| **Orchestration with Kubernetes** |
| --- |

Date: June 1st 2024

Revision: 1.0

| **SOW Reference** | TEAM # 1 /2024-06/001 | **Professor** | **Yang Sun**  [**Yangsun.com@gmail.com**](mailto:Yangsun.com@gmail.com) |
| --- | --- | --- | --- |
| **Project Title** | **TEAM # 1**  **Part # 4**  **Orchestration with Kubernetes** |
| **Team Members Email ID** | **Yilun Jin**  **Maheswar Barrenkala**  **Akshat Jangam**  **Atilla Karaali**  **Shital Hinge**  **Jean Qiong Zhang**  **Caihui Yang**  **Dhruv Basnotra**  **Xu Liu** | **Team Members Email ID** | [**karen.jin.job@gmail.com**](mailto:karen.jin.job@gmail.com)  [**maheswar.barrenkala@gmail.com**](mailto:maheswar.barrenkala@gmail.com)  [**akshat.ja@gmail.com**](mailto:akshat.ja@gmail.com)  [**atilla.karaali@gmail.com**](mailto:atilla.karaali@gmail.com)  [**Shitalkarleg@gmail.com**](mailto:Shitalkarleg@gmail.com)  [**jzhang95014@gmail.com**](mailto:jzhang95014@gmail.com)  [**ashleyyang924@gmail.com**](mailto:ashleyyang924@gmail.com)  [**dhruvbasnotra@gmail.com**](mailto:dhruvbasnotra@gmail.com)  **liuxubupt16@gmail.com** |

| **Orchestration with Kubernetes**  **Objective**  The primary objective of this section is to illustrate how Kubernetes can be utilized to manage and scale a web application efficiently. Kubernetes, a robust container orchestration tool, enables automated deployment, scaling, and management of containerized applications, ensuring high availability, scalability, and manageability.  **Why Kubernetes is Needed**  Kubernetes is indispensable for deploying and managing large-scale applications due to its powerful orchestration capabilities. It simplifies many aspects of managing microservices and containerized applications by providing:   * **Automated rollouts and rollbacks:** Ensuring updates and changes are systematically and safely propagated. * **Service discovery and load balancing:** Automatically assigning IP addresses to containers and a single DNS name for a set of containers, and load balancing traffic between them. * **Storage orchestration:** Automatically mounting the storage system of choice, whether from local storage, public cloud providers, or network storage systems. * **Self-healing:** Restarting failed containers, replacing and rescheduling containers when nodes die, killing containers that don't respond to user-defined health checks, and not advertising them to clients until they are ready to serve.   **Kubernetes Deployment YAML Files and Explanations**  **1. Kubernetes Deployment for Web Application**  File: web-deployment.yaml  apiVersion: apps/v1  kind: Deployment  metadata:  name: web-deployment # Name of the deployment  spec:  replicas: 3 # Number of replicas  selector:  matchLabels:  app: web # Label selector that determines which pods belong to the deployment  template:  metadata:  labels:  app: web # Labels applied to all pods spawned from this deployment  spec:  containers:  - name: web # Name of the container  image: my-web-image:latest # Container image  ports:  - containerPort: 5000 # Port the container exposes  This deployment configures three replicas of the web application, ensuring high availability. Each replica is a pod containing a container that runs the web application.  **2. Kubernetes Deployment for Database**  File: db-deployment.yaml  apiVersion: apps/v1  kind: Deployment  metadata:  name: db-deployment # Name of the deployment  spec:  replicas: 1 # Number of replicas  selector:  matchLabels:  app: db # Label selector for the deployment  template:  metadata:  labels:  app: db # Labels applied to the pods  spec:  containers:  - name: db # Name of the container  image: mysql:5.7 # Container image  env: # Environment variables for MySQL  - name: MYSQL\_ROOT\_PASSWORD  value: "rootpassword"  - name: MYSQL\_DATABASE  value: "mydatabase"  - name: MYSQL\_USER  value: "myuser"  - name: MYSQL\_PASSWORD  value: "mypassword"  ports:  - containerPort: 3306 # MySQL port  volumeMounts:  - name: db-storage # Mounting point for the persistent volume  mountPath: /var/lib/mysql  volumes:  - name: db-storage # Persistent volume claim  persistentVolumeClaim:  claimName: db-pvc  This deployment ensures the database is running within a single pod but remains highly available through persistent storage and environmental configurations that manage the state.  **3. Kubernetes Service for Web Application**  File: web-service.yaml  apiVersion: v1  kind: Service  metadata:  name: web-service # Name of the service  spec:  selector:  app: web # Connects the service to pods with this label  ports:  - protocol: TCP  port: 80 # Port the service exposes  targetPort: 5000 # Port on the pod to forward to  ---  apiVersion: networking.k8s.io/v1  kind: Ingress  metadata:  name: web-ingress # Name of the ingress resource  spec:  rules:  - host: myapp.example.com  http:  paths:  - path: /  pathType: Prefix  backend:  service:  name: web-service  port:  number: 80    This YAML file defines a service that routes traffic to the web application pods and configures an ingress resource for managing external access to the services within the cluster, mapping myapp.example.com to the internal service.  **4. Kubernetes Persistent Volume Claim for Database**  File: db-pvc.yaml  apiVersion: v1  kind: PersistentVolumeClaim  metadata:  name: db-pvc # Name of the PVC  spec:  accessModes:  - ReadWriteOnce # The PVC is mounted as read-write by a single node  resources:  requests:  storage: 20Gi # Amount of storage requested  This persistent volume claim ensures that the database storage remains consistent and available, even if the pod managing the database is rescheduled to a different node.  **Explanation of Kubernetes Concepts Used**   * **Deployments** manage the deployment and scaling of a set of Pods, and provide declarative updates to applications. * **Services** abstract the way to access a set of running Pods, providing a consistent address and handling the load balancing. * **Persistent Volume Claims (PVCs)** provide an abstract storage layer which the Pods consume without being aware of the underlying storage architecture.   **Description of the Deployment and Scaling Strategy**  The deployment strategy involves using multiple replicas for the web application to ensure redundancy and high availability. The database, being a critical but singular component, is deployed with a single replica but backed by a Persistent Volume to ensure data persistence across pod rescheduling and failures. The scaling strategy will rely on Kubernetes’ ability to monitor resource usage and automatically scale the number of web application pods using Horizontal Pod Autoscalers, according to predefined metrics such as CPU and memory usage.  **Explanation of Kubernetes Concepts Used**  **Deployments**: In Kubernetes, a Deployment provides declarative updates for Pods and ReplicaSets. It allows you to describe the desired state of your application, and the Deployment Controller changes the actual state to the desired state at a controlled rate. This enables you to easily manage application scaling and deployment without needing to handle the orchestration code.  **Services**: A Kubernetes Service is an abstraction which defines a logical set of Pods and a policy by which to access them. Services enable a loose coupling between dependent Pods. A Service routes traffic across a set of Pods. Services are discovered by Kubernetes DNS service or environment variables available in any Pod.  **Scaling**: Kubernetes facilitates both manual and automatic scaling. Manual scaling adjusts the number of running replicas in a Deployment at any given time. Automatic scaling, or autoscaling, adjusts the number of Pod replicas in a Deployment based on CPU usage or other select metrics.  **Description of the Deployment and Scaling Strategy**  The deployment strategy for the web application involves creating a Deployment that maintains three replicas of the application pods to ensure high availability. For the database, a single-replica Deployment is used, focusing on persistence and data integrity through a connected Persistent Volume Claim, ensuring data is not lost when the pod is restarted or moved.  Scaling the application dynamically is a critical part of ensuring responsiveness under varying loads. The scaling strategy utilizes Kubernetes' Horizontal Pod Autoscaler (HPA) to automatically adjust the number of web application pods based on observed CPU utilization or other specified metrics. This auto-scaling ensures that the application can handle increases in traffic without manual intervention, maintaining performance and reducing costs during low-traffic periods by scaling down.  Together, these Kubernetes components and strategies provide a robust framework for managing the lifecycle and scalability of containerized applications. The orchestration capabilities of Kubernetes not only simplify deployment and scaling but also enhance the resilience and efficiency of applications in production environments.    END  END |
| --- |