

Rapport de projet

Polyticket Cloud Computing

Equipe E

Sommaire

Sommaire	2
Introduction	3
Analyse du besoin	3
Définition du périmètre MVP	4
Diagramme d'architecture logicielle	5
Modèle de données	8
Diagramme de déploiement cloud	9
1) Partie événements	9
2) Partie tickets	10
Coûts prévisionnels	12
1) Test de la FaaS event-listing	12
2) Test de la PaaS ticket-booking	17
Supervision envisagée du système	20

Introduction

Ce rapport présente les étapes de conception de notre solution au sujet proposé. Va suivre une présentation de ces différentes étapes en commençant par l'analyse du cahier des charges fixés jusqu'à la présentation des solutions de supervisions possibles et envisagées en passant par les différentes architectures.

Ainsi dans le processus de conception d'une solution logicielle, la première étape consiste en l'analyse du besoin. Voici les points importants que nous avons identifiés dans le sujet.

Analyse du besoin

- Le client souhaite un système de vente et revente de tickets pour des concerts et événements sportifs
- Le client souhaite avoir un espace dédié à ses partenaires organisateurs, leur permettant de programmer des événements.
- Le client souhaite mettre en place un système de réservation temporaire de billets lors de l'achat et de remise en vente si le temps est dépassé.
- Pour répondre aux besoins sanitaires en vigueur maintenant, le client veut mettre en place un système d'e-ticket pour les événements proposés sur la plateforme. Les e-tickets sont proposés à deux conditions: soit l'organisateur souhaite en proposer, soit la plateforme en propose automatiquement si le nombre de places physiques disponibles est atteint.
- Le client veut aussi pouvoir, pour le compte d'autres plateformes, remettre en vente des billets reçus par quota quotidien.
- Le client souhaite avoir un espace permettant de mettre à disposition les rediffusions des événements proposant ce format.
- En plus de la mise à disposition des rediffusions des événements le proposant, le client veut qu'il soit possible de visionner en direct ces événements, sous condition d'avoir un e-ticket accordant l'accès.

Dans la suite logique de l'identification des points demandés par les clients, il a fallu déterminer notre MVP, lister les points que nous avons prévu de réaliser. Les points identifiés que nous ne réaliserons pas seront aussi discutés.

Définition du périmètre MVP

1. Un organisateur peut créer un événement en y renseignant des informations comme la date, l'artiste, le nombre de places disponibles, ainsi que des informations sur l'organisation et si l'événement propose des e-tickets.
2. Une fois un événement fini, un organisateur peut accéder aux détails de l'événement et y ajouter la vidéo..
3. Un utilisateur peut voir la liste des événements sur la plateforme.
4. Un utilisateur peut acheter un billet pour un événement.
5. Un utilisateur peut ajouter un billet à son panier et, grâce à la mise en place d'un panier temporaire, il est assuré que son billet est réservé pendant 5 minutes (temps durant lequel il peut payer).
6. Un utilisateur ayant acheté un billet ou e-billet peut accéder à la VOD de l'événement.
7. Les utilisateurs peuvent accéder aux informations des tickets qu'ils ont achetés. Grâce à l'ID de commande généré au moment de l'achat, ils peuvent accéder, par le site, à leur ticket afin de passer un contrôle de ticket lors d'un événement physique.
8. Lorsqu'un utilisateur accède à la VOD, le lien présenté est un lien unique partagé avec tous les autres utilisateurs et il n'est valable que quelques heures.
9. Un organisateur d'événement peut voir combien de tickets ont été vendus.
10. La plateforme ne requiert pas de connexion à un compte afin de passer une commande, ainsi aucune information personnelle n'est conservée.

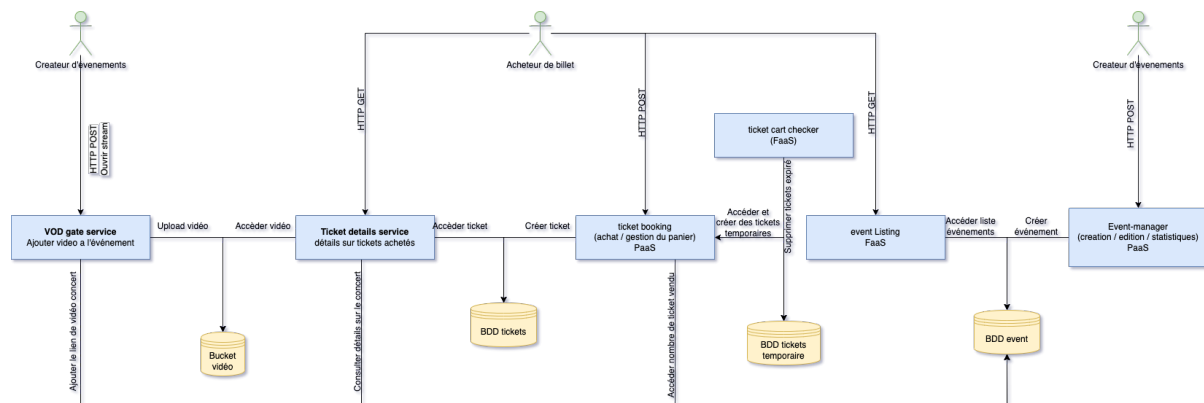
Malheureusement, comme il ne s'agit que d'un MVP certains points n'ont pas pu être inclus. Par exemple, après accord avec le client, il a été décidé que seulement un des deux services de streaming vidéo envisagés serait mis en place, et il s'agit ici du service de VOD, ce qui laisse de côté le service de diffusion en direct d'un événement.

Aussi, à l'heure actuelle, il n'est pas possible de supprimer un événement une fois qu'il est créé, et il n'est pas non plus possible de changer le nombre de places disponibles, nous avons conscience que c'est un certain handicap.

Enfin, le système de paiement, qui se fait par un intermédiaire extérieur, n'a pas été mocké dans le cadre de ce projet.

Une fois que les fonctionnalités de la version minimale du produit furent établies, nous devons passer à la partie architecture afin de prévoir comment nous allons procéder pour implémenter le logiciel. Dans un premier temps, voici le diagramme d'architecture logicielle.

Diagramme d'architecture logicielle



Pour l'architecture, nous avons opté pour une architecture orientée service et non pas un monolithe. Ce choix a été simple puisque les différentes fonctionnalités de l'application n'engendrent pas toutes le même trafic et il y a donc une nécessité d'isoler ces bouts afin de ne pas compromettre le reste des services.

Donc la liste des différents services est la suivante:

- Event manager
- Event listing
- Ticket booking
- Ticket cart checker
- Ticket details
- VOD gate

Dans cette liste, on peut identifier deux groupes de services. Les services de management, destinés aux créateurs d'événements, qui viennent ajouter les leurs sur la plateforme, ce sont 'Event manager' et 'VOD gate'. Le second groupe est celui des services destinés aux consommateurs, les clients qui viennent acheter leurs billets sur la plateforme. Ce groupe se compose de tous les autres services.

En plus de ces services, le logiciel utilise 4 systèmes de stockage:

- Le stockage des événements
- Le stockage pour les tickets en cours d'achat
- Le stockage des tickets achetés.
- Le stockage pour les vidéos à la demande.

Le découpage de l'architecture vers ce modèle a été guidé par les fonctionnalités principales que la plateforme doit avoir. Ainsi chaque service correspond à peu de chose près à un point du périmètre du MVP.

- Event manager est l'interface entre le système et les organisateurs d'événements afin qu'ils puissent ajouter leurs événements sur la plateforme, répondant donc au point numéro 1 et 9 du MVP.
- Event listing est tout simplement le composant qui gère l'affichage de tous les événements que la plateforme recense. Ce service nous permet de respecter le point 3.

- Le service Ticket booking nous permet lui de satisfaire le point 4. Il s'agit du service de 'boutique' de la plateforme où les clients vont acheter les billets pour les différents événements.
- Ticket cart checker est lui aussi un service dont la mission est basique. C'est à travers lui que le système de réservation temporaire d'un billet est fait. Il valide le point 5.
- Ticket detail quant à lui assure la mise à disposition pour les clients des billets qu'ils ont achetés afin de pouvoir rentrer dans un événement ou pour accéder au service à la demande de vidéos et donc nous permet de remplir les points 6 et 7.
- Enfin le service VOD gate va permettre à un organisateur d'uploader la vidéo d'un événement afin qu'elle soit disponible dans le service de vidéos à la demande, ainsi validant les points 2 et 6.

Maintenant que l'intégralité des services ont été brièvement présentés ainsi que leur rôle, voici une présentation un peu plus approfondie de chaque service:

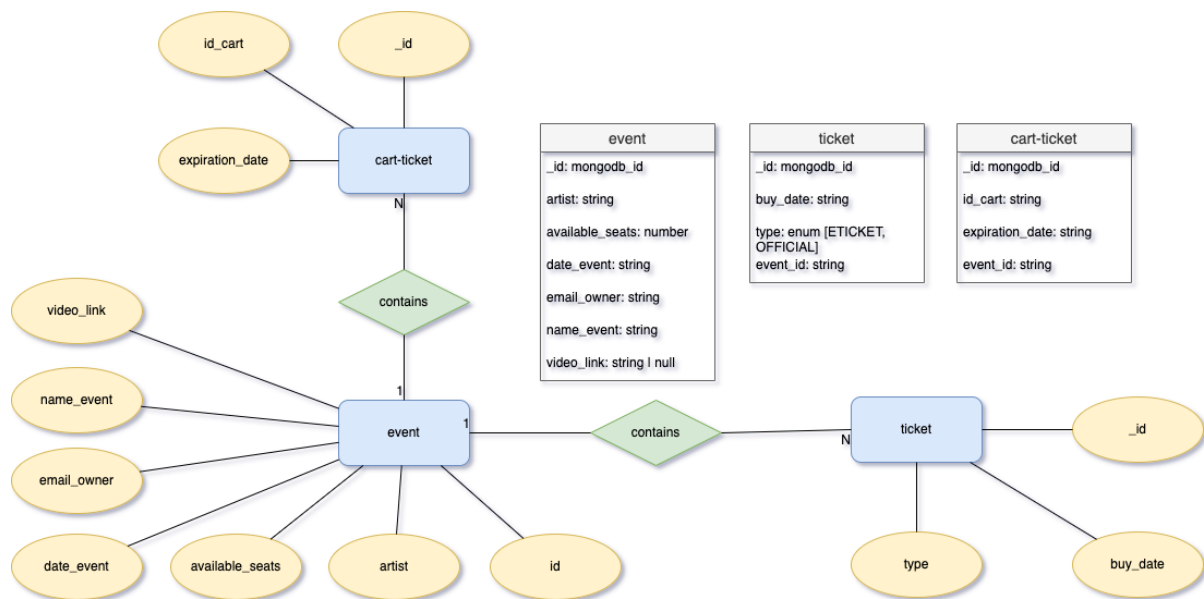
- Event manager:
 - Joue le rôle d'interface entre le système et les organisateurs pour la mise en vente de places pour un événement
 - Il peut également fournir des statistiques (nombre de place vendues) à l'organisateur
 - Quand un événement est créé, il écrit en BD dans la base des événements.
- Event listing:
 - Rôle très simple
 - Lit uniquement les infos contenues dans la base des événements pour les mettre à disposition des utilisateurs.
 - Redirige les utilisateurs quand ils veulent accéder à un événement
- Ticket booking:
 - Permet aux utilisateurs d'acheter des billets
 - Supporte l'achat de billets pour accès physique et dématérialisé.
 - Utilise un système de file d'attente à temps limité.
 - Il crée un ticket temporaire dans la base des tickets temporaires quand un client veut acheter un billet
 - N'utilise pas de système de compte
 - Une fois le billet acheté, il le crée dans la base des billets achetés.
 - Renseigne sur le nombre de places disponibles
- Ticket cart checker:
 - Met en place le système de file d'attente
 - Temps limite paramétrable (5 minutes dans le cas réel d'utilisation)
 - Agit directement sur la base des tickets temporaires

- Ticket détail:
 - Rôle très simple
 - Permet aux utilisateurs de consulter leur billet
 - Demande l'id de commande pour accéder au billet
 - Lit directement les infos dans la base des tickets achetés
 - Affiche des informations sur l'événement concerné par le billet demandé
 - Interface d'accès aux VOD selon les billets du client
- VOD gate:
 - Système gérant l'ajout d'une vidéo à un événement
 - Séparé du service d'event manager
 - Stockage des vidéos dans un système propre à l'hébergeur cloud et non dans la base des événements.

Une fois l'architecture définie, il nous a fallu réfléchir aux technologies que nous voulions utiliser car tous les composants ont leur lot de contraintes, surtout, et principalement, les systèmes de stockage.

- Pour la globalité de l'architecture, nous voulions faire une architecture en service, principalement accessible depuis le web. Un certain nombre de technologies sont disponibles pour répondre à ces besoins, nous avons décidé de choisir NestJs pour réaliser le backend car nous maîtrisons déjà la technologie, elle est plus adaptée à ce type d'architecture que NodeJs, elle se base sur le Typescript, qui est une version plus sécurisée du Javascript en terme de programmation. Bien qu'il ne soit pas toujours le plus simple, le gestionnaire de dépendances npm marche dans la globalité plutôt bien et offre une bonne gestion.
- Le fournisseur d'hébergement cloud est Google Cloud Platform par choix du client.
- La base de données des événements est une base SQL, car elle n'aura que très peu de charge d'écriture (création d'événement par l'organisateur) mais aura en revanche une bonne charge de lecture car elle est accédée pour la mise en avant des événements ainsi que lors de l'achat de billets ou quand les clients consultent les billets qu'ils ont acheté.
- La base de données des tickets temporaires, comme celle des tickets achetés, sont des bases NoSQL. Ce choix a été fait car les deux doivent accepter des gros débits d'écriture et de lecture et une majeure partie du temps probablement en simultané. La technologie adoptée pour ces bases est Mongo Atlas, ce qui engendre un peu de vendor lockin car la base n'étant gérée par nous directement, il pourrait y avoir des soucis de compatibilités si aucune modification n'est faite dans le cas d'un changement d'hébergeur cloud. Par soucis de temps et pour faciliter la gestion, les deux bases sont représentées par 2 collections différentes dans une seule et même base de données ce qui n'est pas une bonne pratique.
- Enfin le système de stockage des VOD utilise lui une technologie propre à GCP et sera donc détaillé dans le déploiement cloud, néanmoins l'architecture souffre sur ce point de vendor lockin.

Modèle de données



Ce diagramme de données représente la corrélation globale des données sur tout le projet. Nous avons séparé le projet en 3 entités (chacune définie par une database ou une collection MongoDB). Un event est un événement au sens large du terme, il peut être un concert, un événement sportif ou caritatif, il est très large. A l'intérieur, on y trouve le nom de l'événement, la date, le nombre de places disponibles, l'artiste, l'email du créateur, et aussi le lien vidéo pour la VOD. Ce lien vidéo reste null tant que le concert n'est pas encore passé, par contre, quand l'organisateur de l'événement upload la vidéo grâce à notre service, le lien de la vidéo est automatiquement mis à jour, et rendu disponible pour les utilisateurs qui ont acheté un ticket ou un e-ticket.

Un event peut contenir plusieurs entités ticket, mais chaque ticket n'est relié qu'à un seul event. Un ticket est acheté par des utilisateurs qui veulent aller à l'événement, il peut être de type ETICKET (pour uniquement accéder à la VOD), ou il peut être de type OFFICIAL, qui est le ticket qui lui permettra d'accéder à la VOD mais aussi au concert physique. On insère aussi à ce ticket la date d'achat. Un utilisateur pourra accéder à son ticket grâce à son ID qu'il aura préalablement enregistré lors de l'achat de ce ticket.

Un event peut contenir aussi plusieurs cart-ticket mais un cart-ticket n'est associé qu'à un seul event. Un cart-ticket est un panier d'achat, il permet de gérer la file d'attente afin de ne permettre qu'à N utilisateurs d'acheter un ticket. Il possède un id_card mais aussi une expiration_date, un utilisateur qui crée un panier a jusqu'à l'expiration_date pour acheter son billet, s'il dépasse cette expiration_date, le ticket est remis en vente.

Diagramme de déploiement cloud

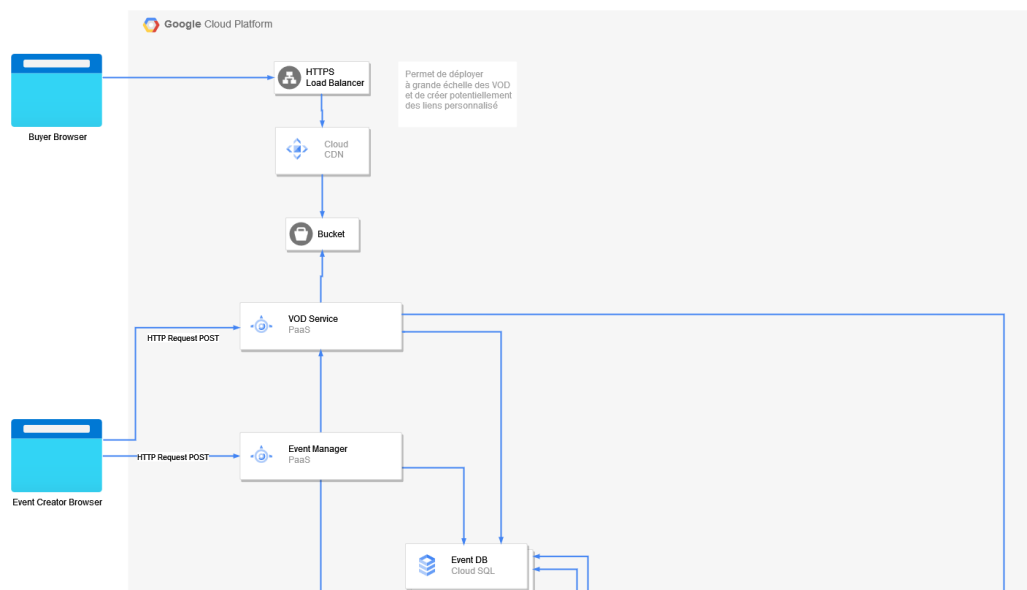
Cette partie traite du processus de déploiement de la plateforme, des choix de méthodes qui ont été fait pour y parvenir. Elle contient également des justifications quant à

la résilience, les points critiques, notamment en rapport avec la charge, et ce qui pourrait être modifié.

Enfin nous aborderons la dépendance de notre solution vis-à-vis de l'hébergeur cloud.

Le diagramme de déploiement va être traité en deux temps, d'abord la partie concernant les événements et ensuite la partie concernant les billets. Puis une vue d'ensemble sera prise sur le projet.

1) Partie événements



Nous pouvons observer ici la partie concernant les événements du diagramme de déploiement.

Tout d'abord, le service Event manager a été déployé selon le modèle PaaS. Ce service aurait pu être déployé en tant que FaaS pour le moment car la logique métier est simple, il n'a pas besoin d'être actif tout le temps. Cependant on l'a tout de même mis en PaaS car nous avons pensé qu'à l'avenir ce service pourrait être en charge d'un grand nombre de fonctionnalités, qui viendraient fortement complexifier la logique métier, et bien que si le trafic sur ce composant ne serait toujours pas permanent, sa densité demanderait un contrôle de l'environnement, qui est inexistant pour une FaaS. C'est aussi un moyen pour éviter d'être trop dépendant de notre hébergeur car ces mécanismes de serverless sont très vendor lock-in.

Pour ce qui est du service de vidéos à la demande, la VOD gate (VOD Service sur le diagramme), a été également déployé sur une PaaS pour pouvoir profiter d'un certain contrôle sur le service sans avoir à gérer l'environnement comme ce serait le cas sur un CaaS, mais aussi pour ne pas se retrouver totalement contraint comme ce serait le cas pour une FaaS.

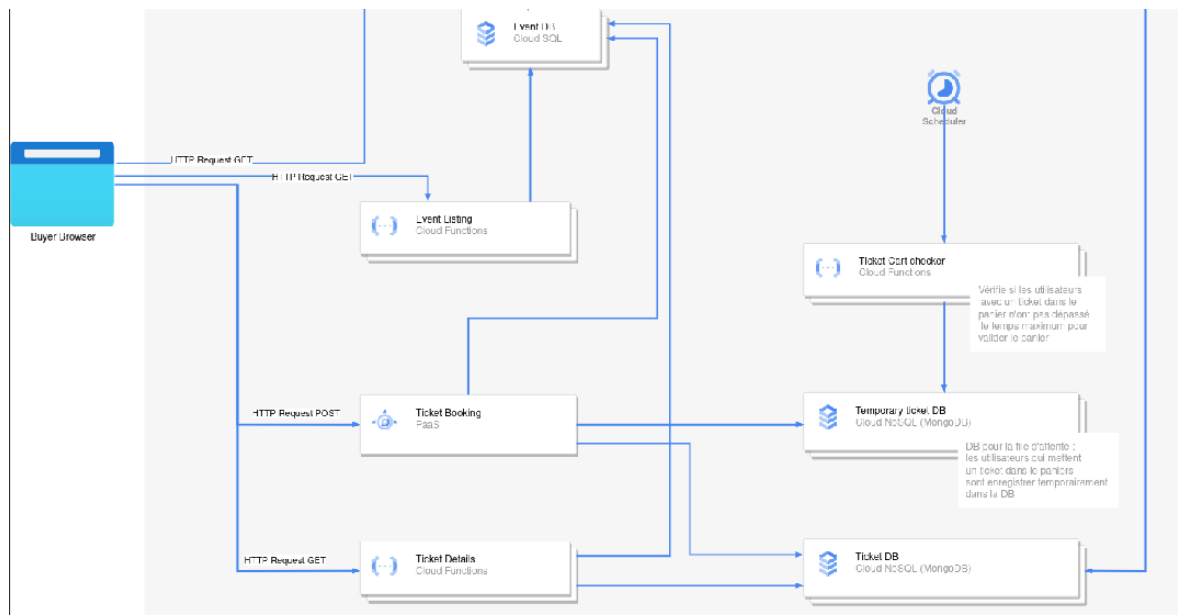
Comme évoqué dans la partie sur l'architecture du système, ces deux services communiquent avec la base de données des événements qui est une base SQL.

Ensuite au-dessus de ces services, nous pouvons voir le système de vidéo à la demande. Il se compose dans un premier temps d'un load balancer, derrière celui-ci se trouve un CDN et enfin on retrouve le système de stockage, le bucket. Ce dernier est simplement un système de stockage propre à GCP. Pour ce qui est du CDN, il s'agit d'un système de cache partagé entre serveur qui permet de mettre en cache des ressources statiques qui sont très demandées, les vidéos ici. Enfin, concernant le load balancer, il a deux fonctions. Tout d'abord sa fonction principale, qui va être de répartir la charge sur les différents serveurs du CDN afin de maintenir le meilleur équilibre possible, puis à travers un service spécial de GCP, l'url signed, il va permettre la génération de liens temporaires ce qui est exactement ce que l'on veut dans le cadre du service de vidéo à la demande.

En termes de point critique, dans le cas présent le load balancer, comme à son habitude, représente un single point of failure. Une solution pour pallier à ça est de mettre en place un réseau de load balancer, mais cela engendre plus de complexité et demande aussi beaucoup plus de gestion, et enfin potentiellement un surcoût si le système est peu chargé.

Pour ce qui est de la dépendance, ici notre système est relativement dépendant de l'hébergeur cloud que l'on utilise. Les seuls mécanismes propriétaires sont le bucket et le système d'url. En terme de quantité cela représente peu mais en termes de fonctionnalité cela représente le cœur du système de VOD donc changer d'hébergeur cloud pourrait engendrer potentiellement un certain coût..

2) Partie tickets



Dans un premier temps, parlons des services. Event listing a été déployé sur une cloud function, qui est la solution FaaS de GCP. Ce choix s'explique par le fait qu'une FaaS est utile dans le cas d'un métier simple et répétitif. C'est exactement le cas du métier

d'évent-listing qui donne simplement aux utilisateurs les infos des différents événements qui existent dans la base de données associée. En revanche, un des points importants de la FaaS qui est le cold start ne correspond pas totalement avec le composant. Ticket details a été déployé sur une FaaS pour les mêmes raisons mais dans son cas, le cold start n'est pas dérangeant car les utilisateurs consulteront les détails de leurs tickets uniquement (normalement) quand il y a des concerts et il n'y a pas des concerts en permanence, alors que, la page principale d'un site de vente de ticket d'événement est le listing des concerts vendus disponible sur la plateforme et cette page là est susceptible d'être demandée n'importe quand donc un passage de FaaS vers PaaS serait à étudier pour voir si le temps perdu au démarrage du composant est une cause de perte d'argent.

Le troisième composant que nous avons déployé sur une cloud function est le ticket cart checker. Il s'agit du service qui va avoir pour rôle de contrôler à intervalle régulier que les requêtes qui sont dans la file d'attente sont toujours valides. Si elles ne le sont pas, elles sont supprimées. Pour se faire, nous utilisons un composant natif de GCP qui est un cloud scheduler qui nous permet d'exécuter une tâche à une fréquence que l'on peut définir nous même. Ainsi, les caractéristiques de fonctionnement sont les mêmes que ce qui est adapté à un composant FaaS d'où le choix de le déployer ainsi. Comme évoqué dans la partie architecture, la base de données qui nous permet de stocker les requêtes d'achat, ainsi que celle qui contient les tickets achetés, sont toutes deux des bases NoSQL MongoDB qui sont gérées par Atlas.

Enfin, il reste le service de ticket booking. Celui-ci n'a pas été déployé sur une FaaS car le système ne correspond pas du tout, donc il a été déployé sur une PaaS ce qui est suffisant étant donné que nous n'avons pas besoin d'avoir un contrôle poussé sur l'environnement de déploiement ou sur le système en particulier.

Dans cette partie du modèle de déploiement nous avons beaucoup utilisé de cloud functions qui est le modèle de déploiement qui est le plus contraignant en termes de vendor lockin, nous sommes donc très dépendant ici. Si un changement venait à être fait disons vers AWS, qui pour implémenter les FaaS utilise un système nommé Lambdas, il y a peu de chances que l'implémentation sur GCP soit parfaitement compatible avec l'implémentation nécessaire chez AWS, donc une certaine dose de travail sera nécessaire dans cette partie du déploiement. En termes de résilience, notre point faible sont les bases de données. Tout notre système repose dessus, et pour le moment aucun système de réplica n'a été mis en place, donc si par exemple tout le système de gestion des tickets tombent sous la charge et que l'on regarde ce qu'il se passe du côté de la partie événement, les gens qui sont actuellement en train de regarder des VOD pourront continuer, mais en revanche, plus personne ne pourra venir et demander à regarder une vidéo et si on considère que la base de données des événements tombe avec le reste alors plus aucun événement ne peut être créé et plus personne ne peut consulter les événements prévus non plus. Ceci est un des gros, si ce n'est le plus gros problème de notre modèle de déploiement.

Coûts prévisionnels

Nous sommes partis du postulat qu'il était nécessaire d'injecter de la charge dans nos différents services afin de savoir quelle charge ils sont capables de tenir. Nous ne pouvons pas faire de généralité, par exemple dire "si le service X est capable de gérer une charge Y, alors le service Z est aussi capable de gérer la même charge Y, si il a les mêmes caractéristiques hardware", pour la simple et bonne raison que chaque service exécute un code différent, et ils ont un métier différent, il faut donc les tester un par un pour avoir une approximation de ce que l'on est capable de gérer comme charge. Cependant, pour une raison de temps disponible, nous ne pouvons pas tester tous les services. Nous avons donc fait le choix de tester les services les plus importants, et même si l'on ne peut pas forcément appliquer des conclusions globales suite aux tests d'une FaaS, nous pouvons tout de même avoir un ordre de grandeur global de comment scaler les autres services.

Afin de réaliser ces tests, nous avons fait le choix d'utiliser Gatling comme outil d'injection de charges. Nous avons longuement hésité à utiliser Jmeter, car un membre du groupe avait de l'expérience sur cet outil, mais, nous nous sommes finalement décidé sur Gatling, dans sa version open-source (géré en scala avec Maven, accessible via le repository git) pour 2 raisons. La première est que, aujourd'hui dans l'industrie, nous serons plus amenés à utiliser Gatling que Jmeter, car c'est un outil plus récent ainsi l'apprendre aujourd'hui est donc plus intéressant pour nos futures expériences. La deuxième raison est que , Jmeter pour faire des requêtes simulant énormément de users, utilise les threads tandis que Gatling, grâce à l'utilisation de Akka, permet de mieux gérer ces grosses charges, et permet donc à tout le monde de faire des injections de grosse envergure, sans avoir un ordinateur très puissant.

Dans la mesure, où, dans notre situation, nos bottlenecks se situent au niveau des bases de données (qui sont centralisées), nous avons fait le choix, pour le test de nos PaaS et FaaS, d'augmenter le plus possible leurs capacités, afin d'essayer au maximum de surcharger le service plutôt que la base de donnée.

1) Test de la FaaS event-listing

Ce micro-service implémente la récupération des événements disponibles afin que notre front-end les affiche. L'utilisateur pourra ensuite ajouter au panier, ou acheter un e-ticket.

PolyTickets
Ticket details
Create a event


Tickets

<div>Last event</div> <div>From Johnny the</div> <div>Sat Oct 22 2022</div>	Buy e-ticket	<div>Buy ticket</div> <div>(Only 97 tickets left)</div>
<div>bonjour event</div> <div>From will the</div> <div>Wed Oct 19 2022</div>	Buy e-ticket	<div>Buy ticket</div> <div>(Only 100 tickets left)</div>
<div>Concert 4</div> <div>From Super Maxi the</div> <div>Sat Jun 22 2024</div>	Buy e-ticket	<div>Buy ticket</div> <div>(Only 0 tickets left)</div>
<div>Concert 4</div> <div>From Super Maxi the</div> <div>Sat Jun 22 2024</div>	Buy e-ticket	<div>Buy ticket</div> <div>(Only 99 tickets left)</div>
<div>Good Morning Yoga</div> <div>From MyYoga the</div> <div>Thu Oct 13 2022</div>	Buy e-ticket	<div>Buy ticket</div> <div>(Only 100 tickets left)</div>
<div>Concert 4</div>	Buy e-ticket	Buy ticket

Afin de tester cette FaaS, nous avons une stratégie (que nous continueront d'utiliser pour les autres services), qui est de mettre les capacités hardware de notre FaaS au minimum, et de voir la charge qu'elle accepte.

Nous avons deux possibilités, soit un scale vertical (qui consiste à augmenter la capacité hardware de notre produit), soit un scale horizontal (qui consiste à augmenter le nombre d'instances de notre service).

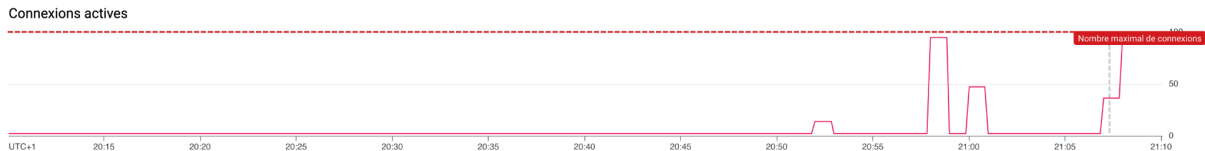
- Configuration initiale :
- Mémoire allouée : 128 MB
 - Délai avant expiration : 10 secondes
 - Autoscaling : de 0 à 1

Dans un premier temps, nous allons essayer de scaler notre service verticalement, nous allons donc augmenter la mémoire allouée et mesurer le nombre de requêtes qui peuvent être prises en compte.

Après quelques tests préliminaires, nous avons configuré Gatling pour exécuter 200 requêtes en une fois, et ensuite, nous mesurons le nombre de réponses correctes et le nombre d'erreurs. Pour être sûr que les résultats soient représentatifs, nous exécuterons 3 fois le même test, et nous ferons une moyenne des résultats.

Mémoire alloué	Réponses 200	Temps de réponse (ms) moyen 200	Réponses 429	Temps de réponse (ms) moyen 429
128 Mb	24 + 26 + 20 = 23.3	6423 ms	176 + 174 + 179 = 176.3	10972 ms
256 Mb	46 + 45 + 52 = 47,7	6868 + 5967 + 6830 = 6555	154 + 155 + 148 = 152.3	11550 + 10830 + 11601 = 11327

512 Mb	$100 + 118 + 114 = 110,6$	$6389 + 7330 + 6656 = 6792$	$100 + 82 + 86 = 89,3$	$11168 + 13236 + 12545 = 12316$
1 Go	$109 + 124 + 117 = 116,6$	$5298 + 6012 + 5390 = 5566,6$	$91 + 76 + 83 = 83,3$	$10051 + 10831 + 10383 = 10421,6$
2 Go	$116 + 123 + 111 = 116,6$	$5009 + 5351 + 4536 = 4965$	$84 + 77 + 89 = 83,3$	$8681 + 8626 + 9000 = 8769$



Sur le graphique ci-dessus, nous pouvons voir le nombre de connexions actives sur la base de données SQL (notre FaaS est uniquement connectée à cette DB event), on peut voir que l'on arrive au seuil de connexions actives, ce qui peut donc expliquer pourquoi nous restons globalement stable après 512MB, alors que avant 512Mb, nous doublons la capacité de charge dès que l'on double la quantité de mémoire allouée. On peut cependant apercevoir que le temps de réponse diminue sur les valeurs supérieures à 512 Mb. Donc, pour ce test, nous pouvons déduire que si l'on veut accepter plus de charges, il est inutile d'aller à plus de 512Mb de mémoire allouée, cependant, si l'on veut un temps de réponse plus petit, nous pouvons augmenter cette mémoire allouée.

Nous allons maintenant tester le scaling horizontal, c'est-à-dire que nous allons augmenter le nombre d'instances et voir si l'on peut augmenter la charge. La mémoire allouée quant à elle, reste à 128 MB

Maximum d'instances	Réponses 200	Temps de réponse (ms) moyen 200	Réponses 429	Temps de réponse (ms) moyen 429
1	$24 + 26 + 20 = 23.3$	6423 ms	$176 + 174 + 179 = 176.3$	10972 ms
2	$31 + 41 + 42 = 38$	$7360 + 6463 + 7045 = 6956$	$169 + 159 + 158 = 162$	$11625 + 11262 + 12606 = 11831$
4	$90 + 72 + 85 = 82,3$	$9357 + 6595 + 8240 = 8064$	$110 + 128 + 115 = 117,6$	$14227 + 11456 + 13762 = 13148$
8	$96 + 102 + 110 = 102,66$	$6193 + 5171 + 5777 = 5713$	$104 + 98 + 90 = 97,3$	$11232 + 10478 + 10686 = 10798$

Nous pouvons voir, grâce à ce test, que plus nous augmentons le nombre d'instances, plus le nombre de réponses correctes que la FaaS (en réalité, ce sont plutôt LES FaaS) peut gérer est grand, nous pouvons donc supposer que nous pouvons gérer une charge potentiellement infinie, tant que nous adaptons correctement la mémoire allouée, et le nombre d'instances. Le scaling horizontal, dans notre situation, n'est pas mieux que le scaling vertical, il faut jouer entre les deux, et ainsi, analyser par rapport aux coûts.

Nous devons cependant faire attention à l'architecture. En effet, nous remarquons que nous sommes bridés par la base de données qui limite le nombre de sessions actives. Ainsi, dès que l'on veut scaler pour augmenter la charge acceptable, il faut que notre base de données SQL scale proportionnellement aux services.

Comme nous avons vu que notre bottleneck est la DB "event-db", nous allons par l'intermédiaire de la fonction augmenter la capacité de notre base de donnée, afin d'avoir un ordre d'idée de l'ajout de performances nécessaire si l'on souhaite vraiment monter en charge.

Pour faire ce test, nous allons uniquement faire un test de scale vertical de la base de données. Ayant une problématique de temps, la duplication des bases de données nous prendrait trop de temps, et d'après nos tests ci-dessus, nous nous sommes aperçus qu'il y a une proportionnalité à avoir entre scale horizontal et vertical. Nous souhaitons juste savoir ici quelle charge peut gérer notre base de données en augmentant ses capacités hardware. Pour faire ceci, nous avons configuré notre FaaS avec 1 à 16 instances actives, une mémoire allouée de 1Go par instance, et 10s de timeout.

Test 1 :

Nombre de processeurs	Mémoire (Go)	Requêtes acceptés	connexion actives maximales autorisées	prix / mois	(prix / mois) / connexion actives max
1	3.75	95	100	51.01\$/mois	0.51\$ / connexion
2	7.5	407	400	100.32\$/mois	0.25\$ / connexion
4	15	495	500	198.95\$/mois	0.4\$ / connexion
16	60	828	800	790.68\$/mois	0.98\$ / connexion

Dans ce tableau, nous pouvons constater que, pour notre service (qui est uniquement de la lecture des events en DB), le nombre de requêtes acceptées est directement proportionnel aux connexions actives maximales autorisées par GCP, avec un ratio de corrélation de 1 pour 1. Nous en déduisons que, l'objectif sera de trouver le nombre maximal de connexions actives autorisées pour le prix minimum. Car, d'après les graphiques du monitoring, nous ne

dépassons généralement pas les 20% de mémoire utilisée au moment du test. La deuxième observation que nous pouvons faire, est qu'il sera nécessaire de scaler horizontalement notre base de donnée, afin qu'elle puisse gérer plus de connexions, car ces connexions pourront être exécutées simultanément.

Pour résumer, dans cette étude, nous sommes passés par 3 étapes, la première est de tester la capacité de notre FaaS grâce à un scaling vertical et horizontal. Nous en avons déduit 2 choses, la première, est que ces deux types de scaling doivent être proportionnels, la deuxième, que dans notre système, la bottleneck était bel est bien notre base de données SQL. Nous avons donc fait une étude, en mettant les capacités de notre FaaS au maximum, pour voir comment notre base de données gère la charge. Nous en avons déduit que suivant notre charge prévue, nous devons augmenter les capacités de cette base de données afin d'y accepter plus de connexions actives. Et que si nous avons énormément de trafic prévu, nous devons scaler horizontalement notre base de données. La dernière étape de cette étude est d'essayer de trouver une proportionnalité entre la charge que nous aurons sur notre service, et une recommandation de la configuration de nos services GCP. Le but est de payer le moins possible de ressources, tout en ayant un service fonctionnel. Afin de faire ce résumé, nous avons testé empiriquement plusieurs combinaisons, afin de trouver la moins coûteuse.

Voici le tableau récapitulatif des tests fait pour créer une estimation des ressources nécessaires. Nous avons testé plusieurs combinaisons de performances, en mettant notre base de données au maximum de sa charge, pour s'assurer d'aller aux limites de la FaaS, et non aux limites de la base de données.

	128 Mb		256 Mb		512 Mb	
1 instance	20 requêtes	0.16 requêtes / Mb	55 requêtes	0.21 requêtes / Mb	103 requêtes	0.2 requêtes / Mb
2 instances	35 requêtes	0.27 requêtes / Mb	101 requêtes	0.39 requêtes / Mb	251 requêtes	0.49 requêtes / Mb
4 instances	70 requêtes	0.55 requêtes / Mb	170 requêtes	0.66 requêtes / Mb	527 requêtes	1.02 requêtes / Mb
8 instances	143 requêtes	1.12 requêtes / Mb	310 requêtes	1.21 requêtes / Mb	1043 requêtes	2.03 requêtes / Mb
16 instances	330 requêtes	2.58 requêtes / Mb	763 requêtes	2.98 requêtes / Mb	1091 requêtes	2.13 requêtes / Mb

	128 Mb		256 Mb		512 Mb	
1 instance	20 requêtes	0.16 requêtes / Mb	55 requêtes	0.21 requêtes / Mb	103 requêtes	0.2 requêtes / Mb
2	35 requêtes	0.13 requêtes / Mb / 2	101 requêtes	0.2 requêtes / Mb / 2	251 requêtes	0.25 requêtes / Mb / 2

instances						
4 instances	70 requêtes	0.14 requêtes / Mb / 4	170 requêtes	0.16 requêtes / Mb / 4	527 requêtes	0.25 requêtes / Mb / 4
8 instances	143 requêtes	0.14 requêtes / Mb / 8	310 requêtes	0.15 requêtes / Mb / 8	1043 requêtes	0.25 requêtes / Mb / 8
16 instances	330 requêtes	0.16 requêtes / Mb / 16	763 requêtes	0.19 requêtes / Mb / 16	1091 requêtes	0.13 requêtes / Mb / 16


Nous pouvons apercevoir que, lorsque l'on est à 512 Mb à 16 instances, nous saturons la base de données, donc nous ne pouvons pas nous fier à la case tout en bas à droite. Cependant, nous nous apercevons clairement que la FaaS en 512Mb est bien plus rentable, niveau mémoire déployée par rapport aux nombres de requêtes associées que la version en 128 Mb et en 256 Mb. Pour une version 256 Mb avec 8 instances, nous sommes sur un coût simulé de 21.9\$/mois, et pour 512Mb, un prix de 43.81\$/mois. Donc globalement, par rapport à l'analyse du coût final, la version 256 Mb serait plus intéressante, car pour la moitié du prix, on obtient un peu plus que pour la version 512Mb (à laquelle on aurait divisé les requêtes/Mb par 2).

Trafic (en users)	Ressources
< 1000	3 * (SQL 2 processeurs, 7.5 Go mémoire) Faas : 16 instances à 512 Mb
< 10.000	30 * (SQL 2 processeurs, 7.5 Go mémoire) Faas : 16 instances à 4 Go
< 100.000	300 * (SQL 2 processeurs, 7.5 Go mémoire) Faas : 164 instances à 8 Go


2) Test de la PaaS ticket-booking

Le but de ce test est un peu différent d'avant, nous allons essayer de regarder si notre produit répond à beaucoup de prises de commande en même temps. Nous pouvons nous imaginer, par exemple, le vendredi soir de l'ouverture de la billetterie pour le dernier concert de Jonhny. Moment attendu par la France entière, notre site est scruté par plus de 50.000 personnes qui veulent s'arracher les 200 places du dernier concert privé de Johnny.



Pour tester ce scénario, nous allons d'abord créer un événement

PolyTickets Ticket details Create a event 


Create new event


Event name *
Jonhny concert privée 

Event available seats *
200

Choose a date *
11/24/2022  

MM/DD/YYYY

Artist name *
Jonhny la légende 

Your email *
manager@jonhny.com 

Search

Edit an event

Nous retrouvons le concert ici, nous allons donc simuler une forte charge sur le bouton buy-ticket, afin de voir comment notre système se comporte.

Jonhny concert privée

From
Jonhny la légende
the
Thu Nov 24 2022

Buy e-ticket

Buy ticket
(Only 200 tickets left)

Pour tester ceci, nous avons une simulation Gatling très simple.

```
// protocol
val httpProtocol = http
  .baseUrl( url = "https://ticket-booking-idnoihwhaq-uc.a.run.app")

// scenario
val scn = scenario( scenarioName = "Add a new johnny event ticket in the cart").exec(
  http( requestName = "Get all events")
    .get("/create-ticket/31")
    .check(
      status.is( expected = 200)
    )
)

// setup
setUp(
  scn.inject(atOnceUsers( users = 1000))
  .protocols( httpProtocol )
)
```

Nous avons envoyé, en même temps, 1000 requêtes d'ajout au panier, alors que nous avons 200 disponibles. En théorie, nous sommes censés recevoir 200 requêtes au status 200 (indiquant que le panier a bien été créé) et 800 requêtes au statut 204 (qui indiquent qu'il n'y a plus de " places de panier disponible ").

Voici la configuration de la PaaS (event-booking) qui aura pour objectif de gérer cette charge:

- 512 Mio de mémoire et 2 processeurs
- 300s de délai avant expiration
- 1 à 64 instances

Pour les bases de données (en dehors de la MongoDB que nous ne pouvons pas faire évoluer car héberger sur Atlas), nous avons mis beaucoup de capacité afin de s'assurer qu'elles ne jouent pas le rôle de bottleneck.

Analyse des résultats :

--- Global Information -----			
> request count	1000	(OK=32	KO=968)
> min response time	3936	(OK=3936	KO=4734)
> max response time	19843	(OK=19843	KO=18873)
> mean response time	10118	(OK=12380	KO=10043)
> std deviation	5261	(OK=6652	KO=5192)
> response time 50th percentile	11669	(OK=16292	KO=11337)
> response time 75th percentile	14009	(OK=18123	KO=13937)
> response time 95th percentile	18768	(OK=19801	KO=18765)
> response time 99th percentile	18839	(OK=19834	KO=18824)
> mean requests/sec	50	(OK=1.6	KO=48.4)

Les résultats ne sont pas très bons, sur les 1000 requêtes passées, nous nous rendons compte que seulement 32 carts ont été générés.

cart-tickets.cart-tickets

32 1
DOCUMENTS INDEXES

Documents Aggregations Schema Explain Plan Indexes Validation

FILTER { field: 'value' } OPTIONS FIND RESET ...

ADD DATA VIEW ...

Displaying documents 1 - 20 of 32 < > C REFRESH

```
{
  "_id": ObjectId('6364158ce9dff76b7b62e332'),
  "id_cart": "6d63e2a9-60e6-4093-a45a-8025c97f16c9",
  "expirationDate": "2022-11-03T19:30:00.747Z",
  "event_id": 31
}
```

```
{
  "_id": ObjectId('6364158ce0705278fccbf5b')
}
```

On peut voir qu'ils ont bien été créés dans la collection MongoDB. Ce chiffre est très faible, mais dû au fait que la base de données MongoDB est hébergée dans un cluster partagé. En effet, pour des raisons de praticité, nous avons fait le choix d'héberger nos 2 collections (tickets et cart-tickets) sur MongoDB atlas, dans la même base de données. Pour la production, nous pouvons clairement voir que ce n'est pas suffisant.

Nous savons que c'est la base de données MongoDB qui a joué le rôle de bottleneck car sur le monitoring MongoDB atlas, nous avons atteint tous les seuils. De plus, en analysant

les logs du service PaaS event-booking, nous nous apercevons que notre PaaS a essayé de se connecter à la base de données mais que celle-ci a automatiquement fait expiré la session.

La solution pour la version de production est de créer des bases de données MongoDB distinctes (donc 2 mongoDB) dans deux machines virtuelles différentes, et de les scaler selon des tests préalable, ou bien de payer le service MongoDB atlas.

Nous sommes un peu déçus par ce résultat, et nous nous rendons compte que l'hébergement partagé par MongoDB Atlas n'était pas un choix technique judicieux.

Grâce aux expériences menées et détaillées ci dessus, un détail complet des prix des différents composants a été réalisé à l'aide de l'outil de simulation fourni par GCP. Les simulations sont retrouvées dans le dossier lié au rapport.

Supervision envisagée du système

Pour la solution de monitoring, qui nous permet d'avoir le retour sur comment nos services ont réagi dans le cloud, nous avons essayé d'implémenter un serveur Prometheus avec la visualisation sous Grafana, mais nous nous sommes aperçus que le monitoring interne à GCP nous était largement suffisant, nous avons donc décider de rester sur cette solution. Pour ce faire, nous avons intégré grâce à l'outil "explorateur de métriques" les métriques relatives à nos FaaS, et à nos PaaS, ainsi que la base de donnée SQL (la base de donnée MongoDB étant hébergée sur Atlas, nous n'y avons pas accès)

Globalement, nous n'avions pas vraiment utilisé le dashboard de monitoring car, en général, lorsque nous faisons nos tests, nous utilisons le dashboard de monitoring directement intégré au service FaaS, PaaS, et SQL. En effet, sur notre dashboard principal, nous avons juste mis des metrics d'instance actives et de trafic, tandis que sur les dashboard des FaaS, PaaS et SQL, nous avons les détails du trafic entrant, des temps de réponses, etc. Nous avons fait ce choix, car, pour nos tests et notre monitoring, nous regardions chaque fois en détails chaque instance, et nous ne ressentions pas le besoin de tout centraliser.

Conclusion

A travers ce projet, nous avons pu découvrir l'environnement cloud général mais aussi l'outil GCP. Il nous a permis de mettre en relation les notions d'architectures que nous avons vu dans les autres cours, tout en nous posant les questions de l'implémentation réelle avec le déploiement.

Ce projet nous a permis de nous rendre compte de la flexibilité du cloud, mais aussi ses limites, notamment par rapport au coût. Nous nous sommes rendu compte que, le cloud de GCP nous permet d'avoir accès à de nombreuses technologies, mais que en même temps, les frais relatifs à ces services peuvent rapidement devenir très élevé, il faut donc que la partie financière de l'entreprise soit au courant de ces contraintes avant de faire le choix du cloud.

Cette étude nous a fait prendre du recul par rapport à ce que l'on entend dans les médias, mais aussi que la gestion de l'infrastructure par l'intermédiaire de serveurs bare-métal et d'une équipe qui gère l'infrastructure, peut, dans certains cas être plus rentable, car pour le cloud, nous sommes sur des coûts variables, tandis que la gestion d'une architecture classique entraîne des coûts qui sont relativement fixe.

Annexes

Cloud CDN



Cache egress - Europe: 2,000 GiB

USD 160.00

Cloud SQL for PostgreSQL

EVENT DB



of instances: 1

Instance type: db-standard-2

Location: Iowa

730.0 total hours per month

SSD Storage: 30.0 GiB

Backup: 0.0 GiB

USD 103.72

Cloud Functions

Event listing



Region: Iowa

Invocations: 10,000	USD 0.00
---------------------	----------

RAM (GiB-seconds): 240,000 per month	USD 0.00
--------------------------------------	----------

CPU (GHz-seconds): 288,000 per month	USD 0.88
--------------------------------------	----------

Networking: 29.297 GiB per month	USD 2.92
----------------------------------	----------

Minimum number of instances: 8	USD 315.39
--------------------------------	------------

USD 319.19

A portion of your estimate fits within the [Cloud Functions free tier](#).

event manager



Region: Iowa

CPU Allocation Type: CPU is only allocated during request processing

CPU: 1

Memory: 0.25 GiB

CPU Allocation Time: 1,500 vCPU-second

Memory Allocation Time: 375 GiB-second

Requests: 50 requests

minimum number of instances: 1

minimum number of instances Price: USD 8.21

USD 8.21

Cloud Run

ticket booking



Region: Iowa

CPU Allocation Type: CPU is only allocated during request processing

CPU: 2

Memory: 0.5 GiB

CPU Allocation Time: 6,000 vCPU-second

Memory Allocation Time: 1,500 GiB-second

Requests: 10,000 requests

minimum number of instances: 1

minimum number of instances Price: USD 16.43

USD 16.43

Ticket cart checker



Region: Iowa

Invocations: 8,760	USD 0.00
RAM (GiB-seconds): 10,950 per month	USD 0.00
CPU (GHz-seconds): 17,520 per month	USD 0.00
Networking: 8.555 GiB per month	USD 0.43
Minimum number of instances: 1	USD 1.37

USD 1.80

Ticket details



Region: Iowa

Invocations: 10,000	USD 0.00
RAM (GiB-seconds): 25,000 per month	USD 0.00
CPU (GHz-seconds): 40,000 per month	USD 0.00
Networking: 97.656 GiB per month	USD 11.12
Minimum number of instances: 1	USD 5.48

USD 16.59

Total Estimated Cost: USD 658.79 per 1 month

VOD service



Region: Iowa

CPU Allocation Type: CPU is only allocated during request processing

CPU: 2

Memory: 0.5 GiB

CPU Allocation Time: 201 vCPU-second

Memory Allocation Time: 50.25 GiB-second

Requests: 10,050 requests

minimum number of instances: 2

minimum number of instances Price: USD 32.85

USD 32.85
