Basic web service application in Python

This repo is for python based microservice with basic webserver and automated CI/CD pipeline, choose the components of needs and can build and deploy a webserver microservice code.

Application Description
This application involves creating a basic web service in Python that serves a simple HTTP GET endpoint. This endpoint returns the message "Hello from Service #1".

Capabilities
Below capabilities are available:

1. Image creation using Docker
2. High volume of requests
3. critical to business operations
4. Interaction with a Database and an external API in the future.
5. High availability and resilience to failures.

Image creation using Docker
  Dockerfile:
    FROM python:3.9-slim
    WORKDIR /app
    COPY . /app
    RUN pip install flask
    EXPOSE 5000
    ENV FLASK_APP=app.py
    CMD ["flask", "run", "--host=0.0.0.0"]

  Build and Run:
    docker build -t soujanyachalla/hello-service .
    docker run -p 5000:5000 soujanyachalla/hello-service

High volume of requests
  The service is designed to handle a high volume of incoming requests efficiently and reliably. This is critical for ensuring smooth and uninterrupted business operations. The following strategies and mechanisms are employed to achieve this:
 # Scalability:
   # Horizontal Scaling:
      The application is containerized and deployed on a Kubernetes cluster. Kubernetes allows for horizontal scaling by adding more replicas of the service to distribute the load evenly.
   # Auto-scaling:
      Kubernetes Horizontal Pod Autoscaler (HPA) is configured to automatically scale the number of pod replicas based on CPU utilization or other custom metrics.
 # Load Balancing:
   # Kubernetes Service:
      A LoadBalancer type service is used to distribute incoming traffic across the available replicas of the application. This ensures that no single instance is overwhelmed by too many requests.
   # External Load Balancer:

For enhanced performance and redundancy, an external load balancer (such as AWS Elastic Load Balancer) can be integrated to distribute traffic across multiple nodes and regions if necessary.

Fault Tolerance:

Replica Management:

By maintaining multiple replicas of the application, Kubernetes ensures high availability. If one replica fails, the service continues to operate with the remaining replicas while Kubernetes works to replace the failed one.

Self-healing:

Kubernetes' self-healing capabilities automatically restart failed containers, ensuring minimal downtime and service disruption.

Performance Optimization:

Resource Allocation:

Resource requests and limits are defined in the Kubernetes manifests to allocate sufficient CPU and memory to each replica. This prevents resource contention and ensures that the application performs optimally under load.

Monitoring and Alerting:

Monitoring:

Tools like Prometheus and Grafana are used to monitor the application’s performance metrics (e.g., response times, request rates, error rates). This helps in identifying performance bottlenecks and scaling issues.

Alerting:

Alerts are configured to notify the DevOps team of any anomalies or performance degradation, enabling quick response to potential issues before they impact the end-users.

By implementing these strategies, the application is well-equipped to handle high volumes of requests, ensuring reliability, high availability, and optimal performance even under heavy load conditions.

critical to business operations

High Availability:

Redundant Architecture:

The application is deployed with multiple replicas across different nodes within the Kubernetes cluster. This redundancy ensures that even if one node or replica fails, the service remains operational.

Geographical Distribution:

For global businesses, the application can be deployed across multiple regions to ensure availability even in the event of a regional outage. This also helps in reducing latency for users across different geographical locations.

Reliability:

Zero Downtime Deployments:

Using Kubernetes features like rolling updates, the application can be updated without any downtime. This ensures continuous availability and minimizes disruption to business operations.

Health Checks:

Liveness and readiness probes are configured to monitor the health of the application. Kubernetes automatically restarts unhealthy pods to maintain service reliability.

Interaction with a Database and an external API in the future.

Database Integration:

Database Choice:

A relational database like PostgreSQL or MySQL will be used for structured data, ensuring ACID (Atomicity, Consistency, Isolation, Durability) properties. Alternatively, a NoSQL database like MongoDB might be considered for unstructured data or high scalability requirements.

External API Integration:

API Client Libraries:

Use client libraries or SDKs provided by the external API provider to simplify integration and ensure compatibility with the API.

API Authentication:

Implement secure authentication methods (e.g., API keys, OAuth tokens) to access the external API, ensuring data security and compliance with best practices.

High availability and resilience to failures.

The service is designed with high availability and resilience to failures to ensure continuous operation and minimal downtime. This is critical for maintaining business continuity and providing a reliable user experience. The following strategies and mechanisms are employed:

Redundant Architecture:

Multiple Replicas:

The application is deployed with multiple replicas across different nodes within the Kubernetes cluster. This redundancy ensures that even if one node or replica fails, the service remains operational.

Cross-Region Deployment:

For global businesses, the application can be deployed across multiple regions to ensure availability even in the event of a regional outage. This also helps in reducing latency for users across different geographical locations.

Automated Failover:

Self-Healing:

Kubernetes' self-healing capabilities automatically detect and replace failed containers, ensuring that the desired number of replicas is always running.

Liveness and Readiness Probes:

Liveness probes monitor the health of the application and trigger restarts if the application becomes unresponsive. Readiness probes ensure that only healthy instances receive traffic, preventing unresponsive instances from affecting user experience.

Zero Downtime Deployments:

Rolling Updates:

Kubernetes rolling updates are used to deploy new versions of the application without downtime. This ensures that users are not impacted during updates and that the service remains available.

Blue-Green Deployments:

For critical updates, blue-green deployment strategies can be used to maintain two identical production environments (blue and green). Traffic is switched to the new environment (green) after validation, ensuring zero downtime during deployment.

Load Balancing:

Kubernetes Service:

A LoadBalancer type service distributes incoming traffic across multiple replicas, ensuring even load distribution and preventing any single replica from being overwhelmed.

External Load Balancer:

An external load balancer (such as AWS Elastic Load Balancer) can be integrated to distribute traffic across multiple nodes and regions, enhancing performance and redundancy.

Resource Management:

Resource Requests and Limits:

Resource requests and limits are defined in the Kubernetes manifests to allocate sufficient CPU and memory to each replica. This prevents resource contention and ensures that the application performs optimally under load.

Auto-scaling:

Kubernetes Horizontal Pod Autoscaler (HPA) automatically scales the number of pod replicas based on CPU utilization or other custom metrics, ensuring that the application can handle varying loads.

Kubernetes Deployment

Kubernetes manifests are created for deploying the containerized application.

Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-service-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hello-service
  template:
    metadata:
      labels:
        app: hello-service
    spec:
      containers:
      - name: hello-service
        image: soujanyachalla/hello-service:latest
        ports:
        - containerPort: 5000
        resources:
          requests:
            memory: "64Mi"
            cpu: "250m"
          limits:
            memory: "128Mi"
            cpu: "500m"
      imagePullSecrets:
      - name: dockerhub-secret
```

Service:

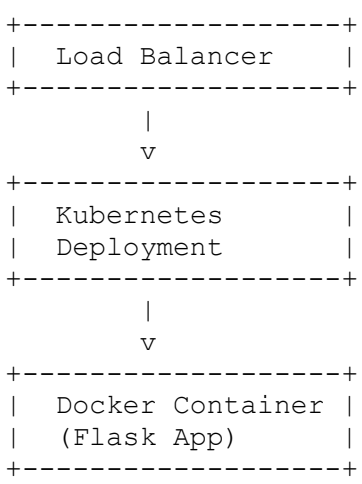
```
apiVersion: v1
kind: Service
metadata:
  name: hello-service
spec:
```

```
  type: LoadBalancer
  selector:
    app: hello-service
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000


## CI/CI pipeline
CI/CD pipeline is set up using GitHub Actions.
name: CI/CD Pipeline

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: 3.9

    - name: Install dependencies
      run: pip install flask

    - name: Build Docker image
      run: docker build -t soujanyachalla/hello-service .

    - name: Login to Docker Hub
      run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{
secrets.DOCKER_USERNAME }}" --password-stdin


    - name: Push the Docker image
      run: docker push soujanyachalla/hello-service:latest

    - name: Run tests
      run: |
        docker run -d -p 5000:5000 soujanyachalla/hello-service
        sleep 5
        curl -f http://localhost:5000

  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
```

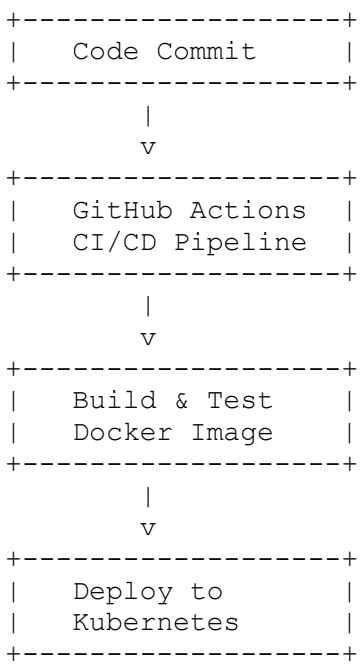
```
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Configure AWS credentials
      uses: aws-actions/configure-aws-credentials@v1
      with:
        aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
        aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        aws-region: us-east-2

    - name: Install kubectl
      run: |
        curl -LO "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
        chmod +x kubectl
        sudo mv kubectl /usr/local/bin/

    - name: Update kubeconfig
      run: |
        aws eks update-kubeconfig --name web-cluster --region us-east-2

    - name: Create Docker Registry Secret if not exists
      run: |
        #!/bin/bash
        SECRET_NAME="dockerhub-secret"
        NAMESPACE="default"

        if kubectl get secret $SECRET_NAME --namespace $NAMESPACE; then
          echo "Secret '$SECRET_NAME' already exists in namespace
'$NAMESPACE'."
        else
          echo "Creating secret '$SECRET_NAME' in namespace
'$NAMESPACE'."
          kubectl create secret docker-registry $SECRET_NAME \
            --docker-server=https://index.docker.io/v1/ \
            --docker-username="${{ secrets.DOCKER_USERNAME }}" \
            --docker-password="${{ secrets.DOCKER_PASSWORD }}" \
            --docker-email="${{ secrets.DOCKER_EMAIL }}" \
            --namespace $NAMESPACE
        fi

    - name: Deploy to Kubernetes
      run: |
        kubectl apply -f deployment.yaml
        kubectl apply -f service.yaml

### Monitoring and Logging Plan
1. **Monitoring**:
    - **Prometheus** for metrics collection.
    - **Grafana** for visualizing metrics.

2. **Logging**:
    - **ELK Stack** (Elasticsearch, Logstash, and Kibana) for centralized
logging and analysis.
```

- **Fluentd** for log aggregation and forwarding to Elasticsearch.

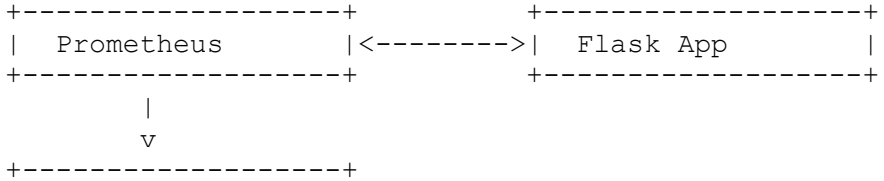
Architecture Diagram:

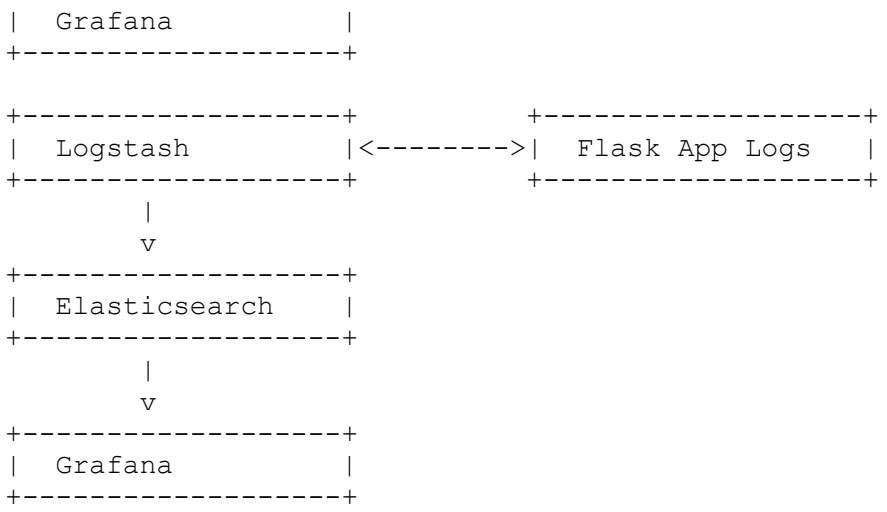


CI/CD Workflow Diagram:



Monitoring and Logging Architecture:




| Grafana |
Logstash
Flask App Logs
Elasticsearch
Grafana