**Mastering Multi-Cloud**

**Labs:** **Multi-Cloud App Deployment on AWS, Azure, and GCP**

### **Case Study**

Form3, a leading provider of cloud-native payment processing technology, operates in a highly regulated global financial services industry. Their mission is to deliver a managed payments infrastructure with a single API, enabling customers to adhere to diverse payment schemes across various countries. Given the critical nature of payment transactions, ensuring uninterrupted service availability and resilience against potential outages is paramount for Form3.

Initially, Form3 leveraged a single cloud provider for its infrastructure. However, evolving regulatory landscapes and the inherent risks associated with cloud concentration prompted a strategic shift towards a multi-cloud approach. This decision was driven by the need to demonstrate platform portability and maintain service continuity even in the event of a major cloud provider disruption. To achieve this, Form3 adopted a sophisticated architecture utilizing Kubernetes clusters across all three major hyperscale cloud providers: Amazon Web Services (AWS) Elastic Kubernetes Service (EKS), Microsoft Azure Kubernetes Service (AKS), and Google Cloud Platform (GCP) Google Kubernetes Engine (GKE).

This multi-cloud Kubernetes strategy allows Form3 to distribute its critical payment application workloads across geographically diverse and independently operated cloud environments. By doing so, they aim to mitigate the risks of vendor lock-in, enhance disaster recovery capabilities, and meet stringent regulatory requirements for operational resilience within the financial sector.

### **Business Challenge**

As Form3 expanded its global footprint and processed an increasing volume of critical payment transactions, reliance on a single cloud provider emerged as a significant business challenge. A major concern was the risk of downtime, as any substantial outage or service disruption could severely impact Form3’s ability to process payments, resulting in financial losses, reputational damage, and potential non-compliance with service level agreements (SLAs) and regulatory obligations. In parallel, regulatory scrutiny continued to intensify, with financial authorities requiring clear evidence of robust disaster recovery capabilities and platform portability; dependence on a single provider raised doubts about Form3’s ability to sustain operations if that provider became unavailable. Additionally, vendor lock-in posed strategic and financial risks, as a tightly coupled, single-cloud architecture constrained flexibility, limited innovation, and potentially increased long-term operational costs. Consequently, Form3 required a solution that would eliminate the single point of failure inherent in a single-cloud deployment and ensure the continuous availability and resilience of its payment platform, enabling seamless and automated traffic failover to alternative cloud environments without manual intervention or service interruption.

### **Solution**

To address these challenges, Form3 implemented a multi-cloud Kubernetes architecture integrated with a Global Load Balancer (GLB) to enable intelligent traffic management and automated failover. The core application was deployed consistently across managed Kubernetes services on all three major cloud platforms, including an Amazon EKS cluster in a designated AWS region, an Azure AKS cluster in an Azure region, and a Google GKE cluster in a GCP region. Each cluster hosted identical instances of the application to ensure redundancy and operational consistency across environments. A central GLB, implemented using an NGINX server on an AWS EC2 instance, was configured as the single entry point for incoming traffic, utilizing the ip\_hash method for session persistence and routing requests to the external IP addresses of the load balancers fronting each Kubernetes service as backend targets.

The GLB continuously monitored the health of application instances across all three cloud providers and, in the event of an outage such as the decommissioning of an EKS or GKE cluster during testing, automatically detected unhealthy backends and redirected traffic to the remaining healthy environments. This approach ensured uninterrupted service delivery with no user-visible disruption. Overall, the multi-cloud design provided Form3 with exceptional resilience and continuous availability, enabling the organization to satisfy stringent regulatory requirements, mitigate the impact of cloud provider failures, and uphold its commitment to delivering reliable, always-on global payment processing services.

The tasks performed in this lab include:

* Create an AWS EKS cluster using AWS Cloud Shell
* Create an Azure AKS cluster and deploy an app in it
* Create a Google GKE cluster and network subnet using the GCP Cloud Shell
* Create an EC2 instance as a GLB using NGINX

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Step 1: Create AWS EKS Cluster**  1. Log in to the AWS Console. Then, click on the **Shell** icon at the top to open AWS CloudShell. Execute the commands below to verify the version of the tools. Also, check that kubectl and ekctl are installed in the AWS CloudShell. By default, AWS CLI is installed in the AWS CloudShell.   |  | | --- | | aws --version  kubectl version --client --short || echo "kubectl missing"  eksctl version || echo "eksctl missing" |     2. Execute the commands below to install kubectl.   |  | | --- | | curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"  chmod +x kubectl  sudo mv kubectl /usr/local/bin/  sudo chmod +x /usr/local/bin/kubectl |     3. Execute the commands below to install ekctl.   |  | | --- | | curl --silent --location "https://github.com/eksctl-io/eksctl/releases/latest/download/eksctl\_$(uname -s)\_amd64.tar.gz" \  -o eksctl.tar.gz  tar -xzf eksctl.tar.gz -C /tmp  sudo mv /tmp/eksctl /usr/local/bin  rm eksctl.tar.gz |     4. Again, execute the commands below to verify that kubectl and ekctl are installed successfully.   |  | | --- | | kubectl version  eksctl version |     5. Execute the below-provided command to create an AWS EKS cluster. The cluster name is **k8s-example** in the region **eu-west-2,** which is the London region. eksctl automates networking, subnets, node groups, and more. It takes more than 15 minutes to create a cluster.   |  | | --- | | eksctl create cluster --name k8s-example --region eu-west-2 |     6. Execute the commands below to list all the nodes, pods, and namespaces currently running.   |  | | --- | | kubectl get nodes  kubectl get pods  kubectl get namespaces |     7. Next, we define a simple app manifest that deploys 5 replicas of a Node app and exposes it with a LoadBalancer service. Execute the following command: **nano deployment-example.yaml** to create a YAML file. After that, copy the script and paste it into the code editor.   |  | | --- | | apiVersion: apps/v1  kind: Deployment  metadata:  name: amazon-deployment  namespace: default  labels:  app: amazon-app  spec:  replicas: 5  selector:  matchLabels:  app: amazon-app  tier: frontend  version: 1.0.0  template:  metadata:  labels:  app: amazon-app  tier: frontend  version: 1.0.0  spec:  containers:  - name: amazon-container  image: ooghenekaro/amazon:2  ports:  - containerPort: 3000  ---  apiVersion: v1  kind: Service  metadata:  name: amazon-service  labels:  app: amazon-app  spec:  type: LoadBalancer  ports:  - port: 80  targetPort: 3000  selector:  app: amazon-app |   Press **Ctrl+X** to initiate the exit process, then press **Y** to save the changes.    8. Execute the following command: **kubectl apply -f deployment-example.yaml** to create a Kubernetes deployment.    9. Execute the following command: **kubectl get pods -w** to display the available pods and running containers in the cluster.    10. Execute the following command: **kubectl get service** to list and display summary information about the Kubernetes Services within the current, default namespace of your cluster.    11. Execute the following command: **kubectl get all** to retrieve and display a summary of several common core Kubernetes resources. Here, you can see the **EXTERNAL-IP** column is showing the load balancer’s DNS or IP. Copy the IP address and paste it into the browser.    12. Paste the copied IP address in the browser and press Enter. You will see the sample app is live. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Step 2: Create Azure AKS Cluster**  1. Log in to the Azure portal. Then click on the Azure Cloud Shell at the top. Execute the command below to create a Resource Group that acts as a logical container to hold all related resources.   |  | | --- | | az group create --name myResourceGroup --location eastus |     2. Execute the command below to create an AKS cluster. This command provisions both the managed control plane and a node pool with one worker node.   |  | | --- | | az aks create \  --resource-group myResourceGroup \  --name myAKSCluster \  --node-count 1 \  --enable-addons monitoring \  --generate-ssh-keys |     3. Execute the command below to configure your kubeconfig file so kubectl can communicate with the AKS cluster.   |  | | --- | | az aks get-credentials --resource-group myResourceGroup --name myAKSCluster |   4. Execute the following command: **kubectl get nodes** to list the cluster nodes    5. Execute the command below to deploy an application to this cluster. We are using a sample microservices-based app provided by Microsoft. It includes multiple components: store-front service, order service, product service, RabbitMQ, and MongoDB.   |  | | --- | | kubectl apply -f https://raw.githubusercontent.com/Azure-Samples/aks-store-demo/main/aks-store-all-in-one.yaml |     6. Execute the commands below to create a separate Kubernetes namespace called team-a and deploy the same app there.   |  | | --- | | kubectl create namespace team-a  kubectl apply -f https://raw.githubusercontent.com/Azure-Samples/aks-store-demo/main/aks-store-all-in-one.yaml -n team-a |     7. Execute the following command: **kubectl get service store-front --watch** to verify that our application is running and obtain the external IP of the load balancer so we can access the app in a browser. Copy the **EXTERNAL-IP**.    8. Open a new browser tab and paste the EXTERNAL-IP that you copied in the previous step and press Enter. You will see the sample app is live. |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Step 3: Create GCP GKE Cluster**  1. Log in to the GCP console. Click on the **GCP** **Cloud Shell** terminal icon at the top. Then execute the following command: **gcloud version** to verify that Google Cloud CLI is installed in the GCP Cloud Shell.    2. Execute the following command: **kubectl version** to check the version and confirm that kubectl is installed in the GCP Cloud Shell.    3. Execute the commands below to set the active GCP project and choose a region. We are using the europe-west2 region, which corresponds to London. Please use your project ID in the command.   |  | | --- | | gcloud config set project <YOUR\_PROJECT\_ID>  gcloud config set compute/region europe-west2  gcloud config set compute/zone europe-west2-a |     4. Execute the command below to create a custom VPC network.   |  | | --- | | gcloud compute networks create gke-network \  --project=gcplabs-318506 \  --subnet-mode=custom |     5. Execute the command below to create a subnet in the europe-west2 region.   |  | | --- | | gcloud compute networks subnets create gke-subnet \  --project=gcplabs-318506 \  --network=gke-network \  --region=europe-west2 \  --range=10.0.0.0/24 |     6. Execute the command below to create the GKE Cluster.   |  | | --- | | gcloud container clusters create gke-platform-demo \  --project=gcplabs-318506 \  --zone=europe-west2-a \  --num-nodes=3 \  --enable-ip-alias \  --network=gke-network \  --subnetwork=gke-subnet |     7. Execute the command below to connect kubectl to the cluster.   |  | | --- | | gcloud container clusters get-credentials gke-platform-demo \  --zone europe-west2-a |     8. Execute the commands below to list all the worker machines and namespaces.   |  | | --- | | kubectl get nodes  kubectl get namespaces |     9. Execute the command below to deploy the app.   |  | | --- | | kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/microservices-demo/main/release/kubernetes-manifests.yaml |     10. Execute the following command: **kubectl get pods** to list all Pods.    11. Execute the following command: **kubectl get deployments** to list and display a summary of all deployment resources.    12. Execute the following command: **kubectl get services** to retrieve and display a list of services currently running. Also, to get the EXTERNAL-IP of the app. Copy the **EXTERNAL-IP**.    13. Open a new browser tab and paste the EXTERNAL-IP that you copied in the previous step and press Enter. You will see the sample app is live. |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Step 4: Create Global Load Balancer (GLB)**  1. We will create a GLB using an AWS EC2 instance. Create an EC2 instance. Select the **GLB\_Server**. Click on the **Connect** button at the top.    2. Copy the SSH command.    3. Open the **CMD,** then paste the SSH command here and press **Enter**. Type **yes** and press **Enter** to continue.    4. Execute the following command: **sudo dnf update -y** to update system packages.    5. Execute the following command: **sudo dnf install -y nginx** to install the NGINX server.    6. Execute the commands below to start the NGINX server and verify its status. NGINX is in an active state.   |  | | --- | | sudo systemctl enable nginx  sudo systemctl start nginx  systemctl status nginx |     7. Execute the following command: **curl http://localhost** to do a quick test. The test is successful.    8. Execute the following command: **sudo nano /etc/nginx/conf.d/global-lb.conf** to create a YAML file.    9. After that, copy the script and paste it into the code editor. In the code, please add the external IP of each AWS, Azure, and GCP.   |  | | --- | | upstream multi\_cloud\_backend {  ip\_hash;  server <AWS\_EKS\_LB\_IP>:80;  server <AZURE\_AKS\_LB\_IP>:80;  server <GCP\_GKE\_LB\_IP>:80;  }  server {  listen 80;  listen [::]:80;  server\_name \_;  gzip off;  proxy\_buffering off;  proxy\_request\_buffering off;  location / {  proxy\_pass http://multi\_cloud\_backend;  proxy\_http\_version 1.1;  proxy\_set\_header Host $http\_host;  proxy\_set\_header X-Real-IP $remote\_addr;  proxy\_set\_header X-Forwarded-For $remote\_addr;  proxy\_set\_header X-Forwarded-Proto http;  proxy\_redirect off;  proxy\_connect\_timeout 10s;  proxy\_send\_timeout 120s;  proxy\_read\_timeout 120s;  }  } |   Press **Ctrl+X** to initiate the exit process, then press **Y** to save the changes.    10. Execute the commands below to remove all other NGINX configs.   |  | | --- | | sudo rm -f /etc/nginx/conf.d/default.conf  sudo rm -f /etc/nginx/default.d/\*.conf |     11. Execute the commands below to test the NGINX configuration files for syntax errors and potential issues without applying any changes to the running server. Then restart the NGINX server.   |  | | --- | | sudo nginx -t  sudo systemctl restart nginx |     12. Copy the public IP address of the GLB\_Server EC2 instance. Then, open a web browser in private (incognito) mode, paste the IP address into the address bar, and press **Enter**. An application interface will be displayed. Depending on the current load-balancing decision, the response may originate from AWS, Azure, or GCP. Please note that the initial page load may take some time.    13. We will sequentially decommission each cloud provider’s Kubernetes cluster to validate the load balancer’s ability to fail over to the remaining healthy applications and maintain service availability. The first step will be to delete the AWS EKS cluster. Execute the command below to delete the AWS EKS cluster.   |  | | --- | | eksctl delete cluster --name k8s-example --region eu-west-2 |     14. Navigate to the browser session where the GLB application is running and refresh the page. A different application instance will be displayed. Since the AWS EKS cluster has been decommissioned, the load balancer will route traffic only to the remaining healthy backends in Azure or GCP. In this case, the load balancer directs the request to the application hosted on GCP, and the GCP application interface is presented.    15. Next, we will decommission the GCP GKE cluster to verify that the load balancer correctly fails over to the last remaining available application, which is hosted on Azure AKS. Execute the command below to delete the GCP GKE cluster. Type **yes** and press **Enter** to continue.   |  | | --- | | gcloud container clusters delete gke-platform-demo --zone europe-west2-a |     16. Navigate to the browser session where the GLB application is running and refresh the page. The application hosted on Azure AKS will be displayed, as it is the last remaining healthy backend to which the load balancer can route traffic.    17. We have successfully implemented a multi-cloud deployment across AWS, Azure, and GCP. If any cloud provider becomes unavailable, the load balancer automatically redirects traffic to the remaining healthy environment, thereby ensuring continuous service availability and achieving high availability.  18. To avoid incurring additional charges, delete all resources across all three cloud providers. The AWS EKS cluster has already been deleted, which also removed the backend services associated with it. The GCP GKE cluster has been deleted; however, the VPC network and subnets still need to be removed. In Azure, deleting the resource group will automatically delete all services and resources contained within it.  19. In the Azure portal, go to the **myResourceGroup**. Click on the **Delete resource group** it will delete the resources in it.    20. In the Google Cloud Console, go to the **VPC networks** page.  21. Click the name of the **custom VPC network** you want to delete.  22. On the network details page, click the **Delete VPC network** button at the top of the page.  23. In the confirmation window, review the details and click **Delete** to confirm the action.  24. Also, delete the other services related to networking in GCP, like firewall rules. |