### **Guide d'Implémentation et Runbook de la Plateforme SkillForge AI**

* **Version :** 1.0
* **Date :** 24 juin 2025
* **Propriétaire :** Ingénieur DevOps

### **Partie 1 : Initialisation de l'Environnement Global (Jour 0)**

#### **1.1. Introduction**

Ce document est un guide pratique destiné à l'ingénieur DevOps ou au fondateur technique pour mettre en place l'infrastructure fondamentale du projet SkillForge AI. Les opérations décrites dans cette section sont généralement effectuées **une seule fois** au tout début du projet. Elles créent le socle sur lequel tout le reste sera construit.

**Prérequis :**

* Disposer d'un compte Google Cloud avec une organisation et un compte de facturation actifs.
* Avoir les droits nécessaires pour créer des projets et gérer les permissions IAM.
* Avoir la [CLI gcloud](https://www.google.com/search?q=%5Bhttps://cloud.google.com/sdk/docs/install%5D(https://cloud.google.com/sdk/docs/install)) installée et authentifiée.

#### **1.2. Création et Configuration du Projet Google Cloud**

creation du compte cloud identity et de l’organisation (car j’utilise un compte gmail personnel): mon utilisateur compte admin de cloud identity est : [sah@emacsah.com](mailto:sah@emacsah.com), le nom de l’organisation est SkillForge AI. tout ceci est a configurer a partir de : inscription a cloud identity free

Nous allons créer un projet GCP dédié qui hébergera toutes nos ressources.

1. **Définir les variables d'environnement :** Ouvrez un terminal et définissez les variables suivantes pour simplifier les commandes.

# Remplacez par l'ID de votre organisation GCP (ex: 123456789012)

export ORGANIZATION\_ID="<ID\_ORGANISATION\_GCP>"

# Remplacez par l'ID de votre compte de facturation (ex: 01A2B3-C4D5E6-F7G8H9)

export BILLING\_ACCOUNT\_ID="<ID\_COMPTE\_FACTURATION>"

# Choisissez un ID unique pour le projet.

export PROJECT\_ID="skillforge-ai-mvp"

**2.** Créer le projet GCP :

gcloud projects create ${PROJECT\_ID} --organization=${ORGANIZATION\_ID}

**3.** Lier le compte de facturation :

gcloud billing projects link ${PROJECT\_ID} --billing-account=${BILLING\_ACCOUNT\_ID}

**4. Définir le projet comme défaut pour la CLI gcloud :**

**gcloud config set project ${PROJECT\_ID}**

**5. Activer les APIs nécessaires :** C'est une étape cruciale pour autoriser l'utilisation des services dont nous aurons besoin.

**gcloud services enable \**

**iam.googleapis.com \**

**cloudresourcemanager.googleapis.com \**

**secretmanager.googleapis.com \**

**artifactregistry.googleapis.com \**

**run.googleapis.com \**

**sqladmin.googleapis.com \**

**redis.googleapis.com \**

**container.googleapis.com \**

**cloudbuild.googleapis.com \**

[**iamcredentials.googleapis.com**](http://iamcredentials.googleapis.com)

#### **1.3. Création et Configuration du Dépôt GitHub**

1. **Création du Dépôt :**
   * **Rendez-vous sur GitHub et créez un nouveau dépôt.**
   * **Nom du dépôt : skillforge-ai-monorepo**
   * **Visibilité : Private**
   * **Initialiser avec : un fichier README.md et un template .gitignore pour Python.**
2. **Création des Branches Initiales :**
   * **Clonez le dépôt fraîchement créé sur votre machine.**
   * **Créez et poussez la branche develop :**

**git checkout -b develop**

**git push -u origin develop**

**# Revenez sur la branche main**

**git checkout main**

1. **Configuration des Règles de Protection de Branche :**
   * Dans les paramètres de votre dépôt GitHub (Settings > Branches), ajoutez des règles de protection pour main et develop.
   * **Pour la branche develop :**
     + Cochez "Require a pull request before merging".
     + Cochez "Require status checks to pass before merging". Sélectionnez ici les jobs de tests et de linting de vos futurs pipelines CI/CD.
     + Cochez "Require conversation resolution before merging".
   * **Pour la branche main :**
     + Appliquez les mêmes règles que pour develop.
     + Cochez en plus "Restrict who can push to matching branches" et ne laissez que les administrateurs du dépôt.

#### **1.4. Configuration de l'Authentification CI/CD (GitHub Actions ↔ GCP)**

Nous allons configurer une connexion sécurisée et sans mot de passe pour permettre à GitHub Actions de déployer sur GCP, en utilisant Workload Identity Federation.

1. Créer un Compte de Service pour la CI/CD sur GCP :

**gcloud iam service-accounts create sa-github-actions-cicd \**

**--display-name="Service Account for GitHub Actions CI/CD"**

**2. Donner les Permissions au Compte de Service :** Pour l'initialisation du projet, nous lui donnons un rôle large. Ces permissions seront affinées plus tard.

gcloud projects add-iam-policy-binding ${PROJECT\_ID} \

--member="serviceAccount:sa-github-actions-cicd@${PROJECT\_ID}.iam.gserviceaccount.com" \

--role="roles/owner"

3. Créer un "Workload Identity Pool" et un Fournisseur :

# Créer le Pool

gcloud iam workload-identity-pools create skillforge-pool \

--location="global" \

--display-name="SkillForge Identity Pool"

# Obtenir l'ID complet du Pool

export POOL\_ID=$(gcloud iam workload-identity-pools describe skillforge-pool --location="global" --format="value(name)")

# Créer le Fournisseur OIDC pour GitHub

gcloud iam workload-identity-pools providers create-oidc github-provider \

--location="global" \

--workload-identity-pool="skillforge-pool" \

--issuer-uri="https://token.actions.githubusercontent.com" \

--attribute-mapping="google.subject=assertion.sub,attribute.actor=assertion.actor,attribute.repository=assertion.repository"

4. **Lier le Dépôt GitHub au Compte de Service GCP :** C'est l'étape finale qui autorise les actions de votre dépôt à usurper l'identité du compte de service GCP.

gcloud iam service-accounts add-iam-policy-binding "sa-github-actions-cicd@${PROJECT\_ID}.iam.gserviceaccount.com" \

--role="roles/iam.workloadIdentityUser" \

--member="principalSet://[iam.googleapis.com/${POOL\_ID}/attribute.repository/VOTRE\_ORGANISATION\_GITHUB/skillforge-ai-monorepo](http://iam.googleapis.com/$%7BPOOL_ID%7D/attribute.repository/VOTRE_ORGANISATION_GITHUB/skillforge-ai-monorepo)"

(Remplacez VOTRE\_ORGANISATION\_GITHUB par le nom de votre organisation ou utilisateur GitHub).

5. **Obtenir l'ID du Fournisseur pour les Workflows :**

* Exécutez cette commande et conservez la sortie. Vous en aurez besoin pour vos fichiers de workflow GitHub Actions.

gcloud iam workload-identity-pools providers describe github-provider \

--location="global" \

--workload-identity-pool="skillforge-pool" \

--format="value(name)"

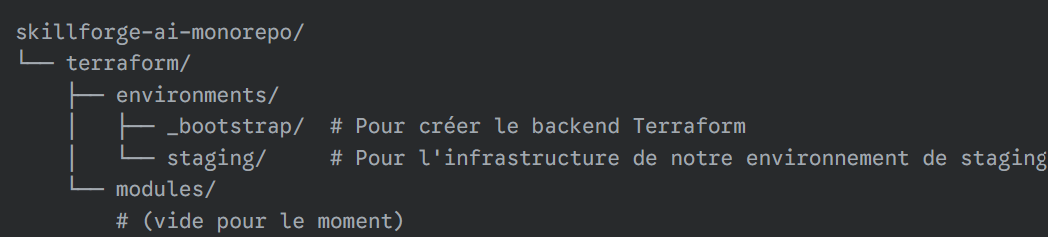
### **Partie 2 : Construction de l'Infrastructure de Base avec Terraform**

#### **2.1. Introduction**

Maintenant que notre projet GCP est initialisé et que notre dépôt Git est prêt, nous allons utiliser Terraform pour créer nos premières ressources cloud. Nous allons nous concentrer sur les composants fondamentaux et partagés : le réseau qui isolera nos services, et les espaces de stockage qui contiendront nos artefacts.

Nous allons suivre une structure de dossiers claire pour notre code Terraform au sein de notre monorepo :

skillforge-ai-monorepo/ └── terraform/ ├── environments/ │ ├── \_bootstrap/ # Pour créer le backend Terraform │ └── staging/ # Pour l'infrastructure de notre environnement de staging └── modules/ # (vide pour le moment)



#### **2.2. Initialisation du Backend Terraform (Le Paradoxe de la Poule et de l'Œuf)**

Nous voulons stocker notre fichier d'état Terraform sur un bucket GCS, mais ce bucket doit d'abord être créé. Nous résolvons ce paradoxe avec une procédure d'amorçage ("bootstrap") en deux temps.

1. **Créer le bucket GCS avec un état local :**
   * Créez le dossier terraform/environments/\_bootstrap/.
   * À l'intérieur, créez un fichier bootstrap.tf avec le contenu suivant, en remplaçant <ID\_DE\_VOTRE\_PROJET\_GCP> par l'ID de votre projet (ex: skillforge-ai-mvp).

# terraform/environments/\_bootstrap/bootstrap.tf

# Configure le fournisseur Google Cloud.

provider "google" {

project = "<ID\_DE\_VOTRE\_PROJET\_GCP>"

region = "europe-west1"

}

# Crée un bucket GCS unique pour stocker l'état de Terraform.

# Le nom doit être globalement unique.

resource "google\_storage\_bucket" "tfstate" {

name = "<ID\_DE\_VOTRE\_PROJET\_GCP>-tfstate"

location = "EUROPE-WEST1"

force\_destroy = false # Sécurité pour ne pas le supprimer accidentellement.

# Active le versioning pour garder un historique des états.

versioning {

enabled = true

}

}

Exécutez les commandes suivantes depuis le dossier terraform/environments/\_bootstrap/ :

# Initialise Terraform avec un backend local

terraform init

# Applique la configuration pour créer le bucket

terraform apply

* + Confirmez par yes. Un bucket GCS a été créé, et un fichier terraform.tfstate a été généré localement dans le dossier \_bootstrap. Ce dossier a rempli sa mission, nous n'y toucherons plus.

#### **2.3. Code Terraform pour l'Environnement de Staging**

Nous allons maintenant travailler dans le dossier terraform/environments/staging/. Tous les fichiers suivants sont à créer dans ce répertoire.

1. **Configurer le Backend distant (backend.tf) :**
   * Créez un fichier backend.tf. Il indique à Terraform d'utiliser le bucket que nous venons de créer pour stocker son état.

# terraform/environments/staging/backend.tf

terraform {

backend "gcs" {

bucket = "<ID\_DE\_VOTRE\_PROJET\_GCP>-tfstate"

prefix = "staging" # Un sous-dossier pour l'état de l'environnement staging

}

}

**2. Configurer le Fournisseur (provider.tf) :**

* Créez un fichier provider.tf pour définir les paramètres de connexion à GCP.

# terraform/environments/staging/provider.tf

provider "google" {

project = "<ID\_DE\_VOTRE\_PROJET\_GCP>"

region = "europe-west1"

}

3. **Créer le Réseau VPC (network.tf) :**

* Créez un fichier network.tf pour définir notre réseau privé.

# terraform/environments/staging/network.tf

# Crée le réseau VPC principal

resource "google\_compute\_network" "vpc\_main" {

name = "skillforge-vpc-staging"

auto\_create\_subnetworks = false # Nous gérons les sous-réseaux manuellement

}

# Crée un sous-réseau dans la région europe-west1

resource "google\_compute\_subnetwork" "subnet\_main" {

name = "skillforge-subnet-staging"

ip\_cidr\_range = "10.0.0.0/24"

region = "europe-west1"

network = google\_compute\_network.vpc\_main.id

}

# Règle de pare-feu pour autoriser tout le trafic interne au VPC

resource "google\_compute\_firewall" "allow\_internal" {

name = "allow-internal-traffic"

network = google\_compute\_network.vpc\_main.name

allow {

protocol = "tcp"

ports = ["0-65535"]

}

allow {

protocol = "udp"

ports = ["0-65535"]

}

allow {

protocol = "icmp"

}

source\_ranges = ["10.0.0.0/24"] # Autorise le trafic depuis notre propre sous-réseau

}

4. **Créer les Registres et Buckets de Stockage (storage.tf) :**

* Créez un fichier storage.tf.

# terraform/environments/staging/storage.tf

# Crée le dépôt Artifact Registry pour nos images Docker

resource "google\_artifact\_registry\_repository" "docker\_repo" {

location = "europe-west1"

repository\_id = "skillforge-docker-repo-staging"

format = "DOCKER"

description = "Docker repository for SkillForge staging environment"

}

# Crée le bucket pour les uploads des utilisateurs

resource "google\_storage\_bucket" "user\_uploads" {

name = "<ID\_DE\_VOTRE\_PROJET\_GCP>-user-uploads-staging"

location = "EUROPE-WEST1"

uniform\_bucket\_level\_access = true

}

# Crée le bucket pour les fichiers statiques du Front-End

resource "google\_storage\_bucket" "frontend\_assets" {

name = "<ID\_DE\_VOTRE\_PROJET\_GCP>-frontend-assets-staging"

location = "EUROPE-WEST1"

uniform\_bucket\_level\_access = true

# Configure le bucket pour l'hébergement de site web statique

website {

main\_page\_suffix = "index.html"

not\_found\_page = "index.html" # Pour les SPAs qui gèrent le routage

}

}

#### **2.4. Appliquer la Configuration de Base**

Maintenant que nos fichiers de configuration pour l'environnement staging sont prêts :

1. **Initialiser Terraform :**
   * Naviguez dans le dossier terraform/environments/staging/ et exécutez init.
   * Cette fois, Terraform va détecter le fichier backend.tf et vous demander si vous voulez copier l'état existant. Comme il n'y en a pas, il va simplement configurer le backend sur GCS.

cd terraform/environments/staging

terraform init

2. **Planifier et Appliquer :**

* Générez un plan d'exécution pour voir ce que Terraform va créer.

terraform plan

Si le plan vous semble correct, appliquez-le pour créer les ressources.

terraform apply

Confirmez par yes.

### **Partie 3 : Déploiement du "Socle de Services" avec Terraform**

#### **3.1. Introduction**

Cette section décrit comment créer nos services de données managés à l'aide de Terraform. Nous allons ajouter de nouveaux fichiers de configuration dans notre répertoire terraform/environments/staging/.

La gestion des mots de passe est un point de sécurité critique. Nous utiliserons Google Secret Manager pour stocker le mot de passe de la base de données de manière sécurisée, et Terraform y fera référence sans jamais l'exposer en clair dans notre code.

#### **3.2. Code Terraform pour Cloud SQL (PostgreSQL)**

1. **Étape Préalable : Créer le Secret Manuellement**
   * Avant d'exécuter Terraform, créez le secret pour le mot de passe de la base de données.
   * Rendez-vous dans la console Google Cloud > "Secret Manager".
   * Créez un nouveau secret nommé postgres-password-staging.
   * Ajoutez une première version à ce secret avec un mot de passe fort et complexe que vous aurez généré.
2. **Créer le fichier database.tf :**
   * Créez un nouveau fichier terraform/environments/staging/database.tf.
   * Ajoutez le contenu suivant. Il définit l'instance de base de données, la connecte à notre réseau privé et configure un utilisateur en lisant le mot de passe depuis Secret Manager.

# terraform/environments/staging/database.tf

# Référence au secret que nous avons créé manuellement.

data "google\_secret\_manager\_secret\_version" "postgres\_password" {

secret = "postgres-password-staging"

}

# Crée l'instance principale de PostgreSQL

resource "google\_sql\_database\_instance" "main\_postgres" {

name = "skillforge-pg-instance-staging"

database\_version = "POSTGRES\_16"

region = "europe-west1"

settings {

tier = "db-n1-standard-1" # Taille de machine adaptée pour du staging

ip\_configuration {

ipv4\_enabled = false # Désactive l'IP publique pour des raisons de sécurité

private\_network = google\_compute\_network.vpc\_main.id

}

backup\_configuration {

enabled = true

}

}

}

# Crée la base de données "skillforge\_db" à l'intérieur de l'instance

resource "google\_sql\_database" "main\_db" {

name = "skillforge\_db"

instance = google\_sql\_database\_instance.main\_postgres.name

}

# Crée l'utilisateur applicatif pour la base de données

resource "google\_sql\_user" "app\_user" {

name = "skillforge\_user"

instance = google\_sql\_database\_instance.main\_postgres.name

# Le mot de passe est récupéré de manière sécurisée depuis Secret Manager.

password = data.google\_secret\_manager\_secret\_version.postgres\_password.secret\_data

}

#### **3.3. Code Terraform pour Memorystore (Redis)**

Redis sera utilisé comme broker de messages pour nos agents IA. Nous le déployons également sur notre réseau privé.

1. **Activer l'API "Service Networking"**
   * Cette API est requise pour que Memorystore puisse se connecter à notre VPC.

gcloud services enable servicenetworking.googleapis.com

2. Créer le fichier cache.tf :

Créez un nouveau fichier terraform/environments/staging/[cache.tf](http://cache.tf).

Ajoutez le contenu suivant :

# terraform/environments/staging/cache.tf

# Réserve une plage d'adresses IP pour les services managés Google

resource "google\_compute\_global\_address" "private\_service\_access" {

name = "private-service-access-staging"

purpose = "VPC\_PEERING"

address\_type = "INTERNAL"

ip\_version = "IPV4"

prefix\_length = 16

network = google\_compute\_network.vpc\_main.id

}

# Crée la connexion de peering entre notre VPC et les services Google

resource "google\_service\_networking\_connection" "main\_peering" {

network = google\_compute\_network.vpc\_main.id

service = "servicenetworking.googleapis.com"

reserved\_peering\_ranges = [google\_compute\_global\_address.private\_service\_access.name]

}

# Crée l'instance Redis

resource "google\_redis\_instance" "main\_redis" {

name = "skillforge-redis-instance-staging"

tier = "BASIC" # Tier de base pour le staging

memory\_size\_gb = 1

location\_id = "europe-west1-b"

connect\_mode = "PRIVATE\_SERVICE\_ACCESS"

authorized\_network = google\_compute\_network.vpc\_main.id

# S'assure que la connexion de peering est établie avant de créer l'instance Redis

depends\_on = [google\_service\_networking\_connection.main\_peering]

}

#### **3.4. Appliquer la Configuration du Socle de Services**

Maintenant que nos nouveaux fichiers de configuration sont prêts :

1. **Initialiser Terraform (si nécessaire) :**
   * Si vous avez ajouté de nouveaux providers, exécutez terraform init depuis le dossier terraform/environments/staging/.
2. **Planifier et Appliquer :**
   * Générez un plan d'exécution pour voir ce que Terraform va créer.

terraform plan

Si le plan vous semble correct, appliquez-le pour créer les ressources.

terraform apply

* Confirmez par yes.

**Attention :** Le provisionnement d'une instance Cloud SQL peut prendre plusieurs minutes (5 à 10 minutes). Soyez patient.

Notre infrastructure de base est maintenant provisionnée et attend de recevoir nos applications. Dans cette partie, nous allons construire les "usines logicielles" - nos pipelines CI/CD - qui prendront le code source écrit par les développeurs et le transformeront en applications déployées.

Pour éviter de répéter la même logique de déploiement pour chaque microservice, nous allons adopter une approche modulaire et réutilisable.

### **Partie 4 : Création des Pipelines CI/CD Réutilisables**

#### **4.1. Introduction et Principe des "Reusable Workflows"**

Plutôt que de créer un long fichier de pipeline pour chacun de nos 10+ services (ce qui serait un cauchemar à maintenir), nous allons créer des **briques de base** sous forme de "Reusable Workflows" (flux de travail réutilisables) dans GitHub Actions.

* **Le Principe :** Nous définissons une seule fois une tâche complexe (comme "construire une image Docker") dans un workflow générique. Ensuite, les pipelines de chaque service peuvent simplement "appeler" ce workflow en lui passant quelques paramètres (comme le chemin vers son Dockerfile).
* **Les Avantages :**
  + **DRY (Don't Repeat Yourself) :** La logique est définie, testée et maintenue en un seul endroit.
  + **Consistance :** Tous nos services sont construits, scannés et déployés exactement de la même manière.
  + **Simplicité :** Les pipelines spécifiques à chaque service deviennent extrêmement simples et lisibles.

Ces workflows réutilisables doivent être créés dans le répertoire .github/workflows/ de notre monorepo.

#### **4.2. Workflow Réutilisable : build-push-docker.yml**

Ce workflow est notre "usine à conteneurs". Sa mission : prendre un Dockerfile, construire une image, la scanner pour les vulnérabilités, et la pousser sur notre registre privé.

1. **Créez le fichier .github/workflows/build-push-docker.yml :**
2. **Ajoutez le contenu suivant :**

# .github/workflows/build-push-docker.yml

name: "Reusable - Build, Scan, and Push Docker Image"

on:

# Définit ce workflow comme étant appelable par d'autres workflows.

workflow\_call:

# Définit les paramètres (entrées) que le workflow appelant doit fournir.

inputs:

image\_name:

description: 'The name of the docker image without the tag'

required: true

type: string

dockerfile\_path:

description: 'The path to the Dockerfile'

required: true

type: string

gcp\_workload\_identity\_provider:

description: 'The WIF provider from GCP'

required: true

type: string

gcp\_service\_account:

description: 'The GCP service account to impersonate'

required: true

type: string

# Définit les données que ce workflow peut retourner à l'appelant.

outputs:

image\_uri:

description: "The full URI of the pushed image"

value: ${{ jobs.build-scan-push.outputs.image\_uri }}

jobs:

build-scan-push:

name: Build, Scan & Push

runs-on: ubuntu-latest

outputs:

image\_uri: ${{ steps.push.outputs.image\_uri }}

steps:

- name: Checkout repository

uses: actions/checkout@v4

- name: Authenticate to Google Cloud

id: auth

uses: google-github-actions/auth@v2

with:

workload\_identity\_provider: ${{ inputs.gcp\_workload\_identity\_provider }}

service\_account: ${{ inputs.gcp\_service\_account }}

- name: Configure Docker to use gcloud

run: gcloud auth configure-docker europe-west1-docker.pkg.dev --quiet

- name: Build Docker Image

id: build

run: |

IMAGE\_URI="europe-west1-docker.pkg.dev/${{ secrets.GCP\_PROJECT\_ID }}/${{ inputs.image\_name }}:${{ github.sha }}"

docker build -t ${IMAGE\_URI} -f ${{ inputs.dockerfile\_path }} .

echo "image\_uri=${IMAGE\_URI}" >> $GITHUB\_OUTPUT

- name: Scan Docker Image for Vulnerabilities

uses: aquasecurity/trivy-action@0.20.0

with:

image-ref: ${{ steps.build.outputs.image\_uri }}

format: 'table'

# Fait échouer le pipeline si des vulnérabilités de sévérité HAUTE ou CRITIQUE sont trouvées.

exit-code: '1'

ignore-unfixed: true

vuln-type: 'os,library'

severity: 'HIGH,CRITICAL'

- name: Push Docker Image to Artifact Registry

id: push

run: |

docker push ${{ steps.build.outputs.image\_uri }}

echo "image\_uri=${{ steps.build.outputs.image\_uri }}" >> $GITHUB\_OUTPUT

#### **4.3. Workflow Réutilisable : deploy-to-cloud-run.yml**

Ce workflow est notre "déployeur". Sa mission : prendre une image Docker existante et la déployer comme nouvelle révision d'un service Cloud Run.

1. **Créez le fichier .github/workflows/deploy-to-cloud-run.yml :**
2. **Ajoutez le contenu suivant :**

# .github/workflows/deploy-to-cloud-run.yml

name: "Reusable - Deploy to Cloud Run"

on:

workflow\_call:

inputs:

image\_uri:

description: 'The full URI of the docker image to deploy'

required: true

type: string

service\_name:

description: 'The name of the Cloud Run service to deploy to'

required: true

type: string

gcp\_region:

description: 'The GCP region where the service is located'

required: true

type: string

gcp\_workload\_identity\_provider:

description: 'The WIF provider from GCP'

required: true

type: string

gcp\_service\_account:

description: 'The GCP service account to impersonate'

required: true

type: string

jobs:

deploy:

name: Deploy to Cloud Run

runs-on: ubuntu-latest

steps:

- name: Authenticate to Google Cloud

id: auth

uses: google-github-actions/auth@v2

with:

workload\_identity\_provider: ${{ inputs.gcp\_workload\_identity\_provider }}

service\_account: ${{ inputs.gcp\_service\_account }}

- name: Deploy new revision to Cloud Run

id: deploy

uses: google-github-actions/deploy-cloudrun@v2

with:

service: ${{ inputs.service\_name }}

image: ${{ inputs.image\_uri }}

region: ${{ inputs.gcp\_region }}

# Déploie 100% du trafic sur la nouvelle révision immédiatement

no-traffic: false

Avec ces deux briques de construction puissantes, nous avons standardisé nos opérations les plus critiques. Les pipelines pour chaque service seront maintenant incroyablement simples : ils n'auront qu'à appeler ces workflows avec les bons paramètres.

C'est ici que tous les éléments que nous avons mis en place (infrastructure de base, pipelines réutilisables) s'assemblent pour accomplir notre objectif final : déployer une application.

Cette section est un guide complet, de bout en bout, pour prendre un de nos microservices, le user-service, et le déployer sur notre environnement de Staging. Ce processus servira de modèle pour tous les autres services.

### **Partie 5 : Déploiement End-to-End d'un Premier Service (Exemple : user-service)**

#### **5.1. Introduction**

Nous allons maintenant créer et configurer tous les artefacts nécessaires au déploiement d'un service spécifique :

1. **L'infrastructure du service** avec Terraform (son service Cloud Run, son compte de service dédié).
2. **Son conteneur de production** avec un Dockerfile optimisé.
3. **Son pipeline de déploiement** avec un fichier de workflow GitHub Actions qui utilise nos briques réutilisables.

#### **5.2. Code Terraform pour le user-service**

Nous allons créer la définition de notre service dans un nouveau sous-dossier pour garder notre code organisé.

1. **Créez le fichier terraform/environments/staging/services/user\_service.tf :**
2. **Ajoutez le contenu suivant :**

# terraform/environments/staging/services/user\_service.tf

# 1. Créer un compte de service dédié pour ce microservice

resource "google\_service\_account" "user\_service\_sa" {

account\_id = "sa-user-service-staging"

display\_name = "Service Account for User Service (Staging)"

}

# 2. Lui donner les permissions nécessaires

# a) Accès à la base de données

resource "google\_project\_iam\_member" "user\_service\_db\_access" {

project = var.project\_id # Assurez-vous d'avoir défini une variable project\_id

role = "roles/cloudsql.client"

member = "serviceAccount:${google\_service\_account.user\_service\_sa.email}"

}

# b) Accès aux secrets dont il a besoin

resource "google\_secret\_manager\_secret\_iam\_member" "user\_service\_db\_password\_access" {

secret\_id = "postgres-password-staging"

role = "roles/secretmanager.secretAccessor"

member = "serviceAccount:${google\_service\_account.user\_service\_sa.email}"

}

# Répétez pour d'autres secrets comme la clé JWT...

# 3. Créer le service Cloud Run

resource "google\_cloud\_run\_v2\_service" "user\_service" {

name = "user-service-staging"

location = "europe-west1"

template {

# Exécute le conteneur avec son compte de service dédié

service\_account = google\_service\_account.user\_service\_sa.email

containers {

# L'image sera mise à jour par le pipeline CI/CD

image = "europe-west1-docker.pkg.dev/${var.project\_id}/skillforge-docker-repo-staging/user-service:latest"

# Monte les secrets en tant que variables d'environnement

env {

name = "JWT\_SECRET\_KEY"

value\_source {

secret\_key\_ref {

secret = "jwt-secret-key-staging"

version = "latest"

}

}

}

}

# Connecte le service à notre réseau privé pour qu'il puisse parler à la BDD

vpc\_access {

connector = google\_vpc\_access\_connector.main\_connector.id # Assurez-vous de créer ce connecteur

egress = "ALL\_TRAFFIC"

}

}

}

#### **5.3. Fichier Dockerfile pour le user-service**

Ce Dockerfile utilise une construction multi-étapes pour créer une image de production légère et sécurisée.

1. **Créez le fichier apps/backend/user-service/Dockerfile :**
2. **Ajoutez le contenu suivant :**

# apps/backend/user-service/Dockerfile

# --- Étape 1: Le "Builder" ---

# Installe les dépendances dans une première étape pour optimiser le cache de layers

FROM python:3.11-slim as builder

WORKDIR /app

RUN pip install pip-tools

COPY requirements.txt .

# Installe uniquement les dépendances de production

RUN pip-sync requirements.txt

# --- Étape 2: Le "Runtime" ---

# L'image finale est basée sur la même image slim

FROM python:3.11-slim

WORKDIR /app

# Crée un utilisateur non-root pour des raisons de sécurité

RUN useradd --no-create-home appuser

USER appuser

# Copie les dépendances installées depuis l'étape "builder"

COPY --from=builder /usr/local/lib/python3.11/site-packages /usr/local/lib/python3.11/site-packages

COPY --from=builder /usr/local/bin /usr/local/bin

# Copie le code source de l'application

COPY ./app ./app

# Commande pour lancer le serveur de production Uvicorn sur le port 8080 (standard pour Cloud Run)

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]

#### **5.4. Fichier de Pipeline CI/CD pour le user-service**

Ce pipeline est simple car il ne fait qu'appeler nos workflows réutilisables.

1. **Créez le fichier .github/workflows/deploy-user-service.yml :**
2. **Ajoutez le contenu suivant :**

# .github/workflows/deploy-user-service.yml

name: "Deploy - User Service"

on:

push:

branches:

- 'develop' # Déclenche sur push vers la branche de staging

paths:

- 'apps/backend/user-service/\*\*' # Ne se déclenche que si le code de ce service change

jobs:

deploy-staging:

name: Deploy to Staging

runs-on: ubuntu-latest

# Permissions nécessaires pour l'authentification OIDC avec GCP

permissions:

contents: read

id-token: write

steps:

- name: Checkout repository

uses: actions/checkout@v4

- name: Build, Scan & Push Docker Image

id: build

# Appelle notre workflow réutilisable

uses: ./.github/workflows/build-push-docker.yml

with:

image\_name: 'skillforge-docker-repo-staging/user-service'

dockerfile\_path: 'apps/backend/user-service/Dockerfile'

# Ces secrets doivent être configurés dans les "Actions secrets" du dépôt GitHub

gcp\_workload\_identity\_provider: ${{ secrets.GCP\_WIF\_PROVIDER }}

gcp\_service\_account: ${{ secrets.GCP\_CICD\_SERVICE\_ACCOUNT }}

- name: Deploy to Cloud Run

# Appelle notre second workflow réutilisable

uses: ./.github/workflows/deploy-to-cloud-run.yml

# S'assure que l'étape de build est terminée

needs: build

with:

# Utilise l'URI de l'image retournée par l'étape de build

image\_uri: ${{ needs.build.outputs.image\_uri }}

service\_name: 'user-service-staging'

gcp\_region: 'europe-west1'

gcp\_workload\_identity\_provider: ${{ secrets.GCP\_WIF\_PROVIDER }}

gcp\_service\_account: ${{ secrets.GCP\_CICD\_SERVICE\_ACCOUNT }}

#### **5.5. Exécution et Vérification**

1. **Appliquer l'infrastructure :** Exécutez terraform apply depuis terraform/environments/staging/ pour créer les ressources Cloud Run et IAM.
2. **Déclencher le pipeline :** Commitez et poussez ces trois nouveaux fichiers sur votre branche develop.
3. **Observer :** Allez dans l'onglet "Actions" de votre dépôt GitHub. Vous verrez le pipeline "Deploy - User Service" s'exécuter.
4. **Vérifier :** Une fois le pipeline terminé avec succès, rendez-vous sur la console Google Cloud, dans la section Cloud Run. Vous devriez voir votre service user-service-staging avec une nouvelle révision active, utilisant la dernière image que vous venez de construire.

Ce processus, une fois en place, peut être dupliqué en quelques minutes pour chaque nouveau microservice, garantissant un déploiement standardisé, sécurisé et entièrement automatisé pour l'ensemble de notre plateforme.

Partie 5 (Version Étendue) : Déploiement End-to-End des Services

#### **A. Code Terraform pour le project-service**

Le fichier est très similaire, seules les permissions IAM changent.

* **Fichier :** terraform/environments/staging/services/project\_service.tf
* **Contenu :**

# terraform/environments/staging/services/project\_service.tf

resource "google\_service\_account" "project\_service\_sa" {

account\_id = "sa-project-service-staging"

display\_name = "Service Account for Project Service (Staging)"

}

# --- PERMISSIONS SPÉCIFIQUES ---

# Ce service a besoin d'accéder à plus de tables.

resource "google\_project\_iam\_member" "project\_service\_db\_access" {

project = var.project\_id

role = "roles/cloudsql.client"

member = "serviceAccount:${google\_service\_account.project\_service\_sa.email}"

}

# Note: Dans une configuration idéale, les GRANTs au niveau SQL seraient gérés par Terraform

# pour donner des droits SELECT/INSERT/UPDATE uniquement sur les tables :

# projects, milestones, skills, project\_skills, project\_assignments, deliverables.

# ... autres accès aux secrets ...

resource "google\_cloud\_run\_v2\_service" "project\_service" {

name = "project-service-staging" # <- Nom du service différent

location = "europe-west1"

# Le reste de la configuration est identique à celle du user-service

# (template, service\_account, vpc\_access, etc.)

# ...

}

#### **B. Dockerfile pour le project-service**

Le Dockerfile est **identique** à celui du user-service, il est simplement situé dans son propre répertoire : apps/backend/project-service/Dockerfile.

#### **. Pipeline CI/CD pour le project-service**

Le fichier de workflow est un quasi-clone, seules les variables changent.

* **Fichier :** .github/workflows/deploy-project-service.yml
* **Contenu (différences mises en évidence) :**

# .github/workflows/deploy-project-service.yml

name: "Deploy - Project Service" # <- Titre différent

on:

push:

branches: ['develop']

paths:

- 'apps/backend/project-service/\*\*' # <- Chemin de déclenchement différent

jobs:

deploy-staging:

# ... permissions, etc. ...

steps:

- name: Build, Scan & Push Docker Image

id: build

uses: ./.github/workflows/build-push-docker.yml

with:

image\_name: 'skillforge-docker-repo-staging/project-service' # <- Nom de l'image différent

dockerfile\_path: 'apps/backend/project-service/Dockerfile' # <- Chemin du Dockerfile différent

# ... reste des `with` identique ...

- name: Deploy to Cloud Run

uses: ./.github/workflows/deploy-to-cloud-run.yml

needs: build

with:

image\_uri: ${{ needs.build.outputs.image\_uri }}

service\_name: 'project-service-staging' # <- Nom du service différent

# ... reste des `with` identique …

### **5.3. Déploiement de l'Application Front-End**

Le déploiement du Front-End est différent : il s'agit de déployer des fichiers statiques sur GCS, et non un service sur Cloud Run.

#### **A. Code Terraform pour le Front-End**

Le code Terraform ne crée pas de service, mais s'assure que le pipeline a les droits pour écrire dans le bucket de stockage. Le bucket lui-même (frontend-assets) a déjà été créé dans la Partie 2.

* **Fichier :** terraform/environments/staging/services/frontend.tf
* **Contenu :**

# terraform/environments/staging/services/frontend.tf

# Donne au compte de service de la CI/CD les droits d'écrire dans le bucket du front-end

resource "google\_storage\_bucket\_iam\_member" "cicd\_frontend\_assets\_writer" {

bucket = google\_storage\_bucket.frontend\_assets.name

role = "roles/storage.objectAdmin"

member = "serviceAccount:sa-github-actions-cicd@${var.project\_id}.iam.gserviceaccount.com"

}

#### **B. Dockerfile pour le Front-End**

Ce Dockerfile sert uniquement pour l'environnement de développement local (docker-compose). Le pipeline CI/CD, lui, n'utilise que la partie build.

* **Fichier :** apps/frontend/Dockerfile
* **Contenu :**

# apps/frontend/Dockerfile

# --- Étape 1: Build ---

# Construit les fichiers statiques de production

FROM node:20-alpine as builder

WORKDIR /app

COPY package\*.json ./

RUN npm ci

COPY . .

# La commande de build de Vite génère les fichiers dans /app/dist

RUN npm run build

# --- Étape 2: Serveur de Production ---

# Utilise un serveur Nginx léger pour servir les fichiers statiques

FROM nginx:1.25-alpine

# Copie les fichiers buildés de l'étape précédente vers le répertoire par défaut de Nginx

COPY --from=builder /app/dist /usr/share/nginx/html

# Copie une configuration Nginx si nécessaire (pour gérer le routage SPA)

COPY nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

#### **C. Pipeline CI/CD pour le Front-End**

Ce pipeline n'appelle pas nos workflows réutilisables car les étapes sont différentes.

* **Fichier :** .github/workflows/deploy-frontend.yml
* **Contenu :**

# .github/workflows/deploy-frontend.yml

name: "Deploy - Frontend Application"

on:

push:

branches: ['develop']

paths:

- 'apps/frontend/\*\*'

jobs:

build-and-deploy-staging:

name: Build and Deploy to Staging GCS

runs-on: ubuntu-latest

permissions:

contents: read

id-token: write

steps:

- name: Checkout repository

uses: actions/checkout@v4

- name: Setup Node.js

uses: actions/setup-node@v4

with:

node-version-file: 'apps/frontend/.nvmrc'

cache: 'npm'

cache-dependency-path: 'apps/frontend/package-lock.json'

- name: Install Dependencies

run: npm ci --prefix apps/frontend

- name: Build Static Files

run: npm run build --prefix apps/frontend

- name: Authenticate to Google Cloud

uses: google-github-actions/auth@v2

with:

workload\_identity\_provider: ${{ secrets.GCP\_WIF\_PROVIDER }}

service\_account: ${{ secrets.GCP\_CICD\_SERVICE\_ACCOUNT }}

- name: Deploy to GCS

uses: google-github-actions/upload-cloud-storage@v2

with:

path: 'apps/frontend/dist'

destination: '${{ secrets.GCP\_PROJECT\_ID }}-frontend-assets-staging'

parent: false

### **5.4. Déploiement des Agents IA**

Les agents IA sont similaires aux services Back-End, mais leur déclenchement est événementiel (Pub/Sub) et non HTTP.

#### **A. Code Terraform pour l'evaluation-agent**

La différence majeure est la configuration du trigger Pub/Sub.

* **Fichier :** terraform/environments/staging/services/evaluation\_agent.tf
* **Contenu :**

# terraform/environments/staging/services/evaluation\_agent.tf

# ... création du service account 'sa-evaluation-agent-staging' ...

# L'agent n'est pas accessible publiquement

resource "google\_cloud\_run\_v2\_service" "evaluation\_agent" {

name = "evaluation-agent-staging"

location = "europe-west1"

ingress = "INGRESS\_TRAFFIC\_INTERNAL\_ONLY" # <- Différence Clé

# ... template, etc. ...

}

# Crée le topic Pub/Sub

resource "google\_pubsub\_topic" "deliverable\_submitted" {

name = "project.deliverable.submitted"

}

# Crée une souscription Pub/Sub qui PUSH vers l'agent Cloud Run

resource "google\_pubsub\_subscription" "evaluation\_agent\_subscription" {

name = "evaluation-agent-sub-staging"

topic = google\_pubsub\_topic.deliverable\_submitted.name

push\_config {

push\_endpoint = google\_cloud\_run\_v2\_service.evaluation\_agent.uri

oidc\_token {

# Utilise le compte de service de l'agent pour s'authentifier

service\_account\_email = google\_service\_account.evaluation\_agent\_sa.email

}

}

}

#### **B. Dockerfile et Pipeline CI/CD pour l'evaluation-agent**

Le Dockerfile et le pipeline CI/CD de l'agent sont **structurellement identiques** à ceux d'un microservice Back-End. Il suffit de changer les chemins, les noms d'images et les noms de services dans les fichiers respectifs.

### **Points de Complétude et d'Excellence Opérationnelle**

#### **1. Terraform : Dépendances Manquantes et Variables**

Dans ma hâte de fournir les configurations des services, j'ai omis de déclarer deux ressources dont ils dépendent, et je n'ai pas formalisé la gestion des variables.

* **1.1. Le Connecteur d'Accès VPC (Critique) :** Pour qu'un service Cloud Run puisse communiquer avec une base de données sur un réseau privé, il a besoin d'un "Serverless VPC Access Connector". Le code suivant doit être ajouté, par exemple dans votre fichier network.tf.

# terraform/environments/staging/network.tf (à ajouter)

# Crée le connecteur qui fait le pont entre Cloud Run et notre VPC.

resource "google\_vpc\_access\_connector" "main\_connector" {

name = "skillforge-vpc-connector-staging"

region = "europe-west1"

ip\_cidr\_range = "10.8.0.0/28" # Doit être une plage non utilisée dans votre VPC.

}

La ligne connector = google\_vpc\_access\_connector.main\_connector.id dans la configuration des services Cloud Run fonctionnera maintenant correctement.

**1.2. Formalisation des Variables Terraform :** J'ai utilisé var.project\_id sans le déclarer. Pour une configuration propre, chaque environnement doit avoir un fichier variables.tf (pour déclarer les variables) et un fichier terraform.tfvars (pour leur assigner une valeur), qui lui n'est pas versionné.

* **Fichier terraform/environments/staging/variables.tf :**

variable "project\_id" {

type = string

description = "The GCP project ID to deploy to."

}

variable "region" {

type = string

description = "The primary GCP region for resources."

default = "europe-west1"

}

Fichier terraform/environments/staging/terraform.tfvars :

project\_id = "skillforge-ai-mvp"

#### **2. Dockerfile : La Vérification de Santé (Healthcheck)**

Un conteneur peut être en cours d'exécution mais l'application à l'intérieur peut être bloquée ou ne pas répondre. Le HEALTHCHECK permet à Docker et à Cloud Run de savoir si l'application est réellement en bonne santé.

* **Ajout au Dockerfile de chaque service Back-End et Agent :** Ajoutez ces lignes vers la fin du Dockerfile, juste avant le CMD.

# ... à la fin des Dockerfiles backend/agents

# Ajoute un endpoint /healthz dans ton application FastAPI qui retourne un statut 200 OK.

HEALTHCHECK --interval=30s --timeout=5s --start-period=15s --retries=3 \

CMD curl -f http://localhost:8080/healthz || exit 1

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]

* Cela indique à Cloud Run de vérifier toutes les 30 secondes si l'application répond correctement. Si ce n'est pas le cas, la révision sera considérée comme défaillante et Cloud Run ne lui enverra pas de trafic.

#### **3. CI/CD : L'Étape Manquante la plus Critique**

J'ai défini le workflow de déploiement mais j'ai omis d'y inclure **explicitement** l'étape la plus importante avant le déploiement d'un service Back-End : l'application des migrations de base de données. Sans cette étape, un déploiement qui dépend d'un nouveau champ en base de données échouera systématiquement.

* **Correction du Pipeline deploy-\*.yml pour les services Back-End et Agents :** Le pipeline doit être modifié pour inclure une étape alembic upgrade head.

# .github/workflows/deploy-user-service.yml (corrigé)

# ...

jobs:

deploy-staging:

# ...

steps:

# ... (Checkout, Auth, Build, Scan, Push) ...

- name: Build, Scan & Push Docker Image

id: build

# ...

# --- ETAPE CRITIQUE AJOUTEE ---

- name: Apply Database Migrations

run: |

# Installe les dépendances nécessaires pour Alembic

pip install alembic psycopg2-binary

# Exécute la migration sur la BDD de staging

# La DATABASE\_URL est injectée ici en tant que secret

alembic -x database\_url=${{ secrets.STAGING\_DATABASE\_URL }} upgrade head

- name: Deploy to Cloud Run

uses: ./.github/workflows/deploy-to-cloud-run.yml

needs: build # Dépend toujours du build

with:

# …

#### **4. Configuration des Secrets dans GitHub**

J'ai utilisé des variables comme ${{ secrets.GCP\_WIF\_PROVIDER }}. Pour une clarté totale, voici la liste des secrets qui doivent être configurés dans l'interface de votre dépôt GitHub (Settings > Secrets and variables > Actions) pour que les pipelines fonctionnent.

* GCP\_PROJECT\_ID : L'ID de votre projet GCP (ex: skillforge-ai-mvp).
* GCP\_WIF\_PROVIDER : L'ID complet du fournisseur Workload Identity que nous avons récupéré à la Partie 1 (ex: projects/12345/.../providers/github-provider).
* GCP\_CICD\_SERVICE\_ACCOUNT : L'email du compte de service pour la CI/CD (ex: sa-github-actions-cicd@...).
* STAGING\_DATABASE\_URL : L'URL de connexion complète à votre base de données de staging, incluant le mot de passe. C'est le seul endroit où cette URL complète sera stockée.

### **Partie 6 : Configuration de l'Observabilité**

#### **6.1. Introduction**

L'objectif de cette section est de mettre en place les outils qui nous permettront de surveiller la santé de notre plateforme, de diagnostiquer les problèmes et d'être alertés en cas d'incident. Nous allons, comme pour le reste de notre infrastructure, définir notre configuration d'observabilité en tant que code.

Tous les fichiers suivants sont à créer dans le répertoire terraform/environments/staging/.

#### **6.2. Code Terraform pour le Dashboard de Monitoring**

Nous allons créer un tableau de bord centralisé dans Google Cloud Monitoring qui affichera les métriques vitales de nos services.

1. **Créer le fichier monitoring.tf :**
2. **Ajoutez le contenu suivant.** Il définit la structure du tableau de bord en JSON, puis crée la ressource dans GCP.

# terraform/environments/staging/monitoring.tf

# Utilisation d'une variable locale pour stocker la définition JSON du dashboard,

# ce qui rend le code plus lisible.

locals {

# Définition du Dashboard au format JSON.

# Ce JSON peut être généré initialement via l'interface graphique de GCP puis exporté.

main\_dashboard\_json = jsonencode({

"displayName": "Dashboard Principal - Staging - SkillForge AI",

"gridLayout": {

"columns": "2",

"widgets": [

{

"title": "Taux d'Erreur 5xx (Tous les Services)",

"xyChart": {

"dataSets": [

{

"timeSeriesQuery": {

"timeSeriesQueryLanguage": "fetch cloud\_run\_revision | metric 'run.googleapis.com/request\_count' | filter metric.response\_code\_class = '5xx' | align rate(1m) | every 1m | group\_by [], [value\_request\_count\_aggregate: sum(value.request\_count)]"

},

"plotType": "LINE"

}

],

"timeshiftDuration": "0s"

}

},

{

"title": "Utilisation CPU - Cloud SQL",

"xyChart": {

"dataSets": [

{

"timeSeriesQuery": {

"timeSeriesQueryLanguage": "fetch cloudsql\_database | metric 'cloudsql.googleapis.com/database/cpu/utilization' | group\_by 1m, [value\_utilization\_mean: mean(value.utilization)] | every 1m"

},

"plotType": "LINE"

}

],

"timeshiftDuration": "0s"

}

}

]

}

})

}

# Crée le dashboard dans Google Cloud Monitoring.

resource "google\_monitoring\_dashboard" "main\_dashboard" {

project = var.project\_id

dashboard\_json = local.main\_dashboard\_json

}

#### **6.3. Code Terraform pour les Politiques d'Alerte**

Nous définissons maintenant les règles qui nous notifieront automatiquement en cas de problème.

1. **Ajoutez le code suivant au fichier monitoring.tf :**

# terraform/environments/staging/monitoring.tf (à la suite)

# Crée un canal de notification par e-mail (remplacez par un e-mail de votre groupe de distribution).

resource "google\_monitoring\_notification\_channel" "email\_alerts" {

display\_name = "Alertes DevOps par Email"

type = "email"

labels = {

email\_address = "devops-alerts@yourcompany.com"

}

}

# --- Alerte 1: Taux d'erreur 5xx élevé ---

resource "google\_monitoring\_alert\_policy" "high\_5xx\_rate" {

display\_name = "[Staging] Taux d'Erreur 5xx élevé sur les services Cloud Run"

combiner = "OR" # Déclenche si une des conditions est remplie.

conditions {

display\_name = "Le taux d'erreur 5xx dépasse 2% pendant 5 minutes"

condition\_threshold {

filter = "metric.type=\"run.googleapis.com/request\_count\" resource.type=\"cloud\_run\_revision\" metric.label.response\_code\_class=\"5xx\""

comparison = "COMPARISON\_GT"

threshold\_value = 0.02 # 2%

duration = "300s" # 5 minutes

# Calcule le ratio entre les erreurs 5xx et toutes les requêtes.

trigger {

percent = 100

}

denominator\_filter = "metric.type=\"run.googleapis.com/request\_count\" resource.type=\"cloud\_run\_revision\""

}

}

# Lie l'alerte au canal de notification par e-mail.

notification\_channels = [google\_monitoring\_notification\_channel.email\_alerts.name]

}

# --- Alerte 2: Utilisation CPU élevée sur la base de données ---

resource "google\_monitoring\_alert\_policy" "high\_sql\_cpu" {

display\_name = "[Staging] Utilisation CPU de l'instance Cloud SQL élevée"

combiner = "OR"

conditions {

display\_name = "Le CPU dépasse 90% pendant plus de 15 minutes"

condition\_threshold {

filter = "metric.type=\"cloudsql.googleapis.com/database/cpu/utilization\" resource.type=\"cloudsql\_database\""

comparison = "COMPARISON\_GT"

threshold\_value = 0.9 # 90%

duration = "900s" # 15 minutes

trigger {

count = 1

}

}

}

notification\_channels = [google\_monitoring\_notification\_channel.email\_alerts.name]

}

#### **6.4. Déploiement Final**

1. Naviguez dans le dossier terraform/environments/staging/.
2. Exécutez terraform plan pour visualiser la création du dashboard et des politiques d'alerte.
3. Exécutez terraform apply et confirmez par yes.

### **Conclusion Finale du Guide d'Implémentation**

Ce guide vous a accompagné à travers toutes les étapes de la construction et du déploiement de la plateforme SkillForge AI. En partant de zéro, nous avons :

1. **Initialisé** un projet GCP et un dépôt GitHub sécurisés.
2. **Construit** une infrastructure réseau, de stockage et de données robuste avec Terraform.
3. **Créé** des usines logicielles (pipelines CI/CD) modulaires et réutilisables.
4. **Déployé** l'ensemble de nos services et agents de manière entièrement automatisée.
5. **Mis en place** les outils d'observabilité pour surveiller la santé de notre plateforme.