**Understanding Document**

**V.NIKHIL KUMAR**

**BALAJI. R**

**Problem Statement:**

Using Kubernetes, we need to create a continuous integration and continuous delivery pipeline for Word press CMS and Should address below Concern:

· High Availability

· Multiple Environments

· Zero downtime deployment

· Feature branch deployment

· Rollback strategies

· Observability

· Alerts and Notifications

**Deploying Wordpress in Amazon EKS from docker image.**

* Create a docker image of the web application
* Push our docker image to AWS ECR
* Create a VPC with public and private subnets for our EKS Cluster
* Create a Kubernetes Cluster
* Create Kubernetes workers(public and private workers)
* Deploy our web application on Kubernetes

**How kubernetes will satisfy the use cases:-**

**High Availability**:

Single master single load balancer setup:

* In this configuration, there is a single master server and a single load balancer.
* The load balancer routes incoming traffic to the master server.
* If the master server fails, the load balancer will be unable to route traffic and the system will be unavailable.

A single master and single load balancer setup has a few advantages over other setups:

1. Simplicity: This setup is relatively simple compared to other setups, as it involves only one master and one load balancer. This makes it easier to set up and maintain.
2. Cost-effectiveness: This setup requires fewer resources and infrastructure, which can make it more cost-effective compared to other setups that require multiple masters and load balancers.
3. Improved performance: A single master and single load balancer setup can improve performance compared to setups with multiple masters and load balancers. This is because the load balancer can direct traffic to the single master, which can process requests more efficiently compared to multiple masters that may have to communicate with each other to process requests.
4. Easier to scale: Scaling this setup is generally easier compared to setups with multiple masters and load balancers. This is because you only need to add more resources to the single master and load balancer, rather than having to manage multiple masters and load balancers.

However, it's important to note that there may be trade-offs and limitations to using a single master and single load balancer setup. For example, this setup may not be as resilient to failures compared to setups with multiple masters and load balancers, as the failure of the single master or load balancer could result in the entire system going down.

Multi-master single load balancer setup:

* In this configuration, there are multiple master servers and a single load balancer.
* The load balancer routes incoming traffic to the least busy or most available master server.
* If a master server fails, the load balancer will automatically route traffic to the remaining servers, providing improved availability.

There are several advantages to using a multi-master and single load balancer setup over both a multi-master and multi-load balancer setup and a single master and single load balancer setup:

1. High availability: With multiple masters, the system can continue to function even if one of the masters fails. This increases the availability of the system.
2. Load balancing: The single load balancer can distribute incoming requests evenly across the multiple masters, improving the overall performance and efficiency of the system.
3. Scalability: The load balancer can be configured to scale horizontally, allowing it to handle a larger number of requests as needed. The masters can also be added or removed as needed to meet changing demand.
4. Simplicity: This setup is relatively simple to implement and maintain compared to a multi-master and multi-load balancer setup, as there is only one load balancer. This makes it easier to understand and troubleshoot any issues that may arise.

However, a multi-master and multi-load balancer setup may offer some additional benefits, such as increased fault tolerance and the ability to handle a larger number of requests. In cases where extremely high availability and scalability are critical, a multi-master and multi-load balancer setup may be more suitable.

Multi-master multi load balancer setup in Kubernetes:

* In this configuration, there are multiple replicas of a Deployment managed by a Kubernetes cluster, and multiple load balancers.
* The load balancers distribute traffic evenly among the replicas.
* If a replica fails, Kubernetes will automatically restart it or replace it with a new one, providing improved availability.

There are several advantages to using a multi-master and multi-load balancer setup over both a multi-master and single load balancer setup and a single master and single load balancer setup:

1. High availability: With multiple masters and load balancers, the system can continue to function even if one of the masters or load balancers fails. This increases the availability of the system.
2. Load balancing: Each load balancer can distribute incoming requests evenly across the multiple masters, improving the overall performance and efficiency of the system.
3. Scalability: Each load balancer can be configured to scale horizontally, allowing it to handle a larger number of requests as needed. The masters and load balancers can also be added or removed as needed to meet changing demand.
4. Fault tolerance: With multiple load balancers, the system is more resilient to failures, as incoming requests can be redirected to another load balancer if one fails.

However, a multi-master and multi-load balancer setup may be more complex and costly to implement and maintain compared to a single master and single load balancer setup or a multi-master and single load balancer setup. In cases where simplicity and cost are critical, one of these other setups may be more suitable.

Overall, a multi-master multi load balancer setup in Kubernetes provides improved availability and performance compared to a single master single load balancer setup by allowing for the distribution of traffic across multiple servers and multiple load balancers, and enabling failover in the event of a server failure. It is useful in situations where a single load balancer may not be able to handle the load or where it is important to distribute traffic evenly across multiple load balancers for improved performance.

**Multiple Environments**

**For Solving this Problem we can use Multiple Cluster.**

**Reasons to have multiple clusters**

· Separation of production/development/test: especially for testing a new version of Kubernetes, of a service mesh, of other cluster software

· Compliance: according to some regulations some applications must run in separate clusters/separate VPNs

· Better isolation for security

· Cloud/on-prem: to split the load between on-premise services

**Reasons to have a single cluster**

· Reduce setup, maintenance and administration overhead

· Improve utilization

· Cost reduction

Considering a not too expensive environment, with average maintenance, and yet still ensuring security isolation for production applications, I would recommend:

· 1 cluster for DEV and STAGING (separated by namespaces)

· 1 cluster for PROD

**For Managing Mutiple Clusters:**

**Combine a powerful CI/CD tool with** [**helm**](https://github.com/helm/helm).

Combining a powerful continuous integration and delivery (CI/CD) tool with Helm can be a useful approach for managing multiple environments, such as staging and production, in Kubernetes. Here are some steps you can follow to set this up:

Set up our CI/CD tool

Create a Helm chart for our application

Set up your staging environment

Set up your production environment

Use your CI/CD tool to deploy updates

Overall, using a CI/CD tool in combination with Helm can provide a powerful and flexible way to manage multiple environments in Kubernetes, allowing you to automate the deployment process and quickly deploy updates to your applications.

KUBERNETES FEATURES USED:

—>Namespaces

—>Labels

—>Helm Charts

**Zero Downtime Deployment**

### Updating an application

Users expect applications to be available all the time and developers are expected to deploy new versions of them several times a day. In Kubernetes this is done with rolling updates. Rolling updates allow Deployments' update to take place with zero downtime by incrementally updating Pods instances with new ones. The new Pods will be scheduled on Nodes with available resources.

Rolling updates allow the following actions:

* Promote an application from one environment to another (via container image updates)
* Rollback to previous versions
* Continuous Integration and Continuous Delivery of applications with zero downtime

We can also use blue green deployment is an application release model that gradually transfers user traffic from a previous version of an app or microservice to a nearly identical new release—both of which are running in production.

There are several advantages to using Rolling Deployments (also known as Rolling Updates) over Blue/Green Deployments:

1. Cost savings:
2. Faster deployment:
3. Less complex:
4. More flexible:

Due to the advantages above we are moving forward with Rolling updates.

**This can be perform a rolling update using kubectl.**

**Feature Branch Deployment:**

The ability to deploy code to test a particular feature/bug without actually deploying it to development, staging, or production. Kubernetes makes this pretty easy with **the use of namespaces**.

In Kubernetes, a feature branch deployment allows you to test and deploy changes to your application in a separate environment before merging them into the main branch. This can be useful for testing new features or for making small, incremental changes to your application without affecting the main branch.

To make a feature branch deployment in Kubernetes, you can follow these steps:

* Create a new branch
* Build and test the feature
* Create a new namespace
* Deploy the feature
* Test the feature
* Merge the feature:

For achieving this we need to do couple of things :

* Namespace
* ConfigMaps
* Secrets
* Manifests for the services that are deployed
* A script to tie everything along

**Rollback Strategies:**

The service that we can use here is: ReplicaSet Under Deployment.

Every time you create a Deployment, the deployment creates a ReplicaSet and delegates creating (and deleting) the Pods. The sole responsibility for the ReplicaSet is to count Pods. Instead, the Deployment manages ReplicaSets and orchestrates the rolling update.

**After the rolling update, the previous ReplicaSet is not deleted** — not immediately at least.

Instead, it is kept around with a replicas count of 0.

If you try to execute another rolling update from version 2 to version 3, you might notice that at the end of the upgrade, you have two ReplicaSets with a count of 0.

In other words, **keeping the previous ReplicaSets around is a convenient mechanism to roll back to a previously working version of your app.**

By default Kubernetes stores the last 10 ReplicaSets and lets you roll back to any of them.

But you can change how many ReplicaSets should be retained by changing the “spec.revisionHistoryLimit” in your Deployment.

**Observability :**

This can be achieved by 3rd party apps such as Prometheus and Grafana which is easy and free. It is suggested as such Grafana is used to monitor the infrastructure and log analytics, predominantly to improve their operational efficiency. Dashboards make tracking users and events easy as it automates the collection, management, and viewing of data

**Alerts and Notification:**

Here we can use two services particularly

* AWS CloudWatch-monitors the EKS service according to the needs(autoscaling,any other alerts).
* AWS SNS service- Sends the notification to us if the alert is triggered in the cloud watch.
* This can also be achieved by 3rd party apps such as Prometheus and Grafana.

**Final Conclusion**

Strategies to be Used to Solve Use cases:

* High availability-Multi master Multi cloud setup
* Multiple Environments-Combining CICD Tool with Helm for maintaining multiple clusters
* Zero downtime deployment-Rolling Updates
* Feature branch deployment - Use of namespaces
* Rollback strategies- ReplicaSet Under Deployment
* Observability- Prometheus and Grafana
* Alerts and Notifications- AWS CloudWatch, AWS SNS, Prometheus and Grafana