Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

INTRODUCTION TO DEVOPS ON GCP – STUDY MATERIAL

CHAPTER 1: INTRODUCTION TO DEVOPS

1.1 What is DevOps?

DevOps is a **software development methodology** that emphasizes **continuous integration, continuous delivery (CI/CD), and automation** to improve software quality, deployment speed, and reliability.

✓ **Combines Development & Operations** – Bridges the gap between software development and IT operations.
✓ **Automation** – Reduces manual effort in testing, deployment, and monitoring.
✓ **Faster Releases** – Enables rapid, reliable software delivery.
✓ **Scalability & Security** – Uses cloud-native solutions for infrastructure management.

1.2 DevOps on Google Cloud

Google Cloud Platform (GCP) provides a suite of **DevOps tools** to enable **efficient CI/CD, monitoring, security, and infrastructure automation**.

✓ **Cloud Build** – Automates CI/CD pipelines.
✓ **Artifact Registry** – Stores and manages Docker images.
✓ **Google Kubernetes Engine (GKE)** – Manages containerized applications.
✓ **Cloud Run** – Serverless deployment for microservices.
✓ **Cloud Operations (Monitoring, Logging, and Tracing)** – Observability for applications.
✓ **Infrastructure as Code (IaC)** – Automate infrastructure with Terraform and Deployment Manager.

📌 **Example:**
A **fintech startup** implements **DevOps on GCP** to **automate deployment of microservices using Kubernetes and Cloud Build**.

CHAPTER 2: SETTING UP A DEVOPS ENVIRONMENT ON GCP

2.1 Prerequisites

✓ A **Google Cloud account** with billing enabled.
✓ Install **Google Cloud SDK** and authenticate using:

```
gcloud auth login
```

✓ Enable required APIs:

```
gcloud services enable cloudbuild.googleapis.com
container.googleapis.com artifactregistry.googleapis.com
```

2.2 Setting Up a Cloud DevOps Project

1. Open **Google Cloud Console** → Navigate to **IAM & Admin**.
2. Create a **new project** called my-devops-project.
3. Assign **roles** (Editor, Owner, or DevOps Engineer).

📌 **Example:**
A **software company** creates a **GCP DevOps project** to manage CI/CD workflows for a cloud-native application.

Chapter 3: CI/CD with Cloud Build & Artifact Registry

3.1 Automating Builds with Cloud Build

Cloud Build automates the **building, testing, and deployment** of applications.

Step 1: Define a Cloud Build Configuration (cloudbuild.yaml)

```
steps:
- name: 'gcr.io/cloud-builders/npm'
  args: ['install']
- name: 'gcr.io/cloud-builders/npm'
  args: ['test']
- name: 'gcr.io/cloud-builders/docker'
  args: ['build', '-t', 'gcr.io/$PROJECT_ID/my-app', '.']
- name: 'gcr.io/cloud-builders/docker'
  args: ['push', 'gcr.io/$PROJECT_ID/my-app']
images:
- 'gcr.io/$PROJECT_ID/my-app'
```

Step 2: Trigger a Build with Cloud Build

```
gcloud builds submit --config cloudbuild.yaml .
```

📌 **Example:**
A **mobile app development team** uses **Cloud Build to automate testing and container image builds**.

3.2 Managing Containerized Applications with Artifact Registry

Step 1: Create a Private Docker Registry

gcloud artifacts repositories create my-docker-repo --repository-format=docker --location=us-central1

Step 2: Push a Docker Image to Artifact Registry

docker tag my-app gcr.io/$PROJECT_ID/my-docker-repo/my-app

docker push gcr.io/$PROJECT_ID/my-docker-repo/my-app

📌 **Example:**
A **banking company** stores **secure Docker images** in **Artifact Registry** before deploying them to Kubernetes.

CHAPTER 4: DEPLOYING APPLICATIONS WITH KUBERNETES & CLOUD RUN

4.1 Deploying to Google Kubernetes Engine (GKE)

1. **Create a Kubernetes Cluster**
2. gcloud container clusters create my-cluster --num-nodes=3
3. **Deploy an Application to GKE**
4. apiVersion: apps/v1
5. kind: Deployment
6. metadata:

```
name: my-app
spec:
replicas: 3
        selector:
matchLabels:
            app: my-app
template:
        metadata:
 labels:
             app: my-app
spec:
            containers:
            - name: my-app
             image: gcr.io/my-project-id/my-app:latest
             ports:
            - containerPort: 8080
```

Apply the Deployment

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

📌 **Example:**
An **e-commerce company** deploys **containerized web services** to **Google Kubernetes Engine (GKE)** for scalability.

4.2 Deploying Serverless Apps with Cloud Run

1. **Deploy a Cloud Run Service**
2. gcloud run deploy my-service --image gcr.io/$PROJECT_ID/my-app --platform managed --region us-central1

📌 **Example:**
A **chatbot application** runs on **Cloud Run** to **handle real-time customer queries with minimal infrastructure management**.

CHAPTER 5: MONITORING & SECURITY IN DEVOPS ON GCP

5.1 Logging & Monitoring with Cloud Operations

✓ Use **Cloud Monitoring** for real-time performance tracking.
✓ Enable **Cloud Logging** to capture application logs.
✓ Integrate with **Cloud Trace** for latency analysis.

gcloud logging read "resource.type=gce_instance"

📌 **Example:**
A **game development company** uses **Cloud Logging** to **analyze game server crashes and improve uptime**.

5.2 Security Best Practices for DevOps on GCP

✓ **Use IAM roles** to restrict user access.
✓ **Enable Binary Authorization** to verify container images before deployment.
✓ **Enable Cloud Security Scanner** for vulnerability detection.
✓ **Use Secret Manager** to store API keys securely.

gcloud secrets create my-secret --replication-policy="automatic"

📌 **Example:**
A **healthcare platform** secures **patient data** using **IAM, Secret Manager, and Cloud Security Scanner**.

Chapter 6: Infrastructure as Code (IaC) on GCP

6.1 Automating Infrastructure with Terraform

Step 1: Install Terraform

```
sudo apt-get install terraform
```

Step 2: Define a GKE Cluster in Terraform (main.tf)

```
resource "google_container_cluster" "primary" {
  name     = "my-cluster"
  location = "us-central1-a"
  initial_node_count = 3
}
```

Step 3: Deploy Infrastructure

```
terraform init
terraform apply
```

📌 **Example:**
A **DevOps team** uses **Terraform** to automate provisioning of **Kubernetes clusters and virtual machines**.

Chapter 7: Exercise & Review Questions

Exercise:

1. **Set up a Cloud Build pipeline** to build and test a Docker container.
2. **Deploy an application to Google Kubernetes Engine (GKE)**.
3. **Use Terraform** to provision a virtual machine in GCP.
4. **Enable Cloud Logging & Monitoring** for an application.
5. **Secure a Cloud Run service** using IAM and Secret Manager.

Review Questions:

1. What are the benefits of **DevOps on Google Cloud**?
2. How does **Cloud Build automate CI/CD workflows**?
3. What is the role of **Artifact Registry** in container management?
4. How do **Terraform & Deployment Manager** help with Infrastructure as Code (IaC)?
5. What are **security best practices** for DevOps on GCP?

Conclusion: Enabling DevOps Efficiency on Google Cloud

✓ Google Cloud provides a full-stack DevOps toolset.
✓ Automate builds, testing, and deployments using CI/CD.
✓ Ensure security, monitoring, and scalability with cloud-native solutions.

🚀 **Mastering DevOps on GCP improves software delivery speed and reliability!**

TERRAFORM FOR GCP INFRASTRUCTURE AUTOMATION

CHAPTER 1: INTRODUCTION TO TERRAFORM & INFRASTRUCTURE AS CODE (IAC)

1.1 What is Terraform?

Terraform is an **open-source Infrastructure as Code (IaC) tool** that allows developers to **define, provision, and manage cloud resources** using a declarative configuration language.

1.2 Why Use Terraform for Google Cloud (GCP)?

✓ **Automates GCP Infrastructure** – Deploys resources like VMs, storage, and networks.
✓ **Infrastructure as Code (IaC)** – Uses configuration files for repeatability.
✓ **Version Control & Collaboration** – Works with Git for team collaboration.
✓ **Multi-Cloud Support** – Can manage resources across GCP, AWS, and Azure.
✓ **State Management** – Keeps track of the infrastructure state to prevent inconsistencies.

📌 **Example:**
A **startup automates its GCP infrastructure** using Terraform to create **VMs, Cloud Storage, and IAM roles with a single command**.

CHAPTER 2: SETTING UP TERRAFORM FOR GCP

2.1 Prerequisites

✓ **Google Cloud Account** with billing enabled.
✓ **Install Terraform CLI** – Download from Terraform's official site.
✓ **Set up Google Cloud SDK** (gcloud init).
✓ **Enable APIs**:

gcloud services enable compute.googleapis.com iam.googleapis.com

2.2 Authenticate Terraform with GCP

1. Generate a **Service Account Key**:
2. gcloud iam service-accounts create terraform-admin --display-name "Terraform Admin"
3. Assign **Owner Role**:
4. gcloud projects add-iam-policy-binding my-gcp-project --member="serviceAccount:terraform-admin@my-gcp-project.iam.gserviceaccount.com" --role="roles/owner"
5. Download the service account key:
6. gcloud iam service-accounts keys create key.json --iam-account=terraform-admin@my-gcp-project.iam.gserviceaccount.com

📌 **Example:**
A **cloud engineer sets up Terraform authentication** to manage GCP resources securely.

CHAPTER 3: WRITING YOUR FIRST TERRAFORM CONFIGURATION FOR GCP

3.1 Structure of a Terraform Configuration File

Terraform uses **.tf files** to define resources.

Basic Terraform configuration (main.tf):

```
provider "google" {
 credentials = file("key.json")
 project     = "my-gcp-project"
 region      = "us-central1"
}


resource "google_compute_instance" "vm_instance" {
 name         = "my-vm"
 machine_type = "e2-medium"
 zone         = "us-central1-a"

 boot_disk {
  initialize_params {
   image = "debian-cloud/debian-10"
  }
 }

 network_interface {
  network = "default"
  access_config {}
 }
```

}

📌 **Example:**
A **developer provisions a GCP VM** using Terraform instead of manually creating it in the console.

Chapter 4: Terraform Workflow – Plan, Apply, Destroy

4.1 Initializing Terraform

Run this command to initialize the Terraform project:

terraform init

4.2 Previewing Infrastructure Changes

Before applying changes, review what will be created:

terraform plan

4.3 Applying the Configuration

Provision the resources defined in main.tf:

terraform apply -auto-approve

4.4 Destroying Infrastructure

To remove all created resources:

terraform destroy -auto-approve

📌 **Example:**
A **DevOps team automates VM creation, modification, and deletion** using Terraform workflows.

Chapter 5: Managing Terraform State & Variables

5.1 Terraform State (terraform.tfstate)

✓ Keeps track of the current infrastructure.
✓ Helps Terraform determine what changes are needed.
✓ Stored **locally** or in **remote backends** (GCS, S3, etc.).

5.2 Using Terraform Variables

Instead of hardcoding values, use variables:

Define variables (variables.tf):

```
variable "project_id" {
  description = "GCP project ID"
  type        = string
}

variable "region" {
  description = "GCP region"
  type        = string
  default     = "us-central1"
}
```

Reference variables in main.tf:

```
provider "google" {
  project = var.project_id
  region  = var.region
}
```

Pass variables while applying the configuration:

```
terraform apply -var="project_id=my-gcp-project"
```

📌 **Example:**
A **cloud architect parameterizes Terraform configurations** for easy reuse across multiple environments.

CHAPTER 6: TERRAFORM MODULES FOR REUSABLE INFRASTRUCTURE

6.1 What Are Terraform Modules?

Modules allow you to **reuse and manage infrastructure components** efficiently.

6.2 Creating a Module for VM Deployment

Step 1: Create a module folder (modules/vm/main.tf)

```
resource "google_compute_instance" "vm" {
 name        = var.vm_name
 machine_type = var.machine_type
 zone        = var.zone

 boot_disk {
  initialize_params {
   image = "debian-cloud/debian-10"
  }
 }

 network_interface {
  network = "default"
```

```
  access_config {}
 }
}
```

Step 2: Define module variables (modules/vm/variables.tf)

```
variable "vm_name" {}
variable "machine_type" {}
variable "zone" {}
```

Step 3: Use the Module in Your Main Configuration

```
module "vm_instance" {
 source      = "./modules/vm"
 vm_name     = "app-server"
 machine_type = "e2-medium"
 zone        = "us-central1-a"
}
```

📌 **Example:**
A **large enterprise standardizes Terraform configurations** using reusable modules for different teams.

CHAPTER 7: TERRAFORM WITH REMOTE STATE & BACKEND STORAGE

7.1 Storing State in Google Cloud Storage (GCS)

1. **Create a storage bucket**:
2. gsutil mb gs://my-terraform-state/

3. **Configure Terraform to use GCS as the backend (backend.tf)**:

```
terraform {
  backend "gcs" {
    bucket = "my-terraform-state"
    prefix = "terraform/state"
  }
}
```

10. **Initialize Terraform again**:

```
terraform init
```

📌 **Example:**
A **team of DevOps engineers shares Terraform state across multiple team members using GCS backend**.

CHAPTER 8: CI/CD FOR TERRAFORM USING CLOUD BUILD

8.1 Automating Terraform Deployment with Cloud Build

Create a cloudbuild.yaml file to automate Terraform execution:

```
steps:
- name: "hashicorp/terraform"
  args: ["init"]
- name: "hashicorp/terraform"
  args: ["plan"]
- name: "hashicorp/terraform"
```

args: ["apply", "-auto-approve"]

Run Terraform automatically using:

gcloud builds submit --config cloudbuild.yaml

📌 **Example:**
A **finance company automates infrastructure provisioning** using Terraform and Cloud Build for secure and scalable deployments.

CHAPTER 9: EXERCISE & REVIEW QUESTIONS

Exercise:

1. **Set up Terraform for GCP and create a Compute Engine VM.**
2. **Use variables to define project ID, region, and machine type.**
3. **Store Terraform state in Google Cloud Storage.**
4. **Create a module for deploying multiple VMs.**
5. **Set up Cloud Build to automate Terraform deployment.**

Review Questions:

1. What are the benefits of using Terraform for GCP?
2. How does Terraform manage infrastructure state?
3. What is the difference between terraform plan and terraform apply?
4. How do Terraform modules improve infrastructure automation?
5. How can Terraform be integrated with CI/CD pipelines?

CONCLUSION: MASTERING TERRAFORM FOR GCP INFRASTRUCTURE

✓ Terraform simplifies GCP infrastructure provisioning with declarative IaC.
✓ Modules and variables improve code reusability and efficiency.

✓ Remote state management and CI/CD pipelines enhance team collaboration.

🚀 Mastering Terraform enables organizations to scale and automate their cloud infrastructure efficiently!

CLOUD DEPLOYMENT MANAGER – AUTOMATING GCP RESOURCES

CHAPTER 1: INTRODUCTION TO CLOUD DEPLOYMENT MANAGER

1.1 What is Cloud Deployment Manager?

Google **Cloud Deployment Manager** is an **Infrastructure as Code (IaC) tool** that allows users to **define, deploy, and manage Google Cloud resources** using configuration files. It automates infrastructure provisioning, enabling faster and more reliable deployments.

1.2 Why Use Cloud Deployment Manager?

✓ **Automates GCP Resource Deployment** – Reduces manual configuration effort.
✓ **Declarative Configuration** – Defines resources in YAML or Python templates.
✓ **Ensures Consistency** – Avoids configuration drift in cloud environments.
✓ **Version Control & Reusability** – Store configurations in Git for CI/CD workflows.
✓ **Scalable & Repeatable** – Easily replicate infrastructure across projects.

1.3 Key Use Cases

✓ **Deploying Virtual Machines (VMs)** – Automate GCE instance creation.
✓ **Managing Networking Resources** – Define and deploy VPCs, Firewalls, and Load Balancers.
✓ **Provisioning Databases & Storage** – Automate Cloud SQL,

BigQuery, and Cloud Storage setups.

✓ **CI/CD Pipeline Integration** – Use Deployment Manager in DevOps workflows.

📌 **Example:**

A **healthcare startup** automates its **entire cloud infrastructure** using Deployment Manager to **provision Compute Engine, Cloud Storage, and IAM roles**.

CHAPTER 2: SETTING UP CLOUD DEPLOYMENT MANAGER

2.1 Prerequisites

✓ **Enable the Deployment Manager API**

gcloud services enable deploymentmanager.googleapis.com

✓ **Set Up IAM Permissions**

Assign roles like:

- **Deployment Manager Editor**
- **Compute Admin**
- **Storage Admin**

```
gcloud projects add-iam-policy-binding [PROJECT_ID] \
  --member=user:[YOUR_EMAIL] --
role=roles/deploymentmanager.editor
```

📌 **Example:**

An **e-commerce company** assigns Deployment Manager roles to its **DevOps engineers** to automate resource provisioning.

CHAPTER 3: WRITING A DEPLOYMENT CONFIGURATION FILE

3.1 Understanding YAML Configuration

Deployment Manager uses **YAML configuration files** to define resources.

✓ Define a **basic deployment** in vm-config.yaml:

```
resources:
 - name: my-vm-instance
   type: compute.v1.instance
   properties:
    zone: us-central1-a
    machineType: zones/us-central1-a/machineTypes/n1-standard-1
    disks:
    - boot: true
     autoDelete: true
     initializeParams:
      sourceImage: projects/debian-cloud/global/images/family/debian-10
    networkInterfaces:
    - network: global/networks/default
```

📌 **Example:**
A **software company** defines a YAML configuration file to **deploy a VM instance with networking settings**.

3.2 Creating a Deployment

To deploy resources, run the following command:

gcloud deployment-manager deployments create my-deployment --config vm-config.yaml

✓ Check the deployment status:

gcloud deployment-manager deployments describe my-deployment

📌 **Example:**
A **banking institution** uses Deployment Manager to **automate VM deployments** for its financial services applications.

CHAPTER 4: USING TEMPLATES FOR REUSABLE DEPLOYMENTS

4.1 Writing a Python-Based Template

Deployment Manager supports **Python templates** for dynamic configurations.

✓ Create a vm-template.py file:

```
def GenerateConfig(context):
  resources = [{
    'name': context.env['name'],
    'type': 'compute.v1.instance',
    'properties': {
      'zone': context.properties['zone'],
      'machineType': context.properties['machineType'],
      'disks': [{
        'boot': True,
        'autoDelete': True,
        'initializeParams': {
```

```
            'sourceImage': 'projects/debian-
cloud/global/images/family/debian-10'
          }
        }],
        'networkInterfaces': [{
          'network': 'global/networks/default'
        }]
      }
  }]
  return {'resources': resources}
```

✓ Define a **YAML configuration** that calls the Python template (vm-template.yaml):

```
imports:
 - path: vm-template.py
resources:
 - name: my-vm-from-template
   type: vm-template.py
   properties:
    zone: us-central1-a
    machineType: n1-standard-1
```

4.2 Deploy Using a Template

```
gcloud deployment-manager deployments create template-
deployment --config vm-template.yaml
```

📌 **Example:**
A **tech company** creates a reusable **VM template** for **deploying development and production environments**.

Chapter 5: Managing & Updating Deployments

5.1 Updating a Deployment

Modify vm-config.yaml, then update:

gcloud deployment-manager deployments update my-deployment --config vm-config.yaml

📌 **Example:**
A **gaming company** updates **compute instance configurations** dynamically using **deployment updates**.

5.2 Deleting a Deployment

To delete a deployment and free up resources:

gcloud deployment-manager deployments delete my-deployment

📌 **Example:**
A **startup** deletes old deployments to **reduce cloud costs**.

Chapter 6: Best Practices for Cloud Deployment Manager

✓ **Use Templates** – Modularize deployments for reusability.
✓ **Version Control Config Files** – Store YAML files in **Git for CI/CD**.
✓ **Use IAM Policies** – Restrict deployment access with **fine-grained permissions**.
✓ **Automate with Cloud Build** – Trigger deployments as part of a **CI/CD pipeline**.

📌 **Example:**
A **logistics company** integrates Deployment Manager with **Cloud Build** for continuous infrastructure deployment.

CHAPTER 7: EXERCISE & REVIEW QUESTIONS

Exercise:

1. **Create a VM deployment using YAML.**
2. **Modify the VM configuration and update the deployment.**
3. **Create a reusable Python template for VM instances.**

Review Questions:

1. What are the **benefits of Cloud Deployment Manager**?
2. How does a **YAML-based deployment differ from a Python template deployment**?
3. What are **best practices for managing deployment configurations**?
4. How can Deployment Manager be **integrated with CI/CD pipelines**?
5. What are the **steps to update and delete a deployment**?

CONCLUSION: AUTOMATING CLOUD INFRASTRUCTURE WITH DEPLOYMENT MANAGER

✓ Cloud Deployment Manager simplifies infrastructure deployment using declarative configuration.
✓ Reduces human error, ensures consistency, and enhances

automation.

✓ Supports modularization, version control, and integration with CI/CD pipelines.

✓ Essential tool for DevOps teams managing Google Cloud environments.

🚀 Mastering Deployment Manager helps organizations scale and automate cloud infrastructure efficiently!

KUBERNETES & CONTAINER ORCHESTRATION – STUDY MATERIAL

CHAPTER 1: INTRODUCTION TO KUBERNETES & CONTAINER ORCHESTRATION

1.1 What is Kubernetes?

Kubernetes (K8s) is an **open-source container orchestration platform** that automates the deployment, scaling, and management of containerized applications.

✓ **Automates Deployment** – Manages application containers efficiently.
✓ **Scalability** – Dynamically scales applications based on demand.
✓ **Self-Healing** – Restarts failed containers automatically.
✓ **Load Balancing** – Distributes traffic efficiently across services.

1.2 Why Use Kubernetes?

✓ **Manages Complex Containerized Applications** – Handles multiple containers efficiently.
✓ **Ensures High Availability** – Detects failures and restarts services.
✓ **Enables Multi-Cloud & Hybrid Deployments** – Works across AWS, GCP, Azure, and on-premise environments.

📌 **Example:**
A **fintech company** uses Kubernetes to deploy and scale **real-time payment processing applications** across multiple cloud environments.

CHAPTER 2: KUBERNETES ARCHITECTURE

2.1 Key Components of Kubernetes

Component	Description
Master Node	Controls cluster operations, schedules workloads.
Worker Nodes	Runs application containers.
Pods	The smallest deployable unit containing one or more containers.
Deployments	Manages the lifecycle of application instances.
Services	Exposes applications internally or externally.
Ingress	Manages external access to services, such as HTTP/S routing.
ConfigMaps & Secrets	Store configuration data securely.

📌 **Example:**
An **e-commerce platform** runs **multiple microservices** inside Kubernetes **pods**, each handling different parts of the application, like payments, user authentication, and product inventory.

CHAPTER 3: SETTING UP KUBERNETES

3.1 Installing Kubernetes Locally (Minikube)

1. Install **Minikube** for local testing:
2. curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

3. chmod +x minikube-linux-amd64
4. sudo mv minikube-linux-amd64 /usr/local/bin/minikube
5. Start a local Kubernetes cluster:
6. minikube start

3.2 Setting Up Kubernetes on Google Cloud (GKE)

1. Enable **Google Kubernetes Engine (GKE)**:
2. gcloud services enable container.googleapis.com
3. Create a Kubernetes Cluster:
4. gcloud container clusters create my-cluster --num-nodes=3
5. Connect to the cluster:
6. gcloud container clusters get-credentials my-cluster

📌 **Example:**
A **SaaS company** sets up Kubernetes on **Google Kubernetes Engine (GKE)** to **automate deployments and manage workloads**.

CHAPTER 4: DEPLOYING APPLICATIONS IN KUBERNETES

4.1 Writing a Deployment YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
```

```
 selector:
  matchLabels:
   app: my-app
 template:
  metadata:
   labels:
    app: my-app
  spec:
   containers:
   - name: my-container
    image: my-app-image:v1
    ports:
    - containerPort: 8080
```

4.2 Applying a Deployment

kubectl apply -f deployment.yaml

4.3 Checking Deployment Status

kubectl get deployments

kubectl get pods

kubectl describe pod <pod-name>

📌 **Example:**
A **logistics company** deploys a **fleet tracking application** in Kubernetes with **multiple replicas** to ensure availability.

CHAPTER 5: KUBERNETES NETWORKING & LOAD BALANCING

5.1 Exposing Applications via Services

1. **ClusterIP (Internal Communication)**
2. kind: Service
3. apiVersion: v1
4. metadata:
5. name: my-service
6. spec:
7. selector:
8. app: my-app
9. ports:
10. - protocol: TCP
11. port: 80
12. targetPort: 8080
13. **LoadBalancer (External Access)**
14. kind: Service
15. apiVersion: v1
16. metadata:
17. name: my-app-service
18. spec:
19. type: LoadBalancer
20. selector:
21. app: my-app

```
    ports:
     - protocol: TCP
       port: 80
       targetPort: 8080
```

Checking Services

```
kubectl get services
```

📌 **Example:**
A **streaming platform** exposes services using **LoadBalancer Service** to serve **video content globally**.

CHAPTER 6: SCALING APPLICATIONS IN KUBERNETES

6.1 Horizontal Pod Autoscaling

```
kubectl autoscale deployment my-app --cpu-percent=50 --min=2 --max=10
```

6.2 Manual Scaling

```
kubectl scale deployment my-app --replicas=5
```

6.3 Checking Scaling Status

```
kubectl get hpa
```

📌 **Example:**
An **online shopping app** uses **Horizontal Pod Autoscaler (HPA)** to handle **Black Friday traffic spikes** dynamically.

CHAPTER 7: KUBERNETES SECURITY BEST PRACTICES

7.1 Using Role-Based Access Control (RBAC)

1. Define an RBAC Policy:
2. apiVersion: rbac.authorization.k8s.io/v1
3. kind: Role
4. metadata:
5. namespace: default
6. name: pod-reader
7. rules:
8. - apiGroups: [""]
9. resources: ["pods"]
10. verbs: ["get", "watch", "list"]
11. Apply the Role:
12. kubectl apply -f rbac.yaml

7.2 Using Secrets to Secure Credentials

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: YWRtaW4=  # Base64 encoded
  password: cGFzc3dvcmQ=
```

📌 **Example:**
A **banking system** uses **RBAC and Kubernetes Secrets** to **protect API credentials and user data**.

Chapter 8: Monitoring & Logging in Kubernetes

8.1 Monitoring Kubernetes Clusters with Prometheus

1. Install Prometheus in Kubernetes:
2. kubectl apply -f https://raw.githubusercontent.com/prometheus-operator/prometheus-operator/main/bundle.yaml
3. View Prometheus Dashboard:
4. kubectl port-forward svc/prometheus-k8s 9090:9090

8.2 Logging with Fluentd & Elasticsearch

1. Deploy Fluentd as a DaemonSet:
2. kubectl apply -f fluentd-daemonset.yaml
3. View Logs:
4. kubectl logs -f <pod-name>

📌 **Example:**
A **DevOps team** integrates **Prometheus & Fluentd** to **monitor Kubernetes clusters and analyze logs**.

Chapter 9: Exercise & Review Questions

Exercise:

1. **Deploy a Kubernetes Cluster** using Minikube or GKE.

2. **Create a Deployment & Scale It** using Kubernetes Autoscaler.
3. **Expose Your Application Using LoadBalancer Service.**
4. **Set Up RBAC & Secure Secrets** for sensitive data.
5. **Monitor Kubernetes Cluster** using Prometheus & Fluentd.

Review Questions:

1. What are the key components of a Kubernetes cluster?
2. How do **Deployments and Services** work in Kubernetes?
3. What is **Horizontal Pod Autoscaler (HPA)**?
4. How does **Kubernetes ensure security with RBAC & Secrets**?
5. What are **best practices for monitoring Kubernetes workloads**?

Conclusion: Mastering Kubernetes & Container Orchestration

✓ Kubernetes simplifies deploying and managing containerized applications.
✓ Supports scaling, self-healing, and automated deployments.
✓ Integrates with monitoring and security tools for optimized cloud operations.

CLOUD MONITORING & LOGGING FOR DEVOPS – STUDY MATERIAL

CHAPTER 1: INTRODUCTION TO CLOUD MONITORING & LOGGING

1.1 What is Cloud Monitoring & Logging?

Cloud Monitoring & Logging are essential DevOps practices for **observability, performance optimization, and troubleshooting** in cloud-based applications.

✓ **Cloud Monitoring** – Tracks application **health, latency, performance metrics, and system alerts**.
✓ **Cloud Logging** – Collects and stores **logs** from applications, containers, and infrastructure.
✓ **Incident Detection** – Helps detect **failures, security threats, and anomalies**.
✓ **Automated Alerts & Dashboards** – Provides **real-time notifications** and insights.

1.2 Why Use Cloud Monitoring & Logging in DevOps?

✓ **Ensures System Reliability** – Prevents downtime by identifying bottlenecks.
✓ **Improves Debugging** – Helps **troubleshoot production issues quickly**.
✓ **Supports Compliance & Security** – Logs system activities for **audit and compliance**.
✓ **Optimizes Performance** – Enables **real-time application performance monitoring**.

📌 **Example:**
An **e-commerce company** uses **Cloud Monitoring** to **track website**

response times and send alerts if checkout failures exceed a threshold.

CHAPTER 2: GOOGLE CLOUD MONITORING – OBSERVABILITY IN DEVOPS

2.1 Features of Google Cloud Monitoring

✓ **Infrastructure & Application Monitoring** – Monitors **VMs, Kubernetes, databases, APIs, and microservices**.
✓ **Metrics Collection** – Captures **CPU, memory, disk, and network usage**.
✓ **Custom Dashboards** – Displays real-time performance **visualizations**.
✓ **Alerting & Notifications** – Sends alerts via **email, Slack, or PagerDuty**.
✓ **Service-Level Objectives (SLOs)** – Helps define **uptime targets** for applications.

2.2 Enabling Cloud Monitoring

1. Open **Google Cloud Console** → Go to **Monitoring**.
2. Click **Enable Cloud Monitoring** for your project.
3. Install the **Ops Agent** on VM instances:
4. curl -sSO https://dl.google.com/cloudagents/add-google-cloud-ops-agent-repo.sh
5. sudo bash add-google-cloud-ops-agent-repo.sh --also-install
6. Navigate to **Metrics Explorer** → Select **Compute Engine Metrics**.

📌 **Example:**
A **DevOps team** configures **Google Cloud Monitoring** to track **server CPU and memory usage** on Kubernetes.

Chapter 3: Cloud Logging – Collecting & Analyzing Logs

3.1 What is Cloud Logging?

Cloud Logging **collects, stores, and analyzes** logs from Google Cloud services, Kubernetes, VMs, and applications.

✓ **Centralized Log Storage** – Aggregates logs from **GKE, VMs, and Cloud Run**.
✓ **Real-time Log Analysis** – Uses **Log Explorer** to filter and view logs.
✓ **Log-Based Alerts** – Detects **errors, security breaches, and failed requests**.
✓ **Export Logs** – Sends logs to **BigQuery, Pub/Sub, or Cloud Storage** for analysis.

3.2 Enabling Cloud Logging

1. Open **Google Cloud Console** → Go to **Logging**.
2. Click **Enable Logging** for your project.
3. Install the Logging Agent (for VMs):
4. sudo apt-get install google-fluentd
5. sudo service google-fluentd start
6. Use **Log Explorer** to filter logs by severity (e.g., ERROR, WARNING).

📌 **Example:**
A **banking company** configures **Cloud Logging** to monitor **failed transactions and security events**.

Chapter 4: Setting Up Monitoring Dashboards

4.1 Creating a Custom Dashboard

1. Open **Google Cloud Console** → Go to **Monitoring**.
2. Click **Dashboards** → **Create Dashboard**.
3. Add **Widgets** (Charts, Gauges, and Tables).
4. Select **Metrics** (e.g., CPU, memory, HTTP request latency).
5. Click **Save Dashboard**.

📌 **Example:**
A **media streaming service** creates a **dashboard tracking video buffering time** to detect streaming quality issues.

Chapter 5: Configuring Alerts & Notifications

5.1 Creating an Alert Policy

1. Open **Google Cloud Console** → Go to **Monitoring** → **Alerting**.
2. Click **Create Policy** → **Select a Metric** (e.g., High CPU Usage).
3. Set a **Threshold Condition** (e.g., **CPU > 80% for 5 min**).
4. Choose **Notification Channels** (Email, Slack, PagerDuty).
5. Click **Create Alert**.

📌 **Example:**
A **DevOps team** sets up **alerts to notify engineers on Slack** when **database response time exceeds 500ms**.

Chapter 6: Log-Based Metrics & Security Monitoring

6.1 Creating Log-Based Metrics

1. Open **Google Cloud Console** → Go to **Logging** → **Logs-Based Metrics**.
2. Click **Create Metric** → Enter a **Metric Name**.
3. Define **Log Filter** (e.g., HTTP 500 errors).
4. Save and visualize the metric in **Cloud Monitoring**.

6.2 Detecting Security Threats with Log-Based Alerts

✓ Monitor **unauthorized access attempts** (403 Forbidden).
✓ Track **API key misuse**.
✓ Detect **failed login attempts**.

📌 **Example:**
A **cybersecurity team** configures **log-based alerts** for **failed SSH login attempts on cloud servers**.

Chapter 7: Automating Monitoring & Logging with Terraform

7.1 Automating Monitoring Dashboard Creation

Create a Terraform script (monitoring.tf):

```
resource "google_monitoring_dashboard" "dashboard" {

 dashboard_json = <<EOT
```

```
{
  "displayName": "DevOps Monitoring",
  "widgets": [
   {
    "title": "CPU Usage",
    "xyChart": {
     "dataSets": [
      {
       "timeSeriesQuery": {
        "timeSeriesFilter": {
         "filter":
"metric.type=\"compute.googleapis.com/instance/cpu/utilization\""
        }
       }
      }
     ]
    }
   }
  ]
 }
 EOT
}
```

Deploy with Terraform:

terraform init

terraform apply

📌 **Example:**
A **DevOps engineer** automates **Cloud Monitoring setup** using **Terraform** to track Kubernetes pod failures.

CHAPTER 8: EXERCISE & REVIEW QUESTIONS

Exercise:

1. **Enable Cloud Monitoring** and configure a **CPU usage dashboard**.
2. **Set up Cloud Logging** and filter logs based on HTTP error codes.
3. **Create an alert policy** for high database response times.
4. **Automate dashboard creation** using Terraform.
5. **Export logs to BigQuery** for advanced analytics.

Review Questions:

1. What is the difference between **Cloud Monitoring and Cloud Logging**?
2. How do **alerts and notifications** improve system reliability?
3. What are **log-based metrics,** and how are they used?
4. How can **Terraform automate monitoring** in Google Cloud?
5. What are **best practices for monitoring microservices in Kubernetes**?

Conclusion: Ensuring DevOps Success with Cloud Monitoring & Logging

✓ Cloud Monitoring & Logging improve observability in DevOps workflows.

✓ Enables real-time performance tracking, troubleshooting, and security monitoring.

✓ Supports automation with Terraform and API integrations.

✓ Essential for maintaining high availability and system reliability.

🚀 **Mastering Cloud Monitoring & Logging ensures faster debugging, efficient DevOps practices, and optimized cloud operations!**

SITE RELIABILITY ENGINEERING (SRE) BEST PRACTICES

CHAPTER 1: INTRODUCTION TO SITE RELIABILITY ENGINEERING (SRE)

1.1 What is SRE?

Site Reliability Engineering (SRE) is a **discipline that applies software engineering principles to IT operations**. It focuses on improving **system reliability, availability, and scalability** through automation and monitoring.

1.2 Why is SRE Important?

✓ **Minimizes Downtime** – Reduces service outages and incidents.
✓ **Automates Operations** – Uses software engineering to manage infrastructure.
✓ **Improves Performance** – Enhances system availability and scalability.
✓ **Balances Reliability & Innovation** – Ensures stable deployments while enabling frequent releases.

1.3 Key Responsibilities of an SRE

✓ **Monitor system health & performance**.
✓ **Automate infrastructure and deployments**.
✓ **Manage incidents & reduce Mean Time to Recovery (MTTR)**.
✓ **Ensure Service Level Objectives (SLOs) are met**.
✓ **Optimize CI/CD pipelines**.

📌 **Example:**
A **global e-commerce company** implements **SRE best practices** to

ensure **99.99% uptime** and **automated failover** in case of regional failures.

CHAPTER 2: SRE PRINCIPLES & PRACTICES

2.1 SRE Core Principles

✓ **Service Level Indicators (SLIs)** – Metrics measuring system health (e.g., latency, error rate).
✓ **Service Level Objectives (SLOs)** – Targeted reliability goals (e.g., 99.95% availability).
✓ **Service Level Agreements (SLAs)** – Commitments to users regarding uptime and service quality.
✓ **Error Budgets** – Defines acceptable failure limits before slowing new releases.

2.2 SRE vs DevOps

Feature	SRE	DevOps
Focus	Reliability & uptime	Faster software delivery
Main Goal	Minimize downtime	Automate development & deployment
Metrics	SLIs, SLOs, error budgets	CI/CD efficiency, deployment frequency
Toolset	Monitoring, Incident Response	Continuous Integration/Deployment

📌 **Example:**
A **banking application** enforces **strict SLAs** (99.99% uptime) and **error budgets** to control software changes in production.

CHAPTER 3: MONITORING & OBSERVABILITY IN SRE

3.1 Observability Pillars

✓ **Metrics** – Measure system performance (e.g., CPU usage, request latency).
✓ **Logs** – Capture detailed event information for debugging.
✓ **Traces** – Track requests across distributed services.

3.2 Key Monitoring Tools in SRE

✓ **Google Cloud Operations Suite (Stackdriver)** – Provides logging, monitoring, and tracing.
✓ **Prometheus & Grafana** – Open-source monitoring and visualization tools.
✓ **ELK Stack (Elasticsearch, Logstash, Kibana)** – Log management and analytics.
✓ **Jaeger & OpenTelemetry** – Distributed tracing tools.

3.3 Setting Up Monitoring in Google Cloud

1. Enable **Cloud Monitoring**:
2. gcloud services enable monitoring.googleapis.com
3. Install **Prometheus for GKE**:
4. kubectl apply -f prometheus-config.yaml
5. Create **alerting policies** for latency and error rates.

📌 **Example:**
A **video streaming platform** uses **Google Cloud Monitoring** to track **latency spikes during peak traffic hours**.

CHAPTER 4: INCIDENT MANAGEMENT & RESPONSE

4.1 Steps in Incident Response

1. **Detection** – Use monitoring tools to identify issues.
2. **Triage** – Assess severity and impact.
3. **Mitigation** – Apply temporary fixes to restore service.
4. **Root Cause Analysis (RCA)** – Investigate the root cause.
5. **Postmortem & Continuous Improvement** – Document findings and improve processes.

4.2 Incident Severity Levels

Severity Level	Impact
SEV-1 (Critical)	Full system outage, requires immediate response.
SEV-2 (High)	Partial outage or major degradation.
SEV-3 (Medium)	Minor service issue, but still operational.
SEV-4 (Low)	Non-critical bug, performance optimization.

4.3 Incident Management Best Practices

✓ **Automate incident alerts** using **PagerDuty** or **Google Cloud Alerts**.
✓ **Use Runbooks** – Document predefined steps to resolve incidents quickly.
✓ **Conduct Postmortems** – Learn from failures to prevent recurrence.

📌 **Example:**
A **social media company** uses **automated alerts and runbooks** to respond to server failures in less than **5 minutes**.

CHAPTER 5: AUTOMATION & INFRASTRUCTURE AS CODE (IAC)

5.1 Automating Reliability Tasks

✓ **Auto-scaling** – Adjusts infrastructure based on demand.
✓ **Automated Deployments** – CI/CD pipelines for safer releases.
✓ **Configuration Management** – Tools like Terraform, Ansible, and Puppet.

5.2 Using Terraform for Google Cloud

Create an auto-scaling **GKE cluster** with Terraform:

```
resource "google_container_cluster" "primary" {
 name     = "my-cluster"
 location = "us-central1"

 node_pool {
  name       = "default-pool"
  node_count = 3
  machine_type = "e2-medium"
 }
}
```

Deploy infrastructure:

```
terraform init
terraform apply -auto-approve
```

📌 **Example:**
A **fintech company automates Kubernetes cluster provisioning** using Terraform to ensure **rapid disaster recovery**.

CHAPTER 6: SRE SECURITY BEST PRACTICES

6.1 Key Security Practices in SRE

✓ **Least Privilege Access** – Restrict permissions using IAM.
✓ **Encryption** – Encrypt data at rest and in transit.
✓ **Security Monitoring** – Use **Google Cloud Security Command Center** for threat detection.

6.2 Implementing IAM Best Practices

1. **Create a Service Account for an Application**:
2. gcloud iam service-accounts create app-service --display-name "App Service Account"
3. **Assign Least Privilege Role**:
4. gcloud projects add-iam-policy-binding my-project --member="serviceAccount:app-service@my-project.iam.gserviceaccount.com" --role="roles/viewer"

📌 **Example:**
A **healthcare company** follows **zero-trust principles** by enforcing **least privilege IAM policies** for sensitive patient data.

CHAPTER 7: SRE BEST PRACTICES & CULTURE

7.1 SRE Best Practices

✓ **Automate Everything** – Reduce manual work using scripts and infrastructure as code.
✓ **Measure SLIs & Enforce SLOs** – Define reliability metrics and set objectives.
✓ **Embrace Failure & Conduct Blameless Postmortems** – Learn

from incidents.

✓ **Prioritize Operational Load Management** – Balance development vs. on-call duties.

✓ **Implement Chaos Engineering** – Simulate failures to test system resilience.

📌 **Example:**

A **banking platform** runs **chaos engineering experiments** to **test how well their system handles sudden traffic spikes**.

Chapter 8: Exercise & Review Questions

Exercise:

1. **Set up a monitoring dashboard** in Google Cloud Operations Suite.
2. **Simulate an incident and create a response playbook**.
3. **Automate VM scaling using Terraform**.
4. **Define an error budget for a web application**.
5. **Deploy a security monitoring tool** in Google Cloud Security Command Center.

Review Questions:

1. What are **Service Level Indicators (SLIs),** and why are they important?
2. How does **SRE differ from DevOps**?
3. What are **key incident response best practices**?
4. How does **Terraform help with SRE automation**?
5. Why is **error budgeting critical in SRE**?

Conclusion: Building Reliable Systems with SRE

✓ SRE enhances reliability through automation, monitoring, and incident response.

✓ SLIs, SLOs, and error budgets ensure system stability.

✓ Automation & IaC reduce operational toil.

✓ Security and compliance are integral to reliable systems.

🚀 Mastering SRE best practices helps organizations build fault-tolerant, scalable, and efficient cloud infrastructure!

ASSIGNMENT

AUTOMATE INFRASTRUCTURE DEPLOYMENT USING TERRAFORM

SOLUTION: AUTOMATING INFRASTRUCTURE DEPLOYMENT USING TERRAFORM ON GCP

Step 1: Set Up Your Environment

1.1 Prerequisites

Before using Terraform, ensure you have:

✓ **Google Cloud Account** – With billing enabled.

✓ **Google Cloud SDK** – Installed and configured.

✓ **Terraform CLI** – Download and Install Terraform.

✓ **IAM Permissions** – Ensure you have Owner or Editor role to deploy resources.

1.2 Authenticate Terraform with Google Cloud

1️⃣ Create a Google Cloud Service Account for Terraform:

```
gcloud iam service-accounts create terraform-admin --display-name "Terraform Admin"
```

2️⃣ Assign IAM roles to the service account:

```
gcloud projects add-iam-policy-binding [PROJECT_ID] \
  --member="serviceAccount:terraform-admin@[PROJECT_ID].iam.gserviceaccount.com" \
  --role="roles/editor"
```

3️⃣ Generate a Service Account Key:

```
gcloud iam service-accounts keys create key.json \
  --iam-account=terraform-admin@[PROJECT_ID].iam.gserviceaccount.com
```

📌 **Example:**
A **DevOps engineer sets up Terraform authentication** to automate infrastructure deployment in GCP.

Step 2: Write a Terraform Configuration File

Create a new Terraform project directory and navigate into it:

mkdir gcp-terraform && cd gcp-terraform

Create a new file main.tf and define GCP resources.

2.1 Define the GCP Provider

```
provider "google" {
 credentials = file("key.json")
 project     = "my-gcp-project"
 region      = "us-central1"
}
```

2.2 Create a Compute Engine Virtual Machine

```
resource "google_compute_instance" "vm_instance" {
 name         = "terraform-vm"
 machine_type = "e2-medium"
 zone         = "us-central1-a"

 boot_disk {
  initialize_params {
   image = "debian-cloud/debian-10"
```

```
  }
 }

 network_interface {
  network = "default"
  access_config {}
 }
}
```

2.3 Define a Google Cloud Storage Bucket

```
resource "google_storage_bucket" "terraform_bucket" {
 name     = "terraform-gcs-bucket-12345"
 location = "US"
}
```

📌 **Example:**
A **startup automates VM and Storage Bucket deployment** using Terraform instead of manual creation.

Step 3: Deploy Infrastructure with Terraform

3.1 Initialize Terraform

Run the following command to download provider plugins and initialize the project:

```
terraform init
```

3.2 Preview Changes

Before applying, review the changes Terraform will make:

terraform plan

3.3 Apply the Terraform Configuration

To create resources, run:

terraform apply -auto-approve

✓ Terraform will output the resources it created:

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

📌 **Example:**
A **cloud administrator automates VM provisioning** for development environments using Terraform.

Step 4: Managing and Updating Infrastructure

4.1 Modify the Infrastructure

Modify the machine_type in main.tf from e2-medium to e2-standard-2.

Apply the changes:

terraform apply -auto-approve

✓ Terraform will detect the change and update the VM instance.

4.2 Destroy Infrastructure

To remove all resources:

terraform destroy -auto-approve

📌 **Example:**
A **finance company automates infrastructure scaling and removal** using Terraform.

Step 5: Using Terraform Modules for Reusability

Terraform modules allow reusability across multiple projects.

5.1 Create a Terraform Module for VMs

Create a directory modules/vm/ and a file main.tf:

```
resource "google_compute_instance" "vm" {
 name       = var.vm_name
 machine_type = var.machine_type
 zone       = var.zone

 boot_disk {
  initialize_params {
   image = "debian-cloud/debian-10"
  }
 }

 network_interface {
  network = "default"
  access_config {}
 }
}
```

Create variables.tf:

```
variable "vm_name" {}

variable "machine_type" {}

variable "zone" {}
```

Use the module in main.tf:

```
module "vm_instance" {
  source       = "./modules/vm"
  vm_name      = "web-server"
  machine_type = "e2-medium"
  zone         = "us-central1-a"
}
```

📌 **Example:**
A **large enterprise uses Terraform modules** to create reusable **infrastructure blueprints**.

Step 6: Terraform Remote State Storage in GCS

6.1 Store State in Google Cloud Storage

1️⃣ Create a GCS bucket for Terraform state:

```
gsutil mb gs://terraform-state-bucket-1234/
```

2️⃣ Modify backend.tf to use GCS for remote state storage:

```
terraform {
  backend "gcs" {
    bucket = "terraform-state-bucket-1234"
    prefix = "terraform/state"
```

}

}

Reinitialize Terraform:

terraform init

📌 **Example:**
A **team of DevOps engineers shares Terraform state** using GCS for collaboration.

Step 7: CI/CD Automation for Terraform with Cloud Build

7.1 Automate Terraform Deployment Using Cloud Build

Create a cloudbuild.yaml file:

steps:

- name: "hashicorp/terraform"

  args: ["init"]

- name: "hashicorp/terraform"

  args: ["plan"]

- name: "hashicorp/terraform"

  args: ["apply", "-auto-approve"]

Run Terraform automatically:

gcloud builds submit --config cloudbuild.yaml

📌 **Example:**
A **software company integrates Terraform with CI/CD pipelines** for automated infrastructure updates.

Conclusion: Automating Infrastructure with Terraform on GCP

✓ Terraform automates GCP infrastructure provisioning using declarative code.

✓ Modules and remote state storage improve efficiency and collaboration.

✓ CI/CD integration enables automated and continuous infrastructure updates.

🚀 Mastering Terraform helps teams build scalable, reproducible, and cost-effective cloud environments!

DEPLOY AND MANAGE CONTAINERS USING GKE

SOLUTION: DEPLOY AND MANAGE CONTAINERS USING GOOGLE KUBERNETES ENGINE (GKE)

Step 1: Set Up Google Cloud and Enable GKE

1.1 Create a Google Cloud Project

If you don't have a project, create one using:

gcloud projects create my-gke-project --set-as-default

Set your project:

gcloud config set project my-gke-project

1.2 Enable GKE API

To use GKE, enable the required APIs:

gcloud services enable container.googleapis.com

📌 **Example:**
A **DevOps team** creates a **new project in GCP** to deploy a containerized microservices application.

Step 2: Create a Kubernetes Cluster on GKE

2.1 Create a GKE Cluster

Run the following command to create a **3-node cluster**:

gcloud container clusters create my-cluster --num-nodes=3 --region=us-central1

2.2 Authenticate & Connect to the Cluster

gcloud container clusters get-credentials my-cluster --region=us-central1

2.3 Verify the Cluster

kubectl get nodes

📌 **Example:**
A **gaming company** sets up a **3-node GKE cluster** to host a **real-time multiplayer game backend**.

Step 3: Deploy a Containerized Application on GKE

3.1 Create a Deployment YAML File

Save the following YAML as deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: my-app
spec:
 replicas: 3
 selector:
  matchLabels:
   app: my-app
 template:
  metadata:
   labels:
```

```
    app: my-app
  spec:
   containers:
   - name: my-container
    image: gcr.io/my-gke-project/my-app:latest
    ports:
    - containerPort: 8080
```

3.2 Apply the Deployment

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

3.3 Verify the Deployment

```
kubectl get deployments
kubectl get pods
```

📌 **Example:**
A **news website** deploys its **containerized web app** to GKE for **high availability and scalability**.

Step 4: Expose the Application Using a Service

4.1 Create a Service YAML File

Save the following YAML as service.yaml:

```
apiVersion: v1
kind: Service
metadata:
 name: my-app-service
```

```
spec:
 type: LoadBalancer
 selector:
  app: my-app
 ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

4.2 Apply the Service

```
kubectl apply -f service.yaml
```

4.3 Get the External IP

```
kubectl get service my-app-service
```

📌 **Example:**
A **retail company** exposes its **e-commerce site** through a **LoadBalancer service** for external access.

Step 5: Scaling the Deployment

5.1 Manually Scale the Application

```
kubectl scale deployment my-app --replicas=5
```

5.2 Enable Auto-Scaling

```
kubectl autoscale deployment my-app --cpu-percent=50 --min=2 --max=10
```

5.3 Verify Auto-Scaling

kubectl get hpa

📌 **Example:**
A **food delivery startup** enables **autoscaling on GKE** to **handle peak demand during lunch and dinner hours**.

Step 6: Updating & Rolling Back Deployments

6.1 Update the Deployment

Modify the **image version** in deployment.yaml:

```
image: gcr.io/my-gke-project/my-app:v2
```

Apply the update:

kubectl apply -f deployment.yaml

6.2 Rollback to a Previous Version

kubectl rollout undo deployment my-app

📌 **Example:**
A **financial services company** rolls back an update after detecting **API failures in production**.

Step 7: Monitoring & Logging in GKE

7.1 Enable GKE Logging & Monitoring

gcloud services enable monitoring.googleapis.com logging.googleapis.com

7.2 View Logs Using kubectl

kubectl logs -f <pod-name>

7.3 Enable Stackdriver Logging

gcloud logging read "resource.type=gke_cluster" --limit=5

📌 **Example:**
A **security firm** monitors logs in **Cloud Logging** to detect **suspicious activities on its Kubernetes workloads**.

Step 8: Deleting the GKE Cluster (Optional)

To delete the cluster and free up resources:

gcloud container clusters delete my-cluster --region=us-central1

📌 **Example:**
A **startup running a hackathon** deletes its **temporary GKE cluster** after the event.

Summary of GKE Deployment Steps

Step	Command
Create GKE Cluster	gcloud container clusters create my-cluster --num-nodes=3 --region=us-central1
Authenticate Cluster	gcloud container clusters get-credentials my-cluster
Deploy an App	kubectl apply -f deployment.yaml
Expose the App	kubectl apply -f service.yaml
Get External IP	kubectl get service my-app-service
Scale Up	kubectl scale deployment my-app --replicas=5
Enable Auto-Scaling	kubectl autoscale deployment my-app --cpu-percent=50 --min=2 --max=10

Update Deployment	kubectl apply -f deployment.yaml
Rollback Deployment	kubectl rollout undo deployment my-app
View Logs	kubectl logs -f <pod-name>
Delete Cluster	gcloud container clusters delete my-cluster --region=us-central1

Conclusion: Mastering GKE for Containerized Workloads

✓ GKE simplifies the deployment and scaling of containerized applications.

✓ Supports auto-scaling, monitoring, and rolling updates for production environments.

✓ Integrates seamlessly with other GCP services for a fully managed cloud-native solution.

🚀 Mastering GKE helps businesses deploy, scale, and manage applications efficiently in a cloud-native ecosystem!