

## Sujet de projet : Cloud Monitor (Golang)

## **Contexte**

Scaleway (ou toute autre plateforme cloud) gère des milliers d'instances réparties sur différents datacenters.

Les équipes ont besoin d'un outil simple et rapide pour surveiller l'état de leurs instances, collecter leurs métriques, et exposer ces informations via une API.

Ton objectif est de concevoir un mini service de monitoring cloud, codé entièrement en Go, qui simule un environnement cloud avec plusieurs serveurs et permet de :

- suivre leurs performances,
- exposer les métriques collectées,
- et gérer le cycle de vie du monitoring (ajout, suppression, arrêt propre).

Le projet doit être structuré comme une vraie application de production, avec une architecture claire, un code idiomatique Go, et une bonne gestion de la concurrence.

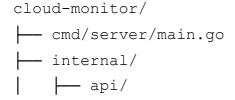
## 🎯 Objectifs pédagogiques

- Maîtriser les concepts fondamentaux de Go : goroutines, channels, context, erreurs, interfaces.
- Concevoir une API REST propre et maintenable.
- Gérer la concurrence et la synchronisation des tâches.
- Découvrir les bonnes pratiques d'architecture logicielle en Go.
- Approcher les problématiques cloud / observabilité / monitoring.

# 🍃 Rendu attendu

Tu devras livrer:

1. Un repository Git clair et organisé :



#### 2. Un **README** expliquant:

- o le but du projet,
- o les instructions d'installation et d'exécution,
- o les routes disponibles,
- les choix techniques,
- et une courte démonstration (captures ou gif).
- 3. Des tests unitaires sur les packages critiques (monitor, api, metrics).

## **Solution** Fonctionnalités demandées

#### Niveau 1 - Base

- Lancer un serveur HTTP(:8080 par défaut).
- Endpoint /instances:
  - GET → liste les instances surveillées.
  - POST → ajoute une instance (nom, IP fictive, intervalle de refresh).
  - DELETE /instances/:id → supprime une instance.
- Les instances sont stockées en mémoire (dans une map protégée par un mutex).

### Niveau 2 - Monitoring concurrent

- Chaque instance a un worker qui exécute une goroutine dédiée.
- Le worker collecte périodiquement des métriques simulées :
  - CPUUsage (0-100%)
  - MemoryUsage (0-100%)
  - Uptime (secondes depuis le lancement)

- Les métriques sont actualisées toutes les X secondes selon la configuration.
- Les workers s'arrêtent proprement à la suppression de l'instance ou lors du SIGINT.

#### Niveau 3 – API des métriques

- Endpoint /metrics/:id:
  - GET → retourne les dernières métriques de l'instance.
- Endpoint /health:
  - GET → retourne l'état global du système (nb d'instances, uptime du serveur, etc.).

### Niveau 4 - Persistance (optionnel mais recommandé)

- Les métriques sont stockées en base (SQLite ou PostgreSQL).
- Une abstraction Storage doit exister pour permettre de changer facilement d'implémentation (in-memory / SQL).

#### Niveau 5 — Configuration et logs

- Fichier config.yaml ou .env pour définir:
  - le port HTTP,
  - la fréquence par défaut,
  - le mode de stockage.
- Logs structurés (zap, zerolog ou package perso).

#### Niveau 6 — CLI (bonus)

- Une commande cloudctl avec cobra :
  - cloudctl list → liste les instances
  - cloudctl add <name> <ip> → ajoute une instance
  - cloudctl metrics <id> → affiche les métriques

# Bonus "Scaleway-style"

Ajoute une vraie intégration avec l'API publique Scaleway (ou une simulation).

- Déploie ton app sur Scaleway Instances ou Serverless Containers.
- Implémente une interface simple (HTML/JS ou Grafana) pour visualiser les métriques.

## **Exemples de routes**

```
# Ajouter une instance
curl -X POST -H "Content-Type: application/json" \
    -d '{"name":"instance-1","ip":"10.0.0.1","interval":5}' \
    http://localhost:8080/instances

# Récupérer la liste
curl http://localhost:8080/instances

# Récupérer les métriques
curl http://localhost:8080/metrics/1

# Supprimer une instance
curl -X DELETE http://localhost:8080/instances/1
```