MLA// EXTENDED ASSESSMENT TASK

NAME OF SESSION	Deployment

ASSESSMENT FOCUS	Deployment Strategies

Scenario:

You are a software engineer working on a microservices-based e-commerce platform. Your team is responsible for deploying and managing these microservices using Docker and Kubernetes. You are tasked with designing a robust and scalable deployment strategy for a new microservice that handles product recommendations.

Task:

1. **Explain how you would containerize this new microservice using Docker.** Describe the steps involved in creating a Dockerfile and building a Docker image.
2. **Describe how you would deploy this containerized microservice to a Kubernetes cluster.** Explain the Kubernetes resources you would use (e.g., deployments, services, pods) and their configuration.
3. **Discuss the benefits of using Docker and Kubernetes** for deploying and managing this microservice in terms of scalability, reliability, and maintainability.
4. **Outline any limitations** of the process you have described.

Each of the 4 sections should be 200-300 words.

Deploying a Product Recommendations Microservice with Docker and Kubernetes

1. Containerising the Microservice with Docker

Containerising the product recommendations microservice begins with creating an optimised Dockerfile that packages the application and its dependencies into a portable Docker image. An effective approach utilises a multi-stage build to minimise image size and enhance security. The initial builder stage employs a complete environment, such as node:18 for Node.js or python:3.11 for Python, to compile the code, execute tests, and generate production-ready artifacts. The subsequent final stage uses a minimal base image, such as Alpine or Distroless, and copies only the necessary production artifacts from the builder stage, resulting in a smaller and more secure image.

Here are the main steps: First, write a Dockerfile and pick the images to use for building and running the app. Copy the code and install the necessary components while building it. Copy lists of needed packages before the main code, so builds are faster later. Build or get the app ready. In the final step, copy only the necessary files into the smaller image, add a user who is not an administrator for security, and set the command that will run the app.

Define a .dockerignore file to exclude unnecessary files and directories from the build context, such as node_modules, .git, or local build caches. This exclusion accelerates the

build process and produces cleaner images. Build the Docker image by running docker build -t recommendations-service:v1.0.0 ., where the -t flag sets the image name and version tag.

Test the container locally using' docker run -p 8080:8080 recommendations-service: v1.0.0' and confirm it functions correctly, responding to health checks and handling requests properly. After successful local testing, push the image to a container registry, such as Docker Hub, Amazon Elastic Container Registry (ECR), or Google Container Registry, using' docker push your-registry/recommendations-service: v1.0.0', which enables Kubernetes cluster access.

2. Deploying to Kubernetes Cluster

Deploying the containerised microservice to a Kubernetes cluster requires defining several primary Kubernetes resources: a Deployment, a Service, and optionally a Horizontal Pod Autoscaler (HPA). These resources work together to manage the application lifecycle, networking, and scalability.

The Deployment resource manages the desired state for the application, ensuring a specified number of identical Pods run at all times. Pods represent the smallest deployable unit in Kubernetes, containing one or more containers. The Deployment configuration specifies several key parameters. Set the replica count to a minimum of three for high availability (replicas: 3). The Pod template specifies the container image pulled from the registry, resource requests and limits for CPU and memory, liveness probes to restart unresponsive Pods, and readiness probes to ensure Pods receive traffic only when ready. Implement a RollingUpdate strategy to update Pods incrementally to new versions without downtime, maintaining service availability during upgrades.

The Service resource provides a stable network endpoint, as Deployment Pods are ephemeral. For internal microservices, use a ClusterIP Service, which assigns a stable internal IP address and DNS name. This enables other microservices in the e-commerce platform to communicate reliably with the recommendations service. For external access, define an Ingress resource with appropriate routing rules, TLS termination, and rate-limiting annotations.

The Horizontal Pod Autoscaler automatically adjusts the number of Deployment replicas based on observed metrics such as CPU utilisation or custom metrics like request throughput. For example, configure the HPA to scale up when CPU utilization exceeds 70 percent and scale down when utilization decreases. Configuration management utilizes ConfigMaps for non-sensitive settings and Secrets for sensitive data, which are mounted as environment variables or volumes. These resources are defined in YAML manifest files and applied using kubectl apply -f filename.yaml, ideally through GitOps tools like ArgoCD for automated, auditable deployments.

3. Benefits of Docker and Kubernetes

Docker and Kubernetes provide clear benefits for deploying and managing the product recommendations microservice. They offer strong scalability, ensure high reliability, and promote easy maintainability compared to traditional deployment methods.

Kubernetes Deployments and the Horizontal Pod Autoscaler deliver automated scalability. During periods of high demand, such as holiday sales, the HPA automatically increases the

number of running Pods to handle the increased load. Manual intervention is not required. Docker images ensure every instance is consistent, eliminating configuration drift. When demand drops, resources scale down automatically, reducing costs and maintaining efficient infrastructure.

Kubernetes's self-healing features greatly enhance reliability. Failures are detected automatically, and new Pods are created to maintain availability. Health probes reduce downtime, and rolling updates let changes deploy with zero interruption. Instant rollbacks and constant state reconciliation ensure the system stays healthy and operational.

Maintainability improves because Docker standardises environments and Kubernetes enables clear, version-controlled configuration. This minimises debugging time and streamlines collaboration across development, staging, and production. Isolated resources prevent issues in one microservice from impacting others, and centralised monitoring simplifies support and operations.

4. Limitations of the Described Process

Although robust, the described containerisation and deployment process using Docker and Kubernetes presents several limitations, particularly for newly developed microservices or smaller organisations.

Increased complexity and learning curve represent the primary challenge. Kubernetes introduces substantial abstraction and complexity beyond traditional deployment methods. Setting up and managing a production-grade cluster requires specialised expertise in container orchestration, networking concepts such as Container Network Interfaces (CNIs), Services, and Ingress controllers, as well as security policies and resource management. Configuring YAML manifests correctly demands careful attention to detail. For small teams or organisations deploying a single microservice, the initial overhead can be substantial, potentially outweighing the benefits until scale justifies the investment.

Resource overheads cannot be ignored. Whilst Docker images are considerably smaller than traditional virtual machines, operating multiple microservices in Kubernetes incurs resource overhead. Each Pod requires dedicated resources, including minimum CPU and memory allocations. The Kubernetes control plane itself consumes cluster resources through system components and pods. This increases infrastructure costs compared to single-server deployments, especially during periods of low traffic when the orchestration overhead may exceed actual application resource requirements.

State management and persistent storage add further complexity. The product recommendations service likely relies on a database or caching layer, such as Redis or PostgreSQL, to store recommendation models, user history, or real-time data. The description above addresses only the stateless application component. Managing stateful data in Kubernetes, typically through PersistentVolumes and StatefulSets, introduces additional complexity and potential performance considerations. Design these aspects carefully to maintain high performance for real-time recommendations, using proper database connection pooling and considering external managed services for critical data stores to reduce operational burden and latency concerns.

MLA// EXTENDED ASSESSMENT TASK