Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

GOOGLE APP ENGINE – DEPLOYING SCALABLE WEB APPLICATIONS

CHAPTER 1: INTRODUCTION TO GOOGLE APP ENGINE (GAE)

1.1 What is Google App Engine?

Google App Engine (GAE) is a **fully managed platform-as-a-service (PaaS)** that enables developers to **deploy, manage, and scale web applications** without managing the underlying infrastructure.

✓ **Automatic Scaling** – Adjusts resources based on traffic.
✓ **Supports Multiple Languages** – Python, Java, Node.js, Go, PHP, Ruby, .NET.
✓ **Built-in Security & Monitoring** – Integrates with Google Cloud Logging & IAM.
✓ **Zero Server Management** – Fully managed deployment and maintenance.

1.2 When to Use Google App Engine?

✓ Hosting **web applications and APIs** with **dynamic scaling**.
✓ Deploying **machine learning models as REST APIs**.
✓ Running **backend services** for mobile apps.
✓ Creating **lightweight microservices**.

📌 **Example:**
A **social media startup** deploys its **real-time messaging application** using App Engine to **automatically scale** during peak hours.

Chapter 2: Google App Engine Architecture

2.1 Key Components of App Engine

Component	Function
App Engine Standard Environment	Preconfigured runtime, automatic scaling, quick deployment.
App Engine Flexible Environment	Custom runtime, containerized workloads, supports more libraries.
App Engine Services	Organizes applications into logical units (e.g., frontend, backend services).
App Engine Versions	Supports versioning for rollback and testing.
App Engine Scaling	Autoscaling, basic scaling, and manual scaling options.

📌 **Example:**
A **news website** hosts its **content delivery backend** using **App Engine Standard**, ensuring **fast response times** with auto-scaling.

Chapter 3: Setting Up Google App Engine

3.1 Prerequisites for App Engine Deployment

✓ A **Google Cloud Platform (GCP) account** (Sign up here).
✓ Install **Google Cloud SDK** and set up gcloud.
✓ Enable **App Engine API** in your GCP project.

3.2 Steps to Create an App Engine Application

Step 1: Create a GCP Project

1. Open Google Cloud Console.
2. Click **Select Project → Create New Project**.
3. Enter **Project Name** (e.g., my-app-engine-project).
4. Click **Create**.

Step 2: Enable App Engine

1. Navigate to **App Engine → Dashboard**.
2. Click **Create Application**.
3. Select **Region** (e.g., us-central).
4. Choose **Standard or Flexible Environment**.

📌 **Example:**
A **freelance developer** creates an App Engine project to **host an API for a chatbot**.

CHAPTER 4: DEPLOYING A WEB APPLICATION TO APP ENGINE

4.1 Deploying a Simple Web App Using App Engine Standard

Step 1: Install Google Cloud SDK

gcloud components install app-engine-python

gcloud init

Step 2: Create a Python Web Application

1. Create a new folder:
2. mkdir my-app && cd my-app

3. Add a simple Python web server (main.py):

```
4. from flask import Flask
5.
6. app = Flask(__name__)
7.
8. @app.route('/')
9. def hello():
10.         return 'Hello, App Engine!'
11.
12.     if __name__ == '__main__':
13.  app.run(host='0.0.0.0', port=8080)
14.     Create an app.yaml file:
15.runtime: python39
16.     entrypoint: gunicorn -b :$PORT main:app
17.Deploy the application:
18.     gcloud app deploy
19.     Open the deployed app in a browser:
20.     gcloud app browse
```

📌 **Example:**
A **blogging platform** deploys its **web app using Flask** on **App Engine Standard**.

CHAPTER 5: SCALING & MANAGING APP ENGINE APPLICATIONS

5.1 Scaling Options in App Engine

Scaling Type	Description	Best For
Automatic Scaling	Increases or decreases instances based on traffic.	Web applications with variable traffic.
Basic Scaling	Starts instances based on request load, stops when idle.	Applications with intermittent traffic.
Manual Scaling	Runs a fixed number of instances continuously.	Background processing, API backends.

Enable Autoscaling in app.yaml

```
automatic_scaling:
  min_instances: 1
  max_instances: 5
  target_cpu_utilization: 0.6
```

📌 **Example:**
A **ticket booking system** uses **automatic scaling** to handle **high traffic spikes** during events.

CHAPTER 6: SECURING AN APP ENGINE APPLICATION

6.1 Enforcing HTTPS

Ensure secure connections by forcing HTTPS:

```
handlers:
- url: /.*
  secure: always
```

script: auto

6.2 Restricting Access with IAM

1. Open **Google Cloud Console → IAM & Admin**.
2. Assign **roles** (App Engine Admin, Viewer, etc.).
3. Use **Google Identity-Aware Proxy (IAP)** for additional security.

6.3 Using Google Secret Manager

Secure API keys, database passwords, and sensitive data using:

gcloud secrets create db-password --replication-policy="automatic"

gcloud secrets add-iam-policy-binding db-password --member user:admin@example.com --role roles/secretmanager.secretAccessor

📌 **Example:**
A **finance app** enforces HTTPS and **restricts API access using IAP**.

CHAPTER 7: MONITORING & DEBUGGING APP ENGINE APPLICATIONS

7.1 Monitoring with Google Cloud Logging & Stackdriver

1. Open **Google Cloud Console → Operations → Logging**.
2. Use **Log Explorer** to filter logs for errors.

7.2 Debugging with Cloud Debugger

1. Navigate to **Operations → Debugger**.
2. Select application and set **breakpoints** for real-time debugging.

7.3 Performance Optimization

✓ Enable **Cloud CDN** for caching.

✓ Use **Memorystore (Redis)** for session storage.

✓ Optimize **database queries** for better response time.

📌 **Example:**

An **e-commerce store** monitors app logs in **Cloud Logging** to detect slow queries.

CHAPTER 8: EXERCISE & REVIEW QUESTIONS

Exercise:

1. **Deploy a Python web app** on App Engine Standard.
2. **Configure auto-scaling** with app.yaml.
3. **Set up IAM roles** for restricted app access.
4. **Enable Cloud Logging** to monitor application activity.

Review Questions:

1. What is the difference between **Standard and Flexible environments**?
2. How does **automatic scaling work in App Engine**?
3. Why is **Google Identity-Aware Proxy (IAP)** important?
4. How can **Cloud Debugger** help in troubleshooting App Engine applications?
5. What are **best practices for securing App Engine web apps**?

CONCLUSION: DEPLOYING SCALABLE APPLICATIONS WITH GOOGLE APP ENGINE

✓ **Google App Engine simplifies web application deployment** by eliminating server management.

✓ **Automatic scaling & security features** ensure reliability.

✓ **Integration with Google Cloud services** makes it a powerful choice for modern applications.

🚀 **Mastering App Engine enables developers to build scalable, secure, and high-performance cloud applications!**

GOOGLE CLOUD FUNCTIONS – SERVERLESS EVENT-DRIVEN APPLICATIONS

CHAPTER 1: INTRODUCTION TO GOOGLE CLOUD FUNCTIONS

1.1 What is Google Cloud Functions?

Google Cloud Functions (GCF) is a **serverless, event-driven computing platform** that allows developers to run code in response to **events from Google Cloud services, HTTP requests, or external triggers** without managing servers.

✓ **Serverless Execution** – No need to manage infrastructure.
✓ **Event-Driven** – Responds to real-time events like **Cloud Storage updates, Pub/Sub messages, and HTTP requests**.
✓ **Auto-Scaling** – Scales up and down based on demand.
✓ **Pay-as-You-Go** – Costs are incurred only when the function runs.

📌 **Example:**
A **logistics company** uses Cloud Functions to **automatically resize images** when a user uploads them to Cloud Storage.

CHAPTER 2: CLOUD FUNCTIONS ARCHITECTURE & COMPONENTS

2.1 How Cloud Functions Work

1. **Event Occurs** – Triggered by an **HTTP request, Pub/Sub event, Cloud Storage upload, Firestore update, or Cloud Scheduler event**.
2. **Function Executes** – Runs serverless code to **process the event**.

3. **Response or Action** – The function **returns data** or triggers another **GCP service** (e.g., logs data to BigQuery).

2.2 Key Components

Component	Description
Trigger	Specifies what event invokes the function (HTTP request, Cloud Storage, Pub/Sub, etc.).
Function Code	Contains the logic to process the event.
Execution Environment	Supports Node.js, Python, Go, Java, .NET, Ruby, and PHP.
IAM Permissions	Controls who can invoke and manage the function.

📌 **Example:**
An **e-commerce platform** uses Cloud Functions to **send automated email notifications** when a new order is placed.

CHAPTER 3: DEPLOYING A CLOUD FUNCTION

3.1 Prerequisites

✓ **Google Cloud Account** – Sign up at Google Cloud Console.
✓ **Enable Cloud Functions API** – In **Cloud Console**, go to **APIs & Services → Enable Cloud Functions API**.
✓ **Install Google Cloud SDK (CLI) (Optional)**:

```
curl https://sdk.cloud.google.com | bash

gcloud init
```

3.2 Deploying a Cloud Function Using Cloud Console

1. Navigate to **Google Cloud Console → Cloud Functions**.
2. Click **Create Function**.
3. Set **Function Name** (e.g., hello-world).
4. Select **Trigger Type**:
    - **HTTP Trigger** – Responds to web requests.
    - **Cloud Storage Trigger** – Runs when a file is uploaded or deleted.
    - **Pub/Sub Trigger** – Responds to Pub/Sub messages.
5. Choose **Runtime** (e.g., Node.js, Python, Go).
6. Enter function code or upload a ZIP file.
7. Click **Deploy**.

📌 **Example:**
A **news website** deploys a Cloud Function that **fetches real-time stock market data when triggered by an HTTP request**.

3.3 Deploying a Cloud Function Using gcloud CLI

1. **Create a Python Cloud Function (main.py)**:
2. def hello_world(request):
3. return "Hello, Google Cloud Functions!"
4. **Deploy the function via CLI**:
5. gcloud functions deploy hello-world \
6. --runtime python39 \
7. --trigger-http \

8. --allow-unauthenticated
9. **Get Function URL & Test It**:
10. gcloud functions describe hello-world --format='value(httpsTrigger.url)'

📌 **Example:**
A **healthcare startup** deploys a **Flask-based Cloud Function** to provide **real-time symptom checker APIs**.

CHAPTER 4: CLOUD FUNCTIONS TRIGGERS & USE CASES

4.1 Cloud Function Trigger Types

Trigger Type	Use Case Example
HTTP Trigger	REST API endpoints, webhook processing
Cloud Pub/Sub Trigger	Event-driven data processing (e.g., log processing, notifications)
Cloud Storage Trigger	Image resizing, virus scanning on file uploads
Firestore Trigger	Real-time notifications when Firestore documents change
Cloud Scheduler Trigger	Automated tasks like nightly database backups

📌 **Example:**
A **social media app** triggers a **Cloud Function using Pub/Sub** whenever a user uploads a photo.

4.2 Example: Cloud Function for Image Resizing (Cloud Storage Trigger)

1. **Create a Python Cloud Function (main.py)**:
2. from google.cloud import storage
3. from PIL import Image
4. import io
5.
6. def resize_image(event, context):
7. client = storage.Client()
8. bucket = client.get_bucket(event['bucket'])
9. blob = bucket.get_blob(event['name'])
10.
11. image = Image.open(io.BytesIO(blob.download_as_bytes()))
12. image = image.resize((300, 300))
13.
14. output_blob = bucket.blob(f"resized-{event['name']}")
15. output_blob.upload_from_string(image.tobytes(), content_type='image/jpeg')
16. **Deploy Function**:
17. gcloud functions deploy resize-image \
18. --runtime python39 \
19. --trigger-resource my-image-bucket \
20. --trigger-event google.storage.object.finalize

📌 **Example:**
A **real estate company** resizes property images automatically when uploaded to Cloud Storage.

Chapter 5: Securing & Monitoring Cloud Functions

5.1 Security Best Practices

✓ **Restrict Public Access** – Use **IAM roles** to limit invocation rights:

```
gcloud functions add-iam-policy-binding my-function \
  --member=user:your-email@example.com \
  --role=roles/cloudfunctions.invoker
```

✓ **Use Environment Variables for Secrets** – Avoid hardcoding secrets:

```
gcloud functions deploy my-function --set-env-vars SECRET_KEY=abcd1234
```

✓ **Enable VPC Access** – Connect to private resources securely.

📌 **Example:**
A **fintech startup** uses **IAM restrictions and environment variables** to secure API keys in Cloud Functions.

5.2 Monitoring & Logging with Stackdriver (Cloud Operations)

✓ **Enable Cloud Logging** for Function Execution Logs:

```
gcloud logging read "resource.type=cloud_function"
```

✓ **View Logs in Cloud Console → Operations → Logging**.
✓ **Set Alerts for Function Failures** using **Cloud Monitoring**.

📌 **Example:**
A **cybersecurity firm** sets up **Cloud Monitoring alerts** to detect failed authentication attempts in their API.

Chapter 6: Case Study – Automating Order Processing with Cloud Functions

Problem Statement:

An **e-commerce company** wants to **automate order fulfillment** when a customer places an order.

Solution Implementation:

✓ **Trigger:** Cloud Pub/Sub receives a new order event.
✓ **Function Execution:** Cloud Function reads the order details and updates Firestore.
✓ **Response:** Notifies the shipping team via Cloud Pub/Sub.

Results:

✓ **Reduced order processing time by 80%**.
✓ **Automated inventory updates** using Firestore triggers.
✓ **Eliminated manual workflows, improving efficiency**.

📌 **Key Takeaways:**
✓ **Cloud Functions provide serverless automation**.
✓ **Pub/Sub & Firestore triggers streamline order workflows**.
✓ **Logging & monitoring ensure reliability**.

Chapter 7: Exercise & Review Questions

Exercise:

1. **Deploy a Cloud Function** that sends an email when a file is uploaded to Cloud Storage.
2. **Secure an HTTP function** by restricting access with IAM.
3. **Set up logging and monitoring** for function execution failures.
4. **Integrate Cloud Functions with Pub/Sub** for real-time event processing.

Review Questions:

1. What are the benefits of **Cloud Functions over Compute Engine VMs**?
2. How do **Cloud Pub/Sub triggers** work in event-driven applications?
3. What security best practices should be followed in **Cloud Functions**?
4. How does **Cloud Logging help in debugging function execution**?
5. What are the key differences between **Cloud Functions and Cloud Run**?

CONCLUSION

✓ Google Cloud Functions enable serverless event-driven applications.
✓ They seamlessly integrate with GCP services like Cloud Storage, Pub/Sub, and Firestore.
✓ Security and monitoring are critical for production workloads.

🚀 **Next Steps:**

✓ Deploy a **Cloud Function with an event trigger**.

✓ Secure the function with **IAM & VPC Service Controls**.

✓ Monitor logs and optimize execution performance! 🚀

GOOGLE CLOUD RUN – DEPLOYING CONTAINERIZED APPLICATIONS

CHAPTER 1: INTRODUCTION TO GOOGLE CLOUD RUN

What is Google Cloud Run?

Google Cloud Run is a **fully managed, serverless platform** that allows developers to **deploy and run containerized applications** without worrying about infrastructure management. It is built on **Knative** and provides automatic **scalability, load balancing, and cost efficiency**.

Key Features of Cloud Run

✓ **Serverless:** No infrastructure management—Google automatically scales your app.
✓ **Containerized Deployments:** Supports any language or framework via containers.
✓ **Automatic Scaling:** Scales **up to meet demand** and **down to zero** when idle.
✓ **Pay-per-use:** Only pay for the exact compute time used.
✓ **Secure & Fast:** Provides built-in **HTTPS, IAM-based authentication, and networking controls**.

📌 **Example:**
A **start-up** deploys a **Node.js backend API** using Cloud Run, allowing automatic scaling during peak traffic and reducing costs when inactive.

CHAPTER 2: CLOUD RUN VS OTHER COMPUTE OPTIONS IN GCP

2.1 Comparing Cloud Run with Other GCP Compute Services

Feature	Cloud Run	App Engine	GKE (Kubernetes Engine)	Compute Engine
Infrastructure	Fully managed, serverless	Fully managed, but app-specific	Managed Kubernetes clusters	VM-based (manual scaling)
Scalability	Automatic, down to zero	Automatic	Needs configuration	Manual autoscaling
Customization	Any language in a container	Limited to specific runtimes	Full control over k8s workloads	Full OS-level control
Pricing Model	Pay per request (milliseconds)	Pay per instance	Pay for running cluster nodes	Pay for running VMs
Best for	Stateless APIs, microservices	Web apps, standard runtimes	Large-scale containerized workloads	Custom infrastructure needs

📌 **Example:**

A **small e-commerce business** chooses **Cloud Run for deploying APIs**, while a **large enterprise** prefers **GKE for multi-cluster Kubernetes workloads**.

CHAPTER 3: SETTING UP CLOUD RUN & DEPLOYING A CONTAINERIZED APPLICATION

3.1 Prerequisites

✓ Install **Google Cloud SDK** (gcloud)
✓ Install **Docker** for building container images
✓ Enable **Cloud Run API** in Google Cloud Console

3.2 Step-by-Step: Deploying a Containerized App to Cloud Run

Step 1: Create a Simple Web App (Python Flask Example)

Create a main.py file:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "Hello from Google Cloud Run!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8080)
```

Create a requirements.txt:

```
flask
gunicorn
```

Step 2: Create a Dockerfile

Create a Dockerfile to containerize the app:

```
# Use an official Python runtime as a parent image

FROM python:3.9


# Set the working directory

WORKDIR /app


# Copy the application files

COPY . .


# Install dependencies

RUN pip install -r requirements.txt


# Define environment variable

ENV PORT=8080


# Run the application

CMD ["gunicorn", "-b", "0.0.0.0:8080", "main:app"]
```

Step 3: Build and Push the Docker Image

1. **Authenticate with Google Cloud:**
2. gcloud auth configure-docker
3. **Build the Docker image:**
4. docker build -t gcr.io/YOUR_PROJECT_ID/my-cloud-run-app .
5. **Push the image to Google Container Registry (GCR):**
6. docker push gcr.io/YOUR_PROJECT_ID/my-cloud-run-app

Step 4: Deploy to Cloud Run

1. **Enable Cloud Run API (if not enabled):**
2. gcloud services enable run.googleapis.com
3. **Deploy the containerized app to Cloud Run:**
4. gcloud run deploy my-cloud-run-app \
5. --image gcr.io/YOUR_PROJECT_ID/my-cloud-run-app \
6. --platform managed \
7. --region us-central1 \
8. --allow-unauthenticated
9. **Output Example:**
10. Deploying...done
11. Service URL: https://my-cloud-run-app-xyz.a.run.app
12. **Test the deployment:**
13. curl https://my-cloud-run-app-xyz.a.run.app

📌 **Example:**

A **real estate company** deploys an **AI-based property**

recommendation API to Cloud Run, ensuring **high availability with minimal cost**.

Chapter 4: Securing & Optimizing Cloud Run Applications

4.1 Controlling Access with IAM

✓ **Restrict public access**:

gcloud run services update my-cloud-run-app --no-allow-unauthenticated

✓ **Grant IAM access to specific users:**

gcloud projects add-iam-policy-binding YOUR_PROJECT_ID \

--member=user:developer@example.com \

--role=roles/run.invoker

📌 **Example:**
A **finance company** restricts Cloud Run access **only to authorized users** to prevent unauthorized transactions.

4.2 Configuring Autoscaling & Concurrency

✓ **Limit concurrent requests per instance (default is 80):**

gcloud run services update my-cloud-run-app --concurrency=10

✓ **Limit minimum and maximum instances:**

gcloud run services update my-cloud-run-app --min-instances=1 --max-instances=5

📌 **Example:**
A **news website** sets **min-instances=1** to keep the API warm and **max-instances=100** to handle sudden traffic surges.

Chapter 5: Integrating Cloud Run with Other GCP Services

Service	Integration Use Case
Cloud SQL	Connect Cloud Run with relational databases
Cloud Storage	Store images, files, and logs
Pub/Sub	Event-driven architecture (e.g., trigger functions on new messages)
Firestore/BigQuery	Store real-time user data & analytics
Secret Manager	Securely store API keys and credentials

📌 **Example:**
A **ride-sharing app** stores trip data in **Cloud SQL**, processes events using **Pub/Sub**, and logs analytics in **BigQuery**.

Chapter 6: Case Study – Deploying a Serverless Microservices API

Problem Statement:

A **healthcare startup** needs to deploy an **AI-powered symptom checker API** with the following requirements:

✓ **Scalable API without managing servers**

✓ **Connects to Cloud SQL for patient records**

✓ **Secure access using IAM policies**

Solution Implementation:

1. **Containerized the AI-based symptom checker API** in Cloud Run.
2. **Connected Cloud Run with Cloud SQL** using a private VPC connector.
3. **Enabled IAM-based authentication** to restrict API access.
4. **Configured autoscaling** for **1000 concurrent requests per second**.

Results:

✓ **99.99% uptime** due to Cloud Run's scalability.
✓ **60% cost reduction** compared to Compute Engine.
✓ **Improved security with IAM & private networking**.

CHAPTER 7: REVIEW QUESTIONS & EXERCISES

Exercise:

1. **Deploy a Cloud Run service** using a sample container.
2. **Restrict public access** and allow only IAM-authenticated users.
3. **Integrate Cloud Run with Cloud Storage** to store uploaded files.
4. **Set up autoscaling policies** for your Cloud Run service.

Review Questions:

1. What are the advantages of **Cloud Run over Compute Engine**?
2. How does **Cloud Run autoscaling work**?
3. What are the best practices for **securing a Cloud Run service**?

4. How can you connect **Cloud Run to a database** securely?

Conclusion: Building Scalable Applications with Cloud Run

Google Cloud Run is an **ideal solution for deploying containerized applications** with **zero infrastructure management, auto-scaling, and seamless GCP integrations**. By following best practices, developers can **build secure, cost-effective, and high-performance applications**. 🚀

Cloud Build – Automating Deployment Pipelines in Google Cloud

Chapter 1: Introduction to Cloud Build

What is Cloud Build?

Cloud Build is a **fully managed continuous integration and continuous delivery (CI/CD) service** in **Google Cloud** that automates the **build, test, and deployment** of applications. It allows developers to **create, test, and deploy software efficiently** across various **Google Cloud services and third-party platforms**.

Key Benefits of Cloud Build

✓ **Fully Managed** – Google Cloud handles infrastructure setup and scaling.
✓ **Multi-Environment Deployment** – Supports **GKE, App Engine, Cloud Run, Cloud Functions**.
✓ **Parallel Execution** – Builds, tests, and deploys in parallel for speed.
✓ **Security & Compliance** – Integrates with **IAM, Cloud Audit Logs, and Artifact Registry**.
✓ **Extensible** – Supports **custom build steps, Docker images, and open-source tools**.

📌 **Example:**
A **SaaS company** uses **Cloud Build to automate deployment** of its web application to **Google Kubernetes Engine (GKE)** every time new code is pushed to **GitHub**.

Chapter 2: Cloud Build Architecture & Workflow

2.1 How Cloud Build Works

1️⃣ **Code Push** → Developers push code to **GitHub, GitLab, or Cloud Source Repositories**.

2️⃣ **Trigger Cloud Build** → A **build trigger** detects changes and starts a new build.

3️⃣ **Build Execution** → Cloud Build runs build steps defined in a **YAML file (cloudbuild.yaml)**.

4️⃣ **Testing & Artifact Storage** → Runs tests and stores **Docker images, binaries** in **Artifact Registry**.

5️⃣ **Deployment** → Deploys applications to **App Engine, GKE, Cloud Run, or Compute Engine**.

2.2 Components of Cloud Build

Component	Description
Cloud Build Triggers	Automatically starts builds on code changes.
Cloud Build Config File	Defines build steps (cloudbuild.yaml).
Build Steps	Tasks like running tests, building images, and deploying apps.
Artifact Registry	Stores container images and binaries.
Cloud IAM	Controls access to Cloud Build resources.

📌 **Example:**

An **e-commerce company** automates deployments by configuring **Cloud Build Triggers** to **detect new commits in GitHub** and deploy changes to **Cloud Run**.

CHAPTER 3: SETTING UP CLOUD BUILD FOR CI/CD

3.1 Prerequisites

✓ **Enable Cloud Build API** in **Google Cloud Console**.
✓ **Grant IAM Roles** (roles/cloudbuild.builds.editor) to developers.
✓ **Create a GitHub or Cloud Source Repository** for version control.

3.2 Creating a Cloud Build Configuration File (cloudbuild.yaml)

A **Cloud Build configuration file** (cloudbuild.yaml) defines **build steps, execution order, and deployment commands**.

Example: Building a Docker Image & Deploying to Cloud Run

```
steps:
 - 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']
 - name: 'gcr.io/google.com/cloudsdktool/cloud-sdk'
   args: ['gcloud', 'run', 'deploy', 'my-app', '--
image=gcr.io/$PROJECT_ID/my-app', '--region=us-central1']
```

📌 **Example:**
A **DevOps engineer** sets up a cloudbuild.yaml to **automate the build and deployment** of a Node.js application on **Cloud Run**.

CHAPTER 4: AUTOMATING DEPLOYMENTS WITH CLOUD BUILD TRIGGERS

4.1 Creating a Cloud Build Trigger

1. Open **Google Cloud Console** → Navigate to **Cloud Build**.
2. Click **Triggers** → Click **+ Create Trigger**.

3. Select **Source Repository** (GitHub, GitLab, or Cloud Source Repositories).
4. Define **Trigger Conditions** (e.g., **Run on every push to main branch**).
5. Specify the **cloudbuild.yaml file** → Click **Create**.

📌 **Example:**
A **mobile app development team** sets up a **trigger** to deploy the app **to Firebase Hosting** whenever new code is pushed to the main branch.

Chapter 5: Deploying Applications Using Cloud Build

5.1 Deploying to Google Kubernetes Engine (GKE)

steps:

```
 - name: 'gcr.io/cloud-builders/docker'
   args: ['build', '-t', 'gcr.io/$PROJECT_ID/gke-app', '.']
 - name: 'gcr.io/cloud-builders/docker'
   args: ['push', 'gcr.io/$PROJECT_ID/gke-app']
 - name: 'gcr.io/google.com/cloudsdktool/cloud-sdk'
   args: ['gcloud', 'container', 'clusters', 'get-credentials', 'my-cluster', '--zone=us-central1-a']
 - name: 'gcr.io/google.com/cloudsdktool/cloud-sdk'
   args: ['kubectl', 'apply', '-f', 'deployment.yaml']
```

📌 **Example:**
A **fintech startup** deploys its **microservices to GKE** using Cloud Build and **Kubernetes manifests**.

5.2 Deploying to App Engine

steps:

```
- name: 'gcr.io/google.com/cloudsdktool/cloud-sdk'
  args: ['gcloud', 'app', 'deploy']
```

📌 **Example:**
A **blogging platform** uses **Cloud Build to deploy updates** to **Google App Engine** for seamless content delivery.

5.3 Deploying to Compute Engine

steps:

```
- name: 'gcr.io/google.com/cloudsdktool/cloud-sdk'
  args: ['gcloud', 'compute', 'instances', 'update-container', 'my-instance', '--container-image=gcr.io/$PROJECT_ID/my-app']
```

📌 **Example:**
A **game development company** automates deployment of game servers to **Compute Engine VMs**.

CHAPTER 6: SECURITY & ACCESS CONTROL IN CLOUD BUILD

✓ **IAM Roles for Cloud Build:**

- **Cloud Build Editor (roles/cloudbuild.builds.editor)** – Manage build configurations.
- **Artifact Registry Reader (roles/artifactregistry.reader)** – Access stored artifacts.

✓ **Private Builds with VPC Service Controls:**

- Restricts **Cloud Build access to private networks** for security.
- Prevents **data exfiltration from Cloud Build pipelines**.

📌 **Example:**
A **financial institution** uses **VPC Service Controls** to **restrict Cloud Build access** to internal systems only.

CHAPTER 7: MONITORING & TROUBLESHOOTING CLOUD BUILD

7.1 Viewing Cloud Build Logs

1. Open **Cloud Console** → Navigate to **Cloud Build** → Click **Build History**.
2. Click on a **failed build** to view **error logs**.
3. Run the following command in **Cloud Shell**:

```
gcloud builds log --stream
```

📌 **Example:**
A **DevOps engineer** investigates a **failed build** due to a **missing Dockerfile**.

CHAPTER 8: CASE STUDY – AUTOMATING CI/CD FOR A WEB APP

Problem Statement:

A **tech company** needs to automate the deployment of its **React web application** to **Cloud Run** using Cloud Build.

Solution Implementation:

1. **Created a cloudbuild.yaml** file to build and deploy the application.

2. **Configured Cloud Build Trigger** to run on code changes.
3. **Stored Docker images in Artifact Registry** for version control.
4. **Integrated Security Controls** to restrict unauthorized builds.

Results:

✓ **Deployment time reduced from 30 minutes to under 5 minutes**.
✓ **Improved security** by limiting build access with IAM roles.
✓ **Enabled rollback capabilities** for quick issue resolution.

Chapter 9: Exercise & Review Questions

Exercise:

1. **Set up a Cloud Build Trigger** for a GitHub repository.
2. **Write a cloudbuild.yaml** to deploy an app to Cloud Run.
3. **Deploy a Kubernetes application** using Cloud Build.
4. **Enable logging and debug a failed Cloud Build pipeline**.

Review Questions:

1. What are the **main advantages of Cloud Build** over traditional CI/CD tools?
2. How does Cloud Build **integrate with Google Kubernetes Engine (GKE)**?
3. What IAM roles are required for **managing Cloud Build**?
4. How can **Cloud Build security be enhanced using VPC Service Controls**?

Conclusion: Automating Deployments with Cloud Build

Cloud Build simplifies **CI/CD workflows**, allowing organizations to **automate builds, tests, and deployments** efficiently while maintaining **security, scalability, and cost-effectiveness**. 🚀

CI/CD ON GOOGLE CLOUD - STUDY MATERIAL

CHAPTER 1: INTRODUCTION TO CI/CD ON GOOGLE CLOUD

1.1 What is CI/CD?

CI/CD (Continuous Integration and Continuous Deployment/Delivery) is a **DevOps practice** that automates **code integration, testing, and deployment** to streamline the software development lifecycle.

✓ **Continuous Integration (CI)** – Automates code integration, testing, and build processes.
✓ **Continuous Deployment (CD)** – Automatically deploys code changes to production.
✓ **Continuous Delivery (CD)** – Ensures code is always ready for deployment but requires manual approval.

1.2 Benefits of CI/CD on Google Cloud

✓ **Faster Releases** – Automate build, test, and deployment cycles.
✓ **Higher Code Quality** – Continuous testing reduces bugs.
✓ **Scalability & Reliability** – Google Cloud scales deployments efficiently.
✓ **Seamless Integration** – Works with Kubernetes, App Engine, Cloud Functions, and VMs.

📌 **Example:**
A **fintech startup** implements CI/CD on **Google Cloud Build** to **automate code testing and deployment** for a banking API.

CHAPTER 2: CI/CD TOOLS IN GOOGLE CLOUD

2.1 Key Google Cloud CI/CD Services

Tool	Purpose
Cloud Build	CI/CD service for building, testing, and deploying code.
Artifact Registry	Securely stores and manages Docker images & artifacts.
Cloud Deploy	Automates delivery to GKE (Google Kubernetes Engine).
Cloud Source Repositories	Git repository for storing source code.
Secret Manager	Stores and manages sensitive configuration data.
Cloud Monitoring & Logging	Tracks performance and logs CI/CD pipelines.

📌 **Example:**
A **SaaS company** uses **Cloud Build and Cloud Deploy** to push changes **automatically to Kubernetes** after code merges.

CHAPTER 3: SETTING UP CI/CD WITH CLOUD BUILD

3.1 Prerequisites

✓ **Google Cloud Project** with billing enabled.
✓ Install **Google Cloud SDK** and authenticate with gcloud init.
✓ Enable APIs:

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

3.2 Configure Cloud Source Repositories (Optional)

1. Open **Google Cloud Console → Cloud Source Repositories**.
2. Click **Create Repository** → Name it (e.g., my-app-repo).
3. Clone the repository locally:
4. gcloud source repos clone my-app-repo

CHAPTER 4: CREATING A CLOUD BUILD PIPELINE

4.1 Define a Cloud Build Configuration (cloudbuild.yaml)

Example **Cloud Build YAML** for a **Node.js Application**:

```
steps:
- name: 'gcr.io/cloud-builders/npm'
  args: ['install']

- name: 'gcr.io/cloud-builders/npm'
  args: ['test']

- name: 'gcr.io/cloud-builders/npm'
  args: ['run', 'build']

- name: 'gcr.io/cloud-builders/docker'
  args: ['build', '-t', 'gcr.io/$PROJECT_ID/my-app:$BUILD_ID', '.']
```

```
- name: 'gcr.io/cloud-builders/docker'
  args: ['push', 'gcr.io/$PROJECT_ID/my-app:$BUILD_ID']

images:
- 'gcr.io/$PROJECT_ID/my-app:$BUILD_ID'
```

4.2 Triggering Cloud Build Automatically

1. Open **Cloud Build** → **Triggers**.
2. Click **Create Trigger**.
3. Select **Source Repository** → Choose my-app-repo.
4. Set **Trigger Type**:
    - **Push to a branch** (e.g., main).
    - **Pull request event**.
5. Use **cloudbuild.yaml** as the configuration file.
6. Click **Create**.

📌 **Example:**
A **web development agency** sets up **Cloud Build triggers** to **automatically build and test** code every time a developer pushes to Git.

CHAPTER 5: DEPLOYING APPLICATIONS USING CI/CD

5.1 Deploying to Google Kubernetes Engine (GKE)

Step 1: Create a Kubernetes Cluster

```
gcloud container clusters create my-cluster --num-nodes=3
```

Step 2: Deploy the Application to GKE

1. Create a **Kubernetes Deployment file (deployment.yaml)**:
2. apiVersion: apps/v1
3. kind: Deployment
4. metadata:
5. name: my-app
6. spec:
7. replicas: 3
8. selector:
9. matchLabels:
10. app: my-app
11. template:
12. metadata:
13. labels:
14. app: my-app
15. spec:
16. containers:
17. - name: my-app
18. image: gcr.io/my-project-id/my-app:latest
19. ports:
20. - containerPort: 8080
21. Deploy the application:

22. kubectl apply -f deployment.yaml

📌 **Example:**
A **media company** runs a **video streaming platform** using **CI/CD pipelines to deploy microservices** to GKE.

Chapter 6: Security Best Practices for CI/CD on Google Cloud

✓ **Use IAM Roles & Service Accounts** – Restrict permissions to pipelines.
✓ **Enable Artifact Registry** – Store container images securely.
✓ **Use Google Secret Manager** – Manage sensitive environment variables.
✓ **Enable Cloud Logging & Monitoring** – Track failures in CI/CD pipelines.
✓ **Implement Cloud Armor** – Protect deployed applications from DDoS attacks.

📌 **Example:**
A **fintech startup** integrates **Secret Manager** into CI/CD to **secure API keys and database credentials**.

Chapter 7: Monitoring & Troubleshooting CI/CD Pipelines

7.1 Monitoring Cloud Build Logs

1. Open **Google Cloud Console** → **Cloud Build**.
2. Click on a build to view logs.
3. Filter logs for errors and performance issues.

7.2 Debugging CI/CD Failures

✓ Check **Cloud Build Logs** for failed steps.

✓ Ensure **Docker image permissions** allow deployment.

✓ Validate **Kubernetes YAML manifests** for errors.

📌 **Example:**

A **gaming company** monitors **Cloud Build logs** to troubleshoot **failing deployments** in GKE.

Chapter 8: Exercise & Review Questions

Exercise:

1. **Create a Cloud Build pipeline** that builds a simple web app.
2. **Deploy an application to Google Kubernetes Engine using CI/CD**.
3. **Secure deployment pipelines using IAM and Secret Manager**.

Review Questions:

1. What is the difference between **CI and CD**?
2. How does **Cloud Build automate CI/CD workflows**?
3. What are the benefits of **Google Artifact Registry**?
4. How do **Cloud Build Triggers** automate deployments?
5. What are **best practices for securing CI/CD pipelines**?

Conclusion: Building Scalable CI/CD Pipelines on Google Cloud

✓ **Google Cloud CI/CD automates software development workflows** for faster deployments.

✓ **Cloud Build, Artifact Registry, and Cloud Deploy** streamline **continuous integration & deployment**.
✓ **Security, monitoring, and automation best practices** ensure **reliable cloud applications**.

🚀 **Mastering CI/CD on Google Cloud helps developers build, test, and deploy at scale!**

INTRODUCTION TO ANTHOS FOR HYBRID CLOUD DEPLOYMENT

CHAPTER 1: UNDERSTANDING ANTHOS AND HYBRID CLOUD

1.1 What is Anthos?

Anthos is **Google Cloud's hybrid and multi-cloud platform** that enables organizations to **deploy, manage, and modernize applications across on-premises, Google Cloud, and other cloud providers** (AWS, Azure).

1.2 Why Use Anthos for Hybrid Cloud?

✓ **Consistent Application Management** – Deploy and manage workloads across **on-prem, GCP, AWS, and Azure**.
✓ **Kubernetes-Based Platform** – Uses **Google Kubernetes Engine (GKE) and Anthos Service Mesh** for orchestration.
✓ **Security & Compliance** – Centralized **IAM, Policy Enforcement, and Zero-Trust Security**.
✓ **Multi-Cloud Interoperability** – Run workloads **without vendor lock-in**.
✓ **Cost Optimization** – Optimize resources **across private and public clouds**.

📌 **Example:**
A **financial institution** uses Anthos to **run Kubernetes clusters across their on-prem data center and GCP** while maintaining security compliance.

CHAPTER 2: ANTHOS COMPONENTS & ARCHITECTURE

2.1 Core Components of Anthos

Component	Description
GKE on Google Cloud	Managed Kubernetes service on GCP.
GKE on-prem	Kubernetes clusters running in on-premises data centers.
Anthos Config Management	Ensures consistent configurations across environments.
Anthos Service Mesh	Secure communication between microservices.
Anthos Policy Controller	Enforces security and compliance policies.
Cloud Run for Anthos	Serverless workloads on Kubernetes.
Anthos Clusters on AWS/Azure	Manage Kubernetes clusters in multi-cloud environments.

📌 **Example:**
A **retail company** deploys **GKE on AWS** and manages workloads using **Anthos Config Management** for consistency.

2.2 Anthos Deployment Models

✓ **On-Premises Kubernetes Deployment** – Run Anthos on **VMware, bare metal, or OpenShift**.
✓ **Hybrid Cloud Deployment** – Extend GKE **to on-prem and multi-cloud providers**.
✓ **Multi-Cloud Deployment** – Manage Kubernetes clusters across **GCP, AWS, and Azure**.
✓ **Serverless on Anthos** – Deploy serverless applications using **Cloud Run for Anthos**.

📌 **Example:**
A **manufacturing company** runs **SAP workloads on-prem** and **customer portals on GCP** using Anthos.

Chapter 3: Setting Up Anthos for Hybrid Cloud

3.1 Prerequisites

✓ **Google Cloud Project** – Sign up at Google Cloud Console.
✓ **Enable Anthos API** – In **Cloud Console,** go to **APIs & Services** → **Enable Anthos API**.
✓ **Install Google Cloud SDK (CLI)**:

```
curl https://sdk.cloud.google.com | bash

gcloud init
```

✓ **On-Prem Kubernetes Cluster** – Ensure you have a **VMware vSphere or bare metal environment** for on-prem deployment.

3.2 Deploying Anthos on GCP

Step 1: Enable Anthos API

```
gcloud services enable anthos.googleapis.com
```

Step 2: Create a GKE Cluster

```
gcloud container clusters create anthos-cluster \
  --zone us-central1-a \
  --num-nodes=3
```

Step 3: Install Anthos Config Management

```
gcloud anthos config sync repo \
```

```
--project=my-gcp-project \
--sync-git-url=git@github.com:my-org/my-repo.git \
--sync-branch=main
```

📌 **Example:**
A **telecom provider** deploys **Anthos on GKE** to manage hybrid cloud networking.

3.3 Deploying Anthos on On-Premises Data Centers

1. Install **Google Cloud's Anthos CLI**:
2. gcloud components install anthoscli
3. Connect Anthos to your **on-prem Kubernetes clusters** (VMware, bare metal, or OpenShift).
4. Configure **IAM Roles & Policies** for **hybrid access control**.
5. Set up **Anthos Service Mesh** for secure microservices communication.

📌 **Example:**
A **government agency** deploys Anthos **on VMware vSphere** to ensure **secure data sovereignty**.

3.4 Deploying Anthos on AWS or Azure

1. **Enable Anthos Multi-Cloud API**
2. gcloud services enable anthos.googleapis.com multicloud.googleapis.com
3. **Register an AWS or Azure Kubernetes Cluster**

4. gcloud anthos multi-cloud aws clusters register my-aws-cluster \
5. --location us-east-1 \
6. --project my-gcp-project
7. **Apply Anthos Config Management** for uniform governance.

📌 **Example:**
A **global SaaS company** runs **Kubernetes clusters on AWS and Azure** while managing everything from **Google Cloud Anthos**.

Chapter 4: Anthos Config Management & Policy Enforcement

4.1 Enforcing Configurations Across Hybrid Cloud

✓ Define Kubernetes Configurations in Git
✓ Apply Policies Automatically to All Clusters
✓ Monitor & Enforce Compliance Rules

Deploy Config Management in a GKE Cluster

```
apiVersion: configmanagement.gke.io/v1
kind: ConfigManagement
metadata:
 name: config-management
spec:
 clusterName: anthos-cluster
 git:
  syncRepo: git@github.com:my-org/configs.git
  syncBranch: main
```

policyDir: policies/

📌 **Example:**
A **banking institution** uses **Anthos Config Management** to enforce **PCI-DSS security policies** across hybrid cloud clusters.

Chapter 5: Anthos Service Mesh for Microservices Security

5.1 What is Anthos Service Mesh?

Anthos Service Mesh (ASM) provides **secure communication between microservices** across hybrid cloud environments.

✓ **Zero-Trust Security** – Enforces mutual TLS encryption between services.
✓ **Traffic Management** – Handles service discovery, load balancing, and routing.
✓ **Observability** – Provides monitoring and logging using **Stackdriver and Prometheus**.

5.2 Deploying Anthos Service Mesh

1. **Enable Anthos Service Mesh**
2. gcloud services enable mesh.googleapis.com
3. **Install Istio on GKE Cluster**
4. istioctl install --set profile=demo
5. **Enable Traffic Management Between Services**
6. apiVersion: networking.istio.io/v1alpha3
7. kind: VirtualService
8. metadata:

```
9.  name: frontend-service
10.         spec:
11.  hosts:
12.          - frontend.myapp.com
13.  http:
14.          - route:
15.    - destination:
16.                host: backend-service
17.       port:
18.                 number: 8080
```

📌 **Example:**
A **ride-sharing app** uses Anthos Service Mesh to **secure inter-service communication between API, driver, and payment services**.

CHAPTER 6: CASE STUDY – HYBRID CLOUD DEPLOYMENT FOR A RETAIL CHAIN

Problem Statement:

A **global retail company** wants to **deploy e-commerce applications across multiple cloud providers and on-prem data centers**.

Solution Implementation:

✓ **Deployed Anthos on GKE (GCP) & VMware (On-Premises).**
✓ **Configured Anthos Config Management** for consistent governance.
✓ **Enabled Anthos Service Mesh** to manage microservices across

locations.

✓ **Integrated with Google Cloud Operations** for monitoring.

Results:

✓ **Reduced infrastructure complexity by 50%.**

✓ **Ensured PCI-DSS compliance across hybrid cloud environments.**

✓ **Achieved zero-downtime deployments with Anthos Service Mesh.**

CHAPTER 7: EXERCISE & REVIEW QUESTIONS

Exercise:

1. **Deploy Anthos on a GKE cluster.**
2. **Register an on-prem Kubernetes cluster with Anthos.**
3. **Use Anthos Config Management to enforce security policies.**
4. **Set up Anthos Service Mesh for secure service-to-service communication.**

Review Questions:

1. What are the key benefits of using **Anthos for hybrid cloud**?
2. How does **Anthos Config Management** enforce compliance?
3. What are the differences between **Anthos Service Mesh and Istio**?
4. How does **Anthos integrate with AWS and Azure**?

CONCLUSION

✓ Anthos simplifies hybrid and multi-cloud Kubernetes management.

✓ It enforces security, governance, and observability across cloud environments.

✓ Anthos enables businesses to deploy applications consistently across on-prem and cloud platforms.

🚀 **Next Steps:**

✓ Set up **Anthos on GKE**.

✓ Configure **Anthos Config Management**.

✓ Deploy **Anthos Service Mesh for microservices security**! 🚀

ASSIGNMENT

DEPLOY A FULL-STACK WEB APPLICATION USING APP ENGINE

SOLUTION: DEPLOY A FULL-STACK WEB APPLICATION USING GOOGLE APP ENGINE

Step 1: Understanding Google App Engine

What is Google App Engine?

Google App Engine (GAE) is a **fully managed Platform-as-a-Service (PaaS)** that allows developers to **build, deploy, and scale web applications** automatically. It supports **multiple programming languages** and **auto-scales based on demand**.

Why Use App Engine?

✓ **No infrastructure management** – Focus on writing code while Google handles scaling.
✓ **Automatic scaling** – Scales based on incoming traffic.
✓ **Built-in security** – Supports HTTPS, IAM roles, and firewall rules.
✓ **Supports multiple databases** – Works with **Cloud SQL, Firestore, and BigQuery**.
✓ **Custom & Standard Environments** – Choose between **flexible runtime (custom Docker images) or standard runtime (predefined environments)**.

📌 **Example:**
A **food delivery startup** deploys a **full-stack web app** on App Engine to **manage orders and real-time tracking**, ensuring **high availability and auto-scaling**.

Step 2: Set Up the Full-Stack Application Structure

2.1 Choose the Technology Stack

For this guide, we will deploy a **full-stack web app** using:

- **Frontend:** React (hosted on App Engine or Firebase Hosting).
- **Backend:** Python Flask (API on App Engine).
- **Database:** Google Cloud SQL (PostgreSQL).

2.2 Application Folder Structure

```
fullstack-app/
│── backend/        # Backend API (Python Flask)
│   ├── main.py     # Flask application
│   ├── requirements.txt # Dependencies
│   ├── app.yaml    # App Engine configuration
│── frontend/       # Frontend React App
│   ├── public/     # Static assets
│   ├── src/        # React components
│   ├── package.json  # Node.js dependencies
│   ├── firebase.json # Firebase Hosting configuration
│── deploy.sh       # Deployment script
```

Step 3: Deploy Backend on App Engine

3.1 Set Up Google Cloud Project

1. **Enable Cloud Services:**
2. gcloud services enable appengine.googleapis.com \
3. cloudsql.googleapis.com \

4. sqladmin.googleapis.com
5. **Initialize App Engine:**
6. gcloud app create --region=us-central1

3.2 Develop a Simple Flask Backend

Create backend/main.py:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/")
def home():
    return jsonify({"message": "Hello from Google App Engine!"})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8080)
```

3.3 Define Dependencies in requirements.txt

```
flask
gunicorn
```

3.4 Create app.yaml for App Engine Deployment

runtime: python39

entrypoint: gunicorn -b :$PORT main:app

env_variables:

CLOUD_SQL_CONNECTION_NAME: "your-project-id:us-central1:your-database"

DB_USER: "your-db-user"

DB_PASSWORD: "your-db-password"

DB_NAME: "your-database-name"

handlers:

- url: /.*

script: auto

📌 **Explanation:**

- **runtime: python39** → Uses Python 3.9 runtime.
- **entrypoint** → Runs the Flask app with gunicorn.
- **env_variables** → Stores database credentials.
- **handlers** → Routes all requests to Flask.

3.5 Deploy the Backend to App Engine

1. **Authenticate with Google Cloud:**
2. gcloud auth login
3. **Deploy the Flask application:**

4. gcloud app deploy backend/app.yaml
5. **Get the deployed URL:**
6. gcloud app browse

Step 4: Set Up a Cloud SQL Database

4.1 Create a Cloud SQL Instance (PostgreSQL)

1. **Create a Cloud SQL instance:**
2. gcloud sql instances create fullstack-db \
3. --database-version=POSTGRES_14 \
4. --tier=db-f1-micro \
5. --region=us-central1
6. **Set database credentials:**
7. gcloud sql users set-password postgres \
8. --instance=fullstack-db \
9. --password=yourpassword
10. **Get Cloud SQL connection name:**
11. gcloud sql instances describe fullstack-db | grep connectionName

📌 **Example:** your-project-id:us-central1:fullstack-db

4. **Update app.yaml** with the Cloud SQL connection.

Step 5: Deploy the Frontend

5.1 Set Up a React Frontend

1. **Initialize a React app:**
2. npx create-react-app frontend
3. cd frontend
4. **Install Axios for API Calls:**
5. npm install axios
6. **Modify src/App.js to Fetch Data from Backend:**

```
import React, { useEffect, useState } from "react";
import axios from "axios";

function App() {
 const [message, setMessage] = useState("");

 useEffect(() => {
  axios.get("https://your-app-url.appspot.com/")
   .then(response => setMessage(response.data.message))
   .catch(error => console.error("Error fetching data:", error));
 }, []);

 return (
  <div>
   <h1>{message}</h1>
  </div>
```

```
  );
}

export default App;
```

5.2 Deploy Frontend to Firebase Hosting

1. **Install Firebase CLI:**
2. npm install -g firebase-tools
3. **Initialize Firebase Hosting:**
4. firebase init hosting
5. **Deploy to Firebase:**
6. firebase deploy

📌 **Example:**
Your React app is now hosted on **Firebase Hosting**, fetching data from the **App Engine backend**.

Step 6: Automate Deployment with a Script

Create deploy.sh to automate deployment:

```
#!/bin/bash

echo "Deploying Backend to App Engine..."

cd backend

gcloud app deploy app.yaml --quiet
```

```
echo "Deploying Frontend to Firebase..."
cd ../frontend
npm run build
firebase deploy
```

Run:

```
chmod +x deploy.sh
./deploy.sh
```

Step 7: Secure & Optimize Deployment

7.1 Secure App Engine & Cloud SQL

✓ Restrict public access:

```
gcloud app services update default --no-allow-unauthenticated
```

✓ Use IAM Roles to Restrict Database Access:

```
gcloud projects add-iam-policy-binding YOUR_PROJECT_ID \
  --member=user:developer@example.com \
  --role=roles/cloudsql.client
```

7.2 Enable Logging & Monitoring

✓ Enable Google Cloud Logging:

```
gcloud logging write my-log "Backend deployed successfully"
```

✓ Monitor App Engine logs:

```
gcloud app logs tail -s default
```

Case Study: A Full-Stack E-Commerce Web App

Problem Statement:

A startup wants to launch an **e-commerce website** to manage **products, orders, and users** with:

✓ **A scalable backend**

✓ **Database for storing orders**

✓ **A frontend for customer interactions**

Solution Implementation:

1. **Deployed Flask backend to App Engine**.
2. **Connected Cloud SQL PostgreSQL for order storage**.
3. **Deployed React frontend to Firebase Hosting**.
4. **Used IAM to restrict database access**.

Results:

✓ **Achieved 99.99% uptime**.

✓ **Handled 100,000+ daily users with auto-scaling**.

✓ **Reduced operational costs by 40% compared to Compute Engine**.

Conclusion: Deploying Full-Stack Apps with App Engine

By **leveraging App Engine, Cloud SQL, and Firebase Hosting,** developers can **build and deploy scalable full-stack applications** without managing infrastructure. 🚀

BUILD A SERVERLESS FUNCTION USING CLOUD FUNCTIONS

SOLUTION: BUILD A SERVERLESS FUNCTION USING CLOUD FUNCTIONS

Overview

Google Cloud Functions is a **fully managed serverless execution environment** for running event-driven applications. It automatically scales, requires no infrastructure management, and integrates with other **Google Cloud services** like **Cloud Storage, Pub/Sub, Firestore, and BigQuery**.

Use Case Example:

A company wants to **automatically process and resize images** uploaded to **Cloud Storage** using **Cloud Functions**.

Step 1: Set Up Google Cloud Environment

1.1 Enable Cloud Functions API

1. Open **Google Cloud Console → Navigation Menu → Cloud Functions**.
2. Click **Enable API and Create Function**.

1.2 Install Google Cloud SDK (CLI method)

Run the following command in **Cloud Shell or local terminal**:

gcloud services enable cloudfunctions.googleapis.com

Step 2: Create a Cloud Function

2.1 Define Cloud Function Parameters

- **Function Name:** image-resize-function

- **Trigger Type:** Cloud Storage (Object Finalized)
- **Memory Allocation:** 256MB
- **Execution Timeout:** 60 seconds
- **Runtime:** Python 3.9

2.2 Creating a Function (Using Cloud Console)

1. Open **Cloud Functions Console** → Click **+ Create Function**.
2. Choose **Trigger** → Select **Cloud Storage** and configure:
    - **Event Type:** Finalized (new or updated files).
    - **Bucket Name:** image-upload-bucket.
3. Set **Memory & Timeout** → Click **Next**.
4. Select **Runtime** (Python 3.9) → Add **main.py** and **requirements.txt**.
5. Click **Deploy**.

Step 3: Write Cloud Function Code

3.1 Install Required Dependencies

Create a **requirements.txt** file with the following packages:

google-cloud-storage

Pillow

3.2 Write the Function Code (main.py)

```
import os

from google.cloud import storage

from PIL import Image
```

```
def resize_image(event, context):
  """Triggered when a new image is uploaded to Cloud Storage.
    Resizes the image and saves it to a different bucket.
  """
  bucket_name = event['bucket']
  file_name = event['name']

  # Initialize Storage Client
  storage_client = storage.Client()

  # Define input and output buckets
  input_bucket = storage_client.bucket(bucket_name)
  output_bucket = storage_client.bucket('resized-images-bucket')

  blob = input_bucket.blob(file_name)
  temp_download_path = f"/tmp/{file_name}"

  # Download the image
  blob.download_to_filename(temp_download_path)

  # Open and resize the image
```

```
    with Image.open(temp_download_path) as img:
        img = img.resize((200, 200))
        temp_resized_path = f"/tmp/resized-{file_name}"
        img.save(temp_resized_path, format="JPEG")

    # Upload resized image to output bucket
    output_blob = output_bucket.blob(f"resized-{file_name}")
    output_blob.upload_from_filename(temp_resized_path)

    print(f"Resized image uploaded to {output_bucket.name}/resized-{file_name}")
```

Step 4: Deploy the Function Using Google Cloud SDK

4.1 Deploy the Function (Command Line)

```
gcloud functions deploy image-resize-function \
  --runtime python39 \
  --trigger-resource image-upload-bucket \
  --trigger-event google.storage.object.finalize \
  --entry-point resize_image
```

4.2 Verify Deployment

Run:

```
gcloud functions describe image-resize-function
```

Check the **status** field to confirm successful deployment.

Step 5: Test the Cloud Function

5.1 Upload a Test Image

Upload an image to the **image-upload-bucket**:

gsutil cp test-image.jpg gs://image-upload-bucket/

5.2 Verify Output

Check the resized image in **Cloud Storage**:

gsutil ls gs://resized-images-bucket/

You should see resized-test-image.jpg in the resized images bucket.

Step 6: Monitor & Debug Cloud Function

6.1 View Logs

gcloud functions logs read image-resize-function

6.2 Debugging Errors

Common issues:

✓ **Permission Errors** → Ensure **Cloud Functions has access to Cloud Storage**.

✓ **Cold Starts** → Minimize execution time by using **minimal dependencies**.

✓ **Memory Limits** → Increase memory allocation if handling large images.

Step 7: Secure Cloud Function

7.1 Restrict IAM Access

```
gcloud projects add-iam-policy-binding my-project \
 --member=serviceAccount:my-function-service-account \
 --role=roles/storage.admin
```

7.2 Enable API Gateway for Authentication

Use **Cloud Endpoints** to enforce **OAuth authentication** before function execution.

Case Study: Automating Document Processing with Cloud Functions

Problem Statement:

A **legal firm** wants to automate the **processing of scanned legal documents**.

Solution Implementation:

1. **Created a Cloud Function** to extract text from uploaded PDFs using **Google Vision API**.
2. **Triggered function on Cloud Storage uploads** (document-upload-bucket).
3. **Stored extracted text in Firestore** for fast retrieval.
4. **Enabled IAM security controls** to limit unauthorized access.

Results:

✓ **Processing time reduced by 80%**.
✓ **Automated document indexing for legal cases**.
✓ **Secured document processing pipeline** with IAM roles.

Review Questions & Exercises

Exercise:

1. **Deploy a Cloud Function** that logs metadata of uploaded images.
2. **Modify the function to delete large files (>5MB) from Cloud Storage.**
3. **Enable error logging** and test failure scenarios.

Review Questions:

1. How does Cloud Functions handle **event-driven execution**?
2. What are the differences between **Cloud Functions & App Engine**?
3. How do you secure Cloud Functions with **IAM & API Gateway**?
4. Why is **cold start a challenge** in serverless computing?

Conclusion: Scaling Applications with Cloud Functions

Cloud Functions **simplify event-driven workflows** and help developers **build scalable, secure, and cost-efficient applications**. By integrating with **Cloud Storage, Firestore, Pub/Sub, and BigQuery**, businesses can **automate complex workflows** with minimal operational overhead. 🚀