Managing deployment using Kubernetes engine

Hetrogenenou deployment stands for the blend of 2 infrastructure environment.

Heterogeneous deployments typically involve connecting two or more distinct infrastructure environments or regions to address a specific technical or operational need. Heterogeneous deployments are called "hybrid", "multi-cloud", or "public-private", depending upon the specifics of the deployment.

For this lab, heterogeneous deployments include those that span regions within a single cloud environment, multiple public cloud environments (multi-cloud), or a combination of on-premises and public cloud environments (hybrid or public-private).

Various business and technical challenges can arise in deployments that are limited to a single environment or region:

- ***Maxed out resources**: In any single environment, particularly in on-premises environments, you might not have the compute, networking, and storage resources to meet your production needs.*
- ***Limited geographic reach**: Deployments in a single environment require people who are geographically distant from one another to access one deployment. Their traffic might travel around the world to a central location.*
- ***Limited availability**: Web-scale traffic patterns challenge applications to remain fault-tolerant and resilient.*
- ***Vendor lock-in**: Vendor-level platform and infrastructure abstractions can prevent you from porting applications.*

- ***Inflexible resources****: Your resources might be limited to a particular set of compute, storage, or networking offerings.*

Heterogeneous deployments can help address these challenges, but they must be architected using programmatic and deterministic processes and procedures. One-off or ad-hoc deployment procedures can cause deployments or processes to be brittle and intolerant of failures. Ad-hoc processes can lose data or drop traffic. Good deployment processes must be repeatable and use proven approaches for managing provisioning, configuration, and maintenance.

Three common scenarios for heterogeneous deployment are:

- *multi-cloud deployments*
- *fronting on-premises data*
- *continuous integration/continuous delivery (CI/CD) processes*

Set the zone:

gcloud config set compute/zone us-west1-b

Create a cluster with 3 nodes to create:

***gcloud container clusters create bootcamp ***

***--machine-type e2-small ***

***--num-nodes 3 ***

--scopes "https://www.googleapis.com/auth/projecthosting,storage-rw"

Wtf is this? I thought we need a yaml file for such scaling?

T1: learn about the deployment object:

Kubectl explain deployment.

kubectl explain deployment --recursive

kubectl explain deployment.metadata.name

Create ur own deployment object with kubectl create

kubectl create -f deployments/fortune-app-blue.yaml

Verify that the deployment was created.

Kubectl get deployments

4. *Once the deployment is created, Kubernetes will create a* `ReplicaSet` *for the deployment. You can verify that a* `ReplicaSet` *was created for the deployment:*

```
kubectl get replicasets
```

1. The Basics: Kubernetes Deployments

*A **Deployment** is a Kubernetes object that manages a set of identical Pods. It's the standard way to run a stateless application. Its main jobs are to:*

- ***Maintain a desired state:*** *Ensure a specific number of Pods (replicas) are always running.*
- ***Enable updates:*** *Provide strategies to update the application to a new version with zero downtime.*
- ***Self-heal:*** *Automatically replace Pods that fail or become unresponsive.*

Key Commands for Managing Deployments

Create a Deployment:

Bash

kubectl create -f your-deployment-file.yaml

-

View Deployments:

Bash

kubectl get deployments

-

Scale a Deployment:

Bash

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

-

Check rollout history and status:

Bash

kubectl rollout history deployment/<deployment-name>

kubectl rollout status deployment/<deployment-name>

-

---

2. Rolling Updates (The Default)

*A **rolling update** is the default Kubernetes strategy. It gradually replaces old Pods with new ones, one by one, ensuring the application remains available throughout the update process.*

*Think of it like a **dimmer switch**, slowly transitioning from the old version to the new one.*

How it Works

1. *You update the container image version in your Deployment manifest (e.g., from 1.0.0 to 2.0.0).*
2. *Kubernetes creates a new Pod with the new version.*
3. *Once the new Pod is ready and healthy, Kubernetes terminates one of the old Pods.*
4. *This process repeats until all old Pods are replaced by new ones.*

Key Commands for Rolling Updates

***Trigger an update:** You can edit the live deployment or apply a new file.*

Bash

```
# Option A: Edit the live deployment

kubectl edit deployment <deployment-name>



# Option B: Apply an updated YAML file

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

-

***Pause and Resume:** You can pause an update mid-way to investigate issues.*

Bash

```
kubectl rollout pause deployment/<deployment-name>

kubectl rollout resume deployment/<deployment-name>
```

-

***Rollback (Undo):** If you find a bug, you can quickly revert to the previous version.*

Bash

```
kubectl rollout undo deployment/<deployment-name>
```

-

---

3. Canary Deployments

*A **canary deployment** is a strategy where you release a new version to a small subset of users before rolling it out to everyone. This allows you to test the new version in a live production environment with minimal risk.*

*Think of it like having a **taste-tester** for your new recipe before serving it to all your guests.*

How it Works

1. *You have your main, stable deployment running (e.g., fortune-app-blue with 3 replicas).*
2. *You create a second, separate **canary deployment** with the new version, but with very few replicas (e.g., fortune-app-canary with 1 replica).*
3. *The crucial part: Your **Kubernetes Service** uses a label selector that matches Pods from **both** deployments (e.g., app: fortune-app).*
4. *The Service automatically load-balances traffic, sending most requests to the stable version and a small percentage to the canary version.*

Key Commands for Canary Deployments

Create the canary deployment:

Bash

```
kubectl create -f deployments/fortune-app-canary.yaml
```

-

***Verify traffic split:** You check the service endpoint and observe that some responses come from the new version.*

Bash

```
# Run this loop to see responses from both versions

for i in {1..10}; do curl -s http://<SERVICE_IP>/version; echo; done
```

-

---

4. Blue-Green Deployments

*A **blue-green deployment** (also known as a red-black deployment) is a strategy where you have two identical production environments: "Blue" (the current stable version) and "Green" (the new version). You switch all traffic from Blue to Green at once.*

*Think of this like a **light switch**. Traffic is either 100% on Blue or 100% on Green—never in between.*

How it Works

1. *The "Blue" deployment is live, and the Service is configured to send all traffic to it by selecting its unique label (e.g., version: "1.0.0").*
2. *You deploy the "Green" version as a completely separate deployment. It runs alongside Blue but receives no user traffic.*
3. *You can test the Green environment internally to ensure it works correctly.*
4. *To go live, you **update the Service's selector** to point to the Green deployment's label (e.g., version: "2.0.0"). Traffic instantly switches over.*
5. *The Blue environment is kept on standby for a quick rollback if needed.*

Key Commands for Blue-Green Deployments

Create the Green deployment:

Bash

```
kubectl create -f deployments/fortune-app-green.yaml
```

-

Switch traffic to Green:

Bash

```
# Apply a service file that changes the selector to match the green deployment

kubectl apply -f services/fortune-app-green-service.yaml
```

-

Rollback to Blue:

Bash

```
# Re-apply the original service file to switch traffic back to blue
```

- *kubectl apply -f services/fortune-app-blue-service.yaml*