الجمـهورية الجزائرية الديمقراطية الشعبيـة
وزارة التعليم العالي والبحث العلمي
جـامعـة سعـيـدة د. مـولاي الطـاهـر
كلية الرياضيات و الإعلام الآلي و الاتصالات السلكية و اللاسلكية
قسم: الإعلام الآلي

Mémoire de Master en informatique

Spécialité : Réseau informatique et systémes répartiess

T h è m e

DEVELOPPMENT ET DEPLOIMENT D'APPLICATIONS CLOUD

▪ **Présenté par :**

Abderrahmane SAADI

Houaria Feriel TAOUCI

▪ **Dirigé par :**

Dr.Abdelkader MOSTEFAI

Année universitaire 2024-2025

Remerciements

On remercie Dieu le Tout-Puissant de nous avoir donné la santé et la volonté d'entamer et de terminer ce mémoire.

Tout d'abord, ce travail n'aurait pas pu voir le jour sans l'aide et l'encadrement de ***Dr MOSTEFIA ABDELKADER****. Nous lui exprimons toute notre gratitude pour la qualité exceptionnelle de son encadrement, sa patience, sa rigueur et sa disponibilité durant la préparation de ce mémoire.*

Nos remerciements s'adressent tout particulièrement à ***Dr MOSTEFIA ABDELKADER*** *pour son aide précieuse, son soutien moral et ses encouragements constants.*

Nous remercions également tous nos professeurs pour leur générosité, leur grande patience et leur implication, malgré leurs charges académiques et professionnelles.

Dédicace

Au nom d'Allah, Créateur de la lumière au cœur de l'obscurité, et Source de Force lorsque le cœur faiblit. À Lui la louange, en commencement et en fin, car c'est par Sa volonté que l'on réussit, et par Sa grâce que les pas s'accomplissent.

À ceux qui furent les racines solides de cette réalisation, à mes chers parents, battement du cœur et soutien indéfectible, qui m'ont accompagné par leurs prières et leur patience, et furent pour moi une source intarissable de sérénité.

À ma sœur Zineb, lumière qui a doucement traversé mes ténèbres sans jamais être appelée, comme l'a si bien dit Le Petit Prince : « L'essentiel est invisible pour les yeux, on ne voit bien qu'avec le cœur. »

À Razika, cette main tendue dans chaque circonstance, même avant que je ne la demande, présente avec fermeté lorsque tout vacillait.

À mon frère Lakhdar, qui ne fut pas seulement l'origine de mon choix de spécialité, mais également le miroir de ma confiance quand la clarté me manquait.

À tous mes frères et sœurs dont les noms ne sont pas cités ici, mais dont la présence, les prières et l'attention discrète ont profondément marqué mon cœur.

À mes amies Chaimaa et Souhila, qui ont réchauffé le chemin et partagé ce voyage avec un amour silencieux, senti plus que dit.

Ce travail ne porte pas mon nom seul, il porte l'empreinte de chacun de ceux qui ont soutenu, aidé, ou même simplement offert un regard porteur d'espoir. Une lumière qui éclaire le chemin de ceux qui viendront après moi.

Feriel

Dédicace

Avant toute chose, je rends grâce à Dieu, le Tout-Puissant qui m'a accordé la force, la patience et la persévérance nécessaires pour mener à bien ce travail. Sans Sa volonté, aucun de mes efforts n'aurait été possible.

Je tiens à exprimer ma profonde reconnaissance à toutes les personnes qui ont, de près ou de loin, contribué à la réalisation de ce mémoire.

À mes parents,
Merci de m'avoir transmis le goût du travail et du respect. Merci pour vos valeurs, vos conseils et surtout pour votre présence, discrète mais essentielle à chaque étape de mon parcours.

À mes amis,
Merci pour votre présence, vos mots d'encouragements, vos rires partagés et votre bienveillance. Vous avez été une lumière, une source de motivation et de réconfort. Votre amitié m'est précieuse.

Abderrahmane

Liste des Abréviations

- **API** : Application Programming Interface
- **AWS** : Amazon Web Services
- **BaaS** : Backend as a Service
- **FaaS** : Function as a Service
- **PHP** : Hypertext Preprocessor
- **HTTP** : Hypertext Transfer Protocol
- **UML** : Unified Modeling Language
- **IAM** : Identity And Management
- **KPI** : Key Performance Indicator
- **ECS** : Elastic Container Service
- **CORS** : Cross-Origin Resource Sharing
- **EKS** : Elastic Kubernetes Service

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Cette étude vise à explorer le paradigme de l'informatique sans serveur pour le développement et le déploiement d'applications dans le cloud. Elle s'appuie sur des recherches approfondies incluant des analyses et des résultats expérimentaux, des exemples d'utilisation pratique et un guide pratique pour démarrer avec les plateformes sans serveur et comprendre leurs principes de base. La section « Problématique » peut également être consultée pour plus de précisions.

1.1 Problématique

Le modèle Serverless, et notamment l'approche basée sur les fonctions Lambda, suscite un intérêt croissant dans le domaine du développement cloud. Il promet une simplification du déploiement, une réduction des coûts d'infrastructure et une scalabilité automatique. En parallèle, l'approche basée sur la conteneurisation via des outils comme Docker et Kubernetes reste largement utilisée et maîtrisée.

Cependant, le manque de connaissance approfondie du modèle Serverless et la difficulté à abandonner les architectures traditionnelles freinent son adoption dans de nombreux contextes. Cela soulève une problématique essentielle : dans quelle mesure l'approche Serverless (Lambda) est-elle plus adaptée ou non que l'approche conteneurisée pour certains types d'applications ?

Cette étude vise ainsi à tester, comparer et analyser les deux approches en évaluant leurs performances, leur facilité de mise en œuvre, leurs coûts et leur adaptabilité selon les cas d'usage. L'objectif est de fournir des éléments concrets pour orienter le choix architectural dans un contexte réel de développement cloud.

1.2 Objectifs

L'objectif de ce mémoire est d'étudier le modèle de l'informatique Serverless, en particulier AWS Lambda, et de le comparer à l'approche basée sur les conteneurs (Containers), afin d'évaluer l'efficacité de chaque méthode dans le développement d'applications web cloud. L'étude se concentre sur l'analyse des performances, l'identification des cas d'usage pertinents et la compréhension des contextes où chaque approche peut offrir les meilleurs résultats.

Le travail inclut également une analyse des plateformes disponibles pour l'adoption du Serverless, avec une proposition de guide pratique couvrant la conception, le développement et le déploiement, dans le but d'optimiser l'évolutivité, la gestion des ressources et la réduction des coûts par rapport aux architectures traditionnelles.

À la fin de ce mémoire, on s'attend à identifier les avantages et les limites du modèle Serverless, à explorer les frameworks les plus utilisés, et à comprendre les raisons du ralentissement de son adoption malgré sa réputation de solution d'avenir. L'étude vise ainsi à clarifier les enjeux pratiques de ce paradigme émergent.

1.3 Méthodologie

La méthodologie de la recherche en science de la conception (Design Science Research - DSR) a été utilisée dans cette étude. La DSR est une méthodologie itérative qui comprend l'identification du problème, la conception et le développement d'un artefact, ainsi que l'évaluation de cet artefact innovant afin de proposer des solutions efficaces à des problèmes complexes dans des environnements pratiques et techniques (Claes Wohlin 2021).

Cette méthodologie est particulièrement adaptée à cette étude, car elle met l'accent sur le développement et l'évaluation de solutions techniques à des problèmes réels, tout en permettant des ajustements continus tout au long du processus de recherche et de découverte.

L'objectif principal de ce projet est d'étudier l'efficacité de différentes architectures (Serverless et Docker) dans le développement d'applications web interactives. Ainsi, la méthodologie DSR a été suivie, car elle offre un cadre structuré permettant d'identifier les défis associés à chaque architecture, de concevoir et de développer deux solutions pratiques à l'aide de technologies distinctes, puis de comparer leur performance et leur efficacité à travers une évaluation rigoureuse basée sur des tests de performance, de flexibilité et d'efficacité opérationnelle.

Chapitre 2

Contexte Général du Projet

Cette section présente les concepts et définitions fondamentaux utilisés tout au long de ce document. Nous commençons par les notions principales qui constituent la base de notre travail, puis nous introduisons les autres concepts nécessaires afin de fournir au lecteur toutes les informations utiles à la bonne compréhension du contenu présenté.

2.1 Cloud Computing

Le cloud computing est un modèle permettant un accès réseau universel, pratique et à la demande à un pool partagé de ressources informatiques configurables (par exemple, des réseaux, des serveurs, du stockage, des applications et des services) qui peuvent être rapidement provisionnées et libérées avec un minimum d'effort de gestion ou d'interaction avec le fournisseur de services. [1].

Fig. 2.1 : Cloud Computing

2.1.1 Historique

Le Cloud Computing, anticipé dès les années 1960 par John McCarthy, a émergé à la fin du XXe siècle, Amazon étant un pionnier dans la création de data centers. À partir de 2008, des entreprises comme IBM et Google ont sérieusement investi dans cette techno-

logie, encourageant la migration des infrastructures vers des services payés à l'utilisation. Initialement réservée aux superordinateurs, la puissance de calcul du Cloud est désormais accessible via Internet, simplifiant ainsi l'accès aux ressources informatiques.

2.1.2 Modèles de Déploiement du cloud computing

Les modèles de déploiement du cloud décrivent les différentes manières d'utiliser le cloud et définissent également ses niveaux d'accès. Ainsi, les organisations disposent d'une variété d'options pour choisir une solution cloud en fonction de leurs besoins spécifiques [2].De plus, il existe plusieurs modèles de déploiement de Cloud qui vont permettre aux fournisseurs (CSP) d'offrir les services du Cloud selon les exigences de l'utilisateur (CSU) en termes de sécurité et de qualité de service. À ce titre, nous citons :

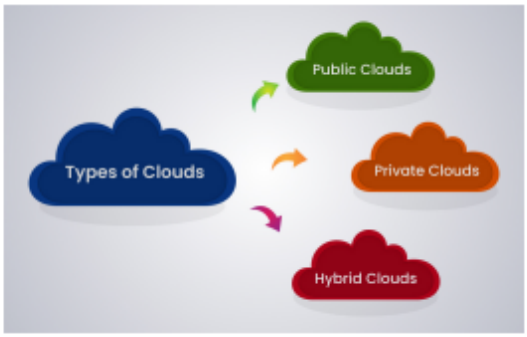


FIG. 2.2 : Les types du cloud computing

Cloud Public : dans ce modèle de déploiement les CSP offrent leurs ressources comme services à tout type d'utilisateurs. Ce modèle peut être détenu, géré et exploité par une entreprise ou une organisation académique ou gouvernementale, ou une combinaison de ces entités

Cloud Privé : ce modèle de déploiement est conçu pour une utilisation exclusive par une seule organisation. Un Cloud privé peut être construit et géré par l'organisation, une tierce partie, ou une combinaison.

Cloud Hybride : Les clouds hybrides sont une combinaison de cloud privés et publics, connectés avec une technologie qui permet aux données et aux applications de travailler ensemble.

2.1.3 Les services du cloud computing

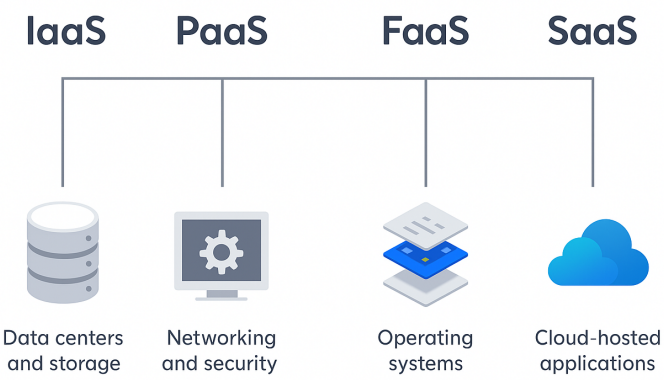


FIG. 2.3 : Les services du cloud computing

1. SaaS (Software-as-a-Service ou logiciel en tant que service) : La migration vers le cloud permet aux entreprises d'optimiser les coûts informatiques. En effet, le cloud computing élimine la nécessité d'investir dans du matériel et des logiciels, et de configurer et de gérer des centres de données sur site : racks de serveurs, alimentation électrique permanente pour l'alimentation et le refroidissement, experts informatiques pour la gestion de l'infrastructure.

2. PaaS (Platform-as-a-Service ou plateforme en tant que service : La Platform-as-a-Service (PaaS) est un type d'offre de cloud computing dans lequel un fournisseur de services fournit une plateforme à ses clients, leur permettant de développer, d'exécuter et de gérer des applications commerciales sans avoir à construire et à maintenir l'infrastructure que ces processus de développement de logiciels requièrent généralement [3].

3. IaaS (Infrastructure-as-a-Service ou Infrastructure en tant que service) : L'infrastructure en tant que service (IaaS) est une forme de cloud computing qui fournit des ressources d'infrastructure informatique à la demande telles que des serveurs, des machines virtuelles (VM), des ressources de calcul, de réseau et de stockage aux consommateurs via Internet et sur une base de paiement à l'utilisation [4].

4. FaaS (Function-as-a-Service ou Fonction en tant que service) : Les plateformes FaaS (Fonction en tant que service) exécutent de petites unités de code en réponse à un événement. Cet événement peut être, par exemple, une requête HTTP, une opération dans une base de données ou un message dans une file d'attente. Les utilisateurs déploient du code sur les plateformes FaaS, et celui-ci est exécuté à la demande en fonction des événements. Le fournisseur de services gère la mise à l'échelle et l'infrastructure, de sorte que le service apparaît comme sans serveur pour l'utilisateur [5].

2.2 Conteneur sans serveur

Dans le monde en évolution du cloud computing, les conteneurs sont devenus une technologie de base pour la création et le déploiement d'applications modernes. Lorsqu'ils sont combinés avec des modèles d'exécution serverless, les conteneurs offrent un paradigme puissant qui élimine la gestion de l'infrastructure tout en conservant la flexibilité de la conteneurisation. Ce document explore le concept de conteneurs serverless, en mettant l'accent sur leur utilisation avec les services AWS tels que Amazon ECS et AWS Fargate.

2.2.1 Définition

Un conteneur est une unité standard de logiciel qui regroupe le code et toutes ses dépendances, assurant une exécution cohérente sur différents environnements du développement à la production. Les conteneurs isolent les applications dans l'espace utilisateur tout en partageant le noyau du système d'exploitation.

2.2.2 Modèles de Déploiement de Conteneurs

1. Autogéré sur EC2 : Les développeurs provisionnent et gèrent manuellement des instances EC2, utilisant des outils comme ECS ou Kubernetes.

2. Kubernetes Géré (Amazon EKS) : Offre un plan de contrôle Kubernetes géré pour déployer et gérer les conteneurs à grande échelle.

3. Amazon ECS sur EC2 : Utilise l'orchestration ECS native d'Amazon sur des instances EC2.

4. Conteneurs Serverless (Amazon ECS avec AWS Fargate) : Exécution totalement serverless où AWS gère le provisionnement, la mise à l'échelle et l'infrastructure.

2.2.3 Conteneurs Docker

Docker est l'une des technologies d'exécution de conteneurs les plus populaires et les plus largement utilisées. Elle permet de créer, livrer et exécuter des applications rapidement et facilement sur n'importe quelle machine. Docker a introduit une méthode simplifiée pour le lancement de micro services, tout en favorisant une nouvelle façon de collaborer entre les équipes. Cela a marqué un tournant pour le mouvement Devos, en ouvrant la voie à de nouvelles technologies et à de nouveau mode de pensée. Grâce au démon Docker fonctionnant au-dessus du système d'exploitation hôte, le processus de lancement de conteneurs à l'infini devient plus facile.

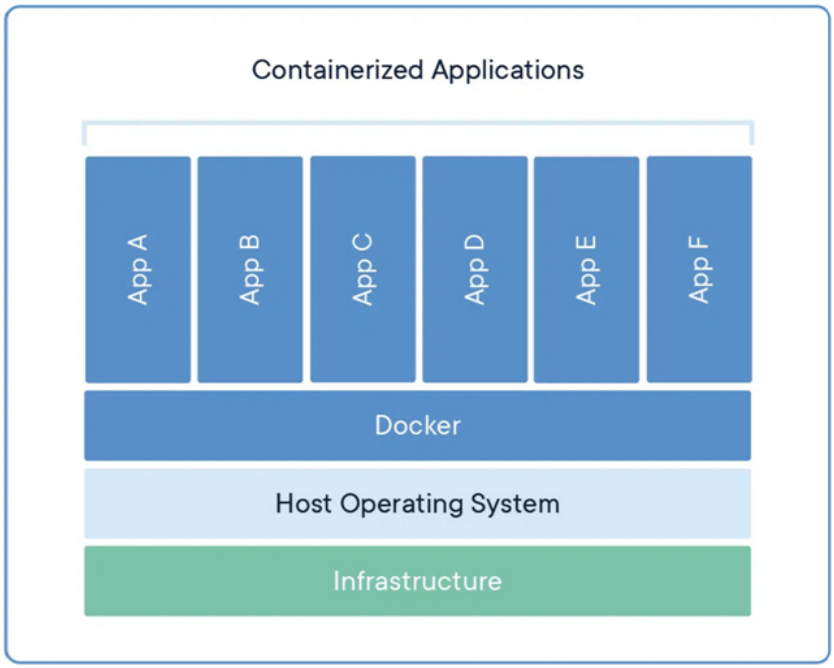


FIG. 2.4 : Docker architecture

Avec Docker agissant comme une interface entre le système d'exploitation de l'hôte et chaque conteneur, il permet à l'administrateur système de gérer et de lancer facilement les images Docker et les conteneurs Docker. Les utilisateurs peuvent stocker des images Docker réutilisables dans un registre privé ou public, et les partager avec leurs équipes ou même avec toute la communauté. En ayant Docker installé sur leur poste de travail, les développeurs peuvent récupérer (pull) l'image Docker et l'exécuter sur leur machine en tant que conteneur Docker [6].

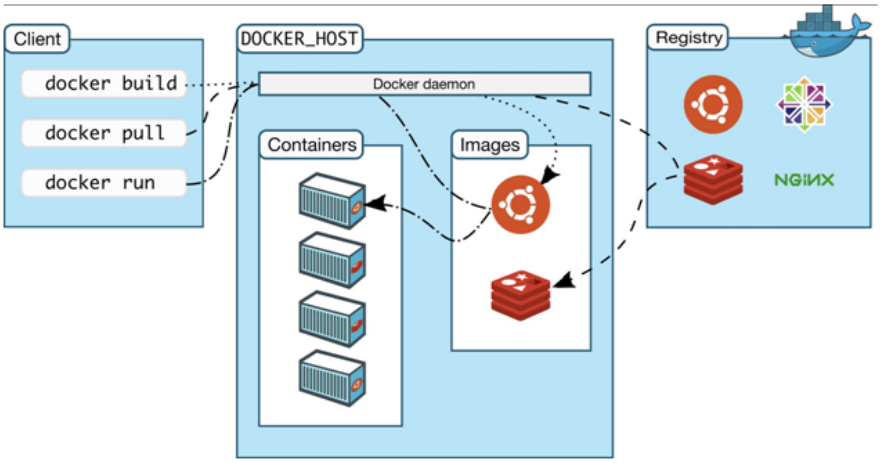


FIG. 2.5 : processus Docker

2.3 Serverless Computing

2.3.1 Historique

En 2009, Netflix a été confrontée à un problème très courant dans la plupart des entreprises informatiques : le besoin de passer à l'échelle. Cependant, leur architecture monolithique rendait cette tâche difficile, voire parfois impossible. À cette époque, le concept de microservices n'existait même pas encore, mais ils ont tout de même pris la décision de décomposer leur application monolithique en plusieurs petites applications autonomes. Cette décision a fait d'eux un pionnier dans ce qui est devenu aujourd'hui une pratique de plus en plus répandue : la transition d'une architecture monolithique vers une architecture basée sur les microservices (Harris, s.d.). Cependant, les microservices nécessitent des orchestrateurs pour les gérer, car cette tâche peut parfois s'avérer coûteuse et chronophage. Cette nouvelle approche n'a donc pas tout résolu. Pour répondre à certains de ces défis restants, une nouvelle architecture a vu le jour. Dans cette section, nous introduirons le concept du Serverless Computing, son architecture, ses avantages et ses inconvénients, ainsi que ses domaines d'application.

2.3.2 Définition

Aujourd'hui, il existe plusieurs définitions pour expliquer la signification de Serverless. Selon l'article The Rise of Serverless Computing, le Serverless est défini comme : « une plateforme qui masque l'utilisation des serveurs pour les développeurs et exécute le code à la demande, avec une mise à l'échelle automatique et une facturation uniquement pour le temps d'exécution du code » (Paul Castro, 2019). D'un autre côté, la CNCF (Cloud Native Computing Foundation) définit le Serverless comme : « Le concept de création et

d'exécution d'applications ne nécessitant aucune gestion de serveur. Il s'agit d'un modèle de déploiement plus granulaire dans lequel les applications, regroupées sous forme d'une ou plusieurs fonctions, sont téléchargées sur une plateforme, puis exécutées, mises à l'échelle et facturées en fonction de la demande exacte du moment » (CNCF, s.d.). Ces différentes définitions soulignent des aspects clés de ce concept, mais quelle que soit la définition adoptée, un élément fondamental du Serverless est que les utilisateurs n'ont pas à se soucier de la gestion ou de l'hébergement des serveurs. Cette responsabilité est entièrement déléguée aux fournisseurs cloud.

2.3.3 Les principaux fournisseurs du Serverless Computing

1. Services Web d'Amazon (AWS) : AWS propose la plus grande variété d'instances de calcul, de classes de stockage, de bases de données et d'analyses, toutes spécialement conçues pour offrir le meilleur rapport coût/performance.

2. Microsoft Azure : Microsoft Azure est la plateforme cloud computing développée par Microsoft. Elle offre une variété d'applications et de services aux particuliers, aux entreprises et aux gouvernements grâce à son infrastructure mondiale.

3. Google Cloud Platform (GCP) : Google Cloud Platform est l'un des cloud providers du marché. GCP regroupe les services Google Cloud et propose des solutions de stockage, de sécurité,

4. IBM Cloud : IBM propose une gamme complète de services de Cloud Computing, y compris des services de calcul, de stockage, de base de données, d'analyse et d'IA, ainsi que des solutions sectorielles et des services de sécurité.

2.3.4 Fonction en tant que service (FaaS) :

Function as a Service (FaaS) est un service de cloud computing qui permet aux clients d'exécuter du code en réponse à des événements, sans gérer l'infrastructure complexe généralement associée à la création et au lancement d'applications de microservices [7].

Les plateformes Function as a Service (FaaS) exécutent de petites unités de code en réponse à un événement. L'événement peut être, par exemple, une requête HTTP, une opération dans une base de données ou un message dans une file de messages. Les utilisateurs déploient leur code sur des plateformes FaaS, et ce code est exécuté à la demande, en fonction des événements. Le fournisseur de services se charge de la mise

à l'échelle et de l'infrastructure, de sorte que le service apparaisse comme sans serveur (serverless) pour l'utilisateur.

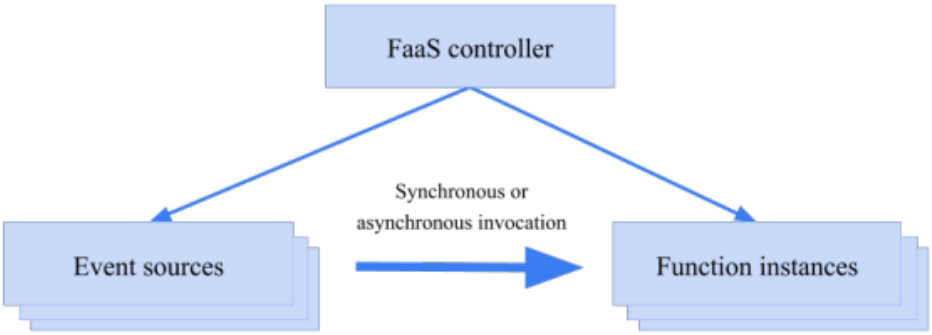


FIG. 2.6 : Modèle de traitement des fonctions FaaS

Chapitre 3

Analyse de la Valeur

Dans cette section, l'objectif est d'analyser la valeur que l'informatique sans serveur (serverless) apporte au client. Il est important d'étudier ce que le client souhaite et comment cette nouvelle architecture va impacter sa vie, que ce soit de manière positive ou négative. Cet aspect doit toujours être pris en compte afin de ne jamais perdre de vue l'objectif à atteindre tout en maintenant la satisfaction du client. Le client peut être à la fois le développeur, qui conçoit et maintient l'application, ou l'administrateur système, qui doit créer et gérer toute l'infrastructure.

3.1 Valeur pour le client

La principale valeur que les clients retirent d'une architecture Serverless est l'augmentation de la fiabilité, de la scalabilité (montée en charge) et de la disponibilité de leurs applications. Grâce à la capacité d'allouer automatiquement les ressources à la demande, l'architecture Serverless élimine le besoin de configurations manuelles, ce qui peut réduire les temps d'arrêt et améliorer les performances. De plus, une architecture Serverless permet aux clients de se concentrer sur leurs fonctions métier principales, plutôt que sur la gestion de l'infrastructure sous-jacente, ce qui peut entraîner des réductions de coûts et une efficacité accrue.

3.2 Valeur perçue

La valeur perçue d'une architecture Serverless réside principalement dans sa capacité à réduire les coûts opérationnels et à faciliter le développement de nouvelles applications et services. La flexibilité et la scalabilité offertes par une architecture Serverless peuvent aider les entreprises à répondre rapidement à l'évolution des besoins métiers et à s'adapter aux exigences du marché. De plus, l'amélioration de la fiabilité et de la disponibilité des applications et services peut renforcer la confiance et la crédibilité aux yeux des clients.

3.3 Avantages et sacrifices

Au fil des années, les gens ont utilisé soit des architectures monolithiques, soit micro services. Les deux sont des modèles valides qui présentent des avantages et des inconvénients, en fonction du scénario du projet. Aujourd'hui, l'objectif principal est de laisser de la place à la croissance et à la montée en charge des projets. Ainsi, les architectures monolithiques seront écartées afin de mieux comprendre les principales différences entre les architectures Serverless et Micro services, ainsi que les avantages et les sacrifices que le modèle Serverless implique.

3.3.1 Avantages

Coûts : L'un des principaux avantages du passage à un modèle Serverless est l'optimisation des coûts. Les utilisateurs ne sont facturés que pendant la période où leur code est en cours d'exécution. Ce mode de fonctionnement est également appelé "pay-as-you-go" (payer à l'usage). Au lieu de payer pour des serveurs souvent inactifs ou dont la capacité de calcul n'est pas pleinement utilisée, les utilisateurs ne paient que pour le temps réel d'exécution de leur code et pour les ressources spécifiques dont ils ont besoin. Afin de permettre une facturation précise — car le temps d'exécution peut être très court — la tarification se fait en unités de temps fines (par exemple, par centaines de millisecondes). De plus, les développeurs n'ont pas à payer pour les frais généraux liés à la création ou à la suppression de serveurs (comme le temps nécessaire au démarrage des machines virtuelles). Ce modèle de coût est très attractif pour les charges de travail qui ne s'exécutent qu'occasionnellement. En revanche, cela représente un grand défi pour les fournisseurs de services cloud, qui doivent planifier et optimiser les ressources afin de partager leur capacité de calcul entre plusieurs utilisateurs .

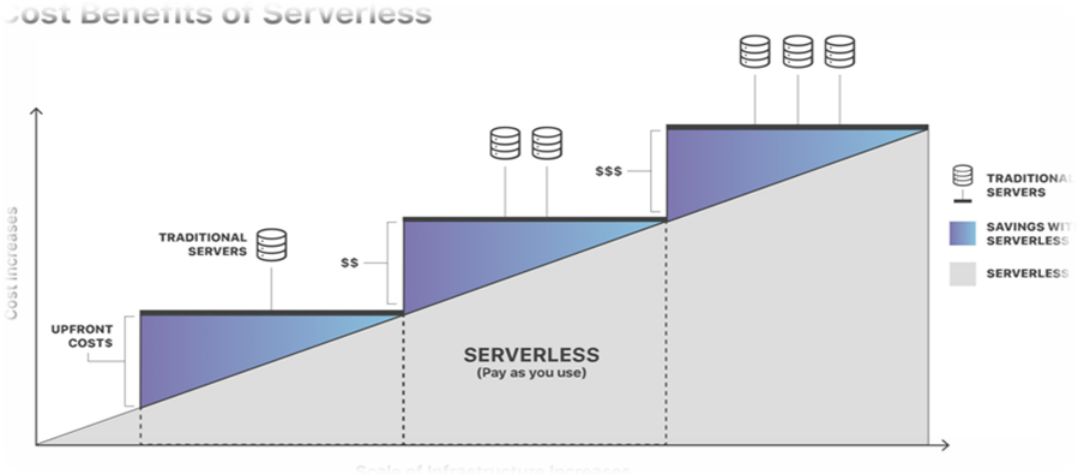


FIG. 3.1 : Avantages en termes de coût de l'informatique Serverless

Scalabilité : Étant donné que les développeurs ne possèdent ni ne gèrent les serveurs qui exécutent leur code, ils se voient délestés de plusieurs responsabilités. Il ne fait plus partie de leurs fonctions de mettre à l'échelle les nœuds ni de créer des règles d'auto-scalabilité ou de scalabilité horizontale. Ce sont les fournisseurs de services cloud qui ont la tâche et la responsabilité de s'assurer que les services sont toujours disponibles et correctement dimensionnés pour éviter toute limitation. Cela représente également un avantage considérable pour les administrateurs systèmes, qui n'ont plus à appliquer les dernières mises à jour de sécurité, ni à se soucier de nombreuses autres préoccupations qui faisaient auparavant partie de leur routine. À la place, ils délèguent cette fonction au fournisseur cloud, désormais responsable de garantir une scalabilité automatique et une disponibilité constante des services.

Flexibilité : Exécuter une fonction ou n'importe quel backend peut être aussi simple que d'appuyer sur un bouton lorsqu'on utilise une architecture Serverless. Les développeurs peuvent facilement déployer et tester de nouvelles fonctionnalités sans se soucier de l'infrastructure sous- jacente.

Tolérance aux pannes : Dans une architecture à microservices, pour éviter les interruptions de service, il est courant de mettre en place une forme de redondance ou de réplication. La plupart des fournisseurs de services Cloud proposent les options suivantes pour renforcer la disponibilité de votre infrastructure (Azure 2022) : • Stockage localement redondant (LRS) • Stockage redondant par zone (ZRS) • Stockage géo-redondant (GRS) • Stockage géo-redondant avec accès en lecture (RA-GRS) Ce sont différentes options permettant de répartir l'infrastructure soit sur différentes machines locales, soit dans différentes zones Cloud (par exemple eu-west-1 et eu-west-2), voire dans des emplacements géographiques différents. Cependant, dans une architecture Serverless, ce n'est pas quelque chose qu'il faut choisir ou gérer soi-même. Le modèle Serverless répartit automatiquement la charge de travail entre plusieurs ressources de calcul, ce qui améliore la capacité de tolérance aux pannes de l'infrastructure et réduit le risque d'interruption de service.

3.3.2 Sacrifices

Démarrage à froid (Cold start) : Ce qui peut être l'un des plus grands avantages de l'architecture Serverless peut aussi représenter l'un de ses plus grands inconvénients. Le fait d'avoir des fonctions éphémères est un atout considérable car cela permet de réduire les coûts, mais cela peut entraîner un temps de réponse plus long que d'habitude. Cela se

produit uniquement lors de la première requête, lorsque la fonction est encore inactive, mais cela reste un phénomène récurrent.

Latence : Ce deuxième inconvénient est lié au premier. Il y a la latence mentionnée dans la section précédente, mais aussi une latence liée au réseau. Puisque la machine qui hébergera le service peut se trouver n'importe où dans le monde, cela peut avoir un impact sur les performances de l'application et son temps de réponse. Le temps de réponse d'une fonction est la somme du temps que met la requête à voyager du client vers la plateforme Serverless, le temps de traitement de la fonction, et enfin le temps de réponse pour revenir vers le client. Le fait que le service s'exécute sur une machine qui héberge également d'autres ressources peut engendrer une « concurrence » sur les ressources de calcul, telles que la mémoire ou le CPU, ce qui peut accentuer encore la latence.

Débogage : Ne pas gérer l'infrastructure sous-jacente peut sembler très avantageux, mais cela comporte aussi des inconvénients. Les applications Serverless peuvent être composées de plusieurs fonctions faiblement couplées, exécutées sur des hôtes distincts. Si une erreur survient dans le processus, il peut être très difficile de la localiser et de la corriger, car la source peut se trouver sur n'importe quelle machine à travers le monde. Les applications Serverless sont également éphémères, ce qui signifie qu'elles cessent d'exister une fois leur tâche terminée, rendant le débogage encore plus difficile puisque tout le contenu disparaît à moins qu'il ne soit stocké dans un emplacement permanent. Cela peut être résolu par des outils et bonnes pratiques comme la journalisation (logging), la surveillance (monitoring), la traçabilité distribuée (distributed tracing) et des frameworks de test.

Sécurité : Les fonctions Serverless traitent des données sensibles et peuvent être exposées à de nombreuses menaces de sécurité. Même si le fournisseur Cloud sécurise au mieux ses machines, il est crucial que les développeurs chiffrent et protègent toutes les données sensibles. Même le code non chiffré peut être attaqué et exploité. Il existe aussi des exigences de conformité, comme les lois sur la protection des données (ex. RGPD), que ces machines partagées peuvent ne pas respecter. Un autre problème de sécurité vient de la dépendance à des bibliothèques tierces. Si une vulnérabilité est découverte et que la dépendance n'est pas mise à jour, cela constitue une faille potentielle pour un attaquant informé. Comme les vulnérabilités connues sont listées dans la base CVE (Common Vulnerabilities and Exposures), les pirates savent quoi chercher pour attaquer ces environnements partagés. Une autre attaque possible est l'attaque DDoS, qui consiste à surcharger une API depuis plusieurs hôtes pour simuler un trafic massif. Bien que les fonctions Serverless aient des limites de mémoire et de CPU, une telle attaque peut épuiser les ressources et

nuire gravement à la performance et à la disponibilité de l'application.

Verrouillage fournisseur (Vendor lock-in) : Une fois que tout est développé sur un fournisseur Cloud spécifique (comme Azure, AWS, etc.), les dépendances deviennent trop fortes. Les SDK et API utilisés pour interagir avec les services FaaS diffèrent d'un fournisseur à l'autre, ce qui rend difficile tout changement dans un projet déjà en production, en raison des risques élevés associés à une migration aussi importante. Lorsque l'on devient dépendant d'un fournisseur, on peut perdre des opportunités offertes par les concurrents à cause de ce manque de flexibilité.

Chapitre 4

Analyse et Implémentation

Ce chapitre décrit l'analyse du cas d'utilisation, suivie de la proposition de conception pour atteindre l'objectif défini. Après avoir pris connaissance du cas d'utilisation et des exigences, la section de conception présentera un guide d'adoption pour cette transition vers un modèle Serverless.

4.1 Analyse

Afin d'évaluer l'efficacité de l'architecture serverless, une comparaison sera réalisée entre deux approches : la première utilisant les fonctions AWS Lambda, et l'autre utilisant des conteneurs déployés via Amazon ECS avec Fargate. L'analyse portera sur les aspects liés à la méthode de mise en œuvre, la performance, la scalabilité et le coût. L'objectif est d'identifier les avantages et les limites de chaque solution afin de choisir l'approche la plus adaptée aux besoins du système.

4.2 Conception de l'architecture

L'architecture du système de chabot incarne une philosophie de conception à la fois simple en apparence et profonde dans sa fonction. Le backend, développé en Python, repose sur un modèle serverless orienté événements, assurant une connexion fluide et flexible avec le cloud. Grâce à AWS API Gateway et au protocole Web Socket, le système permet une communication bidirectionnelle en temps réel, offrant ainsi une expérience conversationnelle naturelle. Chaque message utilisateur déclenche une chaîne précise de fonctions cloud, avec AWS Lambda comme cœur computationnel du système. Ces fonctions invoquent des modèles d'intelligence artificielle via Amazon Bedrock pour répondre aux requêtes textuelles ou générer des images à partir de descriptions. Ainsi, le système établit une synergie entre la compréhension intelligente et la création visuelle, reliant le langage à l'imaginaire. Le résultat est une architecture légère, évolutive et hautement

réactive où la logique est découplée, la latence réduite, et l'intelligence s'écoule du code jusqu'à la conversation.

4.2.1 Plateforme serverless

En utilisant une fonction Serverless, nous avons pu concevoir une infrastructure capable de s'ajuster dynamiquement à la charge des données entrantes, sans jamais compromettre les performances de traitement. Cette approche exploite la scalabilité et l'efficacité économique du modèle serverless, permettant de traiter les requêtes à la demande sans gestion manuelle des serveurs.Pour faire face à la complexité du traitement, la logique métier a été externalisée sous forme d'une fonction serverless. Au lieu d'exécuter cette logique dans un microservice backend classique, elle est désormais gérée par une fonction distante, déclenchée uniquement lorsque nécessaire. AWS Lambda a été adoptée comme outil et plateforme principale pour cette fonction cloud. Puisque l'application est développée en Python, il était crucial de choisir une plateforme compatible nativement avec ce langage, tout en offrant une intégration fluide avec les autres services AWS. AWS Lambda propose un support direct pour Python, en plus d'une interface simple et fiable, ce qui en fait le choix le plus pertinent pour notre application. L'un des avantages majeurs de AWS Lambda est l'existence d'un palier gratuit pour les premiers un million d'invocations mensuelles, ce qui a permis de réduire les coûts pendant la phase de développement du prototype. Au-delà de ce seuil, le modèle de facturation "pay- as-you-go" (paiement à l'usage) garantit que l'on ne paie que pour les ressources effectivement consommées, assurant une optimisation des dépenses. Cette fonction peut être invoquée de plusieurs façons, notamment via API Gateway (pour la communication WebSocket), AWS SDK, Amazon S3, Amazon SQS, etc. Dans notre cas, la fonction est directement connectée au backend qui gère les sessions de dialogue via WebSocket. Chaque message utilisateur est transmis à AWS Lambda pour traitement et génération de réponse à l'aide de Amazon Bedrock. Les conversations sont ensuite enregistrées dans la base de données DynamoDB, assurant la continuité du dialogue

4.2.2 Modelisation des composants et des interactions UML

Le langage UML (Unified Modeling Langage) est un langage de modélisation qui a pour but de faciliter les transactions, lors du développement d'un projet, du besoin originel à la phase D'implémentation. Ce langage se définit comme un langage de modélisation graphique et textuel destiné à comprendre et à définir des besoins, spécifier et documenter des systèmes, esquisser des architectures logicielles, concevoir des solutions et communiquer des points de vue. ML modélise ensemble des données et des traitements en élaborant différents diagrammes.

Diagramme de cas d'utilisation

Un diagramme de cas d'utilisation peut servir à résumer les informations des utilisateurs de votre système (également appelés acteurs) et leurs interactions avec ce dernier. La création de ce type de diagramme UML requiert un ensemble de symboles et de connecteurs spécifiques [8].

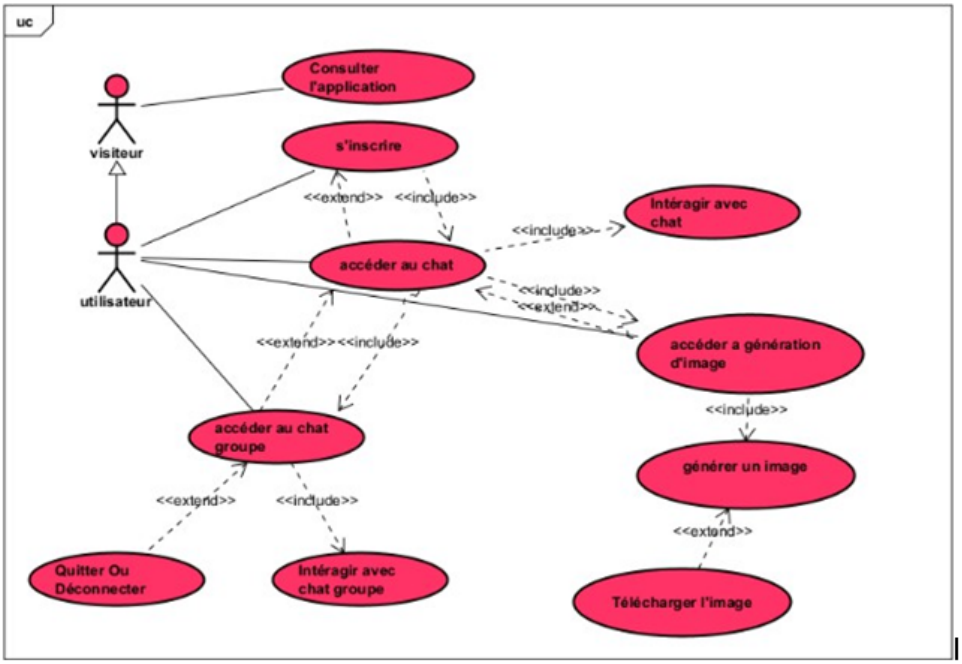


FIG. 4.1 : Diagramme de cas d'utilisation

Diagramme de Séquence

Les diagrammes de séquences permettent de décrire comment les éléments du système interagissent entre eux et avec les acteurs :

- Les objets au cœur d'un système interagissent en s'échangeant des messages.
- Les acteurs interagissent avec le système au moyen d'IHM (Interfaces Homme-Machine).

Source : [9]

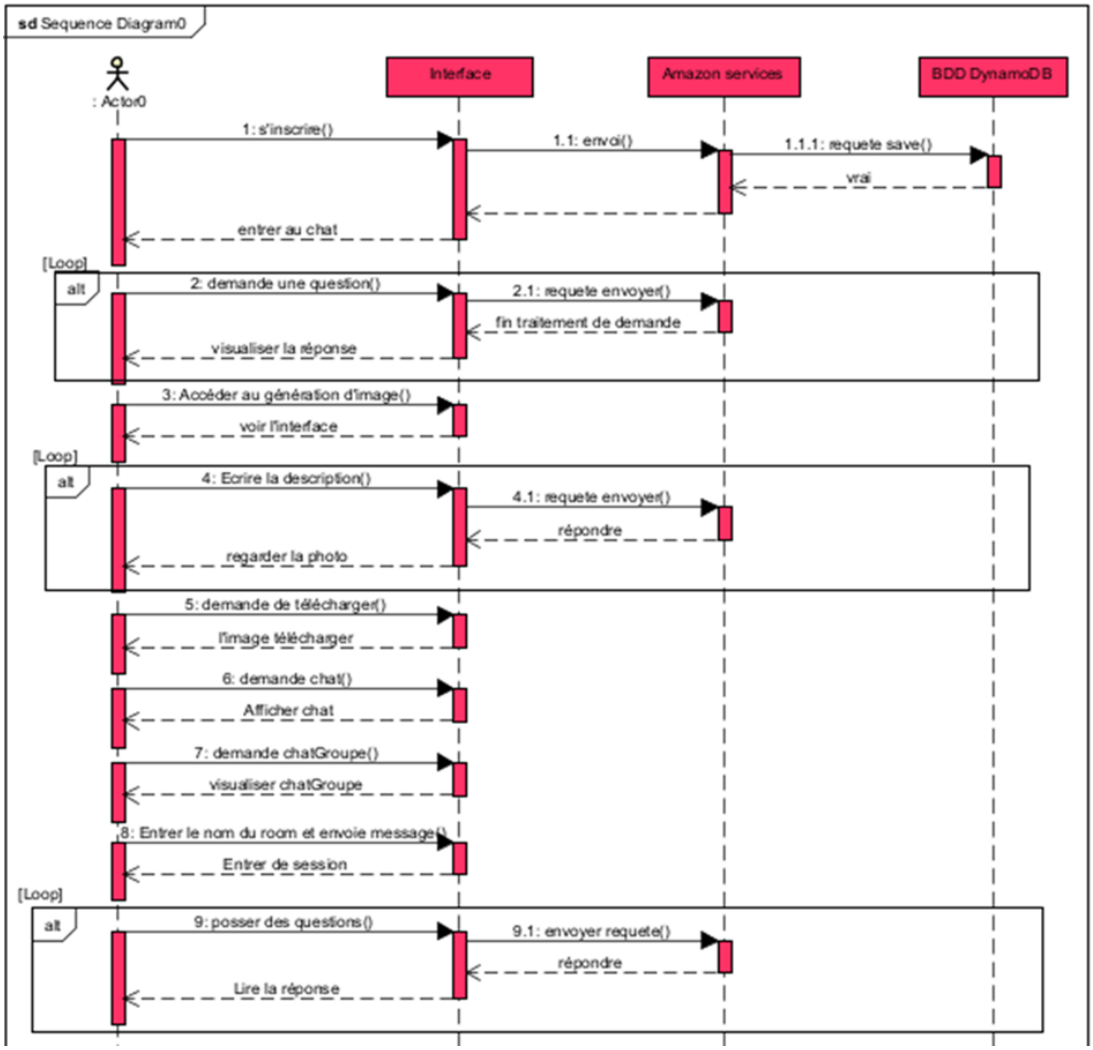


FIG. 4.2 : Diagramme de séquence

Diagramme de classe d'Analyse

Le diagramme de classe d'analyse est un modèle UML utilisé pour représenter les concepts métier d'un système sans détails techniques pour comprendre les besoins fonctionnels avant de passer à la conception technique.

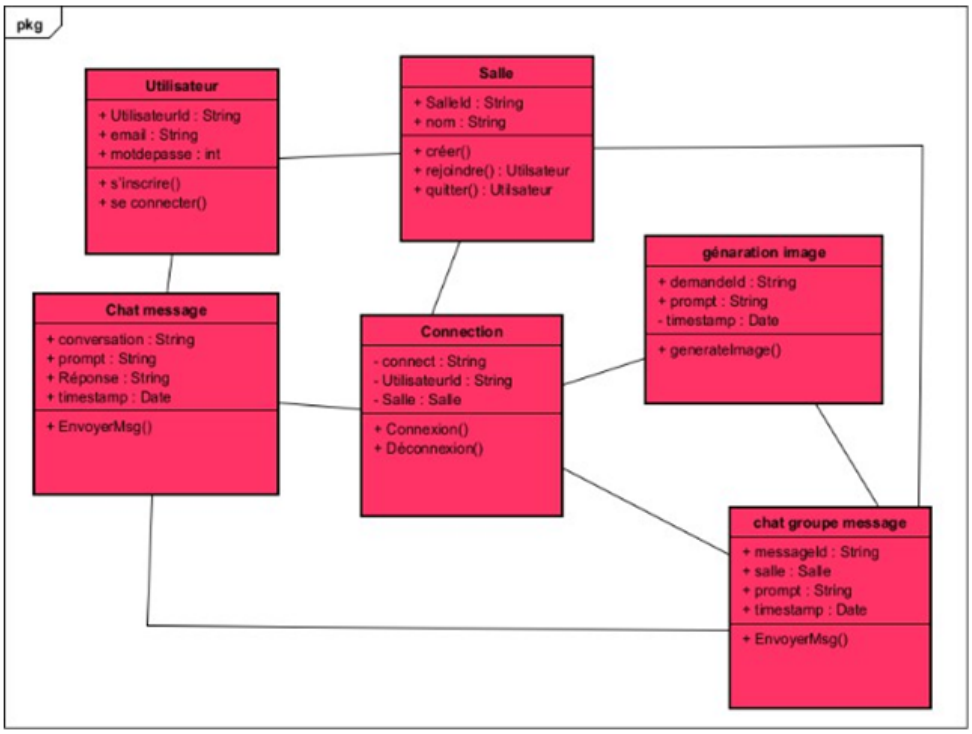


FIG. 4.3 : Diagramme de Classe d'Analyse

Diagramme de Composant

Les diagrammes de composants représentent la structure du système logiciel, qui décrit les composants du logiciel, leurs interfaces et leurs dépendances. [10]

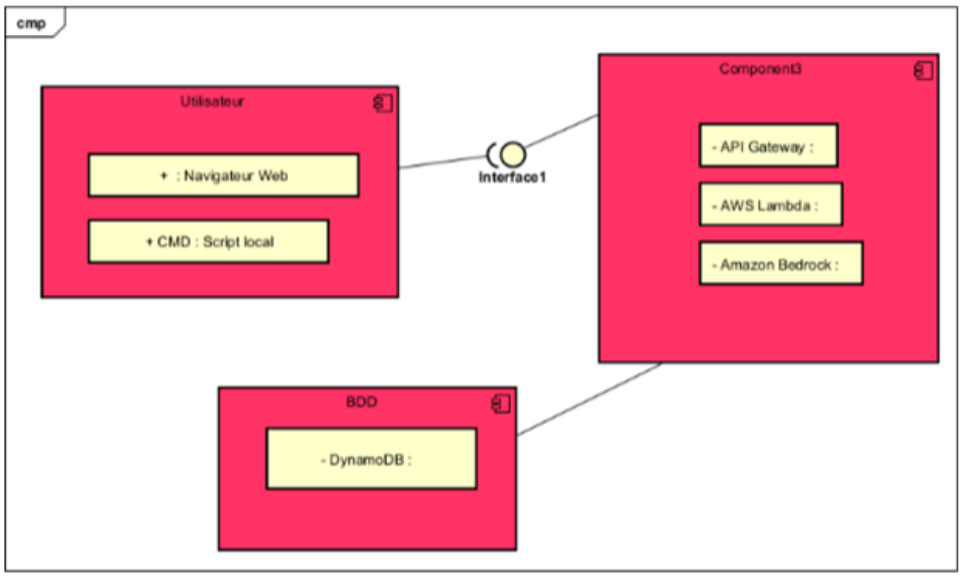


FIG. 4.4 : Diagramme de composant

Diagramme de Déploiement

Les diagrammes de déploiement modélisent l'architecture physique d'un système. Les diagrammes de déploiement affichent les relations entre les composants logiciels et matériels du système, d'une part, et la distribution physique du traitement, d'autre part. [12]

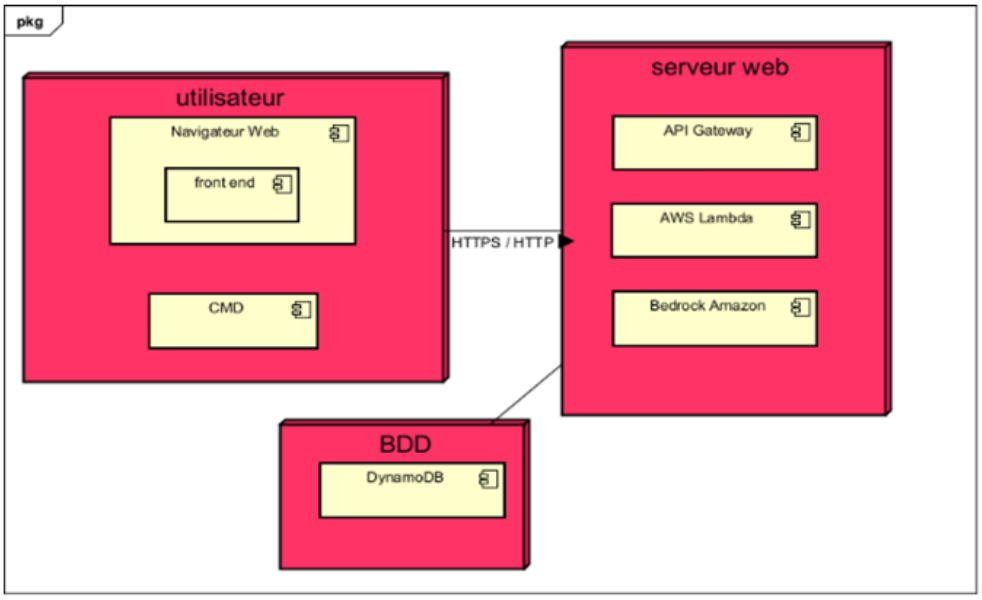


FIG. 4.5 : Diagramme de déploiement

4.3 Guide d'adoption

Dans le cadre de ce travail, deux approches différentes ont été adoptées pour fournir le même service : la première repose sur les fonctions AWS Lambda, tandis que la seconde utilise des conteneurs Docker gérés via Amazon ECS avec Fargate. Cette double adoption vise à tester chaque approche dans un environnement réel identique, et à comparer leurs performances en termes de vitesse, efficacité et consommation des ressources. Les deux solutions ont été construites avec la même logique backend, en utilisant le langage Python et l'intelligence artificielle d'Amazon Bedrock, afin de garantir une comparaison équitable et une analyse précise. Cette démarche permet une évaluation objective de l'efficacité de chaque plateforme Serverless dans le traitement de requêtes d'IA interactives, orientant ainsi vers le choix le plus adapté en matière de performance et de coût.

4.3.1 Création d'un environnement AWS

Avant d'implémenter les fonctions serverless, il est indispensable de disposer d'un environnement AWS correctement configuré. Cela commence par la création d'un compte

AWS Free Tier, qui donne un accès gratuit à plusieurs services pendant une période d'essai. Le processus comprend la validation de l'adresse e-mail, l'ajout des informations de facturation, la vérification du numéro de téléphone et la sélection du plan de support de base (gratuit).

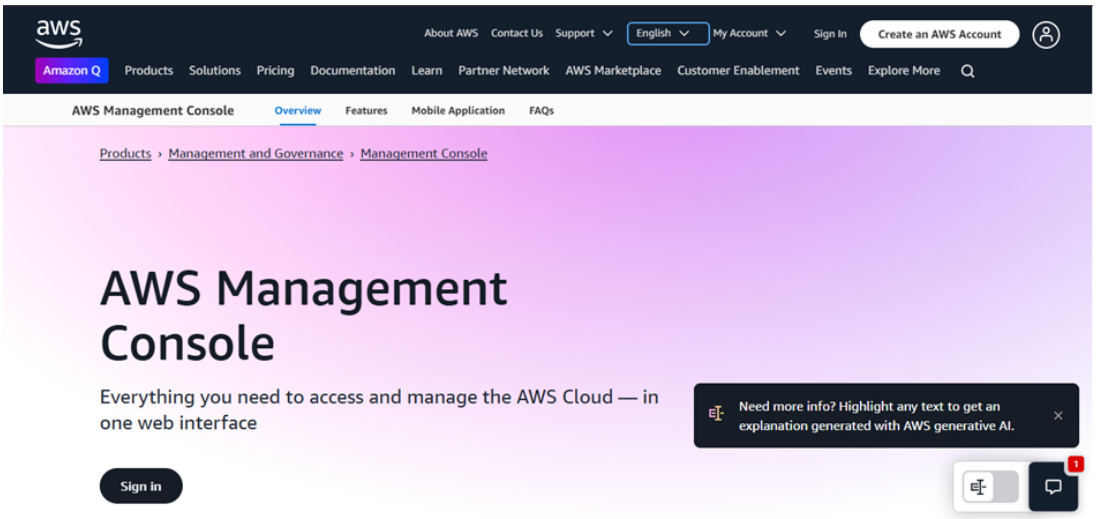


FIG. 4.6 : Création d'un compte AWS

Cependant, dans le contexte algérien, la vérification du numéro de téléphone a représenté un défi majeur. En effet, le système de vérification automatique d'AWS n'est pas toujours compatible avec les opérateurs locaux, ce qui a entraîné des retards importants dans le processus d'activation. La création et la validation complètes du compte ont nécessité près d'un mois, entre tentatives répétées, recours à différents numéros et assistance auprès du support technique.

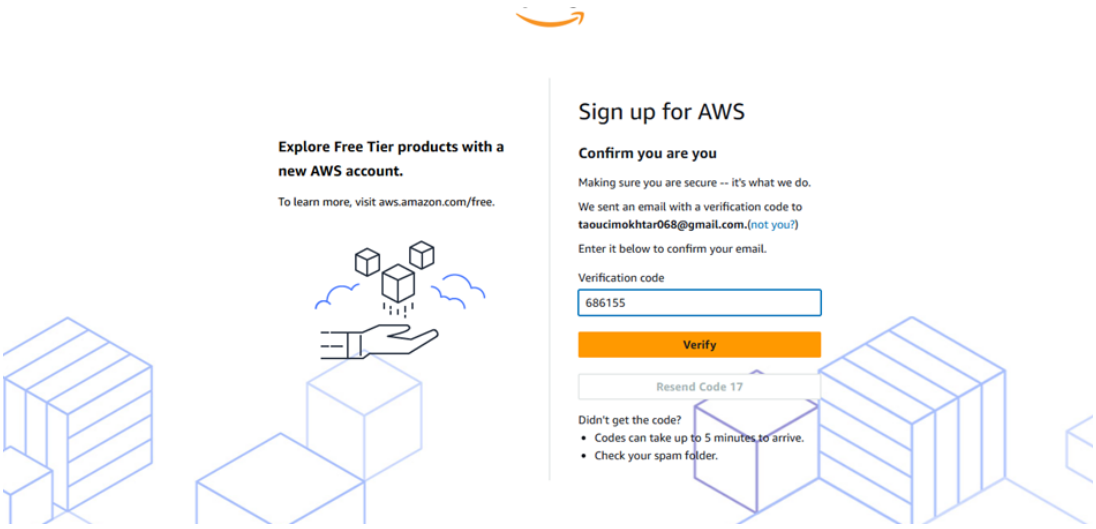


FIG. 4.7 : Vérification du numéro de téléphone

Une fois le compte activé, l'usage du compte root est restreint aux tâches critiques. Un utilisateur IAM a été créé pour une utilisation régulière et sécurisée. Dans ce projet, un

utilisateur nommé admin serverless a été configuré avec les droits AdministratorAccess, permettant la gestion des services AWS et le déploiement des fonctions Lambda.

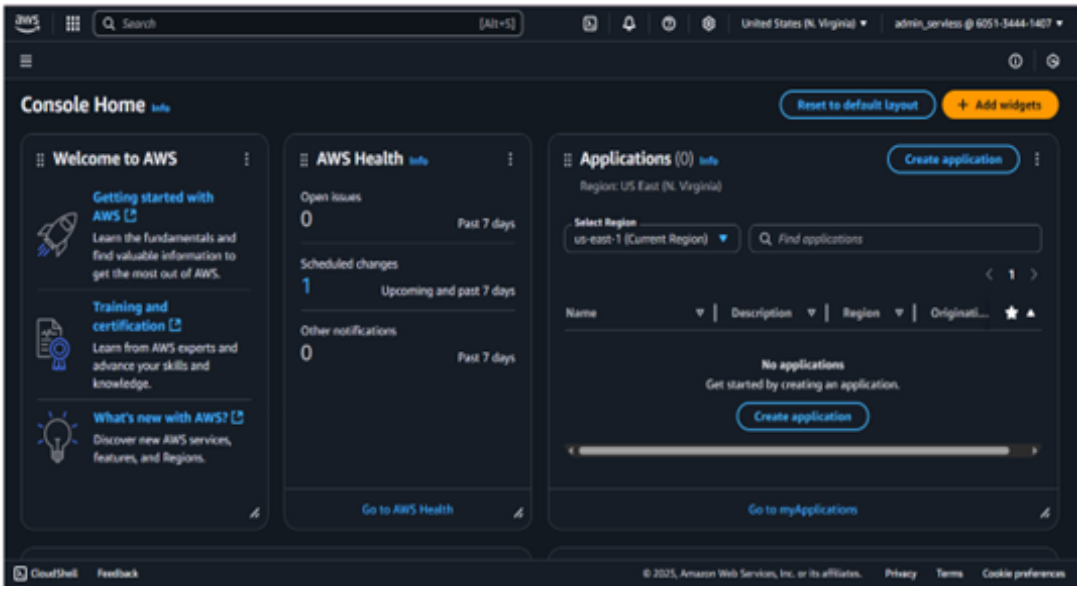


FIG. 4.8 : Aws admin-serverless

4.3.2 Construction du backend avec AWS lambda et intégration des services associés

Dans le cadre de ce projet, AWS Lambda a été adopté comme composant principal pour le traitement asynchrone et fluide des requêtes. Des fonctions Lambda ont été développées, chacune remplissant un rôle spécifique dans le flux de travail du système de messagerie intelligente. Ces fonctions ont été intégrées avec d'autres services d'Amazon tels qu'API Gateway et Dynamo DB. De plus, WebSocket a été utilisé dans l'une des fonctions pour améliorer l'interaction en temps réel.

Création des fonctions AWS Lambda :

Voici les étapes pour créer une fonction AWS Lambda :

- Accéder à la console AWS, puis au service Lambda.
- Choisir "Créer une fonction".
- Sélectionner "Auteur depuis zéro".
- Saisir le nom de la fonction (par exemple : `chatbotfunction`).
- Choisir le langage de programmation (Python 3.10).
- Associer un rôle IAM avec les permissions nécessaires (Dynamo DB, Bedrock…).

- Ajouter ou téléverser le code de la fonction.

Les limites et les performances de lambda gratuit dans l'utilisation de ses ressources pour effectuer toutes les opérations dans AWS sont présentées dans l'image :

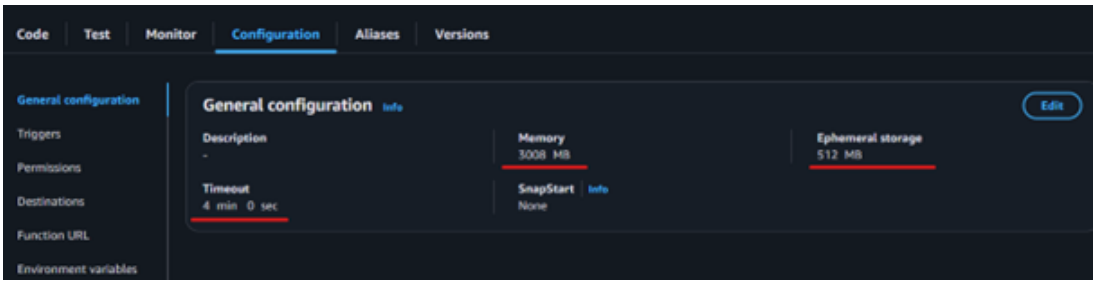


FIG. 4.9 : Augmenter la performance de la fonction lambda

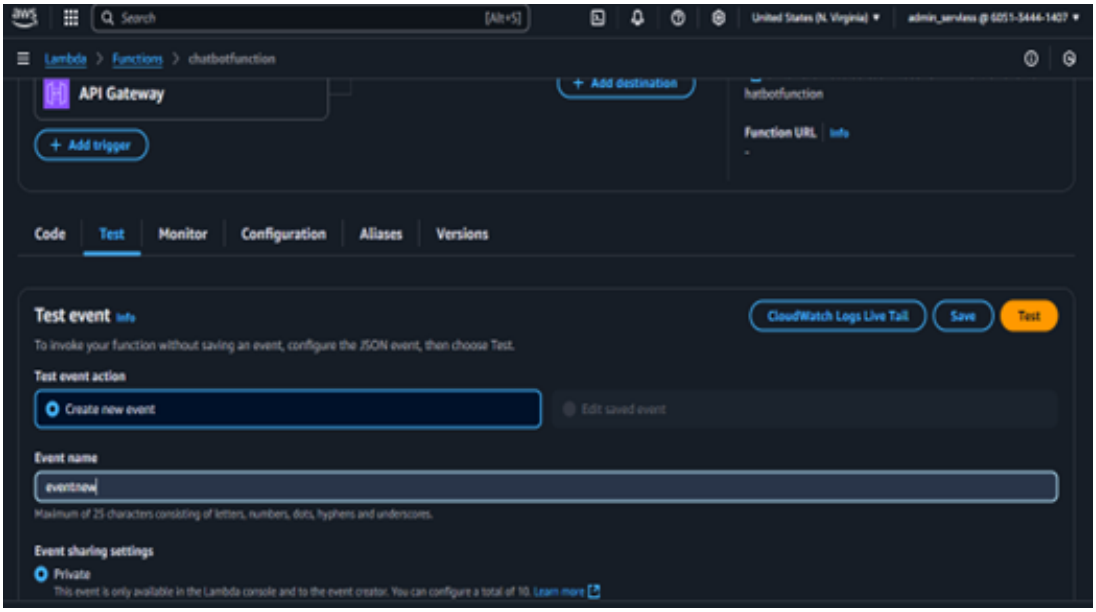


FIG. 4.10 : Test d'un événement pour la fonction lambda

Intégration avec API Gateway :

Des APIs ont été créées avec Amazon API Gateway afin de permettre à l'interface frontale de communiquer avec les fonctions Lambda :

- Créer une API HTTP.
- Définir les routes et les lier aux fonctions Lambda correspondantes.
- Activer le CORS pour permettre l'accès depuis le frontend.
- Déployer l'API et récupérer les URLs finales.

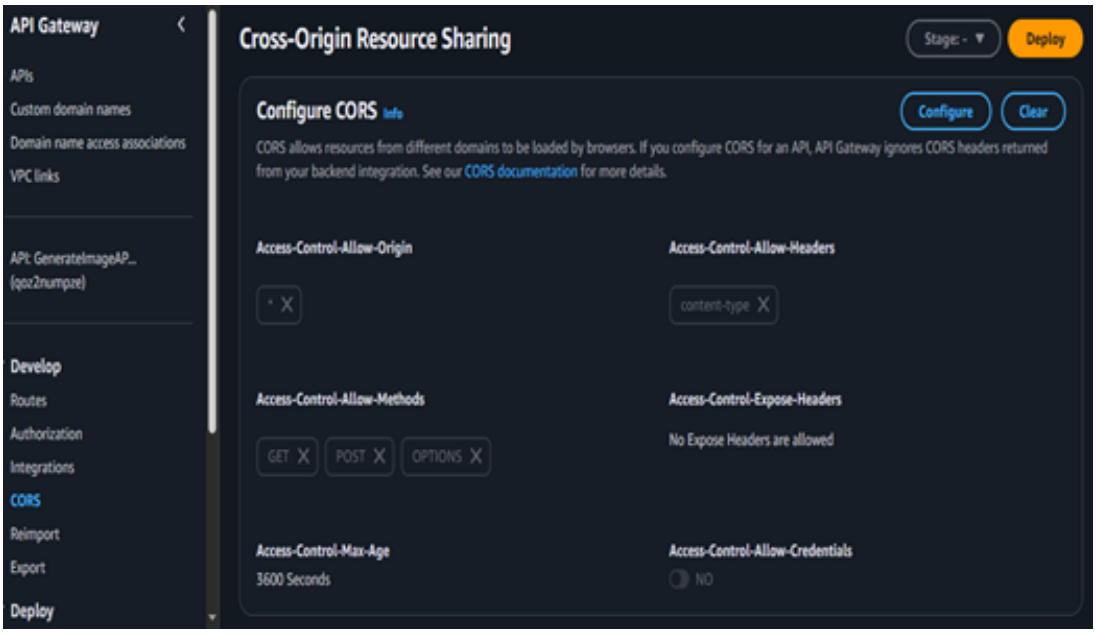


FIG. 4.11 : Configuration des paramétre cors dans api gateway

Utilisation de DynamoDB pour le stockage des données :

- Des tables DynamoDB ont été créées pour stocker les messages et les conversations.
- Connexion entre AWS Lambda et DynamoDB.
- Utilisation de la bibliothèque Python `boto3` pour effectuer les opérations (`put_item`, `query`, ...).

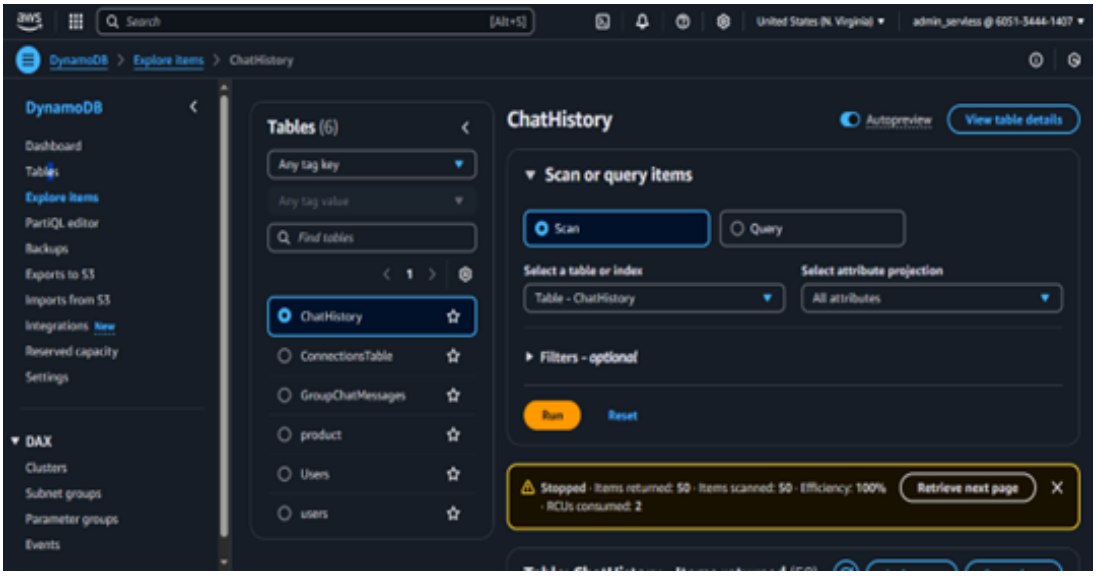


FIG. 4.12 : Table dynamoDB – Interface de gestion

Prise en charge du temps réel avec WebSocket :

Afin d'améliorer l'expérience utilisateur, un canal WebSocket a été mis en place dans les fonction, permettant l'échange de messages en temps réel :

- Création d'une API WebSocket via Amazon API Gateway.
- `$connect` : pour établir la connexion.
- `$disconnect` : pour la terminer.
- `sendMessage` : pour envoyer un message à `chatbotfunction`.
- Stockage du `ConnectionId` dans DynamoDB.
- Utilisation de `postToConnection` dans Lambda pour envoyer la réponse à l'utilisateur connecté.
- Côté client, la connexion a été établie à l'aide de `WebSocket()`.

Cette architecture a permis de mettre en place un système de chat intelligent, entièrement serverless, avec une réponse rapide, un stockage sécurisé et une intégration parfaite avec les outils d'intelligence artificielle d'AWS. La combinaison de Lambda, API Gateway, DynamoDB et WebSocket a assuré un bon rendement et une expérience utilisateur fluide.

4.3.3 Implémentation de la solution basée sur des conteneurs Docker via Amazon ECS de Fargate

Dans le cadre de l'approche alternative à l'architecture Serverless basée sur les fonctions AWS Lambda, une solution conteneurisée a été développée à l'aide de Docker, puis déployée via le service Amazon ECS (Elastic Container Service) en utilisant AWS Fargate, une technologie serverless permettant d'exécuter des conteneurs sans gérer l'infrastructure sous-jacente. Cette approche vise à offrir un meilleur contrôle sur l'environnement d'exécution, une portabilité accrue, ainsi qu'une plus grande flexibilité dans la gestion des dépendances de l'application.

Configuration du projet en local :

Nous avons d'abord installé Docker [12].

Le projet de conteneur PHP a été développé dans le répertoire local suivant :

```
cd C:\xampp\htdocs\codefinalephp
```

Ce répertoire contient le fichier Docker nécessaire à la construction de l'instance. Avant toute opération cloud, l'application PHP a été validée localement via Docker.

Nom de la page	**Le nom de l'API**	**Le nom de fonction Lambda**	**API**	**Le nom de méthode**	**BDD DynamoDB**
`index.html`	Register	registerUser	`https://6btvrzeahe.execute-api.us-east-1.amazonaws.com/prod/registerUser`	Post	Table : `Users`
`index.html`	Login	loginUser	`https://59q04cn1ec.execute-api.us-east-1.amazonaws.com/prod/loginUser`	Post	Table : `Users`
`indexchat.html`	chatbotText	chatbotfunction	`https://bqozj3r1t1.execute-api.us-east-1.amazonaws.com/prod/chatbotfunction`	Post	-
`indexchat.html`	chatbotGet	getUserConversations	`https://bqozj3r1t1.execute-api.us-east-1.amazonaws.com/prod/getUserConversations`	Get	Table : `chatHistory`
`indexpic.html`	GenerateImageAPI	sendImageAPI	`https://q2pmuzpe.execute-api.us-east-1.amazonaws.com/dev/sendImageAPI`	Any	-
`groupchat.html`	GetGroupMessages	getGroupMessages	`https://c86gvbanue.execute-api.us-east-1.amazonaws.com/prod/GetGroupMessages`	Get	Table : `GroupChat Messages`
`groupchat.html`	GroupChatBot	Connect, sendMessage, disconnect	`wss://zdna1syze3.execute-api.us-east-1.amazonaws.com/dev`	`Connect`, `disconnect`, `sendMessage`	`GroupChatMessages`

TAB. 4.1 : Tableau détaillé des outils et des utilisations des services AWS

Préparation locale du projet :

Le projet PHP a été développé dans le répertoire local suivant :

```
cd C:\xampp\htdocs\codefinalephp
```

Ce répertoire contient le fichier Dockerfile nécessaire à la construction de l'image. Avant toute opération cloud, le bon fonctionnement de l'application PHP en local via Docker a été vérifié.

Configuration de l'AWS CLI :

Pour interagir avec les services AWS, l'interface en ligne de commande a été configurée :

```
aws configure
```

Les paramètres saisis :

```
Access Key et Secret Key
```

```
Région : us-east-1
```

```
Format de sortie : json
```

Cela permet de s'authentifier et d'exécuter les commandes AWS à partir de l'environnement local.

Création d'un dépôt Amazon ECR :

Un dépôt privé a été créé dans Amazon Elastic Container Registry (ECR) pour héberger l'image Docker du projet PHP. Cela permet de stocker l'image de manière sécurisée et de la rendre accessible par les services ECS.

```
aws ecr create-repository --repository-name codephpfinale
```

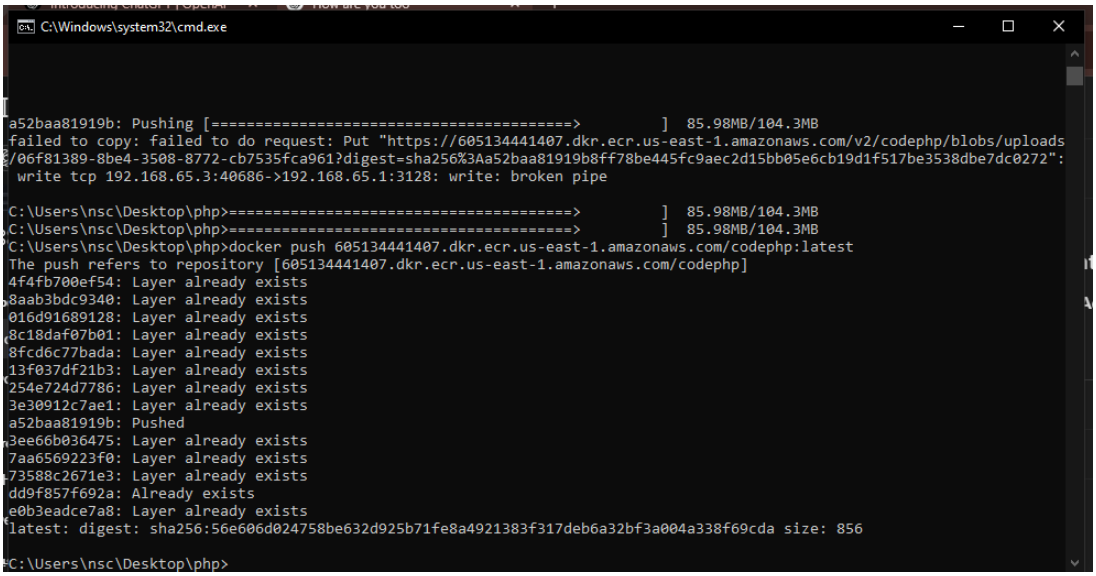


FIG. 4.13 : Création du dépôt privé Amazon ECR

Connexion à ECR, construction et envoi de l'image Docker :

- Authentifier Docker auprès du registre ECR
- Construire l'image Docker localement
- La taguer avec l'URI du dépôt
- L'envoyer (push) vers le dépôt

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS
--password-stdin 605134441407.dkr.ecr.us-east-1.amazonaws.com
```

```
docker build -t codefinalephp .
```

```
docker tag codefinalephp:latest 605134441407.dkr.ecr.us-east-1.amazonaws.com/codefir
```

```
docker push 605134441407.dkr.ecr.us-east-1.amazonaws.com/codefinalephp:latest
```

À l'issue de cette étape, l'image devient disponible pour être utilisée dans un ECS Task et peut être déployée via Fargate sans avoir besoin d'installer un serveur.

Déploiement du conteneur PHP sur Amazon ECS avec Fargate :

Après avoir transféré l'image Docker vers le dépôt Amazon ECR, l'étape suivante a consisté à procéder au déploiement à l'aide du service Amazon ECS, en utilisant le mode Fargate, qui permet l'exécution de conteneurs sans gestion directe des serveurs.

- Création du cluster ECS :
  `aws ecs create-cluster --cluster-name php-cluster`

Ce cluster servira par la suite à exécuter les services hébergés via Fargate.

- Définition de la tâche (Task Definition) :
  Une définition de tâche a été établie afin de spécifier les paramètres techniques d'exécution du conteneur, à savoir :

  ```
  Image utilisée :
  605134441407.dkr.ecr.us-east-1.amazonaws.com/
  codephpfinale:latest

  Rôle d'exécution IAM :
  ecsTaskExecutionRole
  ```

  Cette définition peut être enregistrée soit via la console AWS, soit à l'aide d'un fichier JSON en exécutant la commande suivante :

  ```
  aws ecs register-task-definition
  --cli-input-json file://task-definition.json
  ```

- Lancement du service avec Fargate :
  Une fois la tâche définie, un service a été lancé dans le cluster afin d'exécuter le conteneur en utilisant Fargate, via la commande ci-dessous :

  ```
  aws ecs create-service
  --cluster php-cluster
  --service-name php-fargate-service
  --task-definition codephpfinale-task
  --desired-count 1
  --launch-type FARGATE
  --network-configuration
  "awsvpcConfiguration={
  subnets=[subnet-xxxxxxx],
  securityGroups=[sg-xxxxxxx],
  assignPublicIp=ENABLED}"
  ```

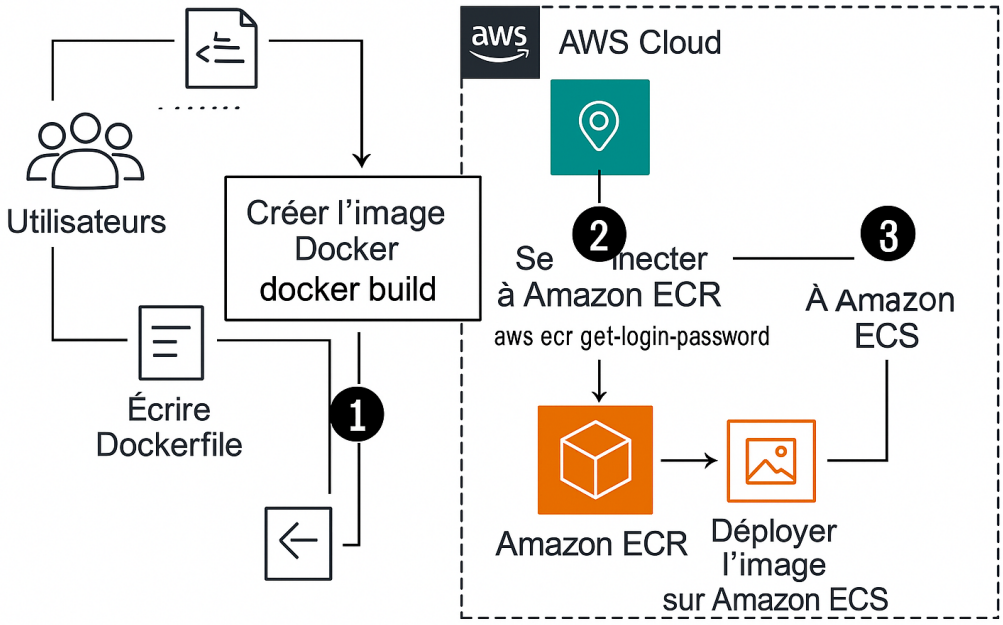


FIG. 4.14 : Architecture de conteneurisation

Le déploiement d'une application conteneurisée avec Docker sur AWS ECS via Fargate offre une solution moderne, scalable et entièrement gérée pour l'hébergement d'applications web. Grâce à l'intégration avec Amazon ECR, la gestion des images devient fluide, tandis que Fargate élimine les contraintes liées à l'infrastructure, permettant aux développeurs de se concentrer pleinement sur leur code.

Chapitre 5

Évaluation

5.1 Introduction

Dans ce chapitre, nous effectuons une évaluation pratique des performances des deux sites sur deux systèmes de déploiement d'applications Web différents : Le premier a été publié sur la plateforme Netlify (https ://gentle-cannoli- 64f1a5.netlify.app), qui est une plateforme sans serveur. La deuxième est hébergé sur un conteneur Docker puis déployé via Amazon ECS (Elastic Container Service), accessible à l'adresse (http ://98.82.18.215:8080).

5.2 Méthodologie de test

Afin de comparer objectivement les performances des deux approches d'hébergement l'architecture serverless basée sur AWS Lambda, et l'architecture conteneurisée via Amazon ECS (Docker) nous avons mis en place une série de tests de charge automatisés à l'aide d'un script Python. Ce script simule plusieurs requêtes consécutives envoyées vers les deux versions du site, tout en mesurant divers indicateurs clés de performance. Les outils utilisés sont : le langage Python, avec les bibliothèques de requêtes, de temps et de statistiques.

5.3 Evaluation

5.3.1 Les données collectées et utilisée

- **Latencies** : les temps de réponse individuels de chaque requête (en millisecondes)
- **Total_sent** : le nombre total de requêtes envoyées
- **Total_success** : le nombre de réponses valides reçues (statut HTTP 200)
- **Average_time** : le temps de réponse moyen

- **Min_time, Max_time** : les temps de réponse minimum et maximum observés
- **Std_deviation** : l'écart-type des latences (pour évaluer la stabilité)
- **Cold_start_count** : nombre de démarrages à froid observés (hypothèse simulée)
- **Throughput_rps** : débit en requêtes par seconde
- **Total_duration** : la durée totale du test

Exemple de sortie obtenue sur un fichier. Json :


```
{
"latencies": [30.4, 80.1, 65.3, 98.4, 54.7],
"total sent": 5,
"total success": 5,
"average time": 65.61,
"min_time": 30.40,
"max_time": 98.46,
"std_deviation": 26.25,
"cold_start_count": 5,
"throughput_rps": 0.0508,
"total_duration": 98.46
}
```


Ces tests ont été réalisés plusieurs fois, dans des conditions similaires, sur les deux plateformes afin d'obtenir des résultats fiables et comparables et de sélectionner le meilleur.

5.3.2 Démarrage a froid (cold start)

Ce qui constitue l'un des principaux avantages de cette architecture Serverless peut aussi constituer son principal inconvénient. Disposer de fonctions éphémères est un atout majeur, Car cela réduit les coûts, mais, en revanche, leur temps de réponse sera plus long que d'habitude. Ce phénomène ne se produira que lors de la première requête, lorsque la fonction est encore inactive, mais il est néanmoins inévitable.

5.3.3 Latence(latencies)

Ce deuxième inconvénient est lié au premier. Il existe la latence évoquée dans la section précédente, mais aussi la latence réseau. La machine hébergeant notre service pouvant être située n'importe où dans le monde, il est important de prendre en compte son impact potentiel sur les performances de l'application et le temps de réponse. Le temps de réponse de la fonction correspond à la somme du temps nécessaire à la requête pour voyager du

client à la plateforme Serverless, du temps de traitement de la fonction et du temps de réponse au client. Le fait que le service s'exécute sur une machine pouvant héberger plusieurs autres ressources parallèlement peut entraîner une course aux ressources de calcul de la machine, telles que la mémoire et le processeur, ce qui peut également contribuer à une latence accrue.

5.3.4 Throughput (Débit)

Le throughput, ou débit, représente le nombre de requêtes traitées avec succès par seconde (requêtes/seconde, ou RPS : Requests Per Second). Il s'agit d'un indicateur fondamental de performance, particulièrement dans les tests de charge. Un débit élevé indique que la plateforme peut gérer un grand volume de trafic sans ralentissement notable.

5.4 Résultats des tests

Dans cette section, nous présentons les résultats détaillés des tests de performance effectués sur les deux plateformes déployées : l'une basée sur une architecture **Serverless** (Netlify avec AWS Lambda), et l'autre hébergée via **Docker** dans Amazon ECS.

L'objectif principal est d'évaluer leur comportement face à des charges simultanées, en se concentrant sur trois fonctionnalités clés de l'application :

- Test de **génération d'image** : évalue le temps de réponse pour la création d'images via des modèles d'IA.
- Test de **chat de groupe** : mesure la réactivité de la fonction de discussion en groupe.
- Test de **chat individuel** : analyse la rapidité du traitement dans une conversation en tête-à-tête.

Pour cela, nous avons utilisé un **script Python personnalisé** qui envoie plusieurs requêtes vers chaque service, en parallèle, et enregistre divers indicateurs de performance comme la latence, le débit (*throughput*), et le nombre de démarrages à froid.

Ce script est basé sur les bibliothèques requests,concurrent.futures,statistics et time.

Les résultats collectés sont ensuite stockés dans un pour faciliter l'analyse statistique et la génération de graphiques.

5.4.1 Teste de génération d'image

Ce test vise à mesurer les performances de la fonctionnalité de **génération d'image** sur deux plateformes distinctes :

- **Architecture Serverless :** Netlify + AWS Lambda
- **Architecture de conteneur :** Amazon ECS avec Docker + PHP

▶ **Le code Python :**

```python
import requests
import time
import json
import random
import statistics
from concurrent.futures import ThreadPoolExecutor, as_completed
URLS = {
    "lambda_picture": "https://qoz2numpze.execute-api.us-east-1.amazonaws.com/dev/lambdapicture",
    "php_picture": "http://98.82.18.215:8080/generate.php"
}
TOTAL_REQUESTS = 15
TIMEOUT = 100

def send_request(url, simulate_instability=False):
    if simulate_instability and random.random() < 0.1:
        time.sleep(random.uniform(0.5, 1.5))
    start = time.time()
    try:
        if 'lambdapicture' in url:
            payload = {
                "prompt": "A beautiful futuristic cityscape at sunset",
                "model": "stable-diffusion-v2"
            }
            response = requests.post(url, json=payload, timeout=TIMEOUT)
        else:
            payload = {
                "prompt": "مدينة مستقبلية جميلة عند غروب الشمس",
                "model": "stable-diffusion-v2"
            }
            response = requests.post(url, data=payload, timeout=TIMEOUT)

        latency = time.time() - start
        return round(latency, 4) if response.status_code == 200 else None
    except Exception:
        return None

def benchmark(name, url):
    print(f"\n⏳ Sending {TOTAL_REQUESTS} requests to {name.upper()}...")
    results = []
    simulate_instability = (name == "php_picture")
    start_time = time.time()

    with ThreadPoolExecutor(max_workers=100) as executor:
        futures = [
            executor.submit(send_request, url, simulate_instability)
            for _ in range(TOTAL_REQUESTS)
        ]
        completed = 0
        for i, future in enumerate(as_completed(futures), 1):
            result = future.result()
            if result is not None:
                results.append(result)
            completed += 1
            if i % 50 == 0:
                print(f"{i} requests completed...")

    end_time = time.time()
    total_time = end_time - start_time
    total_success = len(results)
    cold_starts = len([r for r in results if r > 1.5])
    stats = {
        "latencies": results,
        "total_sent": TOTAL_REQUESTS,
        "total_success": total_success,
        "average_time": round(statistics.mean(results), 4) if results else None,
        "min_time": round(min(results), 4) if results else None,
```

FIG. 5.1 : Code Python pour le test de génération d'images

```
        "max_time": round(max(results), 4) if results else None,
        "std_deviation": round(statistics.stdev(results), 4) if len(results) > 1 else 0,
        "cold_start_count": cold_starts,
        "throughput_rps": round(total_success / total_time, 4) if total_time > 0 else 0,
        "total_duration": round(total_time, 4)
    }
    return stats

if __name__ == "__main__":
    all_results = {}
    for name, url in URLS.items():
        all_results[name] = benchmark(name, url)

    with open("results_picture2.json", "w", encoding="utf-8") as f:
        json.dump(all_results, f, indent=2)

    print("\n✅ Results saved to results_picture2.json")
```

FIG. 5.2 : suite de Code Python pour le test de génération d'images

• Exécution :

On Visual Studio Code :

Lancer localement : **`python -m http.server 8000`**

On PowerShell :

Le test a été exécuté localement avec la commande : **`python benchmarkpic2.py`**

```
PS C:\Users\Abdoo\benchmark_ready2> python benchmarkpic2.py
📤 Sending 5 requests to LAMBDA_PICTURE...
📤 Sending 5 requests to PHP_PICTURE...
💾 Results saved to results_picture2.json
```

FIG. 5.3 : Démarrage de l'exécution de la création d'image dans PowerShell

Le résultat est sauvegardé dans le fichier : resultspicture2.json

```
{ "lambda_picture": {
"latencies": [
20.2693,
26.5456 ],
"total_sent": 15,
5 0-0

Chapitre 5 Evaluation
36
"total_success": 2,
"average_time": 23.4075,
"min_time": 20.2693,
"max_time": 26.5456,
"std_deviation": 4.438,
"cold_start_count": 2,
"throughput_rps": 0.0638,
"total_duration": 31.3347 },
"php_picture": {
"latencies": [
30.7787,
60.5959,
75.8899
],
"total_sent": 15,
"total_success": 3,
"average_time": 55.7548,
"min_time": 30.7787,
"max_time": 75.8899,
"std_deviation": 22.9419,
"cold_start_count": 3,
"throughput_rps": 0.0296,
"total_duration": 101.4307
}
}
```


FIG. 5.4 : resultspicture2.json

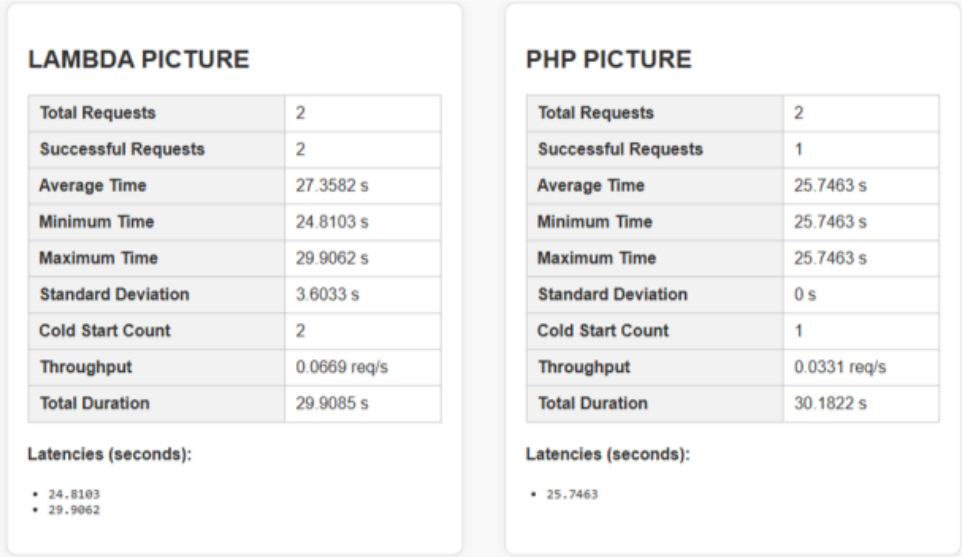
Tableau collecter les données des résultats de différent teste :

Génération image / Evaluation	**FAAS (Serverless)**	**PAAS (conteneur)**
Requête = 1 TimeOut = 20	1	1
Requête = 2 TimeOut = 3	2	1
Requête = 4 TimeOut = 50	2	1
Requête = 5 TimeOut = 50	2	2
Requête = 10 TimeOut = 100	2	4
Requête = 15 TimeOut = 100	2	3

TAB. 5.1 : Réponses réussies de la génération d'images via FaaS et PaaS

- Pour voir le tableau suivant, nous avons fait l'exécution suivante :

`http://localhost:8000/picture_stats.html`



LAMBDA PICTURE

Total Requests	2
Successful Requests	2
Average Time	27.3582 s
Minimum Time	24.8103 s
Maximum Time	29.9062 s
Standard Deviation	3.6033 s
Cold Start Count	2
Throughput	0.0669 req/s
Total Duration	29.9085 s

Latencies (seconds):

- 24.8103
- 29.9062

PHP PICTURE

Total Requests	2
Successful Requests	1
Average Time	25.7463 s
Minimum Time	25.7463 s
Maximum Time	25.7463 s
Standard Deviation	0 s
Cold Start Count	1
Throughput	0.0331 req/s
Total Duration	30.1822 s

Latencies (seconds):

- 25.7463

FIG. 5.5 : Comparaison après l'exécution entre les versions sans serveur et conteneurisées

Sur la base du fichier " resultspicture2.json ", un tableau est conçu à l'aide de HTML pour montrer la comparaison entre les performances des services Lambda (FAAS) et PHP(PAAS) dans la génération d'images. Le tableau affiche, de manière simple et directe, les indicateurs les plus importants, tels que : le nombre total de requêtes, le nombre de requêtes réussies, le temps de réponse moyen, le temps minimum et maximum, le nombre de démarrages à froid et enfin le débit .Pour voir le tableau suivant, nous avons faire l'exécution suivante : `http://localhost:8000/diagramme2.html`

1. Succès total (Total Success) : Les graphiques montrent que les performances du conteneur se dégradent progressivement à mesure que le nombre de requêtes augmente, tout comme celles de Lambda, s'améliorant parfois avec des temps de réponse beaucoup plus longs. En revanche, les réponses d'AWS Lambda restent stables même avec des volumes de requêtes importants, les deux systèmes rejetant la majorité des requêtes. Il est donc clair que le conteneur s'est avéré plus efficace, stable et performant que Lambda, et que Lambda est plus performant et plus rapide avec de faibles volumes de requêtes, car Lambda est soumis à certaines contraintes de performances.

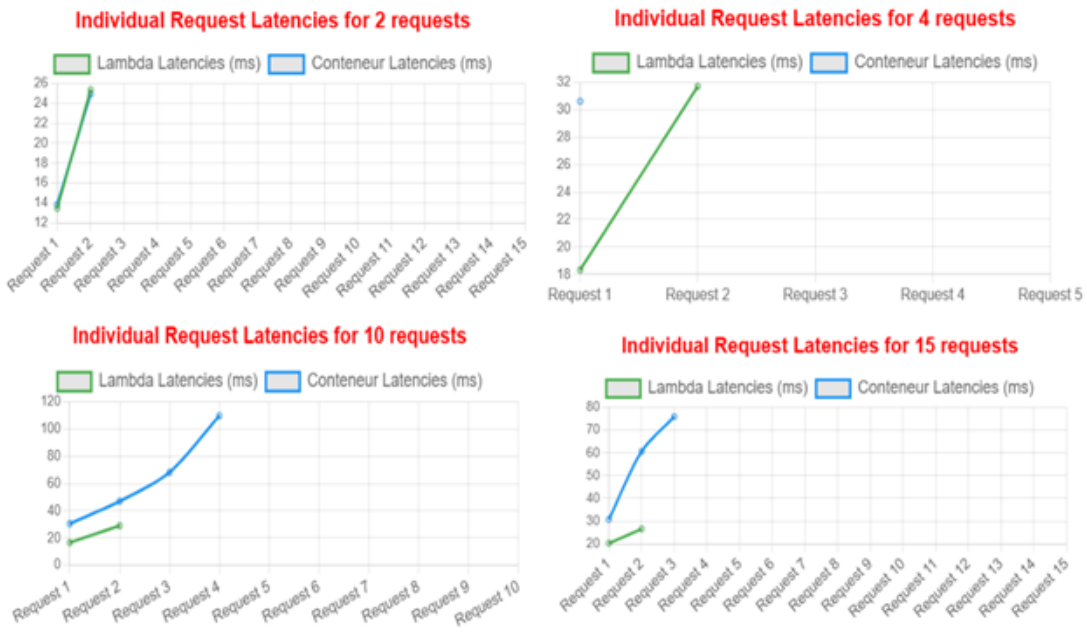


FIG. 5.6 : réussite globale dans différentes implémentations en Génération d'image

2. Démarrage à froid (Cold start) : En analysant les graphiques de démarrage à froid, nous constatons que Lambda (FaaS) et les conteneurs (PaaS) subissent fréquemment ce type de démarrage à froid, ce qui impacte les résultats en rejetant de nombreuses requêtes. Cela s'explique par le temps de création de l'image, qui dépasse souvent le seuil de 13,5 secondes défini comme critère de démarrage à froid. De plus, les limitations de performances inhérentes à l'environnement d'exécution, qu'il s'agisse de Lambda ou de conteneurs exécutant PHP, contribuent à ces cas. Cela démontre que la consommation élevée de ressources et le chargement initial du modèle sont deux facteurs courants qui impactent négativement les temps de réponse dans les deux cas.

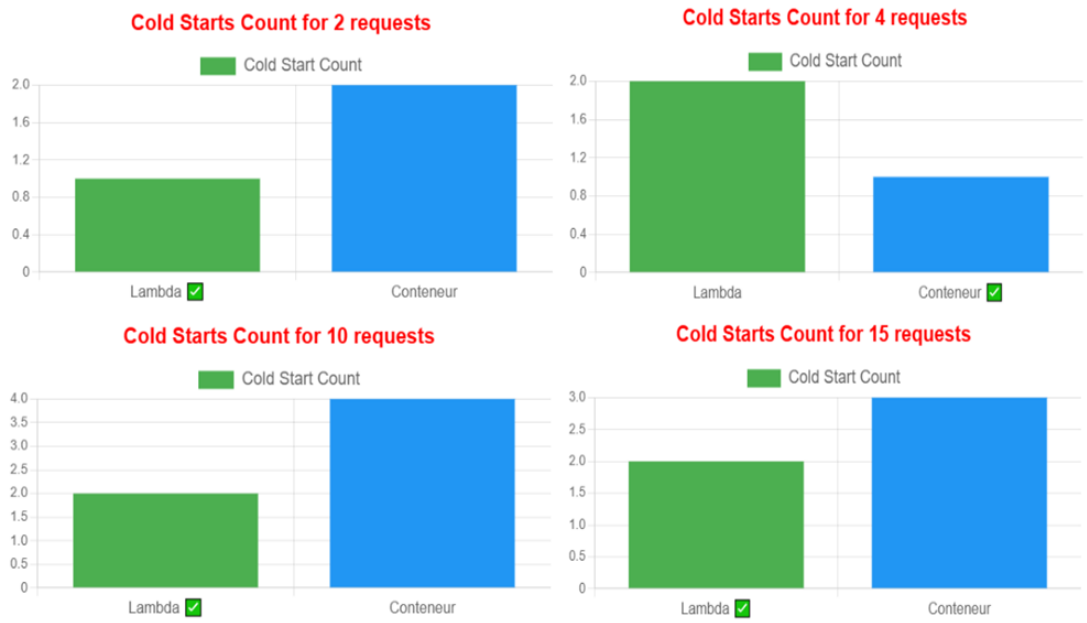


FIG. 5.7 : démarrage à froid dans différentes implémentations en génération d'image

3. Débit (throughput) : En comparant les valeurs de débit, nous constatons que les graphiques indiquent la supériorité de Lambda en termes de vitesse (requêtes/seconde), mais ces indicateurs ne suffisent pas à eux seuls à en juger. Le taux de réussite et le taux de réponse réels doivent également être pris en compte. Container s'est avéré plus efficace pour traiter de manière fiable un grand nombre de requêtes, même avec de légers ralentissements. Lambda n'a surpassé les performances que dans deux expériences, conservant son équilibre et sa vitesse. Il est à noter que les deux ont refusé d'exécuter de nombreuses requêtes. De plus, le temps important nécessaire à la création de l'image a un impact direct sur le débit : plus le temps de traitement est long, plus le nombre de requêtes traitées par seconde est faible.

FIG. 5.8 : Débit dans différentes exécutions en génération d'image

4. Temps de réponse moyen : Les performances virtuelles ne reflètent pas toujours une fiabilité totale. En comparant les temps de réponse moyens, AWS Lambda (l'approche sans serveur) semble offrir des temps de réponse plus rapides, mais l'approche par conteneur est supérieure, car Lambda n'a pas répondu à un grand nombre de requêtes, contrairement à Conteneur. Cependant, il est important de noter que le problème de démarrage à froid affecte Lambda, en particulier pour les fonctions rarement appelées. Ce délai initial de démarrage peut avoir un impact négatif sur les performances en cas d'utilisation intermittente ou d'augmentation soudaine des requêtes. Dans ce cas, Conteneur est donc le meilleur car il a répondu à un plus grand nombre de requêtes.

FIG. 5.9 : temps de réponse moyen dans différentes implémentations en génération d'image

5.4.2 Test de groupe de discussion

Ce test vise à mesurer les performances de la fonctionnalité de chat de groupe sur les deux architectures étudiées :

- Le service déployé sur Netlify avec Lambda (serverless)

- Le service hébergé via Docker sur Amazon ECS (PHP)

L'objectif est d'évaluer la réactivité de l'envoi de messages dans un groupe, un cas d'usage réel impliquant plusieurs utilisateurs simultanés. . Exécution :

On Visual Studio Code :

Lancer localement : `python -m http.server 8000`

On PowerShell :

Le test a été exécuté localement avec la commande :

```
python benchmarkgrp_chat2.py
```

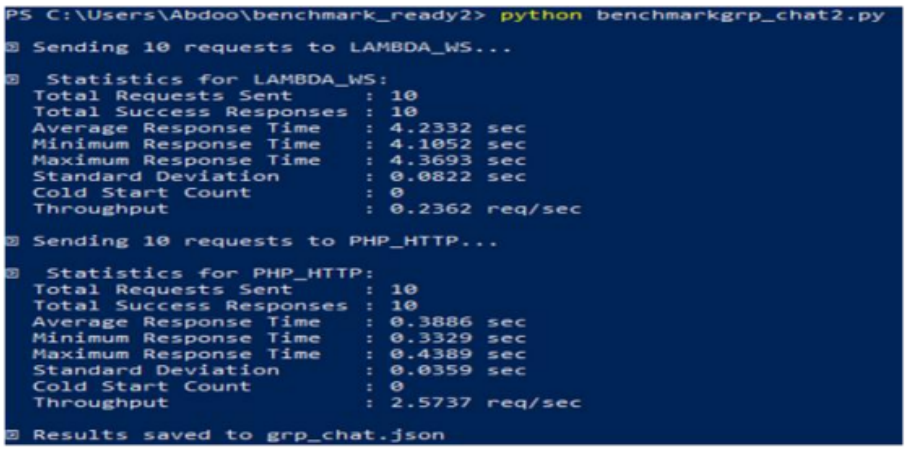


FIG. 5.10 : Démarrage de l'exécution du groupe de discussion dans PowerShell

Tableau collecter les données des résultats de différent teste :

API DE chat Groupe	**FAAS (Serverless lambda)**	**PAAS (PHP)**
Requête = 10 TimeOut = 3	10	10
Requête = 50 TimeOut = 3	50	46
Requête = 80 TimeOut = 7	79	78
Requête = 150 TimeOut = 9	106	65
Requête = 200 TimeOut = 10	163	133
Requête = 250 TimeOut = 14	181	177
Requête = 250 TimeOut = 30	233	233

TAB. 5.2 : Réponses réussies des groupes de discussion FaaS (sans serveur) et PaaS (conteneur)

Pour voir le tableau suivant, nous avons faire l'exécution suivante : :

`http://localhost:8000/resultsgroupchat2.html`

FIG. 5.11 : Comparaison après exécution entre serverless et conteneur

Sur la base du fichier "grpchat.json ", un tableau est conçu à l'aide de HTML pour montrer la comparaison entre les performances des services Lambda (FAAS) et PHP(PAAS) dans chat Groupe. Le tableau affiche, de manière simple et directe, les indicateurs les plus importants, tels que : le nombre total de requêtes, le nombre de requêtes réussies, le temps de réponse moyen, le temps minimum et maximum, le nombre de démarrages à froid et enfin le débit.

1. Succès total (Total Success) : Le graphique met en évidence une différence notable entre les performances du conteneur PHP et celles d'AWS Lambda. On observe que PHP offre des temps de réponse plus rapides, en particulier lors des premières requêtes, tandis que Lambda subit un ralentissement dû au démarrage à froid. Au fur et à mesure que le volume des requêtes augmente, les deux solutions montrent des signes de dégradation, mais PHP reste globalement plus stable. Cela dit, il est important de noter que Lambda a réussi à répondre à l'ensemble des requêtes, contrairement au conteneur PHP, qui a rencontré des échecs sur certaines d'entre elles. Cette fiabilité côté Lambda compense en partie ses latences plus élevées.

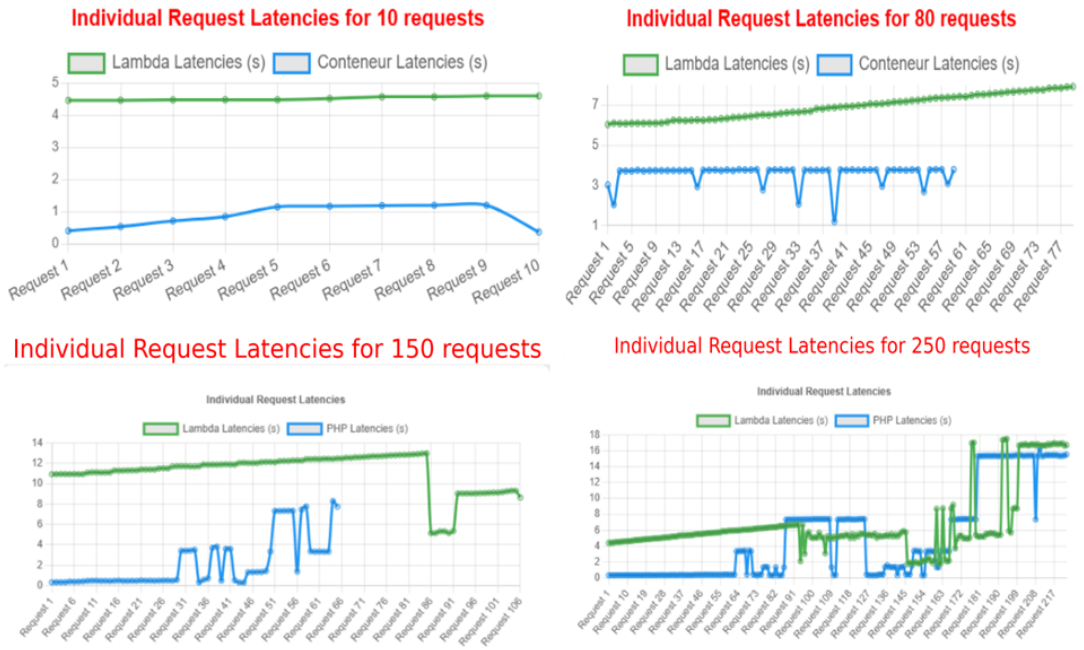


FIG. 5.12 : réussite globale pour différentes implémentations de groupes de discussion

Le graphique met en évidence une différence notable entre les performances du conteneur PHP et celles d'AWS Lambda. On observe que PHP offre des temps de réponse plus rapides, en particulier lors des premières requêtes, tandis que Lambda subit un ralentissement dû au démarrage à froid. Au fur et à mesure que le volume des requêtes augmente, les deux solutions montrent des signes de dégradation, mais PHP reste globalement plus stable. Cela dit, il est important de noter que Lambda a réussi à répondre à l'ensemble des requêtes, contrairement au conteneur PHP, qui a rencontré des échecs sur certaines d'entre elles. Cette fiabilité côté Lambda compense en partie ses latences plus élevées.

2. Démarrage à froid (Cold start) : Dans notre étude, nous avons pris en compte une durée moyenne estimée de 5 secondes pour chaque démarrage à froid (cold start), ce qui permet de mieux interpréter les résultats obtenus. Les graphiques montrent clairement que le nombre de démarrages à froid dans les fonctions AWS Lambda est beaucoup plus élevé par rapport à une application PHP conteneur. Dans chaque test, le nombre de démarrages à froid dans Lambda dépassait 120, alors qu'il restait constant ou faible dans l'application PHP.

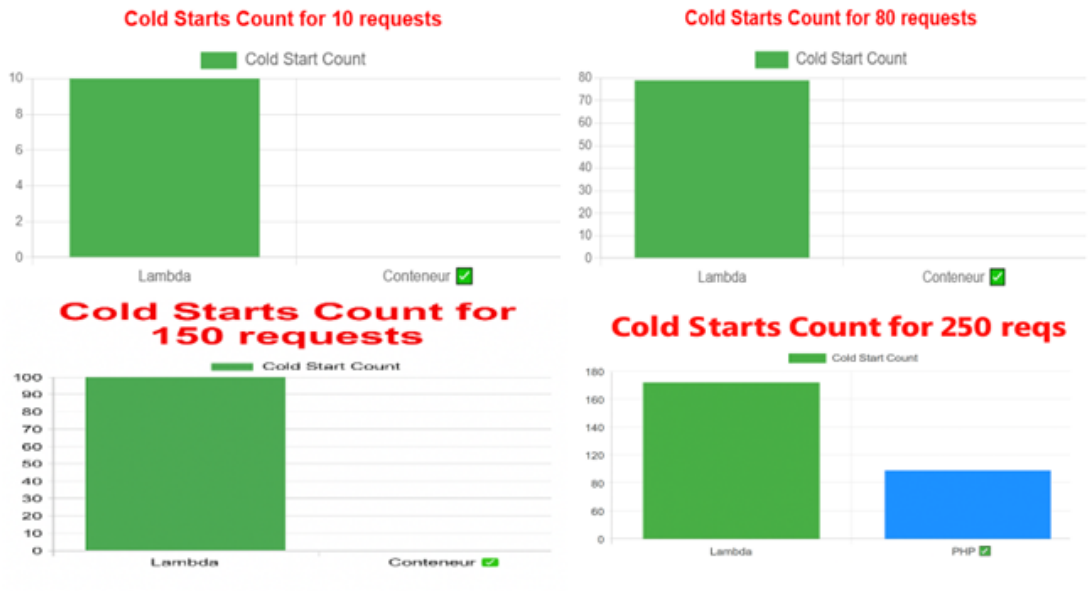


FIG. 5.13 : démarrage à froid dans différentes implémentations dans le groupe de discussion

Cet écart dans le nombre de démarrages à froid entraîne un temps de réponse accru et un débit inférieur dans Lambda, ce qui a été observé lors des tests de performances. La principale raison en est la nature asynchrone et événementielle de Lambda, où des conteneurs sont créés à chaque nouvelle exécution, à moins que des paramètres spéciaux dans AWS tels que la concurrence provisionnée ne soient utilisés, ce qui est coûteux.

3. Débit (throughput) : Les graphiques montrent que l'approche conteneur a un débit plus élevé (requêtes par seconde) que l'application AWS Lambda, approchant 0,19 contre 0,14 pour Lambda. Cette différence est due au nombre de démarrages à froid dans Lambda, qui dépassait 160 dans certains tests, alors qu'il restait inférieur à 100 dans l'approche conteneur.

FIG. 5.14 : Débit dans différentes implémentations dans le groupe de discussion

Cette différence met en évidence que Lambda , bien que puissant en termes de mise à l'échelle automatique et de flexibilité, souffre de latence lors de l'appel de fonctions non initialisées, ce qui a un impact négatif sur le débit global. En revanche, l'approche conteneur s'exécutent dans un environnement stable et toujours actif, ce qui réduit le temps de démarrage et augmente l'efficacité du traitement.

4 Temps de réponse moyen (average time) : Les graphiques montrent que le temps de réponse moyen de Lambda est supérieur à celui d'une application PHP conteneurisée, approchant 7 secondes dans tous les cas, alors qu'il ne dépasse pas 6 secondes dans l'environnement PHP. Cela s'explique par le fait que Lambda répond à de nombreuses requêtes et que le système de conteneurs est moins réactif que Lambda. Lambda présente ici l'avantage de répondre efficacement aux requêtes volumineuses et de s'adapter automatiquement, tandis que son temps d'exécution est plus court que celui d'un système de conteneurs.

FIG. 5.15 : temps de réponse moyen pour différentes implémentations de groupes de discussion

5.4.3 Test de chat

Dans cette section, nous exécutons une série de tests pratiques sur la fonction de chat, visant à évaluer ses performances et à les comparer entre deux environnements différents : l'utilisation des services AWS Lambda (application sans serveur) d'une part, et l'exécution d'une application PHP dans un conteneur Docker d'autre part. La fonctionnalité de chat est un composant essentiel de nombreuses applications interactives. Il est donc essentiel de mesurer avec précision ses performances pour comprendre la réactivité et la stabilité d'un système soumis à des contraintes.

Exécution :

On Visual Studio Code :
Lancer localement : `python -m http.server 8000`

On PowerShell :
Le test a été exécuté localement avec la commande :
`python benchmarkchat.py`

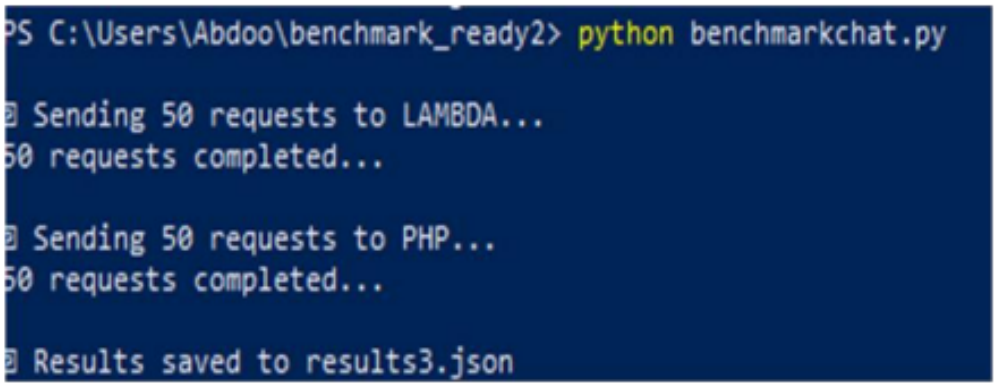


FIG. 5.16 : Démarrage de l'exécution de chat dans PowerShell

Tableau collecter les données des résultats de différent teste :

API DE Chat	**FAAS (Serverless lambda)**	**PAAS (PHP)**
Requete = 10 TimeOut = 2	10	10
Requete = 50 TimeOut = 4	50	24
Requete = 80 TimeOut = 8	80	71
Requete = 150 TimeOut = 12	150	100
Requete = 200 TimeOut = 15	150	14
Requete = 250 TimeOut = 15	250	68
Requete = 300 TimeOut = 20	300	177

TAB. 5.3 : Réponses réussies de Chat FaaS (Serverless) et PaaS (Conteneur)

Pour voir le tableau suivant, nous avons faire l'exécution suivante : `http://localhost:8000/chatdiaaffi.html`

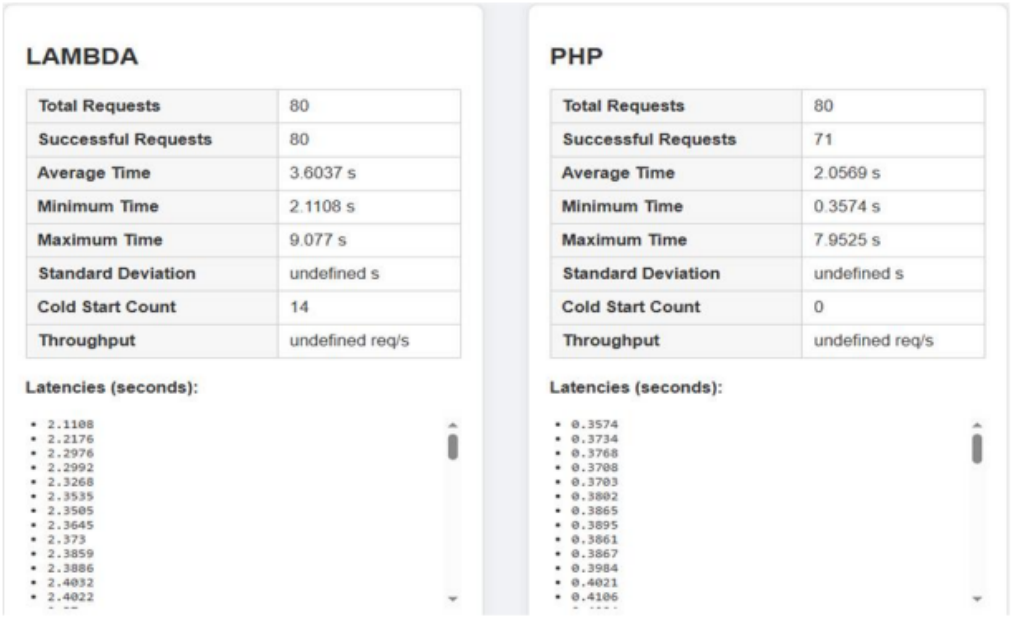


FIG. 5.17 : Comparaison après exécution en chat entre serverless et conteneur

Sur la base du fichier "results9.json ", un tableau est conçu à l'aide de HTML pour montrer la comparaison entre les performances des services Lambda (FAAS) et PHP(PAAS) dans chat. Le tableau affiche, de manière simple et directe, les indicateurs les plus importants, tels que : le nombre total de requêtes, le nombre de requêtes réussies, le temps de réponse moyen, le temps minimum et maximum, le nombre de démarrages à froid et enfin le débit.

1. Succès total (Total Success) : Les graphiques comparent les temps de réponse des conteneurs PHP et AWS Lambda sous différentes contraintes. Les résultats montrent que PHP a obtenu de meilleurs temps de réponse avec moins de requêtes traitées, mais qu'il a commencé à montrer des signes de ralentissement et a rejeté de nombreuses requêtes en grand nombre. Lambda a également démontré un temps de réponse plus élevé, une meilleure évolutivité et un taux de réponse plus élevé.

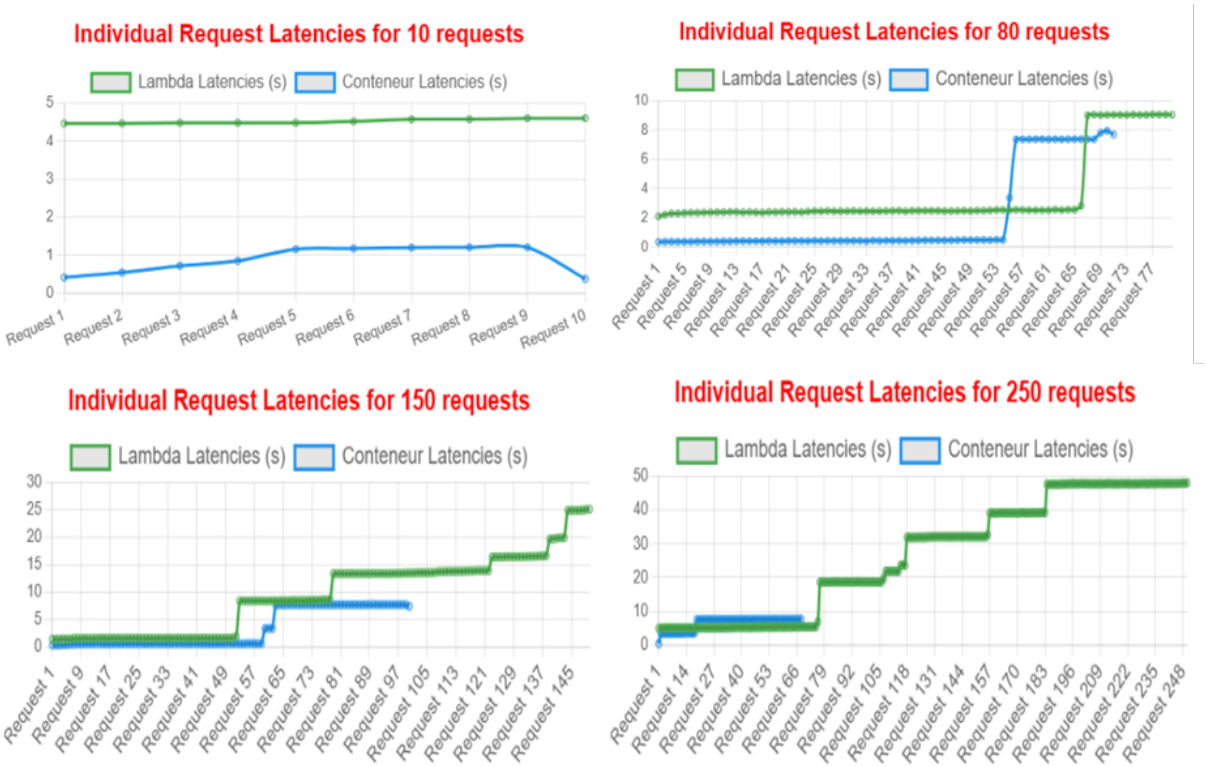


FIG. 5.18 : réussite globale dans différentes implémentations en chat

2. Démarrage à froid (Cold start) : Les graphiques montrent que les fonctions AWS Lambda subissent de fréquentes initialisations à froid à mesure que le nombre de requêtes augmente : 14 initialisations ont été enregistrées pour 50 requêtes, environ 100 pour 100 requêtes et plus de 240 pour 150 requêtes. En revanche, aucune initialisation à froid n'a été enregistrée au niveau du conteneur, ce qui témoigne de performances stables. Notez qu'un grand nombre de requêtes ont été rejetées.

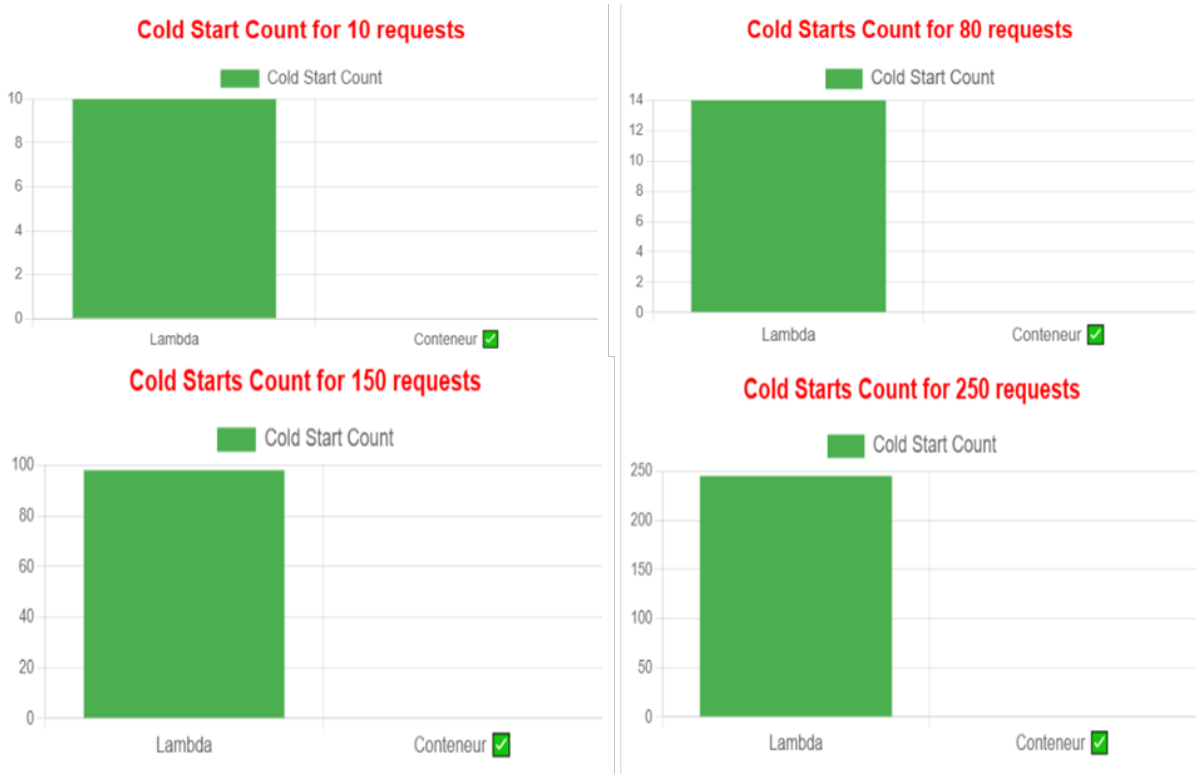


FIG. 5.19 : démarrage à froid dans différentes implémentations dans le chat

Ceci met en évidence l'un des principaux défis de Lambda, compte tenu de son volume élevé de requêtes : les initialisations à froid ont un impact négatif sur le temps de réponse global.

3. Débit (throughput) : Les graphiques mettent en évidence le contraste de débit entre AWS Lambda (FaaS) et PHP dans un environnement conteneurisé (PaaS), mesuré en requêtes par seconde (RPS). En conditions de faible pression et à mesure que la pression augmente, Lambda surperforme progressivement, enregistrant jusqu'à 9 requêtes par seconde grâce à sa capacité de mise à l'échelle horizontale automatique. Le conteneur ne répond pas, ignore un grand nombre de requêtes et ne peut pas gérer la pression.

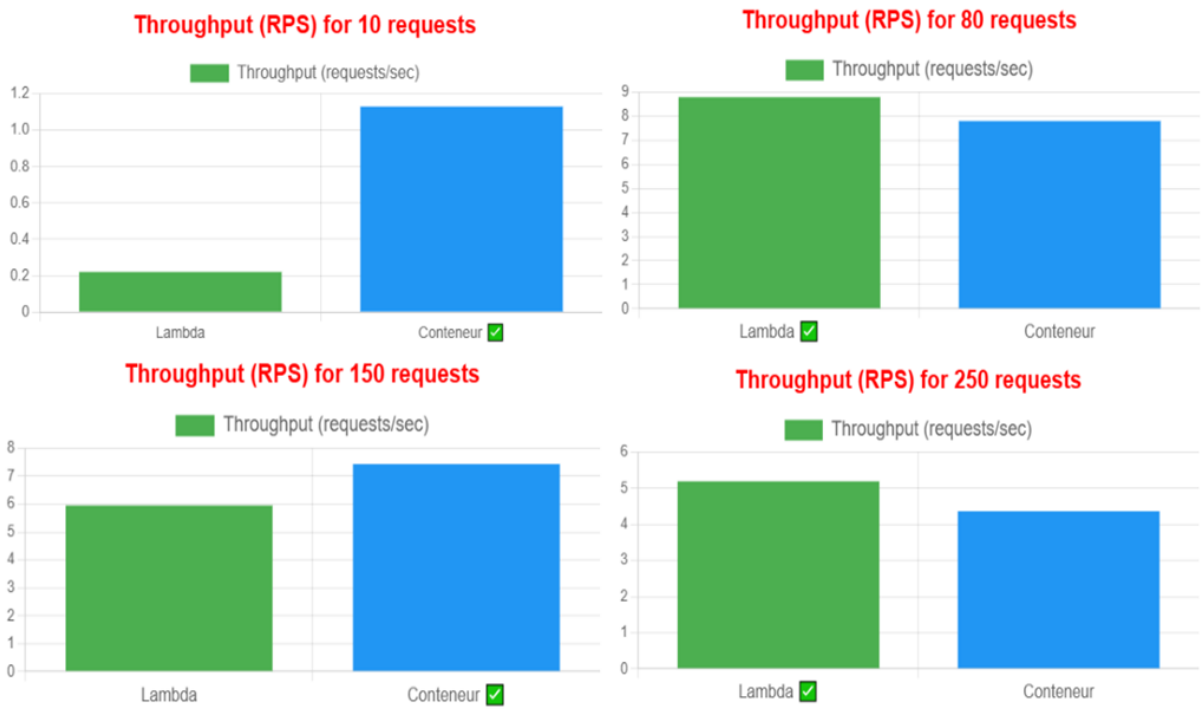


FIG. 5.20 : débit dans différentes implémentations dans le chat

4. Temps de réponse moyen (average time) : Bien que PHP dans un environnement conteneurisé (PaaS) présente un temps de réponse faible, il n'a pas pu traiter toutes les requêtes, ce qui indique des limites dans sa capacité à gérer une pression de requête élevée.

FIG. 5.21 : temps de réponse moyen pour différentes implémentations dans le chat

En revanche, Lambda (faas) a maintenu un taux de réponse élevé dans tous les cas, même avec le nombre croissant de demandes, démontrant une grande capacité d'évolution et de gestion de la pression. Cela rend le temps de réponse de PHP trompeur car il ne représente que les quelques requêtes qu'il a pu traiter, tandis qu'un grand nombre d'autres requêtes ont été rejetées ou ignorées. Nous pouvons conclure qu'AWS Lambda est plus fiable dans les scénarios de forte concurrence, même s'il a un coût en temps plus élevé.

Chapitre 6

Conclusion Générale

Cette étude visait à comparer deux approches d'hébergement d'applications web : une architecture FaaS basée sur le modèle sans serveur AWS Lambda, utilisée avec Netlify, et une solution PaaS en environnement conteneurisé, utilisée avec ECS sur AWS. Dans différents scénarios de test (chat (individuel ou privé), chat de groupe, génération d'images), nous avons mesuré les performances selon plusieurs indicateurs : temps de réponse moyen, débit et effet de démarrage à froid. Les résultats montrent que le système conteneurisé peut offrir un bon temps de réponse dans certains cas, mais qu'il devient rapidement limité à mesure que la charge augmente, avec un taux élevé de requêtes échouées. En revanche, l'architecture sans serveur Lambda, malgré de légers ralentissements occasionnels au démarrage, a pu traiter toutes les requêtes et maintenir d'excellentes performances sous forte charge. Nous avons également constaté que Lambda, grâce à sa fonctionnalité native de mise à l'échelle automatique, gère les performances du système de manière optimale, notamment en période de pointe. Cependant, il est important de prendre en compte le phénomène de démarrage à froid, qui peut entraîner des retards, notamment lors des premières requêtes. Pour remédier à ce problème, AWS propose un service de simultanéité fournie prêt à l'emploi. Cependant, cette option entraîne un coût supplémentaire et peut ne pas convenir à tous les budgets. En résumé, chaque approche présente ses avantages. Les conteneurs offrent une bonne réactivité à court terme, mais ralentissent sous charge. En revanche, les systèmes sans serveur sont flexibles, évolutifs et faciles à gérer. Ils semblent donc plus adaptés à long terme aux applications modernes, flexibles et évolutives.

Bibliographie

[1] Mell, P. et Grance, T. (2011). *The NIST Definition of Cloud Computing*. Consulté le 7/04/2019. Disponible sur : https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

[2] Nader MBAREK, *Chapitre 2: Gestion du niveau de service*, ISTE Group, Livre numérique, septembre 2020. Disponible sur : https://www.google.dz/books/edition/Chapitre_2_Gestion_du_niveau_de_service/JmVDEQAAQBAJ?hl=fr&gbpv=1&kptab=overview

[3] Oracle, *Définition de la Platform-as-a-Service (PaaS)*, Disponible sur : https://www.oracle.com/fr/cloud/definition-paas/

[4] IBM, *IaaS - Infrastructure-as-a-Service*, Disponible sur : https://www.ibm.com/think/topics/iaas

[5] CNCF Serverless Working Group (2018). *CNCF WG-Serverless Whitepaper v1.0*. Consulté le 8/04/2019. Disponible sur : https://github.com/cncf/wg-serverless/raw/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf

[6] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, *iFogSim : A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments*, Software : Practice and Experience, vol. 47, no. 9, pp. 1275–1296, 2017.

[7] IBM, *FaaS - Function-as-a-Service*, Disponible sur : https://www.ibm.com/think/topics/faas

[8] Lucidchart, *Diagramme de cas d'utilisation UML*, Disponible sur : https://www.lucidchart.com/pages/fr/diagramme-de-cas-dutilisation-uml

[9] UML Diagramme de Séquence, Disponible sur : https://lipn.univ-paris13.fr/~gerard/uml-s2/uml-cours05.html

[10] IBM, *Diagrammes de composants*, Disponible sur : https://www.ibm.com/docs/fr/dmrt/9.5.0?topic=diagrams-component

[11] IBM, *Diagrammes de déploiement*, Disponible sur : https://www.ibm.com/docs/fr/rsas/7.5.0?topic=topologies-deployment-diagrams

[12] Docker. Install Docker Desktop on Windows. Disponible sur : https://docs.docker.com/docker-for-windows/install/.

ملخص

- يتناول هذا التقرير مقارنة بين الحاويات والتطبيقات غير الخادمية في بيئة الحوسبة السحابية، حيث قمنا بتطوير تطبيق دردشة باستخدام طريقتين مختلفتين، الأولى تعتمد على FaaS وتم نشرها على منصة Netlify، والثانية باستخدام PaaS من خلال Amazon ECS Fargate. وقد أجرينا اختبارات أداء على كلتا الطريقتين من أجل تقييم الكفاءة والسرعة والتكلفة، وأظهرت النتائج أن النهج غير الخادمي أكثر فاعلية من حيث الأداء وأسهل من حيث الاستخدام وأقل من حيث التكاليف، رغم وجود بعض العيوب مثل البداية الباردة، إلا أنه مناسب أكثر للتطبيقات القابلة للتوسع بسهولة، في حين أن الحاويات، رغم مرونتها وقوتها، إلا أنها تتطلب إدارة أكبر وتكاليف أعلى للتوسع.

Abstract

- This report compares containers and serverless applications in a cloud computing environment. We developed a chat application using two different approaches: the first, FaaS, deployed on the Netlify platform, and the second, PaaS, deployed on Amazon ECS Fargate. We conducted performance tests on both approaches to evaluate efficiency, speed, and cost. The results showed that the serverless approach is more efficient in terms of performance, easier to use, and lower in cost. Despite some drawbacks, such as cold startup, it is better suited for easily scalable applications. Containers, while more flexible and powerful, require more management and higher scalability costs.

Résumé

- Ce rapport compare les conteneurs et les applications sans serveur dans un environnement de cloud computing. Nous avons développé une application de chat utilisant deux approches différentes : la première, FaaS, déployée sur la plateforme Netlify, et la seconde, PaaS, déployée sur Amazon ECS Fargate. Nous avons effectué des tests de performance sur les deux approches afin d'évaluer l'efficacité, la rapidité et le coût. Les résultats ont montré que l'approche sans serveur est plus performante, plus simple d'utilisation et moins coûteuse. Malgré quelques inconvénients, comme le démarrage à froid, elle est mieux adaptée aux applications facilement évolutives. Les conteneurs, bien que plus flexibles et puissants, nécessitent une gestion plus importante et des coûts d'évolutivité plus élevés.