92.148.150:3000/

Git repo: vommidapuchinni/8bytes

Prerequisites

Before running this project locally or provisioning infrastructure, ensure you have:

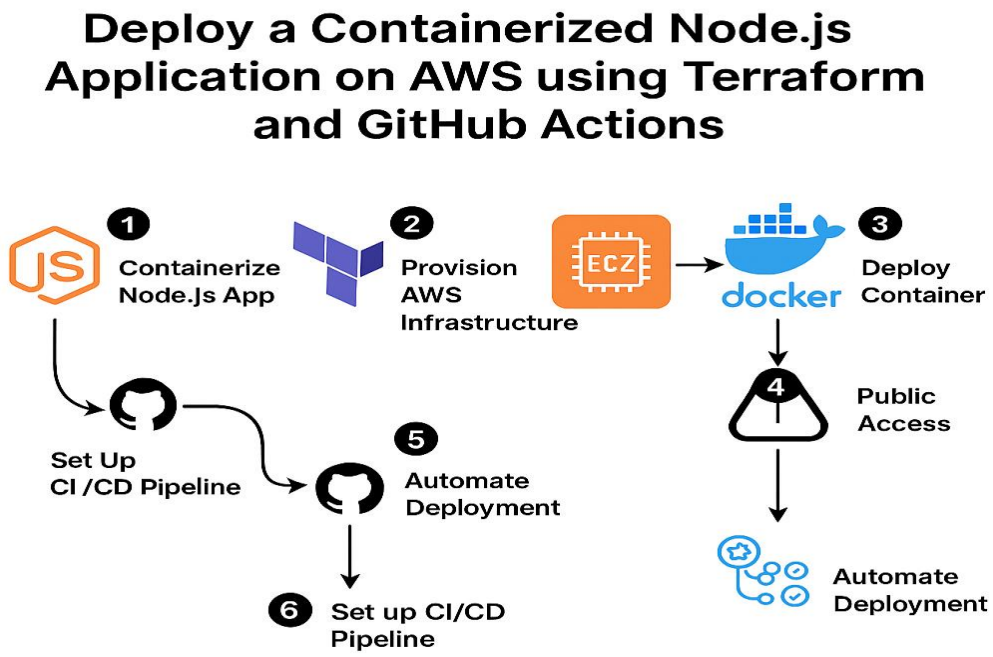
- Node.js & npm
- Docker & Docker Desktop
- Terraform
- AWS Account with IAM user credentials (Access Key ID & Secret Access Key)

Objectives

This project demonstrates hands-on experience in:

1. Infrastructure as Code using Terraform
2. AWS Networking (VPC architecture design)
3. Containerization using Docker
4. Continuous Integration using GitHub Actions
5. Application deployment on EC2 with public access
6. Infrastructure lifecycle and cost management

Architecture Overview



Component	Purpose
VPC	Custom isolated cloud network
Public Subnet	Hosts the EC2 instance
Internet Gateway	Enables internet connectivity
Route Table	Routes outbound internet traffic
Security Group	Allows SSH (22) & App (3000)
EC2 Instance	Hosts the Dockerized application
Docker	Runs the Node.js application
GitHub Actions	Automates Docker build (CI/CD)

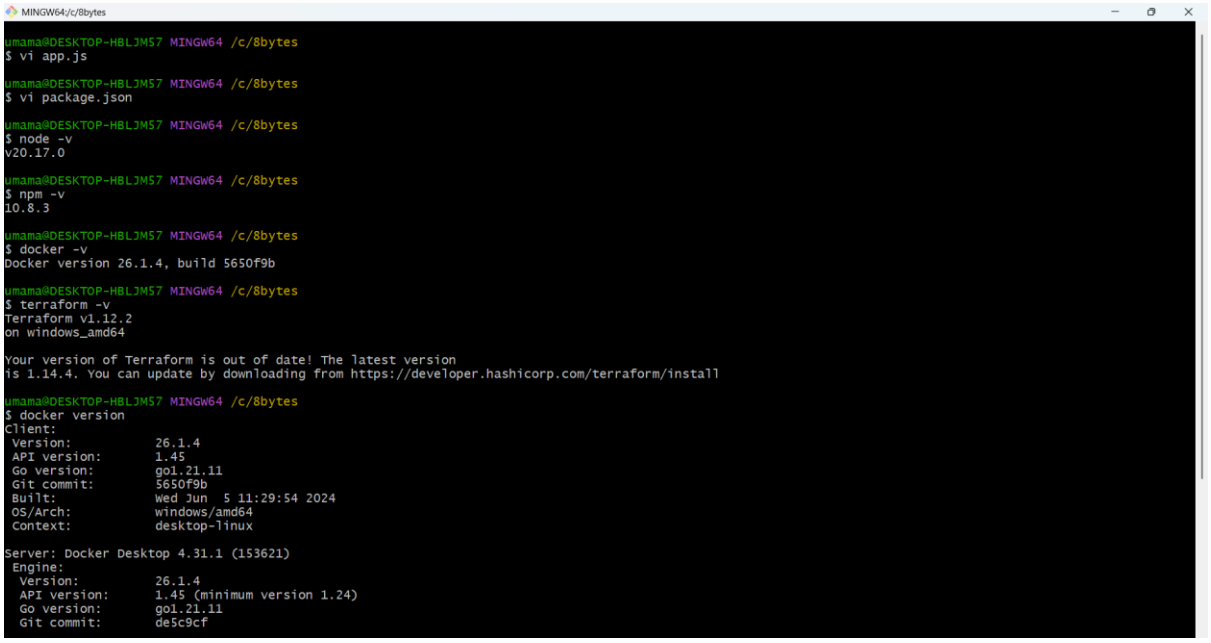
Run Application Locally

Before deploying to AWS, you can test and run the Node.js application on your local machine to ensure it works as expected.

Install dependencies and run the application:

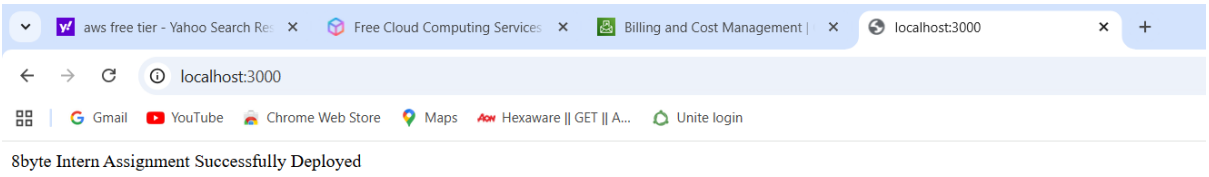
npm install && node app.js

Installations



npm install && node app.js

Open in browser: http://localhost:3000



Expected Output: 8byte Intern Assignment Successfully Deployed

Dockerizing the Application

We will now containerize the Node.js app using Docker to prepare it for deployment on EC2.

FROM node:18

WORKDIR /app

COPY . .

RUN npm install

EXPOSE 3000

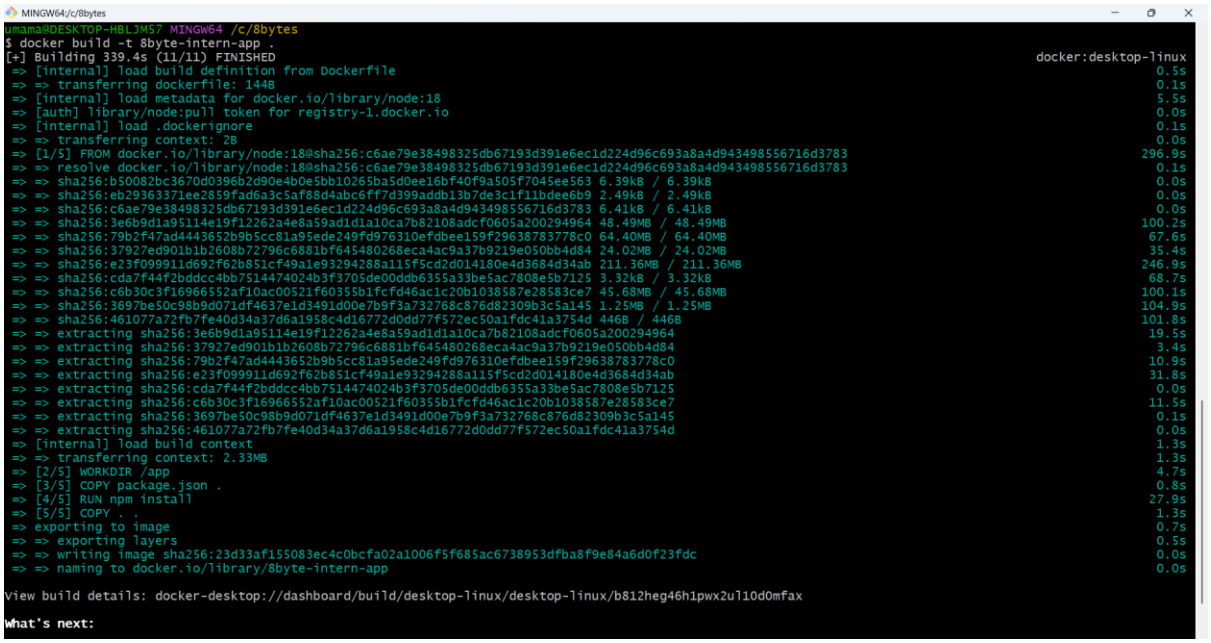
CMD ["node", "app.js"]

Build & Run

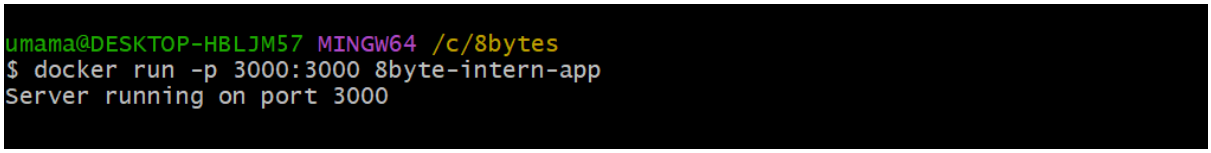
docker build -t 8byte-intern-app . # build docker file

docker run -p 3000:3000 8byte-intern-app

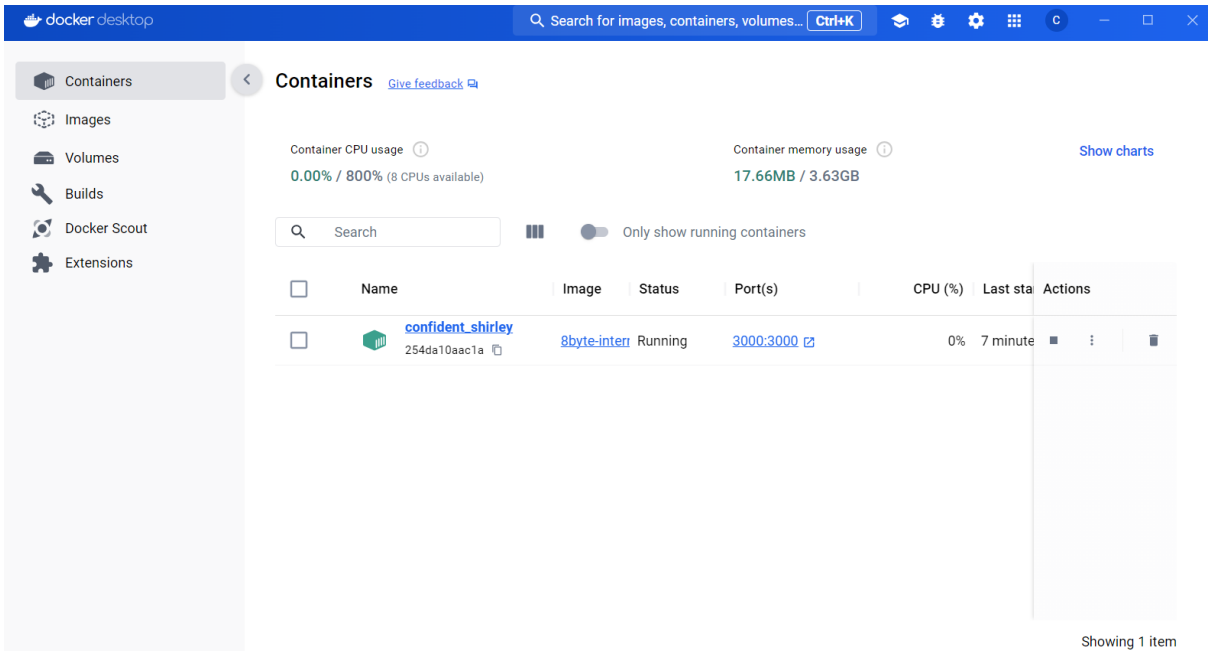
Docker build locally



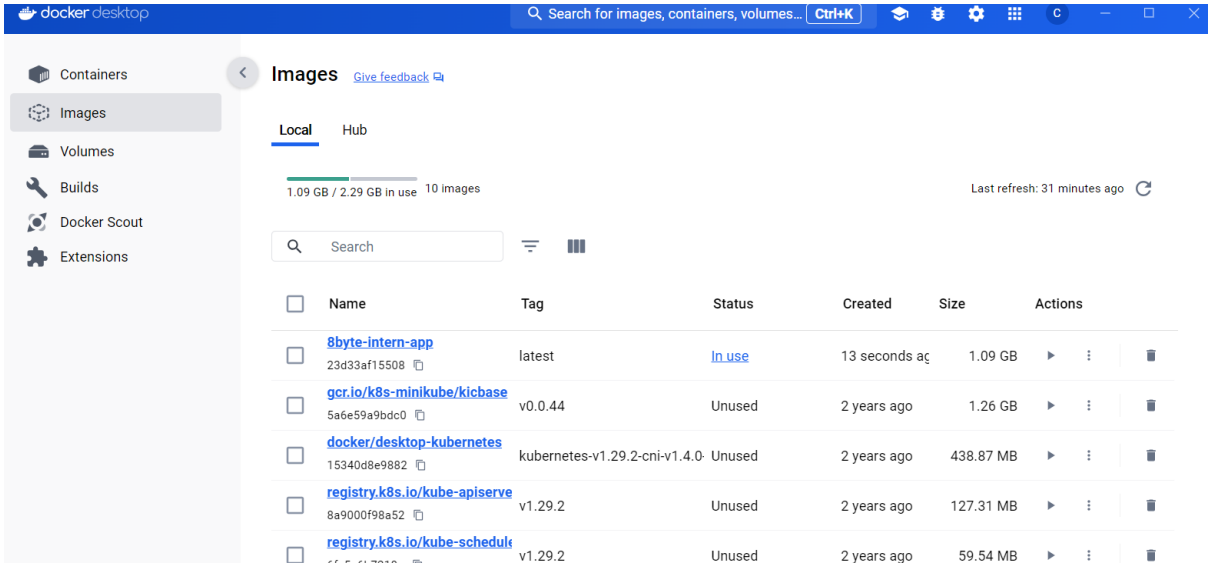
Docker run locally



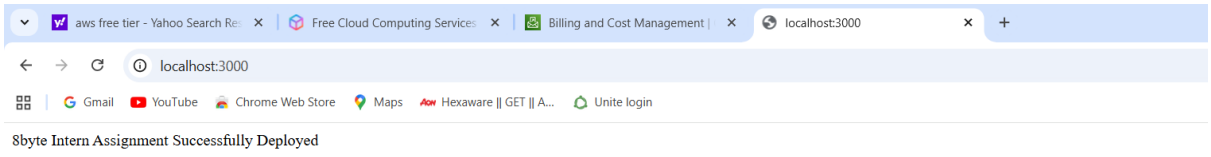
Docker container



Docker image



Access APP



Visit: http://localhost:3000

Infrastructure Provisioning with Terraform

Terraform allows us to define and provision AWS infrastructure in code, making deployments repeatable and manageable.

Directory Structure terraform/ provider.tf, variables.tf, terraform.tfvars, main.tf, outputs.tf

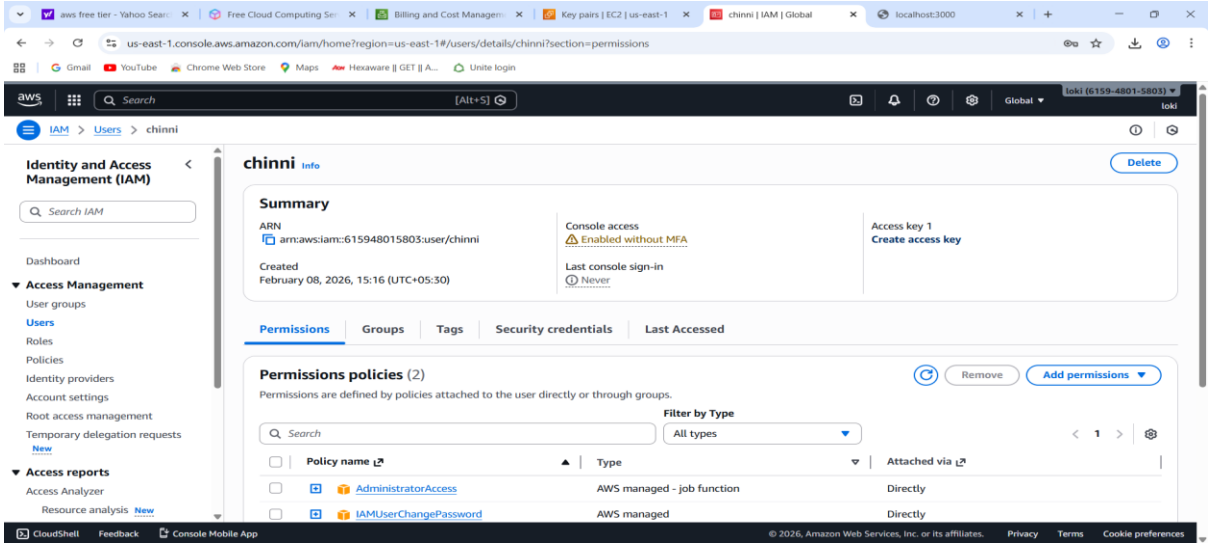
Purpose of Each File

File	Purpose
provider.tf	Configures AWS provider (region: us-east-1)
variables.tf	Defines reusable input variables
Internet Gateway	Stores environment-specific values
main.tf Contains	infrastructure resources
outputs.tf	Displays useful outputs (EC2 Public IP)

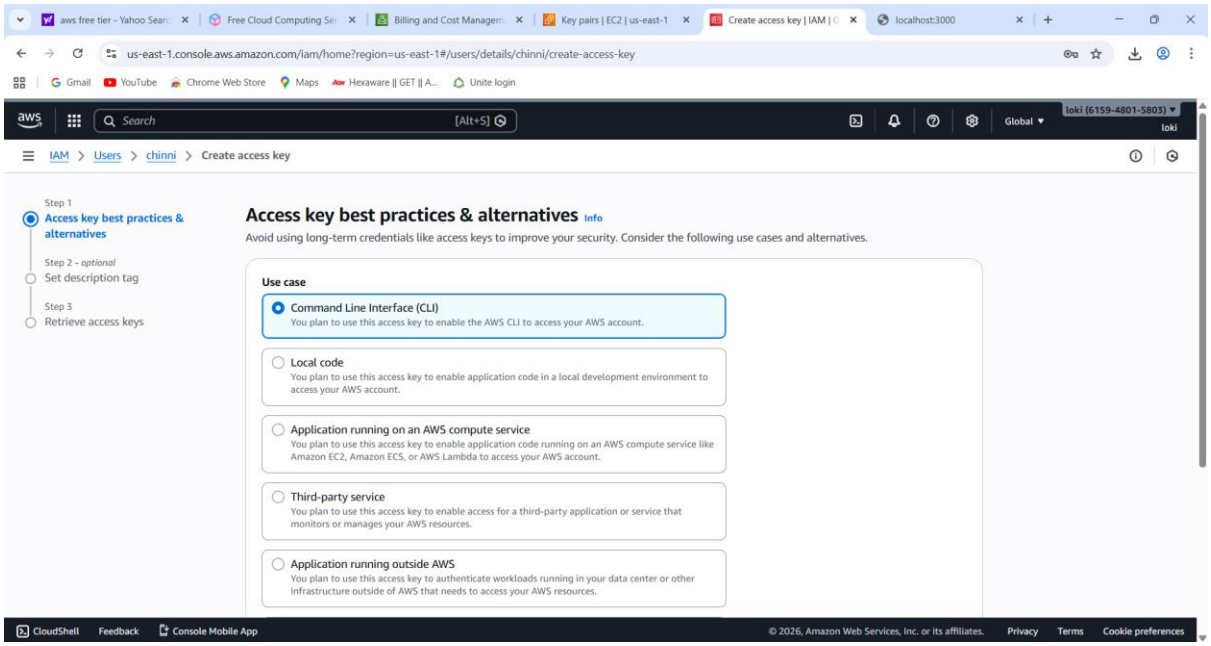
To create resources on AWS locally using Terraform, you must provide AWS credentials. These credentials are obtained by creating an IAM user with programmatic access and generating an Access Key ID and Secret Access Key.

Terraform uses these keys to authenticate and provision AWS resources.

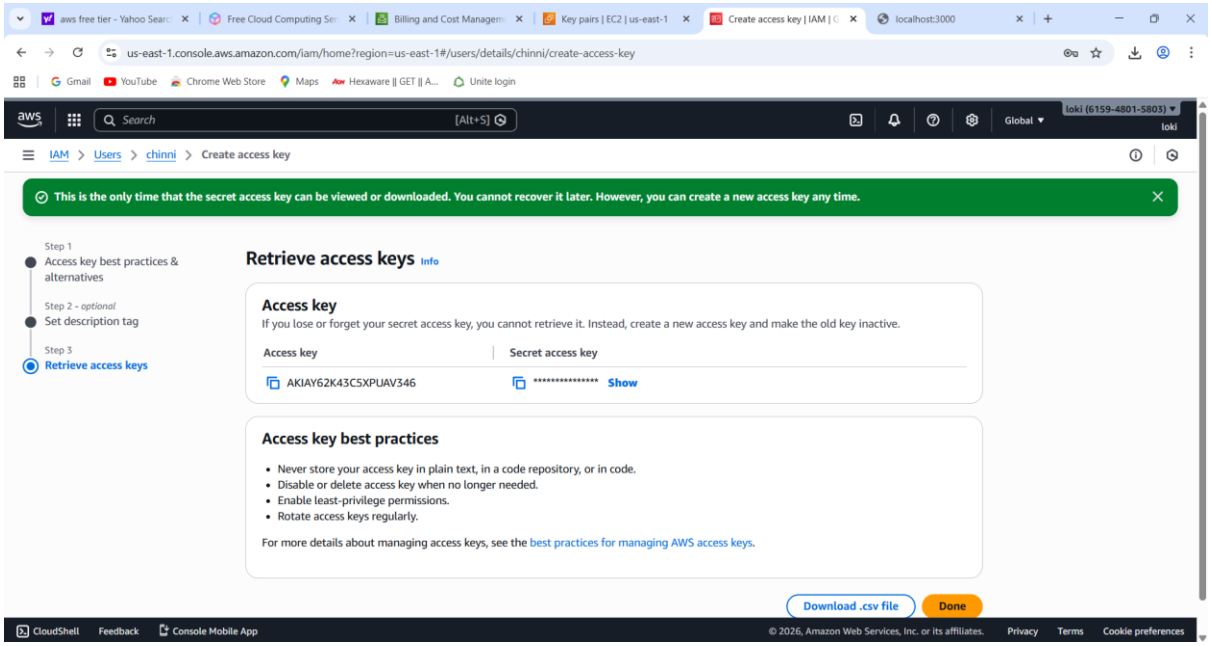
IAM User



Secret keys creation for credentials



Access and secret keys



Terraform Commands

terraform init

```
umama@DESKTOP-HBLJM57 MINGW64 /c/8bytes/terraform
$ terraform init
Initializing the backend...
Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v6.31.0...
- Installed hashicorp/aws v6.31.0 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

terraform plan

```
MINGW64:/c/8bytes/terraform
umama@DESKTOP-HBLJM57 MINGW64 /c/8bytes/terraform
$ terraform plan
data.aws_ami.ubuntu: Reading...
data.aws_ami.ubuntu: Read complete after 2s [id=ami-0030e4319cbf4dbf2]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following
symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.app will be created
  + resource "aws_instance" "app" {
      + ami                                  = "ami-0030e4319cbf4dbf2"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = true
      + availability_zone                    = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + enable_primary_ipv6                  = (known after apply)
      + force_destroy                        = false
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + host_resource_group_arn              = (known after apply)
      + iam_instance_profile                 = (known after apply)
      + id                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_lifecycle                   = (known after apply)
      + instance_state                       = (known after apply)
      + instance_type                        = "t2.micro"
      + ipv6_address_count                   = (known after apply)
      + ipv6_addresses                       = (known after apply)
      + key_name                             = "8byte-key"
      + monitoring                           = (known after apply)
      + outpost_arn                          = (known after apply)
      + password_data                        = (known after apply)
      + placement_group                      = (known after apply)
```

terraform apply

```
MINGW64:/c/8bytes/terraform
umama@DESKTOP-HBLJM57 MINGW64 /c/8bytes/terraform
$ terraform apply
data.aws_ami.ubuntu: Reading...
aws_vpc.main: Refreshing state... [id=vpc-058c06fbf476e3fef]
data.aws_ami.ubuntu: Read complete after 2s [id=ami-0030e4319cbf4dbf2]
aws_internet_gateway.gw: Refreshing state... [id=igw-06c0f1267650d5a3d]
aws_subnet.public: Refreshing state... [id=subnet-046e488860d133780]
aws_security_group.sg: Refreshing state... [id=sg-0b8ba60e90967b419]
aws_route_table.rt: Refreshing state... [id=rtb-053c8a5519a471e67]
aws_route_table_association.a: Refreshing state... [id=rtbassoc-0d0ec0aed0d725e91]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following
symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.app will be created
  + resource "aws_instance" "app" {
      + ami                                  = "ami-0030e4319cbf4dbf2"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = true
      + availability_zone                    = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + enable_primary_ipv6                  = (known after apply)
      + force_destroy                        = false
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + host_resource_group_arn              = (known after apply)
      + iam_instance_profile                 = (known after apply)
      + id                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_lifecycle                   = (known after apply)
      + instance_state                       = (known after apply)
      + instance_type                        = "t3.micro"
      + ipv6_address_count                   = (known after apply)
      + ipv6_addresses                       = (known after apply)
```

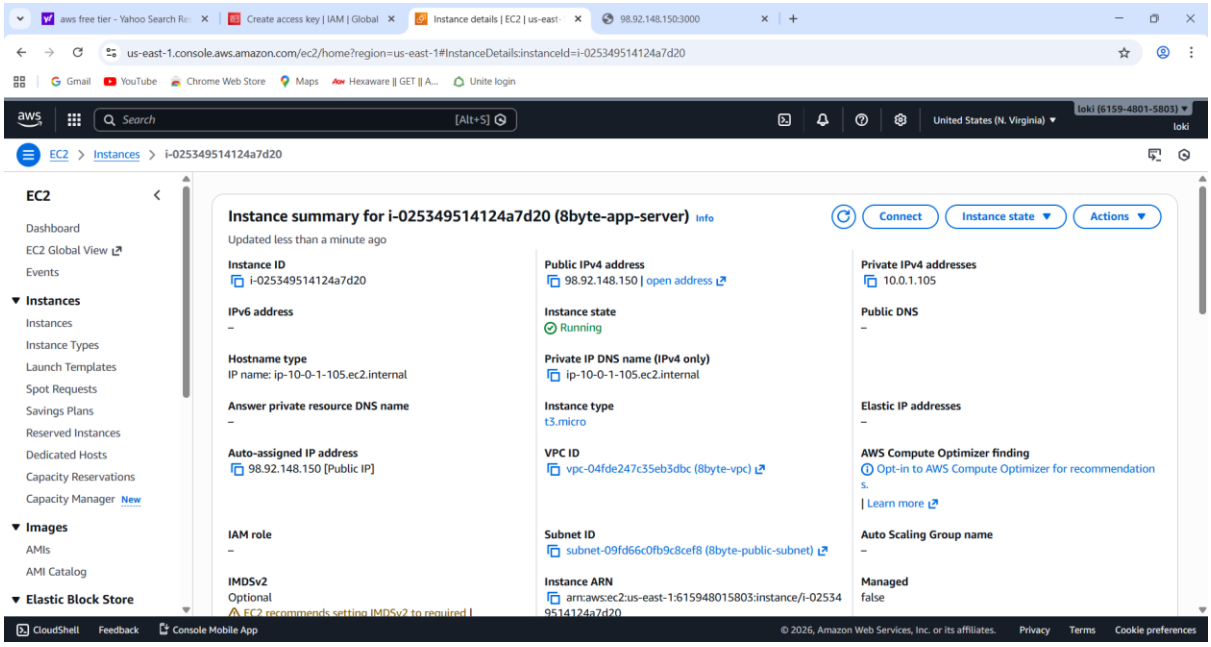
outputs

```
Outputs:

public_ip = "98.92.167.154"
ssh_command = "ssh -i 8byte-key.pem ubuntu@98.92.167.154"

umama@DESKTOP-HBLJM57 MINGW64 /c/8bytes/terraform
$
```

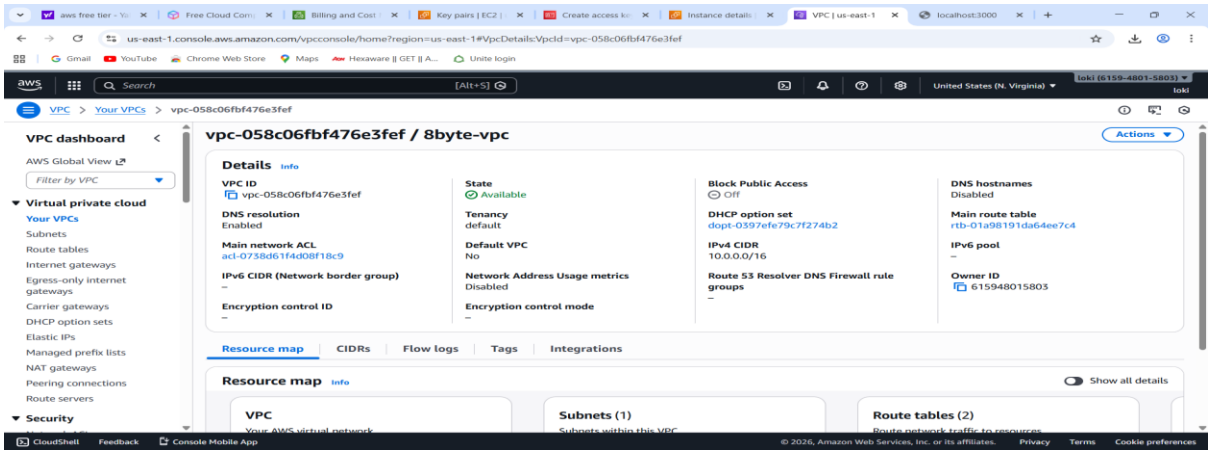
Resources Created: VPC, Public Subnet, Internet Gateway, Route Table & Association, Security Group (ports 22 and 3000), EC2 Instance (Ubuntu with Docker installed via user_data)

Resources:

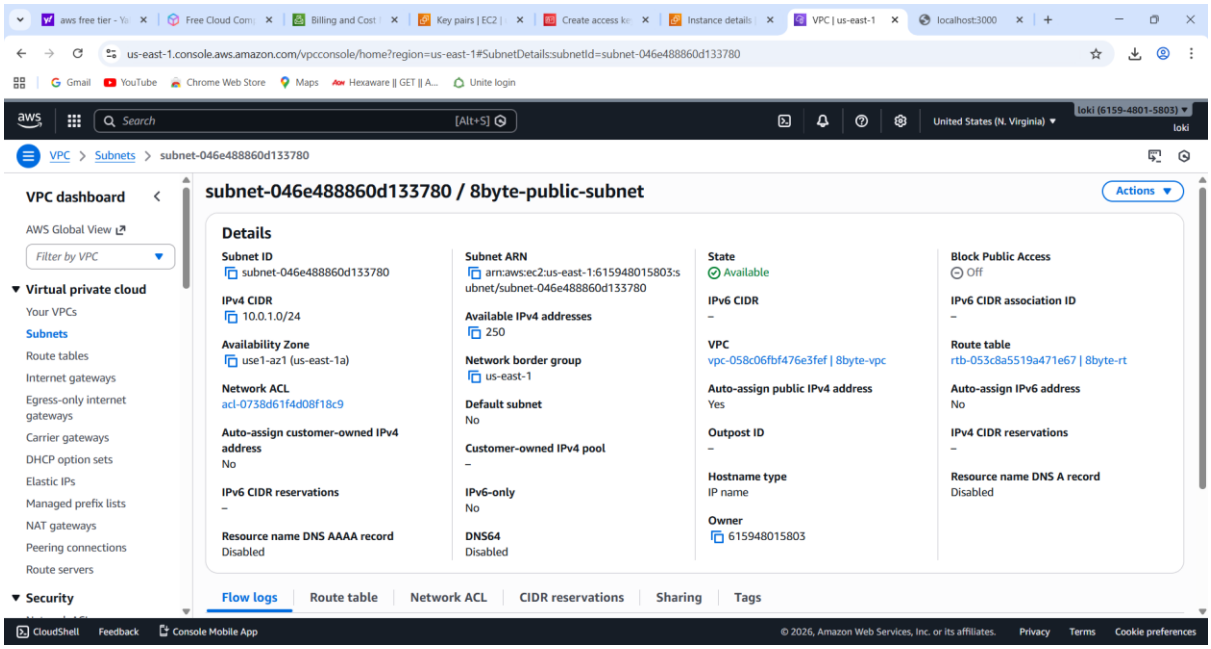
EC2 running



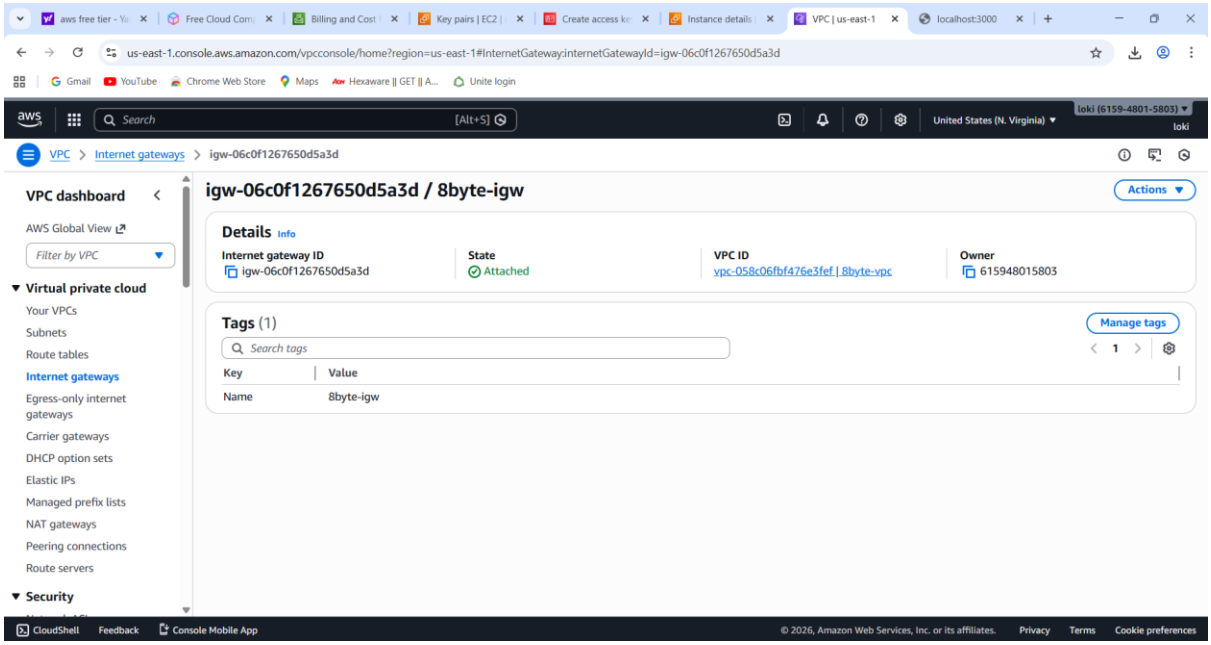
VPC



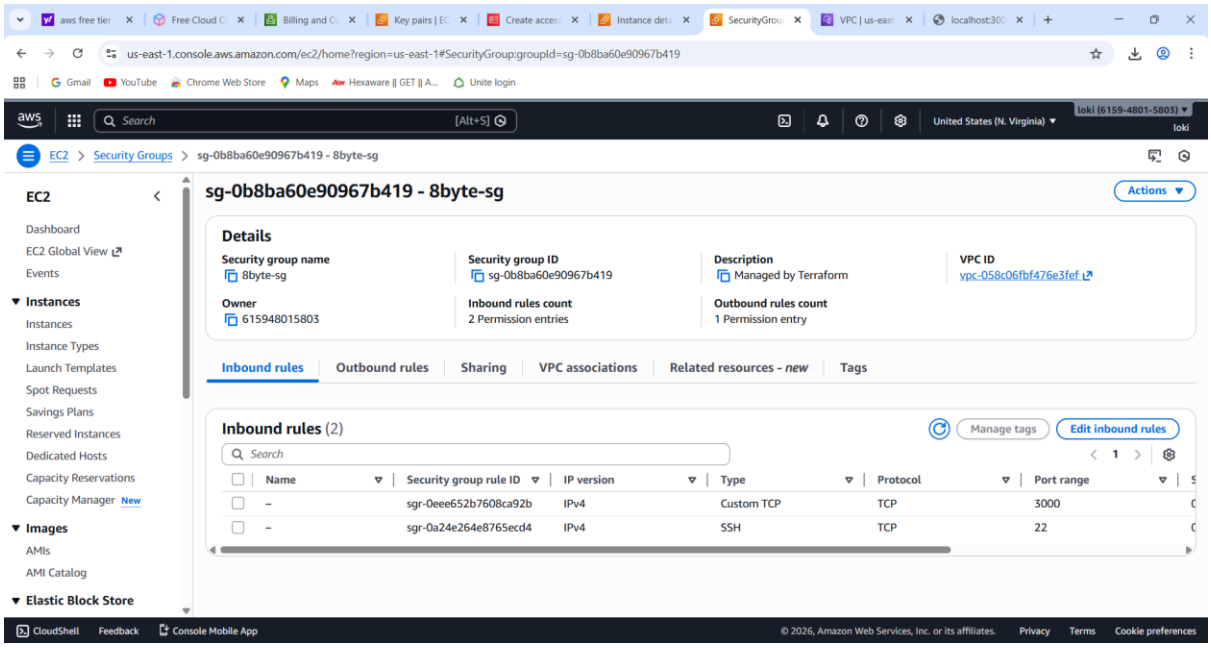
Subnet



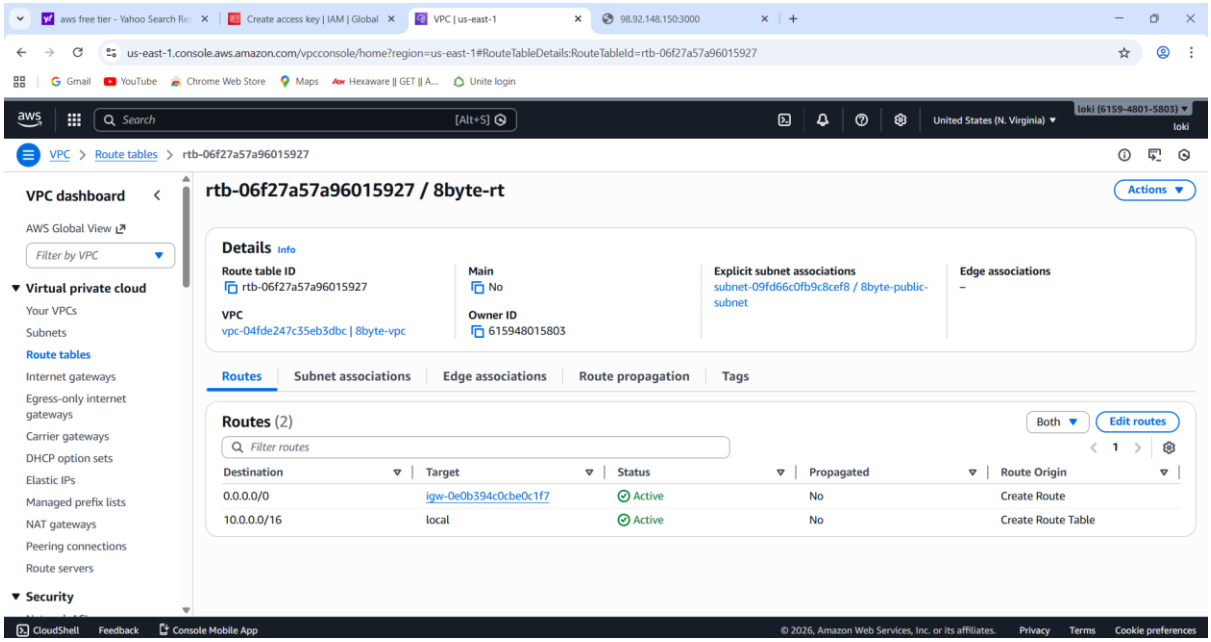
Internet gateway



Security groups

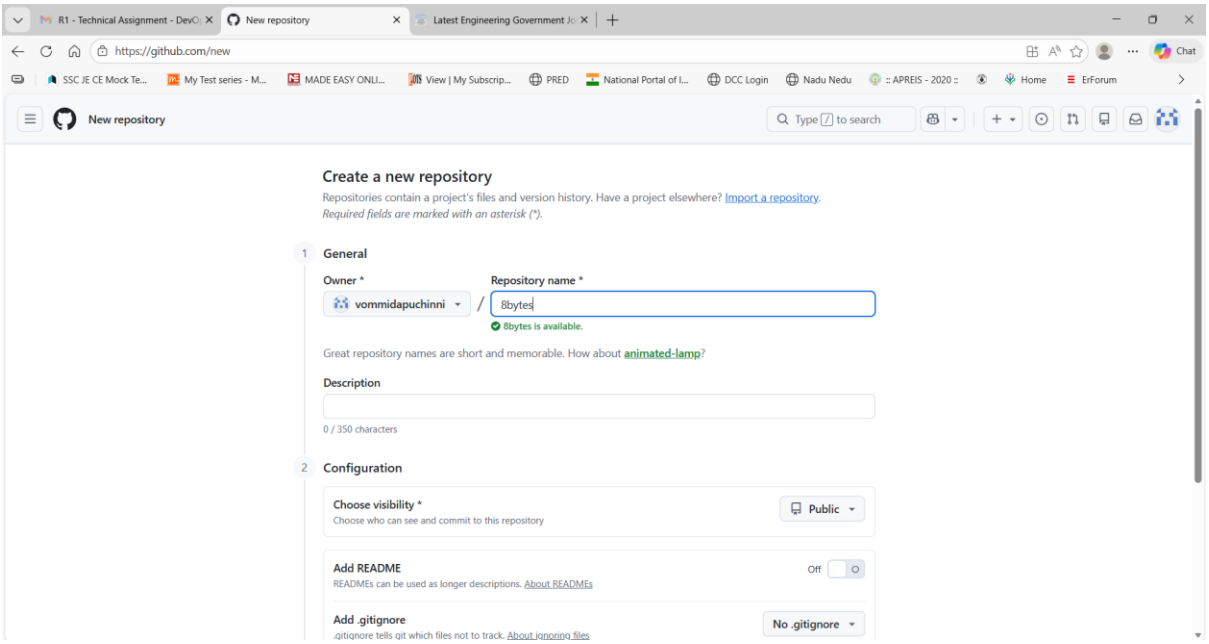


Route table and subnet association

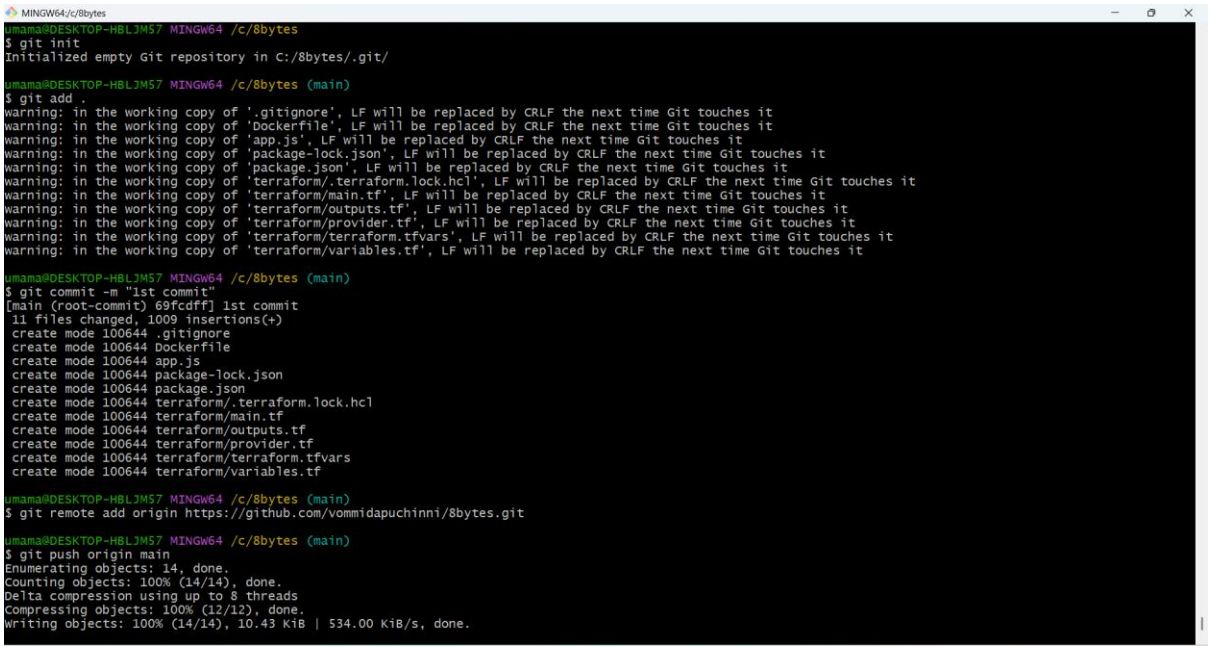


Push the code to github

create new git repository



Push code to github



Deploy Application on EC2

Once the infrastructure is ready, we deploy the Dockerized Node.js application to the EC2 instance and verify it works in the cloud environment.

ssh -i key.pem ubuntu@EC2-PUBLIC-IP

SSH into EC2

```
ubuntu@ip-10-0-1-243: ~

2 additional security updates can be applied with ESM Apps.
Learn more about enabling ESM Apps service at https://ubuntu.com/esm

New release '24.04.3 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


Last login: Sun Feb  8 10:25:07 2026 from 157.50.95.82
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-10-0-1-243:~$ ls
ubuntu@ip-10-0-1-243:~$ docker version
Client:
 Version:           28.2.2
 API version:       1.50
 Go version:        go1.23.1
 Git commit:        28.2.2-0ubuntu1~22.04.1
 Built:             Wed Sep 10 14:50:16 2025
 OS/Arch:           linux/amd64
 Context:           default

Server:
 Engine:
  Version:          28.2.2
  API version:      1.50 (minimum version 1.24)
  Go version:       go1.23.1
  Git commit:       28.2.2-0ubuntu1~22.04.1
  Built:            Wed Sep 10 14:50:16 2025
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.7.28
  GitCommit:
 runc:
  Version:          1.3.3-0ubuntu1~22.04.3
  GitCommit:
 docker-init:
  Version:          0.19.0
  GitCommit:
```

check docker version and status by using the commands
docker version && systemctl status docker

Docker status

```
ubuntu@ip-10-0-1-243:~$ systemctl status docker
● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
     Active: active (running) since Sun 2026-02-08 10:01:39 UTC; 31min ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 2023 (dockerd)
      Tasks: 9
     Memory: 31.9M
        CPU: 536ms
     CGroup: /system.slice/docker.service
             └─2023 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock

Feb 08 10:01:38 ip-10-0-1-243 dockerd[2023]: time="2026-02-08T10:01:38.954613193Z" level=info msg="Loading containers: start."
Feb 08 10:01:39 ip-10-0-1-243 dockerd[2023]: time="2026-02-08T10:01:39.254251548Z" level=info msg="Loading containers: done."
Feb 08 10:01:39 ip-10-0-1-243 dockerd[2023]: time="2026-02-08T10:01:39.277502695Z" level=info msg="Docker daemon" commit="28.2.2-0ubuntu1~22.04.1" containe>
Feb 08 10:01:39 ip-10-0-1-243 dockerd[2023]: time="2026-02-08T10:01:39.277618206Z" level=info msg="Initializing buildkit"
Feb 08 10:01:39 ip-10-0-1-243 dockerd[2023]: time="2026-02-08T10:01:39.287128637Z" level=warning msg="CDI setup error /var/run/cdi: failed to monitor for c>
Feb 08 10:01:39 ip-10-0-1-243 dockerd[2023]: time="2026-02-08T10:01:39.287156981Z" level=warning msg="CDI setup error /etc/cdi: failed to monitor for chang>
Feb 08 10:01:39 ip-10-0-1-243 dockerd[2023]: time="2026-02-08T10:01:39.320655535Z" level=info msg="Completed buildkit initialization"
Feb 08 10:01:39 ip-10-0-1-243 dockerd[2023]: time="2026-02-08T10:01:39.328472831Z" level=info msg="Daemon has completed initialization"
Feb 08 10:01:39 ip-10-0-1-243 dockerd[2023]: time="2026-02-08T10:01:39.328723178Z" level=info msg="API listen on /run/docker.sock"
Feb 08 10:01:39 ip-10-0-1-243 systemd[1]: Started Docker Application Container Engine.

ubuntu@ip-10-0-1-243:~$
```

clone repo to EC2 from github

```
ubuntu@ip-10-0-1-243:~$ git clone https://github.com/vommidapuchinni/8bytes.git
Cloning into '8bytes'...
remote: Enumerating objects: 14, done.
remote: Counting objects: 100% (14/14), done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 14 (delta 0), reused 14 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (14/14), 10.43 KiB | 5.22 MiB/s, done.
ubuntu@ip-10-0-1-243:~$ ls
8bytes
ubuntu@ip-10-0-1-243:~$ cd 8bytes/
ubuntu@ip-10-0-1-243:~/8bytes$ ls
Dockerfile  app.js  package-lock.json  package.json  terraform
```

docker build -t app .

Docker build on EC2

```
ubuntu@ip-10-0-1-243:~/8bytes$ docker build -t 8byte-app .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  112.6kB
Step 1/7 : FROM node:18
18: Pulling from library/node
3e6b9d1a9511: Pulling fs layer
37927ed901b1: Pulling fs layer
79b2f47ad444: Pulling fs layer
e23f099911d6: Pulling fs layer
cda7f44f2bdd: Pulling fs layer
c6b30c3f1696: Pulling fs layer
3697be50c98b: Pulling fs layer
461077a72fb7: Pulling fs layer
cda7f44f2bdd: Waiting
c6b30c3f1696: Waiting
3697be50c98b: Waiting
461077a72fb7: Waiting
e23f099911d6: Waiting
37927ed901b1: Verifying Checksum
37927ed901b1: Download complete
3e6b9d1a9511: Verifying Checksum
3e6b9d1a9511: Download complete
79b2f47ad444: Verifying Checksum
79b2f47ad444: Download complete
cda7f44f2bdd: Verifying Checksum
cda7f44f2bdd: Download complete
3697be50c98b: Verifying Checksum
3697be50c98b: Download complete
461077a72fb7: Verifying Checksum
461077a72fb7: Download complete
c6b30c3f1696: Verifying Checksum
c6b30c3f1696: Download complete
e23f099911d6: Verifying Checksum
e23f099911d6: Download complete
3e6b9d1a9511: Pull complete
37927ed901b1: Pull complete
79b2f47ad444: Pull complete
e23f099911d6: Pull complete
```

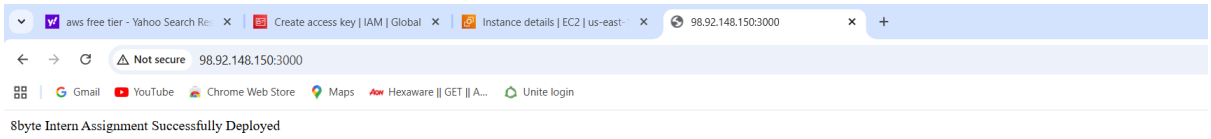
docker run -d -p 3000:3000 app

Docker run on EC2

```
ubuntu@ip-10-0-1-243:~/8bytes$ docker run -d -p 3000:3000 --name app 8byte-app
99f3b6a6855a03f9089abe7a04bda8febf2afdeed554290f72a090210b16646aa
ubuntu@ip-10-0-1-243:~/8bytes$ docker ps
CONTAINER ID   IMAGE       COMMAND                  CREATED         STATUS         PORTS                                                NAMES
99f3b6a6855a   8byte-app   "docker-entrypoint.s…"   37 seconds ago   Up 36 seconds   0.0.0.0:3000->3000/tcp, [::]:3000->3000/tcp   app
ubuntu@ip-10-0-1-243:~/8bytes$ docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
8byte-app    latest    d5e1bdbaf9c0   43 seconds ago   1.1GB
node         18        b50082bc3670   10 months ago    1.09GB
ubuntu@ip-10-0-1-243:~/8bytes$
```

Access in browser: http://:3000

Access app



GitHub Actions CI/CD

We will automate building, testing, and optionally pushing the Docker image using GitHub Actions to implement a full CI/CD pipeline.

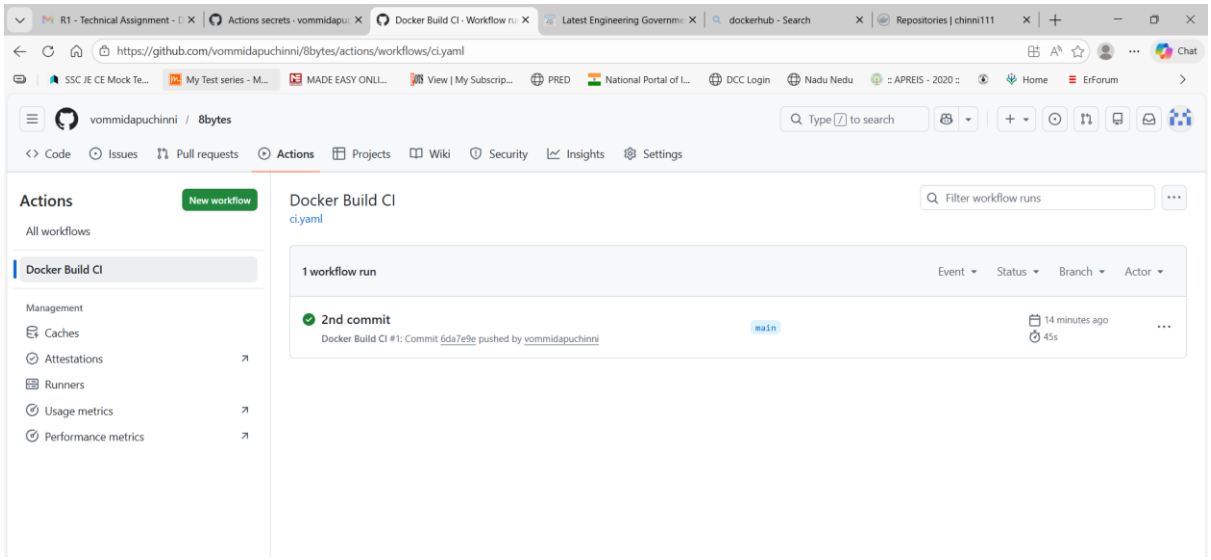
Workflow Location: .github/workflows/ci.yml

Creation of workflow files and push to github

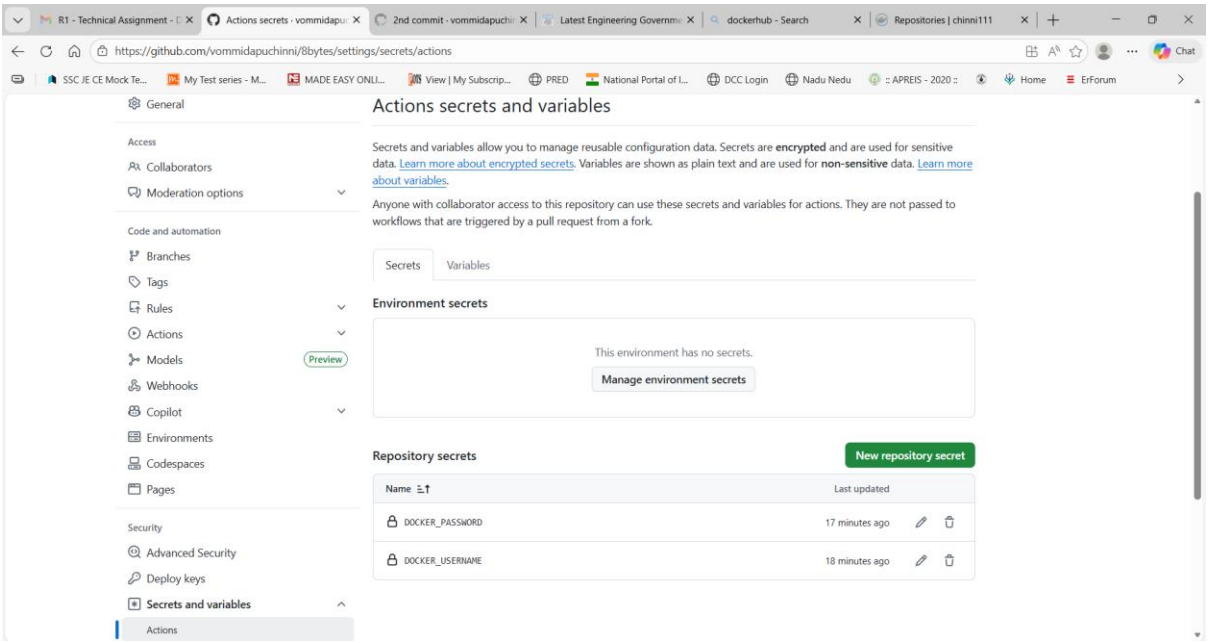
```
MINGW64:/c/8bytes/.github/workflows
bash: cd: .github: No such file or directory

umama@DESKTOP-HBLJM57 MINGW64 /c/8bytes (main)
$ mkdir .github

umama@DESKTOP-HBLJM57 MINGW64 /c/8bytes (main)
$ cd .github

umama@DESKTOP-HBLJM57 MINGW64 /c/8bytes/.github (main)
$ mkdir workflows

umama@DESKTOP-HBLJM57 MINGW64 /c/8bytes/.github (main)
$ cd workflows/

umama@DESKTOP-HBLJM57 MINGW64 /c/8bytes/.github/workflows (main)
$ ls

umama@DESKTOP-HBLJM57 MINGW64 /c/8bytes/.github/workflows (main)
$ vi ci.yaml

umama@DESKTOP-HBLJM57 MINGW64 /c/8bytes/.github/workflows (main)
$ git add .
warning: in the working copy of '.github/workflows/ci.yaml', LF will be replaced by CRLF the next time Git touches it

umama@DESKTOP-HBLJM57 MINGW64 /c/8bytes/.github/workflows (main)
$ git commit -m "2nd commit"
[main 6da7e9e] 2nd commit
 1 file changed, 30 insertions(+)
 create mode 100644 .github/workflows/ci.yaml

umama@DESKTOP-HBLJM57 MINGW64 /c/8bytes/.github/workflows (main)
$ git push origin main
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (5/5), 652 bytes | 163.00 KiB/s, done.
Total 5 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/vommidapuchinni/8bytes.git
   69fcdff..6da7e9e  main -> main

umama@DESKTOP-HBLJM57 MINGW64 /c/8bytes/.github/workflows (main)
$
```

Pipeline Behavior

Triggered on push to the main branch.



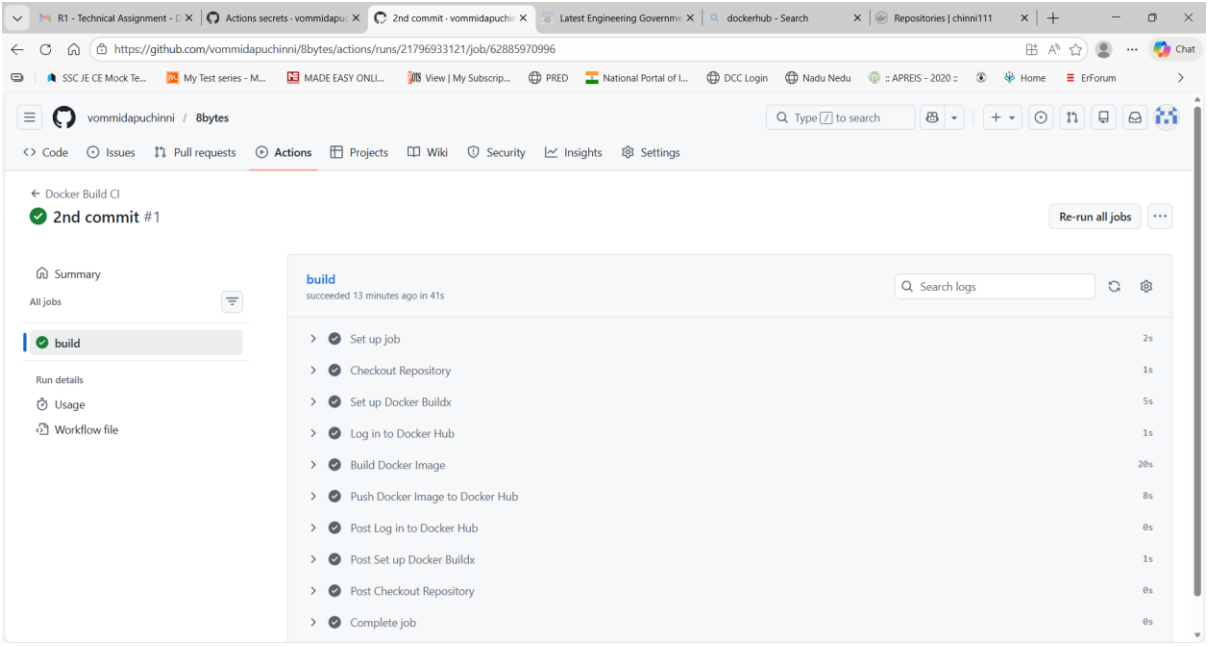
repo secrets to login to Docker Hub



Pipeline Steps:

1. Checkout repository
2. Set up Docker
3. Build Docker image
4. Verify successful build
5. Push image to Docker Hub

Workflow Successfully ran



Docker Hub

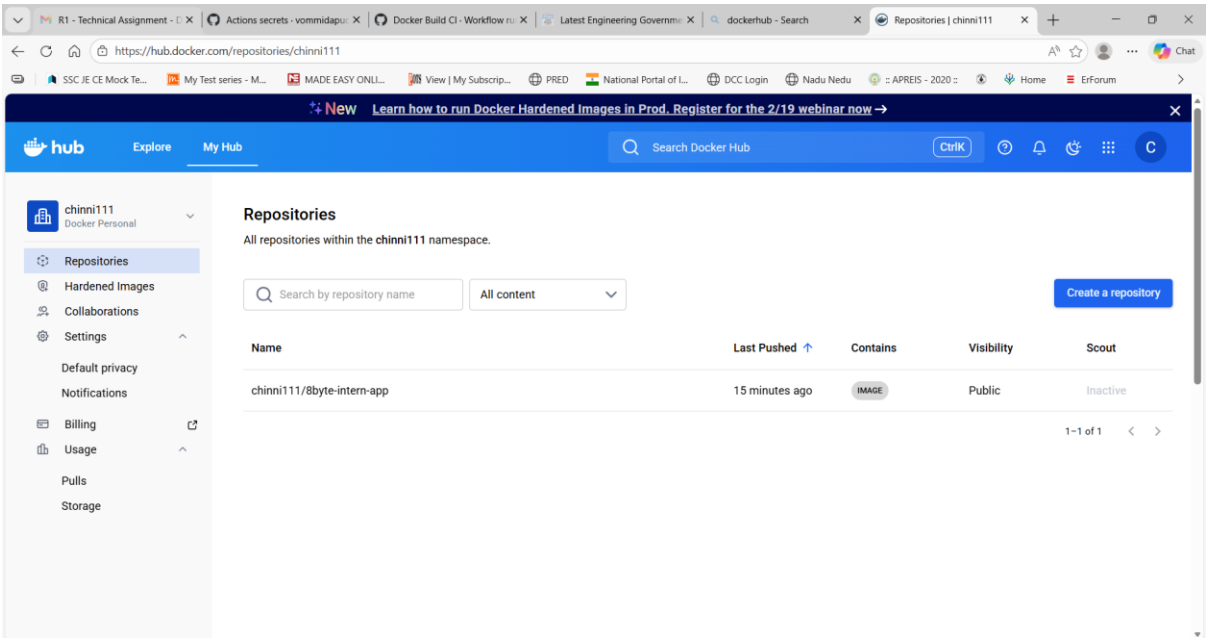
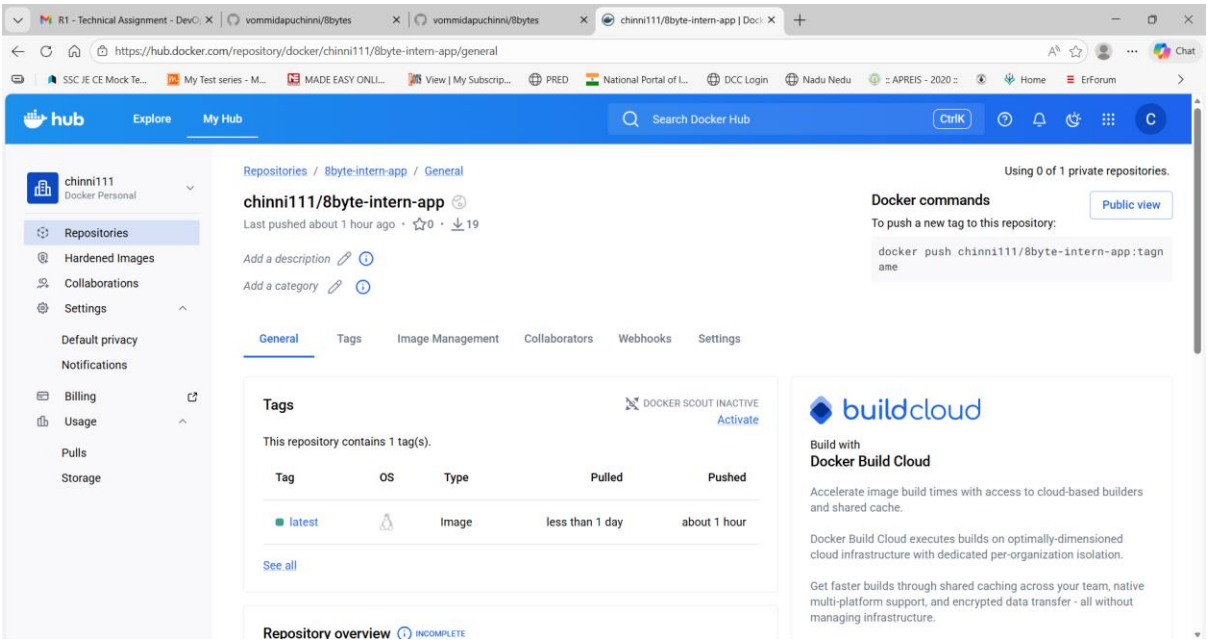
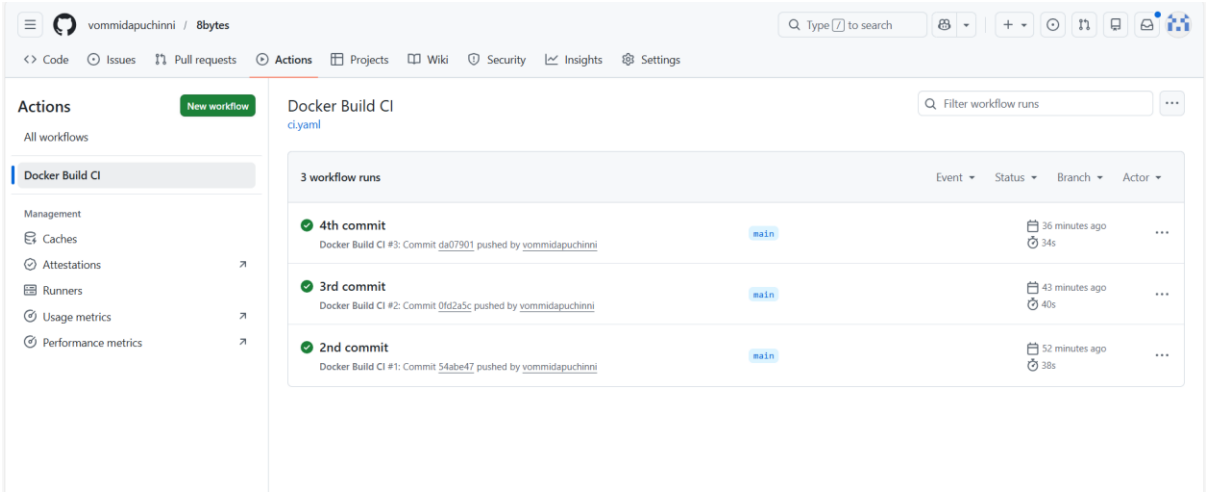


Image pushed to docker hub



This ensures every code change is automatically validated through CI.

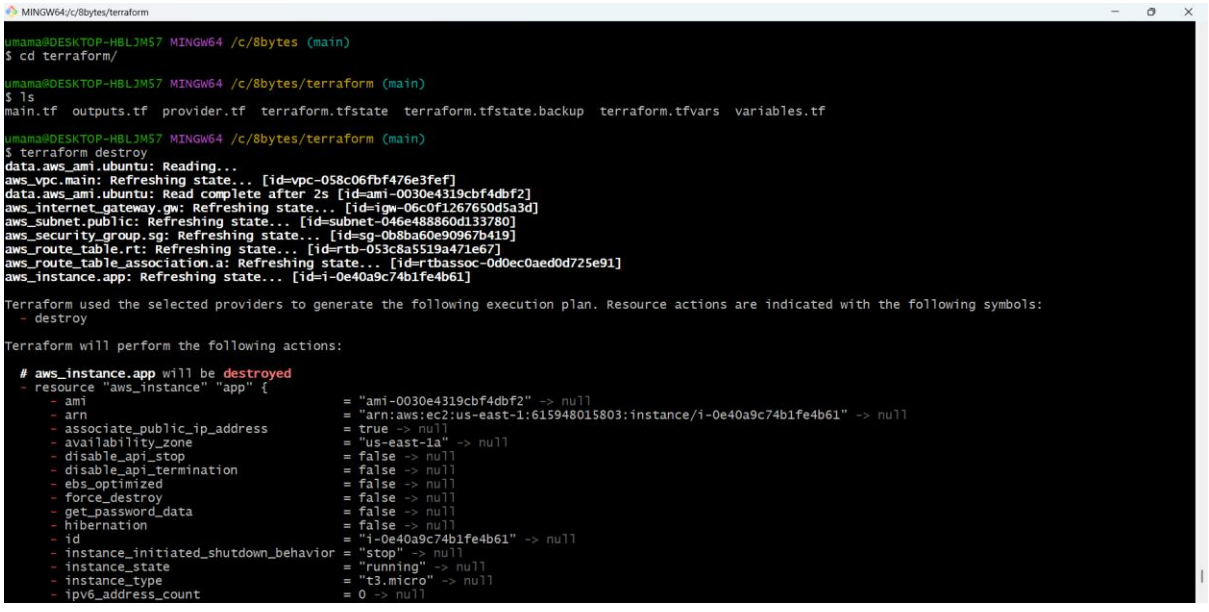


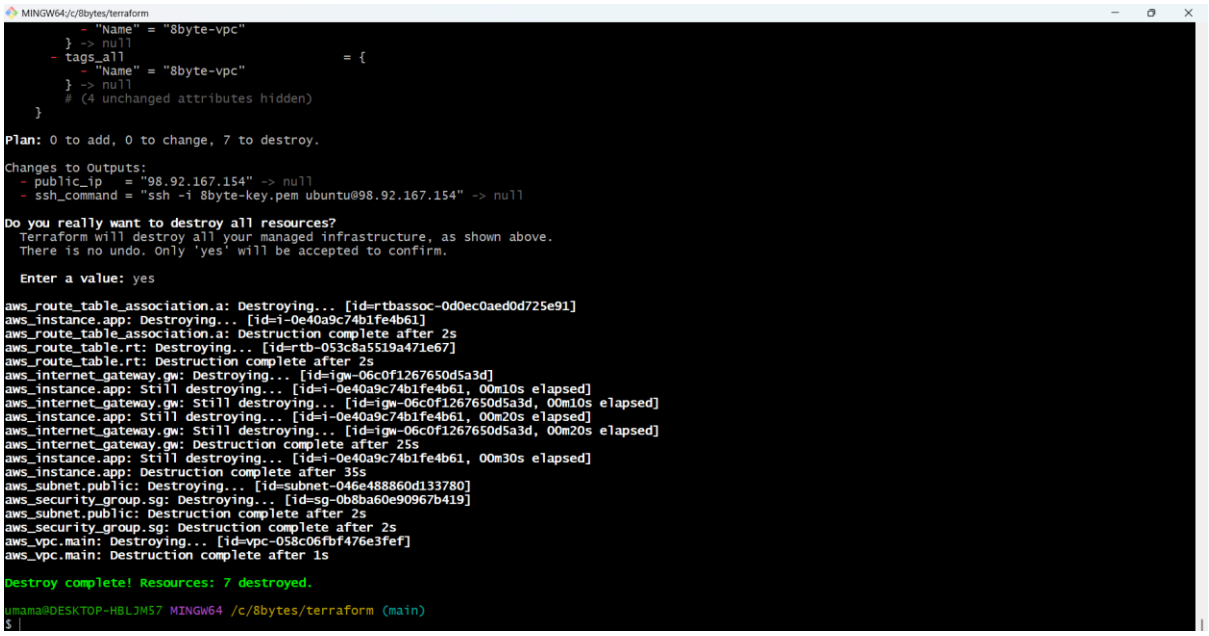
Infrastructure Lifecycle & Cost Management

After successfully deploying and verifying the application on AWS, it is important to manage resources efficiently and avoid unnecessary costs.

The AWS infrastructure was provisioned using Terraform and the application deployment was successfully verified.

To follow responsible cloud usage practices and minimize charges, the infrastructure was destroyed after validation using: terraform destroy





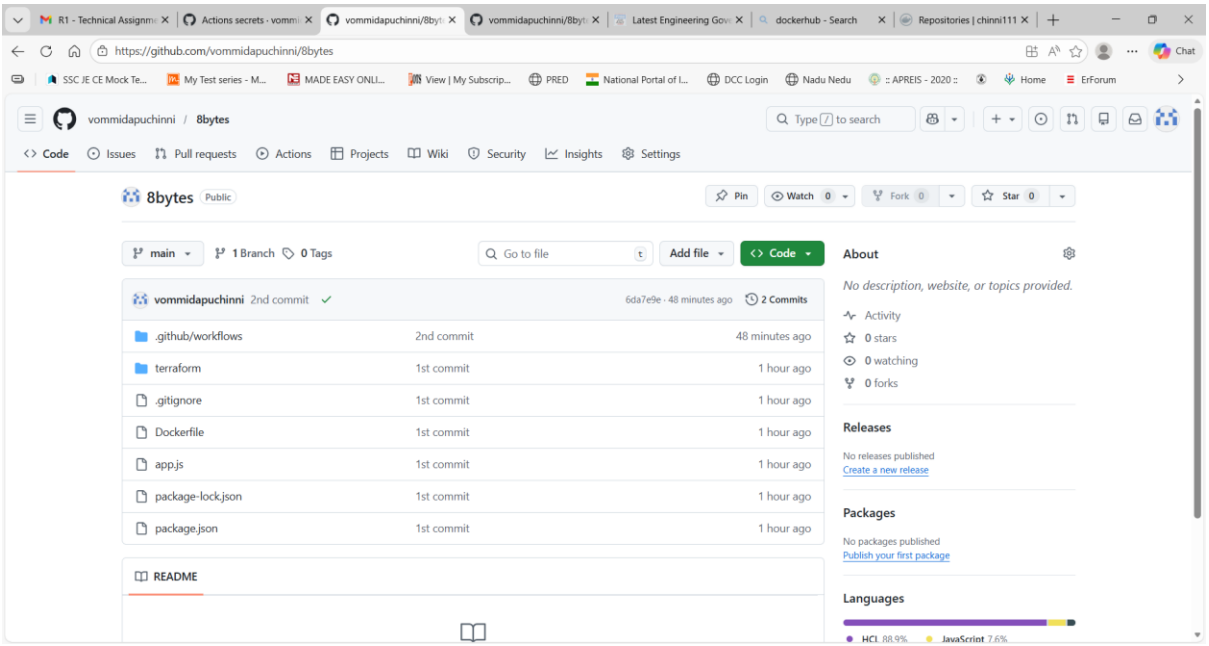
Since the setup is fully automated with Infrastructure as Code, the environment can be recreated at any time. I am prepared to redeploy the complete infrastructure live during evaluation if required.

Final Outcome

This project demonstrates a production-style DevOps workflow that covers the full lifecycle from setup to automation, including:

1. Cloud Infrastructure Provisioning
2. Secure AWS Networking
3. Dockerized Application Deployment
4. CI/CD Automation
5. Reproducible Infrastructure

FINAL Full repo



Conclusion

This project demonstrates a complete DevOps workflow by containerizing a Node.js application, provisioning AWS infrastructure with Terraform, and automating builds using GitHub Actions. It showcases practical skills in cloud deployment, Infrastructure as Code, CI/CD automation, and cost-efficient infrastructure management.