**Blue-Green Deployment in Kubernetes**

**Blue-Green Deployment** involves maintaining two separate environments:

* One is the **Blue** environment (current live version).
* The other is the **Green** environment (new version).

**Steps for Blue-Green Deployment:**

1. **Create the Blue Deployment** (current version of the application):
   * This is the initial application (v1).
   * Example YAML for the **Blue** deployment:

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-app-blue

spec:

replicas: 3

selector:

matchLabels:

app: my-app

version: blue

template:

metadata:

labels:

app: my-app

version: blue

spec:

containers:

- name: my-app

image: my-app:v1

1. **Create a Service to Point to the Blue Environment:**

yaml

Copy code

apiVersion: v1

kind: Service

metadata:

name: my-app-service

spec:

selector:

app: my-app

version: blue

ports:

- protocol: TCP

port: 80

targetPort: 8080

1. **Deploy the Green Version (new version of the application):**
   * Update the deployment to create the **Green** version (v2).

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-app-green

spec:

replicas: 3

selector:

matchLabels:

app: my-app

version: green

template:

metadata:

labels:

app: my-app

version: green

spec:

containers:

- name: my-app

image: my-app:v2

1. **Switch the Service to Point to Green:**
   * After verifying that the **Green** environment is functioning as expected, you can update the Service to point to the new version.

yaml

Copy code

apiVersion: v1

kind: Service

metadata:

name: my-app-service

spec:

selector:

app: my-app

version: green

ports:

- protocol: TCP

port: 80

targetPort: 8080

1. **Delete or Scale Down the Blue Deployment:**
   * After traffic is directed to the **Green** environment, you can either delete the **Blue** deployment or scale it down.

**Canary Deployment in Kubernetes**

**Canary Deployment** involves gradually releasing the new version (v2) of your application to a small subset of users, monitoring it, and gradually increasing the traffic.

**Steps for Canary Deployment:**

1. **Deploy the Current Version (v1):**
   * This is your stable version of the app, initially receiving 100% of the traffic.

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-app-stable

spec:

replicas: 5

selector:

matchLabels:

app: my-app

version: stable

template:

metadata:

labels:

app: my-app

version: stable

spec:

containers:

- name: my-app

image: my-app:v1

1. **Deploy the Canary Version (v2) with Lower Replicas:**
   * This will serve a smaller portion of traffic (e.g., 10%).

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-app-canary

spec:

replicas: 1 # Smaller number of replicas initially

selector:

matchLabels:

app: my-app

version: canary

template:

metadata:

labels:

app: my-app

version: canary

spec:

containers:

- name: my-app

image: my-app:v2

1. **Use a Load Balancer to Split Traffic:**
   * A traffic management tool like **Istio**, **NGINX**, or **Traefik** can help route traffic between the stable and canary versions.

For **Istio**, you can define a VirtualService to split traffic between versions.

yaml

Copy code

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: my-app

spec:

hosts:

- my-app-service

http:

- route:

- destination:

host: my-app-stable

weight: 90 # 90% of traffic to stable

- destination:

host: my-app-canary

weight: 10 # 10% of traffic to canary

1. **Monitor the Canary Deployment:**
   * Monitor metrics like error rates, response times, and logs to ensure the canary version (v2) is working as expected.
2. **Gradually Increase Traffic to Canary:**
   * If the canary deployment is performing well, gradually increase the traffic by adjusting the VirtualService traffic weights (e.g., 50-50 split).
3. **Full Rollout:**
   * After successful testing, you can fully switch to the new version by directing all traffic to **v2** and scaling down or removing the **v1** pods.

yaml

Copy code

http:

- route:

- destination:

host: my-app-canary

weight: 100 # 100% traffic to v2

**Tools You Can Use:**

1. **Istio**: A service mesh that allows you to split traffic between services (used in Canary deployment).
2. **Argo Rollouts**: A Kubernetes controller that provides advanced deployment strategies like Blue-Green and Canary with a user-friendly interface.
3. **NGINX**: An ingress controller that can be used to split traffic for Canary deployments.
4. **Traefik**: A reverse proxy and load balancer for microservices, suitable for traffic splitting.

Both methods offer controlled and safe ways to introduce new versions, with **Blue-Green** focusing on quick, binary switching between environments and **Canary** enabling gradual traffic shift to monitor stability.

4o

[Jenkins-Zero-To-Hero/Interview\_Questions.md at main · nikhil95malempati/Jenkins-Zero-To-Hero (github.com)](https://github.com/nikhil95malempati/Jenkins-Zero-To-Hero/blob/main/Interview_Questions.md)

Thinbackup plugin can be used to backup jenkins

Shared library in Jenkins

In manage Jenkins -> system-> global pipeline libraries there we give repo name and shared library searches for var folder in repo and var folder will have groovy scripts

Folder inside var has to end with .grrovy extension

K8s architecture

 **Node**: A machine that runs containerized applications.

 **Pod**: The smallest deployable unit in Kubernetes, usually containing one or more containers.

 **Cluster**: A set of nodes working together managed by Kubernetes.

 **Master Node**: Controls the Kubernetes cluster and manages worker nodes.

 **Kubelet**: An agent that runs on each node to ensure containers are running in pods.

 **Kube-Proxy**: Manages network communication and load balancing for services. (like ip:nodeport it takes traffic to pod via service)

 **ETCD**: A distributed key-value store that holds cluster data.

 **Scheduler**: Assigns pods to available nodes based on resource requirements.

 **Controller Manager**: Manages controllers that handle tasks like scaling and updates.

 **API Server**: Frontend that handles communication between users and the cluster.

 **ClusterIP**: Exposes the service only within the cluster.

 **NodePort**: Exposes the service on each node’s IP at a static port.

 **LoadBalancer**: Exposes the service externally using a cloud provider’s load balancer.

 **ExternalName**: Maps the service to an external DNS name without proxying traffic.

 **Headless Service**: Directly exposes pods without load balancing or cluster IP, used for stateful applications.

--

Here’s the difference between **LoadBalancer** and **NodePort** in Kubernetes:

1. **NodePort**:
   * Exposes a service on a static port on each node's IP (the same port across all nodes).
   * You can access the service using NodeIP:NodePort from outside the cluster.
   * Requires manual load balancing if you want to distribute traffic across nodes.
2. **LoadBalancer**:
   * Automatically provisions an external load balancer (usually from a cloud provider).
   * Routes external traffic to the service and handles load balancing across the nodes.
   * Simplifies external access, providing a single IP address that routes traffic evenly.

To manually load balance a **NodePort** service in Kubernetes, you need to configure an external load balancer or a DNS-based solution to distribute traffic across multiple node IPs. Here are the steps to do it:

**1. Get Node IPs and NodePort**

* First, get the NodePort and the IP addresses of the cluster nodes:

bash

Copy code

kubectl get svc my-service

kubectl get nodes -o wide

This will give you the NodePort (e.g., 30000) and node IPs (e.g., NodeIP1, NodeIP2).

**2. Configure an External Load Balancer**

You can set up an external load balancer (e.g., HAProxy, NGINX) to distribute traffic across the nodes.

**Example: Using NGINX as a Load Balancer**

* Install NGINX on an external machine.
* Configure the nginx.conf file to route traffic to each node’s IP on the NodePort:

nginx

Copy code

http {

upstream k8s\_nodes {

server NodeIP1:30000;

server NodeIP2:30000;

# Add more node IPs if necessary

}

server {

listen 80;

location / {

proxy\_pass http://k8s\_nodes;

}

}

}

* Restart NGINX to apply the configuration.

Now, NGINX will distribute traffic between the nodes, effectively load balancing the requests across the NodePort services.

**3. DNS-based Load Balancing**

Another method is using a DNS-based approach, where you configure a DNS server to resolve a domain name to multiple node IP addresses, spreading traffic randomly between them.

* Use a DNS provider (e.g., AWS Route 53) to configure a DNS entry with multiple IP addresses (the node IPs). Traffic will be balanced based on DNS resolution.

In the first point:

**"Automatically provisions an external load balancer (usually from a cloud provider)" means:**

When you create a Kubernetes **LoadBalancer** service, Kubernetes integrates with cloud providers like AWS, GCP, or Azure. It automatically requests and provisions a cloud-based load balancer for your service without you having to manually set it up.

* For example, in AWS, Kubernetes will create an **Elastic Load Balancer (ELB)**, in GCP a **Google Cloud Load Balancer**, or in Azure an **Azure Load Balancer**.
* This load balancer will expose your Kubernetes service to the internet, assigning it an external IP address (provided by the cloud provider).

The benefit is that you don’t need to manually configure or manage the load balancer—it’s handled automatically by the Kubernetes control plane in coordination with the cloud provider’s APIs.

K8S security

By defauly pod communicates with another pod we can setup network policies to limit this communication between pods. If u have only one node and using docker desktop for k8s it might not work as docker has basic cni

RBAC

Namespaces for multi tenancy like diff app have ns

Store secrets In vault

Init container will run and complete before the startup of main container in the pod.

Ex – sleep or mounting a volume b4 actual container is starting in the pod

an init container is used to wait for a database to be ready before starting the main application.

apiVersion: v1

kind: Pod

metadata:

name: my-app-pod

spec:

containers:

- name: my-app-container

image: my-app-image:latest

ports:

- containerPort: 8080

env:

- name: DB\_HOST

value: "my-database"

# Main application container

initContainers:

- name: init-container

image: busybox

command: ['sh', '-c', 'until nc -z my-database 5432; do echo waiting for db; sleep 2; done;']

# Init container checks if the database is ready on port 5432 before the app container starts

**Pod Disruption Budget (PDB)**

A **Pod Disruption Budget (PDB)** in Kubernetes ensures a minimum number of pods remain available during voluntary disruptions, like node drains or upgrades.

**Key Points:**

* **Purpose:** To maintain application availability during planned disruptions.
* **Types:**
  + **minAvailable:** Minimum pods that must remain running.
  + **maxUnavailable:** Maximum pods that can be unavailable during disruptions.

**Example Scenario**

If you have a deployment with **5 replicas** and a PDB set to minAvailable: 2, at least **2 pods** must always be operational.

**During a Node Drain:**

* If all pods are on a single node and you drain it, Kubernetes will ensure that 2 pods stay running, delaying the eviction until it can maintain the required availability, or blocking the eviction if no other nodes are available.

Side car container

apiVersion: apps/v1

kind: Deployment

metadata:

name: web-app-deployment

labels:

app: web-app

spec:

replicas: 2

selector:

matchLabels:

app: web-app

template:

metadata:

labels:

app: web-app

spec:

containers:

- name: web-app

image: my-web-app:latest

ports:

- containerPort: 80

volumeMounts:

- name: log-volume

mountPath: /var/log/web-app # The path where the application writes logs

- name: logging-sidecar

image: fluent/fluentd:v1.14-1 # Fluentd as the logging agent

env:

- name: FLUENTD\_CONF

value: "fluent.conf" # Configuration for Fluentd

volumeMounts:

- name: log-volume

mountPath: /var/log/web-app # Mounting the same volume to read logs

ports:

- containerPort: 24224 # Default port for Fluentd

volumes:

- name: log-volume

emptyDir: {} # Temporary storage for logs

A **sidecar container** is a helper container that runs alongside the main application container in the same pod in Kubernetes. It provides additional functionality without modifying the main application itself. Here’s a simple explanation:

**Key Points About Sidecar Containers:**

1. **Companion**: The sidecar container is like a companion to the main application container. They share the same network namespace and can communicate easily.
2. **Functionality**: It adds features such as logging, monitoring, or managing configuration, which helps improve the overall capabilities of the application.
3. **Shared Storage**: Sidecars often use shared storage (volumes) to access data generated by the main application, like logs.
4. **Examples**:
   * **Logging Agent**: A sidecar might collect and forward logs from the main application to a logging service.
   * **Proxy**: It could act as a reverse proxy to manage traffic to the main application.

**Secrets Management on AWS**

1. Systems Manager
2. Secrets Manager
3. Hashicorp Vault

**1. AWS Systems Manager (SSM) Parameter Store**

* **Best for**: Simple secrets management and parameter storage, where there’s less sensitivity around access controls and encryption.
* **Features**:
  + **Stores plaintext and encrypted strings as parameters.** (like info not that highly secure like docker registry url)
  + Integrates well with other AWS services like Lambda and EC2.
  + Supports versioning, tagging, and hierarchy for parameters.
* **Use Cases**:
  + Managing environment variables, API keys, and application settings in a straightforward, low-cost way.
  + When a basic level of secrets management with native AWS integrations is sufficient.

**2. AWS Secrets Manager**

* **Best for**: Advanced secrets management within the AWS ecosystem, especially for database credentials, API keys, and other sensitive data that need frequent rotation.
* **Features**:
  + **Automatic secrets rotation for supported services like RDS databases**. (like rds password rotation)
  + Fine-grained access control via IAM.
  + Auditing capabilities with AWS CloudTrail.
* **Use Cases**:
  + When you need frequent, automated rotation of secrets.
  + Managing secrets for database credentials, API keys, and third-party service credentials where encryption and auditability are critical.

**3. HashiCorp Vault**

* **Best for**: Complex, multi-cloud or hybrid environments, or when you require high customization, extensive security, and dynamic secrets generation.
* **Features**:
  + Platform-agnostic, supporting multiple cloud providers and on-premises infrastructure.
  + Offers dynamic secrets, generating credentials on-demand with limited time-to-live.
  + Role-based access control (RBAC), encryption, audit logging, and extensive plugin support.
* **Use Cases**:
  + When you need to manage secrets across different environments (cloud and on-premises).
  + Scenarios requiring high-security needs, such as dynamically generating database credentials for each access request.
  + When the organization needs advanced auditing, access policies, and controls beyond AWS’s capabilities.