**Kubernetes Deployment Design Document**

**Overview**

The goal is to deploy the URL text retrieval and categorization service onto a Kubernetes cluster, ensuring scalability, monitoring, and efficient deployment. The solution will be containerized and managed with Kubernetes resources, allowing it to handle dynamic loads and provide observability.

**Key Components**

1. **Containerization**:
   * Use Docker to containerize the application.
   * The Dockerfile will package the Java application, ensuring all dependencies are included.
   * The Docker image will be stored in a container registry (e.g., Docker Hub or a private registry).
2. **Kubernetes Resources**:
   * **Deployment**: Manages the application pods, ensuring the desired number of replicas are running.
   * **Service**: Exposes the application within the cluster or to the internet.
   * **Horizontal Pod Autoscaler (HPA)**: Automatically scales the number of pods based on CPU/memory usage or custom metrics.
   * **ConfigMap** and **Secrets** (if needed): Store environment variables, API keys, and other configuration values securely.
3. **Deployment Process**:
   * The deployment will use YAML configuration files to define all Kubernetes resources.
   * Deployments can be managed and updated using tools like kubectl, Helm, or a CI/CD pipeline.

**Detailed Architecture**

1. **Dockerization**:
   * **Dockerfile**:
     + The Dockerfile will start from a base image like openjdk:11-jre-slim.
     + It will copy the compiled JAR file (or compiled classes) of the Java application into the image.
     + Define the entry point to start the Runner class.

Example Dockerfile:

Dockerfile

Copy code

FROM openjdk:11-jre-slim

WORKDIR /app

COPY build/libs/your-app.jar /app/app.jar

ENTRYPOINT ["java", "-jar", "/app/app.jar"]

1. **Kubernetes YAML Files**:
   * **Deployment.yaml**:
     + Defines the application deployment, specifying the number of replicas, container image, and resource limits.
     + Include labels for easy identification and targeting by other K8s resources.

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: url-text-classifier  
 labels:  
 app: url-text-classifier  
spec:  
 replicas: 2  
 selector:  
 matchLabels:  
 app: url-text-classifier  
 template:  
 metadata:  
 labels:  
 app: url-text-classifier  
 spec:  
 containers:  
 - name: url-text-classifier  
 image: your-registry/your-app:latest  
 resources:  
 requests:  
 memory: "512Mi"  
 cpu: "500m"  
 limits:  
 memory: "1Gi"  
 cpu: "1000m"  
 ports:  
 - containerPort: 8080

**Service.yaml**:

* + - Exposes the application, allowing it to be accessed within the cluster or, if configured, externally.
    - Use a ClusterIP for internal access or a LoadBalancer for external access.

apiVersion: v1  
kind: Service  
metadata:  
 name: url-text-classifier-service  
spec:  
 selector:  
 app: url-text-classifier  
 ports:  
 - protocol: TCP  
 port: 80  
 targetPort: 8080  
 type: LoadBalancer

* + **HorizontalPodAutoscaler.yaml**:
    - Configures automatic scaling based on CPU usage.
    - Ensures the application can handle varying loads by increasing or decreasing the number of pods as needed.

apiVersion: autoscaling/v2  
kind: HorizontalPodAutoscaler  
metadata:  
 name: url-text-classifier-hpa  
spec:

scaleTargetRef:  
 apiVersion: apps/v1  
 kind: Deployment  
 name: url-text-classifier  
 minReplicas: 2  
 maxReplicas: 10  
 metrics:  
 - type: Resource  
 resource:  
 name: cpu  
 target:  
 type: Utilization  
 averageUtilization: 70

1. **Monitoring and Logging**:
   * **Prometheus**: Set up Prometheus to collect application and Kubernetes metrics. It will monitor metrics like CPU/memory usage and custom application metrics if exposed.
   * **Grafana**: Use Grafana for visualizing Prometheus metrics. Dashboards can show real-time usage statistics and application performance.
   * **ELK Stack (Elasticsearch, Logstash, Kibana)** or **EFK Stack (Elasticsearch, Fluentd, Kibana)**: For logging, use Fluentd or Logstash to collect logs, Elasticsearch for indexing, and Kibana for visualization.
2. **Scaling and Load Management**:
   * With the **Horizontal Pod Autoscaler** configured, the application will automatically scale based on CPU utilization.
   * To handle large traffic spikes, consider using **Kubernetes Ingress** or a **LoadBalancer** to distribute traffic across pods.
3. **CI/CD Integration**:
   * Set up a CI/CD pipeline (e.g., GitLab CI, Jenkins, GitHub Actions) to automate the building, testing, and deployment process.
   * The pipeline will:
     + Build the Docker image and push it to the registry.
     + Apply the updated Kubernetes manifests (e.g., with kubectl apply or Helm).
     + Roll out updates with zero downtime using Kubernetes’ rolling update strategy.

**Summary**

* **Scalability**: Achieved with Horizontal Pod Autoscaler and Kubernetes’ ability to manage multiple replicas.
* **Monitoring**: Implemented using Prometheus and Grafana for metrics, with ELK/EFK for logging.
* **Deployment**: Dockerize the application and manage Kubernetes resources through YAML configurations, with CI/CD pipeline integration.