Task Tracker API – Blue-Green Deployment

Overview

The **Task Tracker API** is a containerized application deployed using a **Blue-Green deployment strategy** to ensure **zero downtime updates**. This deployment utilizes **Docker containers**, **Nginx as a reverse proxy**, and **AWS EC2 instances**. The system is designed for high availability, automated deployment, and robust monitoring using **Prometheus and Grafana**.

Key Features:

- Containerized API using Docker
- Blue-Green deployment for zero downtime
- Automated CI/CD pipeline with GitHub Actions
- PostgreSQL database with persistent storage
- Monitoring with Prometheus and Grafana
- Secure deployment on AWS EC2

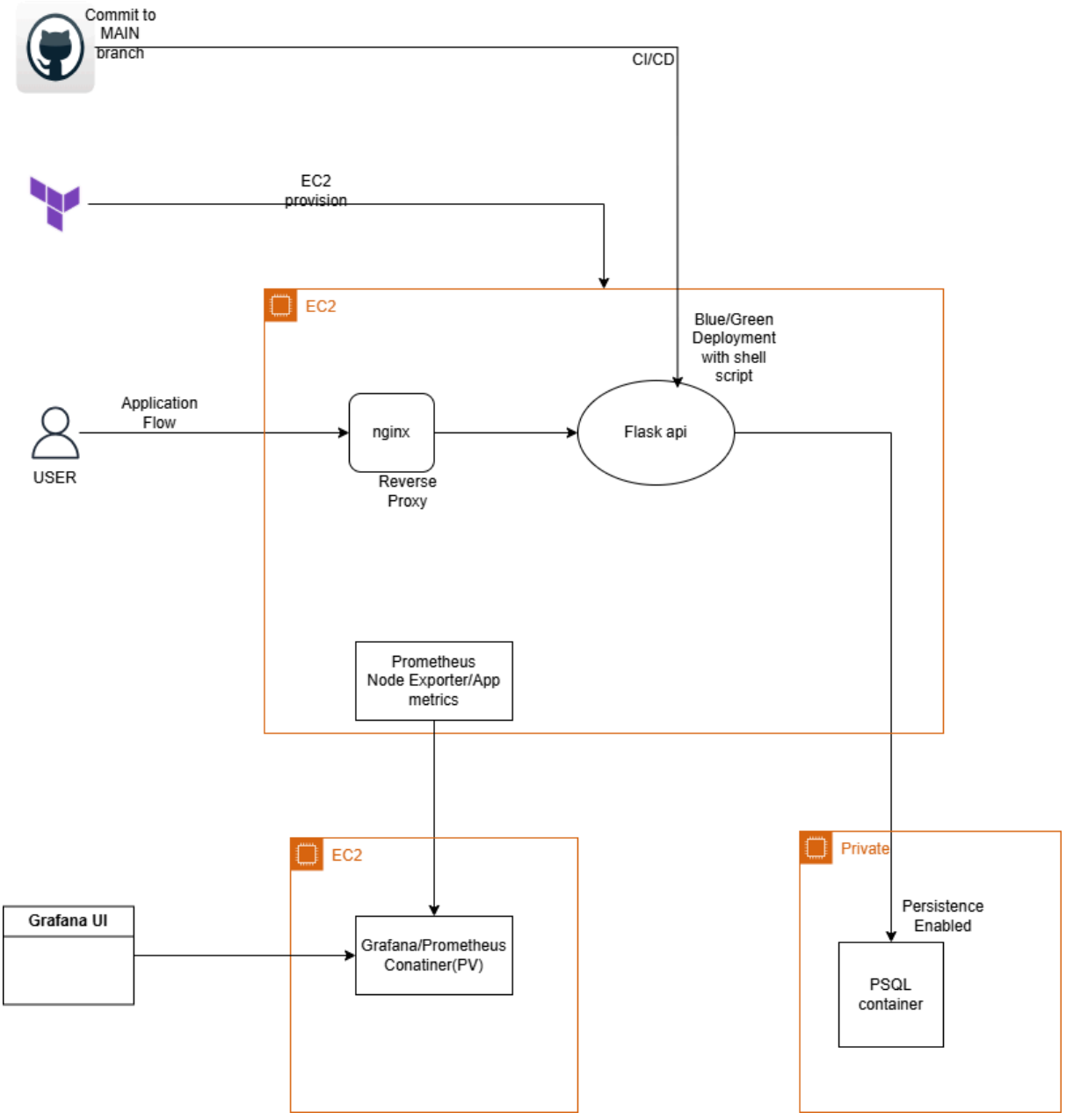
Architecture

Components:

1. **Users:** Access the Task Tracker API via HTTP requests.
2. **Task Tracker API:** Containerized application running in Docker.
3. **Blue-Green Deployment:** Two containers (Blue and Green) run simultaneously; traffic is switched based on health checks.
4. **AWS EC2 Instance:** Hosts the Docker containers.
5. **AWS Security Groups:** Manage access to EC2 instances.

6. **Nginx:** Reverse proxy routing requests to active container.
7. **CI/CD Pipeline:** Automates Docker image build and deployment.
8. **Monitoring:** Prometheus and Grafana track system and application metrics.

Architecture Diagram



Prerequisites

Before deployment:

1. Set up the **Database VM** and **Monitoring VM** (either via Terraform or manually).
2. Configure **persistent volumes (PV)** for the database and **persistent volume (PV)** for monitoring data.
3. Install **Docker** on all EC2 instances.
4. Ensure AWS credentials are configured for Terraform and EC2 access.

Step 1 – Infrastructure Setup

Terraform is used to provision the infrastructure.

Steps:

Repo-Url: https://github.com/ganesh082000/task-tracker-api/tree/main

1. Clone the repository:

```
git clone <repo-url>
cd <repository-folder>
```

2. Install Terraform locally and configure AWS credentials.
3. Navigate to the Terraform directory and execute:

```
cd terraform/
   terraform init
   terraform apply
```

Terraform provisions:

- EC2 instance for the application
- S3 bucket for storage (if needed)

After `terraform apply`, Terraform outputs the **public IP** of the EC2 instance and S3 Bucket.

Step 2 – Database Setup

2.1 Provision Database EC2 Instance

- Create an EC2 instance for PostgreSQL (manual or via Terraform).
- Ensure proper **security group rules** for database access.

2.2 Run PostgreSQL in Docker

```
docker run -d \
  --name my_postgres \
  -p 5432:5432 \
  -e POSTGRES_USER=<db-user> \
  -e POSTGRES_PASSWORD=<db-passwd> \
  -e POSTGRES_DB=taskdb \
  -v pg_data:/var/lib/postgresql/data \
  postgres:15
```

2.3 Update `.env` File

Set database credentials for the application:

```
DB_HOST=localhost
DB_PORT=5432
DB_NAME=taskdb
DB_USER=<db-user>
DB_PASSWORD=<db-passwd>
```

Step 3 – Application Deployment

3.1 Dockerize the Application

Build the Docker image:

```
docker build -t yourdockerhubusername/task-tracker:latest .
```

3.2 Deploy with Blue-Green Strategy

Use `deploy.sh` to deploy the application:

```
./deploy.sh
```

- The script ensures health checks before switching traffic between **Blue** and **Green** containers.
- Nginx routes user requests to the active container.

Step 4 – CI/CD Pipeline

- **GitHub Actions** automates building and deploying Docker images.
- Pipeline triggers on push to the **main** branch.
- Configuration file: `./github/workflows/deploy.yml`.

Step 5 – Monitoring (Prometheus & Grafana)

5.1 Install Prometheus & Grafana

- Deploy on the **Monitoring EC2 instance**.
- Configure **PVC** for persistent metric storage.

5.2 Node Exporter Installation

Collect system metrics from the EC2 instance:

```
cd /opt
wget
https://github.com/prometheus/node_exporter/releases/download/v1.3.1
/node_exporter-1.3.1.linux-amd64.tar.gz
tar -xvzf node_exporter-1.3.1.linux-amd64.tar.gz
```

```
cd node_exporter-1.3.1.linux-amd64
```

5.3 Prometheus Configuration

Update `prometheus.yml` to scrape metrics:

```
scrape_configs:
  - job_name: 'node'
    static_configs:
      - targets: ['<ec2-public-ip>:9100']

  - job_name: 'application'
    static_configs:
      - targets: ['<ec2-public-ip>:8000']
```

Step 6 – Nginx Reverse Proxy & Blue-Green Strategy

- Nginx routes traffic to the active container on **port 80**.
- Supports **zero-downtime updates** using Blue-Green deployment.

Step 7 – Testing the API

Use **cURL** or **Postman**:

Create Task:

```
curl -X POST http://<ip>/tasks \
  -H "Content-Type: application/json" \
  -d '{
        "title": "Learn FastAPI",
        "start_date": "2026-01-10",
        "end_date": "2026-01-15",
        "completed": false
      }'
```

Get Tasks:

```
curl http://<your-ec2-public-ip>/tasks
```

Technologies Used

- **Python & FastAPI** – API development
- **Docker** – Containerization
- **PostgreSQL** – Database
- **AWS EC2** – Hosting
- **Nginx** – Reverse proxy
- **Prometheus & Grafana** – Monitoring
- **GitHub Actions** – CI/CD
- **Terraform** – Infrastructure as Code

Design Decisions and Challenges

1. Blue-Green Deployment Strategy

- **Decision:** Chose **Blue-Green deployment** to ensure **zero downtime** during updates.
- **Reason:** Avoids service disruption while deploying new versions.
- **Challenge:** Managing two containers (Blue & Green) and ensuring **traffic switches only after health checks pass**.
- **Solution:** Automated with `deploy.sh` to check container health before switching Nginx routes.

2. Containerization with Docker

- **Decision:** Used **Docker** for both application and database.
- **Reason:** Ensures consistent environment across development, staging, and production.
- **Challenge:** Persistent data for PostgreSQL across container restarts.
- **Solution:** Mounted **Docker volumes** (`pg_data`) to persist database data.

3. Reverse Proxy with Nginx

- **Decision:** Deployed **Nginx** as a reverse proxy for traffic management.
- **Reason:** Required for routing traffic to the active container and handling HTTP requests efficiently.
- **Challenge:** Properly configuring Nginx to switch between Blue and Green containers automatically.
- **Solution:** Integrated Nginx with the deployment script to dynamically update the active upstream.

4. Database Selection

- **Decision:** Used **PostgreSQL**.
- **Reason:** Robust relational database suitable for task management and easily containerized.
- **Challenge:** Ensuring secure access and proper configuration for the API.
- **Solution:** Configured environment variables (`.env`) and restricted access using AWS Security Groups.

5. Monitoring and Metrics

- **Decision:** Implemented **Prometheus and Grafana** for monitoring.
- **Reason:** Needed visibility into application and system performance.
- **Challenge:** Scraping metrics from both the EC2 instance and containers.

- **Solution:** Installed **Node Exporter** for system metrics and exposed API metrics for Prometheus.

6. CI/CD Automation

- **Decision:** Used **GitHub Actions** to automate builds and deployments.
- **Reason:** Reduce manual errors and speed up deployment.
- **Challenge:** Integrating Blue-Green deployment into the pipeline.
- **Solution:** `deploy.sh` script incorporated into GitHub Actions workflow for seamless automated deployment.

7. Security and Access Control

- **Decision:** Configured **AWS Security Groups** and environment variables for secure access.
- **Reason:** Protects database and application from unauthorized access.
- **Challenge:** Balancing security with ease of access for the deployment scripts.
- **Solution:** Restricted EC2 inbound traffic to only required ports and sources.