A Mini Project Report

*On*

**" Cloud-Optimized Task Management App Using Docker on AWS''**

In Subject:: : Cloud Computing and Analytics

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No** | **Name** | **PR NO.** | **ROLL NO** |
| 1 | Sejal Tongire | 22210161 | 373059 |
| 2 | Shreyash Vachkal | 22210530 | 373062 |
| 3 | Nishant Wankhede | 22210394 | 373065 |
| 4 | Purushottam Habib | 22320146 | 373073 |



Department of Artificial Intelligence and Data Science VIIT

**2024-2025**

Guidance Sign HOD Sign & Stamp Principal Sign

Renu Kachoriya

**Contents**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Topic** | | **Page No.** |
| **Chapter-**  **1** | **Introduction** | |  |
|  | 1.1 | Introduction |  |
|  | 1.2 | Requirements |  |
|  | 1.3 | Design & Problem Statement |  |
|  | 1.4 | Proposed work |  |
| **Chapter-**  **2** | **Methodology** | |  |
|  | 2.1 | Approach |  |
|  | 2.3 | Case Study |  |
|  | 2.5 | Code |  |
|  | 2.6 | Output |  |
| **Chapter-**  **3** | **Conclusion** | |  |
|  | **References** | |  |

**Chapter 1 : INTRODUCTION**

Containerization has fundamentally transformed the development and deployment of modern applications by providing a consistent and efficient method for managing software and its dependencies. This innovative technology involves encapsulating an application along with all its necessary libraries, dependencies, and configurations into a lightweight, standardized unit called a container. Unlike traditional virtual machines, containers share the host system's operating system, resulting in faster startup times, reduced resource usage, and simplified portability across environments.

One of the core advantages of containerization is **lightweight isolation**, allowing applications to run independently of the underlying infrastructure. This isolation not only improves security by separating applications but also ensures that changes or failures in one container do not affect others. Additionally, **consistent deployment** across different environments eliminates the common "it works on my machine" problem by guaranteeing that the application behaves identically regardless of where it is deployed. This reliability is critical in dynamic environments, such as development, testing, and production.

Another vital benefit is **scalability**. Containers enable applications to scale horizontally by quickly adding or removing instances in response to fluctuating demand. This flexibility is particularly important in today’s cloud-native era, where applications often need to handle unpredictable traffic patterns efficiently.

As a global leader in cloud computing, **Amazon Web Services (AWS)** provides a robust ecosystem for deploying containerized applications. AWS offers highly reliable and scalable services such as **Amazon Elastic Compute Cloud (EC2)** and **Amazon Elastic Container Service (ECS)**, which make it an ideal platform for managing containerized workloads. With AWS, organizations can leverage features like automatic scaling, monitoring, and security, ensuring that containerized applications are not only performant but also resilient and secure.

This report delves into the process of deploying a containerized ToDo application on AWS, showcasing the integration of Docker for containerization and AWS EC2 instances for hosting the application. It highlights the step-by-step methodology, including packaging the application into containers, deploying them on AWS infrastructure, and implementing robust security and monitoring mechanisms. The report aims to illustrate how combining containerization with AWS cloud infrastructure can deliver an efficient, scalable, and reliable solution for modern application deployment needs.

**1.2 Requirements**

* **Software Requirements**:
  + Docker: To containerize the ToDo app.
  + AWS CLI: For managing AWS resources.
  + Python/Node.js: Backend for the ToDo app.
  + React/HTML/CSS: Frontend for the ToDo app.
* **Hardware Requirements**:
  + Minimum 2-core CPU and 4GB RAM for local development.
  + AWS EC2 instances for production deployment.
* **AWS Services**:
  + EC2 for hosting the container.
  + ECR (Elastic Container Registry) for storing Docker images.
  + CloudWatch for monitoring.
  + Auto Scaling and Elastic Load Balancing for scaling.

**1.3 Design & Problem Statement**

**Problem Statement:**

Deploying applications across different environments often leads to inconsistencies due to dependency mismatches and configuration differences. These issues can cause applications to fail in production, resulting in downtime, troubleshooting, and delays. As applications scale, managing **scalability** and **reliability** becomes more complex, requiring dynamic scaling and resource optimization to handle increased traffic and maintain performance. Traditional infrastructure struggles to provide consistent performance and high availability, especially under fluctuating demands.

Containerization addresses these problems by packaging an application and its dependencies into a portable unit that runs consistently across environments. It simplifies deployment, eliminates dependency issues, and enables **horizontal scalability**. However, managing containers at scale on cloud platforms like AWS still presents challenges such as infrastructure setup, security, and monitoring. This report explores how AWS and Docker can help overcome these challenges and provide a reliable, scalable solution for containerized applications.

**1.3 Design**

The design of the containerized ToDo application deployment on AWS focuses on ensuring a seamless, scalable, and reliable architecture. It integrates modern development practices, robust containerization techniques, and AWS's powerful cloud infrastructure. Below are the detailed components of the design:

**1. Containerizing the ToDo Application**

* The first step is to build the ToDo application using a suitable technology stack, such as Flask or Node.js for the backend and React or plain HTML/CSS for the frontend.
* The application and its dependencies are packaged into a Docker container to ensure portability and consistency across different environments.
* A **Dockerfile** is created to automate the process of building the container image. This file specifies the base image (e.g., Python or Node.js), application dependencies, working directory, and the command to start the application.
* By containerizing the application, all dependencies are encapsulated, reducing the risk of compatibility issues when moving between development, testing, and production environments.

**2. Hosting on AWS EC2 Instances**

* AWS EC2 instances are used to host the containerized application, providing scalable compute resources. EC2 is chosen for its flexibility, enabling customization of instance types and configurations based on application requirements.
* The container image is stored in **Amazon Elastic Container Registry (ECR)**, a fully managed Docker container registry service, and then pulled to EC2 instances during deployment.
* EC2 instances are configured with appropriate **security groups** and **network access rules** to ensure secure communication between the application and its users.
* Auto Scaling is enabled for the EC2 instances to dynamically adjust the number of running instances based on the application's traffic and resource utilization, ensuring optimal performance during peak loads.

**3. Implementing Monitoring and Load Balancing**

* **Monitoring**:
  + AWS **CloudWatch** is integrated to provide real-time monitoring of resource utilization (CPU, memory, and disk), application logs, and performance metrics.
  + Alerts are configured to notify the team in case of anomalies, such as high resource usage or application errors, enabling proactive issue resolution.
* **Load Balancing**:
  + To ensure high availability and reliability, **Elastic Load Balancing (ELB)** is implemented. ELB distributes incoming traffic across multiple EC2 instances hosting the containerized app, preventing any single instance from becoming a bottleneck.
  + Load balancers also handle health checks, ensuring that only healthy instances receive traffic, thereby improving fault tolerance.

**4. Scalability and Reliability Features**

* The design emphasizes horizontal scalability by allowing the addition of more containers as traffic increases. This is achieved using Auto Scaling groups, which automatically manage the lifecycle of EC2 instances.
* The use of AWS's globally distributed infrastructure ensures reliability, as the application can be deployed in multiple availability zones to handle regional outages.
* By leveraging containers, updates to the application are deployed with minimal downtime. Containers allow rolling updates or blue-green deployments, ensuring continuous availability during updates.

**5. Security Considerations**

* The container image is scanned for vulnerabilities before deployment, ensuring a secure application environment.
* AWS **Identity and Access Management (IAM)** is used to define fine-grained access policies, limiting permissions to only what is necessary for EC2 instances and other services.
* Network security is enforced through the use of **VPCs (Virtual Private Clouds)**, security groups, and network ACLs to control inbound and outbound traffic.

**1.4 Proposed Work**

The proposed work for deploying a containerized ToDo application on AWS outlines a systematic approach to building, deploying, securing, and scaling a modern web application. The following key steps provide a comprehensive framework for achieving this goal:

**1. Develop a Containerized ToDo App Using Docker**

The first phase involves creating a ToDo application that incorporates both backend and frontend functionality:

* **Backend Development**: Use frameworks such as Flask (Python) or Node.js (JavaScript) to develop the application's backend. This will handle core functionalities like managing tasks, database operations, and API endpoints.
* **Frontend Development**: Build the user interface using technologies like React, Vue.js, or plain HTML/CSS. This ensures a user-friendly experience for managing tasks in the ToDo app.

Once the application is developed, it will be **containerized** using Docker to make it portable and consistent across environments.

* A **Dockerfile** will be created to define the container image, specifying the base operating system (e.g., Ubuntu, Alpine Linux), application dependencies, working directory, and commands to start the application.
* The Docker container ensures that all dependencies are packaged with the application, eliminating environment-specific issues and enabling seamless deployment across various systems.

This approach guarantees that the application behaves identically in development, testing, and production environments.

**2. Deploy the Docker Container on an EC2 Instance**

After building the Docker container, it needs to be hosted on a scalable and reliable infrastructure:

* **Push the Container Image to AWS ECR**: The container image will be stored in **Amazon Elastic Container Registry (ECR)**, a secure, fully managed Docker container registry provided by AWS. This ensures easy access to the container image for deployment.
* **Provision EC2 Instances**: AWS Elastic Compute Cloud (EC2) instances will be provisioned as the hosting environment. EC2 offers flexible compute capacity, allowing us to choose instance types and configurations tailored to the application's requirements.
* **Deploy the Container**: Using Docker commands or an orchestration tool like AWS ECS, the container will be pulled from the ECR and deployed on the EC2 instance. The EC2 instance will act as the runtime environment, providing compute power, memory, and storage for the containerized application.

This deployment approach combines the portability of Docker containers with the scalability of AWS EC2 instances to deliver a reliable hosting solution for the ToDo app.

**3. Configure Security and Networking for Accessibility**

Security and accessibility are critical aspects of the deployment process:

* **Security Groups**: Configure security groups to allow specific inbound and outbound traffic. For example, only HTTP and HTTPS traffic may be allowed for users accessing the application, while SSH access will be restricted to authorized developers.
* **Network Configuration**: Use **Virtual Private Clouds (VPCs)** to isolate the EC2 instances, ensuring secure communication between the application and AWS services like the database or monitoring tools. Subnets and route tables will be defined to manage network traffic effectively.
* **Access Controls**: Implement AWS Identity and Access Management (IAM) roles and policies to ensure that only authorized users and services can access resources.
* **Application Accessibility**: Configure domain name system (DNS) settings and optionally use **AWS Route 53** to map a user-friendly domain name to the application, making it easier for users to access.

These measures ensure that the ToDo app is accessible to legitimate users while safeguarding against unauthorized access and potential threats.

**4. Enable Monitoring and Auto-Scaling for Optimal Performance**

To maintain the application’s performance and availability, robust monitoring and scaling mechanisms will be implemented:

* **Monitoring with AWS CloudWatch**:
  + Integrate AWS CloudWatch to collect performance metrics like CPU usage, memory consumption, and disk activity.
  + Use log aggregation and analysis to identify application errors and diagnose issues.
  + Set up alarms to notify the team of anomalies or critical failures, enabling proactive resolution.
* **Auto-Scaling**:
  + Enable AWS Auto Scaling to dynamically adjust the number of EC2 instances hosting the application.
  + Scaling policies will be configured to respond to traffic patterns, ensuring that the application remains responsive during high-demand periods and minimizes costs during low-demand periods.
* **Load Balancing**:
  + Use **Elastic Load Balancer (ELB)** to distribute incoming traffic across multiple EC2 instances. This ensures even utilization of resources and provides fault tolerance in case of instance failure.
  + ELB will also perform health checks to ensure traffic is routed only to healthy instances.

These monitoring and scaling features will ensure that the application remains robust, performant, and cost-efficient, even under varying traffic loads.

**Chapter 2 : Methodology**

**2.1 Approach**

The project adopts a structured and systematic approach to develop, deploy, and manage the containerized ToDo application on AWS. Each step has been carefully planned to ensure the application is scalable, reliable, and secure. Below is the expanded methodology:

**1. Containerization**

The process begins with developing the ToDo application and preparing it for deployment using Docker:

* **Application Development**:
  + The ToDo application is built using a backend framework like Flask or Node.js to handle tasks such as user authentication, task creation, and database management. The frontend, developed using frameworks like React or plain HTML/CSS, ensures an intuitive and responsive user interface.
* **Dependency Management**:
  + All necessary dependencies, libraries, and configurations are identified and included in the Docker image. This ensures consistency across development, testing, and production environments.
* **Building the Docker Container**:
  + A **Dockerfile** is created, detailing the base image, dependencies, and commands to build the application. This script ensures a portable and standardized container image that can be run anywhere.
  + Docker commands such as docker build are used to create the container image, and docker run is used to test the container locally before deployment.

**2. AWS Setup**

Setting up the AWS infrastructure is crucial to ensure the containerized application is hosted securely and performs optimally:

* **EC2 Instance Provisioning**:
  + AWS Elastic Compute Cloud (EC2) instances are provisioned as virtual servers to host the application. These instances are selected based on the required compute, memory, and storage capacities to handle application workloads efficiently.
* **Storage and Registry**:
  + The Docker container image is pushed to **Amazon Elastic Container Registry (ECR)**, which acts as a secure storage repository for container images. This allows seamless integration between the container registry and AWS services.
* **Networking Configuration**:
  + Virtual Private Cloud (VPC) is set up to isolate the EC2 instances within a private network. Subnets, route tables, and gateways are configured to control internal and external communication.
  + **Security Groups**: Firewall rules are established to allow HTTP/HTTPS traffic for application users while restricting unauthorized access. SSH access is limited to administrators for enhanced security.

**3. Deployment**

Deploying the containerized application involves running the Docker container on the provisioned EC2 instance:

* **Image Deployment**:
  + The container image is pulled from Amazon ECR to the EC2 instance using Docker commands or automation scripts.
  + The container is launched and tested on the EC2 instance to verify functionality and compatibility with the AWS environment.
* **Application Accessibility**:
  + The EC2 instance is assigned a public IP or DNS name, allowing users to access the ToDo application through a web browser. DNS configuration using **Route 53** ensures a user-friendly domain name for accessibility.
* **Testing**:
  + Functional and performance testing is conducted post-deployment to ensure the application meets user expectations and handles traffic effectively.

**4. Scaling and Monitoring**

To ensure the application remains performant and resilient under varying workloads, scaling and monitoring mechanisms are implemented:

* **Horizontal Scaling**:
  + **AWS Auto Scaling** is configured to dynamically add or remove EC2 instances based on predefined policies, such as CPU utilization or incoming request rate. This ensures that the application can handle peak loads while minimizing costs during low-demand periods.
* **Load Balancing**:
  + **Elastic Load Balancer (ELB)** is deployed to distribute incoming traffic evenly across multiple EC2 instances. This not only improves resource utilization but also ensures high availability by routing traffic away from unhealthy instances.
* **Real-Time Monitoring**:
  + AWS CloudWatch is integrated to monitor application performance and infrastructure health. Metrics such as CPU usage, memory consumption, and request latency are tracked, and logs are aggregated for analysis.
  + Alarms are configured to notify administrators in case of anomalies, enabling quick troubleshooting and resolution.

**CODE:** C:\Users\Dell\Downloads>ssh -i "CCA-key.pem" ubuntu@ec2-54-81-241-148.compute-1.amazonaws.com

Welcome to Ubuntu 24.04.1 LTS (GNU/Linux 6.8.0-1019-aws x86\_64)

\* Documentation: https://help.ubuntu.com

\* Management: https://landscape.canonical.com

\* Support: https://ubuntu.com/pro

System information as of Fri Nov 22 15:21:40 UTC 2024

System load: 0.1 Processes: 110

Usage of /: 46.4% of 6.71GB Users logged in: 1

Memory usage: 26% IPv4 address for enX0: 172.31.39.101

Swap usage: 0%

\* Ubuntu Pro delivers the most comprehensive open source security and

compliance features.

https://ubuntu.com/aws/pro

Expanded Security Maintenance for Applications is not enabled.

36 updates can be applied immediately.

To see these additional updates run: apt list --upgradable

Enable ESM Apps to receive additional future security updates.

See https://ubuntu.com/esm or run: sudo pro status

Last login: Fri Nov 22 15:04:23 2024 from 117.99.249.102

ubuntu@ip-172-31-39-101:~$ cd django-todo

ubuntu@ip-172-31-39-101:~/django-todo$ ls

Dockerfile LICENSE README.md db.sqlite3 dockerfile manage.py staticfiles todoApp todos venv

ubuntu@ip-172-31-39-101:~/django-todo$ python3 manage.py runserver 0.0.0.0:8000

Watching for file changes with StatReloader

Performing system checks...

System check identified some issues:

WARNINGS:

todos.Todo: (models.W042) Auto-created primary key used when not defining a primary key type, by default 'django.db.models.AutoField'.

HINT: Configure the DEFAULT\_AUTO\_FIELD setting or the TodosConfig.default\_auto\_field attribute to point to a subclass of AutoField, e.g. 'django.db.models.BigAutoField'.

System check identified 1 issue (0 silenced).

November 22, 2024 - 20:52:11

Django version 4.2.11, using settings 'todoApp.settings'

Starting development server at http://0.0.0.0:8000/

Quit the server with CONTROL-C.

^Cubuntu@ip-172-31-39-101:~/django-todo$ sudo apt instal docker.io

E: Invalid operation instal

ubuntu@ip-172-31-39-101:~/django-todo$ sudo apt install docker.io

Reading package lists... Done

Building dependency tree... Done

Reading state information... Done

docker.io is already the newest version (24.0.7-0ubuntu4.1).

0 upgraded, 0 newly installed, 0 to remove and 32 not upgraded.

ubuntu@ip-172-31-39-101:~/django-todo$ vi Dockerfile

ubuntu@ip-172-31-39-101:~/django-todo$ docker build . -t todo-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/

permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Post "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/build?buildargs=%7B%7D&cachefrom=%5B%5D&cgroupparent=&cpuperiod=0&cpuquota=0&cpusetcpus=&cpusetmems=&cpushares=0&dockerfile=Dockerfile&labels=%7B%7D&memory=0&memswap=0&networkmode=default&rm=1&shmsize=0&t=todo-app&target=&ulimits=null&version=1": dial unix /var/run/docker.sock: connect: permission denied

ubuntu@ip-172-31-39-101:~/django-todo$ sudo docker build . -t todo-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 49.97MB

Step 1/7 : FROM python:3.11

3.11: Pulling from library/python

b2b31b28ee3c: Pull complete

c3cc7b6f0473: Pull complete

2112e5e7c3ff: Pull complete

af247aac0764: Pull complete

6f47d6e91428: Pull complete

0ded565f3746: Pull complete

c03fa7c846df: Pull complete

Digest: sha256:706d1233c61a31507c4f8939cfd6a924610b51174c095f33e2c537fb904a1e76

Status: Downloaded newer image for python:3.11

---> fa2b568a9d25

Step 2/7 : RUN apt-get update && apt-get install -y python3-distutils

---> Running in 4e07619b0030

Get:1 http://deb.debian.org/debian bookworm InRelease [151 kB]

Get:2 http://deb.debian.org/debian bookworm-updates InRelease [55.4 kB]

Get:3 http://deb.debian.org/debian-security bookworm-security InRelease [48.0 kB]

Get:4 http://deb.debian.org/debian bookworm/main amd64 Packages [8789 kB]

Get:5 http://deb.debian.org/debian bookworm-updates/main amd64 Packages [2468 B]

Get:6 http://deb.debian.org/debian-security bookworm-security/main amd64 Packages [204 kB]

Fetched 9250 kB in 1s (6197 kB/s)

Reading package lists...

Reading package lists...

Building dependency tree...

Reading state information...

python3-distutils is already the newest version (3.11.2-3).

python3-distutils set to manually installed.

0 upgraded, 0 newly installed, 0 to remove and 2 not upgraded.

Removing intermediate container 4e07619b0030

---> 0f18c10a47b6

Step 3/7 : RUN pip install django==3.2

---> Running in 493200d06ff5

Collecting django==3.2

Downloading Django-3.2-py3-none-any.whl.metadata (3.9 kB)

Collecting asgiref<4,>=3.3.2 (from django==3.2)

Downloading asgiref-3.8.1-py3-none-any.whl.metadata (9.3 kB)

Collecting pytz (from django==3.2)

Downloading pytz-2024.2-py2.py3-none-any.whl.metadata (22 kB)

Collecting sqlparse>=0.2.2 (from django==3.2)

Downloading sqlparse-0.5.2-py3-none-any.whl.metadata (3.9 kB)

Downloading Django-3.2-py3-none-any.whl (7.9 MB)

━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.9/7.9 MB 62.7 MB/s eta 0:00:00

Downloading asgiref-3.8.1-py3-none-any.whl (23 kB)

Downloading sqlparse-0.5.2-py3-none-any.whl (44 kB)

━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 44.4/44.4 kB 3.7 MB/s eta 0:00:00

Downloading pytz-2024.2-py2.py3-none-any.whl (508 kB)

━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 508.0/508.0 kB 41.1 MB/s eta 0:00:00

Installing collected packages: pytz, sqlparse, asgiref, django

Successfully installed asgiref-3.8.1 django-3.2 pytz-2024.2 sqlparse-0.5.2

WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system package manager. It is recommended to use a virtual environment instead: https://pip.pypa.io/warnings/venv

[notice] A new release of pip is available: 24.0 -> 24.3.1

[notice] To update, run: pip install --upgrade pip

Removing intermediate container 493200d06ff5

---> de36403c4714

Step 4/7 : COPY . .

---> 5ada1e3d3fbf

Step 5/7 : RUN python manage.py migrate

---> Running in e0fe4128a50d

System check identified some issues:

WARNINGS:

todos.Todo: (models.W042) Auto-created primary key used when not defining a primary key type, by default 'django.db.models.AutoField'.

HINT: Configure the DEFAULT\_AUTO\_FIELD setting or the TodosConfig.default\_auto\_field attribute to point to a subclass of AutoField, e.g. 'django.db.models.BigAutoField'.

Operations to perform:

Apply all migrations: admin, auth, contenttypes, sessions, todos

Running migrations:

No migrations to apply.

Removing intermediate container e0fe4128a50d

---> 457733639552

Step 6/7 : EXPOSE 8000

---> Running in 46a3e601ada1

Removing intermediate container 46a3e601ada1

---> 72136dd415c7

Step 7/7 : CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]

---> Running in 1ac0f5714daa

Removing intermediate container 1ac0f5714daa

---> 658a75f3a65e

Successfully built 658a75f3a65e

Successfully tagged todo-app:latest

ubuntu@ip-172-31-39-101:~/django-todo$ sudo docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

ubuntu@ip-172-31-39-101:~/django-todo$ sudo docker run -p 8000:8000 658a75f3a65e

Watching for file changes with StatReloader

System check identified some issues:

WARNINGS:

todos.Todo: (models.W042) Auto-created primary key used when not defining a primary key type, by default 'django.db.models.AutoField'.

HINT: Configure the DEFAULT\_AUTO\_FIELD setting or the TodosConfig.default\_auto\_field attribute to point to a subclass of AutoField, e.g. 'django.db.models.BigAutoField'.

System check identified 1 issue (0 silenced).

[22/Nov/2024 21:03:10] code 400, message Bad request version ('\x19í\x96xM\x17}\x99')

[22/Nov/2024 21:03:10] You're accessing the development server over HTTPS, but it only supports HTTP.

[22/Nov/2024 21:03:10] code 400, message Bad request version ('òµT+VPU\x1b±¨\x05¼ù\x93\x02´õ\x8dÞ')

[22/Nov/2024 21:03:10] You're accessing the development server over HTTPS, but it only supports HTTP.

[22/Nov/2024 21:03:11] "GET / HTTP/1.1" 302 0

[22/Nov/2024 21:03:11] "GET /todos HTTP/1.1" 301 0

[22/Nov/2024 21:03:12] "GET /todos/ HTTP/1.1" 200 4717

[22/Nov/2024 21:03:12] "GET /static/css/style.css HTTP/1.1" 200 225

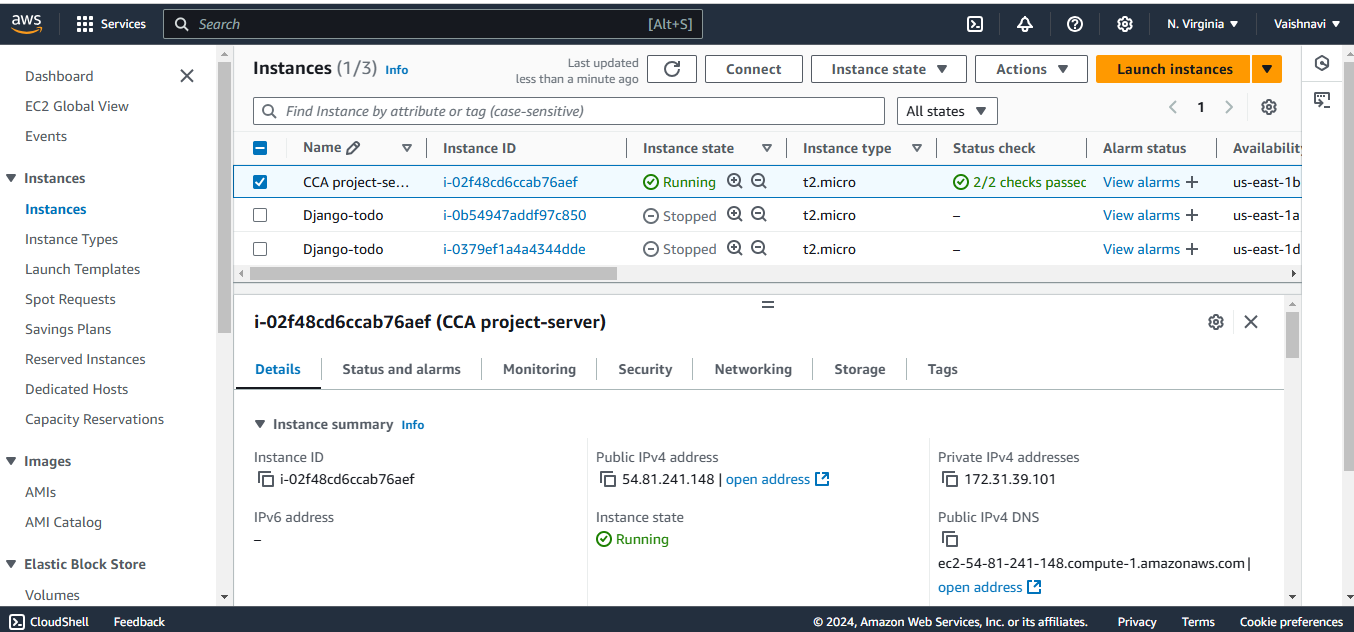
Not Found: /favicon.ico

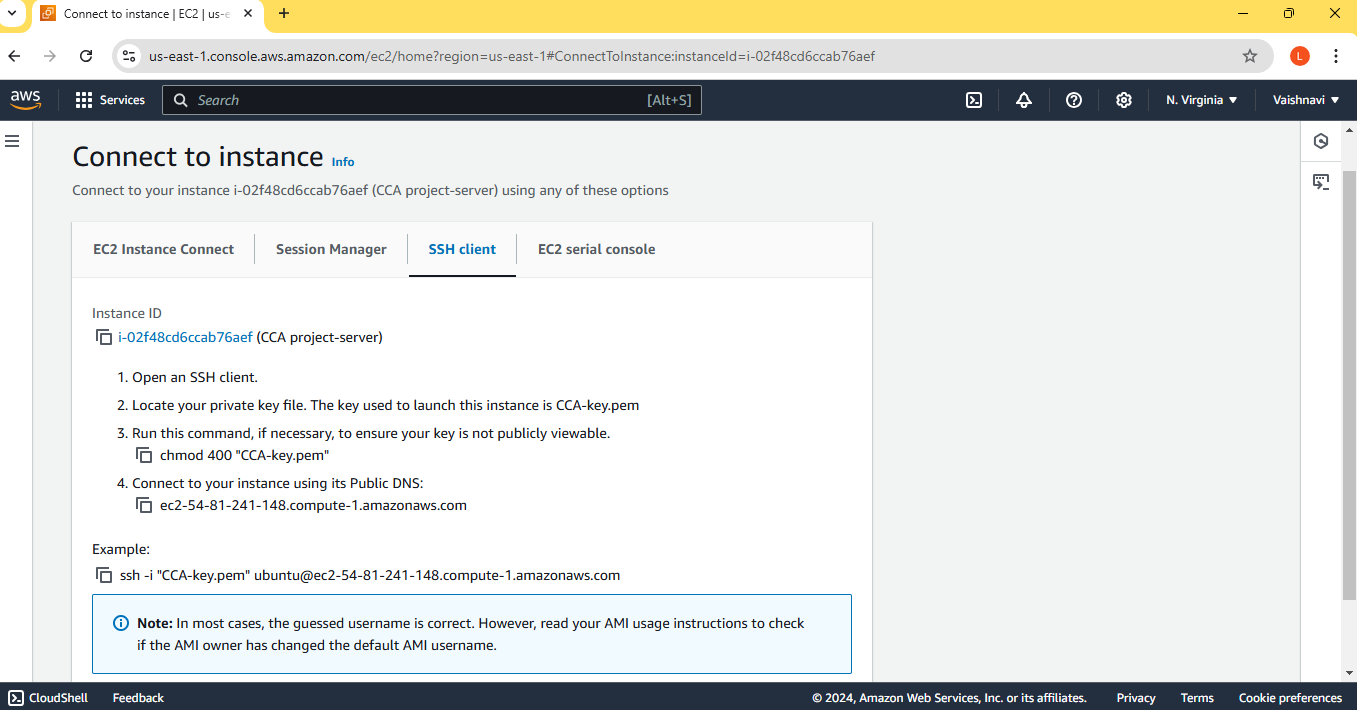
[22/Nov/2024 21:03:13] "GET /favicon.ico HTTP/1.1" 404 2459

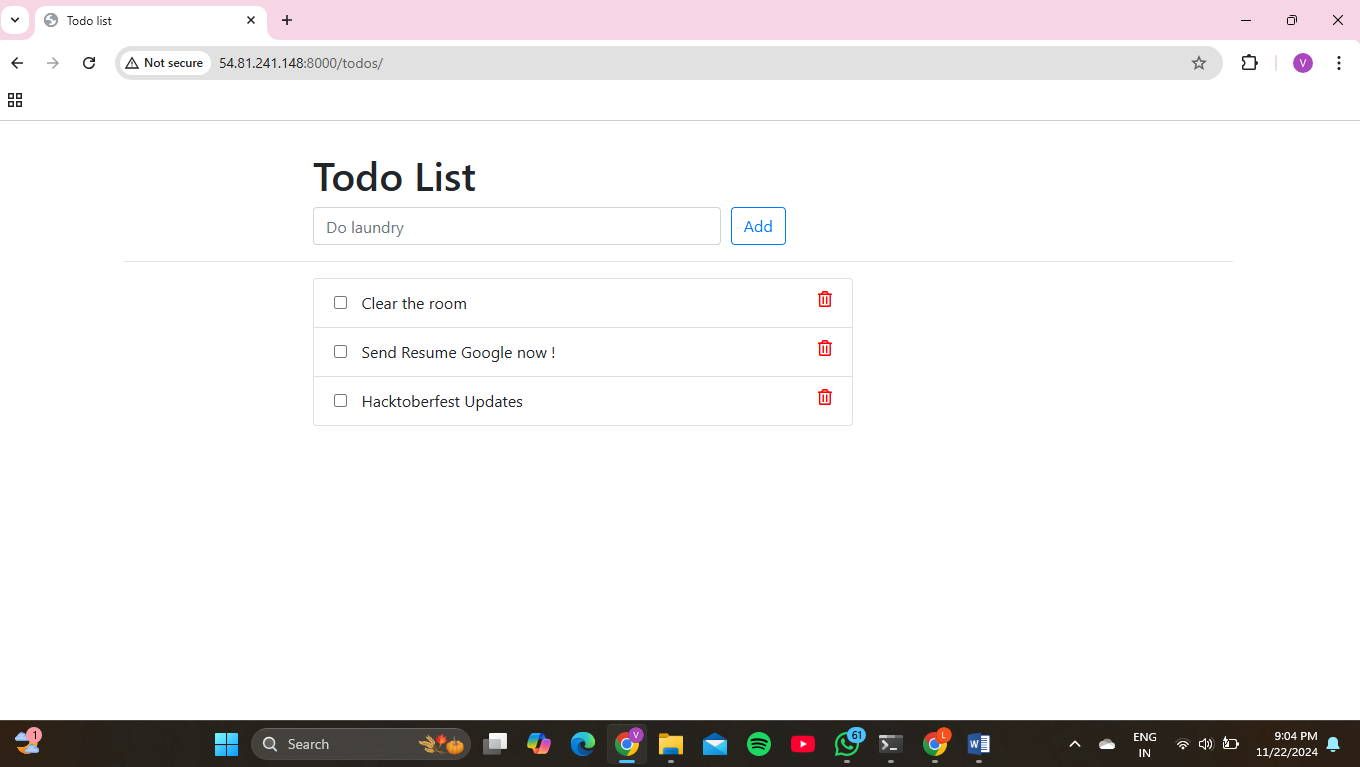
[22/Nov/2024 21:03:17] "GET /todos/23/delete HTTP/1.1" 302 0

[22/Nov/2024 21:03:17] "GET /todos/ HTTP/1.1" 200 4102

**Images:**







**Chapter 3 : CONCLUSION**

**Summary:**

* Successfully deployed a containerized ToDo application on AWS.
* Leveraged Docker for consistency, EC2 for hosting, and AWS services for scalability and monitoring.

**Lessons Learned**:

* Importance of security configurations for containerized environments.
* Need for thorough monitoring to identify bottlenecks.

**Future Scope**:

* Explore Kubernetes for container orchestration.
* Integrate advanced AWS services like Fargate or Lambda for a serverless approach.
* Optimize costs through reserved instances or spot instances.

**REFERENCES-**

1. <https://nutanix.medium.com/containerized-application-deployment-on-aws-using-docker-cloud-470f07565492>
2. <https://aws.amazon.com/app2container/>
3. <https://www.fernandomc.com/posts/easiest-way-to-deploy-aws-containers/>
4. <https://dev.to/swapi123/deployed-a-todo-app-on-kubernetes-2og1>
5. <https://docs.aws.amazon.com/whitepapers/latest/develop-deploy-dotnet-apps-on-aws/running-applications-in-containers.html>