SAFLE DEVOPS ENGINEER ASSIGNMENT

Introduction

This repository contains the code for deploying a containerized **Node.js application** that uses a **MongoDB** database, automated through a **CI/CD pipeline**. The infrastructure is provisioned using **Terraform**, and the application is built and deployed with **Docker**.

Infrastructure Deployment

Follow these steps to deploy the infrastructure on **Google Cloud Platform (GCP)** using **Terraform**.

1. **Install Terraform:** Make sure that Terraform is installed on your local machine. Follow the official Terraform installation guide here: Install Terraform.
2. **Clone the Repository:** Clone the repository to your local machine:

   ```
   git clone https://github.com/your-repo-url.git
   cd your-repo-folder
   ```

3. **Set up Google Cloud Credentials:** Ensure you have a **Google Cloud service account** with the appropriate permissions. The service account credentials file (`secret.json`) should be placed in your repository or securely referenced. Set the environment variable `GOOGLE_APPLICATION_CREDENTIALS`:

   ```
   export
   GOOGLE_APPLICATION_CREDENTIALS=path_to_your_service_account_key.json
   ```

4. **Terraform Variables:** Set the required Terraform variables. You can either export them as environment variables or provide them in a `terraform.tfvars` file:

   ```
   export GCP_PROJECT="your-project-id"
   export GCP_REGION="asia-south1"
   ```

5. **Initialize Terraform:** Run the following command to initialize the Terraform workspace:

   ```
   terraform init
   ```

6. **Apply Terraform Configuration:** To deploy the infrastructure, use:

   ```
   terraform apply
   ```

   Confirm the action by typing `yes` when prompted.

7. **Verify the Infrastructure:** After deployment, verify that your infrastructure is set up correctly. This includes:

   - Google Cloud VPC, subnets, and firewall rules
   - SQL database instance (MySQL)
   - Compute instances and load balancer for high availability
   - Health checks and scaling configurations.

CI/CD Pipeline

The **CI/CD pipeline** automates the process of building, testing, containerizing, deploying, and verifying your application. This pipeline ensures that every change made to the codebase is automatically built and deployed in a seamless and efficient way, reducing manual intervention and promoting a more reliable delivery process.

The pipeline is defined in **Azure DevOps** using a **YAML configuration**. It is divided into **stages** that are executed sequentially. Below is a detailed breakdown of each stage and the jobs/tasks performed within them.

Overview of CI/CD Pipeline Structure

The pipeline consists of the following stages:

1. **Build**
2. **Dockerize**
3. **Deploy Infrastructure**
4. **Deploy App**
5. **Post-Deploy**

Each stage has one or more jobs, and each job consists of a series of tasks.

---

1. Build Stage

Purpose:
This stage is responsible for setting up the environment, installing dependencies, and running unit tests to ensure that the application is working correctly before it is containerized.

Job: `BuildNodeApp`

- **Checkout Code**: The `Checkout@1` task checks out the latest code from the repository to the build pipeline.
- **Use Node.js**:
  The `UseNode@2` task installs the specific version of Node.js (in this case, version `18.x`) to ensure compatibility with the application. This is crucial as the app depends on Node.js for runtime.

  ```
  - task: UseNode@2
    inputs:
      versionSpec: '18.x'
  ```

- **Install Node.js dependencies**:
  The `npm install` command installs all the necessary packages listed in `package.json`. This includes libraries like `express`, `jest`, and `supertest` that are essential for the app to run and be tested.

  ```
  - script: |
      npm install
    displayName: 'Install Node.js dependencies'
  ```

- **Run Unit Tests**:
  The `npm test` command runs the test suite using `jest`, ensuring that the application behaves as expected before it is built and deployed. If the tests fail, the pipeline will stop, preventing the deployment of broken code.

  ```
  - script: |
  ```

```
      npm test
    displayName: 'Run unit tests'
```

If all tests pass, the pipeline proceeds to the next stage.

---

2. Dockerize Stage

Purpose:
This stage is responsible for building the Docker image for the application and pushing it to the Azure Container Registry (ACR). This allows the app to be deployed as a container, ensuring portability and consistency across environments.

Job: `BuildAndPushDockerImage`

- **Checkout Code**: The `Checkout@1` task checks out the latest code to the build environment.
- **Docker Build and Push**:
  The `Docker@2` task is responsible for building the Docker image and pushing it to Azure Container Registry (ACR). It uses the provided `Dockerfile` to build the image and tags it with the `latest` tag.

  ```
  - task: Docker@2
    inputs:
      containerRegistry: $(containerRegistry)
      repository: $(imageName)
      command: 'buildAndPush'
      Dockerfile: $(dockerfilePath)
      buildContext: .
      tags: |
        latest
  ```

  Explanation:

  - `containerRegistry`: Specifies the Azure Container Registry (ACR) where the image will be pushed.
  - `repository`: Defines the name of the image (e.g., `nodejs-api-app`).
  - `command`: The `buildAndPush` command instructs Docker to both build and push the image.
  - `Dockerfile`: Path to the Dockerfile that defines how to build the image.
  - `tags`: Tags the Docker image with `latest`, indicating the most recent version of the app.

---

3. Deploy Infrastructure Stage

Purpose:
In this stage, **Terraform** is used to deploy and manage the infrastructure. This ensures that the cloud resources (e.g., VMs, load balancers, and databases) are properly configured and ready for deployment.

Job: `DeployInfrastructureWithTerraform`

- **Checkout Code**: Again, the `Checkout@1` task retrieves the latest code from the repository.
- **Install Terraform**:
  The `TerraformInstaller@0` task installs the required version of Terraform (in this case, version `1.3.0`).

```
- task: TerraformInstaller@0
  inputs:
    terraformVersion: '1.3.0'
```

- **Install Google Cloud SDK**:
  The `GoogleCloudSDKInstaller@0` task installs the Google Cloud SDK to enable Terraform to interact with GCP.

```
- task: GoogleCloudSDKInstaller@0
  inputs:
    versionSpec: 'latest'
```

- **Terraform Init**:
  The `terraform init` command initializes Terraform and sets up the backend configuration (i.e., where the Terraform state will be stored).

```
- script: |
    terraform init -backend-config=$(terraformBackendConfig)
  workingDirectory: $(terraformDirectory)
  displayName: 'Terraform Init'
```

- **Terraform Plan**:
  The `terraform plan` command is used to create an execution plan and show what changes will be made to the infrastructure.

```
- script: |
    terraform plan -out=tfplan
  workingDirectory: $(terraformDirectory)
  displayName: 'Terraform Plan'
```

- **Terraform Apply**:
  The `terraform apply` command applies the changes to the infrastructure, provisioning the resources like VMs, load balancers, and databases.

```
- script: |
    terraform apply -auto-approve tfplan
  workingDirectory: $(terraformDirectory)
  displayName: 'Terraform Apply'
```

---

4. Deploy App Stage

Purpose:
In this stage, the containerized application is deployed to the cloud infrastructure. The Docker image is pulled from ACR and run in the container.

Job: `DeployContainerizedApp`

- **Pull Docker Image from ACR**:
  The `docker pull` command pulls the latest version of the Docker image from Azure Container Registry.

```
- script: |
    docker pull $(containerRegistry)/$(imageName):latest
  displayName: 'Pull Docker Image from ACR'
```

- **Run Docker Container**:
  The `docker run` command deploys the container and binds port `80` to make the app accessible.

```
- script: |
    docker run -d -p 80:80 $(containerRegistry)/$(imageName):latest
  displayName: 'Deploy Docker Container'
```

---

5. Post-Deploy Stage

Purpose:
The post-deploy stage runs health checks and verifies that the deployed application is functioning properly.

Job: `RunPostDeployTests`

- **Health Check**:
  The `curl` command checks the `/health` endpoint of the deployed application to confirm that it is responding as expected (status `OK`).

```
- script: |
    curl http://localhost:80/health
  displayName: 'Run post-deploy tests'
```

If the `/health` endpoint returns `status: "OK"`, the application is considered successfully deployed.

Triggering the CI/CD Pipeline

The pipeline is **automatically triggered** when changes are pushed to the `main` branch. However, if you want to trigger the pipeline manually, follow these steps:

1. Go to **Azure DevOps**.
2. Navigate to **Pipelines** > **Select your pipeline**.
3. Click on **Run Pipeline** to trigger the pipeline.

Monitoring and Alerts in Google Cloud Platform (GCP)

Monitoring and alerting are critical for maintaining the health, performance, and reliability of infrastructure and applications. In a cloud-native environment like Google Cloud Platform (GCP), it's essential to ensure that your virtual machines (VMs), databases, load balancers, and applications are performing optimally. Google Cloud provides a comprehensive suite of tools for monitoring, logging, and alerting, which can be leveraged to track resource utilization, detect anomalies, and trigger timely notifications.

Google Cloud Monitoring (formerly Stackdriver)

Google Cloud Monitoring (GCM) allows you to monitor the performance and availability of various GCP services, including **Compute Engine**, **Google Kubernetes Engine (GKE)**, **Cloud SQL**, **Cloud Functions**, **load balancers**, and custom application metrics. GCM collects metrics such as **CPU utilization**, **memory usage**, **disk I/O**, and **network traffic** from your VM instances, which are essential for ensuring the health of your infrastructure.

Key Benefits of Google Cloud Monitoring:

- **Real-time Metrics**: Collect and analyze metrics from all GCP resources in real time.
- **Custom Dashboards**: Visualize metrics in custom dashboards for easy access to critical data, such as resource consumption and system health.
- **Integration with Cloud Logging**: Google Cloud Monitoring integrates seamlessly with **Cloud Logging** to provide insights into your application logs and resource performance.
- **Cloud SQL Monitoring**: With Cloud SQL, you can monitor database-specific metrics such as **connections**, **query performance**, and **storage usage**.

Setting Up Google Cloud Monitoring:

1. **Enable Cloud Monitoring API**: First, enable the **Google Cloud Monitoring API** from the GCP Console, if it's not already enabled.
2. **Install Monitoring Agent**: To gather additional system-level metrics (e.g., CPU, memory, disk space), install the **Cloud Monitoring agent** on each VM instance. This can be done using a startup script (e.g., for a Debian-based system):
3. **Create Dashboards**: In the Google Cloud Console, create custom dashboards that track key metrics such as CPU usage, memory consumption, and disk I/O for VMs. This will allow you to visually monitor the health of your infrastructure.

Setting Up Cloud Monitoring Alerts

Once Google Cloud Monitoring is configured, you can define alert policies to track specific thresholds or conditions. These alerts are crucial in identifying performance issues, security breaches, or failures before they affect the application.

Common Metrics to Monitor and Set Alerts for:

1. **High CPU Usage**: If your VM instances are running at high CPU utilization (e.g., above 80% for a set period), this could indicate resource contention or inefficient application code.

   Alert Condition:

   - Metric: **CPU utilization**
   - Threshold: **> 80% for 5 minutes**
   - Action: Send an alert via **Cloud Pub/Sub** or **email** to the system administrator.

2. **Database Connection Issues**: If your application is experiencing issues connecting to the database, this can lead to performance degradation or downtime. Monitoring database connection errors or failures is critical for maintaining application availability.

   Alert Condition:

   - Metric: **Database connections**
   - Threshold: **Connection failures or timeouts** exceed a set number within a given time frame (e.g., 5 failures in 1 minute).
   - Action: Send an alert to the team via email or SMS.

3. **Load Balancer Health Check Failures**: A load balancer's health checks ensure that only healthy instances are serving traffic. If instances behind a load balancer fail their health checks, traffic can be routed to unhealthy instances, resulting in downtime or poor user experience.

   Alert Condition:

   - Metric: **Health check status**
   - Threshold: **> 1 failed health check** within 5 minutes.
   - Action: Notify the team through **Slack** or **email** when an instance fails its health check.

Alerting Mechanism: Email, SMS, and Notifications

Alerts are only valuable if they can be acted upon quickly. Google Cloud Monitoring integrates with **Google Cloud Pub/Sub** and **Cloud Functions** to provide instant notifications when an alert condition is met.

1. **Notification Channels**:

   - **Email Notifications**: Alerts can be sent directly to emails of designated administrators or teams, ensuring they are immediately informed of issues.
   - **SMS Notifications**: For high-priority issues, SMS notifications can be triggered using **Cloud Pub/Sub** and third-party SMS services (e.g., Twilio).
   - **Slack Alerts**: Cloud Monitoring integrates with **Slack** to send real-time alerts to designated Slack channels.

2. **Setting Up Notification Channels**:

   - You can configure notification channels under **Google Cloud Monitoring > Alerting > Notification Channels**. After this, when an alert is triggered, the system will automatically notify the relevant stakeholders.

Post-Deployment Health Checks

A **post-deployment health check** is a critical component of your CI/CD pipeline to ensure that the application is functioning correctly after deployment. After deploying the app (as part of the CI/CD pipeline), a **health check** is triggered by querying the `/health` endpoint of the application. If the endpoint responds with a status code 200 and a message like `status: "OK"`, the deployment is deemed successful.

Health Check Implementation:

- The CI/CD pipeline includes a step that runs an HTTP request to the `/health` endpoint of the app after it has been deployed.
- If the `/health` endpoint returns a response like `{ "status": "OK" }`, the pipeline continues. Otherwise, an alert can be triggered indicating that the deployment has failed, allowing the team to take corrective action.

Design Decisions

1. Google Cloud Platform (GCP) as the Cloud Provider

We chose **Google Cloud Platform (GCP)** as the cloud provider for this project for several reasons:

- **Managed Services**: GCP provides a rich set of **managed services** like **Cloud SQL** for databases and **Compute Engine** for virtual machines. These services reduce the operational overhead of managing infrastructure, enabling the team to focus on building the application rather than managing servers.
- **Global Network Infrastructure**: Google's global network is one of the most robust in the industry, providing low-latency connections, high availability, and scalability.
- **Cost-effectiveness**: GCP offers competitive pricing, especially when combined with its extensive free-tier offerings for smaller-scale deployments.
- **Integration with Other Google Services**: GCP offers seamless integration with other Google services such as **BigQuery**, **Cloud Pub/Sub**, and **Cloud Functions**, which could be beneficial for future feature expansion.

Overall, GCP was chosen for its rich set of cloud services and the flexibility it offers in terms of scaling and cost management.

2. Docker for Containerization

Docker was selected for containerizing the application for the following reasons:

- **Portability**: Docker containers encapsulate the application and its dependencies, ensuring that the application can run consistently across various environments (local development, CI/CD pipeline, cloud infrastructure, etc.).
- **Scalability**: Docker makes it easier to scale the application horizontally, as new containers can be spun up to meet demand.
- **Isolation**: Each container runs independently, isolating the application from other processes on the host system. This is especially important in multi-tenant cloud environments, where isolation between different services or microservices is critical for security and stability.
- **Ease of Deployment**: Docker allows the application to be deployed as a consistent image across various environments, simplifying the deployment process.

Docker's flexibility and ecosystem made it an ideal choice for our containerization needs, providing the right balance of isolation, scalability, and portability.

3. Terraform for Infrastructure Provisioning

The decision to use **Terraform** for provisioning infrastructure was based on the following considerations:

- **Infrastructure as Code (IaC)**: Terraform allows us to define infrastructure in code, making it easy to version, track, and reproduce environments. This improves team collaboration, enables automation, and reduces manual errors.
- **Provider-Agnostic**: Although the project is deployed on GCP, Terraform is a provider-agnostic tool. This means that if we decide to migrate to another cloud provider or incorporate multi-cloud strategies in the future, we can reuse the same configurations with minimal changes.
- **Automation**: With Terraform, we can automate infrastructure provisioning, making it easier to spin up and tear down resources as needed. This is crucial for consistent and reproducible environments.
- **Version Control**: Terraform allows the state of infrastructure to be managed in a versioned manner, improving traceability and auditing.

Using Terraform enhances the reproducibility, reliability, and maintainability of infrastructure.

4. CI/CD with Azure DevOps

Azure DevOps was selected for automating the CI/CD pipeline for the following reasons:

- **Built-in Support for Docker**: Azure DevOps natively supports Docker, enabling easy integration of Docker builds and deployments.
- **Terraform Support**: Azure DevOps provides built-in tasks for managing Terraform workflows, simplifying the automation of infrastructure provisioning as part of the pipeline.
- **Cloud Integrations**: Azure DevOps integrates with multiple cloud providers, including GCP, which is essential for deploying our application and managing infrastructure on Google Cloud.
- **Flexibility**: Azure DevOps pipelines offer flexibility, allowing us to create custom stages and jobs, such as building the application, running tests, provisioning infrastructure, and deploying the application in a structured way.

Azure DevOps provides a robust, flexible platform that can automate all aspects of the development lifecycle, from code integration to deployment, ensuring continuous delivery and consistent application updates.

Trade-offs

1. Database Choice: MySQL vs NoSQL

For this project, we chose **MySQL** as the database instead of NoSQL options like **MongoDB** or **Cassandra**. The decision was primarily driven by the following reasons:

- **Structured Data & ACID Compliance**: Our application requires structured data with complex relationships between entities. MySQL's support for **ACID (Atomicity, Consistency, Isolation, Durability)** compliance makes it a natural fit for these requirements, ensuring data integrity during transactions.
- **Familiarity**: The development team is more familiar with MySQL and its ecosystem, which reduces learning time and the risk of errors during development.

However, MySQL does have some limitations:

- **Horizontal Scalability**: While MySQL can be scaled vertically (by increasing resources on a single server), it faces challenges when scaling horizontally, especially for read/write-intensive applications. NoSQL databases, such as MongoDB, are generally better suited for horizontal scaling.
- **Fixed Schema**: NoSQL databases offer greater flexibility with schema design, which can be an advantage for rapidly evolving applications. However, for the structured nature of this project, MySQL's rigid schema was a better fit.

This trade-off may need to be revisited as the application grows and scaling requirements increase.

2. Scalability: VM Autoscaling vs Kubernetes

The decision to use **Compute Engine** VMs with **autoscaling** for this application, instead of a more complex **Kubernetes** setup, was based on the simplicity and immediate requirements of the project. While GCP autoscaling for VMs works well for basic load handling, it has some limitations compared to Kubernetes in terms of **dynamic scaling**, **container orchestration**, and **self-healing** capabilities.

- **Trade-off**: Kubernetes offers more robust scaling and management of containerized applications, but it introduces additional complexity and requires more expertise in managing the Kubernetes cluster.

As the application grows, Kubernetes may become a more appropriate choice for container orchestration, providing better scalability, availability, and fault tolerance.

Challenges

1. Service Account Key Management

Managing **service account keys** securely is a constant challenge in cloud environments. Service account keys provide access to cloud resources, and if compromised, they could lead to significant security risks.

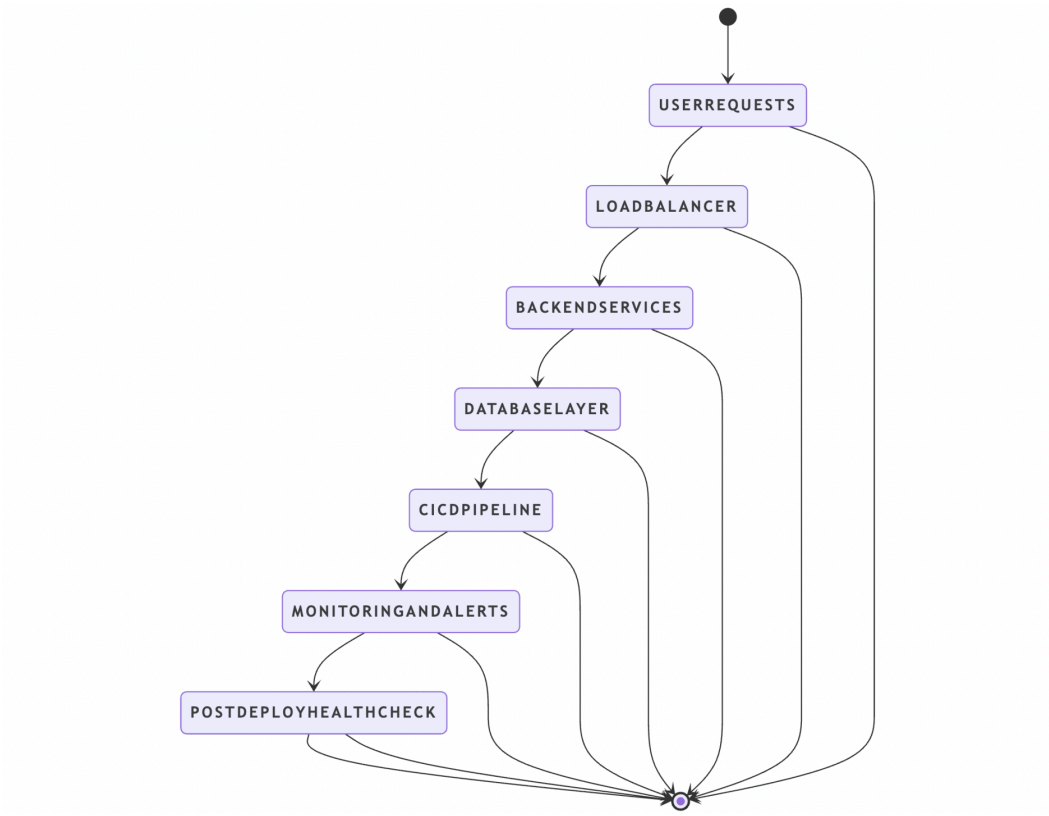
- **Challenge**: Storing the keys securely while ensuring that they are available to services that need them, without hardcoding them in the codebase or infrastructure files.
- **Solution**: In this case, we used **environment variables** to store sensitive keys, ensuring they are not exposed in source code. However, it is crucial to rotate keys regularly and ensure that they are assigned the minimum necessary permissions via **Identity and Access Management (IAM)** roles.

2. State Management in Terraform

Managing Terraform **state files** securely and consistently is a critical challenge. Terraform stores the state of your infrastructure in a file, and if the state file is not managed properly, it can lead to **state corruption** or **inconsistencies** when multiple team members are working on the same infrastructure.

- **Challenge**: When working in a team environment, managing and storing Terraform state files can become difficult, especially when the state file is local or improperly shared.
- **Solution**: We recommend using **Google Cloud Storage** as a backend for Terraform state, which provides a shared, centralized location for state files. This allows for version control and ensures that all team members are working with the most up-to-date version of the state.

Architectural Diagram



Deliverables

The deliverables for this project are a comprehensive set of files and configurations that encapsulate the infrastructure setup, application containerization, CI/CD pipeline, and monitoring setup. Each deliverable is an essential component for ensuring the successful deployment and operation of the application. Below is an overview of the key deliverables:

1. GitHub Repository

The GitHub repository contains all the source code, infrastructure configurations, and deployment automation scripts. It serves as the single source of truth for all project components, ensuring consistency across environments and facilitating collaboration within the team.

Components of the GitHub Repository:

- **Terraform Configuration Files for Infrastructure Provisioning** The Terraform configuration files are used to provision cloud infrastructure in **Google Cloud Platform (GCP)**. These files define and configure resources like compute instances, managed instance groups, load balancers, databases, and networking components, ensuring that the required infrastructure is deployed in a consistent and repeatable manner. Key files include:

    - **network.tf**: Defines the networking resources such as VPC and subnets.
    - **instance.tf**: Configures virtual machines and instance templates for scaling.
    - **loadbalancer.tf**: Sets up the load balancer to distribute traffic to backend instances.
    - **database.tf**: Defines the MySQL database and its configurations.
    - **outputs.tf**: Specifies outputs like the IP address of the load balancer and database instance details.
    - **variables.tf**: Contains variables for configuration such as project ID, region, and database password.

- **Dockerfile and Docker-Compose.yml for Containerization** The **Dockerfile** defines how to package the application and its dependencies into a container image, ensuring consistency across different environments. It specifies the base image, working directory, dependencies, and the command to start the application. The **docker-compose.yml** file is used for multi-container orchestration, setting up the application container and a MongoDB container for local development and testing. It also defines necessary environment variables and port mappings.
- **CI/CD Pipeline Configuration** The **CI/CD pipeline configuration**, specifically the **Azure DevOps YAML pipeline**, automates the application build, testing, Docker image creation, infrastructure provisioning, and deployment. It includes multiple stages:

    - **Build**: Installs dependencies, runs tests, and ensures the application is ready for deployment.
    - **Dockerize**: Builds and pushes the Docker image to a container registry (Azure Container Registry).
    - **Deploy Infrastructure**: Uses Terraform to provision the necessary infrastructure (VMs, databases, load balancers) on GCP.
    - **Deploy Application**: Deploys the containerized application to the cloud.
    - **Post-Deploy Tests**: Executes health checks to verify the application is running correctly.

- **Monitoring and Alerts Configuration** Configuration for monitoring and alerting is also a key deliverable. It includes:

    - **Google Cloud Monitoring Setup**: Configures Google Cloud Monitoring (formerly Stackdriver) to track key metrics such as CPU usage, database connections, and health check failures.
    - **Alerting**: Defines alerts and notifications (email, SMS, or Pub/Sub) for critical issues, ensuring that the team is notified when an alert is triggered.

2. Documentation (README)

The **README** file provides detailed instructions for setting up, deploying, and maintaining the infrastructure and application. It includes:

- **Deployment Instructions**: Detailed steps to deploy infrastructure using Terraform, build and deploy the application using Docker, and run the CI/CD pipeline.
- **CI/CD Pipeline Trigger**: Instructions on how to trigger the pipeline and deploy new versions of the application.
- **Monitoring and Alerts Setup**: Instructions for setting up Google Cloud Monitoring and configuring alerts to ensure the system operates smoothly.
- **Troubleshooting**: Common issues and how to resolve them during deployment, containerization, or monitoring.

3. Running Application

Finally, the **running application** (or the steps to access it locally) is a key deliverable. If deployed on a public cloud, the application's URL will be provided in the outputs. Otherwise, instructions will be provided to run the application locally using **Docker** and **Docker Compose**. This allows stakeholders to verify the deployed application and perform tests.

Conclusion

In conclusion, the project was designed to deliver a robust, scalable, and automated system for deploying and managing a containerized application. The chosen technologies and tools—**Google Cloud Platform**, **Docker**, **Terraform**, and **Azure DevOps**—were selected to streamline the deployment process, improve maintainability, and ensure scalability. Each component was carefully crafted to meet the current requirements while allowing flexibility for future expansion.

The **Terraform infrastructure code** automates cloud provisioning, ensuring that the infrastructure is consistent and easily reproducible. **Docker** enables the application to be containerized for portability and ease of deployment across different environments. The **CI/CD pipeline** orchestrates the build, test, deployment, and monitoring processes, ensuring that the application can be quickly and reliably deployed and updated.

Monitoring and alerting via **Google Cloud Monitoring** ensures that the application's health is continually tracked, providing early warnings in case of performance degradation or failures. This proactive approach to monitoring helps mitigate risks and ensures that any issues are promptly addressed.

However, there were some trade-offs and challenges faced during the project. For instance, the choice of **MySQL** over **NoSQL** databases may limit horizontal scalability in the future, but was deemed appropriate for the structured nature of the data in this application. Similarly, while **VM autoscaling** was sufficient for the current project, **Kubernetes** could be considered for more complex scaling needs down the line.

Overall, the project serves as a solid foundation for further development, with the ability to scale and adapt as requirements evolve. The **automated infrastructure**, **containerization**, and **CI/CD pipeline** together ensure a high level of efficiency, consistency, and reliability in the deployment and management of the application.