

Cloud computing

Rapport PolySnap



Equipe G

Zoubair Hamza - Yahiaoui Imène - Gazzeh Sourour
Ben Aissa Nadim - Al Achkar Badr

Sommaire:

Introduction.....	3
Analyse des besoins.....	4
Définition du périmètre MVP.....	6
Diagramme d'architecture logicielle.....	8
Modèles des données.....	12
Diagramme de déploiement.....	14
Services utilisés dans le déploiement :	15
Les 12 facteurs.....	17
Présentation de la supervision envisagée du système.....	19
Monitoring et Alerting :	19
Coûts prévisionnels :	21
Prise de recul.....	25

Introduction

Le présent rapport présente Polysnap, une plateforme de réseau social dédiée au partage de moments de vie à travers une variété de médias. Il commence par une analyse des besoins du système, pour ensuite décrire en détail le périmètre du Minimum Viable Product (MVP), l'architecture logicielle, la stratégie de déploiement cloud, ainsi que les coûts prévisionnels et la supervision envisagée du système.

Analyse des besoins

Lors de l'analyse des besoins, nous avons identifié cinq personas représentant des utilisateurs potentiels, chacun caractérisé par un profil unique et des besoins spécifiques par rapport à notre application.

Persona 1 : Flora

- Profil : Mère au foyer avec un intérêt pour la cuisine et la vie familiale.
- Besoins:
 - En tant que mère au foyer, je veux une procédure de création de compte simple afin de faciliter mon accès à la plateforme.
 - En tant que mère au foyer, je veux des fonctionnalités de groupe pour le partage des recettes et des conseils avec ma famille, en privilégiant les images et les vidéos.

Persona 2 : Valentin

- Profil : Étudiant étranger en Allemagne
- Besoins:
 - En tant qu'étudiant étranger en Allemagne, je veux une fonctionnalité pour ajouter de nouveaux contacts et élargir mon réseau social en Allemagne, afin de développer ma vie sociale dans un nouveau pays.
 - En tant qu'étudiant étranger en Allemagne, je veux une messagerie texte pour maintenir un lien constant avec mes amis, malgré la distance, afin de rester connecté.

Persona 3 : Paul

- Profil : Créateur de contenu et utilisateur actif des réseaux sociaux
- Besoins:
 - En tant que créateur de contenu et utilisateur actif des réseaux sociaux, je veux partager mes créations artistiques avec une communauté, en privilégiant le partage de vidéos et d'images dans des stories éphémères, afin de permettre une expression artistique dynamique et visuelle.

Persona 4 : Alex

- Profil : d'un professionnel qui attache de l'importance à la confidentialité et à la sécurité dans les communications.
- Besoins :
 - En tant que professionnel soucieux de la confidentialité de mes communications, je veux pouvoir échanger des "messages éphémères" pour assurer la sécurité de mes échanges d'informations sensibles, en veillant à ce que ces messages ne restent pas accessibles indéfiniment.

Persona 5 : Sarah

- Profil: Voyageuse passionnée qui adore découvrir de nouveaux endroits.
- Besoins:
 - En tant que voyageuse passionnée, je veux une fonctionnalité qui me permet de partager ma position en temps réel avec mes amis au travers d'une conversation, afin de les tenir informés de mes aventures et de la sécurité lors de mes voyages.

Définition du périmètre MVP

Suite à notre analyse des besoins du client et des spécifications qui nous ont été fournies, nous avons mis en œuvre un MVP qui couvre la plupart des user stories qui nous ont été demandées. Voici une liste de ce qui a été implémenté sous forme de scénarios :

Persona 1 - Flora

Scénario : Flora, une mère au foyer passionnée de cuisine, souhaite utiliser une plateforme de cuisine pour partager des recettes et des conseils avec sa famille. Voici comment elle réalise ses besoins :

1. Flora accède à la plateforme et choisit de "Créer un compte".
2. Elle suit une procédure de création de compte simple en fournissant son nom, son adresse e-mail et un username.
3. Une fois inscrite, Flora crée un groupe de messagerie de cuisine et commence à partager ses recettes, des images et des vidéos pour interagir avec sa famille.

Persona 2 - Valentin

Scénario : Valentin, un étudiant étranger en Allemagne, veut maintenir le contact avec ses amis malgré la distance. Voici comment il réalise ses besoins :

1. Il crée un compte et rejoint la plateforme.
2. Valentin ajoute ses amis à ses contacts à l'aide de leurs usernames.
3. Il commence à créer des discussions individuelles (ou de groupes) avec ses amis et à envoyer des messages textuels pour rester en contact avec eux.

Persona 3 - Paul

Scénario : Paul, un créateur de contenu actif sur les réseaux sociaux, souhaite partager ses créations artistiques de manière dynamique avec ces contacts proches. Voici comment il réalise ses besoins :

1. Paul se connecte à sa plateforme de médias sociaux préférée Polysnap.
2. Il crée une story éphémère pour partager des vidéos et/ou des images de ses créations artistiques.
3. Ses contacts peuvent voir ces stories et apprécier son expression artistique visuelle pour les 24 heures qui suivent.

Persona 4 - Alex

Scénario : Alex veut partager des détails potentiellement confidentiels avec un client via une application de messagerie. Afin de protéger les informations sensibles, Alex préfère envoyer ses messages en éphémère. Voici comment il réalise ses besoins :

1. Alex ouvre l'application Polysnap.
2. Il initie une discussion avec le client et active la fonction "messages éphémères" pour les messages qui veut faire disparaître en spécifiant la durée avant disparition.
3. Le client consulte les messages, mais après un laps de temps spécifié, les messages disparaissent automatiquement, garantissant que les informations restent sécurisées et confidentielles. Cette fonctionnalité offre à Alex la tranquillité d'esprit en sachant que les données sensibles ne seront pas accessibles après la conversation.

Sous formes de fonctionnalités, voici ce qu'on a implémenté :

PolyUser : Cette fonctionnalité englobe le processus de création de comptes pour les utilisateurs de la plateforme, la connexion à la plateforme et l'ajout des contacts.

PolySmoke : Il s'agit d'une messagerie instantanée permettant des échanges tant en privé qu'en groupe. Cette fonctionnalité prend en charge différents types de contenu, qu'il s'agisse de médias ou de texte, et propose également la possibilité de messages éphémères.

PolyStories : Cette fonction permet aux utilisateurs de partager des stories visuelles, comprenant aussi bien des images que des vidéos, avec une durée de visibilité limitée à 24 heures.

Diagramme d'architecture logicielle

Lien de la version SVG plus claire : [Lien SVG](#) (à ouvrir dans draw.io)

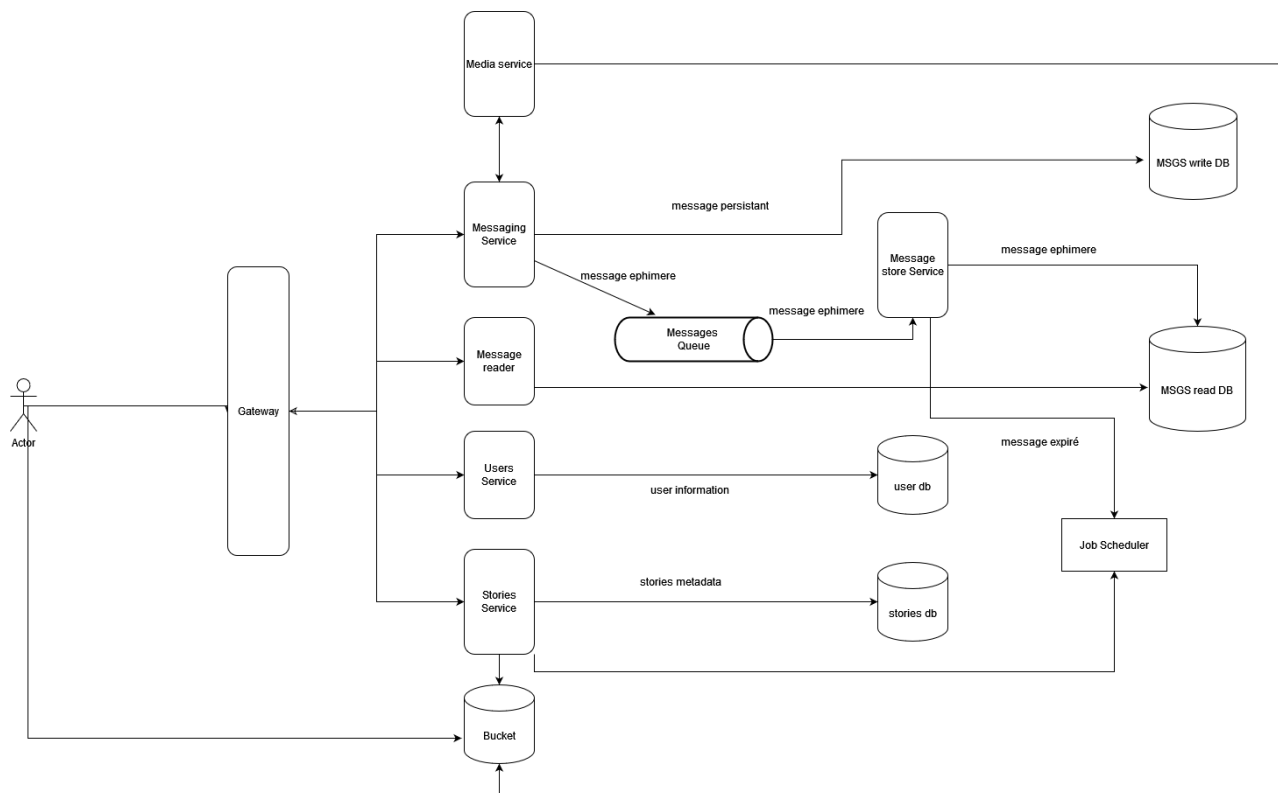


Figure 1 : Diagramme d'architecture de PolySnap

Nous avons mis en place une architecture fondée sur des microservices, où chaque élément représente un microservice de type "stateless". Nous avons divisé nos microservices autour des capacités de l'entreprise et les avons rendus stateless pour pouvoir évoluer horizontalement sans se soucier de la perte d'état, de l'incohérence de l'état ou de l'état persistant qui nécessiterait une mise en correspondance du contexte des requêtes en provenance des clients, permettant ainsi la montée en échelle.

Cette approche fragmente clairement les responsabilités fonctionnelles au sein de notre système. Chaque microservice est conçu pour gérer une partie spécifique des fonctionnalités de notre métier et il a la capacité de fonctionner de manière autonome, simplifiant ainsi les processus de développement, de déploiement et de mise à jour de nos fonctionnalités.

- **User service**

Ce microservice a été créé principalement pour la gestion des comptes et une authentification légère des utilisateurs.

Notre architecture diffère d'une approche multi-instance et repose sur une installation unique (architecture multi-tenant). Dans ce contexte, il était impératif pour nous d'intégrer un service de création de compte spécifiquement dédié à chaque utilisateur.

Cette mesure vise à restreindre l'accès de chaque utilisateur à ses propres données (i.e. discussions, stories, contacts...), empêchant ainsi tout accès aux stories ou échanges d'un autre utilisateur qui ne figure pas dans sa liste de contacts.

- **Stories service**

Ce microservice est conçu pour répondre aux besoins de partage avec les contacts de médias en voie de disparition, également appelés "stories". Le média peut être une vidéo ou une image, et un utilisateur peut partager plusieurs stories qui ne sont consultables que par ces contacts..

Ce microservice reçoit des requêtes d'upload des stories, génère une url signée pour le faire directement vers un stockage d'objets et la renvoie au client. Les contraintes de taille et de type sont prédéfinies dans l'url. Une fois le téléchargement réussi, il stocke les métadonnées de l'histoire dans une base de données relationnelle (type, heure d'upload en UTC,...), avec l'url permettant de récupérer l'histoire.

Lorsqu'il reçoit des demandes pour voir les histoires d'un utilisateur, il vérifie la disponibilité du média, avant de renvoyer le lien pour le récupérer dans le stockage d'objets. Un planificateur de tâches est chargé de déclencher la suppression périodique des médias du stockage d'objets par ce microservice.

- **Messaging service**

C'est le microservice qui devra avoir le débit le plus élevé dans notre architecture.

Il est chargé de recevoir les demandes de création d'un chat ou d'envoi de messages à un chat spécifique. Il effectue les vérifications nécessaires concernant l'appartenance des utilisateurs au chat en question. Enfin, il stocke les messages dans une base de données NOSQL à forte intensité d'écriture, afin d'assurer la montée en échelle et la persistance des données, puis les envoie à un broker de messages.

Si un message ne parvient pas à être écrit dans la base de données, aucun message n'est écrit dans le système entier et l'expéditeur peut réessayer de l'envoyer. Il s'agit du service "Commande" dans le modèle CQRS mis en œuvre dans notre architecture.

Si le message contient également une pièce jointe (média), il demande une URL signée à notre "service-média" avec la même approche expliquée précédemment. Nous reviendrons plus tard sur ce service.

De même, les messages éphémères sont stockés avec une date d'envoi en UTC, une fenêtre de durée de visualisation et une liste de viewers avec la date de visualisation.

- **Messaging-Store service**

Ce microservice lit à partir de la file d'attente, et écrit à son rythme, les nouveaux messages dans une base de données relationnelle qui est aussi performante en lecture qu'en écriture.

Il est appelé périodiquement par le planificateur de tâches pour supprimer définitivement les médias expirés (vus par tout le monde et disparus). Pour ce faire, il appelle le service média.

- **Messaging-reader service**

Ce microservice est le service de Query dans notre partie CQRS. Il reçoit les demandes de récupération des messages d'une certaine conversation.

Il a été séparé du précédent car nous ne voulons pas perturber l'écriture et doubler la responsabilité de notre service Message-Store. La principale raison est que ce service ne se contente pas que d'accueillir les demandes de lecture des chats, mais il vérifie également l'appartenance au chat en question.

Il lit de la même base de données où "Messaging store" stocke les messages.

- **Media service**

Ce microservice reçoit toutes les demandes de stockage des médias de messages dans un stockage d'objets, avec les contraintes imposées par le microservice "Messaging service", telles que la taille, le type et le chemin d'accès pour récupérer les media d'un message donné par exemple.

Le chemin indique le nom du fichier qui peut inclure l'identifiant du chat, l'identifiant de l'expéditeur et le nom du fichier. Ceci est particulièrement utilisé pour vérifier l'existence d'un autre objet portant le même nom et éviter de l'écraser.

Le mécanisme d'url signée a été utilisé une nouvelle fois pour permettre à l'utilisateur de télécharger directement vers le stockage d'objets sans passer par nos microservices et les surcharger. C'est aussi un moyen de restreindre le téléchargement aux seuls utilisateurs de notre plateforme, et de mettre en place des contrôles de fichiers en plus de cela.

Les métadonnées des médias sont stockées par le microservice “Messaging service”, vu que c’est lui qui garde et lie les pièces jointes aux messages. Ces métadonnées sont propagées après vers “Message store service” dans la queue.

- **API Gateway**

Le Gateway fonctionne comme un intermédiaire entre le front-end et les différents microservices de notre backend. Il achemine efficacement les requêtes vers les microservices appropriés et renvoie les réponses correspondantes. Tout d’abord, le gateway simplifie la complexité de la communication entre les différentes parties de notre application. Il permet de masquer les détails techniques des microservices individuels. Cela facilite la maintenance, car les développeurs du front end n’ont pas besoin de connaître les détails de chaque microservice, mais peuvent plutôt s’appuyer sur la gateway pour diriger les requêtes vers les services appropriés.

En ce qui concerne la sécurité, il protège les endpoints sensibles de nos microservices. Cette protection est essentielle pour prévenir un large éventail de menaces potentielles, notamment les attaques par déni de service (DDoS) et les tentatives d’intrusion.

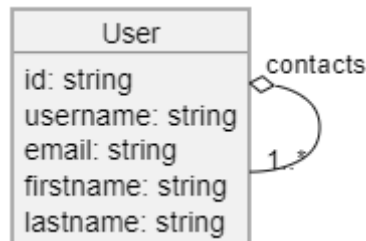
- **Job Scheduler**

Ce service n’a qu’un seul rôle: exécuter des tâches périodiquement en appelant les services appropriés. En effet, la suppression effective et permanente des médias et messages éphémères n’a lieu que périodiquement afin d’éviter une surcharge extrême de nos services.

De plus, ce Job Scheduler externe assure une gestion efficace des suppressions périodiques, évitant les doublons de tâches, même en cas d’exécution sur plusieurs instances.

Modèles des données

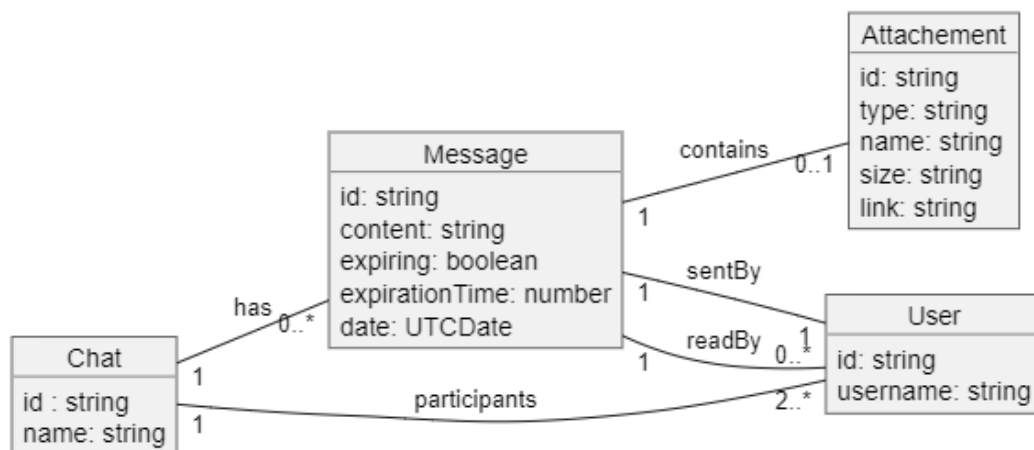
- **Users Service :**



La relation "User" représente un utilisateur et possède cinq attributs : un identifiant (id), un nom d'utilisateur (username), une adresse e-mail (email), un prénom (firstName) et un nom de famille (lastName).

Cette relation est associée à elle-même pour représenter les contacts de l'utilisateur. Cela se traduit par une deuxième table dans notre modèle de données physique, appelée "Contacts", qui contient l'identifiant de l'utilisateur et l'identifiant du contact. La clé primaire de cette table est une combinaison des deux.

- **Messaging service, Messaging Store Service and Messaging Reader Service :**



Les trois microservices partagent le même modèle de données qui concerne la fonctionnalité de messagerie.

La relation "Chat" représente une conversation et possède trois attributs : un identifiant (id), un nom (name).

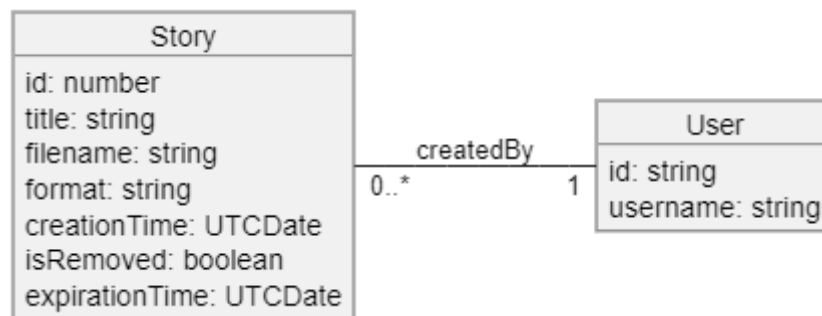
La relation "Message" quant à elle comporte cinq attributs : un identifiant (id), un contenu (string), un indicateur d'expiration (expiring), un temps d'expiration (expirationTime), une date d'envoi (date).

La relation "Attachment" représente une pièce jointe et possède cinq attributs : un identifiant (id), un type, un nom, une taille et un lien.

La relation "User" quant à elle comporte deux attributs : un identifiant (id) et un nom d'utilisateur (username). C'est un modèle partagé!

Un chat contient plusieurs messages et y participe plusieurs utilisateurs (au moins 2), les messages sont envoyés par un utilisateur mais vus par plusieurs utilisateurs (les participants). Les messages peuvent contenir une pièce jointe.

- **Stories Service :**



La relation story contient les attributs suivants : un identifiant "id" unique, un titre ou du nom de la story, le nom du fichier qui contient le contenu de la story, le format du fichier de la story , par exemple, JPG, PNG, MP4, etc. Cela indique le type de média associé à cette story. La date et l'heure de création de la story, un indicateur isRemoved qui indique si la story a été supprimée ou non. Si sa valeur est "true", cela signifie que la story n'est plus visible/accessible. Cela représente la date et l'heure à laquelle la story expirera. Une story est publié par un user, un user peut publier plusieurs stories.

Diagramme de déploiement

Le lien vers la version SVG plus claire : [Lien SVG](#) (.svg à télécharger et ouvrir localement)

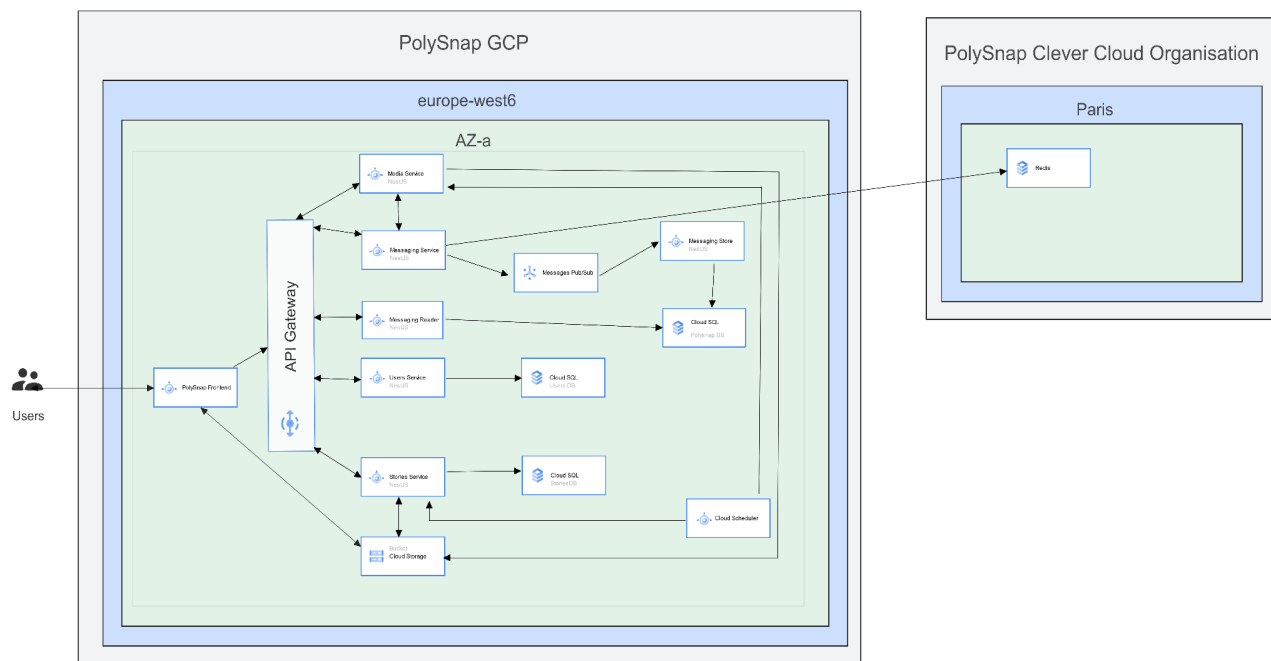


Figure 2 : Diagramme de déploiement de PolySnap

Nous avons choisi d'utiliser une plateforme en tant que service (PaaS) sur Google Platform Service (GCP) pour le déploiement de notre architecture. Ce choix, en faveur d'un PaaS plutôt que d'un CaaS ou bien autres modèles de service, est motivé par l'indépendance environnementale de notre application; on n'a pas des dépendances particulières à inclure, éliminant ainsi le besoin d'un contrôle granulaire sur l'environnement de l'application. Toutes les spécifications sont intégrées dans notre gestionnaire de packages npm. La configuration est transmise via des variables d'environnement, et le fournisseur de cloud prend en charge plusieurs versions de Node.js, incluant la dernière version LTS (20).

De plus, nous avons estimé qu'il n'était pas nécessaire d'opter pour une plateforme en tant que service (FaaS) car nous devons être disponibles en permanence. Aucun de nos services ne connaît de temps d'arrêt significatif, c'est pourquoi nous avons préféré éviter le démarrage à froid du FaaS ainsi que son coût supplémentaire (paiement pour les 15 premières minutes).

Nous avons également utilisé clever cloud pour provisionner une base de données NoSQL redis. Nous voulions essayer un autre fournisseur de cloud pour son faible coût, en économisant des crédits sur GCP.

Enfin, nous avons choisi de déployer le système dans le centre de données europe-west6, qui est un centre de données GCP situé à Zurich, et on a déployé notre organisation clever cloud à Paris, afin d'éviter toute latence. Notre application est conçue pour être optimale pour les utilisateurs français, car elle est déployée nationalement.

Services utilisés dans le déploiement :

- **Google AppEngine**

Du point de vue de “plateforme en tant que service” (PaaS), Google App Engine est un choix judicieux. Avec les services tels que **la gestion du trafic, la journalisation centralisée et la surveillance**, Il simplifie la gestion de notre architecture de microservices, nous permettant de nous concentrer sur la livraison rapide de fonctionnalités.

- **Base de données Cloud SQL**

Cette base de données relationnelle est sollicitée en écriture ET en lecture. L'utilisation d'une base de données relationnelle pour stocker les messages présente deux avantages majeurs. Premièrement, cela permet d'archiver les conversations de manière organisée, en structurant les données. Deuxièmement, les transactions ACID garantissent la cohérence des données lors d'opérations simultanées de lecture et d'écriture. Cette approche assure une gestion fiable et précise des messages au sein de notre application.

- **Base de données Redis**

Le microservice de messagerie dédié à l'écriture exploite une base de données Redis. Redis excelle particulièrement dans les opérations d'écriture en raison de sa nature en mémoire vive. Cette caractéristique lui permet de stocker rapidement les données.

- **Pub/Sub**

L'acheminement des messages vers Pub/Sub avant leur enregistrement dans une base de données présente des avantages considérables. Tout d'abord, Google Pub/Sub est spécialement conçu pour gérer des volumes extrêmement élevés de messages, assurant ainsi que le système puisse s'adapter à une demande croissante sans compromettre les performances. De plus, dans un service de messagerie, il est impératif de garantir la sécurité et la fiabilité des données. En stockant les messages de manière durable dans Pub/Sub jusqu'à leur traitement, on assure une protection contre la perte de données, même en cas de défaillance des microservices.

- **Cloud Storage : Bucket**

Nous utilisons un bucket Google Cloud Platform (GCP) pour stocker nos contenus multimédias. Le choix du bucket se justifie par sa spécialisation dans le stockage efficace d'objets binaires comme images, vidéos, fichiers audio. Ces "blobs" (Binary Large Objects), ne requièrent pas de structuration complexe. Les buckets offrent une grande évolutivité pour gérer d'importants volumes de données non structurées. GCP propose également des services pour publier rapidement son contenu dans un CDN.

- **Cloud scheduler**

Ce job scheduler de GCP permet de planifier des tâches à intervalles réguliers ou à des heures spécifiques, en utilisant une syntaxe cron ou un horaire simple. Il interagit facilement avec le PaaS, App Engine, dans lequel nos services sont déployés pour exécuter les tâches programmées.

Les 12 facteurs

Nous avons appliqués les 12 facteurs de cloud à notre projet :

1. Codebase :

Nous avons utilisé git comme gestionnaire de code source et outil de gestion des versions. Github héberge notre base de code mono-répo.

2. Dependencies:

Vu que nous utilisons NodeJs comme technologies de nos microservices, les dépendances de notre application sont clairement spécifiées et gérées via npm pour assurer une exécution stable et prévisible.

3. Configuration :

Les configurations spécifiques à chaque microservice sont stockées dans des variables d'environnement (spécifiées dans le app.yaml de App Engine), et non directement dans le code.

4. Backing Services:

En cas de non disponibilité du service Cloud SQL de GCP, nous pouvons facilement basculer sur une base de données sur clever cloud par exemple en changeant simplement les credentials dans nos fichiers de configuration.

5.Build, Release, Run :

Nos processus de construction et de déploiement sont automatisés grâce à npm, facilitant ainsi des cycles de développement efficaces.

6. Processes:

Tous les processus en cours d'exécution de notre application (microservices) sont sans état. Cela permet de faciliter et de promouvoir la mise à l'échelle horizontale.

7. Port binding:

Chaque microservice expose ses API via un port spécifique et accepte les requêtes entrantes en HTTP, facilitant l'accès de l'extérieur à travers le Gateway.

8. Concurrency:

Notre système peut automatiquement ajuster ses capacités en fonction de la charge, en augmentant ou en réduisant les instances selon les besoins. On a choisi de faire un auto-scaling avec un maximum et un minimum d'instances définies.

9. Disposability :

Les services démarrent rapidement quand une nouvelle instance est créée grâce à la rapidité du démarrage de NestJS. Pour assurer un arrêté gracieux, tous les microservices interceptent les signaux SIGTERM et SIGKILL et se déconnectent de la base de données avant de s'arrêter.

10. Dev/Prod parity :

Nous avons conservé les mêmes dépendances dans les deux environnements et nous les avons déployées rapidement après avoir développé les fonctionnalités. Cela permet de réduire au minimum le décalage horaire et le décalage par rapport au personnel entre les deux déploiements.







11. Logs :

Notre application génère des logs pertinents à la réception de chaque requête (que cela soit au niveau des contrôleurs ou bien au niveau des fonctions des services eux même) pour faciliter la surveillance et le débogage, en utilisant les outils de journalisation offerts par la plateforme cloud.

Présentation de la supervision envisagée du système

Monitoring et Alerting :

Pour surveiller notre système, nous avons utilisé le service Monitoring fourni par Google Cloud. Les éléments principaux qu'on a décidé d'observer sont l'utilisation de mémoire et l'utilisation du CPU pour les différentes instances déployées. La figure suivante représente les alertes qu'on a mis en place dans notre projet.

Nom à afficher ↑	Type ?	Dernière modification par	Dernière modification le	Date de création	Activé
App engine down	Metrics	zoubairhamza60@gmail.com	30 octobre 2023	30 octobre 2023	 on
CPU UTILIZATION FOR APP ENGINES	Metrics	zoubairhamza60@gmail.com	30 octobre 2023	30 octobre 2023	 on
DATABASE CPU USAGE	Metrics	zoubairhamza60@gmail.com	30 octobre 2023	30 octobre 2023	 on
LOST MESSAGES IN PUB SUB	Metrics	zoubairhamza60@gmail.com	30 octobre 2023	30 octobre 2023	 on
Latence between SQL and microservice	Metrics	zoubairhamza60@gmail.com	30 octobre 2023	30 octobre 2023	 on
SQL DISK USAGE	Metrics	zoubairhamza60@gmail.com	30 octobre 2023	30 octobre 2023	 on

- CPU utilization for app engines : Cette alerte va nous permettre de mesurer l'utilisation du CPU de nos services app engines, cela va permettre de détecter les anomalies qui impactent les performances de l'application. Si l'utilisation du disque dépasse 50%, une notification est envoyée automatiquement par Email.
- Database CPU usage : Mesure de l'utilisation du CPU par les instances Postgres.
- Latence between SQL and microservice : Cette alerte est primordiale pour mesurer les latences entre les microservices et bases de données que ce soit lecture ou écriture
- Delay in pub/sub publish : Mesurer le temps de publication des messages dans les topics.
- SQL DISK USAGE: Observation du stockage de nos bases de données. Cela permet d'éviter les problèmes liés à un espace disque insuffisant, tels que des erreurs d'écriture.
- App engine down : Cette alerte est pour nous notifier en cas de panne d'un service app engine.

La figure suivante représente une alerte qu'on a reçu lors d'un cold start de notre service de messagerie.

Cloud Pub/Sub Topic - Publish request latency

Publish request latency for cloud-398911 message_queue with metric labels {has_ordering=false, response_code=success, schema_type=avro} is above the threshold of 400000.000 with a value of 597907.956.

Summary

Start time

Oct 16, 2023 at 5:24PM UTC (less than 1 sec ago)

Project

[cloud-398911](#)

Policy

[delay in pub/sub publish](#)

Condition

Cloud Pub/Sub Topic - Publish request latency

Metric

pubsub.googleapis.com/topic/send_request_latencies

Threshold

above 400000

Observed

597907.956

Le seuil définie pour les latences de la queue de message est 400 ms, lors du cold start le système a mesuré presque 600 ms lors de l'envoi du message.



Coûts prévisionnels

- **Google Cloud Provider :**



Dans nos déploiement nous avons utilisé les ressources App Engine, API Gateway, Cloud storage, Cloud Scheduler, une base de donnée de type PostgreSQL (Cloud Sql) et un queue de message Pub/Sub.

1) App Engine (Paas) :

Nous avons utilisé pour les différents services des App Engine de type F1 avec 256 MO de RAM et 600 MHZ de CPU. Cette version propose une scalabilité automatique selon la charge des utilisateurs. En utilisant le simulateur Google Cloud, on a calculé les coûts suivants :

App Engine standard environment instances	
Zurich	 
Instance Type: F1	
Instance Hours: 4,380 per month	
EUR 216.97	
Total Estimated Cost: EUR 216.97 per 1 month	
Estimate Currency	
EUR - Euro	

2) Cloud Storage :

Cloud Storage	
1x Standard Storage	 
Location: Zurich	
Total Amount of Storage: 10 GiB	EUR 0.24
Always Free usage included: No	
EUR 0.24	

En passant le système à grande échelle, nous devons augmenter la taille du stockage pour supporter une charge importante des utilisateurs.

3) API Gateway :

La facturation pour l'API Gateway se calcule à partir du nombre d'appels effectués.

Nombre d'appels d'API par mois et par compte de facturation	Coût par million d'appels d'API
Entre 0 et 2 millions	0,00 \$
Entre 2 millions et 1 milliard	3,00 \$
Plus de 1 milliard	1,50 \$



Nous estimons avoir entre 0 et 2 millions d'appels par mois, alors ce service sera gratuit dans notre système.

4) Base de données Cloud SQL (PostgreSQL) :

Nous utilisons une seule instance PostgreSQL avec les caractéristiques suivantes :

- vCPU : 2
- Mémoire : 8 Go
- Stockage SSD : 10 G

A l'aide du simulateur, nous avons trouvé les tarifs suivants :

Cloud SQL for PostgreSQL	
DB-STANDARD-1	 
Edition: Enterprise	
Number of instances: 1	
Location: Paris	
Total hours per month: 730.0	
Instance type: db-standard-1	EUR 54.06
SSD Storage: 10.0 GiB	EUR 1.87
Backup: 0.0 GiB	EUR 0.00
EUR 55.92	
Total Estimated Cost: EUR 272.89 per 1 month	
Estimate Currency	
EUR - Euro	

5) Google Cloud Scheduler :

Nous avons utilisé la ressource Google Cloud Scheduler pour effectuer la suppression des messages et des stories expirés depuis la base de données. En total, nous avons 2 tâches exécutées quotidiennement.

Cloud Scheduler	
Total amount of jobs: 2	
EUR 0.00	

6) Pub / Sub :

Le coût est calculé à partir du débit des messages qui passent par la queue de messages. Chaque mois, les 10 premiers Gio de débit sont gratuits. Ensuite, le tarif est de 40 euro par Tio. Pour une première utilisation, on suppose que ce service est gratuit mais le coût va augmenter selon la demande des utilisateurs.

Pub/Sub	
Message delivery type: Basic	
Volume: 581 Bi	EUR 0.00
EUR 0.00	
Total Estimated Cost: EUR 0.00 per 1 month	
Estimate Currency	
EUR - Euro	

- **Clever Cloud** :

Nous avons déployé une base de données Redis sur Clever Cloud pour faire un backup de messages en cas de panne du système Pub / Sub.

Redis – L ^

vCPUs : 1, Taille BDD max : 512 Mio,
Limite de connexions : 500, Bases de données : 10

19,00 € 1 19,00 €

19,00 € HT
estimé/30 jours

Temporalité

Prix/30 jours ▼

Devise

€ EUR ▼

- **Conclusion :**

Ainsi, le coût mensuel total pour tous nos déploiement sur les providers Google Cloud et Clever Cloud est égal à **292,13 €** par mois. Ce qui donne un coût annuel de **3.505,56 €**. Ces tarifs vont surement changer si on décide de changer la zone de déploiement vers un autre DataCenter ou bien si on augmente la capacité d'un déploiement spécifique, surtout pour les bases de données afin d'avoir un système à haute disponibilité.

Prise de recul

- **Forces :**

Notre architecture, fondée sur des microservices stateless et cloud native, offre une montée en charge élastique grâce à l'auto-scal. Nous avons fait le choix de la technologie NestJS pour sa capacité à démarrer rapidement et à effectuer une fermeture des connexions à la base de données en douceur.

Nous avons veillé à garantir une indépendance significative vis-à-vis du fournisseur de cloud pour éviter le vendor lock-in. Cependant, une exception à cette approche se présente au niveau du bucket, en raison de son incompatibilité avec les normes S3, bien qu'il soit facile à utiliser..

De plus, notre architecture repose sur des principes solides de conception, notamment l'adoption du modèle CQRS (Command Query Responsibility Segregation), ce qui renforce la cohérence de notre système. Ce choix permet de séparer les opérations de lecture et d'écriture, améliorant ainsi la gestion des requêtes et des mises à jour.

En outre, nous traitons les messages de manière asynchrone et non bloquante, ce qui permet d'optimiser les performances et de garantir une expérience fluide aux utilisateurs, même en cas de charges de travail importantes.

Nous avons également mis en place une configuration d'auto-scaling. Cette stratégie a été calibrée pour maintenir une utilisation cible du CPU à un niveau optimal de 70%. Cela signifie que le système est capable d'ajuster automatiquement le nombre d'instances en fonction de la charge, assurant ainsi des performances constantes et une utilisation efficiente des ressources. Nous avons défini un seuil minimal d'une instance pour garantir une disponibilité ininterrompue et éviter des "cold starts", tandis que le seuil maximal est fixé à dix instances pour éviter une surallocation inutile de ressources. Cette configuration garantit une expérience fluide et fiable pour nos utilisateurs, même en période de demande fluctuante.

Nous avons fait un test de charge avec postman en simulant l'envoi de messages de 100 utilisateurs au même temps (nombre d'utilisateurs virtuels maximal que postman supporte):

Run order

Deselect All

Select All

Reset

☐ POST create users

☐ PATCH add contact

☐ POST createstory

☐ PUT upload

☐ POST save story

☐ GET get contact stories

☒ POST message 1

☒ POST message 2

Functional

Performance

Test how your APIs perform under load

Simulate real-world traffic from your local machine and observe the performance of your APIs. Learn more about [performance testing](#) »

Set up your performance test

Virtual users ①

100

Test duration

5 mins

Load profile ①

☒ Fixed

☐ Ramp up

Data file ①

FEATURE TRIAL

Select file

Run

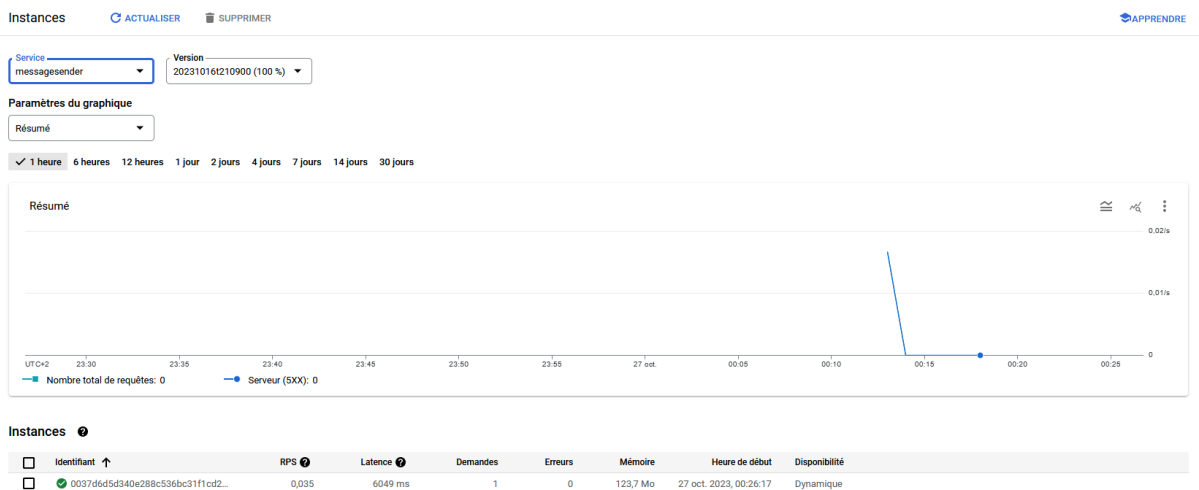
100 VUs

0 5 mins

Preview

Simulate 100 virtual users running the collection in parallel for 5 minutes.

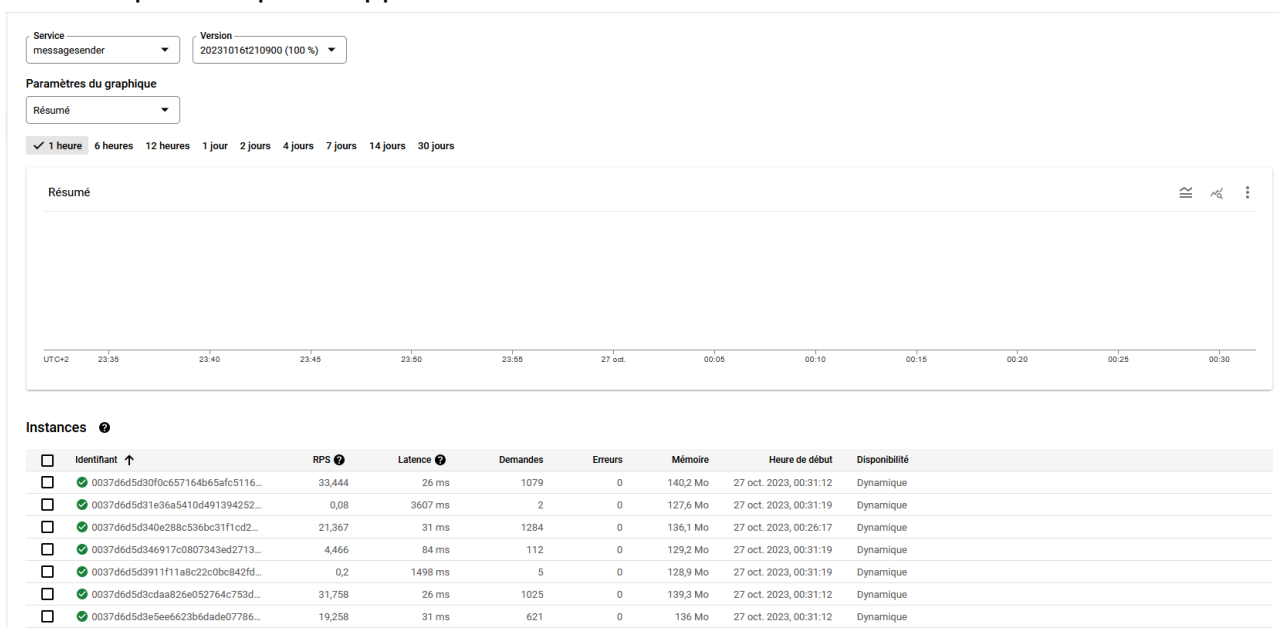
Nous pouvons observer avant l'exécution du test que le service message sender a une seule instance qui tourne :



Instances ①

<input type="checkbox"/>	Identifiant ↑	RPS ②	Latence ②	Demandes	Erreurs	Mémoire	Heure de début	Disponibilité
<input type="checkbox"/>	0037d6d5d340e288c536bc31f1cd2...	0,035	6049 ms	1	0	123,7 Mo	27 oct. 2023, 00:26:17	Dynamique

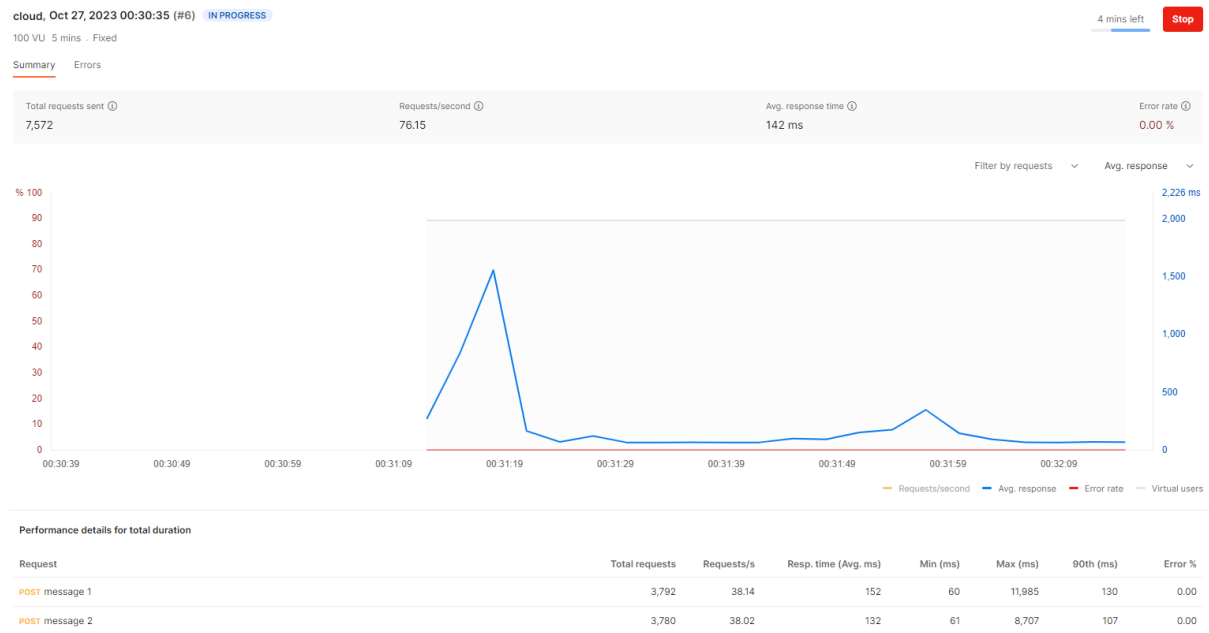
Après quelques secondes du lancement du test, des nouvelles instances se créent automatiquement pour supporter le flux de trafic montant :



Instances ②

<input type="checkbox"/>	Identifiant ↑	RPS ②	Latence ②	Demandes	Erreurs	Mémoire	Heure de début	Disponibilité
<input type="checkbox"/>	0037d6d5d30f0c657164b65afc5116...	33,444	26 ms	1079	0	140,2 Mo	27 oct. 2023, 00:31:12	Dynamique
<input type="checkbox"/>	0037d6d5d31e36a5410d491394252...	0,08	3607 ms	2	0	127,6 Mo	27 oct. 2023, 00:31:19	Dynamique
<input type="checkbox"/>	0037d6d5d340e288c536bc31f1cd2...	21,367	31 ms	1284	0	136,1 Mo	27 oct. 2023, 00:26:17	Dynamique
<input type="checkbox"/>	0037d6d5d346917c0807343ed2713...	4,466	84 ms	112	0	129,2 Mo	27 oct. 2023, 00:31:19	Dynamique
<input type="checkbox"/>	0037d6d5d3911f11a8c22c0bc842fd...	0,2	1498 ms	5	0	128,9 Mo	27 oct. 2023, 00:31:19	Dynamique
<input type="checkbox"/>	0037d6d5d3cd3aa82e052764c753d...	31,758	26 ms	1025	0	139,3 Mo	27 oct. 2023, 00:31:12	Dynamique
<input type="checkbox"/>	0037d6d5d3e5ee6623b6dadae07786...	19,258	31 ms	621	0	136 Mo	27 oct. 2023, 00:31:12	Dynamique

Sur postman, nous pouvons remarquer une augmentation soudaine du temps de réponse, suivie d'une diminution lorsque le redimensionnement horizontal est effectué:



- **Faiblesses**

Dans l'optique de nos dépendances, une légère dépendance au provider (vendor lock-in) persiste en ce qui concerne notre utilisation de cloud storage pour le stockage des stories et médias.

Nous utilisons le Cloud storage de GCP, dont l'API est très simple à utiliser, mais qui est incompatible avec la norme S3. En cas de migration de notre application vers AWS, nous serons amenés à effectuer des modifications au niveau du code pour qu'elle fonctionne dans la nouvelle plateforme.