Comparaison de plateformes de *Function as a Service*

Jovan El Hoayek et Nicolas Ouellet-Payeur
Polytechnique Montréal

***Résumé*—Function as a Service (FaaS) est un récent paradigme d'applications infonuagiques qui a gagné en popularité récemment. Il offre de bonnes caractéristiques de scalabilité et des prix avantageux. Ce travail présente une comparaison pour des prix et de la performance de plusieurs plateformes infonuagiques. Deux applications FaaS ont été construites sur trois plateformes infonuagiques : Google Cloud Functions, AWS Lambda, et Azure Functions. Des mesures de performance et de scalabilité ont été effectuées sur chacune de ces applications, puis comparées entre elles. Globalement, nous avons trouvé qu'AWS Lambda est la technologie la plus robuste et performante pour les cas d'utilisation testés. Nous espérons que ces conclusions pourront guider différentes organisations dans leur choix de plateforme infonuagique, ou servir de point de départ pour des recherches futures.**

I. Introduction

Au cours des derniers mois, Google Cloud Platform a lancé en bêta un service appelé Google Cloud Functions (GCF) [1]. Il s'agit d'un service de type Function as a Service (FaaS), un paradigme très jeune. Il suit un modèle sans serveur (serverless) pour permettre la mise à l'échelle facile et automatique du service selon la charge [2]. D'autres services similaires font compétition à GCF et existent depuis plus longtemps, comme AWS Lambda [3] et Azure Functions [4]. Amazon Web Service (AWS), Google Cloud, et Microsoft Azure, sont les trois plateformes d'infonuagique les plus populaires en ce moment [5] [6].

Nous avons réalisé une recherche visant à comparer ces différents outils, répondant ainsi à la question : comment les plateformes populaires de FaaS se comparent-elles en termes de performance et de prix ? Notre recherche pourra aider les entreprises intéressées par les services FaaS à déterminer la meilleure plateforme pour leurs besoins en leur offrant une comparaison de ces services.

Dans cet article, nous commençons par décrire le contexte de la recherche et les travaux connexes. Par la suite, nous expliquons l'approche adoptée, puis présentons les résultats obtenus. Nous discutons ensuite de ces résultats et émettons certaines recommandations.

II. Contexte

A. Architectures sans serveur

Les architectures sans serveur permettent à une plateforme infonuagique (cloud platform) de mettre à l'échelle des services automatiquement, selon la charge [7] [8]. Ceci permet aux développeurs d'opérer à un niveau d'abstraction plus élevé, où ils n'ont pas besoin de s'inquiéter des caractéristiques d'un service sous haute charge. Les principaux avantages de ce paradigme sont :

- un haut niveau de scalabilité (scalability) [7] [8] ;
- des déploiements plus simples [7] [8] ;
- un niveau d'abstraction plus élevé pour les développeurs [7] [8].

Cependant, il existe quand même des inconvénients. En effet, la mise en place de ce paradigme entraîne :

- moins de contrôle sur les détails d'implémentation [8] ;
- difficulté à changer de plateforme infonuagique si celle qu'on utilise est inadéquate [7] [8].

D'autre part, la communauté autour des technologies *serverless* est encore petite et peu développée [9]. Cela fait en sorte qu'il n'y a pas de réels standards et de modèles d'évaluation dans le domaine, et que le support communautaire est faible [9].

B. Function as a Service

Un exemple d'architecture serverless est Function as a Service (FaaS) [8] [10]. Quelqu'un qui implémente une fonction avec FaaS écrit une fonction qui peut être appelée à travers un protocole de communication, comme HTTP ou un protocole de remote procedure calls (RPC). La plateforme infonuagique gère automatiquement la configuration des serveurs à utiliser, en fonction de la charge du système [8] [10]. En somme, FaaS offre les avantages de RPC et d'une architecture sans serveur. Puisque FaaS est conceptuellement très simple, son utilisation peut être attirante pour plusieurs de projets. Comme on peut le voir dans la figure 1, les plateformes FaaS sont de plus en plus populaires.

C. Plateformes évaluées

Pour cette étude, trois plateformes on été testées : Google Cloud Functions (GCF), AWS Lambda, et Azure Functions. Elles ont été choisies car ces plateformes sont des *leaders* du marché infonuagique [5] [6]. Ces trois plateformes sont fonctionnellement très similaires. Ce sont des plateformes offertes par de grandes compagnies d'infonuagique, qui implémentent toutes une architecture FaaS pour offrir une bonne scalabilité aux utilisateurs. Les trois ont des systèmes de prix plutôt similaires, et supportent le langage de programmation JavaScript avec l'environnement Node.js [2] [3] [4].

À noter : au moment d'écrire cet article, GCF est le seul des trois à être encore en bêta. Ainsi, il se peut que les caractéristiques de performance ou de prix de GCF changent dans le futur. De plus, Azure Functions a parfois des problèmes

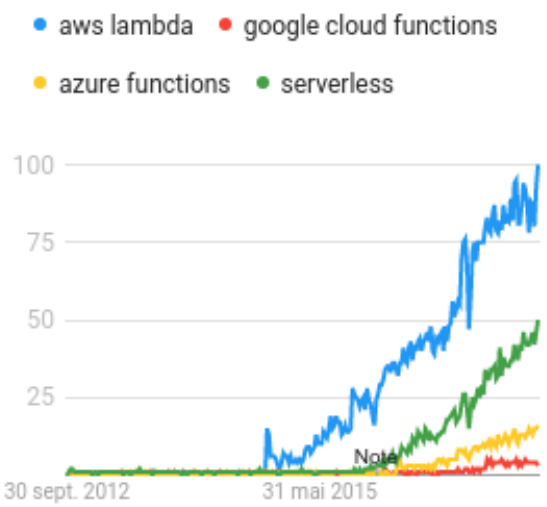

FIGURE 1. Évolution de l'intérêt de recherche pour les plateformes FaaS [11]

avec les appels HTTP faits avec Node.js [12]. Cela nous a forcé à utiliser un autre langage de programmation pour cette plateforme.

III. TRAVAUX CONNEXES

De précédentes études ont déjà été réalisées comparant les plateformes FaaS. Par exemple, McGrath et Brenner ont comparé certains critères de performance d'une architecture serverless qu'ils ont développée à AWS Lambda, Azure Functions, Google Cloud Functions et Apache OpenWhisk [13]. Ils ont examiné, entre autres, le comportement des diverses plateformes en ce qui concerne l'activation et la désactivation des conteneurs qui exécutent les fonctions. Ils ont déterminé que les conteneurs d'Azure Functions expirent après quelques minutes et ont besoin d'un certain temps avant de redémarrer, alors que AWS Lambda et Google Cloud Functions n'exhibent pas ce comportement. De plus, ils ont remarqué que, lorsque la fonction déployée ne fait rien, Azure Functions est capable de traiter, en général, plus d'exécutions de la fonction par seconde que Google Cloud Functions et AWS Lamda lorsque le nombre de requêtes concurrentes est inférieur à 10, et que AWS Lambda en traite plus que Google Cloud Functions pour ces même paramètres. Les participants au *Workshop on Serverless Computing 2017* ont trouvé que les résultats de McGrath et Brenner semblaient "erratiques" et que le modèle de comparaison qui était proposé avait besoin d'être plus développé [9].

IV. APPROCHE

A. Applications

Pour comparer la performance et le prix des diverses plateformes de FaaS, nous avons développé deux applications différentes basées sur les cas d'utilisation typiques de ces plateformes. Ces deux applications ont des caractéristiques de performance différentes, car elles n'utilisent pas les mêmes ressources.

Pour AWS Lambda et Google Cloud Functions, les deux applications sont codées en JavaScript et exécutées par l'environnement Node.js. Ceci permet d'éviter des différences de performance causées par les langages de programmation différents. Cependant, nous avons été incapables d'effectuer des requêtes HTTP avec Node.js sur Azure Functions. Les applications sur Azure Function sont donc implémentées en C#.

1) BCrypt: La première application est une fonction qui exécute BCrypt [14] sur une chaîne de de caractères et retourne le résultat. C'est une application limitée par le processeur (*CPU-bound*). BCrypt est un algorithme de hachage pour mots de passe configurable : il peut prendre plus ou moins de temps à s'exécuter. Ainsi, nous avons configuré l'algorithme pour prendre environ 2 secondes à chaque exécution, sur les trois plateformes testées (paramètre de 13 pour BCrypt). Ceci nous permet de tester la scalabilité de l'application sans nécessiter beaucoup de ressources de notre client. De plus, un temps de calcul beaucoup plus long est irréaliste, car BCrypt pourrait être utilisé au moment où un utilisateur se connecte à un système ; les utilisateurs n'aiment pas attendre plusieurs secondes pour savoir s'ils ont bien entré leur mot de passe.

Pour que l'application soit facile à déployer sur les trois plateformes choisies, nous avons utilisé une implémentation de BCrypt en JavaScript pur [15]. Puisque JavaScript présente des caractéristiques de performances différentes que C ou C++, la librairie BCrypt utilisée est un peu plus lente qu'une implémentation native de BCrypt : environ 30% plus lente [16]. Sur Azure, à cause des problèmes de la plateforme avec Node.js [12], nous avons plutôt utilisé une librairie C#, BCrypt.Net [17].

L'application reçoit des requêtes HTTP POST, au format JSON. Chaque requête contient une chaîne de caractères à passer à la librairie BCrypt. Le corps d'une requête peut donc ressembler à ceci, si un client veut hacher la chaîne `"hello world"` :

```
{
  "message": "hello world"
}
```

La fonction calcule le hachage BCrypt pour la chaîne de caractères (ce qui prend environ 2 secondes sur les plateformes testées), puis retourne directement le résultat dans le corps de sa réponse HTTP.

2) ShoutCloud: La seconde application est une fonction qui effectue des appels à un service à travers HTTP. C'est une application limitée par le réseau (*network-bound*). Chaque fois que la fonction est appelée, elle fait un appel à l'API public de ShoutCloud. ShoutCloud [18] est une API qui permet de convertir des chaines de caractères en majuscules. Bien cette opération soit simple et presque triviale, ShoutCloud est très *scalable* [18] et nous permet de simuler des cas d'utilisation *network-bound*.

L'application reçoit des requêtes HTTP POST, au même format que l'application BCrypt. Elle convertit cette chaîne de caractères en un format JSON que ShoutCloud accepte,

puis envoie à ShoutCloud une requête POST dont le corps ressemble à ceci :

```
{
  "INPUT": "hello world"
}
```

ShoutCloud renvoie une réponse JSON qui comporte, entre autres, la chaîne de caractères en majuscules. Le corps de la réponse envoyée par ShoutCloud ressemble à ceci :

```
{
  "INPUT": "hello world",
  "OUTPUT": "HELLO WORLD"
}
```

L'application extrait cette chaîne du JSON (`hello world` dans cet exemple), et l'utilise comme corps de la réponse HTTP. L'opération complète prend, à faible charge, entre 50 et 200 millisecondes sur les plateformes testées.

Le code des applications est disponible ici : https://github.com/kpwbo/comparing-FaaS

B. Client

L'outil utilisé pour les mesures de performances est Locust [19], un outil de test de charge (*load testing*) pour les sites web. Locust appelle les fonctions en effectuant des requêtes HTTP en parallèle. Il est configuré avec un nombre d'utilisateurs (n), qui correspond au nombre de requêtes simultanées à effectuer. Dès qu'une réponse est reçue, Locust effectue une nouvelle requête, pour que le nombre de requêtes simultanées soit constant. Nous avons arrêté les tests après $(100 \times n)$ appels de la fonction par utilisateur, pour chaque valeur de n testée et pour chaque plateforme. Une limite de 5 minutes est mise à chaque exécution, pour éviter que les tests s'éternisent.

Locust a été exécuté sur une machine virtuelle hébergée par DigitalOcean [20], une plateforme infonuagique tierce. Ceci nous permet d'avoir une machine dédiée aux tests avec une bonne bande passante, tout en simulant des requêtes HTTP venant de l'extérieur. Cette machine est assez puissante pour continuellement envoyer 1000 requêtes HTTP concurrentes avec Locust. Puisque la machine virtuelle est hébergée sur le nuage, les spécifications précises du matériel nous sont inconnues. Voici les spécifications connues de la machine virtuelle :

- Processeur : Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz (2 cœurs réservés).
- Mémoire : 2 Gio.
- Stockage : Disque SSD de 40 Gio.

Le code du client est disponible ici : https://github.com/kpwbo/comparing-FaaS

C. Métriques

Pour chacune de ces deux applications et pour chaque plateforme de FaaS, un test de performance a été effectué. Nous avons simulé l'utilisation « simultanée » de la fonction par un certain nombre d'utilisateurs (1, 10, 100, 1000), et nous avons mesuré, pour chaque cas, le temps médian pour complètement traiter la requête d'un utilisateur, incluant l'envoi de la requête et la réception de la réponse. Nous avons de plus noté le nombre d'échecs de traitement. La limite de 1000 pour le nombre d'utilisateurs a été choisie empiriquement, en observant que taux d'échec augmente de façon significative après ce seuil. Cela pourrait être dû à la limite de bande passante de la machine qui exécute Locust.

Par la suite, pour chaque application, nous avons utilisé les grilles tarifaires de chaque plateforme ainsi que les données obtenues lors du test de performance pour estimer les coûts mensuels, en dollars canadiens, de l'utilisation de la plateforme, et ce, dépendamment du nombre d'utilisateurs « simultanés » de l'application, que nous avons assumé constant au cours du mois.

Voici les ressources utilisées par chacune des plateformes. Il s'agit d'un quantité de mémoire et de la vitesse du processeur réservés à la fonction. La plupart des spécifications de ces machines nous sont inconnues.

- Google Cloud Functions : 1024 Mio de mémoire, processeur 1,4 GHz.
- AWS Lambda : 1024 Mio de mémoire, vitesse de processeur inconnue.
- Azure Functions : 128 Mio de mémoire, vitesse de processeur inconnue.

Nous avons utilisé ces paramètres pour réduire les incertitudes. En effet, GCF et AWS Lambda offrent une vitesse de processeur similaire à la quantité de mémoire réservée. Pour Azure Functions, par contre, la quantité de mémoire réservée n'affecte pas la vitesse du processeur. L'utilisateur ne peut pas non plus fixer lui-même la quantité de mémoire utilisée, et elle est arrondie vers le haut par tranches de 128 Mio (128 mibioctets). Puisque nos deux applications utilisent peu de mémoire, nous avons observé que la mémoire utilisée s'arrondit à 128 Mio, pour Azure Functions.

Les processeurs pour chaque plateforme pourraient avoir des vitesses très différentes. Cela doit être pris en compte dans notre analyse. En effet, le temps de calcul de base d'une fonction (mesuré avec $n = 1$) pourrait être plus élevé sur certaines plateformes que sur d'autres.

Pour mesurer les prix, nous avons d'abord consulté les tables de prix de GCF [21], AWS Lambda [22] et Azure Functions [23]. Sur les trois plateformes, le coût augmente de façon linéaire avec le temps d'exécution de la fonction, les ressources allouées à la fonction, et le nombre de fois qu'elle est appelée. On peut donc estimer le coût à l'aide des mesures de performance effectuées avec Locust.

Un coût additionnel est aussi rencontré : celui pour l'utilisation du réseau sortant (*outgress*). En effet, les trois plateformes chargent aussi un certain montant par Gio sortant [21] [24] [25]. Nous avons approximé le nombre de Gio avec deux composantes :

- La réponse HTTP envoyée par le serveur. Nous avons mesuré la taille de la réponse HTTP sur chacune des trois plateformes, en comptant les *headers* HTTP, et l'avons multiplié par le nombre de requêtes mensuelles.

TABLE I
PRIX UNITAIRES DE CHAQUE PLATEFORME [21] [22] [23] [24] [25]

Métrique	AWS Lambda	GC Function	Azure Function
1 requête ($)	0,0000002	0,0000004	0,0000002
1 GHz-seconde ($)	0	0,00001	0
1 Gio-seconde ($)	0,00001667	0,0000025	0,000016
1 Gio sortant ($)	0,09	0,12	0,087

TABLE II
QUANTITÉ OFFERTE GRATUITEMENT PAR CHAQUE PLATEFORME [21] [22] [23] [24] [25]

Métrique	AWS Lambda	GC Function	Azure Function
Requêtes	1000000	2000000	1000000
GHz-secondes	0	200000	0
Gio-secondes	400000	400000	400000
Gio sortants	1	5	5

— Pour l'application ShoutCloud, la requête envoyée à ShoutCloud. Nous avons effectué des tests qui envoient des requêtes HTTP identiques, mais à un autre serveur. Ce serveur mesure la taille des requêtes qu'il reçoit. Cette taille est ensuite multipliée par le nombre de requêtes mensuelles.

La table I présente les prix unitaires de chaque paramètre pour chaque plateforme de FaaS pour notre configuration. Les montants de la table I sont donnés en dollars américains. D'autre part, chaque plateforme offre un certain nombre de requêtes, de temps d'exécution et de données sortantes gratuitement. La table II montre ces quantités.

Les unités utilisées pour calculer le prix sont :

— Requêtes : chaque appel de la fonction compte comme une requête.
— GHz-secondes : la fréquence du processeur, en GHz multipliée par le nombre de secondes d'exécution de la fonction. Seul GCF utilise cette mesure.
— Gio-secondes : la quantité de mémoire réservée pour la fonction, multipliée par le nombre de secondes d'exécution de la fonction.
— Gio sortants : la quantité de données envoyée par le serveur vers des clients ou des services tiers.

Il est à noter que, pour calculer le nombre de GHz-seconde et de Gio-secondes de l'exécution de la fonction, chaque plateforme possède des règles différentes. AWS Lambda et GCF arrondissent le nombre de secondes à 100 ms, toujours vers le haut. Donc, si une fonction qui utilise 1 GHz et 1 Mio s'exécute en 130ms, elle sera facturée pour 0,2 GHz-seconde et 0,2 Gio-seconde. Azure Functions, quant à lui, arrondit vers le haut à la milliseconde près. Donc, 130 ms correspondent à 0,13 Gio-seconde pour Azure.

Azure est la seule plateforme qui nous empêche de réserver une quantité de mémoire fixe pour la fonction. La quantité de mémoire réelle utilisée par la fonction sert à déterminer le nombre de Gio-secondes. Elle est arrondie vers le haut au 128 Mio près.

TABLE III
TEMPS DE RÉPONSE MÉDIAN DE BCRYPT (MS)

n	AWS	GCF	Azure
1	1400	2400	2616
10	1800	2400	20000
100	1800	2500	113000
1000	2200	2500	168000

TABLE IV
TAUX D'ÉCHEC DE BCRYPT (%)

n	AWS	GCF	Azure
1	0,00	0,00	0,00
10	0,00	0,00	0,00
100	0,00	1,08	25,29
1000	11,70	3,79	54,12

TABLE V
PRIX MENSUEL DE BCRYPT ($)

n	AWS	GCF	Azure
1	50,08	92,97	6,54
10	724,55	963,88	966,53
100	7 308,14	10 076,33	54 725,75
1000	89 275,26	100 802,89	813 432,82

Après avoir estimé les coûts mensuels de notre application, nous les avons validés. Nous avons utilisé les outils de calcul de prix de Google Cloud [26], AWS [27] [28] et Azure [29]. Nous avons également utilisé un outil d'un tiers, Serverless-Calc [30] pour comparer les prix des trois plateformes.

V. RÉSULTATS

Les tables III et VI présentent le temps de réponse médian d'une requête faite aux applications BCrypt et ShoutCloud, respectivement, selon le nombre d'utilisateurs concurrents et la plateforme de FaaS utilisée. Les figures 2 et 5 représentent ces données sous la forme de graphiques.

Les tables IV et VII, quant à elles, montrent le taux d'échec des requêtes faites aux applications BCrypt et ShoutCloud, respectivement, selon le nombre d'utilisateurs concurrents et la plateforme de FaaS utilisée. Les figures 3 et 6 représentent ces données sous la forme de graphiques.

D'autre part, les tables V et VIII présente l'évolution du prix des plateformes de FaaS selon le nombre d'utilisateurs concurrents pour BCrypt et ShoutCloud, respectivement. Les figures 4 et 7 représentent ces données sous la forme de graphiques.

Les données obtenues sont disponibles ici : https://github.com/kpwbo/comparing-FaaS

A. BCrypt

Pour une application *CPU-bound* comme BCrypt, AWS Lambda semble avoir un bon temps de réponse médian, et ce, peu importe le nombre d'utilisateurs concurrents. De plus, son taux d'échec est nul pour un nombre d'utilisateurs inférieur

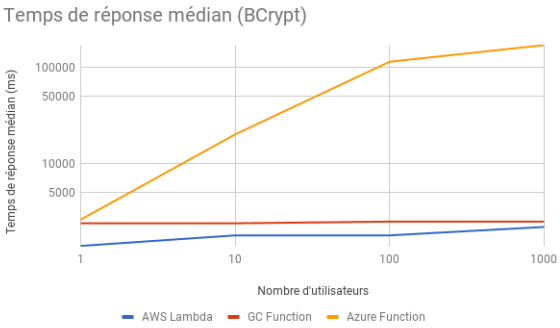

FIGURE 2. Temps de réponse médian de BCrypt

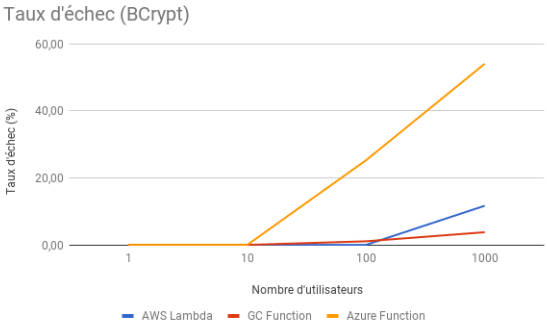

FIGURE 3. Taux d'échec de BCrypt

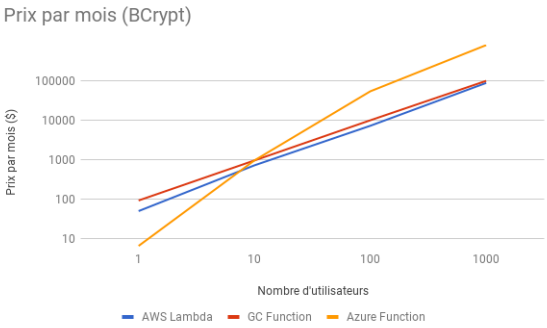

FIGURE 4. Prix mensuel de BCrypt

à 1000, mais augmente de manière significative pour 1000 utilisateurs. Le prix mensuel de la plateforme pour cette application semble croître selon le nombre d'utilisateurs de manière linéaire.

D'un autre côté, pour BCrypt, le temps de réponse médian de Google Cloud Functions ne varie presque pas avec l'augmentation du nombre d'utilisateurs concurrents. Son taux d'échec, quant à lui, augmente légèrement lorsque le nombre d'utilisateurs concurrents devient élevé. Le prix mensuel évolue linéairement avec le nombre d'utilisateurs concurrents.

Pour ce qui est d'Azure Functions, le temps de réponse médian de BCrypt augmente significativement lorsque le nombre d'utilisateurs concurrents augmente, même lorsque ce

TABLE VI
TEMPS DE RÉPONSE MÉDIAN DE SHOUTCLOUD (MS)

n	AWS	GCF	Azure
1	70	110	120
10	71	110	120
100	140	120	180
1000	67	1100	1600

TABLE VII
TAUX D'ÉCHEC DE SHOUTCLOUD (%)

n	AWS	GCF	Azure
1	0,00	0,00	0,00
10	0,00	0,10	0,00
100	0,00	7,33	0,00
1000	0,00	56,39	0,00

TABLE VIII
PRIX MENSUEL DE SHOUTCLOUD ($)

n	AWS	GCF	Azure
1	0,29	5,15	0,28
10	39,22	86,37	5,45
100	858,07	903,30	140,86
1000	4 610,50	44 997,75	8 342,42

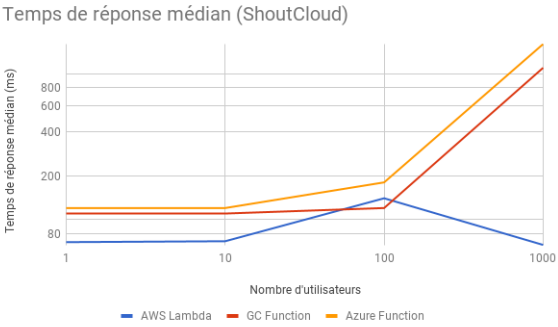

FIGURE 5. Temps de réponse médian de ShoutCloud

nombre est bas. De plus, le taux d'échec devient très élevé au point où la majorité des requêtes échouent. L'augmentation du prix mensuel est importante lorsque le nombre d'utilisateurs concurrents est faible, mais diminue plus ce dernier augmente.

B. *ShoutCloud*

Pour AWS, le temps de réponse médian est très bon pour l'application *network-bound*. De plus, le temps de réponse demeure constant et ne semble pas être affecté par le nombre d'utilisateurs concurrents. Au niveau du prix, AWS est idéal pour les grands nombres d'utilisateurs, car le prix augmente de façon essentiellement linéaire.

En ce qui concerne Google Cloud Functions, le temps de réponse peut varier beaucoup selon le nombre d'utilisateurs. GCF montre aussi un très haut taux d'échec quand le nombre d'utilisateurs est élevé. Au niveau du prix, GCF est mauvais :

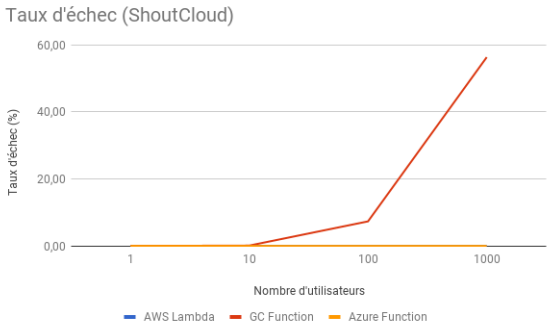

FIGURE 6. Taux d'échec de ShoutCloud

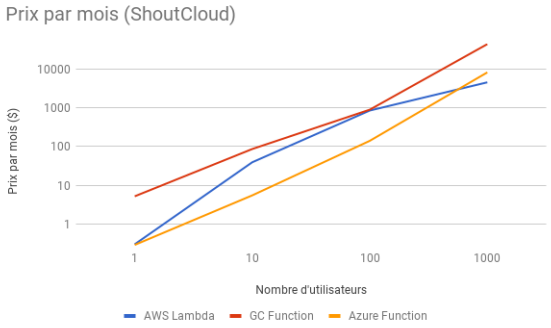

FIGURE 7. Prix mensuel de ShoutCloud

prix de base élevé qui augmente beaucoup quand le nombre d'utilisateurs augmente.

Finalement, Azure Functions démontre des caractéristiques de performance variables. Il est un peu lent, surtout avec plus de 100 utilisateurs, mais a un taux d'échec nul, même lorsque le nombre d'utilisateurs est élevé. Puisque Azure Functions ralentit avec un grand nombre d'utilisateurs, le coût peut monter en flèche. Malgré tout, le prix mensuel est très avantageux s'il y a moins de 1000 utilisateurs.

VI. DISCUSSION

A. Analyse des résultats

1) Comparaison des plateformes: Nous pouvons comparer les trois plateformes de FaaS avec les résultats que nous avons obtenus. Pour l'application BCrypt, AWS Lambda est la plateforme la plus performante. En effet, son temps de réponse médian est le plus bas des trois plateformes, et ce, peu importe le nombre d'utilisateurs concurrents. D'autre part, AWS Lambda est aussi la plateforme la plus fiable en général, avec un taux d'échec nul pour tous les nombres d'utilisateurs concurrents sauf 1000. Pour ce dernier, c'est Google Cloud Function qui est la plus fiable. De son côté, Azure Function présente, de loin, la pire performance et la pire fiabilité. En ce qui concerne le prix, toutefois, Azure Function est, de loin, la meilleure plateforme, mais seulement pour 1 utilisateur concurrent. À 10 utilisateurs concurrents ou plus, AWS offre le meilleur prix. Si nous avions à classer les trois plateformes de manière générale pour les applications *CPU-bound*, nous dirons que AWS Lambda est la meilleure plateforme, suivie de Google Cloud Function, puis Azure Function.

Pour l'application ShoutCloud, AWS Lambda offre le meilleur temps de réponse médian pour tous les nombres d'utilisateurs concurrents sauf pour 100, où Google Cloud Function offre mieux. AWS Lambda et Azure Function présentent toutes deux une excellente fiabilité avec un taux d'échec nul peu importe le nombre d'utilisateurs concurrents, alors que Google Cloud Function devient de moins en moins fiable plus le nombre d'utilisateurs concurrents augmente. Pour ce qui est du prix, Azure Function coûte le moins cher pour tous les nombres d'utilisateurs sauf 1000. Pour ce dernier, AWS Lambda offre le meilleure prix. Notre classement général des trois plateformes pour les applications *network-bound* est le suivant : AWS Lambda, Azure Function, Google Cloud Function.

2) Choisir une plateforme de FaaS: L'observation des résultats nous permet de dégager plusieurs conclusions. Une entreprise pourrait utiliser ces résultats pour choisir la meilleure plateforme de FaaS pour ses besoins, en sachant le nombre d'utilisateurs concurrents qu'elle prévoit avoir ainsi que le type de fonction qu'elle souhaite déployer. Par exemple, si l'on prévoit avoir 100 utilisateurs concurrents de la fonction et que celle-ci est *network-bound* (comme notre application ShoutCloud), on peut observer que :

- AWS Lambda fournit un temps de réponse correct et un taux d'échec nul, mais à un prix élevé.
- GC Function possède le meilleur temps de réponse parmi les trois plateformes, mais présente le pire taux d'échec et le prix le plus élevé.
- Azure Function offre un prix qui est de loin le moins élevé des trois services ainsi qu'un taux d'échec nul, mais son temps de réponse est le pire.

Ces observations permettent de choisir la meilleure plateforme dépendamment des contraintes budgétaires et techniques spécifiques. Si le budget est limité, Azure Function semble être la meilleure plateforme dans ce cas-ci. Sinon, si la fiabilité de l'application est un requis important, AWS Lambda peut faire l'affaire, alors que si on recherche une performance élevée, on choisirait GC Function.

En plus de regarder la performance et le prix des plateformes pour le nombre d'utilisateurs concurrents prévus, on peut voir ce qui se passerait si ce nombre augmente ou diminue. En reprenant l'exemple précédent, si on prévoit que le nombre d'utilisateurs concurrents baissera à 10, on remarque que :

- AWS Lambda remplace GC Function comme plateforme la plus performante.
- Le taux d'échec de GC Function devient minime.
- Azure Function reste le plus abordable en termes de prix, mais il l'est encore plus qu'avant.

Ainsi, si on prévoit une baisse ou une hausse de l'utilisation avec le temps, on pourra observer les changements de la performance et du prix des trois plateformes et en tenir compte dans le choix de la meilleure plateforme pour l'application

que l'on veut déployer, afin d'éviter le plus possible les coûts reliés aux changements de plateforme.

3) Données aberrantes: On peut remarquer une donnée aberrante pour l'application *CPU-bound*. En effet, avec 1000 utilisateurs, BCrypt semble s'exécuter 76 fois plus lentement sur Azure que sur AWS. Cela est probablement dû à un défaut (*bug*) dans le contrôleur d'Azure Functions [31]. Ce défaut empêche Azure Functions de se mettre à l'échelle automatiquement quand une fonction *CPU-bound* est déclenchée par une requête HTTP. Ainsi, la fonction s'exécute sur une seule machine virtuelle plutôt que de diviser la charge de travail sur plusieurs machines virtuelles.

Donc, jusqu'à la correction de ce défaut dans le contrôleur d'Azure Functions, il ne faudrait jamais utiliser Azure Functions quand on a besoin de scalabilité pour une fonction *CPU-bound* déclenchée par HTTP. Cependant, Azure performe bien dans le cas d'utilisation *network-bound*, et performe peut-être mieux dans un cas d'utilisation *CPU-bound* qui est déclenché par autre chose qu'une requête HTTP. Par exemple, une fonction peut être déclenchée par une queue ou un ensemble d'événements (*Event Hub*).

De plus, Azure Functions facture la mémoire réelle utilisée, plutôt que la mémoire réservée. Ainsi, Azure Functions coûte moins cher du point de vue de la mémoire. Puisque les deux applications testées utilisent très peu de mémoire, Azure pourrait être avantageux au niveau du prix si ce n'était du temps d'exécution gonflé par le défaut observé dans le cas d'utilisation *CPU-bound*.

4) Comparaison des résultats avec la littérature: Nous pouvons comparer nos résultats avec ceux trouvés dans la littérature. McGrath et Brenner [13], tel que vu plus haut, avaient comparé, entre autres, les performances des trois plateformes que nous avons utilisées. Ils avaient remarqué qu'Azure Functions était plus performant que AWS Lambda et qu'AWS Lambda était plus performant que Google Cloud Functions, pour une fonction vide et un nombre d'utilisateurs concurrents d'au plus 10. Or, pour nos deux applications, cela n'est pas entièrement le cas. En effet, pour BCrypt et ShoutCloud, en général, AWS Lambda est plus performant que Google Cloud Functions, et Google Cloud Functions est plus performant qu'Azure. Ainsi, nos résultats sur la comparaison entre AWS Lambda et Google Cloud Functions concordent avec ceux de McGrath et Brenner, mais nos résultats sur la comparaison entre Azure Functions et les autres plateformes ne concordent pas. Cela peut s'expliquer par diverses raisons, comme le fait que McGrath et Brenner ont alloué moins de mémoire, et donc un processeur moins puissant, à AWS Lambda et Google Cloud Functions que nous leur avons alloué, ce qui réduit la performance de ces derniers et permet à Azure Functions d'être plus performant qu'eux.

B. Limites de l'expérience

1) Budget limité: Le budget alloué à notre recherche était peu élevé. En effet, nous n'avions, au total, que quelques dollars canadiens à dépenser. Or, de l'argent est nécessaire pour héberger le client ainsi que pour rouler les tests sur les applications sur les trois plateformes. Nous avons donc dû limiter notre recherche sur plusieurs aspects.

Par exemple, nous n'avons considéré qu'un petit nombre d'utilisateurs concurrents. Pour plusieurs applications, un nombre d'utilisateurs concurrents plus élevé que 1000 est possible. Ainsi, nous prévoyions réaliser des tests pour 10000 et 100000 utilisateurs concurrents, mais, vu le budget limité, cela n'a pas été possible.

De plus, nous voulions calculer la moyenne de trois essais pour chacune des données pour réduire l'incertitude, mais cela plus que triplerait nos coûts et nécessiterait donc un budget plus important.

D'autre part, notre budget a limité nos options en termes de clients. Nous voulions utiliser Locust dans un système distribué pour réduire l'effet de la bande passante limitée sur nos tests. Cependant, cela nécessiterait un nombre de machines que nous ne pouvions pas nous permettre.

2) Temps de latence: Un facteur important dans la mesure de la performance des applications est le temps de latence entre le client et les instances de la fonction. En effet, la fonction n'est pas nécessairement toujours exécutée au même endroit, même si elle est déployée sur la même plateforme. Les fonctions sont plutôt déployées dans une région spécifique (par exemple, le sud des États-Unis). Cela signifie que le temps de latence varie d'une instance à l'autre ce qui impacte les mesures de temps de réponse. Pour réduire cet effet, nous avons tenté de choisir les mêmes régions pour les trois plateformes.

Un autre problème relié au temps de latence est que nous l'avons inclus dans le temps d'exécution des fonctions pour le calcul des prix. Or, pour toutes les plateformes de FaaS utilisées, le temps d'exécution des fonctions est calculé sans le temps de latence. Cela a pour effet de systématiquement gonfler les prix. Pour y remédier, il faudrait mesurer le temps de latence pour chaque donnée et s'assurer de la soustraire du temps d'exécution pour le calcul des prix.

3) Cas d'utilisation limités: Nous n'avons testé que deux cas d'utilisation différents :

— Limité par le processeur (*CPU-bound*).
— Limité par la latence du réseau (*latency-bound*).

Mais il y a plusieurs autres facteurs qui peuvent limiter la capacité des serveurs. Ils pourraient exhiber des caractéristiques de performance et de prix différentes. Il est donc possible que les conclusions à tirer soient différentes pour ces cas d'utilisation. Voici des exemples de cas d'utilisation différents :

— Limité par la quantité de mémoire (*memory-heavy*).
— Limité par les entrées/sorties (*IO-bound*), par exemple en utilisant une base de données ou des fichiers.
— Limité par la bande passante (*bandwidth*) du réseau.

4) ShoutCloud: Un des facteurs limitants pour la performance de l'application *network-bound* est l'utilisation du service ShoutCloud. Par exemple, nous ne savons pas où ShoutCloud est hébergé. Il est possible que ce soit géographiquement plus proche des serveurs d'Amazon que nous avons utilisé, voire même que les serveurs de ShoutCloud

soient hébergés chez Amazon. Cela expliquerait pourquoi AWS Lambda offre une meilleure performance que les autres plateformes pour l'application qui utilise ShoutCloud.

Aussi, chaque API public a des propriétés différentes. Il est possible qu'une API publique ne soit pas suffisamment *scalable* pour justifier l'utilisation de FaaS. Il se peut aussi qu'Azure Functions soit meilleure qu'AWS Lambda pour certains cas d'utilisation, si l'API public est aussi hébergé par Azure. Il est donc difficile de généraliser nos résultats à tous les cas d'utilisation qui sont *network-bound*.

Pour un cas d'utilisation réel qui fait appel à une API publique, il serait donc idéal d'effectuer des mesures sur le véritable API qu'on souhaite utiliser. Il est possible que l'API qu'on souhaite utilisé soit hébergé sur une des trois plateformes testées, ou que les serveurs de l'API publique soit plus proches géographiquement de ceux d'une des trois plateformes. Ceci peut affecter le temps de réponse et le prix de l'application.

5) Déclencheurs non-HTTP: Pour toutes les plateformes que nous avons testées, nous avons utilisé des déclencheurs HTTP, c'est-à-dire qu'une ressource devait être accédée par HTTP pour lancer la fonction. Toutes les plateformes testées, toutefois, proposent d'autres sortes de déclencheurs, la plupart étant interne à la plateforme. Par exemple, Google Cloud Function permet d'avoir des déclencheurs liés à son service de stockage (Google Cloud Storage). Il se peut que ces déclencheurs soient plus efficaces que les déclencheurs HTTP (notamment dans le cas d'Azure Function), car ils sont mieux intégrés aux autres services, ou peut-être moins efficaces. Cela affecte notre interprétation des résultats.

6) Langages de programmation: Azure Functions a démontré plusieurs défauts avec l'environnement JavaScript Node.js. Initialement, nous planifiions d'utiliser le même langage de programmation sur les trois plateformes, pour comparer de façon équitable la performance. JavaScript est le seul langage supporté par les trois plateformes, et semble être un bon choix. Cependant, des défauts critiques nous empêchent d'effectuer des mesures utiles avec Azure Functions et Node.js, particulièrement pour l'application *network-bound*.

Notre solution a été d'utiliser C# plutôt que JavaScript pour Azure Functions. Puisque les environnements C# et JavaScript ont des caractéristiques de performance différentes [32], les résultats peuvent être biaisés.

De plus, Node.js est un environnement à un seul fil d'exécution (*single-threaded*) [33], alors que C#/.NET est un environnement à plusieurs fils d'exécution [34]. Il est donc possible qu'ils démontrent des caractéristiques différentes de mise à l'échelle (*scaling*).

Nous n'avons pas tenté de compenser pour les différences de performance et de mise à l'échelle dues aux langages de programmation.

7) Autres aspects des plateformes: Les trois plateformes d'infonuagique analysées ne sont pas seulement des plateformes de FaaS. Elles offrent bien sûr beaucoup d'autres produits, qui peuvent s'intégrer avec leur offre de FaaS. Cette intégration avec les autres produits de la plateforme peut apporter de la valeur à chacune des trois plateformes infonuagiques.

Au meilleur de nos connaissances, ces produits possèdent des caractéristiques qui varient d'une plateforme à l'autre. Puisque nous n'avons analysé qu'un seul produit de ces plateformes, notre comparaison est nécessairement incomplète. Azure est mauvais pour le cas d'utilisation *CPU-bound*, mais une organisation pourrait par exemple vouloir porter un projet C#/.NET existant, qui ne fonctionne que sur Windows. Dans ce cas-là, Azure Functions pourrait être avantageux, car il est hébergé sur Windows plutôt que Linux [35].

Beaucoup d'autres aspects doivent entrer dans le choix d'une plateforme, car c'est souvent tout un écosystème qu'on choisit, et non un seul aspect.

VII. Conclusion

En conclusion, les plateformes de FaaS sont une technologie prometteuse qui permet de créer des systèmes hautements scalables facilement. Notre objectif de recherche était de répondre à la question suivante : comment les plateformes populaires de FaaS se comparent-elles en termes de performance et de prix ?

Nous avons, pour ce faire, mesuré le temps de réponse médian, le taux d'échec et le prix des diverses plateformes pour un nombre d'utilisateurs concurrents et une application donnés. Ces métriques permettent de comparer les plateformes de FaaS objectivement et de soutenir le choix de la plateforme la plus appropriée pour un cas d'utilisation donné. Nous avons comparé ces métriques pour deux applications particulières, BCrypt (*CPU-bound*) et ShoutCloud (*network-bound*), déployées sur trois plateformes de FaaS, AWS Lambda, Google Cloud Functions et Azure Functions, pour un nombre d'utilisateurs concurrents variant entre 1 et 1000.

Nous pouvons alors répondre à notre question de recherche : nous avons déterminé que, en général, AWS Lambda est la meilleure plateforme, suivie de Google Cloud Functions puis d'Azure Functions. En termes de performance, fiabilité et prix, les plateformes se comparent ainsi :

- AWS Lambda :
  - La meilleure performance pour les 2 applications.
  - Taux d'échec minime pour l'application *CPU-bound*, nul pour l'application *network-bound*.
  - Le prix le plus avantageux pour les 2 applications.
- Google Cloud Functions :
  - Bonne performance pour l'application *CPU-bound*, et médiocre bonne pour l'applicaton *network-bound*.
  - Taux d'échec minime pour l'application *CPU-bound*, mais beaucoup plus élevé pour l'application *network-bound*.
  - Prix similaire à AWS Lambda pour l'application *CPU-bound*, mais beaucoup plus élevé que les autres pour l'application *network-bound*.
- Azure Functions :
  - Scalabilité inexistante pour l'application *CPU-bound*. Performance médiocre pour l'application *network-bound*.

— Taux d'échec extrême pour l'application *CPU-bound*. Taux d'échec nul pour l'application *network-bound*.
— Prix excessif pour l'application *CPU-bound*. Prix plus raisonnable pour l'application *network-bound.*

Si on accorde un poids égal à la performance, la fiabilité et le prix, voici le classement des plateformes, de la meilleure plateforme à la pire, pour chaque application testée :

— BCrypt (*CPU-bound*) :
  1) AWS Lambda
  2) Google Cloud Functions
  3) Azure Functions
— ShoutCloud (*network-bound*) :
  1) AWS Lambda
  2) Azure Functions
  3) Google Cloud Functions

Il est à noter que seulement deux cas d'utilisation ont été testés, et que l'utilisation réelle qu'une de ces plateformes peut être différente des deux applications testées. De plus, il est possible que les caractéristiques de performance de ces plateformes changent dans le futur, ce qui pourrait changer les résultats obtenus avec les mêmes tests. De plus, les trois plateformes étudiées offrent un écosystème complet en plus de leur plateforme de FaaS. Il y a donc beaucoup plus d'éléments à prendre en compte dans le choix d'une plateforme.

Pour de futures recherches sur le sujet, nous recommandons d'améliorer et d'adapter la comparaison de plateformes FaaS en corrigeant ses failles et ses défauts et d'évaluer d'autres types d'applications, comme les applications *bandwidth-bound*. Nous suggérons aussi aux entreprises d'effectuer des mesures semblables pour soutenir leurs choix de plateformes de FaaS.

RÉFÉRENCES

[1] J. Polites, "Building serverless applications with Google Cloud Functions," communication présentée à Google Cloud Next '17, San Francisco, 8-10 mars 2017, [En ligne]. Disponible : https://www.youtube.com/watch?v=kXk78ihBpiQ

[2] J. Polites. (2017) Google Cloud Platform Blog : Google Cloud Functions : a serverless environment to build and connect cloud services. [En ligne]. Disponible : https://cloudplatform.googleblog.com/2017/03/Google-Cloud-Functions-a-serverless-environment-to-build-and-connect-cloud-services_13.html

[3] J. Barr. (2014) AWS Lambda – Run Code in the Cloud. [En ligne]. Disponible : https://aws.amazon.com/blogs/aws/category/aws-lambda/

[4] N. Mashkowski. (2016) Introducing Azure Functions | Blog | Microsoft Azure. [En ligne]. Disponible : https://azure.microsoft.com/en-us/blog/introducing-azure-functions/

[5] H. Schlosser. (2017) Technology trends 2017 : These are the most popular cloud platforms - JAXenter. [En ligne]. Disponible : https://jaxenter.com/technology-trends-2017-these-are-the-most-popular-cloud-platforms-132384.html

[6] A. Patrizio. (2017) Top 50 Cloud Companies - Datamation. [En ligne]. Disponible : https://www.datamation.com/cloud-computing/cloud-computing-companies.html

[7] I. Baldini et al., « Serverless Computing : Current Trends and Open Problems », CoRR, vol. abs/1706.03178, p. 1-20, 10 juin 2017. [En ligne]. Disponible : https://arxiv.org/pdf/1706.03178.pdf

[8] M. Fowler. (2016) Serverless Architectures. [En ligne]. Disponible : https://martinfowler.com/articles/serverless.html

[9] G. C. Fox *et al.*, « Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research », communication présentée à First International Workshop on Serverless Computing (WoSC) 2017, Atlanta, Géorgie, États-Unis d'Amérique, 5 juin 2017, p. 1–2. [En ligne]. Disponible : https://arxiv.org/pdf/1708.08028.pdf

[10] S. Hendrickson et al., « Serverless Computation with OpenLambda », communicaiton présentée à 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '16), Denver, Colorado, États-Unis d'Amérique, 20-21 juin 2016, p. 1-7. [En ligne]. Disponible : https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_hendrickson.pdf

[11] Google. (2017) aws lambda, google cloud functions, azure functions, serverless - Découvrir - Google Trends. [En ligne]. Disponible : https://trends.google.com/trends/explore?date=today%205-y&q=aws%20lambda,google%20cloud%20functions,azure%20functions,serverless

[12] Plusieurs auteurs. (2017). node.js - Not able to do http requests within azure function - Stack Overflow. [En ligne]. Disponible : https://stackoverflow.com/questions/38002658/not-able-to-do-http-requests-within-azure-function

[13] G. McGrath et P.R. Brenner, « Serverless Computing : Design, Implementation, and Performance », communication présentée à 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), Atlanta, Géorgie, États-Unis d'Amérique, 5-8 juin 2017, p. 405-410. [En ligne]. Disponible : http://ieeexplore.ieee.org/document/7979855/

[14] N. Provos et D. Marzies. (1999) A Future-Adaptable Password Hashing Scheme. [En ligne]. Disponible : https://www.usenix.org/legacy/events/usenix99/provos/provos_html/node1.html

[15] D. Wirtz. (2017) bcryptjs. [En ligne]. Disponible : https://www.npmjs.com/package/bcryptjs

[16] D. Wirtz. (2017) Benchmark · dcodeIO/bcrypt.js Wiki · GitHub. [En ligne]. Disponible : https://github.com/dcodeIO/bcrypt.js/wiki/Benchmark

[17] BCryptNet. (2017) GitHub - BcryptNet/bcrypt.net : BCrypt.Net - Bringing updates to the original bcrypt package. [En ligne]. Disponible : https://github.com/BcryptNet/bcrypt.net

[18] SHOUTCLOUD. (2017) SHOUTCLOUD : THE CLOUD THAT SHOUTS BACK. [En ligne]. Disponible : http://shoutcloud.io/

[19] Locust. (2017) Locust - A modern load testing framework. [En ligne]. Disponible : https://locust.io/

[20] Digital Ocean. (2017) DigitalOcean : Cloud computing designed for developers. [En ligne]. Disponible : https://www.digitalocean.com/

[21] Google. (2017) Pricing | Cloud Functions Documentation | Google Cloud Platform. [En ligne]. Disponible : https://cloud.google.com/functions/pricing

[22] Amazon. (2017) AWS Lambda – Pricing. [En ligne]. Disponible : https://aws.amazon.com/lambda/pricing/

[23] Microsoft. (2017) Pricing - Functions | Microsoft Azure. [En ligne]. Disponible : https://azure.microsoft.com/en-us/pricing/details/functions/

[24] Amazon. (2017) EC2 Instance Pricing – Amazon Web Services (AWS). [En ligne]. Disponible : https://aws.amazon.com/ec2/pricing/on-demand/

[25] Microsoft. (2017) Pricing - Bandwidth | Microsoft Azure. [En ligne]. Disponible : https://azure.microsoft.com/en-us/pricing/details/bandwidth/

[26] Google. (2017) Google Cloud Platform Pricing Calculator | Google Cloud Platform. [En ligne]. Disponible : https://cloud.google.com/products/calculator/

[27] Amazon. (2017) AWS Lambda Pricing Calculator. [En ligne]. Disponible : https://s3.amazonaws.com/lambda-tools/pricing-calculator.html

[28] Amazon. (2017) Amazon Web Services Simple Monthly Calculator. [En ligne]. Disponible : http://calculator.s3.amazonaws.com/index.html

[29] Microsoft. (2017) Pricing Calculator | Microsoft Azure. [En ligne]. Disponible : https://azure.microsoft.com/en-us/pricing/calculator/

[30] P. Sbarski. (2017) Serverless Cost Calculator. [En ligne]. Disponible : http://serverlesscalc.com/

[31] Plusieurs auteurs. (2017) Consumption Plan Scaling Issues · Issue #1206 · Azure/azure-webjobs-sdk-script. [En ligne]. Disponible : https://github.com/Azure/azure-webjobs-sdk-script/issues/1206

[32] Auteur inconnu. (Date inconnue) The Computer Language Benchmarks Game. [En ligne]. Disponible : http://benchmarksgame.alioth.debian.org/u64q/which-programs-are-fastest.html

[33] Node.js Foundation. (2017) Overview of Blocking vs Non-Blocking | Node.js. [En ligne]. Disponible : https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/

[34] Microsoft. (2017) Multithreaded Applications (C# and Visual Basic). [En ligne]. Disponible : https://msdn.microsoft.com/en-us/library/ck8bc5c6(v=vs.110).aspx

[35] Microsoft. (2017) Choose between Flow, Logic Apps, Functions, and WebJobs | Microsoft Docs. [En ligne]. Disponible : https://docs.microsoft.com/en-us/azure/azure-functions/functions-compare-logic-apps-ms-flow-webjobs