LOG8415 - Concepts avancés en infonuagique
Analyse de files d'attente

Baptiste Pauletto
Matricule 2096684
Étudiant au Departement Génie Informatique et Génie Logiciel
École Polytechnique de Montréal, Québec, Canada
baptiste.pauletto[at]polymtl.ca
Répertoire Github
Démonstration

December 17, 2021

1 Introduction (*Introduction*)

Dans le cadre de ce projet personnel pour le cours d'infonuagique **LOG8415**, j'ai souhaité traiter un problème rencontré à plusieurs reprises et qui, selon moi, pourrait bénéficier des avantages de l'infonuagique.

1.1 Problème (*Problem*)

Régulièrement, dans le cadre professionnel pour des ouvertures d'événements dans des billetteries (type concert) ou pour des nouveaux accès à des services, on constate la mise en place d'une file d'attente plus ou moins équitable entre les participants. Effectivement, l'objectif principal (pas toujours respecté) est d'être capable de subvenir à la charge de travail en temps utile mais d'autres objectifs secondaires sont également à considérer. Parmi eux, on veut garder une équité entre toutes les requêtes entrantes (FIFO), fournir une estimation réaliste de la durée d'attente et permettre l'accès au service demandé sans le surcharger.
Les trois objectifs cités précédemment sont raisonnablement atteignables avec les moyens disponibles via l'utilisation de technologies de l'infonuagique. En effet, ces systèmes sont capables de se mettre à l'échelle efficacement pour subvenir à des demandes très intenses pendant des durées de temps définies. De plus, cela permettrait aux clients de ce type de système de ne pas s'encombrer de machines extrêmement performantes alors qu'il n'en ferait un usage que très ponctuel. Cependant, afin d'avoir un objectif atteignable dans le cadre de ce projet personnel, mon objectif principal restera la mise en place de l'architecture, la compréhension des enjeux de ses composants et l'automatisation de son déploiement.

C'est en ayant ce contexte en tête, que j'ai alors décidé de mettre en place une architecture capable de potentiellement répondre à ces besoins. D'autre part, l'envie personnelle d'en apprendre plus sur les possibilités offertes par des fournisseurs de service en infonuagique, type AWS/Azure, se

couplait bien avec le cadre de l'étude. C'est donc autour de cette question de la mise en place de files d'attente sur une infrastructure d'infonuagique que s'axera ce projet.

1.2 État de l'art (*Literature Review*)

En ce qui concerne l'état de l'art, la majorité des publications dans le monde de l'infonuagique s'axent autour de l'accès aux ressources partagées entre tous les utilisateurs d'un fournisseur, qui serait limité et qui aurait donc à établir une file d'attente, pour ordonner l'accès aux ressources. Nous pouvons assimiler notre cas d'étude à cette situation en considérant que nous sommes en charge d'établir la file d'attente avant l'accès aux ressources (achat/accès à un service).

Parmi les solutions proposées, on retrouve le fait de modéliser nos files d'attentes globales en un grand nombre de files réduites mises en place sur les serveurs d'applications virtuelles directement. En effet, dans l'article [7.1], ils modélisent les applications webs comme des queues et les machines virtuelles comme des fournisseurs de service. Cette approche a l'avantage de mieux répartir les responsabilités et charges au sein du système global, ce qui permet d'avoir une meilleure granularité si l'on doit scale-up ou down. Un autre mécanisme, basé cette fois sur la pré-emption pour des tâches de hautes importances est également mis en avant dans l'article [7.2]. Pour réaliser cette pré-emption, il se base sur un hyper-paramètre, celui de la combinaison des *SLA* afin de déterminer quelle tâche doit terminer au plus tôt et peut prendre la main sur une autre.

D'autre part, un phénomène particulier fait également partie du principe de file d'attente en ligne, celui de l'abandon et de l'essai à nouveau. Pour comprendre sa distribution et son fonctionnement, l'article [7.3] propose une approche mathématique sous la forme de réseau de service Markovien et détaille l'effet d'un grand nombre de paramètres sur ce type de comportements (temps d'attente moyen, incidence sur le retrait de la file...). Bien que cela n'impacte pas directement le problème, la lecture de cet article permet de se rendre compte de notre capacité à modéliser de tels phénomènes, modélisation qui pourrait ensuite être prise en charge par notre solution.

1.3 Solution proposée (*Proposed Solution*)

La solution proposée à cette problématique sera donc de mettre en place un service capable de recevoir des messages s'apparentant à des demandes de connexion ou de paiement et de procéder à tout un traitement par la suite. Effectivement, une fois ce message reçu, il faudra le placer dans une file d'attente avec d'autres messages pour pouvoir les traiter convenablement, tout en respectant l'ordre. Une fois le tour venu au message courant d'être traité, il sera pris en charge par une fonction qui réalisera le traitement nécessaire pour enregistrer l'utilisateur et potentiellement, lui permettre d'accéder à un service. N'ayant qu'un temps limité pour implémenter cette solution et ayant beaucoup à apprendre sur ses composants, certaines parties seront simplifiées afin se concentrer sur l'essentiel du traitement : la réception du message et son acheminement correct. Davantage de détails quant aux parties prenantes de ce système seront fournis dans les parties suivantes.

Si cette solution apporte des résultats intéressants sur cette partie clef du traitement, il serait alors intéressant d'étendre l'analyse à un véritable système pour en vérifier la pertinence.

2 Architecture (*Architecture*)

Dans cette partie dédiée à l'architecture de notre solution, nous allons entrer dans les détails concernant chacun des composants de notre système et expliquer leurs rôles afin d'avoir une vision

d'ensemble du projet. Afin de se familiariser avec les différents noms des outils offerts par AWS (utilisés dans la partie suivante), la figure reprenant notre architecture fait usage de ces derniers.

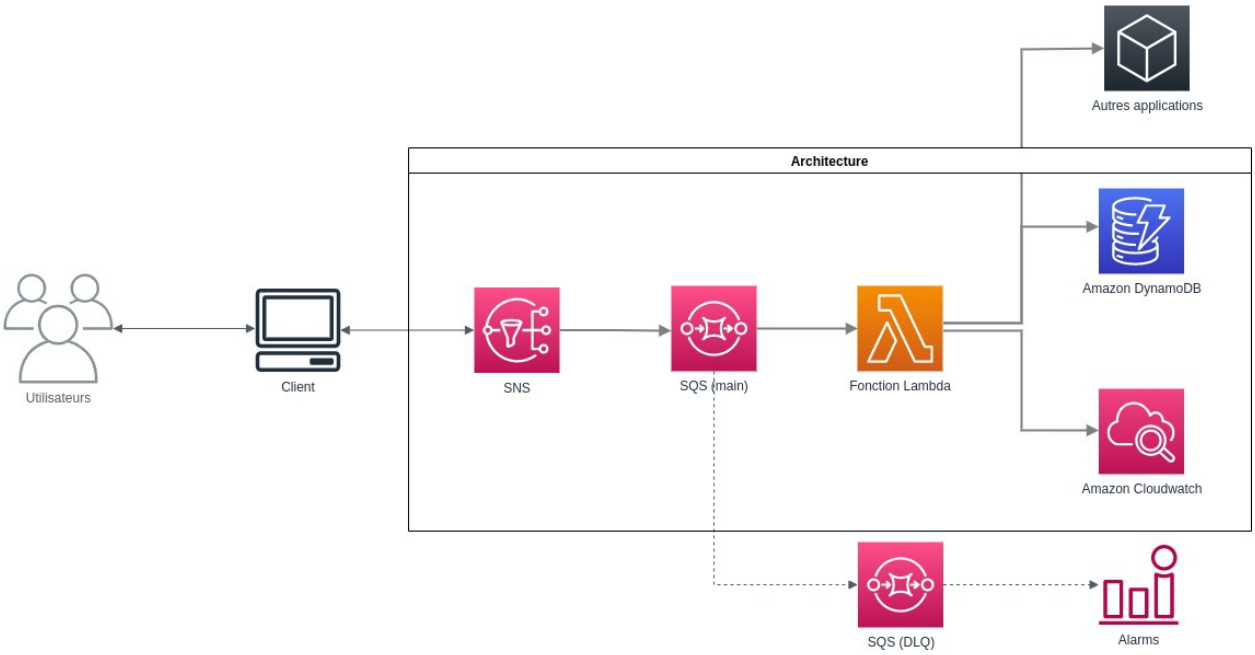

Figure 1: Architecture globale du projet (composants AWS)

Afin de faciliter la compréhension de l'architecture, nous détaillerons cette dernière en expliquant les composants de la gauche vers la droite.

Sur la Figure 1, on aperçoit donc à l'entrée de notre architecture, un ensemble d'usagers qui échangent avec un client défini. Ce client transmet ensuite ses requêtes vers notre frontière entre le monde extérieur et le service proposé par notre architecture infonuagique. Pour simuler cette situation d'envoi de messages de la part d'un client extérieur au système, nous avons fait usage d'une machine **Docker**. Cette dernière se contentera d'envoyer un ensemble de requêtes à notre système, reprenant alors l'idée que cela pourrait être de vrais utilisateurs en train de nous solliciter. Toutes les étapes suivantes se déroulent sur la plateforme du fournisseur de service.

Effectivement, dans un premier temps, l'ensemble des messages reçus sont structurés sous la forme de messages avec un ensemble d'attributs et de champs bien précis permettant de simplifier les analyses à venir. C'est le travail que rempli cette interface avec le monde extérieur, sous le nom de *Simple Notification Service* chez AWS. Afin de permettre une mise en situation plus réaliste, nous allons utiliser SNS, qui propose la création de *topics* (sujets) reprenant le patron *publish/subscribe*. En effet, nous allons alors exploiter ces sujets pour simuler différentes interactions simultanées (pour prendre un exemple concret, on peut imaginer deux billetteries mises en vente en même temps).

Une fois ces messages formatés et publiés au sein de sujets, le prochain composant entre en jeu puisqu'il constitue la deuxième partie du patron (le *subscriber*). Effectivement, la file d'attente sera mise en place ici avec une réception des messages indépendamment des sujets auxquels ils sont liés afin d'assurer l'équité entre toutes les entrées dans le système. Un des avantages de ce système,

est la possibilité (si souhaitée par le client) de multiplier les files d'attente afin de permettre à des utilisateurs intéressés par un évènement souffrant moins de l'effet d'un fort trafic d'accéder au service demandé plus rapidement. Cela serait alors accompli par des files d'attentes parallèles qui s'abonneraient à des sujets uniques. Le service AWS utilisé ici est *Simple Queue Service* qui permet de réaliser tout ce dont nous avons besoin pour nos cas d'utilisation.

La prochaine étape, à la sortie de cette file d'attente, est le traitement convenable du message. Pour ce faire, *Lambda*, approche *serverless* permettant d'exécuter du code sans se soucier des ressources nécessaires a été le composant qui a retenu notre attention. Effectivement, puisque nous souhaitions pouvoir permettre à notre client de procéder à un traitement de son choix (enregistrement en base de données, modification de permissions d'accés ou autre), nous avons opter pour la versatilité de ce composant. C'est à dire qu'à cette étape, tout peut être envisagé pour le traitement du message en entrée. Dans le cadre de notre analyse, nous avons opté pour le stockage en base de données de l'information utile. Cependant, il est tout à fait imaginable de mettre en place des traitements plus lourds étant donné les capacités offertes par Lambda et l'élasticité permettant de répondre au mieux à la demande en entrée.

La dernière étape se trouve dans l'analyse de nos résultats et performances au travers de plusieurs outils. Tout d'abord, vérifier que le traitement réalisé à l'étape précédente a effectivement fonctionné en consultant le contenu de la base de données DynamoDB et en vérifiant si l'ordre est bien conservé. Cette étape de consultation pourrait être remplacée par un autre traitement plus complexe dans un cadre plus réaliste mais pour notre analyse, cela nous permettait de s'assurer du bon fonctionnement de notre système. Parallèlement, il est également possible de placer des repères sur certaines métriques afin de suivre la consommation des ressources et d'avoir connaissance accrue de l'état du système. C'est au travers de *Cloudwatch*, composant AWS, que l'on peut réaliser cela sur chacune des étapes de notre architecture. Effectivement, un suivi détaillé d'un grand nombre d'indicateurs (exécutions concurrentes, erreurs, durée...) permet de comparer les solutions et d'agir en conséquence si cela s'avère nécessaire.

Maintenant que la représentation de notre architecture est claire, attardons nous plus longuement sur les particularités et capacités des services AWS utilisés.

3 Services AWS (*AWS Services*)

Dans chacune des sous-sections suivantes, nous allons voir, plus précisément, quels sont les services choisis pour prendre les différents rôles composants notre architecture. Les services utilisés proviennent tous d'un seul et même fournisseur, **Amazon Web Services** afin de simplifier la mise en place de l'architecture et de permettre l'automatisation de sa mise en place par l'usage de scripts/code.

3.1 SDK Python (*boto3*)

Dans un premier temps, l'outil qui nous a permis de mettre en place tous les autres au travers de code **Python**, plutôt que la console AWS pour gagner du temps et automatiser chacune des étapes de la mise en place de notre architecture : *boto3*. Ce dernier fonctionne de la même manière que pourrait le faire des scripts, mais simplifie grandement la prise en main grâce à une documentation

très détaillée. Effectivement, à la moindre réalisation, nous disponsons d'informations cruciales concernant les paramètres requis, les optionnels, les types de données gérés, etc... De plus, le guide du développeur d'AWS recommande son utilisation et propose parfois des indications quant à des situations classiques rencontrées en utilisant *boto3*.

Nous avons donc fait usage de ce dernier pour créer des *wrappers* afin de faire appel aux fonctions AWS qui nous intéressaient pour mettre en place notre architecture (SNS, SQS, Lambda, DynamoDB...). Ces fichiers sont disponibles dans l'archive du code ou sur **Github**, dans le sous dossier */python* du projet.

3.2 Simple Notification Service (*SNS*)

Simple Notification Service ou SNS, au travers des sujets précédemment présentés, nous a permis de modéliser les potentielles séparations entre utilisateurs d'un même système. Effectivement, en constituant la porte d'entrée de notre architecture, c'est ce composant qui gère la publication des messages représentant les entrées dans le système dans les différents *topics*. Il propose un ensemble de paramètres pour pouvoir affiner le traitement de ces sujets, parmi ceux-ci : des attributs nécessaires pour certains sujets, des formats à respecter, des sujets de type FIFO. Ce dernier paramètre est utilisé dans une partie de notre analyse afin de comparer entre files d'attente standards et queue FIFO.

D'autre part, les messages publiés dans les sujets peuvent être reçus par de nombreux composants AWS, mais dans notre cas, nous allons nous concentrer sur les files d'attente, d'où SQS, le prochain composant.

3.3 Simple Queue Service (*SQS*)

Simple Queue Service ou SQS quant à lui nous a permis de créer un ensemble de files d'attente avec des attributs propres à chacune à la manière de SNS. Effectivement, des attributs tels que les types de données géré, le délai avant distribution ou encore le type de file d'attente sont gérés par ce service. Les deux types de files d'attente qui nous ont intéressé sont les files standards qui supportent un nombre quasi-illimité d'appels par seconde mais qui se base sur le principe du "au-moins une fois". C'est à dire qu'il est possible qu'à cause de la complexité induite pour soutenir un nombre d'appels simultanés, certains rares messages soit remis plus d'une fois. Ayant connaissance de ce phénomène, si c'est la solution que l'on choisit, il faudra alors faire attention à construire des messages idempotents afin qu'une réinterprétation de ces derniers ne change pas le résultat final. L'autre type de file d'attente est les queues FIFO (*First In First Out*), elles ont les mêmes propriétés que les queues standards mais en plus, elles assurent le respect de l'ordre d'arrivé et supprime les potentiels duplicats. C'est donc ce contexte là qui se rapproche le plus de nos besoins.

Maintenant que l'on connait les types de file d'attente utilisés, on peut passer aux fonctions qui vont s'occuper de prendre en charge les messages à la sortie de ces files.

3.4 Fonctions Lambda (*Lambda functions*)

Les fonctions *Lambda* sont un des composants principaux de l'approche *serverless* d'AWS. Effectivement, elles nous permettent d'exécuter du code dans un ensemble de langages sans se soucier de l'implémentation physique et logique des serveurs derrière (FaaS). C'est en ce point qu'elles constituent une excellente alternative à des systèmes comme AWS EC2 (IaaS) qui demandent des

experts pour les paramétrer et de la connaissance technique spécialisée dans notre cas. En faisant usage de Lambda, on ne se soucie plus de ces réglages puisqu'ils sont gérés par le fournisseur de service et que l'on est seulement responsable du code que l'on place sur ces fonctions. Cette simplicité a tout de même un coût mais reste très bas prix et une propose une excellente fiabilité.

Dans le cadre de notre projet, nous avons utilisé ces dernières pour permettre d'ajouter en base de données l'information contenue dans les messages extraits de la file d'attente en entrée. Il est tout à fait possible de réaliser des opérations bien plus complexes qui nécessiteront alors à Lambda de scale-up en cas de forte demande et de scale-down, une fois, le pic d'activité terminé. C'est dans ce sens que l'on comprend la pertinence d'utiliser Lambda pour ce type de problèmes réels (pics très ponctuels).

3.5 Base de données DynamoDB (*DynamoDB database*)

La base de données *DynamoDB*, comme *Lambda*, se base sur l'approche *serverless* et suit les principes *NoSQL* en proposant un stockage sous la forme clef-valeur (non-relationnelle). Ces bases de données sont extrêmement fiables puisqu'elles proposent de la gestion de flux pour des applications à haute performance, de la mise en cache en mémoire, des sauvegardes automatisées et de la réplication inter-région pour assurer une disponibilité maximale.

Pour notre cas, nous n'avons pas fait usage de flux directement (notamment avec AWS *Kinesis*), mais la base de données *DynamoDB* aurait pu être capable de gérer une telle quantité de données arrivant simultanément sans problèmes. Effectivement, dans notre approche, nous nous sommes contentés de prouver le fonctionnement de nos fonctions Lambda en faisant des entrées en base de données correspondantes aux messages reçus (stockage des attributs importants comme numéro de ticket, numéro de client, classe du ticket...).

3.6 Métriques CloudWatch (*CloudWatch Metrics*)

Les métriques *CloudWatch* sont proposées pour l'ensemble des services d'AWS pour obtenir un suivi du coût de chacun d'entre eux puisque c'est le système sur lequel se base AWS (On paye ce que l'on utilise). De plus, pour chaque composant, un ensemble de métriques est fourni pour s'assurer du bon fonctionnement de notre architecture (nombre d'erreurs, durée d'exécution, invocations ...) et ce qui nous a permis, tout au long du projet, d'être sûr que tout fonctionnait comme nous l'entendions.

4 Flux de données et méthodologie (*Dataflow and methodology*)

Dans cette partie, nous nous attarderons sur comment les composants ont pu communiquer entre eux pour établir le flux de données au sein d'AWS et sur la méthodologie employée pour tester notre architecture (scénarios).

4.1 Abonnements, déclencheurs et rôles (*Subscriptions, triggers and roles*)

Les outils principaux qui nous ont permis d'établir une communication automatisée entre les parties prenantes de notre architecture sont les abonnements, les déclencheurs et les rôles. Chacun a des particularités qui lui sont propres et en cumulant les trois, on parvient à faire fonctionner le système dans son ensemble.

4.1.1 Abonnements (*Subscriptions*)

Dans un premier temps, les abonnements, qui sont liés au patron de conception *publish/subscribe* prennent place entre les deux premiers composants de l'architecture (*SQS* et *SNS*). Effectivement, pour qu'une file d'attente puisse gérer les messages qui sont publiés, nous devons faire savoir à AWS qu'un lien existe entre les deux, ce lien est matérialisé par un abonnement. En plus de cet abonnement signifiant que nos deux composants sont liés, il faut également autoriser SNS à publier au sein d'un topic qui sera ensuite récupéré par *SQS*.

Cette partie est supposée être gérée automatiquement depuis AWS à la création d'un abonnement entre les deux, mais dans l'exercice, nous nous sommes rendus compte que ce n'était pas le cas via *boto3* alors que depuis la console en ligne, cela se faisait automatiquement. Pour pallier ce problème, nous avons donc du mettre à jour la politique propre à notre file d'attente *SQS* pour rajouter cette autorisation de publication pour *SNS*. Cette réalisation peut-être retrouvée dans le code du wrapper *SQS*, dans la méthode *generate_policy()*.

Une fois cette connexion établie et fonctionnelle, on peut passer à la prochaine étape : les déclencheurs.

4.1.2 Déclencheurs (*Triggers*)

Les déclencheurs sont utilisés dans *Lambda* pour symboliser la réaction à un évènement d'une source définie. Dans notre cas, un nouveau message arrivant dans la *SQS* constitue un déclencheur pour notre fonction *Lambda* qui va devoir alors s'en saisir et la traiter comme son code lui indique de le faire. De plus, c'est dans ce dernier qu'il est possible de choisir la taille des groupements de messages si l'on souhaite procéder par groupe plutôt que message après message. Effectivement, c'est l'attribut *BatchSize*, lors de la création du déclencheur qui indiquera à *Lambda* si les messages reçus sont groupés ($BatchSize > 1$) ou s'ils arrivent un par un ($BatchSize = 1$).

La mise en place des déclencheurs peut être trouvée dans le code du wrapper *Lambda*, dans la méthode *add_trigger()*. Maintenant que notre fonction *Lambda* est capable de recevoir et traiter les messages reçus par la file d'attente, voyons l'importance des rôles dans cette architecture.

4.1.3 Rôles (*Roles*)

Les rôles IAM (*Identity and Access Management*) sont une des composantes clefs du fonctionnement global d'AWS puisqu'ils permettent de déterminer les autorisations des utilisateurs. Ils sont également utilisés par les utilisateurs pour définir des autorisations d'accès aux ressources qui leurs sont propres. Effectivement, ils se basent sur des stratégies qui regroupent un ensemble de droit pour un composant bien défini (par exemple, *CloudWatch*) et peuvent ensuite être assignés à des composants et/ou utilisateurs (si l'on souhaite donner l'accès à un autre utilisateur AWS, par exemple). Dans notre cas, nous étions l'unique propriétaire de tous les composants mais nous souhaitions également pouvoir utiliser les rôles afin d'autoriser, par exemple, une fonction *Lambda* à écrire dans une table de base de données *DynamoDB*.

Pour ce faire, nous avons du créer une stratégie regroupant les autorisations d'écriture et de lecture sur une base de données DynamoDB, puis l'associer à un rôle. Ce rôle particulier a finalement été associé à notre fonction *Lambda*, sans lequel elle n'aurait pas pu écrire dans la BD. Cette association de rôle peut être trouvée dans le code du wrapper *Lambda*, lors de la création de la fonction, méthode *create_function()*.

Une autre information importante concernant les rôles est le rôle par défaut associer à tous les composants de notre architecture, celui de publier dans *Cloudwatch*. Effectivement, c'est grâce à ce rôle que nous avons pu récupérer tout au long de l'analyse, des informations sur les composants pour s'assurer du bon fonctionnement de l'architecture globale. On peut remarquer que pour notre rôle particulier, lié à notre fonction *Lambda* détaillé précédemment, il a suffit d'ajouter la stratégie au rôle créé pour lui permettre à son tour d'être suivi par *Cloudwatch*.

Maintenant que notre système est opérationnel, nous allons nous attarder sur la méthodologie utilisée pour vérifier le bon fonctionnement global du système.

4.2 Scénarios (*Scenarios*)

La méthodologie utilisée est celle des scénarios, qui se rapproche de ce que nous avons pu effectuer durant les autres travaux pratiques. Dans notre cas, nous allons en voir trois avec chacun leurs particularités afin de pouvoir comparer les résultats dans la partie suivante. Pour chacun des scénarios suivants, nous allons donc détailler les étapes qui le composent ainsi que les spécifités qui le caractérisent (type de file d'attente, simultanéité des messages etc...) mais ils partageront tout de même une caractéristique commune, le nombre de messages envoyés : 500.

4.2.1 Baseline

Le premier scénario constitue notre baseline, les caractéristiques le concernant sont les suivantes :

- Nombre de sujets *SNS* : 1
- Nombre de file d'attente *SQS* : 1
- Type de file d'attente : Standard
- Envoi des messages : Séquentiel (l'un après l'autre depuis *boto3*)

L'envoi des messages séquentiel nous permettra d'avoir une valeur témoin pour les futurs envois parallélisés dans les scénarios suivants en terme d'exécutions concurrentes.

4.2.2 Scénario 1 : File d'attente standard

Le second scénario s'attarde sur les files d'attentes standards, mais avec un envoi parallélisé des messages cette fois-ci, les caractéristiques le concernant sont les suivantes :

- Nombre de sujets *SNS* : 5
- Nombre de file d'attente *SQS* : 1
- Type de file d'attente : Standard
- Envoi des messages : Parallélisé (Utilisation de la librairie *multiprocessing* depuis *boto3*)

De plus, en multipliant le nombre de sujets, on se rapproche de la réalité avec potentiellement une seule file d'attente pour desservir plusieurs sujets (par exemple, plusieurs artistes à des prix différents pour un même festival).

4.2.3 Scénario 2 : File d'attente FIFO

Le troisième scénario s'attarde sur les files d'attentes FIFO, avec également un envoi parallélisé des messages pour essayer d'estimer quel pourrait être les avantages/inconvénients de l'utilisation de cette technologie, les caractéristiques le concernant sont les suivantes :

- Nombre de sujets *SNS* : 5
- Nombre de file d'attente *SQS* : 1
- Type de file d'attente : FIFO
- Envoi des messages : Parallélisé (Utilisation de la librairie *multiprocessing* depuis *boto3*)

Remarque : Il aurait été intéressant de tester davantage de scénarios avec des quantités de message bien plus grandes mais dans un souci de temps et de ressources, je n'ai pas pu poursuivre. Je continuerai tout de même cette analyse en dehors du cadre du cours puisqu'il me semble que cette dernière pourrait avoir du potentiel dans le milieu professionnel.

4.3 Résultats de l'analyse (*Analysis results*)

Dans les diagrammes à barres suivants, on observe différentes métriques liées à *Lambda*, qui représentent plusieurs points d'intérêts quant à l'exécution correcte du processus.

Dans un premier temps, la Figure 2 reprend le nombre d'appel à la fonction *Lambda* servant de témoin afin de s'assurer que aucun message ne s'est perdu entre la publication depuis le SNS et le passage dans la file d'attente SQS. Les trois valeurs étant à 500 (nombre de message effectivement envoyé), nous pouvons alors noter l'absence de pertes. Cette absence de perte était certaine puisque comme anoncé dans la partie dédiée aux services AWS, peu importe le type de queue utilisé, tous les messages seront remis. Cependant, on aurait pu s'attendre à observer peut-être un message remis une seconde fois à cause de la stratégie **au moins une fois**, mise en place dans les queues standards. La faible quantité de messages envoyée est évidemment à la source de la non-observation de ce phénomène, mais même avec des tentatives avec 1 million de messages, je n'ai pas réussi à observer le phénomène.

Le diagramme suivant, celui de la Figure 3, répertorie la durée moyenne d'éxecution de tous les appels à des fonctions *Lambda* dans un intervalle de 5 minutes. On remarque donc que l'ensemble des 500 messages envoyés dans les différents topics *SNS* pour les différents scénarios ont tous été traités en moins d'une seconde par message. Effectivement, si on s'attarde dans *CloudWatch* et que l'on observe les durées maximales et minimales, les valeurs obtenues sont 1.506 et 0.167 secondes respectivement. La majorité des temps de traitement se situent entre 200ms et 400ms mais certaines données tirent la moyenne vers le haut irrégulièrement. Ce phénomène peut s'expliquer par les arrivées de premier message de certains groupes (en moyenne 1400ms). Nous comprenons donc l'intérêt potentiel de grouper intelligemment les messages pour ne pas démultiplier la création de groupes contenant des éléments uniques.

Enfin, il est intéressant de remarquer les performances légèrement meilleures en moyenne des queues standards par rapport aux FIFO qui se remarque déjà avec uniquement 500 messages. Cet écart se marque de plus en plus avec l'augmentation de la quantité de messages et s'explique par l'absence des restrictions qui sont liées à la mise en place d'une file d'attente, *premier arrivé permier sorti*. Il s'agit donc d'un choix laissé à la discrétion de l'utilisateur, gagner en performance en se

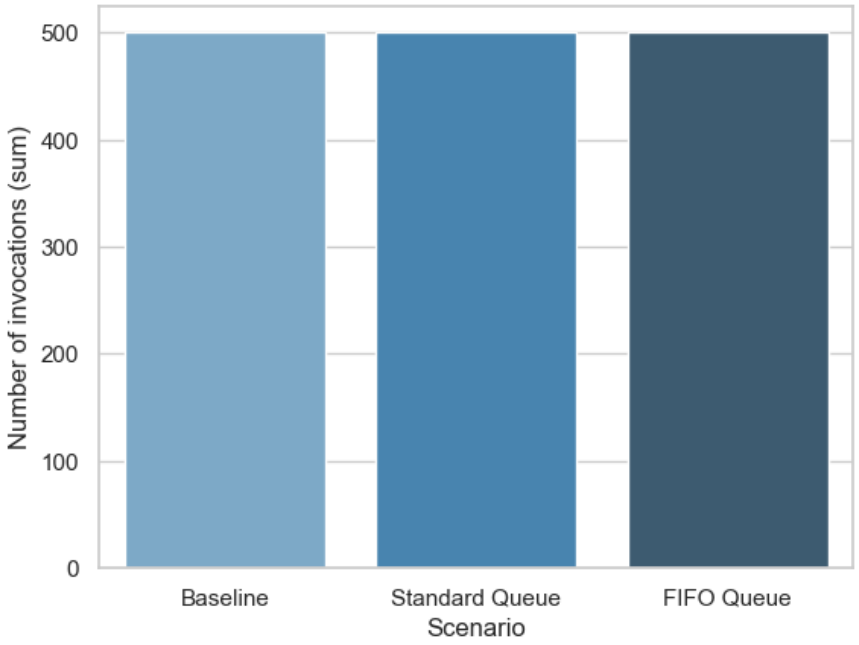

Figure 2: Nombre d'invocations de la fonction *Lambda* pour chaque scénario

détachant des contraintes FIFO ou bien s'assurer d'une réception unique et d'un ordre conservé au prix de la dégradation de performances en temps.

Le dernier diagramme, présent dans la Figure 4 est consacré au nombre moyen d'exécutions concurrentes au sein des fonctions *Lambda*. Sachant que ces deux scénarios ont été testés en faisant usage de la librairie *multiprocessing*, nous sommes capables de distinguer la différence avec une valeur témoin séquentielle, celle du scénario baseline. Effectivement, dans les deux cas, puisque la quantité de messages arrivant simultanément était plus grande, davantage d'exécutions concurrentes ont été lancées. Il est important de noter qu'on ne remarque pas pour autant de mélange dans l'ordre des messages par rapport à l'heure d'arrivée.

D'autre part, grâce aux restrictions logiques liées aux queues FIFO, on remarque à petite comme à grande échelle (testé sur 1M de messages), un léger gain quant au nombre d'éxecutions concurrentes. Ce gain n'étant pas vraiment significatif puisqu'il n'apporte pas plus d'informations quant à la qualité du traitement. En effet, il permet simplement d'indiquer que les exécutions concurrentes sont mieux gérées au sein des files d'attentes FIFO. Cette situation est tout à fait attendue puisque l'on dispose de deux outils supplémentaires pour gérer la concurrence, le *MessageDeduplicationId* et le *MessageGroupId* qui sont pris en charge par AWS pour lier/délier des messages arrivant dans une file d'attente *SQS*.

5 Instructions pour la démonstration (*Instructions for the demo*)

Les instructions pour la démonstration sont résumées dans le *README.md* du répertoire GitHub du projet, mais je vais également les reprendre ici.

- Avant de débuter la démarche, assurez vous d'avoir vos accès AWS à jour et configurez

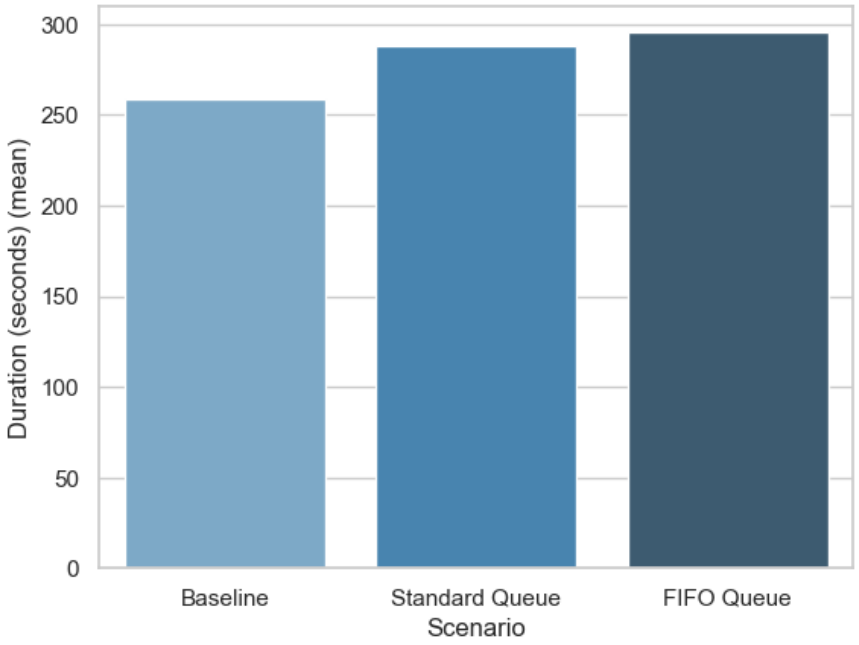

Figure 3: Durée moyenne d'exécution d'une fonction *Lambda* pour chaque scénario

correctement (*/.aws/credentials* et */.aws/config*).

- Dans un premier temps, après avoir cloné le répertoire Github disponible en cliquant sur le lien en dessous du titre du projet (ou **ici**), il faudra se placer à l'intérieur du répertoire.
- La prochaine étape consiste à construire l'image Docker permettant d'exécuter le projet directement depuis n'importe quel hôte. Pour cela, on exécute les commandes suivantes :

```
> docker build -t log8415 .
> ./run_docker.sh
> cd python/
```

- Nous sommes désormais prêts à exécuter les différents scénarios, il faudra donc utiliser le commandes suivantes en fonction des scénarios souhaités (exécutions indépendantes) :

```
> python3 baseline.py
> python3 scenario_1.py
> python3 scenario_2.py
```

- Une fois les scénarios choisis exécutés, nous sommes en capacité de générer nos métriques, pour cela, on utilise la commande suivante :

```
> python metrics.py
```

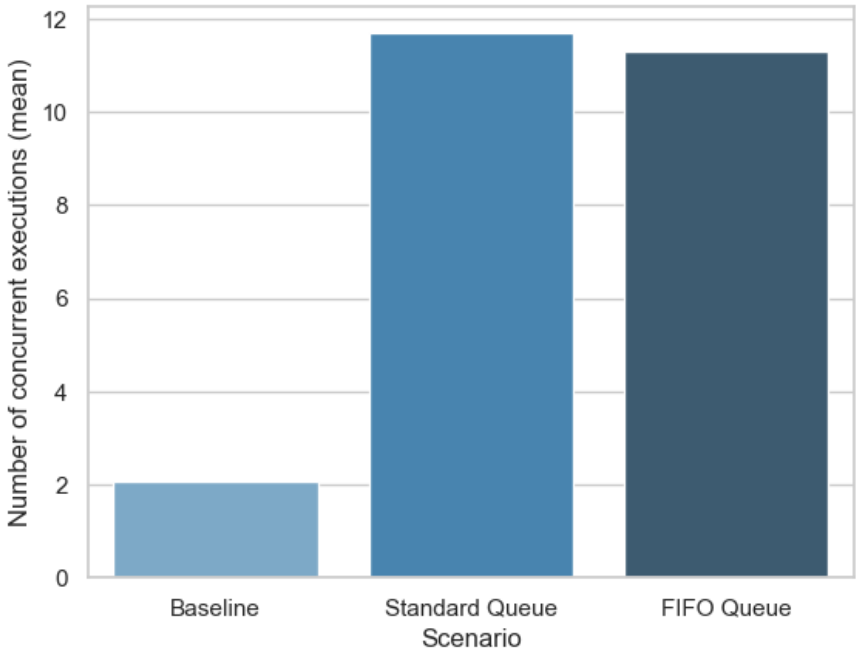

Figure 4: Moyenne du nombre d'exécutions concurrentes de la fonction *Lambda* pour chaque scénario

Il est important de noter que les graphiques sont basés sur la dernière demi-heure, sur une période de 5 minutes et contiennent 3 colonnes qui peuvent être à zéro si le scénario lié à la colonne n'a pas été exécuté.

6 Résumé (*Summary*)

Ce projet personnel m'aura permis d'en apprendre beaucoup sur bon nombre de composants du fournisseur de services d'infonuagique AWS en plus de me permettre de mettre en place une architecture faisant interagir ces derniers. L'étendue du projet, de la compréhension des rôles de chacun des intervenants à l'automatisation du déploiement m'a permis d'avoir une vue d'ensemble des communications possibles avec les technologies comme *SNS*, *SQS*, *Lambda* ou encore *DynamoDB*. C'est avec cette vue d'ensemble que j'ai alors pu concevoir une solution potentielle au problème initialement formulé, celui de la gestion des files d'attente via les technologies de l'infonuagique. Les tests menés au travers de scénarios m'ont permis d'avoir un premier aperçu des capacités de ces systèmes et comprendre pourquoi ils ont une telle place aujourd'hui dans l'industrie.

D'autre part, sur un aspect financier cette fois-ci, en utilisant l'estimateur de coût d'AWS et en se basant sur 5 millions de requêtes par mois (50 évènements à 100k utilisateurs), nous arriverions à un coût mensuel de 85 dollars pour une telle architecture. Ce coût reste extrêmement abordable pour une société qui gère habituellement des évènements à plusieurs milliers de personnes. C'est pourquoi je pense qu'il pourrait être intéressant de mettre à l'essai un tel système dans un cadre plus proche du monde professionnel.

Pour continuer sur ce qui pourrait être intéressant à l'avenir, nous pourrions implémenter par

exemple une file pour les messages non délivrés (*DLQ* dans le schéma de l'architecture) pour traiter ces derniers en conséquence. Nous pourrions également faire remonter des informations quant à des incohérences dans le comportement de boto3 comme : l'absence de la mise à jour de politique lors de la création d'abonnement entre *SNS* et SQS ou encore le non-retrait des abonnements entre *SNS* et *SQS* lors de la délétion du *SNS* qui est supposé supprimer tous les abonnements liés selon la documentation.

7 Articles utilisés

- 7.1 : V. Goswami, S. S. Patra and G. B. Mund, "Performance analysis of cloud with queue-dependent virtual machines," 2012 1st International Conference on Recent Advances in Information Technology (RAIT), 2012, pp. 357-362, doi: 10.1109/RAIT.2012.6194446.

- 7.2 : C. S. Pawar and R. B. Wagh, "Priority based dynamic resource allocation in Cloud computing with modified waiting queue," 2013 International Conference on Intelligent Systems and Signal Processing (ISSP), 2013, pp. 311-316, doi: 10.1109/ISSP.2013.6526925.

- 7.3 : Mandelbaum, A., Massey, W.A., Reiman, M.I. et al. Queue Lengths and Waiting Times for Multiserver Queues with Abandonment and Retrials. Telecommunication Systems 21, 149–171 (2002). https://doi.org/10.1023/A:1020921829517