In-Class Assessment 2

Sun Yixuan202383930020

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

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

- Orchestration tools manage and scale application servers through core capabilities like unified management, resource abstraction, and fault self-healing.
- They abstract underlying physical or virtual machine resources into schedulable units, allowing administrators to specify application requirements through declarative configurations instead of manually assigning applications to specific servers. The tools automatically allocate appropriate nodes from the cluster resource pool to avoid resource waste or overload.
- They continuously monitor the health status of application servers or containers. If issues such as crashes, unresponsiveness, or resource exhaustion occur, they automatically restart instances on other healthy nodes within the cluster to ensure service availability.
- They support metric-triggered dynamic scaling or manual adjustment of instance counts. When scaling out, new instances are automatically scheduled to idle nodes with relevant dependencies configured; when scaling in, redundant instances are safely terminated to release resources.
- For distributed applications, they can manage inter-instance communication and storage mounting across multiple servers, simplifying deployment complexity.

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

- For automated deployment, users define the desired final state through

declarative configurations. The tool's controllers compare the desired state with the current cluster state and automatically execute deployment steps such as pulling images and creating containers, supporting rolling deployments to avoid service interruptions.

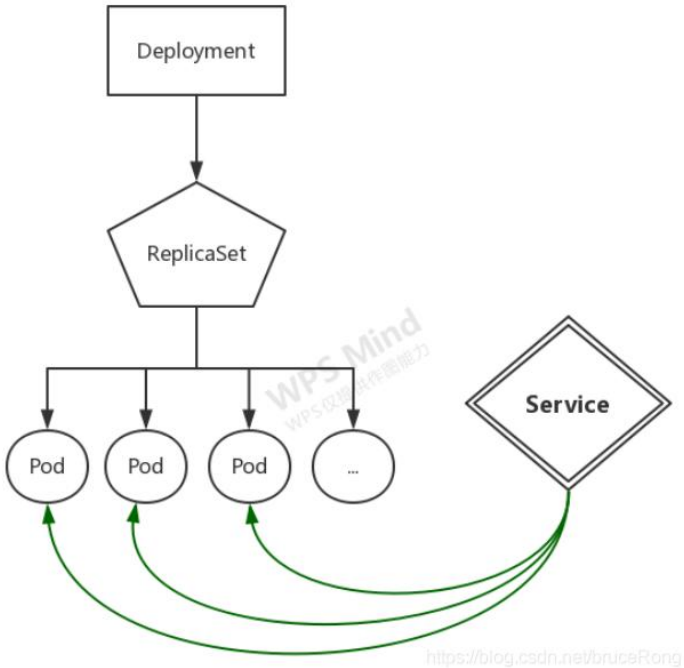
- Automated scaling includes both static and dynamic methods. Static scaling involves users modifying the desired number of replicas, with controllers automatically scheduling new instances. Dynamic scaling uses Horizontal Pod Autoscalers to monitor application metrics and automatically adjust replica counts.
- For automated management, probes regularly detect instance status, restarting failed instances and temporarily removing unready ones from service access lists. Users can configure resource requests and limits for instances, with tools allocating resources accordingly and preventing resource abuse. They also support integration with monitoring and logging tools to automatically collect operational metrics and logs for troubleshooting.

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

- Pod
    - The smallest deployment unit in Kubernetes, which can contain one or more tightly coupled containers, such as an application container and a log collection container.
    - Has a short and volatile lifecycle, with possible IP changes after restart and no automatic recreation after deletion.
    - Mainly functions as a container carrier for running application instances, generally not directly operated by users but managed by other components.
- Deployment
    - Used to manage the lifecycle of Pods, including deployment, updates, and scaling.
    - Does not directly contain Pods but manages their creation and destruction through controllers, capable of automatically recreating failed Pods to ensure the number of replicas meets expectations.

    - Serves as a user-oriented deployment entry point. Users manage applications through Deployments, whose core functions include declarative Pod deployment, rolling updates/rollbacks, and static or dynamic scaling.

- Service
    - Provides a stable network access entry for Pods with load balancing and service discovery capabilities.
    - Does not contain Pods but associates matching Pods through label selectors, with persistence—its IP and access method remain unchanged even if all associated Pods are recreated.
    - Acts as an access entry for users or other services, providing a fixed ClusterIP for internal cluster access, distributing requests to associated Pods, and supporting access via service names without needing to remember Pod IPs.



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

- A Namespace in Kubernetes is a logical unit for resource isolation and grouping, dividing cluster resources such as Pods, Deployments, and Services into different virtual clusters.

- Its core objectives include:
    - Resource isolation, allowing duplicate resource names in different Namespaces to avoid naming conflicts.
    - Permission control, enabling RBAC configuration based on Namespaces, such as restricting developers to only operate resources in the dev Namespace.
    - Resource quotas, allowing setting resource limits for each Namespace to prevent a single team from occupying excessive cluster resources.
- Example: The default Namespace created by default in a Kubernetes cluster. All resources not assigned to a specific Namespace, such as Pods created with:
  kubectl run nginx --image=nginx
  are automatically assigned to the default Namespace, which can be viewed using the command:
  kubectl get pods -n default.

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

- Role of the Kubelet
    - A core component running on every cluster node (both Master and Worker nodes), acting as the node's manager.
    - Executes Pod scheduling instructions, listening to Pod scheduling requests sent by the Kubernetes API Server and calling the container runtime to create and start containers according to Pod configurations.
    - Maintains Pod health status, regularly checking container status using probes defined in Pod specifications—automatically restarting crashed containers and notifying the API Server to remove unready ones from the Service load balancing list.
    - Continuously collects resource usage of the current node and the running status of Pods, reporting them to the API Server to ensure the cluster control plane has real-time knowledge of the entire cluster's status.
- kubectl command to check nodes in a cluster

- Use the kubectl get nodes command to view basic information about all nodes in the cluster, including node names, statuses, roles, and versions.
- To view detailed node information such as resource usage, labels, and taints, use the kubectl describe nodes <node name> command, for example, kubectl describe nodes worker-1.

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

- ClusterIP
  - Provides a unique virtual IP within the cluster, used only for internal cluster communication.
  - Access is limited to Pods and other Services within the cluster.
  - Ports are automatically assigned within the default range 10.96.0.0/12.
  - Suitable for communication between services within the cluster (e.g., microservice A calling microservice B) and cannot be accessed from outside the cluster.
- NodePort
  - Builds on ClusterIP by opening a fixed port on each node, with ports in the default range 30000-32767 or manually specified.
  - Accessible both inside and outside the cluster, with external access via "node IP:NodePort".
  - Suitable for external access in test environments or small-scale scenarios (e.g., developers temporarily accessing test services).
- LoadBalancer
  - Builds on NodePort and can automatically integrate with cloud provider load balancers (e.g., AWS, Alibaba Cloud), with external access via the load balancer's public IP.
  - Load balancer ports are assigned by the cloud provider (e.g., port 80 for HTTP, 443 for HTTPS).
  - Suitable for external access in production environments (e.g., public users accessing web applications).

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

- To scale a Deployment to 5 replicas, use the kubectl scale command, which adjusts the desired number of replicas in the Deployment configuration. The format is:

  kubectl scale deployment <deployment name> --replicas=5

- For example, if scaling a Deployment named "nginx-deploy", run:

  kubectl scale deployment nginx-deploy --replicas=5

- After execution, verify the scaling result with the following command, which shows the current state of the Deployment:

  kubectl get deployment nginx-deploy

- Check that both the "DESIRED" (target number of replicas) and "READY" (number of fully operational replicas) columns display 5, confirming the scaling was successful.

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

- Kubernetes supports rolling updates for Deployments, which gradually replace old Pods with new ones to avoid service interruptions. Two common methods to perform this update are:

Method 1: Direct image update with kubectl set image

- This command directly modifies the Deployment to use the new image, triggering an automatic rolling update. The format is:

  kubectl set image deployment <deployment name> <container name>=<new image address:tag>

- For example, to update the "nginx" container in the "nginx-deploy" Deployment from nginx:1.21 to nginx:1.23, run:

  kubectl set image deployment nginx-deploy nginx=nginx:1.23

Method 2: Edit the Deployment configuration

- Open the Deployment's YAML configuration file for editing with:

  kubectl edit deployment <deployment name>

- Locate the spec.template.spec.containers[0].image field (specifies the current image) and replace the old image address with the new one.
- Save and close the file—Kubernetes detects the configuration change and starts a rolling update automatically.

Monitoring the update process

- Track the update in real time with:

  kubectl get pods -w

- This command shows old Pods being terminated and new Pods starting. Ensure the "READY" count never drops to 0, confirming the service remains available throughout the update.

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

- To expose a Deployment to external traffic, create a Kubernetes Service that routes external requests to the Deployment's Pods. The choice of Service type (NodePort or LoadBalancer) depends on the environment:

Scenario 1: Test or small-scale environments (NodePort)

- A NodePort Service exposes the Deployment on a fixed port across all cluster nodes, allowing external access via any node's IP. Create it with:

  kubectl expose deployment <deployment name> --type=NodePort --port=<container port> --target-port=<actual port inside the container>

- For example, to expose the "nginx-deploy" Deployment (which uses port 80 internally), run:

  kubectl expose deployment nginx-deploy --type=NodePort --port=80 --target-port=80

- Retrieve the assigned NodePort (a port in the 30000-32767 range) with:

  kubectl get service nginx-deploy

- Access the service externally using the IP of any cluster node and the NodePort:

http://<cluster node IP>:<NodePort>

Scenario 2: Production environments (LoadBalancer)

- A LoadBalancer Service integrates with cloud provider load balancers (e.g., AWS ELB, GCP Load Balancer) to provide a stable public IP. Create it with:

  kubectl expose deployment <deployment name> --type=LoadBalancer --port=<container port> --target-port=<actual port inside the container>

- For example, to expose "nginx-deploy" via a cloud load balancer:

  kubectl expose deployment nginx-deploy --type=LoadBalancer --port=80 --target-port=80

- Get the public IP assigned by the cloud provider (listed under EXTERNAL-IP) with:

  kubectl get service nginx-deploy

- Access the service externally using this public IP:

  http://<EXTERNAL-IP>:<port>

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

- Kubernetes scheduling is handled by kube-scheduler, with core logic involving two steps: filtering and scoring, ensuring Pods are scheduled to the most suitable nodes.
- Filtering phase: Eliminates nodes that don't meet the criteria. The scheduler filters out nodes that can run the Pod from all nodes based on the Pod's scheduling requirements. If no nodes meet the criteria, the Pod remains in "Pending" state. Common filtering criteria include:
  - Resource satisfaction: The remaining CPU, memory, and other resources on the node are not less than the Pod's resource requests.
  - Node affinity: The Pod declares a desire or requirement to run on nodes with specific labels, retaining only nodes matching the labels.
  - Taints and tolerations: Nodes with taints will only have Pods scheduled to them if the Pods declare toleration for those taints.
  - Port conflicts: If the Pod needs to use host networking, no other Pod on the

node can occupy the same port.

    - Storage requirements: The PersistentVolume mounted by the Pod must match the node's storage type and availability zone.

- Scoring phase: Selects the optimal node from the candidate nodes. After filtering, a list of candidate nodes is obtained, and the scheduler scores each candidate node (0-100 points), with the highest-scoring node being the final scheduling target. Common scoring criteria include:
    - Balanced resource utilization: Preferring nodes with more balanced CPU and memory utilization.
    - Node affinity weights: Nodes matching the Pod's declared preference for certain node types receive extra points.
    - Pod affinity/anti-affinity: Adjusting node scores based on declarations.
    - Taint toleration: Nodes with taints that Pods only tolerate (not prefer) are penalized.
- Special cases:
    - Users can directly specify a node in the Pod configuration using the nodeName field, bypassing the scheduler's filtering and scoring. However, this static scheduling method is not recommended as it reduces scheduling flexibility.
    - Third-party schedulers like Volcano can be deployed to meet specific scenario requirements.

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

- Role of Ingress
    - A component in Kubernetes for managing entry rules for external access to services within the cluster, acting as the cluster's reverse proxy and URL routing manager.
    - Addresses limitations of Services, such as Services' inability to route based on URL paths or domain names, and the need to create multiple LoadBalancers when exposing multiple Services externally.

    - Core functions include forwarding external requests to different Services based on URL paths and domain names; centrally managing SSL certificates at the Ingress layer, decrypting HTTPS requests before forwarding to backend Services; optimizing request distribution by combining with the load balancing capabilities of backend Services; and supporting advanced features like path rewriting and request rate limiting in some Ingress controllers (e.g., Nginx Ingress Controller).
    - Note that Ingress itself is a rule configuration and requires an Ingress Controller (e.g., Nginx Ingress Controller, Traefik) to take effect. Ingress Controllers monitor Ingress rules, convert them into reverse proxy configurations, and actually handle external requests.
- Differences between Ingress and Service
    - Core positioning: Ingress is a rule manager for external access, handling HTTP/HTTPS routing; Service is a network access entry for services, providing load balancing and service discovery.
    - Dependencies: Ingress relies on an Ingress Controller to function and must associate with Services (requests are ultimately forwarded to Pods through Services); Service directly associates with Pods and doesn't require other components except for cloud provider LoadBalancers.
    - Routing capabilities: Ingress supports fine-grained routing based on domain names and URL paths (e.g., multi-domain and multi-path mapping); Service only supports port-based routing (one Service corresponds to one set of ports and cannot distinguish paths or domains).
    - Protocol support: Ingress mainly supports HTTP/HTTPS (with partial support for TCP/UDP); Service supports various protocols like TCP, UDP, and SCTP.
    - External access methods: Ingress needs to be exposed through an Ingress Controller (usually one LoadBalancer Service), with external access only requiring the Ingress's domain name/IP; Service needs to be exposed through NodePort or LoadBalancer, with each Service potentially corresponding to one external port/IP.
    - Applicable scenarios: Ingress is suitable for multiple services sharing one external entry point (e.g., multiple web services in a cluster needing access through different domain names/paths); Service is suitable for independent access entry points for single services (e.g., database services, internal microservices).