1 - Architecture

1.1 - Les options

1.1.1 - Approche Single Pod + Async

Un seul pod `alert-worker` traite plusieurs messages **en parallèle** grâce à `asyncio` (ex. `prefetch_count=10` et `asyncio.create_task`).

Avantages :

- Consommation optimale des ressources d'un pod.
- Architecture plus simple
- Réduction de la latence car un seul pod maintient la connexion AMQP

Inconvénients :

- Limité par les ressources d'un seul conteneur.
- Scaling horizontal compliqué ?? c'est pas possible via k8s?

Cas d'usage :

- Faible volume d'alertes ou I/O-bound, ce qui est le cas ici à priori car la seule action du worker est de télécharger la vidéos pour extraire des metadatas

1.1.2 - Approche Multi-Pods (scaling horizontal)

Plusieurs pods `alert-worker` (ex. 3 à 10 réplicas) consomment la même queue RabbitMQ.

Avantages :

- Scalabilité horizontale

Inconvénients :

- Plus de connexions AMQP (1 par pod).
- Lancement et gestion de plusieurs pods.
- Peut être légèrement plus coûteux en ressources.

Cas d'usage :

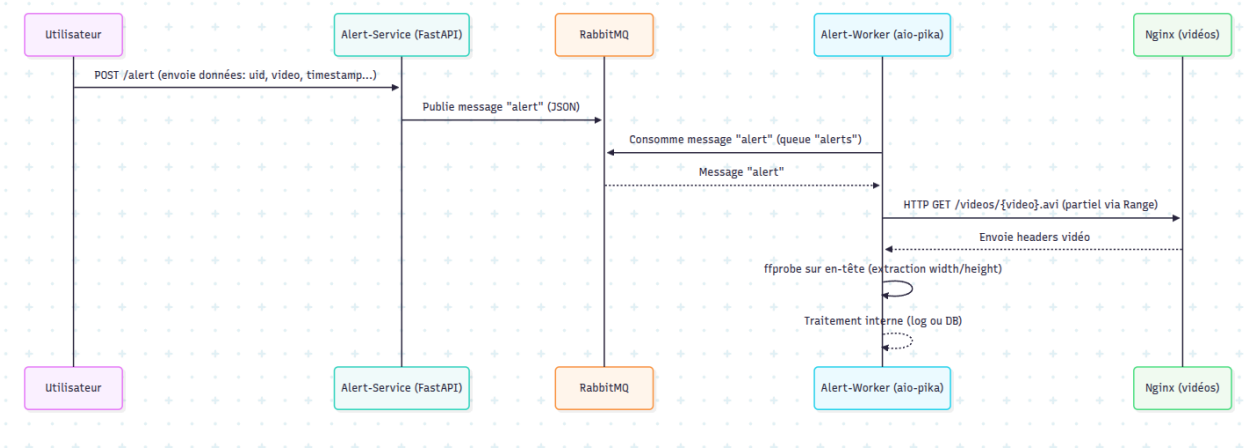
- Volume élevé d'alertes.
- Traitements CPU intensifs.
- Besoin de tolérance aux pannes.

1.1.3 - Décision technique

La solution choisie est hybride :

- Le worker supporte `asyncio` pour paralléliser localement (2-5 messages).
- Kubernetes scale pour adapter le nombre de pods selon la charge.

1.2 - Diagramme de séquence



2 - Missing

- Réponse au client, il faut passer par une notification en push via un Webhook
- Pas de tests, ni unitaire ni d'intégration
- Auto-scaling (HPA) basé sur la taille de la queue RabbitMQ.
- Monitoring (Prometheus + Grafana) pour suivre le nombre d'alertes traitées et les erreurs.
- Tests de charge pour déterminer le ratio optimal entre `CONCURRENT_TASKS` (async) et `replicas` (pods).
- Architecture hexagonale pour découpler la DB et RabbitMQ