In-Class Assessment 2: Kubernetes Orchestration, Deployment and Cluster Management

Name: Yin Yuang
Student ID: 202383930027
Class: Software Engineering Class 1

Introduction to Cloud Computing
(Autumn,2025)

Nanjing University of Information Science and Technology, China
Waterford Institute

1. Orchestration tools, such as Kubernetes, play a key role in server infrastructure for modern applications

(a) Explain how these tools help manage and scale application servers

Kubernetes helps manage and scale application servers through the following mechanisms:

1. **Automated Operations:** Automatically handles Pod deployment, scaling, failure recovery, and rolling updates.
2. **Resource Management:** Precisely controls CPU and memory allocation using Requests and Limits.
3. **Elastic Scaling:** Supports Horizontal Pod Autoscaler (HPA) to dynamically adjust replica count based on workload.
4. **Load Balancing:** Built-in service discovery and load balancing ensure proper traffic distribution.
5. **Self-healing Capability:** Automatically detects and replaces unhealthy containers to ensure high availability.

(b) Describe how orchestration tools facilitate automated deployment, scaling, and management of application servers

Orchestration tools enable automation through:

1. **Declarative Configuration:** Uses YAML files to define desired states, with Kubernetes continuously maintaining them.
2. **Rolling Updates:** Gradually replaces old Pod versions with new ones for zero-downtime deployments.
3. **Blue-Green Deployment:** Runs old and new versions simultaneously for fast switching and rollback.
4. **Auto-scaling:** Automatically adjusts replica count based on resource utilization or metrics.
5. **Configuration Management:** Centralized ConfigMap and Secret management for configuration and sensitive data.

2. Explain the difference between a Pod, Deployment, and Service

Table 1: Comparison of Core Kubernetes Resources

Resource Type	Purpose	Lifecycle	Use Cases
Pod	Smallest deployable unit, contains one or more containers	Ephemeral, IP address may change	Running application containers
Deployment	Manages Pod replicasets, provides update and rollback capabilities	Long-lived, manages Pod lifecycle	Stateless application deployment
Service	Provides stable network endpoint and service discovery	Long-lived	Service exposure and load balancing

3. What is a Namespace in Kubernetes? Please list one example.

A **Namespace** is a mechanism for logical resource isolation within a Kubernetes cluster, used to create separate virtual environments.

Example

```
apiVersion: v1
kind: Namespace
metadata:
  name: production
```

Use Cases

- Environment isolation: development, staging, production
- Team isolation: team-a, team-b
- Project isolation: project-x, project-y

4. Explain the role of the Kubelet. How do you check the nodes in a Kubernetes cluster? (kubectl command expected).

4.1 Role of the Kubelet

- Agent running on each worker node.
- Responsible for Pod lifecycle management.
- Reports node and Pod status to API Server.
- Performs container health checks.

4.2 Commands to Check Nodes

- View all nodes: `kubectl get nodes`
- Detailed node information: `kubectl describe node <node-name>`
- Check resource usage: `kubectl top nodes`

5. What is the difference between ClusterIP, NodePort, and LoadBalancer services?

Table 2: Comparison of Kubernetes Service Types

Service Type	Access Scope	Use Cases	Configuration
ClusterIP	Internal only	Internal service communication	`type: ClusterIP`
NodePort	NodeIP:Port	Dev/testing, exposing service on every Node's IP	`type: NodePort`
LoadBalancer	Internet access via Cloud Provider Load Balancer	Cloud production exposure and managed load balancing	`type: LoadBalancer`

6. How do you scale a Deployment to 5 replicas using kubectl?

Method 1: Direct Scaling (`kubectl scale`)

- **Purpose:** This is the fastest and most direct way to change the replica count of an existing Deployment, suitable for immediate operational adjustments.
- **Command:** The command directly targets the Deployment object and patches the `replicas` field.

```
kubectl scale deployment/my-app --replicas=5
```

Method 2: Edit Deployment (`kubectl edit`)

- **Purpose:** This method is used for interactive editing of the live Deployment resource definition, allowing for modification of multiple fields simultaneously (not just replicas).
- **Command:** The command opens the resource's live YAML configuration in a text editor (like Vim or Nano). The user must manually find the `replicas` field under the `spec` section, set its value to `5`, and save the file.

```
kubectl edit deployment my-app
```

Method 3: Apply Updated Configuration File (`kubectl apply`)

- **Purpose:** This is the preferred method for workflows based on **version control (GitOps)**. The desired state is explicitly defined and tracked in a local configuration file.
- **Command:** `kubectl` compares the local file state with the cluster state and applies the changes, triggering the scaling event.

```
kubectl apply -f deployment.yaml
```

7. How would you update the image of a Deployment without downtime?

Kubernetes achieves zero-downtime updates through the Deployment's default **RollingUpdate** strategy, which gradually replaces the old ReplicaSet with a new one. Based on this mechanism, the image can be updated in several ways.

Methods Overview

First, the most direct way is using a **command-based update**:

Method 1 — Quick Command

```
kubectl blueset image deployment/my-app my-container=nginx:1.21
```

This method triggers a rolling update immediately without modifying any YAML files.

Next, for small on-the-fly adjustments, interactive editing may be used:

Method 2 — Live Editing (Not recommended for production)

```
kubectl edit deployment my-app
```

This allows modifying the image field directly in the editor; however, it cannot be version-controlled, making it unsuitable for production environments.

Finally, in real engineering and GitOps workflows, the declarative approach is preferred:

Method 3 — Declarative Update (Recommended)

Update the image field in the local `deployment.yaml` and apply the changes:

```
kubectl apply -f deployment.yaml
```

This ensures that configuration changes are traceable and auditable.

Monitoring and Rollback

After triggering the update, the rollout process can be monitored with:

```
kubectl rollout status deployment/my-app
```

To inspect version history for potential rollback:

```
kubectl rollout bluehistory deployment/my-app
```

If the new version causes issues, the Deployment can be rolled back without downtime:

```
kubectl rollout undo deployment/my-app
```

In summary, all these methods rely on Kubernetes' RollingUpdate mechanism, enabling smooth and uninterrupted

8. How do you expose a Deployment to external traffic?

Exposing a Deployment to external clients involves addressing several challenges, such as service discovery, load balancing, NAT, security, and production readiness. The following table summarizes these challenges and corresponding solutions:

Table 3: Challenges and Solutions for Exposing Kubernetes Deployments Externally

Challenge	Description	Solution
Service Discovery	External clients need a stable and reachable endpoint	Stable external IP / DNS via **Service** or **Ingress**
Load Balancing	Handling traffic distribution across multiple Pod replicas	Built-in load balancing using Service or Ingress
NAT	Pod IP not externally routable due to Network Address Translation	Use **NodePort**, **LoadBalancer**, or **Ingress**
Security	Protecting the public endpoint from various public access risks	Implement **TLS**, routing rules, and IP whitelists
Production Readiness	Ensuring the application has High Availability (HA) and scalability	Configure **Rolling Updates** and zero-downtime deployment

To address these challenges, Kubernetes offers **Ingress** as the recommended solution for HTTP/HTTPS traffic. Ingress provides Layer 7 routing, allowing multiple Services to be exposed via a single external IP or domain. It requires an **Ingress Controller**, such as:

- **Nginx Ingress Controller** — widely used and easy to deploy
- **Traefik** — supports dynamic routing and cloud-native integration
- **HAProxy** — high-performance and highly customizable
- **Istio Ingress** — integrates with service mesh for advanced traffic management
- **AWS ALB Ingress Controller** — managed load balancing for AWS environments

By combining Services and Ingress, Kubernetes can efficiently route external traffic to the appropriate Pods while handling load balancing, security, and high availability.

9. How does Kubernetes scheduling decide which node a Pod runs on?

Kubernetes Scheduler determines the optimal node for a Pod to run on, ensuring efficient resource usage, high availability, and adherence to constraints. The scheduler evaluates several factors:

- **Resource requirements:** CPU, memory, and GPU requests/limits are considered to match Pods to suitable nodes.
- **Node selectors (labels):** Pods can specify node labels to control placement on specific nodes.

- **Affinity/Anti-affinity rules:** Influence scheduling based on Pod co-location preferences or avoidance.
- **Taints and tolerations:** Nodes can repel Pods unless they tolerate specific taints.
- **Resource availability and constraints:** Scheduler ensures nodes have enough capacity and obey cluster-level policies.

Overall, the scheduler evaluates these factors to place Pods efficiently while meeting workload and cluster requirements.

10. What is the role of Ingress and how does it differ from a Service?

In Kubernetes, an **Ingress** is a resource that manages external access to Services, typically over HTTP or HTTPS. Its primary role is to provide **Layer 7 (application-level) routing**, enabling multiple Services to be exposed under a single IP or domain name. Ingress can perform advanced traffic management functions such as:

- **Domain and path-based routing:** Direct requests to different Services based on hostnames or URL paths.
- **SSL/TLS termination:** Handle encryption at the Ingress level, simplifying certificate management.
- **Virtual hosting:** Serve multiple domains using the same Ingress Controller.
- **Load balancing:** Distribute traffic across multiple backend Pods.
- **Centralized access control:** Apply rules for authentication, whitelisting, or rate-limiting.

To function, an Ingress requires an **Ingress Controller**, which implements the routing rules and handles incoming traffic. Popular controllers include Nginx, Traefik, HAProxy, and cloud-managed options like AWS ALB Ingress Controller.

Difference Between Ingress and Service

While both Ingress and Service expose Pods to external or internal traffic, they operate at different network layers and serve distinct purposes:

Feature	**Service**	**Ingress**
Network Layer	L4 (TCP/UDP)	L7 (HTTP/HTTPS)
Routing	Port-based	Hostname/path-based routing
SSL/TLS	Requires manual configuration	Built-in termination support
External Access	NodePort / LoadBalancer	Single IP with multiple Services via rules
Use Case	Internal or basic external access	Fine-grained external traffic management

Table 4: Comparison between Service and Ingress

In summary, Services provide stable networking for Pods and basic load balancing, while Ingress offers application-layer routing, centralized traffic management, and secure external access for multiple Services.