esaip La Salle
ÉCOLE D'INGÉNIEURS

Rapport Technique : Plateforme de Diffusion Cloud-Native

Date : 05 Février 2026

Auteur : M.Marsault, A.Bonneau, L.Maillet

Matière : Cloud Computing

1. Architecture & Dépôt GitHub

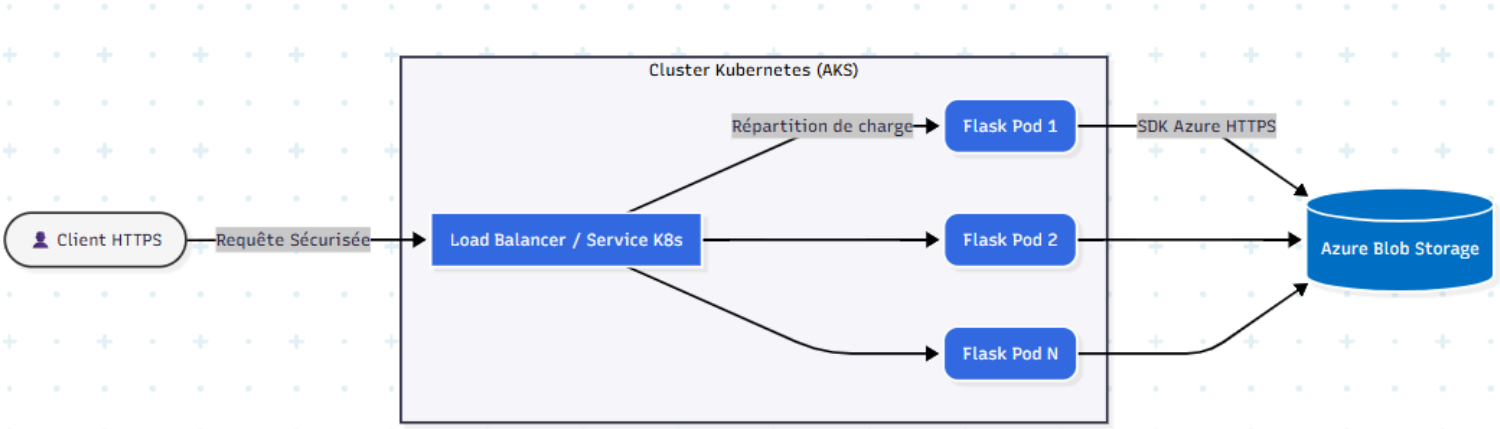
1.1 Architecture Globale

Pour répondre au besoin de diffuser du contenu statique de manière dynamique, nous avons conçu une architecture **Cloud-Native** reposant sur les principes suivants :

- **Découplage :** L'application est un microservice Flask (stateless) conteneurisé.
- **Stockage externe :** Les données (JSON/YAML) sont stockées sur **Azure Blob Storage**, garantissant la persistance indépendamment du cycle de vie de l'application.
- **Orchestration :** L'infrastructure cible est Kubernetes (AKS) pour assurer la scalabilité et la haute disponibilité.

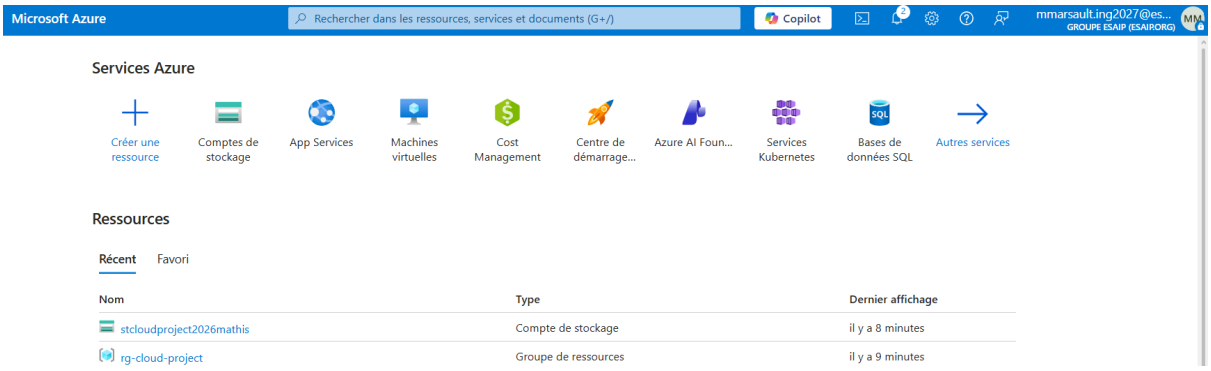
Schéma d'architecture logique :

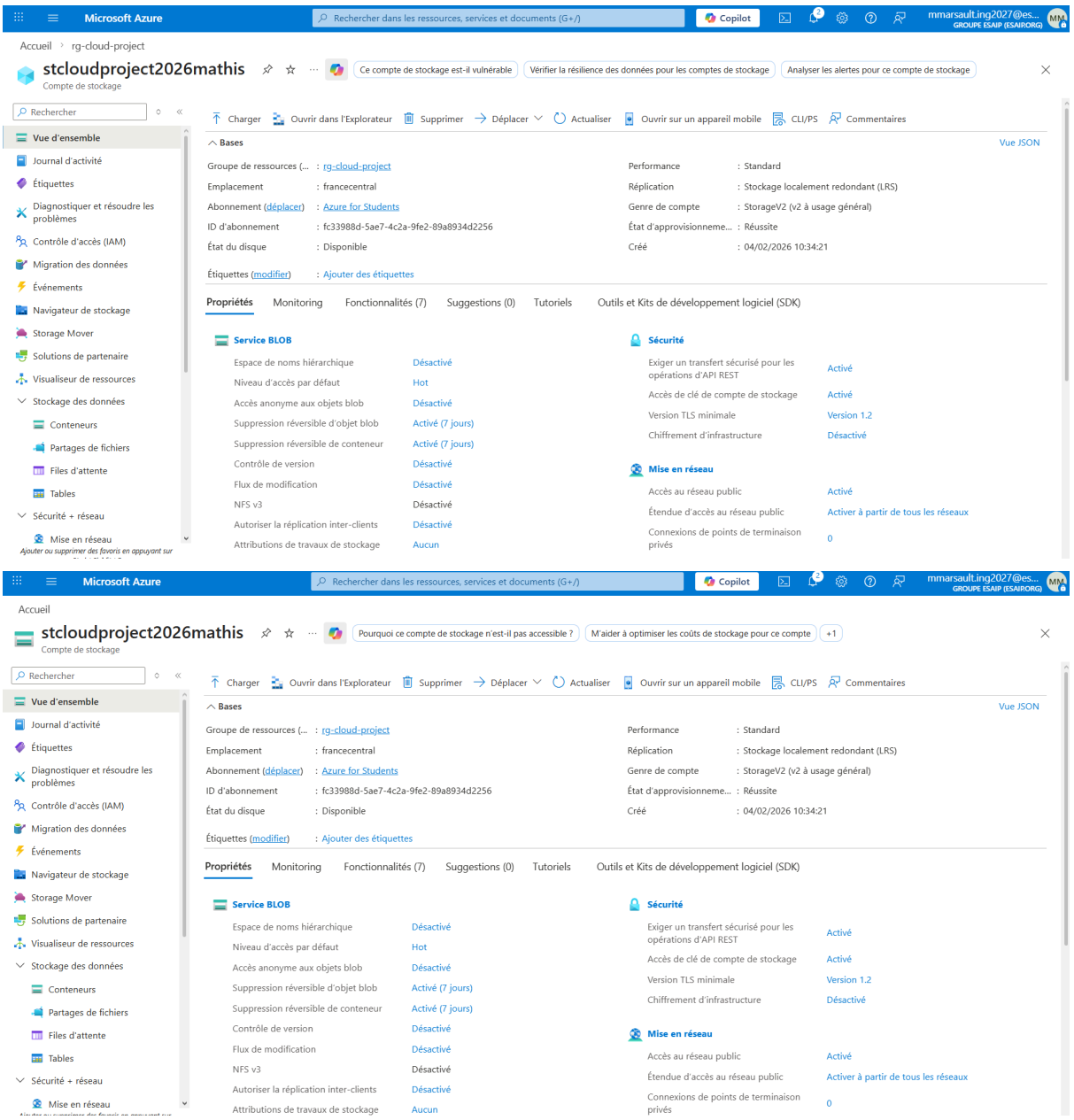
Client HTTPS -> Load Balancer (Service K8s) -> Pods Flask (xN) -> Azure Blob Storage



Réalisé sur MermaidLive

Initialisation sur Azure:





1.2 Structure du Dépôt et Stratégie Git

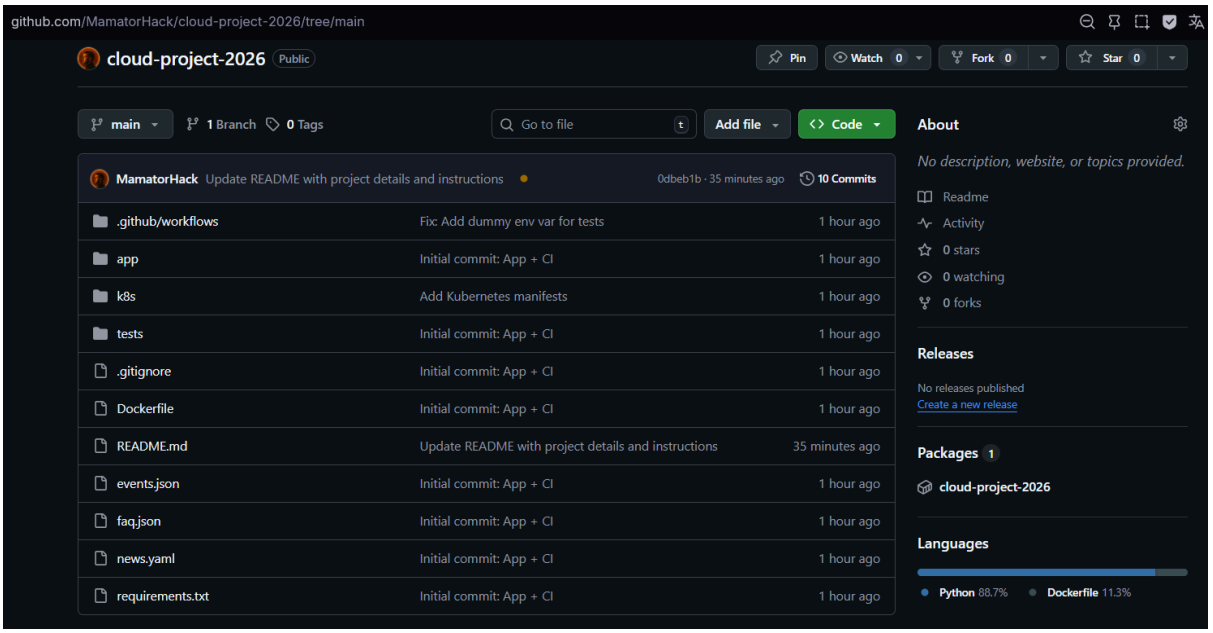
Le projet est hébergé sur GitHub : `MamatorHack/cloud-project-2026`.

Réponses aux questions :

1. **Pourquoi cette stratégie Git (Trunk-based) ?**
   Nous avons opté pour le **Trunk-based development**. Étant une petite équipe sur un cycle de développement court, cette stratégie permet d'itérer rapidement sans la lourdeur de gestion des branches de *Git Flow*. Chaque fonctionnalité validée est directement intégrée au `main`, déclenchant la CI.
2. **Comment garantir un historique lisible et maintenable ?**
   Nous utilisons des messages de commit atomiques et descriptifs (ex: `Fix: lowercase`

image name, Add Kubernetes manifests). L'usage de Pull Requests avec "Squash" permet de garder l'historique principal propre.

3. **Quels fichiers ne doivent jamais être versionnés ?**
   Les fichiers contenant des secrets (.env), les dossiers de compilation (__pycache__, venv/), et les configurations locales de l'IDE (.vscode/, .idea/). Notre fichier .gitignore est configuré en conséquence.



---

2. Flask Local (Développement Application)

L'application Backend a été développée en Python avec le framework **Flask**.

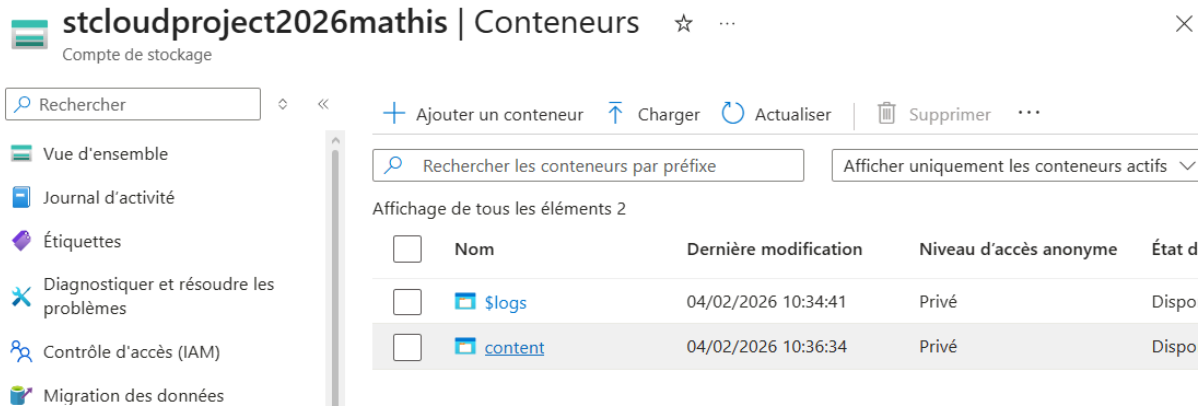
2.1 Fonctionnalités Implémentées

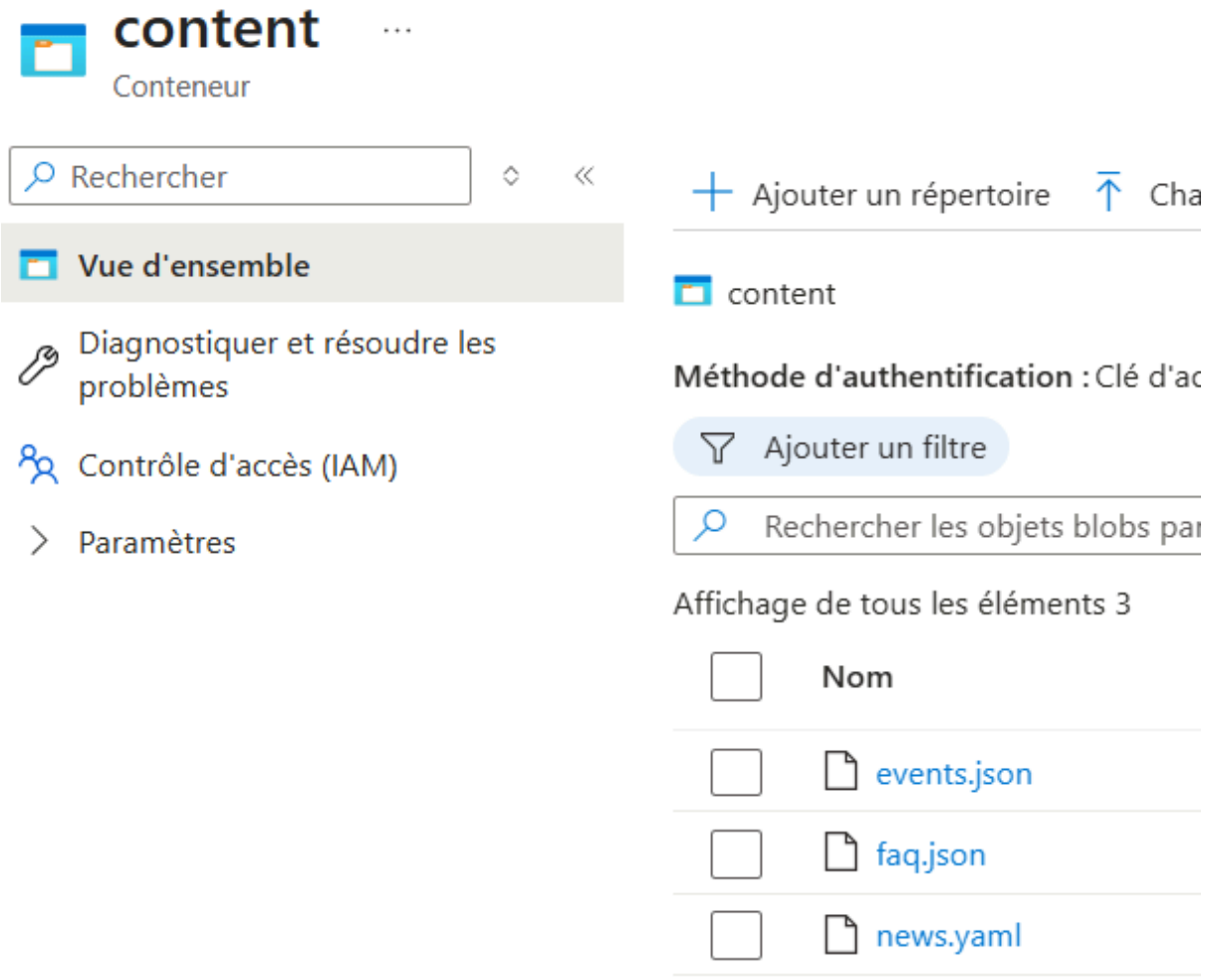
Conformément au cahier des charges, l'application :

1. Se connecte à un conteneur Azure Blob Storage nommé content.
2. Expose les endpoints REST requis : /api/events, /api/news, /api/faq.
3. Intègre un **cache mémoire** (TTL 60s) pour réduire la latence et les coûts de transaction vers Azure.
4. Récupère la chaîne de connexion via la variable d'environnement AZURE_STORAGE_CONNECTION_STRING pour respecter les bonnes pratiques de sécurité (12-Factor App).

2.2 Données

Les fichiers events.json, faq.json et news.yaml ont été uploadés sur le compte de stockage stcloudproject2026mathis.


stcloudproject2026mathis | Conteneurs
Compte de stockage
Rechercher
Vue d'ensemble
Journal d'activité
Étiquettes
Diagnostiquer et résoudre les problèmes
Contrôle d'accès (IAM)
Migration des données
Ajouter un conteneur
Charger
Actualiser
Supprimer
Rechercher les conteneurs par préfixe
Afficher uniquement les conteneurs actifs
Affichage de tous les éléments 2
Nom
Dernière modification
Niveau d'accès anonyme
État d
$logs
04/02/2026 10:34:41
Privé
Dispo
content
04/02/2026 10:36:34
Privé
Dispo


content
Conteneur
Rechercher
Vue d'ensemble
Diagnostiquer et résoudre les problèmes
Contrôle d'accès (IAM)
Paramètres
Ajouter un répertoire
Cha
content
Méthode d'authentification : Clé d'ac
Ajouter un filtre
Rechercher les objets blobs par
Affichage de tous les éléments 3
Nom
events.json
faq.json
news.yaml


127.0.0.1:5000
Plateforme Cloud-Native
API Running

3. Tests Flask (Assurance Qualité)

Nous avons appliqué une approche **Test-Driven** pour garantir la robustesse du code avant tout déploiement.

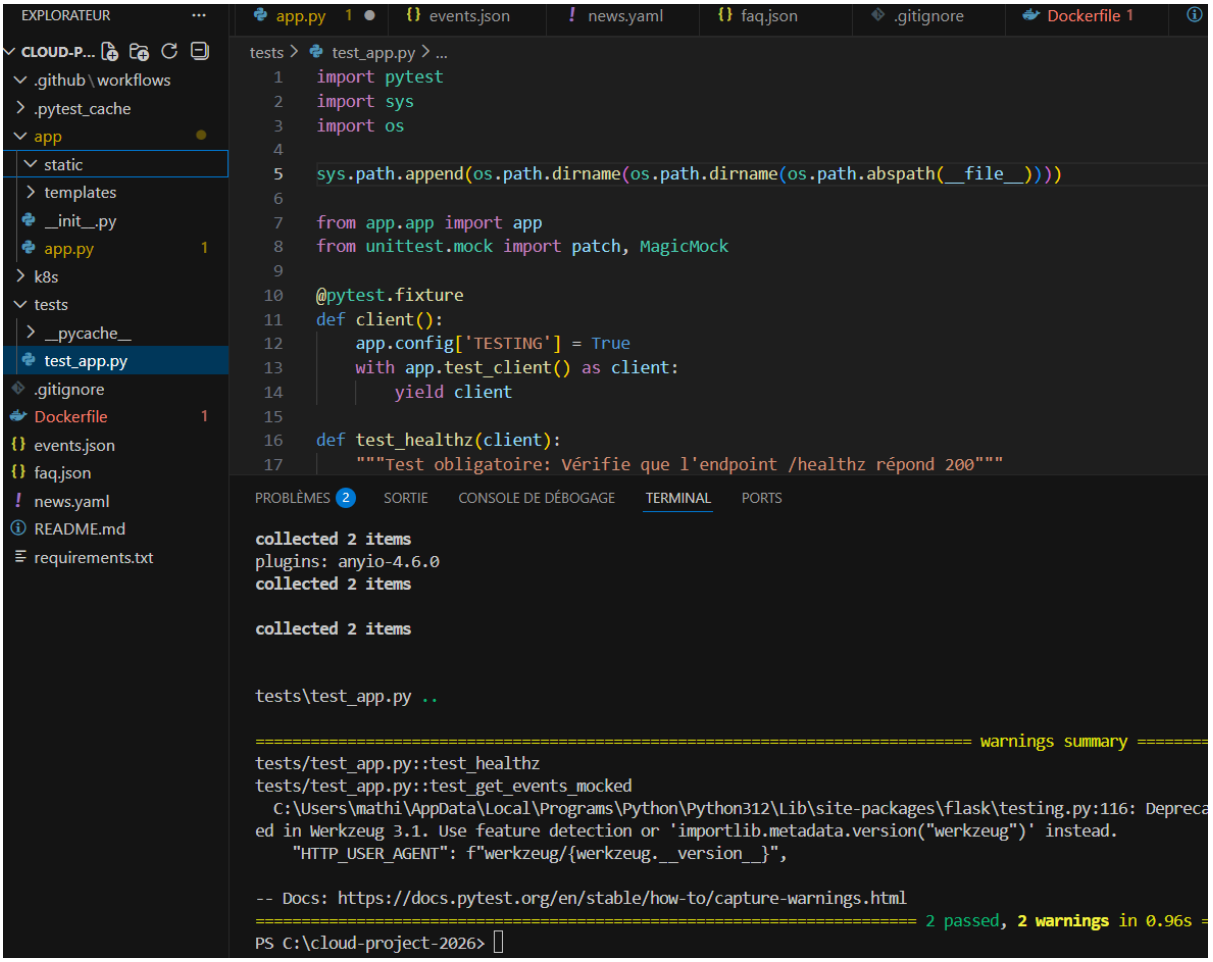
3.1 Stratégie de Mocking

Pour rendre les tests **indépendants de l'environnement Azure** (et donc gratuits et exécutables hors ligne), nous utilisons la librairie `unittest.mock`.

- Nous simulons la réponse du `BlobServiceClient`.
- Cela permet de valider la logique de parsing JSON/YAML sans effectuer de véritable appel réseau.

3.2 Périmètre Testé

- **Health Checks :** `/healthz` (Code 200 OK).
- **Fonctionnels :** `/api/events` (Vérification de la structure JSON de retour).



4. Conteneurisation (Docker)

L'application a été packagée sous forme d'image Docker pour garantir la portabilité.

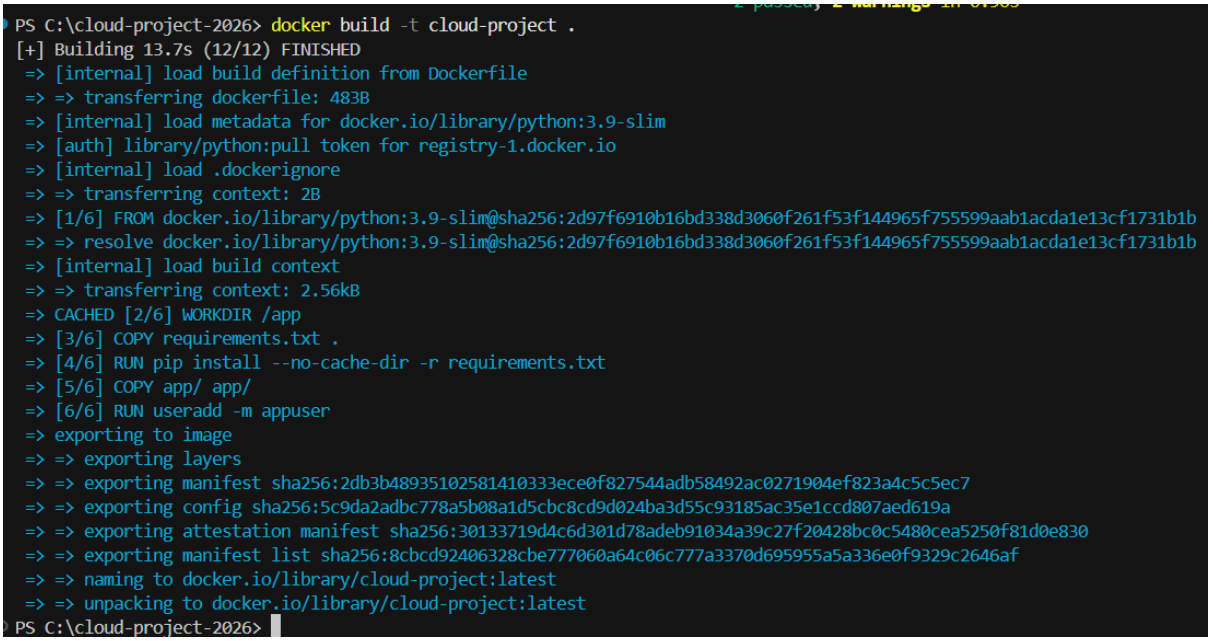
4.1 Optimisation du Dockerfile

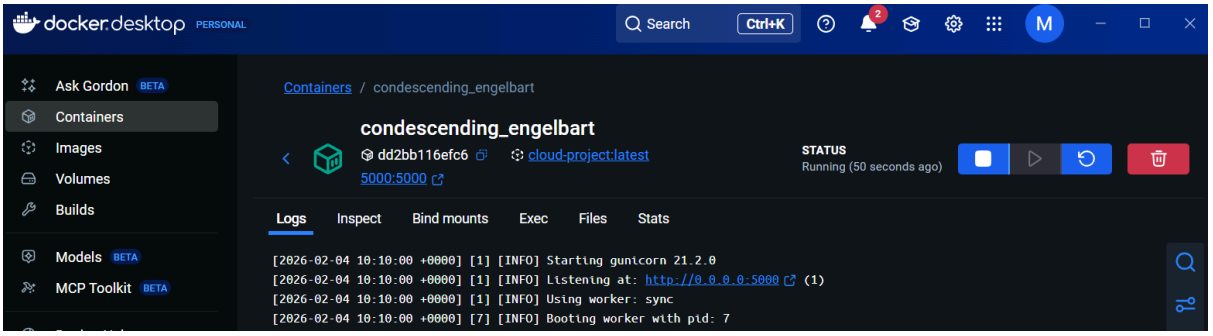
Nous avons construit un Dockerfile optimisé pour la production :

- **Image de base :** `python:3.9-slim` pour réduire la surface d'attaque et la taille.
- **Sécurité :** Création d'un utilisateur système dédié (`useradd -m appuser`) pour ne pas exécuter le conteneur en `root`.
- **Dépendances :** Installation stricte via `requirements.txt`.

Réponses aux questions :

1. **Comment réduire la taille de l'image ?**
   En utilisant des images "slim" ou "alpine", en évitant d'installer des outils de debug (curl, vim) et en nettoyant le cache de pip (`--no-cache-dir`). L'utilisation du multi-stage build est aussi une solution recommandée.
2. **Pourquoi l'image Docker est-elle un paquet binaire d'application ?**
   Elle est immuable. Une fois construite, elle contient le code, le runtime et les librairies système nécessaires. Cela garantit que l'artefact testé en local est *bit-pour-bit* identique à celui déployé en production.
3. **Pourquoi le conteneur doit-être stateless ?**
   Pour permettre la scalabilité horizontale (ajout de répliques) et la résilience. Si un conteneur plante, Kubernetes le remplace par un nouveau qui n'a pas besoin de récupérer l'état local du précédent (les données sont dans le Blob Storage).






```
PS C:\cloud-project-2026> docker run -p 5000:5000 -e AZURE_STORAGE_CONNECTION
Ou1Fm2l5EH7ob3lsHJSYx43bmtcNjuf0Y6QrKtiIIee2SASLxrFVnjjKHtiMZkFMr+ASt9UOhSw==
[2026-02-04 10:05:44 +0000] [1] [INFO] Starting gunicorn 21.2.0
[2026-02-04 10:05:44 +0000] [1] [INFO] Listening at: http://0.0.0.0:5000 (1)
[2026-02-04 10:05:44 +0000] [1] [INFO] Using worker: sync
[2026-02-04 10:05:44 +0000] [7] [INFO] Booting worker with pid: 7
[2026-02-04 10:07:18 +0000] [1] [INFO] Handling signal: int
[2026-02-04 10:07:18 +0000] [7] [INFO] Worker exiting (pid: 7)
[2026-02-04 10:07:18 +0000] [1] [INFO] Shutting down: Master
```

Docker va télécharger Python, installer les libs, créer l'utilisateur sécurisé, et emballer le tout.

---

5. CI (Intégration Continue sans AKS)

Nous avons mis en place un pipeline **GitHub Actions** complet défini dans `.github/workflows/ci.yaml`.

5.1 Workflow Automatisé

À chaque `push` sur la branche `main`, le pipeline exécute :

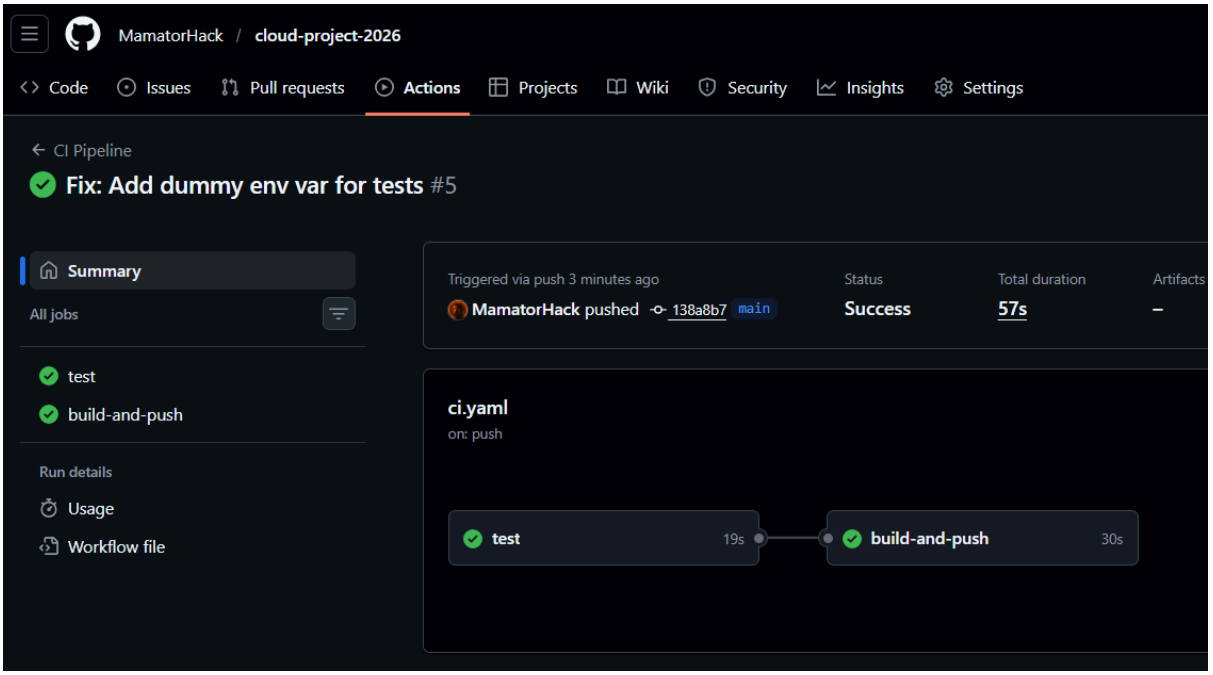
1. **Job Test :** Installation de Python, résolution des dépendances (fix version `Werkzeug`), injection de variables d'environnement factices pour les tests, et exécution de `pytest`.
2. **Job Build & Push :** Connexion au registre, construction de l'image Docker et publication.

5.2 Choix Techniques

Réponses aux questions :

1. **Pourquoi GHCR plutôt qu'Azure Container Registry (ACR) ?**
   GHCR (GitHub Container Registry) est nativement intégré à notre dépôt de code, ne nécessite pas de gestion complexe de service principal (le `GITHUB_TOKEN` suffit), et est gratuit pour les projets publics et étudiants.
2. **Comment gérer les secrets dans le pipeline ?**
   Les secrets sensibles (crédentials Azure si nécessaires) sont stockés dans les **GitHub Repository Secrets** et injectés dynamiquement comme variables d'environnement. Ils n'apparaissent jamais en clair dans les fichiers YAML ou les logs.

3. **Quelle stratégie de rollback ?**
   En cas de régression, nous pouvons soit relancer un ancien workflow GitHub Actions ("Re-run jobs"), soit utiliser `git revert` pour annuler le commit problématique, ce qui déclenche un nouveau build sain.



Nous avons automatisé tout le cycle de production logicielle.

Concrètement :

- **Job `test` validé :** GitHub a démarré un ordinateur, installé Python, installé tes dépendances (avec la bonne version de Werkzeug), et lancé tes tests. Tout fonctionne, le code est robuste.
- **Job `build-and-push` validé :** GitHub a réussi à lire le `Dockerfile`, construire l'image, et l'envoyer dans le registre (GHCR).

Résultat : l'application est désormais "packagée" et stockée dans le Cloud, prête à être téléchargée par n'importe quel serveur.

---

6. Orchestration Kubernetes (AKS)

6.1 État du Déploiement

Les manifestes Kubernetes (`deployment.yaml`, `service.yaml`) ont été créés et sont présents dans le dossier `k8s/` du dépôt.

Cependant, le déploiement effectif sur Azure Kubernetes Service (AKS) n'a pas pu aboutir.

Cause technique : Limitations strictes de l'abonnement *Azure for Students*.

Malgré nos tentatives d'optimisation (utilisation de VM `Standard_D2s_v3`, changement de région vers West Europe, nettoyage des IP publiques), la création du cluster échoue systématiquement sur une erreur de **Quota vCPU** (limite de 4 vCPU régionaux atteinte par l'infrastructure minimale du cluster) ou de **Policy** (régions interdites).

6.2 Concepts Kubernetes

Réponses aux questions :

1. **Rôle de chaque ressource ?**
    - **Deployment :** Définit l'état désiré (ex: 2 répliques de l'app Flask) et gère les mises à jour progressives.
    - **Service :** Expose l'application réseau (LoadBalancer) et répartit la charge entre les pods.
    - **Secret :** Stocke la `CONNECTION_STRING` encodée en base64 pour sécuriser l'accès au Blob.
2. **Différence entre readiness et liveness ?**
    - **Liveness :** Vérifie si l'app est "vivante". Si échec -> Le pod est redémarré (utile en cas de deadlock).
    - **Readiness :** Vérifie si l'app est "prête à recevoir du trafic". Si échec -> Le pod est temporairement retiré du LoadBalancer (utile au démarrage).
3. **Impact des resources sur la scalabilité ?**
   Définir des `requests` (garanti) et `limits` (plafond) pour le CPU/RAM permet au scheduler Kubernetes de placer intelligemment les pods sur les nœuds et d'éviter qu'un pod buggé ne sature tout le cluster.

---

7. Monitoring & Sécurité

7.1 Surveillance

Réponses aux questions :

1. **Métriques utiles :** Taux d'erreurs HTTP (5xx), latence des requêtes, et saturation CPU/RAM des conteneurs.
2. **Pourquoi éviter une journalisation excessive ?**
   Les logs ont un coût (stockage et ingestion Azure Monitor) et trop de logs ("bruit") rendent le débogage difficile en noyant les vraies erreurs.
3. **Comment limiter les coûts Azure Monitor ?**
   En ne loggant que les niveaux `WARNING` et `ERROR` en production et en définissant une politique de rétention des logs courte (ex: 30 jours).

7.2 Sécurité

Réponses aux questions :

1. **Pourquoi ne pas stocker de secrets dans Git ?**
   Git conserve l'historique indéfiniment. Un secret commité une seule fois est compromis pour toujours et peut être récupéré par des bots malveillants.

2. **Avantages Managed Identity vs clé statique ?**
   L'identité gérée (Managed Identity) supprime la gestion des mots de passe. C'est l'infrastructure Azure qui authentifie le pod AKS auprès du Blob Storage. Une clé statique doit être stockée, rotée et protégée, ce qui ajoute un risque de fuite.
3. **Risques de fuite dans les logs ?**
   Si l'application loggue l'environnement au démarrage ou les requêtes brutes, la chaîne de connexion (contenant la clé) peut se retrouver en clair dans les logs, accessible à toute personne ayant accès au monitoring.

---

8. Retour d'expérience

Ce projet nous a permis de mettre en œuvre une chaîne DevOps complète.

Ce qui a fonctionné :

- L'architecture logicielle est fonctionnelle et respecte les principes Cloud-Native (Stateless, 12-Factor).
- La conteneurisation et l'automatisation via GitHub Actions sont opérationnelles : le code est testé, buildé et publié automatiquement.
- La collaboration via Git a été fluide.

Difficultés et Limites :

- Le point bloquant majeur a été l'infrastructure Azure. Les quotas étudiants sont extrêmement restrictifs pour Kubernetes (nécessitant plusieurs vCPUs pour le plan de contrôle et les nœuds). Malgré une configuration IaC (Infrastructure as Code) correcte, le déploiement final reste théorique dans ce contexte précis.
- Une amélioration future consisterait à utiliser une identité gérée (Workload Identity) plutôt qu'une clé d'accès pour renforcer la sécurité.