PART 1 — Kubernetes Deployment of TodoPro on Amazon EKS

Title:

TodoPro – Containerization and Kubernetes Deployment on Amazon EKS (Part 1 of the DevOps Project)

---

1. Introduction

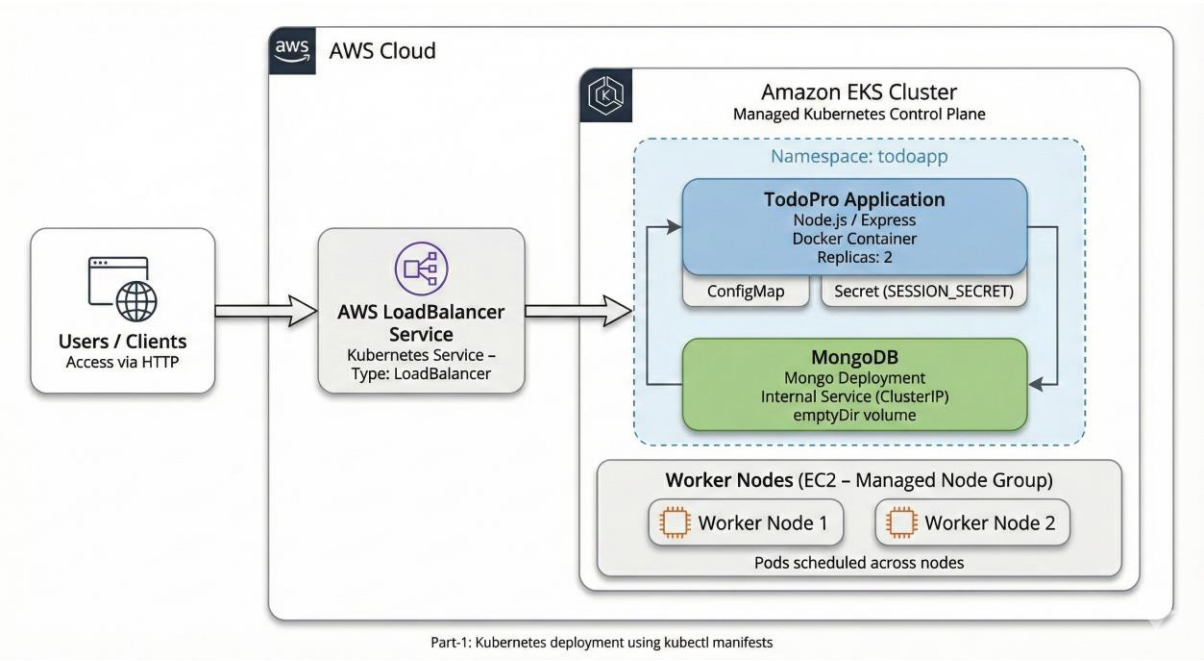
TodoPro is a full-stack Node.js application with MongoDB for persistent data storage. This document outlines Part 1 of the DevOps project, focusing on:

- Containerizing the application
- Building and testing Docker images
- Pushing images to Docker Hub
- Deploying the application on Amazon EKS

This forms the foundation for Part 2, where the deployment will be fully automated using Helm charts.

---

2. Architecture



Part-1: Kubernetes deployment using kubectl manifests

---

3. Prerequisites

3.1 Local Machine Requirements

- Node.js installed
- Docker installed
- AWS CLI installed
- kubectl installed
- eksctl installed

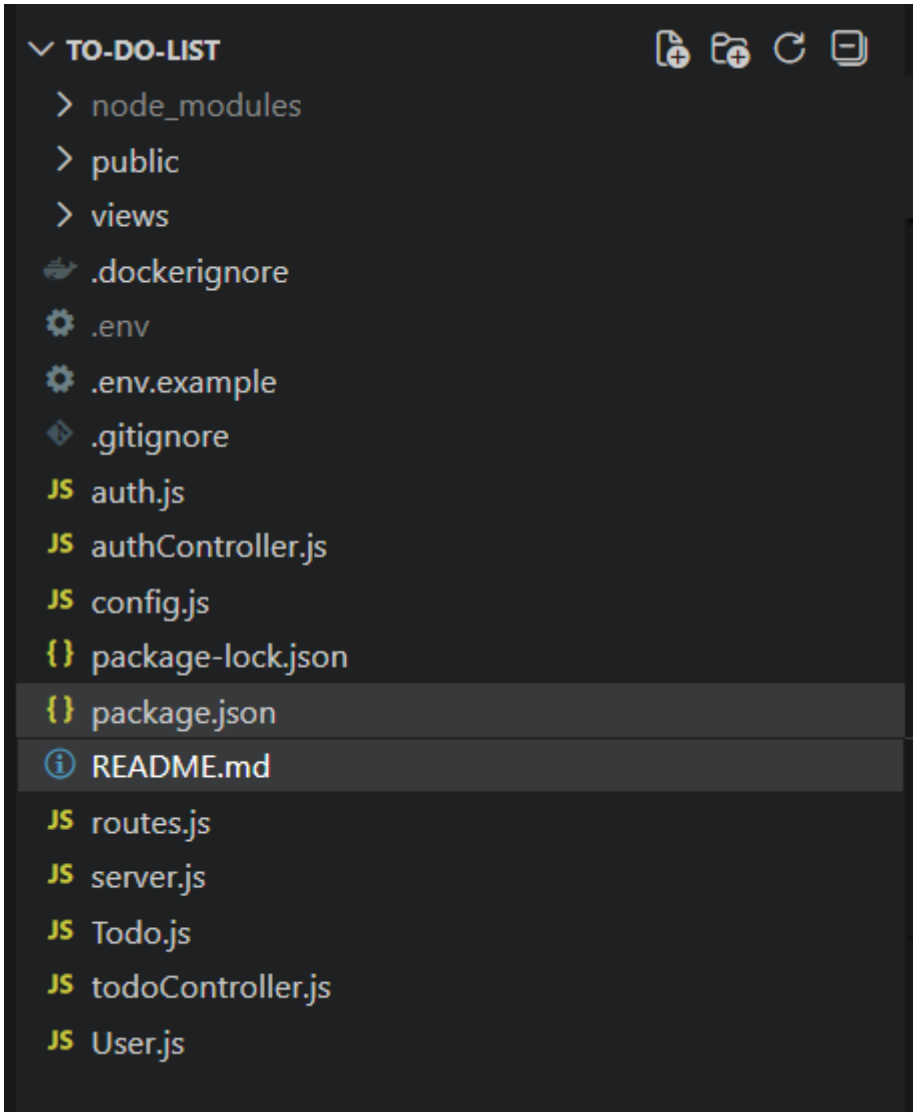
Verify installations:

```
docker --version
aws --version
kubectl version --client
eksctl version
```

3.2 AWS Requirements

- AWS account with required permissions
- IAM user with programmatic access
- VPC (created automatically by eksctl if not provided)

3.3 Application Requirements



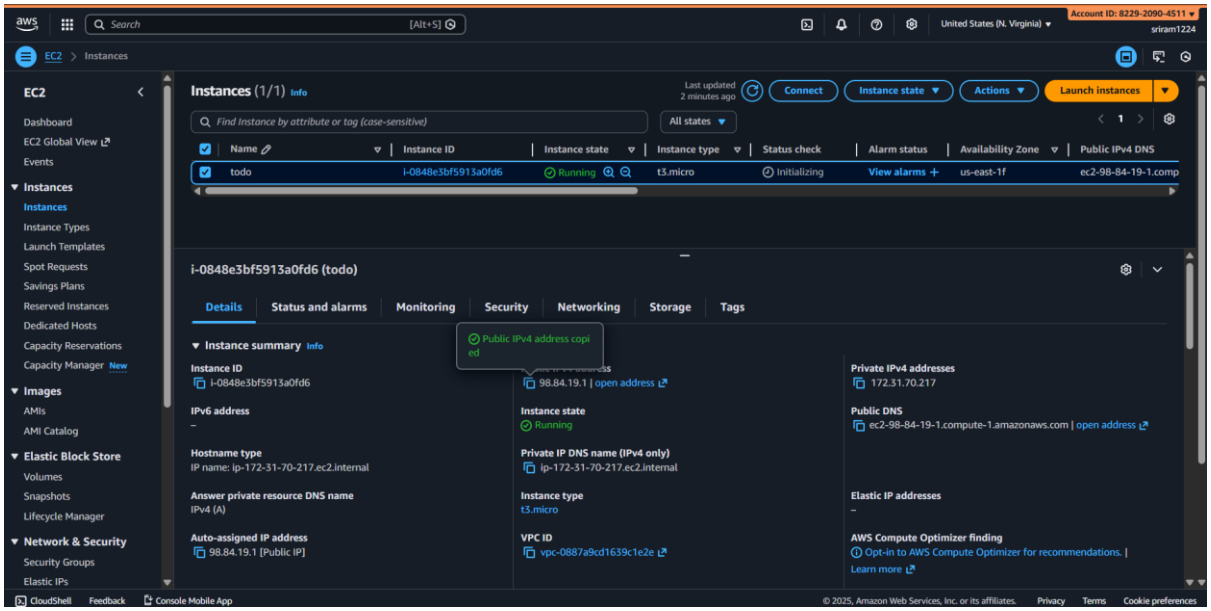
---

4 . Deployment Steps

This section begins from **Docker image creation**.

Step 1 — Launching the EC2 Instance

Screenshot



Description

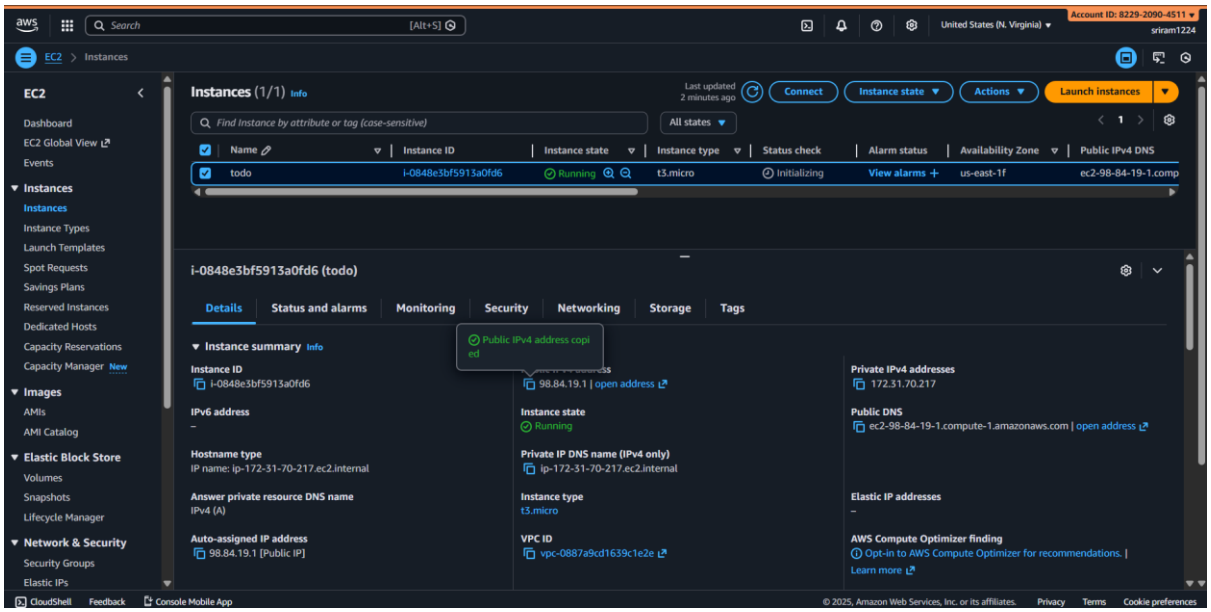
This screenshot shows the EC2 instance launched for building and testing the Docker images required for the TodoPro application.
The instance is in a running state and displays key details such as:

- **Instance ID**
- **Instance Type**
- **Public IPv4 address**
- **Status checks passed**
- **Network and security settings**

This EC2 environment serves as the workspace where Docker is installed, the project repository is cloned, and the Docker images are built and validated before deploying them to Kubernetes.

Step 2 — Verifying the EC2 Instance Details

Screenshot



Description

This screenshot displays the detailed information of the EC2 instance used for building and testing Docker images before deploying to Kubernetes.

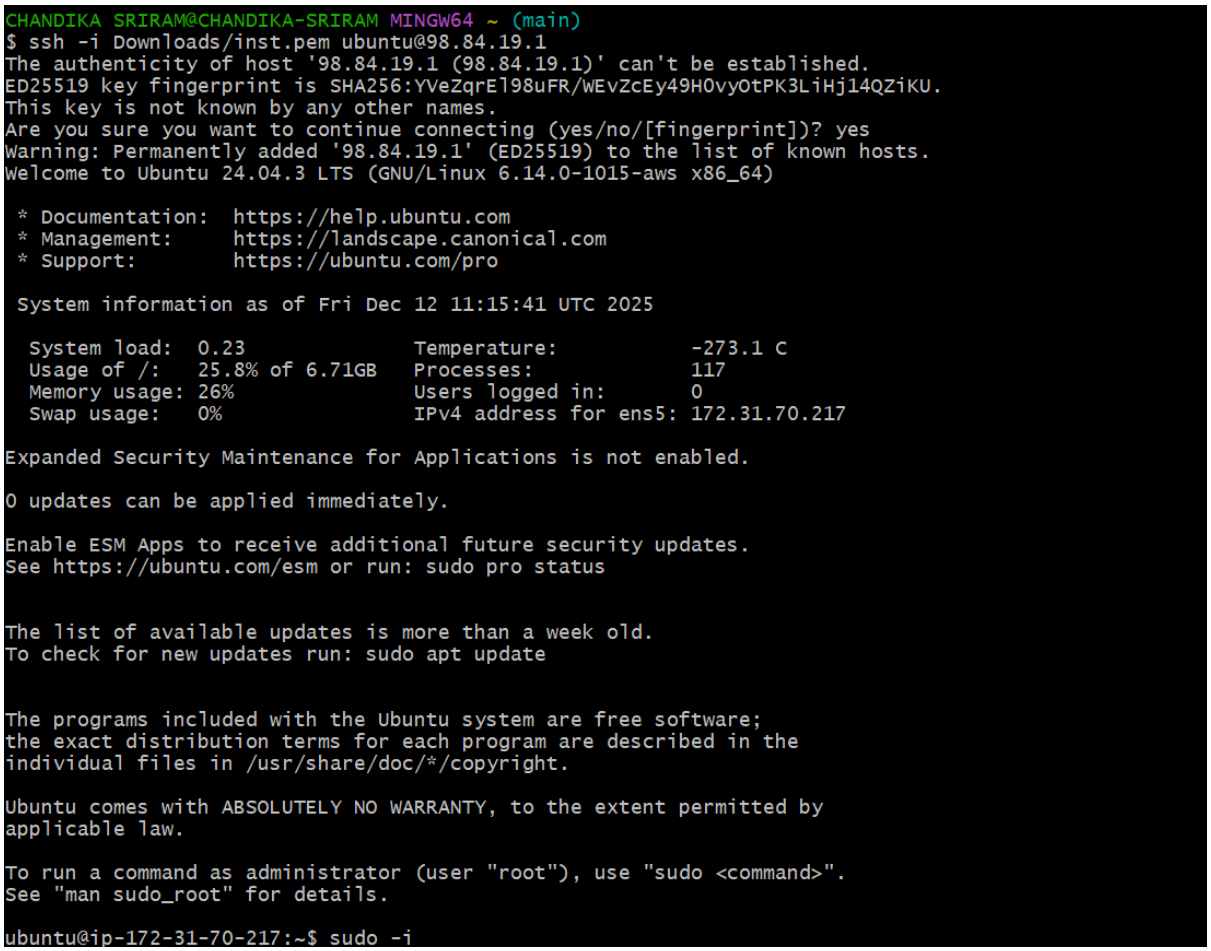
Key elements visible in the screenshot include:

- **Instance ID**: Unique identifier for the EC2 instance
- **Instance State**: Indicates the machine is currently in a *Running* state
- **Instance Type**: The selected instance type (`t3.micro`) used for this environment
- **Public IPv4 Address**: Used to access the server remotely and test the application
- **Private IP Address**: Used for internal communication within the VPC
- **Availability Zone**: Specifies where the instance is hosted (us-east-1f)
- **Hostname & DNS**: Shows AWS-generated public DNS for direct access

This confirms that the EC2 instance is fully operational and ready for the next steps, including SSH access, Docker installation, and application setup.

Step 3 — Connecting to the EC2 Instance via SSH

Screenshot



Description

This screenshot confirms a successful SSH connection to the EC2 instance using the `.pem` key pair. After authentication, the terminal displays the Ubuntu system welcome message and system resource summary, indicating the instance is operational and ready for configuration.

Key points shown in the screenshot:

- **SSH Authentication Successful:** The user connects using the private key associated with the EC2 instance.
- **Ubuntu Version:** The instance is running Ubuntu 24.04 LTS (latest stable release).
- **System Load & Resource Usage:** Shows CPU load, memory usage, and disk usage.
- **Network Details:** Displays the internal private IP assigned to the instance.
- **Privilege Escalation:** The `sudo -i` command elevates the session to the root user for installation tasks.

At this stage, the server is fully accessible, and the environment is ready for installing Docker, cloning the application repository, and building container images.

Step 4 — Adding Docker's Official Repository and Updating Packages

Screenshot

```
root@ip-172-31-70-217:~# # Add Docker's official GPG key:
sudo apt update
sudo apt install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
sudo tee /etc/apt/sources.list.d/docker.sources <<EOF
Types: deb
URIs: https://download.docker.com/linux/ubuntu
Suites: $(. /etc/os-release && echo "${UBUNTU_CODENAME:-$VERSION_CODENAME}")
Components: stable
Signed-By: /etc/apt/keyrings/docker.asc
EOF

sudo apt update
Hit:1 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble InRelease
Get:2 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates InRelease [126 kB]
Get:3 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-backports InRelease [126 kB]
Get:4 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble/universe amd64 Packages [15.0 MB]
Get:5 http://security.ubuntu.com/ubuntu noble-security InRelease [126 kB]
Get:6 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble/universe Translation-en [5982 kB]
Get:7 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble/universe amd64 Components [3871 kB]
Get:8 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble/universe amd64 c-n-f Metadata [301 kB]
Get:9 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble/multiverse amd64 Packages [269 kB]
Get:10 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble/multiverse Translation-en [118 kB]
Get:11 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble/multiverse amd64 Components [35.0 kB]
Get:12 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble/multiverse amd64 c-n-f Metadata [8328 B]
Get:13 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates/main amd64 Packages [1675 kB]
Get:14 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates/main Translation-en [309 kB]
Get:15 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates/main amd64 Components [175 kB]
Get:16 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates/main amd64 c-n-f Metadata [15.8 kB]
Get:17 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates/universe amd64 Packages [1501 kB]
Get:18 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates/universe Translation-en [304 kB]
Get:19 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates/universe amd64 Components [378 kB]
Get:20 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates/universe amd64 c-n-f Metadata [31.4 kB]
Get:21 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates/restricted amd64 Packages [2404 kB]
Get:22 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates/restricted Translation-en [549 kB]
Get:23 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates/restricted amd64 Components [212 B]
Get:24 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates/multiverse amd64 Packages [30.3 kB]
Get:25 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates/multiverse Translation-en [5808 B]
Get:26 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates/multiverse amd64 Components [940 B]
Get:27 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates/multiverse amd64 c-n-f Metadata [484 B]
Get:28 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-backports/main amd64 Packages [40.4 kB]
Get:29 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-backports/main Translation-en [9208 B]
Get:30 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-backports/main amd64 Components [7140 B]
Get:31 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-backports/main amd64 c-n-f Metadata [368 B]
Get:32 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-backports/universe amd64 Packages [29.2 kB]
Get:33 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-backports/universe Translation-en [17.6 kB]
Get:34 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-backports/universe amd64 Components [11.0 kB]
Get:35 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-backports/universe amd64 c-n-f Metadata [1444 B]
```

Description

This screenshot shows the process of configuring the EC2 instance to install Docker from the **official Docker repository**, ensuring access to the latest stable Docker packages.

The key activities visible in the screenshot include:

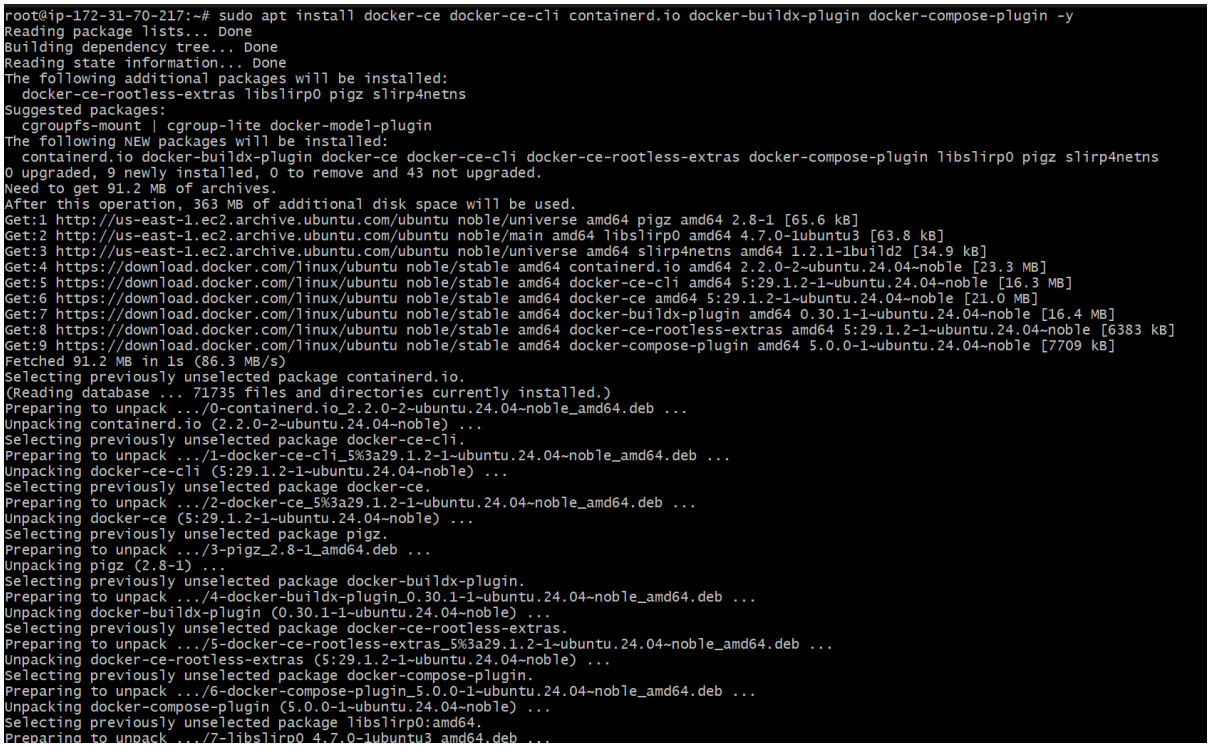
- **Updating package lists** to fetch the latest metadata
- **Installing required dependencies** (such as `ca-certificates`, `curl`)
- **Adding Docker's official GPG key** to validate Docker packages
- **Configuring the Docker APT repository** for Ubuntu
- **Refreshing the package index again** to include the newly added Docker repository

This step ensures that Docker is installed from a secure, verified, and up-to-date source rather than using outdated default Ubuntu packages.

The instance is now fully prepared to install Docker Engine, Docker CLI, and Docker Compose in the next step.

Step 5 — Installing Docker Engine, Docker CLI, and Docker Compose

Screenshot



Description

This screenshot shows the installation of core Docker components on the EC2 instance after configuring Docker's official repository. The packages installed include:

- **docker-ce** — Docker Community Edition Engine
- **docker-ce-cli** — Docker command-line interface

- **containerd.io** — Docker container runtime
- **docker-buildx-plugin** — Enhanced builder for multi-platform image builds
- **docker-compose-plugin** — Enables Docker Compose functionality (v2+)

The output displays:

- Package downloads from official Docker repositories
- Unpacking and installing of all required dependencies
- Successful setup of Docker Engine and associated tools

With this step completed, the EC2 instance is now fully ready to build Docker images, run containers, and test the TodoPro application locally before deploying it to Kubernetes.

Step 6 — Cloning the TodoPro Project from GitHub

Screenshot

```
root@ip-172-31-70-217:~# git clone https://github.com/sriramch163/Todo-List.git
Cloning into 'Todo-List'...
remote: Enumerating objects: 23, done.
remote: Counting objects: 100% (23/23), done.
remote: Compressing objects: 100% (22/22), done.
remote: Total 23 (delta 1), reused 23 (delta 1), pack-reused 0 (from 0)
Receiving objects: 100% (23/23), 30.32 KiB | 7.58 MiB/s, done.
Resolving deltas: 100% (1/1), done.
root@ip-172-31-70-217:~# ls
Todo-List  snap
root@ip-172-31-70-217:~# cd Todo-List
root@ip-172-31-70-217:~/Todo-List#
```

Description

This screenshot shows the TodoPro application repository being cloned from GitHub onto the EC2 instance. This step brings the complete application source code into the server, enabling Docker image creation and container testing.

Key actions captured in the screenshot:

- **Git Clone:** The command clones the repository located at `https://github.com/sriramch163/Todo-List.git`
- **Repository Download:** Git successfully downloads 23 objects including source files, controllers, views, and configuration files.
- **Directory Listing:** The project folder `Todo-List` is now present on the instance.
- **Navigation:** The directory is entered using `cd Todo-List`, preparing the environment for Dockerfile creation and image building.

At this stage, the source code is available locally on the EC2 machine, which is essential for creating Docker images and proceeding with application containerization.

Step 7 — Creating the Dockerfile for the TodoPro Application

Screenshot

Description

This screenshot shows the Dockerfile created for containerizing the TodoPro Node.js application.
The Dockerfile defines how the application image is built, including dependency installation, adding a non-root user, exposing the service port, and starting the Node.js server.

The key stages visible in the Dockerfile include:

- **Base Image:** Uses `node:18-alpine` (lightweight and secure Node.js LTS version)
- **Working Directory:** Sets `/app` as the workspace for the application
- **Copying Package Files:** Copies `package.json` and installs only production dependencies
- **Copying Application Code:** Moves all necessary files into the container
- **Security Best Practice:** Creates a non-root user (`todoapp`) to run the application
- **Port Exposure:** Exposes port **3000** for accessing the service
- **Startup Command:** Runs `npm start` to launch the Node.js server

Note on Image Usage (Important for Documentation)

If you are **creating your own custom Docker images**, you will use the Dockerfile shown above to build the TodoPro application image.

For MongoDB, you can either:

- Use the **official MongoDB image** from Docker Hub (`mongo:6.0`),
  or
- Use your own custom MongoDB image uploaded to Docker Hub.

Custom Images Used in This Project

You used the following images hosted on Docker Hub:

Application Image:

`sriram084/todo:v1`

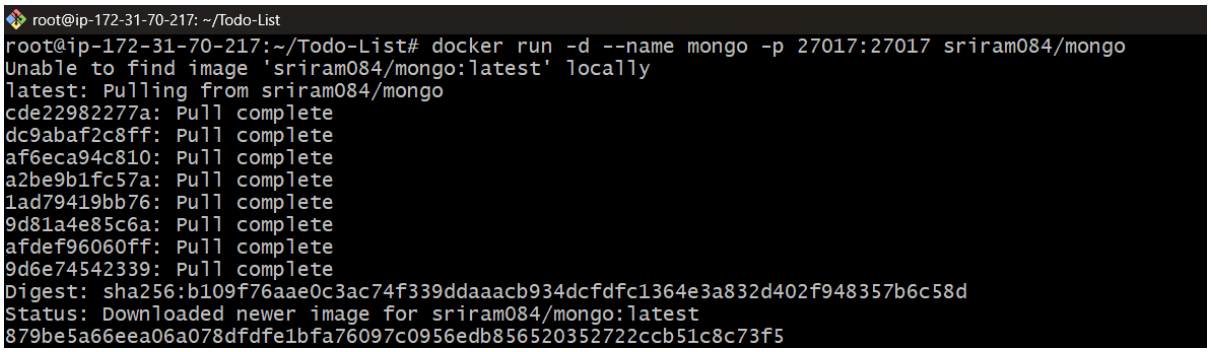
MongoDB Image:

`sriram084/mongo:latest`
Docker Hub Repository Link : https://hub.docker.com/repositories/sriram084

These images were pulled and executed successfully on the EC2 instance, confirming that the application and MongoDB operated correctly in a containerized environment.

Step 8 — Pulling and Running the MongoDB Container

Screenshot



Description

This screenshot shows the MongoDB container being launched using the image stored in your Docker Hub repository:

`sriram084/mongo:latest`

Since the image was not available locally, Docker automatically pulled it from Docker Hub. The output shows:

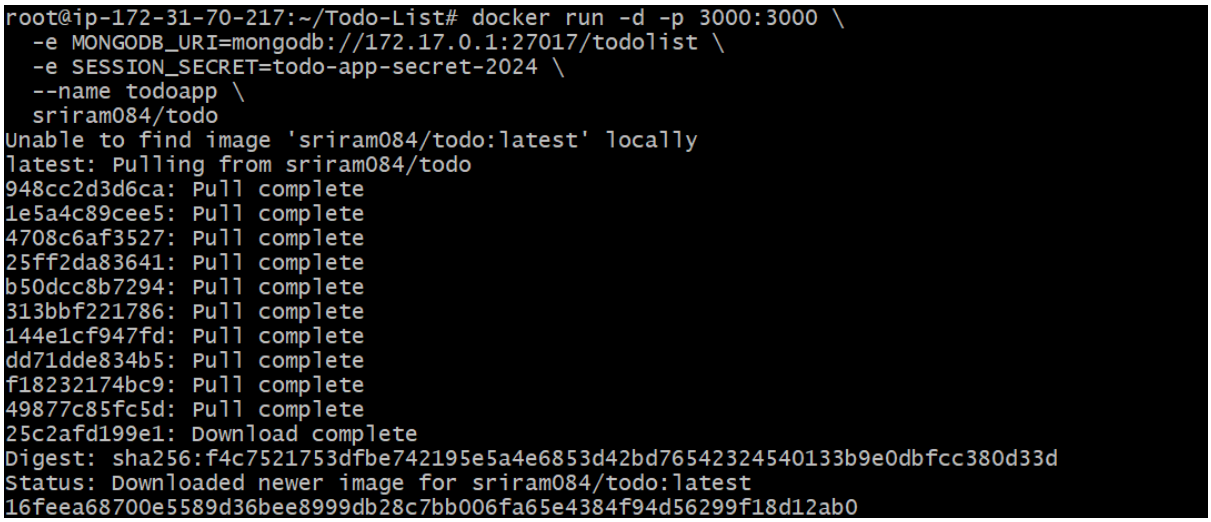
- Multiple layers being downloaded and extracted
- Final confirmation that the image has been successfully retrieved
- The MongoDB container starting in detached mode

Once running, this container provides the database backend required by the TodoPro application. It listens on port **27017**, and the application will connect to it using the internal Docker network.

This validates that your custom MongoDB image is functional and ready to be used in a containerized environment before deploying it in Kubernetes.

Step 9 — Pulling and Running the TodoPro Application Container

Screenshot



Description

This screenshot shows the TodoPro application container being launched using your custom Docker image:

`sriram084/todo:latest`

Since the image was not available locally on the EC2 instance, Docker automatically pulled it from Docker Hub. The output displays:

- Multiple image layers being downloaded and unpacked
- Success message confirming the image has been retrieved
- A new container named **todoapp** starting in detached mode

The container is run with the following environment variables:

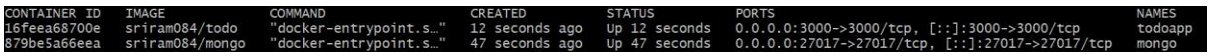
- **MONGODB_URI** — connection string pointing to the MongoDB container
- **SESSION_SECRET** — secure key used for session management inside the Node.js application

Port **3000** is mapped so the application can be accessed publicly from the EC2 instance.

This validates that your custom application image is working correctly and capable of running independently in a containerized environment.

Step 10 — Verifying Running Containers (Application + MongoDB)

Screenshot



Description

This screenshot confirms that both the TodoPro application and MongoDB containers are running successfully on the EC2 instance.

The `docker ps` output shows:

1. TodoPro Application Container

- **Image:** `sriram084/todo:latest`
- **Status:** Running
- **Port Mapping:** `3000 → 3000`
- **Container Name:** `todoapp`

This makes the application accessible via:

`http://<EC2-Public-IP>:3000`

2. MongoDB Container

- **Image:** `sriram084/mongo:latest`
- **Status:** Running
- **Port Mapping:** `27017 → 27017`
- **Container Name:** `mongo`

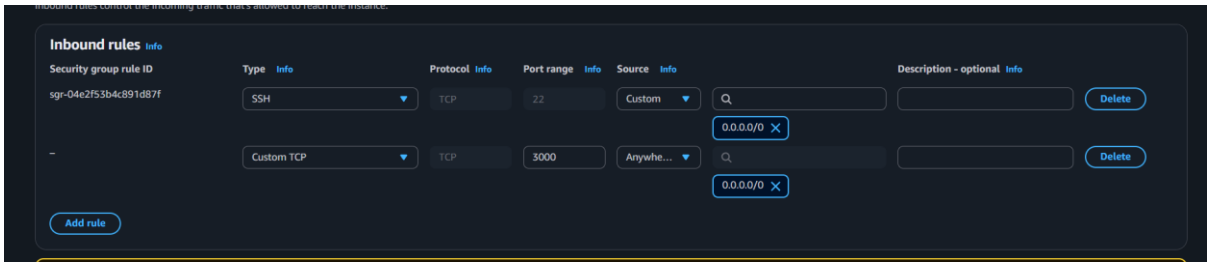
MongoDB serves as the backend database and is reachable by the application through the internal Docker bridge network.

Conclusion

This screenshot verifies that the entire containerized stack is functioning correctly. The database and application containers are running, exposed, and ready to be tested through the browser before moving to Kubernetes deployment.

Step 11 — Configuring Security Group Inbound Rules

Screenshot



Description

This screenshot displays the inbound rules configured in the EC2 instance's security group. These rules control what traffic is allowed to reach the server from the internet.

Two rules are shown:

---

1. SSH Access (Port 22)

- **Type:** SSH
- **Protocol:** TCP
- **Port:** 22
- **Source:** 0.0.0.0/0
  This rule allows users to securely connect to the EC2 instance via SSH. This was required for installing Docker, cloning the repository, and running the application.

---

2. Application Access (Port 3000)

- **Type:** Custom TCP
- **Protocol:** TCP
- **Port:** 3000
- **Source:** 0.0.0.0/0
  This rule enables the TodoPro web application to be accessed publicly using the EC2 instance's public IP:

```
http://<EC2-Public-IP>:3000
```

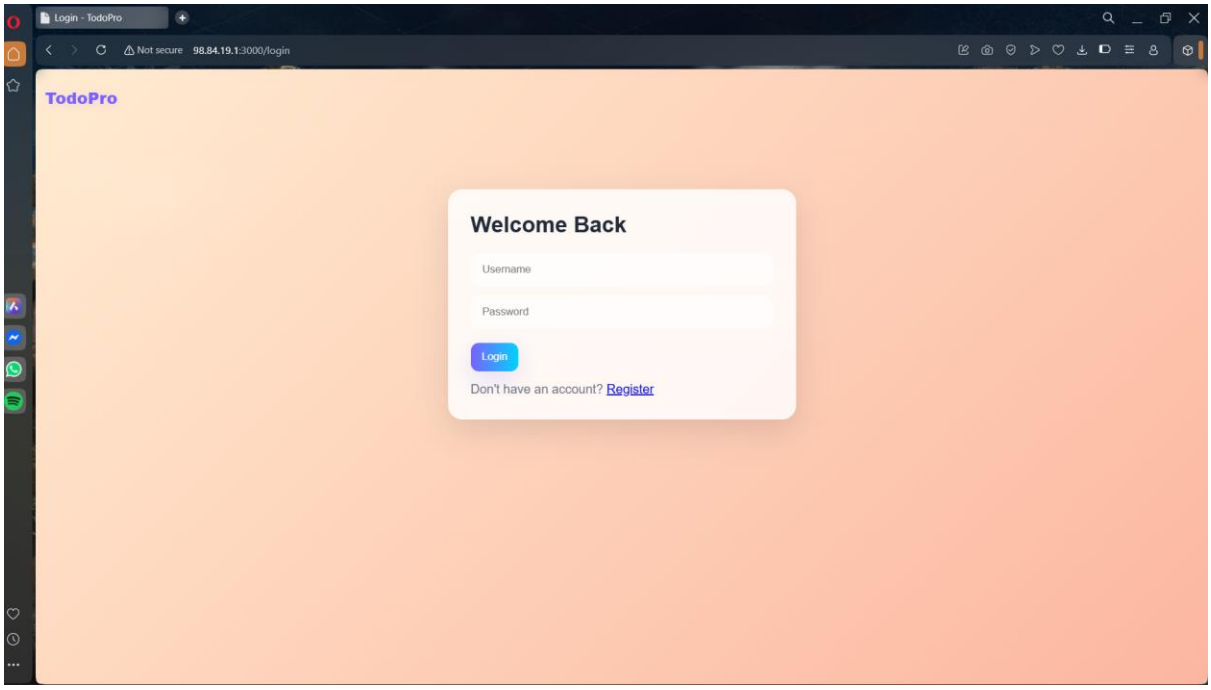
Purpose

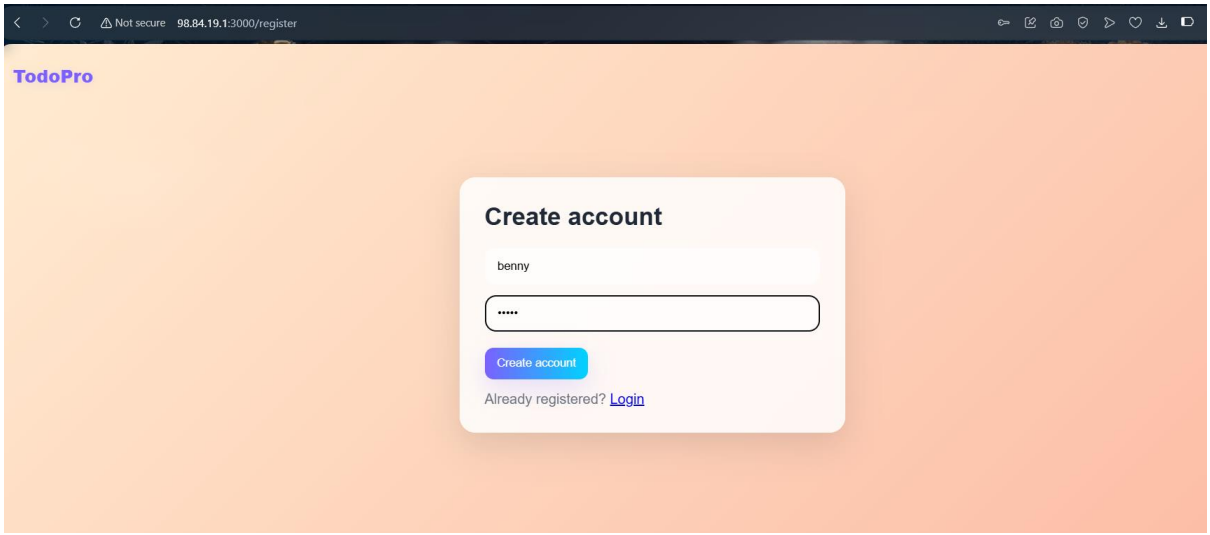
These security group settings ensure:

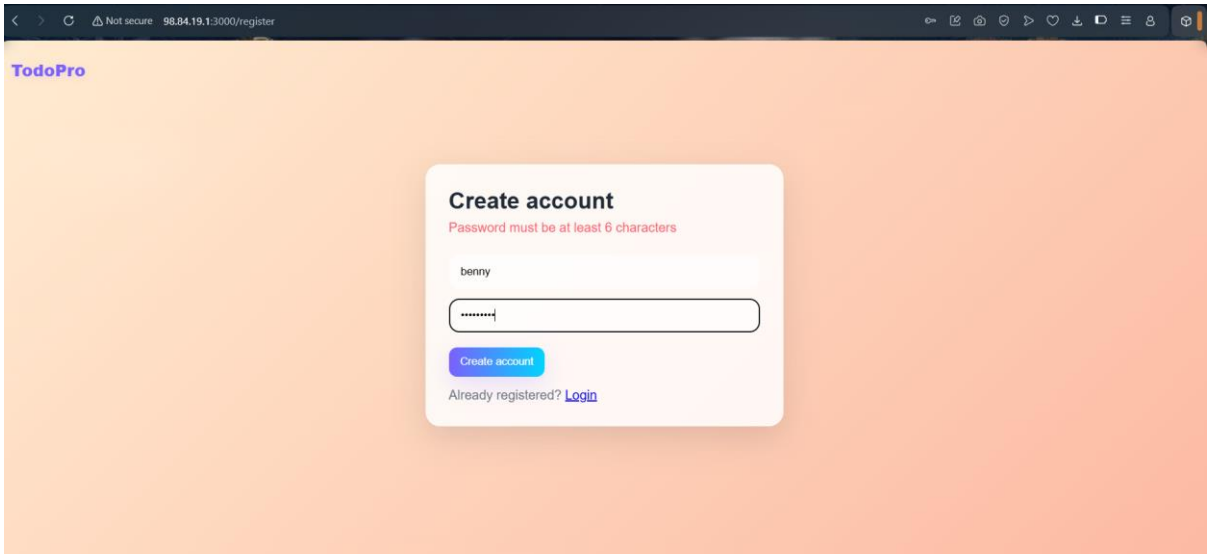
- Administrators can access the instance for configuration and troubleshooting
- End-users can reach the running TodoPro application
- Containers running inside the instance can be tested externally before moving to Kubernetes

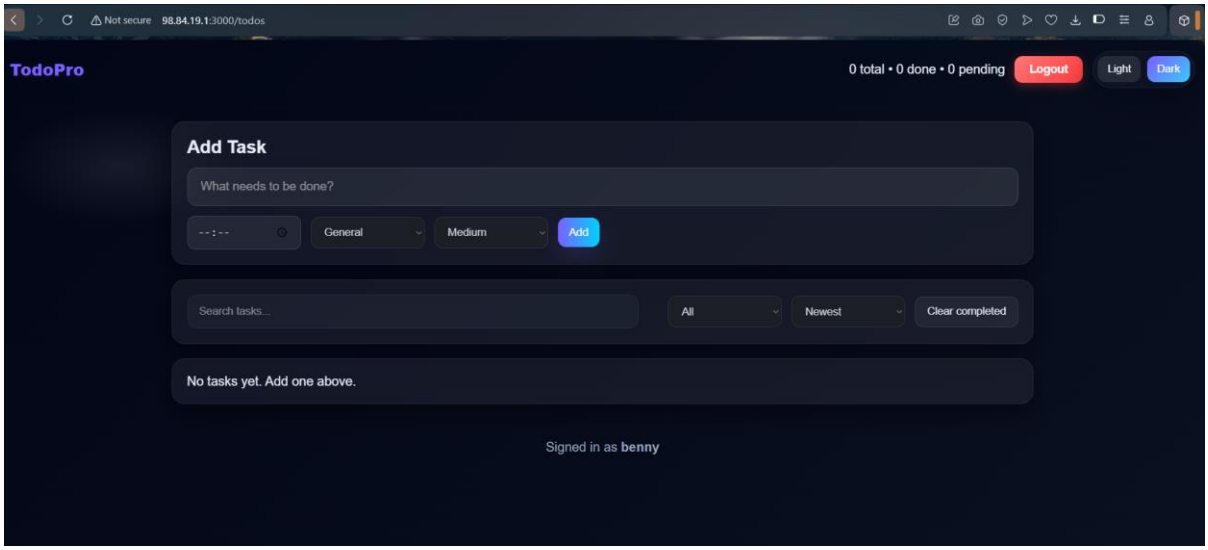
Step 12 — Accessing the TodoPro Application in the Browser

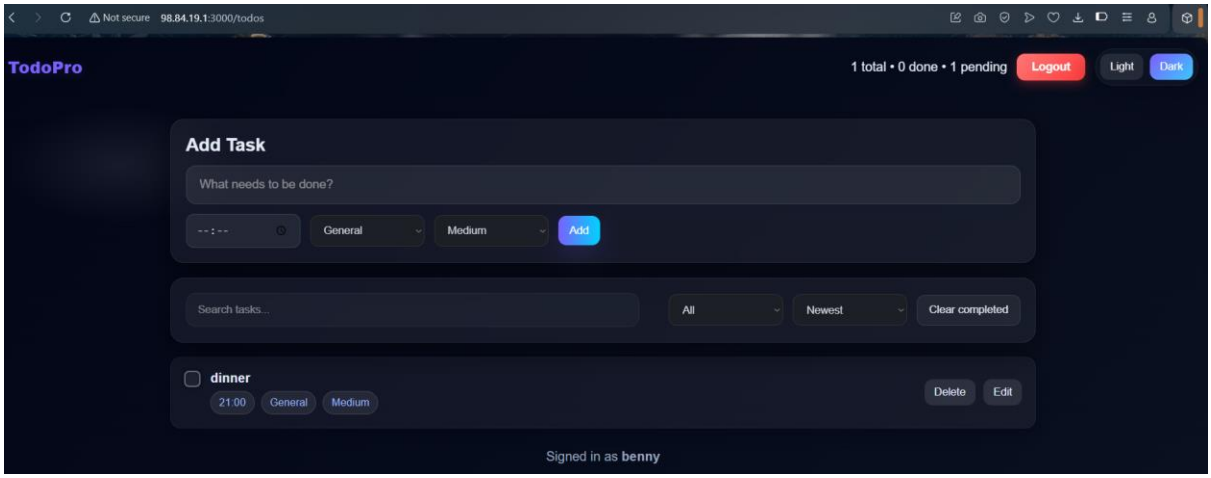
Screenshots






Not secure 98.84.19.1:3000/register
TodoPro
Create account
Password must be at least 6 characters
benny
Create account
Already registered? Login


Not secure 98.84.19.1:3000/todos
TodoPro
0 total • 0 done • 0 pending
Logout
Light
Dark
Add Task
What needs to be done?
--:--
General
Medium
Add
Search tasks...
All
Newest
Clear completed
No tasks yet. Add one above.
Signed in as benny


Not secure 98.84.19.1:3000/todos
TodoPro
1 total • 0 done • 1 pending
Logout
Light
Dark
Add Task
What needs to be done?
--:--
General
Medium
Add
Search tasks...
All
Newest
Clear completed
dinner
21:00
General
Medium
Delete
Edit
Signed in as benny

Description

These screenshots confirm that the TodoPro application, running inside the Docker container on the EC2 instance, is fully functional and accessible through the public IP address of the server.

You opened the application in the browser using:

```
http://98.84.19.1:3000
```

The sequence of screenshots demonstrates the complete workflow:

---

1. Application Login Page

The first screenshot shows the **login interface**, confirming that the frontend UI is loading correctly from the containerized application.

---

2. Create Account Page

You navigated to the **Register** page, where a new user account can be created.
The form fields (username and password) render properly, validating the UI is fully responsive.

---

3. Password Validation Error

The application correctly detects weak passwords and displays a client-side validation message:

```
Password must be at least 6 characters
```

This confirms that validation logic in the Node.js backend and frontend is functioning as expected.

---

4. Successful Login and Dashboard Access

After registering with a valid password and logging in:

- The application redirected you to the **Todos dashboard**
- Session management worked correctly
- The UI switched to the authenticated view
- Theme toggle (Light/Dark mode) appeared
- User status shows: `Signed in as benny`

---

5. Adding and Viewing Tasks

The final screenshots show:

- A task being added successfully (e.g., “dinner”)
- Proper rendering of time, category, and priority
- Working filters and task actions (Edit, Delete)

This verifies that:

- The backend API is running correctly
- MongoDB container is storing data
- The application container is communicating with MongoDB through the environment variables
- The UI updates instantly based on backend responses

Conclusion for Step 12

This step confirms that your **entire Docker-based deployment is working end-to-end**, including:

- API communication
- Database operations
- User authentication
- Task creation and management
- UI rendering and navigation

Your application is now fully validated and ready to move to the next phase

Deploying TodoPro on Amazon EKS (Kubernetes).

Step 1 — Installing AWS CLI and eksctl

Purpose

This step prepares the local machine for interacting with AWS and Amazon EKS.
The AWS CLI is required to authenticate and manage AWS resources, while `eksctl` is used to create and manage EKS clusters in a simplified and automated way.

Both tools are mandatory before proceeding with Kubernetes deployment on EKS.

Tools Installed

- **AWS CLI** – Used to configure AWS credentials and manage AWS services
- **eksctl** – Official CLI tool for creating and managing EKS clusters

---

Result

- AWS CLI installation completed
- eksctl installation completed
- System is ready for AWS authentication and EKS cluster creation

Step 2 — Creating an IAM User for AWS CLI and EKS Access

Purpose

This step involves creating an IAM user that will be used for programmatic access to AWS through the AWS CLI.
The IAM user provides the required permissions to create and manage Amazon EKS clusters and related AWS resources.

Using a dedicated IAM user ensures secure, controlled access when interacting with AWS services from the local machine.

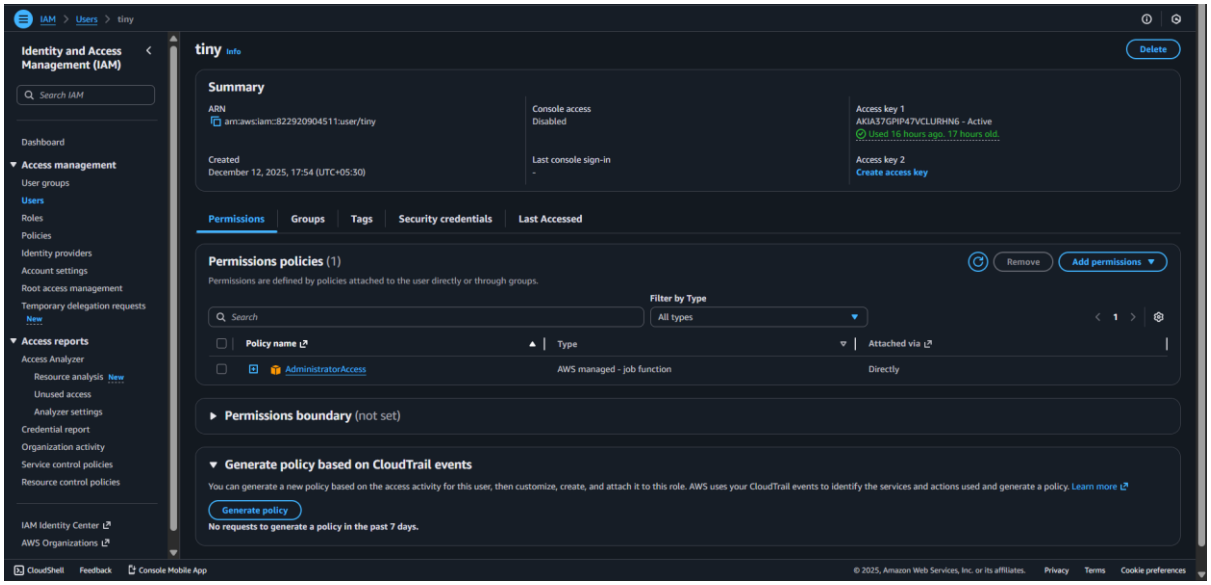
IAM User Configuration

The IAM user is configured with:

- Programmatic access enabled
- Access keys generated for AWS CLI usage
- Administrator-level permissions attached to support EKS operations

These permissions are required during cluster creation, node provisioning, and networking setup.

Description of the Screenshot



The screenshot displays:

- IAM user summary information
- Active access key available for programmatic access
- `AdministratorAccess` policy attached directly to the user

This confirms that the IAM user is fully authorized to perform EKS and Kubernetes-related operations.

Result

- IAM user created successfully
- Access key generated and active
- Required permissions attached for EKS management

Step 3 — Configuring AWS CLI with IAM Credentials

Purpose

This step configures the AWS CLI on the local machine using the IAM user credentials created in the previous step.
Configuring AWS CLI ensures that all AWS-related commands (including EKS cluster creation) are executed under the correct AWS account and region.

This configuration is mandatory before using `eksctl` or `kubectl` with Amazon EKS.

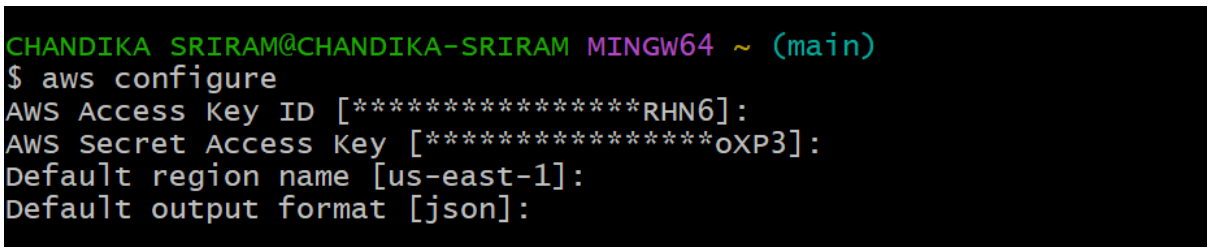
AWS CLI Configuration Details

During this step, the following details are provided:

- **AWS Access Key ID** — IAM user access key
- **AWS Secret Access Key** — IAM user secret key
- **Default Region** — `us-east-1`
- **Default Output Format** — `json`

These values allow the AWS CLI to authenticate and communicate securely with AWS services.

Description of the Screenshot



The screenshot shows:

- Execution of the `aws configure` command
- Entry of Access Key ID and Secret Access Key
- Region set to `us-east-1`
- Output format set to `json`

This confirms that AWS CLI is successfully configured and ready for use.

Result

- AWS CLI authentication completed successfully
- AWS account and region linked to the local environment
- System is now ready to create and manage Amazon EKS clusters

Step 4 — Generating an SSH Key Pair for EKS Node Access

Purpose

This step involves generating an SSH key pair that will be used to securely access the EC2 worker nodes created as part of the Amazon EKS cluster.

The SSH key enables administrators to log in to worker nodes for troubleshooting, log inspection, and low-level diagnostics when required.

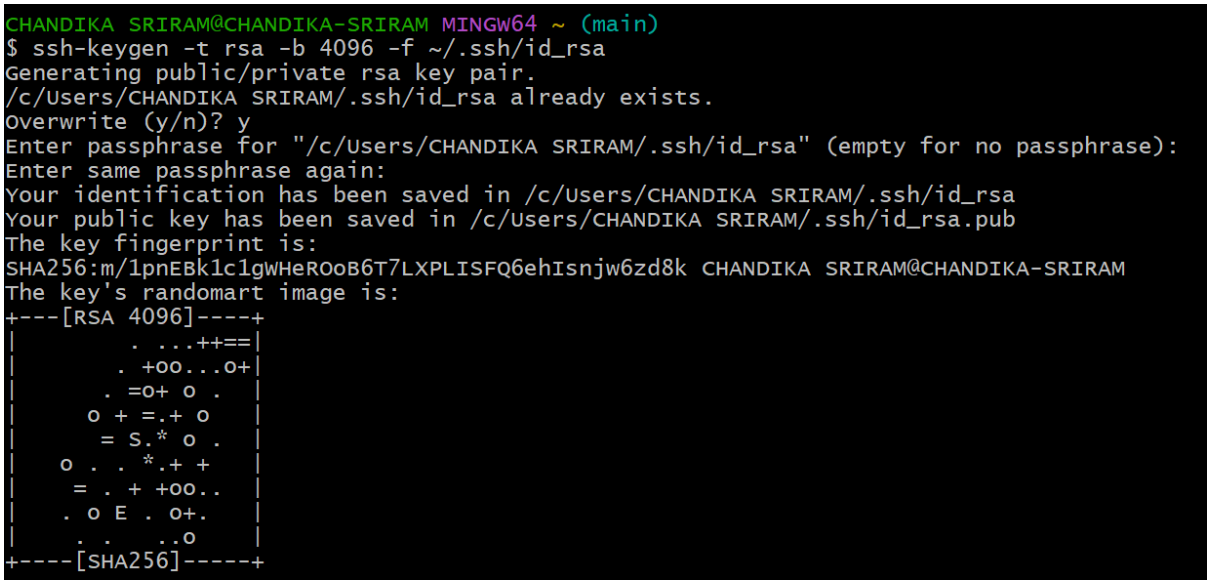
SSH Key Generation

An RSA-based SSH key pair is generated with the following characteristics:

- Key type: RSA
- Key length: 4096 bits
- Storage location: `~/.ssh/id_rsa`

If a key already exists, it is overwritten to ensure consistency during cluster creation.

Description of the Screenshot



The screenshot shows:

- Execution of the SSH key generation command
- Confirmation of private and public key creation
- Storage paths for both keys
- Generated fingerprint and randomart image

This confirms that the SSH key pair has been successfully created and is ready to be used during EKS node group configuration.

Result

- SSH private and public keys generated successfully
- Key available for secure EC2 worker node access
- Environment prepared for EKS cluster node provisioning

Step 5 — Creating the Amazon EKS Cluster Using eksctl

Purpose

This step creates an Amazon Elastic Kubernetes Service (EKS) cluster that will host the TodoPro application and MongoDB workloads.

The EKS cluster provides a managed Kubernetes control plane along with EC2-based worker nodes, enabling scalable and production-ready container orchestration.

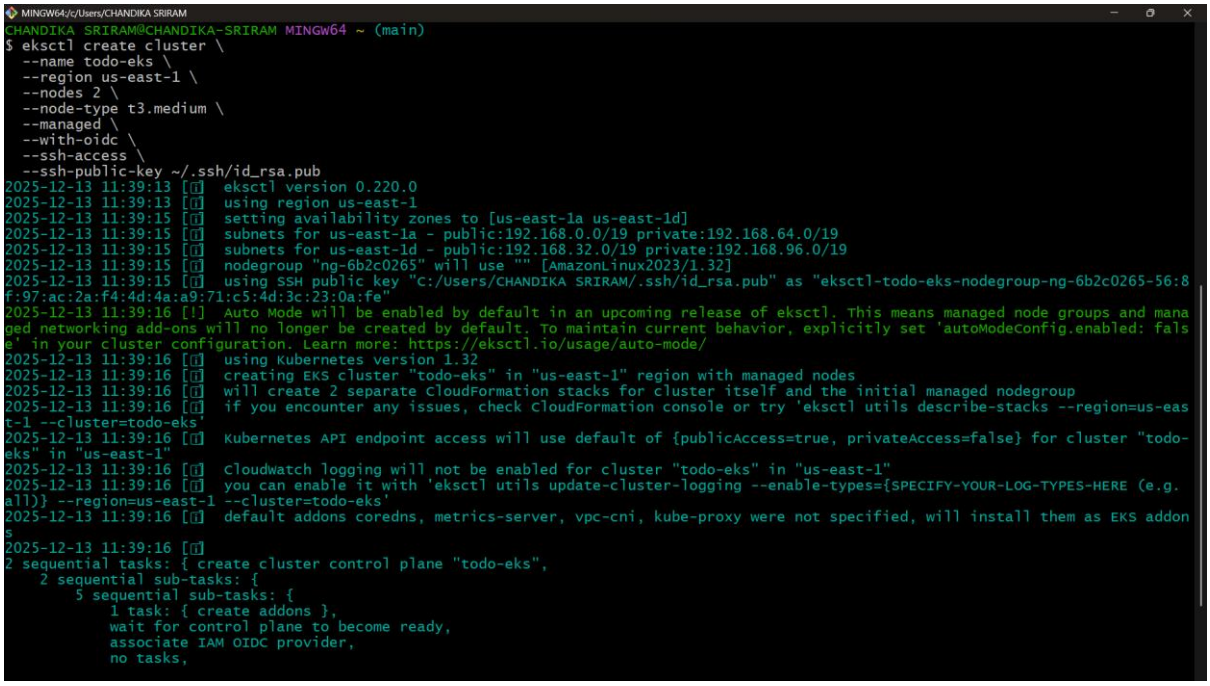
EKS Cluster Configuration

The cluster is created with the following characteristics:

- **Cluster name:** `todo-eks`
- **Region:** `us-east-1`
- **Node group type:** Managed node group
- **Number of worker nodes:** 2
- **Instance type:** `t3.medium`
- **Kubernetes version:** 1.32
- **SSH access enabled:** Yes
- **SSH key:** Local public key (`~/.ssh/id_rsa.pub`)
- **OIDC provider:** Enabled (required for IAM roles for service accounts)

This configuration ensures sufficient compute capacity for application and database pods while maintaining ease of management.

Description of the Screenshot



The screenshot shows:

- Execution of the `eksctl create cluster` command
- Region and availability zone selection
- Automatic VPC and subnet creation
- Managed node group provisioning
- Association of SSH public key with worker nodes
- Cluster control plane creation in progress

The log output confirms that CloudFormation stacks are being created for both the EKS control plane and the managed node group.

Result

- Amazon EKS cluster creation initiated successfully
- Managed node group provisioning started
- Kubernetes control plane setup in progress

Once completed, the cluster becomes ready to accept Kubernetes workloads.

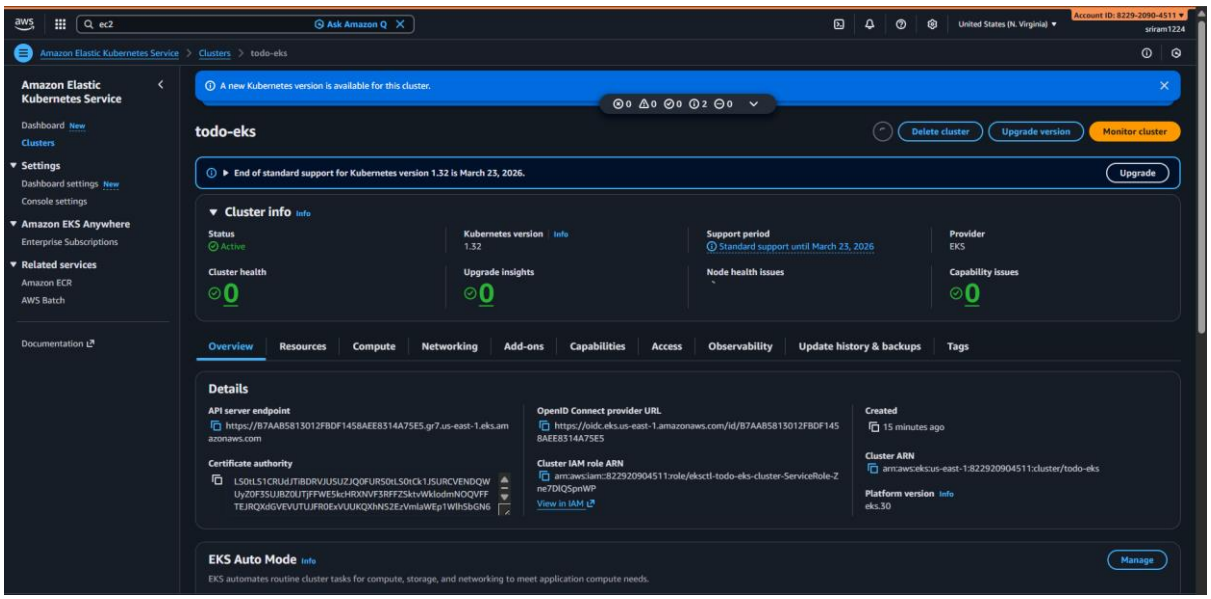
Verifying EKS Cluster Creation in AWS Console

Purpose

This step verifies that the Amazon EKS cluster has been successfully created and is in a healthy, active state before deploying workloads.

Verification through the AWS Management Console ensures that the control plane, networking, and cluster components are fully operational.

Description of the Screenshot



The screenshot shows:

- EKS cluster named **`todo-eks`**
- Cluster status marked as **Active**
- Kubernetes version **1.32**
- No cluster health issues reported
- Managed control plane successfully provisioned
- OpenID Connect (OIDC) provider enabled
- Cluster ARN and API server endpoint available

This confirms that the EKS cluster has been created successfully and is ready to accept Kubernetes deployments.

Step 6 — Verifying Worker Nodes and kubectl Connectivity

Purpose

This step verifies that the EKS worker nodes have successfully joined the cluster and that `kubectl` is correctly configured to communicate with the EKS control plane.

A successful node validation confirms that the cluster is ready to deploy application workloads.

Worker Node Validation

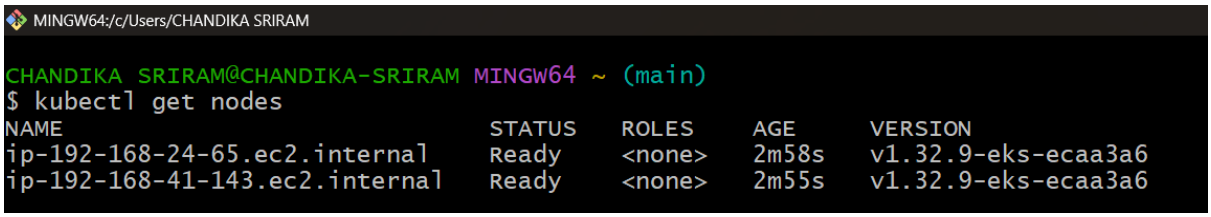
The Kubernetes command-line tool (`kubectl`) is used to list all nodes registered with the cluster.

The output confirms:

- Multiple worker nodes are available
- All nodes are in **Ready** state
- Nodes are running the expected EKS Kubernetes version

This ensures the managed node group is functioning as expected.

Description of the Screenshot



The screenshot shows:

- Execution of `kubectl get nodes`
- Two worker nodes listed with internal EC2 hostnames
- Node status marked as **Ready**
- Kubernetes version `v1.32.x-eks`

This confirms that `kubectl` is correctly connected to the EKS cluster and the worker nodes are healthy.

Step 7 — Creating Kubernetes Configuration Files

Purpose

This step focuses on creating Kubernetes configuration files required to securely manage application configuration and sensitive data for the TodoPro application.

Instead of hardcoding values inside the application or Docker image, Kubernetes-native objects such as **Secrets** and **ConfigMaps** are used. This follows Kubernetes best practices for security and configuration management.

Why This Step Is Required

- Keeps sensitive values out of source code
- Allows configuration changes without rebuilding images
- Enables clean separation between application code and environment-specific data
- Improves security and maintainability

Step 7.1 — Creating Kubernetes Secret File

File Name

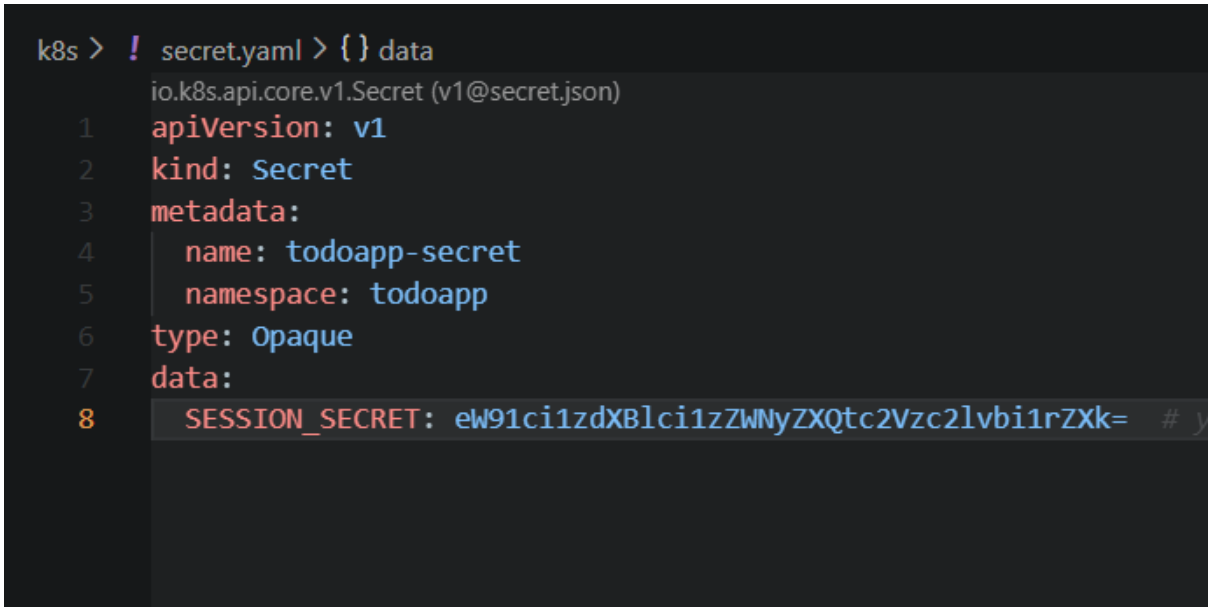
`k8s/secret.yaml`

Purpose

This Kubernetes Secret stores sensitive information required by the TodoPro application, specifically the session secret used for authentication and session management.

The secret is created as an **Opaque** type, which is suitable for arbitrary key–value pairs.

Description of the Screenshot



The screenshot shows:

- A Kubernetes Secret definition using `apiVersion: v1`
- Secret name set to `todoapp-secret`
- Namespace specified as `todoapp`
- Secret type defined as `Opaque`
- Base64-encoded value for `SESSION_SECRET`

This confirms that sensitive data is securely stored in Kubernetes and can be injected into application pods as environment variables.

Key Points

- The value of `SESSION_SECRET` is Base64 encoded, as required by Kubernetes
- The secret will later be referenced in the application Deployment manifest
- Namespace-level scoping ensures isolation from other workloads

Step 7.2: MongoDB Deployment Configuration

File: `k8s/mongo-deployment.yaml`

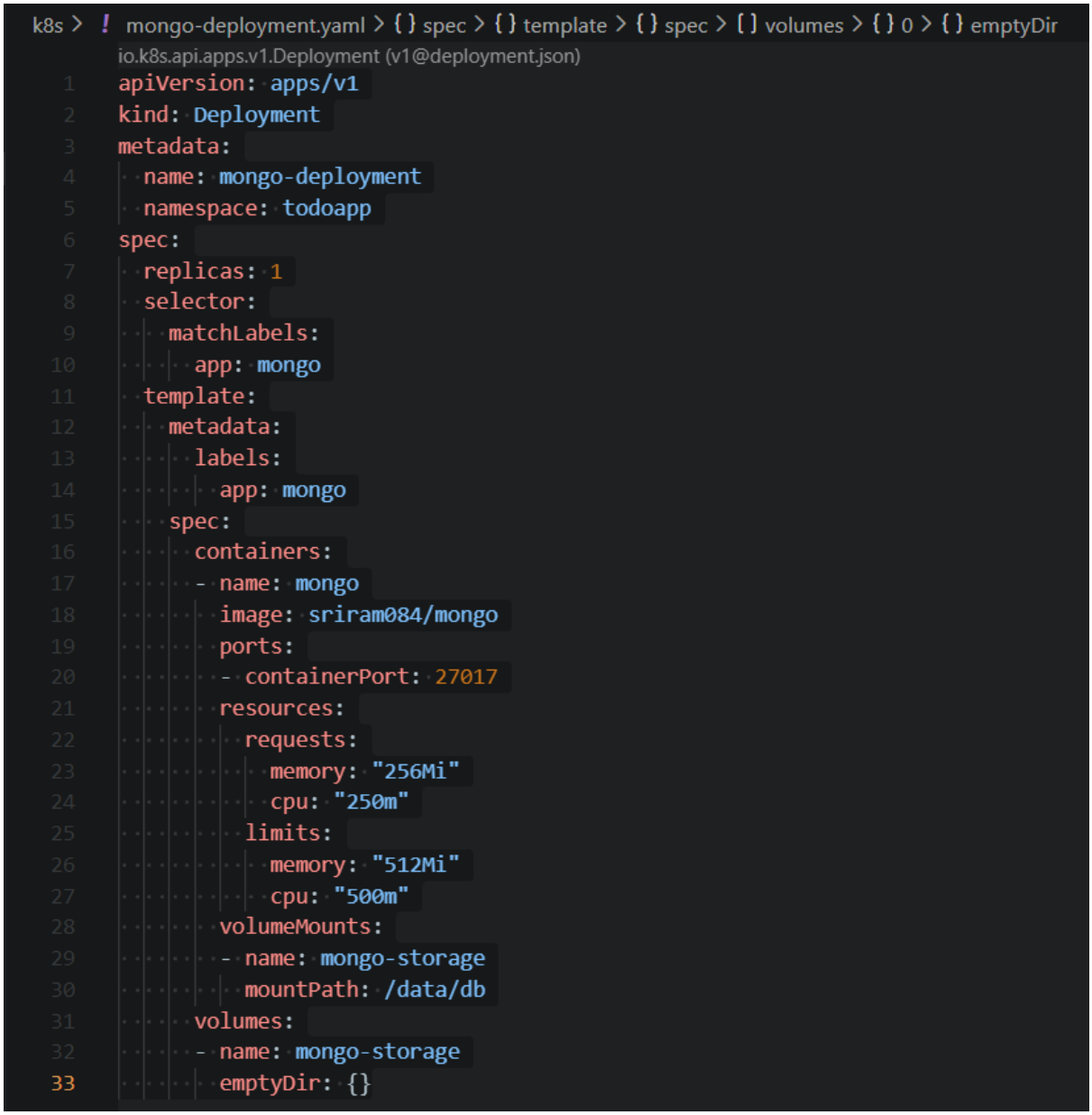
Description:
This file defines a **Kubernetes Deployment** for MongoDB within the `todoapp` namespace. Key points covered in this configuration:

- Deploys **one replica** of MongoDB for the application.
- Uses the **custom Docker image** `sriram084/mongo`.

- Exposes MongoDB on **port 27017** inside the cluster.
- Sets **CPU and memory requests/limits** to control resource usage.
- Mounts an `emptyDir` volume at `/data/db` to store MongoDB data during the pod lifecycle.

This deployment ensures MongoDB runs as a managed pod inside the Kubernetes cluster.

Screenshot:



Step 7.3: MongoDB Service Configuration

File: `k8s/mongo-service.yaml`

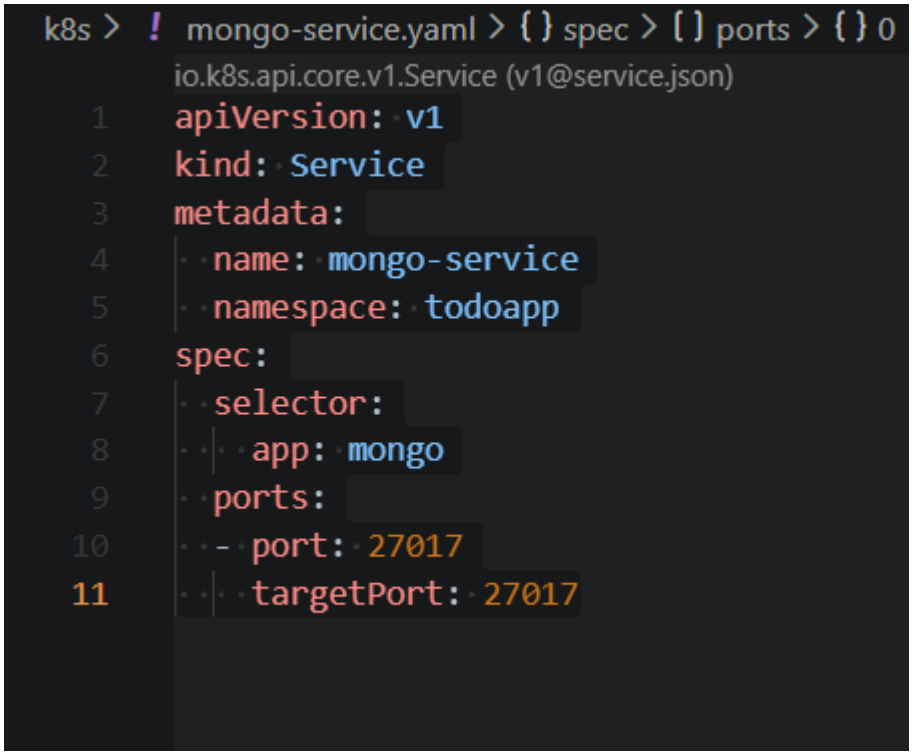
Description:
This file creates a **ClusterIP Service** for MongoDB.
The service enables internal communication between the Todo application and MongoDB using a stable DNS name.

Key characteristics:

- Selects MongoDB pods using the label `app: mongo`
- Exposes port **27017**
- Service is accessible **only within the cluster**, ensuring database security

This service allows the Todo application to connect to MongoDB using `mongo-service:27017`.

Screenshot:



Step 7.4: Namespace Definition

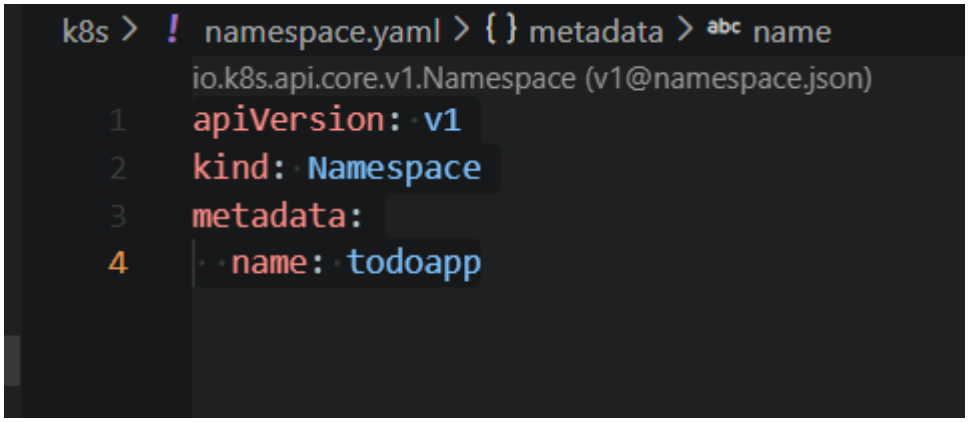
File: `k8s/namespace.yaml`

Description:
This file defines a dedicated Kubernetes namespace named **`todoapp`**.
All application-related resources (Deployments, Services, ConfigMaps, Secrets) are deployed inside this namespace.

Using a namespace:

- Improves resource isolation
- Simplifies management and cleanup
- Follows Kubernetes best practices for multi-component applications

Screenshot:



Step 7.5: Kubernetes Secret Configuration

File: `k8s/secret.yaml`

Description:
This file defines a Kubernetes **Secret** used to store sensitive application data securely.

Details:

- Secret name: `todoapp-secret`
- Stored in the `todoapp` namespace
- Stores `SESSION_SECRET` in **base64-encoded format**
- Type is `Opaque`, suitable for generic secrets

This secret is later injected into the Todo application pod as an environment variable.

Screenshot:

Step 7.6: Todo Application Deployment Configuration

File: `k8s/todoapp-deployment.yaml`

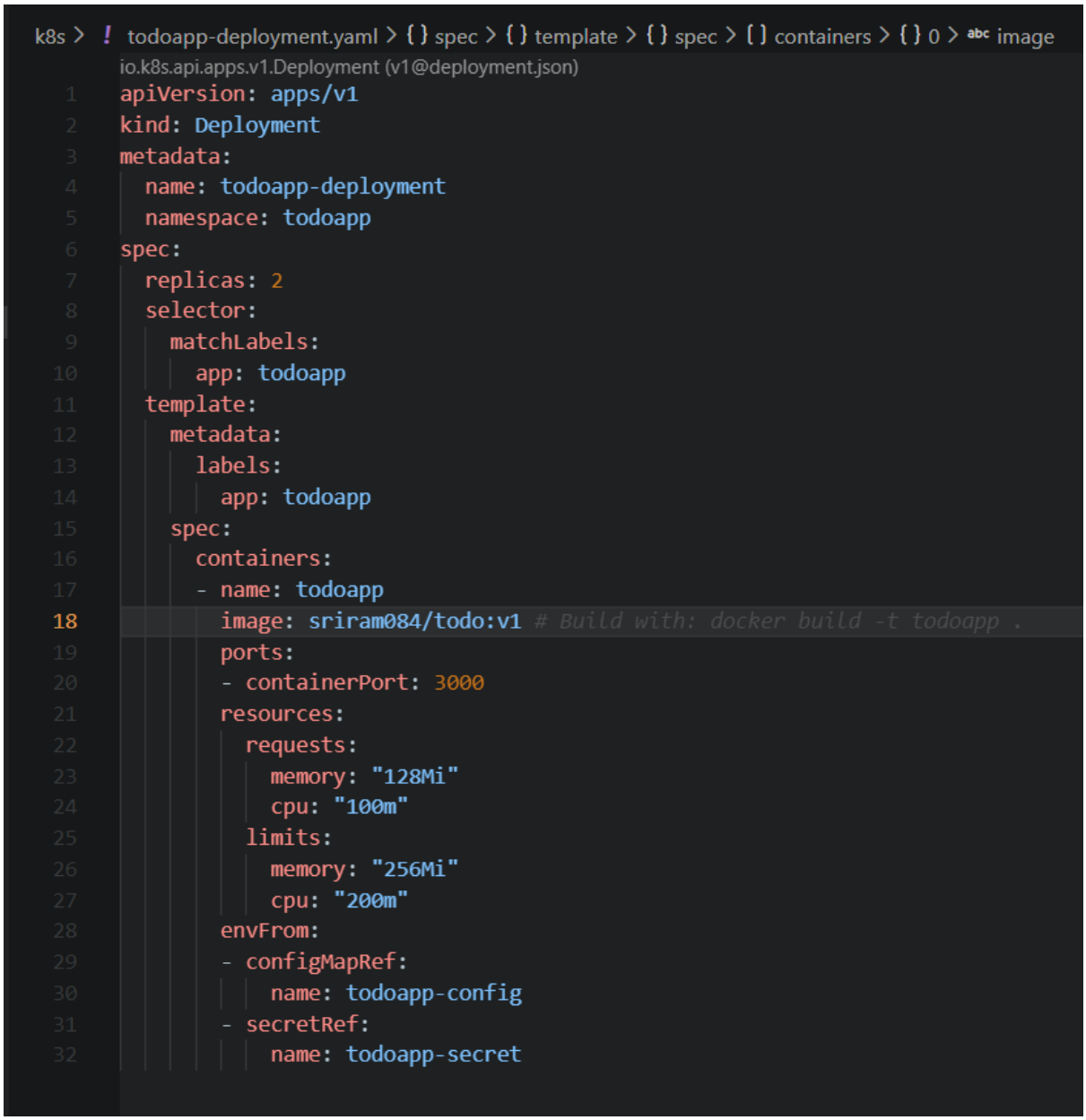
Description:
This file defines the Kubernetes Deployment for the **Todo application backend**.

Key highlights:

- Deploys **2 replicas** for high availability
- Uses the Docker image `sriram084/todo:v1`
- Exposes application on **container port 3000**
- Defines CPU and memory requests/limits
- Injects environment variables using:
    - `ConfigMap` (`todoapp-config`)
    - `Secret` (`todoapp-secret`)

This deployment ensures the application is scalable, secure, and production-ready.

Screenshot:



Step 7.7: Todo Application Service (LoadBalancer)

File: `k8s/todoapp-service.yaml`

Description:
This file creates a **LoadBalancer Service** to expose the Todo application to the internet.

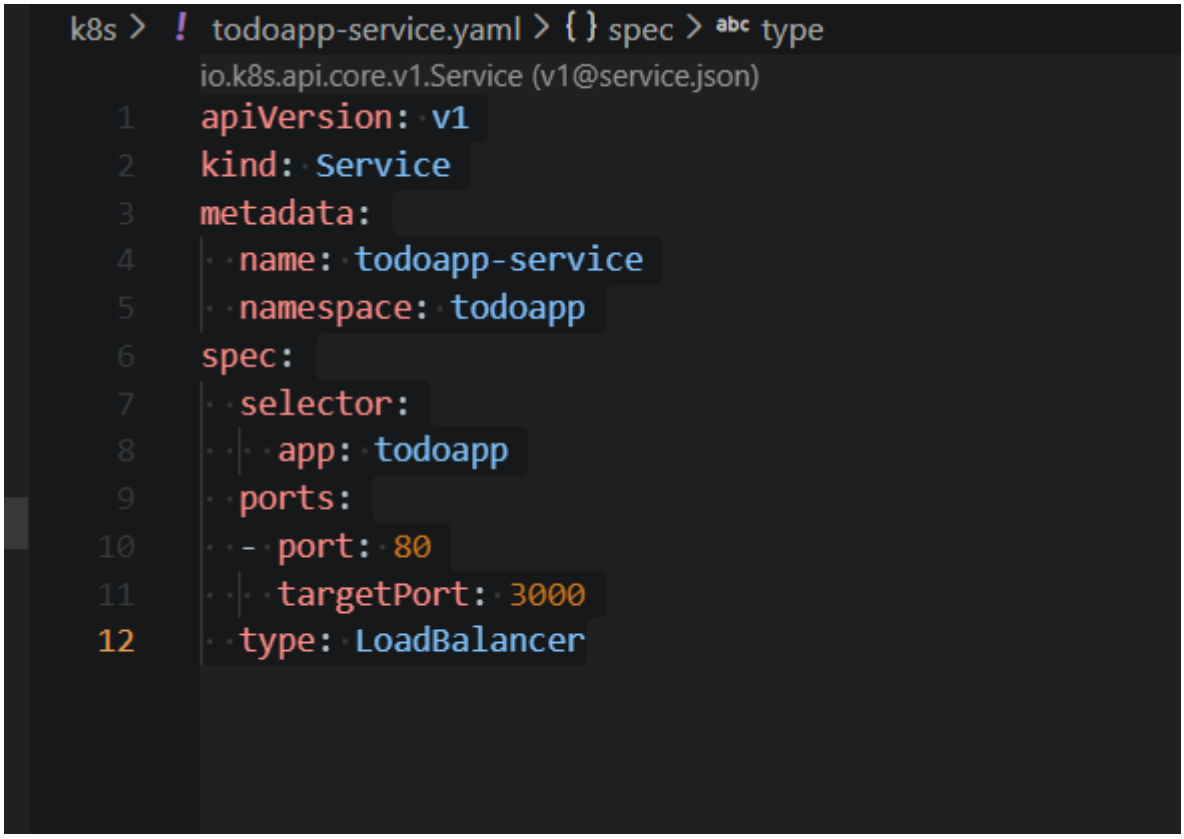
Service details:

- Type: `LoadBalancer`

- Maps port **80** externally to container port **3000**
- Automatically provisions an AWS Elastic Load Balancer (ELB)
- Makes the application accessible via a public URL

This step enables end users to access the Todo application from a browser.

Screenshot:



Step 8: Automating Kubernetes Deployment Using `deploy.sh`

File: `k8s/deploy.sh`

Description:
Instead of manually applying each Kubernetes manifest one by one, a **deployment shell script** was created to automate the entire process. This script ensures the resources are deployed in the correct order and reduces human error during deployment.

What this script does, in order:

- Prints a message indicating the start of the deployment process.
- Applies all core Kubernetes resources:
  - Namespace
  - ConfigMap

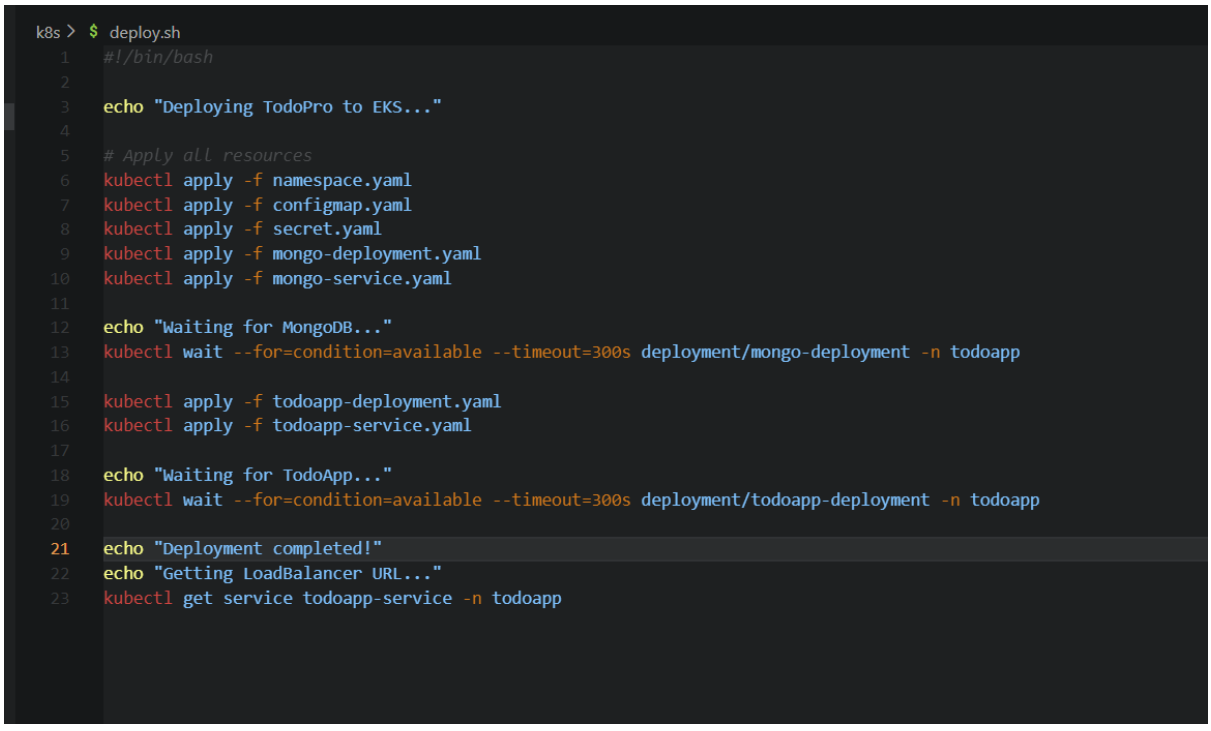
    - Secret
    - MongoDB Deployment
    - MongoDB Service
- Waits until the MongoDB deployment is **fully available** before continuing.
- Deploys the Todo application:
    - Todo application Deployment
    - Todo application Service
- Waits for the Todo application pods to become **available and ready**.
- Prints a success message once deployment is completed.
- Fetches and displays the **LoadBalancer service details**, which includes the external URL used to access the application.

This approach:

- Makes the deployment **repeatable and consistent**
- Saves time
- Follows real-world DevOps automation practices
- Keeps the workflow clean and professional

In short, instead of babysitting `kubectl apply` commands, you let a script do the boring work. As it should be.

Screenshot:



Step 9: Executing the Kubernetes Deployment Script

File Executed: `k8s/deploy.sh`

Description:
In this step, the previously created deployment script is executed to deploy the entire TodoPro application stack onto the EKS cluster.

By running a single script, all Kubernetes resources are provisioned automatically in the correct sequence. This eliminates manual intervention and ensures a consistent, repeatable deployment process.

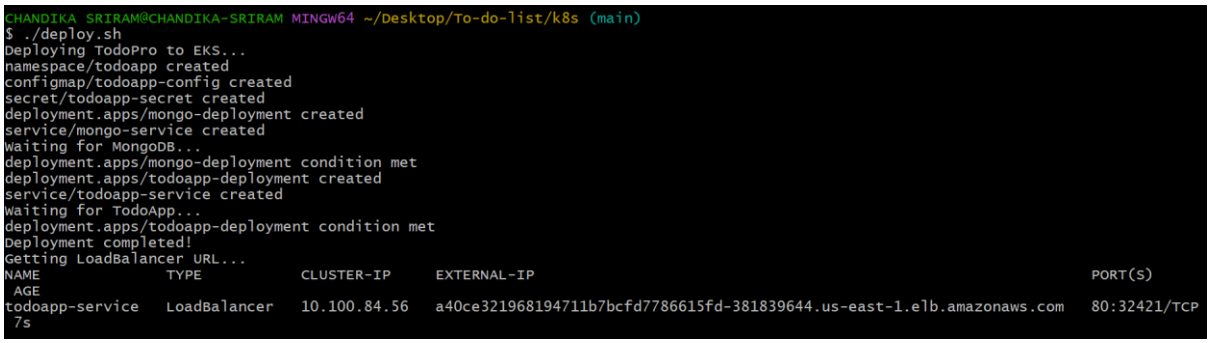
The script performs the following actions during execution:

- Creates the **todoapp namespace**
- Applies the **ConfigMap** and **Secret** required by the application
- Deploys **MongoDB** and exposes it internally using a ClusterIP service
- Waits until the MongoDB deployment is fully available
- Deploys the **Todo application** with multiple replicas
- Exposes the Todo application using a **LoadBalancer service**
- Waits until the Todo application pods are available
- Retrieves and displays the LoadBalancer service details

This step confirms that:

- Kubernetes manifests are valid
- Resource dependencies are respected
- The cluster is capable of running the application workloads successfully

Screenshot:



Step 10: Verifying LoadBalancer Creation and External Access

Description:
After the Kubernetes deployment is completed, the TodoPro application is exposed externally using a **Service of type LoadBalancer**.
In this step, the AWS Console is used to verify that Kubernetes has successfully provisioned an AWS Elastic Load Balancer (ELB) and that it is correctly linked to the EKS worker nodes.

When a LoadBalancer service is created in EKS:

- AWS automatically provisions a **Classic Load Balancer**
- The LoadBalancer is configured as **internet-facing**

- Traffic on port **80** is forwarded to the application running on the worker nodes
- The DNS name generated by AWS becomes the public entry point for the application

This confirms proper integration between Kubernetes and AWS infrastructure.

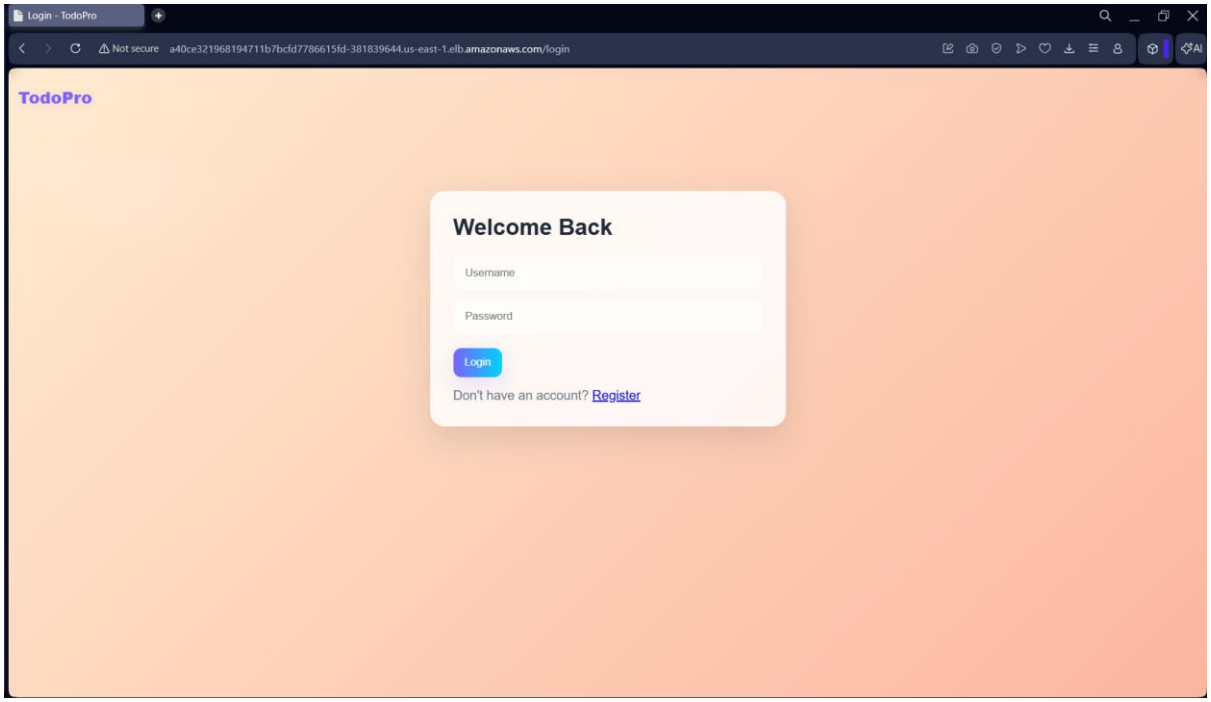
LoadBalancer Validation

The AWS EC2 → Load Balancers console shows:

- A newly created **Classic Load Balancer**
- Status indicating active instances registered
- Availability across multiple subnets and Availability Zones
- A public **DNS name** assigned to the LoadBalancer

This DNS name is used to access the TodoPro application from a browser.

Description of the Screenshot



The screenshot displays:

- LoadBalancer type: **Classic**
- Scheme: **Internet-facing**
- Listener configured on **TCP port 80**
- Backend traffic forwarded to the worker node port
- Public DNS name ending with `elb.amazonaws.com`

This confirms that the Kubernetes service has successfully created and configured the AWS LoadBalancer.

Step 11: Accessing the TodoPro Application via LoadBalancer URL

Description:
In this step, the TodoPro application is accessed externally using the **public DNS name** generated by the AWS LoadBalancer.

After the LoadBalancer is provisioned, Kubernetes exposes the application over the internet. By appending the appropriate route (`/login`) to the LoadBalancer DNS name, the application's user interface becomes accessible through a web browser.

This step validates the **end-to-end deployment**, confirming that:

- The Kubernetes services are correctly configured
- Traffic from the internet reaches the EKS LoadBalancer
- Requests are forwarded to the Todo application pods
- The application is running and responding as expected

Application Validation

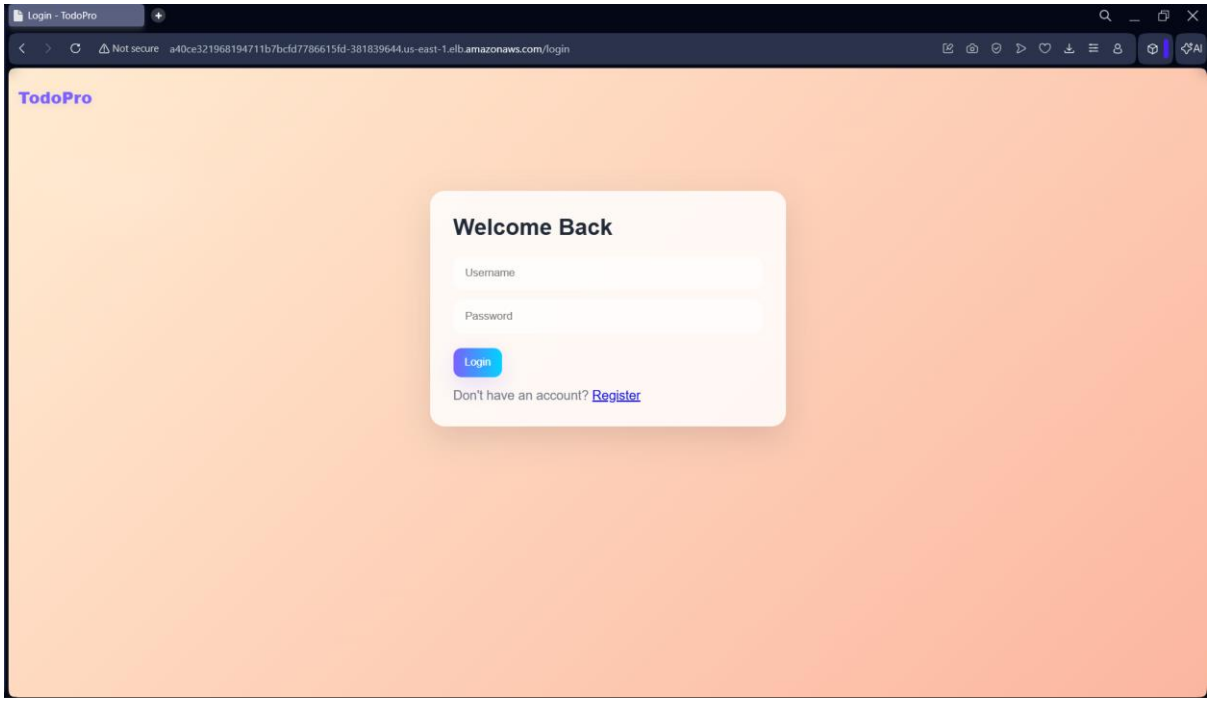
The application interface displays:

- TodoPro login page
- User authentication options (Login / Register)
- Fully rendered frontend UI served from the EKS cluster

This confirms successful integration between:

- AWS LoadBalancer
- EKS cluster
- Kubernetes services and deployments
- Application containers

Description of the Screenshot



The screenshot shows:

- Browser accessing the application using the LoadBalancer DNS URL
- TodoPro login page rendered successfully
- No connectivity or routing errors

This demonstrates that the application is publicly accessible and fully operational.

Step 12: Creating Users and Managing Tasks in TodoPro

Description:
In this step, the functional validation of the TodoPro application is completed by creating multiple users and adding tasks for each user. This confirms that the application is not only accessible but also fully operational with backend persistence and user-specific data handling.

Two different users are registered using the application's **Register** feature. Each user logs in independently and creates tasks, validating authentication, authorization, and database integration.

User Creation Validation

The following actions are performed:

- Register **User 1** using the registration page
- Register **User 2** using a separate user account
- Verify successful login for both users
- Confirm session handling and user isolation

This ensures that the authentication flow and session management are functioning correctly.

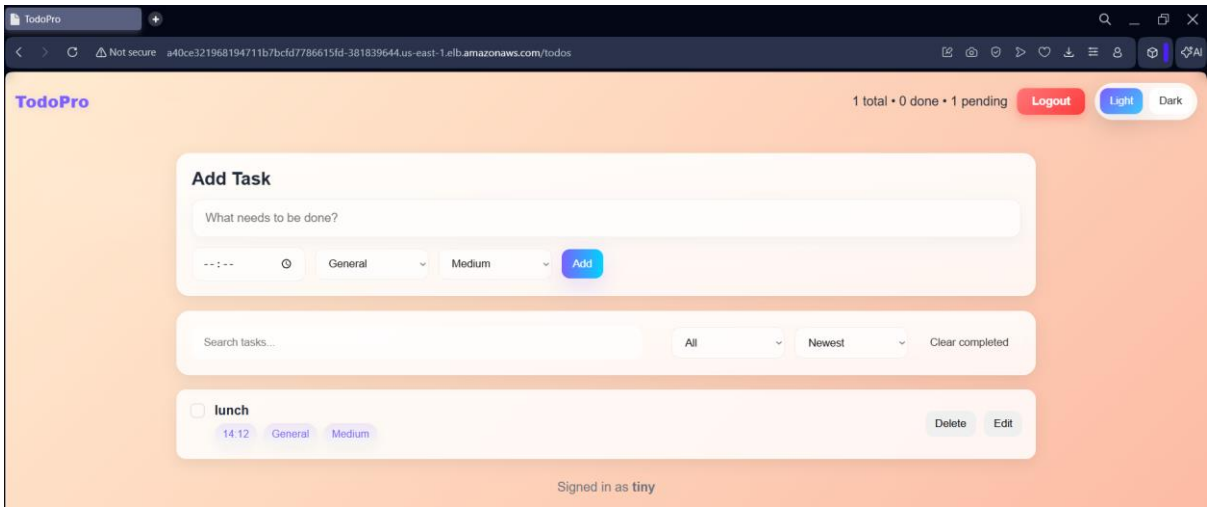
Task Management Validation

For each user:

- Multiple tasks are added with different attributes (task name, category, priority, time)
- Tasks are displayed correctly after creation
- Task counters update dynamically (total, pending, completed)
- Tasks remain user-specific and are not shared across accounts

This validates:

- MongoDB database connectivity
- Proper use of environment variables and secrets
- Correct backend-to-database communication



1: User Validation – User `tiny`

Description:
In this step, the application is validated by logging in as the user **`tiny`** and performing task management operations. This confirms that the user authentication flow, session handling, and task persistence are functioning correctly for individual users in the Kubernetes-deployed environment.

User Login Verification

- The user **`tiny`** successfully logs in using the TodoPro login page.
- The dashboard loads correctly after authentication.
- The username is displayed at the bottom of the dashboard, confirming an active session.

Task Creation and Display

- A task named **"lunch"** is added using the **Add Task** section.
- Task attributes such as:
  - Time
  - Category (**General**)
  - Priority (**Medium**)

  are saved and displayed correctly.
- The task counter updates accurately, showing:
  - **1 total**
  - **0 done**
  - **1 pending**

UI and Feature Validation

- Task list renders correctly for the logged-in user.
- Edit and Delete actions are available for the task.
- Search, filter, and theme toggle options are visible and functional.
- The data shown is specific to user **`tiny`**, confirming proper user isolation.

Step 13: Verifying MongoDB Data Inside the Kubernetes Cluster

Description:
In this step, the MongoDB database running inside the Kubernetes cluster is accessed directly to verify that application data is being stored correctly. This is done by executing an interactive shell session inside the MongoDB pod and connecting to the database using `mongosh`.

This step validates backend persistence and confirms that user and task data created through the TodoPro application is successfully written to MongoDB.

MongoDB Pod Access

- An interactive session is opened into the MongoDB deployment running in the `todoapp` namespace.
- The MongoDB shell (`mongosh`) is launched from inside the container.
- The connection is established locally within the pod on port **27017**.

Database Connection Verification

- MongoDB server version and shell version are displayed, confirming a successful connection.
- The database is accessible without connection errors.
- Startup logs confirm that the MongoDB instance is running and operational inside Kubernetes.

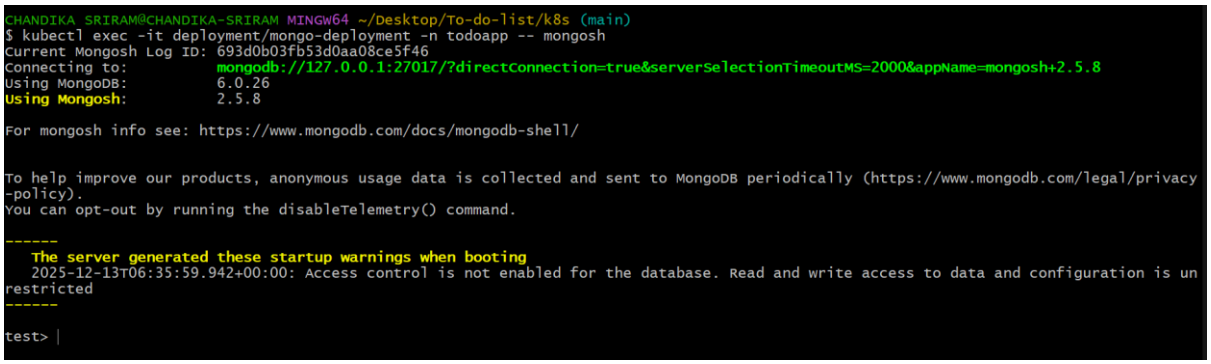
Data Persistence Validation

Using the MongoDB shell, the following can be verified:

- Presence of application databases
- User records created via the TodoPro UI
- Task documents corresponding to each user

This confirms:

- Proper service-to-database communication
- Correct use of Kubernetes Services and networking
- Successful persistence of application data inside the cluster



Security Note

A startup warning indicates that access control is not enabled for MongoDB.
This is acceptable for development and demo environments.

For production deployments, MongoDB authentication and credentials should be enabled using Kubernetes Secrets.

Step 14: Verifying User Data Stored in MongoDB

Description:
In this step, the MongoDB database is queried directly to verify that user data created through the TodoPro application is successfully stored inside the Kubernetes-managed database. This confirms end-to-end data persistence from the application UI to the backend database.

Database Selection

- The active MongoDB shell session switches to the application database:
  - **Database name:** `todolist`
- This database is used by the TodoPro backend to store users and tasks.

User Collection Validation

- A query is executed on the `users` collection to retrieve all registered users.
- The output displays multiple user records, including:
  - Unique user IDs
  - Usernames
  - Encrypted (hashed) passwords
  - Creation and update timestamps

```
test> use todolist
switched to db todolist
todolist> db.users.find()
[
  {
    _id: ObjectId('693d0a50f32f1ae672773b33'),
    username: 'tiny',
    password: '$2a$12$idLkv1UZFOANi6vOOLL1L.aWxDzXmyG5dFtJC12G8rIdPOCK/e8sG',
    createdAt: ISODate('2025-12-13T06:40:16.839Z'),
    updatedAt: ISODate('2025-12-13T06:40:16.839Z'),
    __v: 0
  },
  {
    _id: ObjectId('693d0a98f32f1ae672773b3b'),
    username: 'hunter',
    password: '$2a$12$Aj6e2bJ7P1mlDVYmoSnX.Oqg4ORgiAVMzH9jZOuU7VYigRIFPLZAi',
    createdAt: ISODate('2025-12-13T06:41:28.632Z'),
    updatedAt: ISODate('2025-12-13T06:41:28.632Z'),
    __v: 0
  }
]
```

Observations

- Two users are present in the database:
  - `tiny`
  - `hunter`
- Passwords are stored securely using hashing, not in plain text.
- Timestamps confirm that users were created during application usage.
- Each user entry is uniquely identified, confirming proper schema handling.

What This Confirms

- Successful communication between the TodoPro backend and MongoDB
- Correct database writes from the application running in EKS
- Secure handling of user credentials
- Proper isolation and persistence of user data

Step 15: Verifying User-Specific Tasks Stored in MongoDB

Description:
In this step, the `todos` collection in MongoDB is queried to verify that tasks created by a specific user are correctly stored and associated with that user. This confirms proper task-to-user mapping and data persistence in the backend database.

Query Executed

- A filtered query is run on the `todos` collection using the user's unique ObjectId:
- `db.todos.find({ userId: ObjectId("693d0a50f32f1ae672773b33") }).pretty()`
- This ensures only tasks belonging to the selected user (`tiny`) are returned.

Retrieved Task Details

The output shows a task document containing:

- **task:** `lunch`
- **time:** `14:12`
- **category:** `General`
- **priority:** `Medium`
- **completed:** `false`
- **userId:** Linked to the correct user ObjectId
- **createdAt / updatedAt:** Timestamps confirming when the task was added

```
todolist> db.todos.find({userId: ObjectId('693d0a50f32f1ae672773b33')}).pretty()
...
[
  {
    _id: ObjectId('693d0a5ef32f1ae672773b37'),
    userId: ObjectId('693d0a50f32f1ae672773b33'),
    task: 'lunch',
    time: '14:12',
    category: 'General',
    priority: 'Medium',
    completed: false,
    createdAt: ISODate('2025-12-13T06:40:30.739Z'),
    updatedAt: ISODate('2025-12-13T06:40:30.739Z'),
    __v: 0
  }
]
```

Observations

- The task is correctly associated with the intended user.
- Task metadata (category, priority, time) is preserved accurately.
- The task status (`completed: false`) reflects the UI state.
- MongoDB schema is functioning as designed.

What This Confirms

- Successful persistence of task data from the TodoPro UI
- Proper user-to-task relationship handling
- Backend API correctly writes and retrieves task records
- MongoDB integration inside Kubernetes is fully operational

Project Conclusion (Part – 1)

This project successfully demonstrates the **end-to-end deployment of a full-stack Todo application (TodoPro)** on **AWS Elastic Kubernetes Service (EKS)** using **containerized microservices and native Kubernetes manifests**.

Throughout this implementation, the application lifecycle was managed from **local development to cloud production**, covering infrastructure provisioning, containerization, orchestration, and runtime verification.

What Was Achieved in Part – 1

- **Infrastructure Setup**
  - AWS CLI, eksctl, and kubectl configured
  - Secure IAM user with appropriate permissions
  - SSH key generation and cluster access setup
- **Kubernetes Cluster Provisioning**

    - EKS cluster created using eksctl
    - Managed node groups deployed
    - Cluster verified via `kubectl get nodes`
- **Application Containerization**
    - Custom Docker image built for the TodoPro Node.js application
    - MongoDB deployed using a custom container image
    - Images stored and pulled from Docker Hub
- **Kubernetes Resource Management**
    - Namespace isolation for the application
    - ConfigMaps and Secrets for configuration and sensitive data
    - Deployments and Services for both application and database
    - LoadBalancer service exposing the application publicly
- **Automated Deployment**
    - Centralized deployment using `deploy.sh`
    - Sequential rollout ensuring MongoDB availability before application startup
- **Application Validation**
    - Application accessed via AWS LoadBalancer DNS
    - User registration and authentication validated
    - Multiple users and tasks successfully created
    - Data persistence verified directly inside MongoDB pods
- **Backend Verification**
    - MongoDB queried from within Kubernetes pods
    - User and task records confirmed at database level

Key Outcome

The TodoPro application is now:

- Fully containerized
- Running on a production-grade Kubernetes cluster
- Exposed securely via AWS Load Balancer
- Persisting and managing user-specific data correctly

This confirms a **successful Kubernetes-native deployment using raw YAML manifests**.

Project Status

Part – 1: Completed

This phase focused on **manual Kubernetes resource definitions** to build strong foundational understanding of:

- Kubernetes objects
- Networking
- Secrets and configuration
- Application-to-database communication

What's Next – Part 2

In the **next project (Part – 2)**, the same TodoPro application will be deployed using:

Helm – Kubernetes Package Manager

Upcoming enhancements:

- Helm charts for application and MongoDB
- Parameterized values using `values.yaml`
- Reusable, versioned deployments
- Simplified upgrades and rollbacks
- Environment-specific configurations (dev / prod)

This will demonstrate **production-grade Kubernetes deployment practices** using Helm.

Final Note

This project establishes a strong real-world foundation in:

- Docker
- Kubernetes
- AWS EKS
- Cloud-native application deployment

Part – 2 with Helm will build on this foundation and elevate the deployment to enterprise standards.