


République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université AMO de Bouira

Faculté des Sciences et des Sciences Appliquées

Département d'Informatique

Mémoire de Master

en Informatique

Spécialité : *GSI*

Thème

Étude des Benchmarks pour les Microservices

Encadré par

— MAHFOUD Zohra

Réalisé par

— KHEDDOUCI Maya

— SIFOUANE Katia

2023/2024

People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University of AKLI MOHAND OULHAJ –Bouira-
Faculty of Science and Applied Sciences
Computer science department

Master thesis

Computer science

speciality : *GSI*

Theme

A comparative study of microservices bechmarks

Supervised by

— MAHFOUD Zohra

Submitted by

— KHEDDOUCI Maya

— SIFOUANE Katia

2023/2024

Remerciements

C'est avec un grand plaisir qu'on réserve ces quelques lignes en signe de gratitude et de profonde reconnaissance à tous ceux qui, de près ou de loin, ont contribué à la réalisation et l'aboutissement de ce travail.

Nous tenons tout d'abord à remercier le bon Dieu le tout puissant, qui nous a donné la force, la santé, la volonté et le courage d'accomplir ce modeste travail.

En second lieu, nous tenons à remercier, notre encadrante Mme MAHFOUD zohra, pour sa disponibilité, sa patience, et ses conseils.

Nous tenons à exprimer notre reconnaissance envers les amis et les collègues qui nous ont apporté leur soutien moral et intellectuel tout au long de notre démarche.

Nous remercions également toute l'équipe pédagogique du département d'informatique de l'université de Bouira.

Nous adressons nos vifs remerciements aux membres de jury, pour l'honneur qu'ils nous ont fait d'évaluer notre mémoire de fin d'étude.

Dédicaces

Je dédie ce modeste travail

À mes chers parents, qui m'encouragent dans les instants délicats et qui me soutienne tout au long de mon parcours scolaire.

À mes chers frères : Hani et Lyes

À mon cousin Rinas

À toute la famille Kheddouci et Aoudjit

Et à ma copine et mon binôme Katia

Maya.

Dédicaces

Je dédie ce travail

À mes parents,qui ont été à mes côtés depuis toujours

À mes frères Kassi et Fayez

À ma sœur Tina

À chaque personne qui m'a soutenu et encouragé, même par un simple mot gentil

À ma copine Maya

Katia.

Résumé

Les architectures de microservices, reconnues pour leur flexibilité et leur scalabilité, suscitent des interrogations quant à leurs performances effectives. Ce mémoire examine la manière dont les microservices réagissent sous diverses charges de travail, en s'appuyant sur le benchmark DeathStarBench et l'application SocialNetwork. L'étude évalue les performances en termes de temps de réponse, utilisation du CPU et de la RAM. Les performances, variant selon les scénarios de charge. Les résultats des tests montrent que l'architecture des microservices permet une amélioration des performances et une meilleure gestion des ressources , particulièrement sous des charges de travail élevées.

Mots clés : Microservices, Benchmark, Performance, Scalabilité, DeathStarBenc

Abstract

Architectures based on microservices are known for their flexibility and scalability, but they raise questions about their actual performance. This research examines how microservices perform under various workloads, using the well-known benchmark DeathStarBench and the SocialNetwork application. The study evaluates performance in terms of response time, CPU and RAM usage. Performance, which varies according to load scenarios. The test results show that the microservices architecture allows for improved performance and better resource management, especially under high workloads.

Key words : Microservices, Benchmark, Performance, Scalability, DeathStarBench.

ملخص

تُعرف الهندسات المعتمدة على الخدمات المصغرة بمرونتها وقابليتها للتوسع، لكنها تثير تساؤلات حول أدائها الفعلي .يفحص هذا البحث كيف تتفاعل الخدمات المصغرة تحت أحمال عمل متنوعة، مستعينًا بالمعيار المعروف بـ DeathStarBench وتطبيقSocialNetwork. تقيم الدراسة الأداء من حيث وقت الاستجابة، واستخدام المعالج والذاكرة العشوائية .يتفاوت الأداء وفقًا لسيناريوهات الحمل .تُظهر نتائج الاختبارات أن بنية الخدمات المصغرة تُمكّن من تحسين الأداء وإدارة أفضل للموارد، خاصةً تحت أحمال العمل المرتفعة.

Table des matières

Table des figures

Liste des tableaux

Liste des abréviations

API	Application Programming Interface
MSA	Microservice Architecture
NoSQL	Not only Structured Query Language
SQL	Structured Query Language
SOA	Architecture Orientée Services
IOT	Internet of Things
HTTP	Hypertext Transfer Protocol
RPC	Remote Procedure Cal
IPC	Inter-Process Communication
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
J2EE	Java 2 Platform, Enterprise Edition
IA	Intelligence Artificielle
CI/CD	Continuous Integration/Continuous Delivery

Introduction générale

Dans un monde où les technologies évoluent rapidement, les architectures de microservices se sont imposées comme un modèle incontournable pour le développement de logiciels modernes. Les microservices offrent une flexibilité et une scalabilité accrues par rapport aux architectures monolithiques traditionnelles, permettant aux entreprises de répondre plus efficacement aux exigences croissantes des utilisateurs et des marchés. Cependant, cette transition vers les microservices pose également de nouveaux défis en matière de performance, de gestion des ressources et de maintenance.

Dans ce contexte, ce projet de fin d'études présente une étude comparative des benchmarks des microservices en visant à analyser les performances des microservices dans divers scénarios de charge et d'utilisation. En utilisant le benchmark DeathStarBench, et plus spécifiquement l'application Social Network, cette recherche explore les performances des microservices à travers une série de tests individuels et parallèles, évaluant des métriques clés telles que le temps de réponse et l'utilisation du CPU et de la RAM.

Dans le premier chapitre du mémoire, nous offrons une introduction aux microservices. Nous présentons l'évolution des architectures logicielles, en passant par les architectures monolithiques et orientées services (SOA), pour arriver à l'architecture de microservices. Nous détaillons ensuite les avantages des microservices, notamment leur flexibilité et leur scalabilité, ainsi que les défis associés à leur développement et leur maintenance. Enfin, nous illustrons ces concepts à travers des cas d'usage concrets et des exemples pratiques.

Nous consacrons le deuxième chapitre aux benchmarks des microservices. Nous commençons le chapitre par une vue d'ensemble des principaux benchmarks existants, tels que

µBench, DeathStarBench, TeaStore, Online Boutique, Bookinfo, Petclinic, Sock Shop et JPetStore. Chaque benchmark est décrit en détail, mettant en lumière ses caractéristiques et ses applications spécifiques. Nous procédons ensuite à une analyse comparative des différentes applications de benchmarking, évaluant leur performance selon des critères précis. Ce chapitre se termine par une discussion sur les critères de performance importants pour les microservices et les outils et méthodes couramment utilisés pour leur benchmarking.

Dans le troisième et dernier chapitre, nous présentons notre méthodologie de travail, les tests effectués, ainsi que les résultats obtenus. Nous décrivons l'environnement de travail, y compris l'environnement matériel utilisé. Ce chapitre détaille notre méthodologie pour mesurer le temps de réponse moyen et l'utilisation des ressources système. Nous y présentons trois séries d'expériences : tests individuels, tests en parallèle, et comparaison des charges. Chaque série de tests est accompagnée d'une analyse des résultats obtenus, mettant en lumière les performances des microservices dans différents scénarios. Nous concluons ce chapitre par une discussion sur les implications de ces résultats et des recommandations pour l'optimisation des architectures de microservices.

Nous clôturons ce mémoire par une conclusion générale qui récapitule nos découvertes, reflétant l'importance et l'impact des architectures de microservices dans le développement de logiciels modernes. Cette synthèse abordera également les défis rencontrés et les perspectives futures.

Chapitre 1

Les microservices

1.1 Introduction

Dans ce chapitre, nous allons explorer le monde des microservices. Nous discuterons de l'évolution des architectures logicielles, en commençant par l'architecture monolithique, puis en passant à l'architecture orientée services (SOA), et enfin en arrivant à l'architecture de microservices. Ensuite, nous discuterons des avantages et des défis associés aux microservices. Après avoir examiné le développement et l'utilisation des microservices, nous terminerons avec des exemples pratiques pour illustrer leur utilisation dans le monde réel.

1.2 Évolution des architectures logicielles

1.2.1 Architecture monolithique

Une architecture monolithique est un modèle de conception de logiciel où toutes les fonctionnalités d'une application sont regroupées en un seul exécutable. Cette approche implique que l'application, bien que pouvant être constituée de plusieurs modules, doit être déployée et exécutée comme une unité indivisible. En conséquence, les modules ne peuvent pas fonctionner indépendamment les uns des autres.[13]

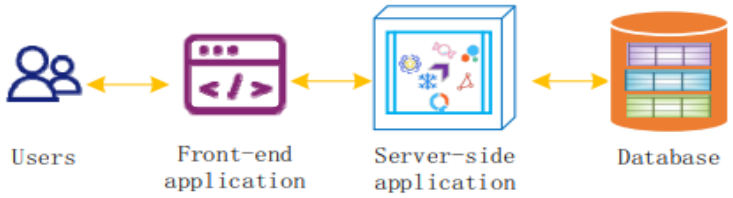

FIGURE 1.1 – Architecture monolithique [1]

Comme le montre la figure 1.1, une application monolithique se compose typiquement d'une application côté utilisateur (front-end), d'une application côté serveur (back-end) et d'une base de données. Toutes ces composantes sont intégrées et déployées ensemble. Ce modèle centralisé, bien qu'efficace pour des applications de petite à moyenne taille, devient de plus en plus difficile à maintenir et à faire évoluer au fur et à mesure que l'application grandit en complexité et en fonctionnalité. De plus, l'intégration de nouvelles technologies dans une architecture monolithique rigide et standardisée est souvent contraignante et pose des défis significatifs.

1.2.2 Architecture orientée services (SOA)

Au cours de la décennie 1990, le SOA a émergé comme une innovation majeure visant à découpler les applications du côté du service et à favoriser la réutilisation des composants. Comme illustré dans la figure 1.2, l'architecture SOA peut être décomposée en plusieurs fonctions de serveur d'application, chacune orientée vers des services faiblement couplés. Chaque service peut être hébergé dans des conteneurs distincts, et ils interagissent entre eux via un bus de services d'entreprise, tout en ayant accès à une base de données commune.[1]

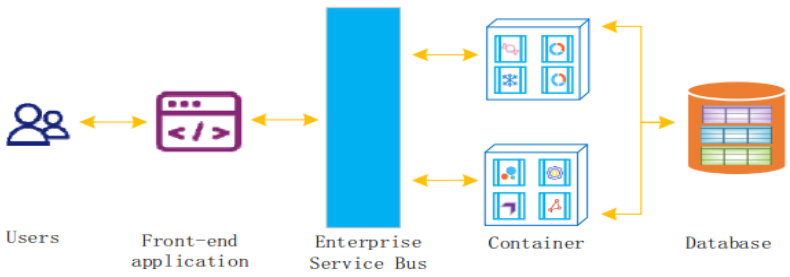

FIGURE 1.2 – Architecture orientée services [1]

1.2.3 Architecture de microservices

Les microservices, qui sont une évolution des principes de l'architecture orientée services (SOA), organisent une application en un ensemble compact de services logiciels indépendants. Pour augmenter l'indépendance des applications, l'architecture des microservices suggère de décomposer les applications en plusieurs services distincts, chacun axé sur une fonction métier spécifique. Comme le montre la figure 1.3, l'application serveur est décomposée en de nombreux microservices de petite taille, chaque service assumant une fonction métier particulière et étant conçu pour fonctionner dans des conteneurs distincts. Chaque conteneur a sa propre base de données privée, qui n'est pas directement accessible par les autres conteneurs.[1]

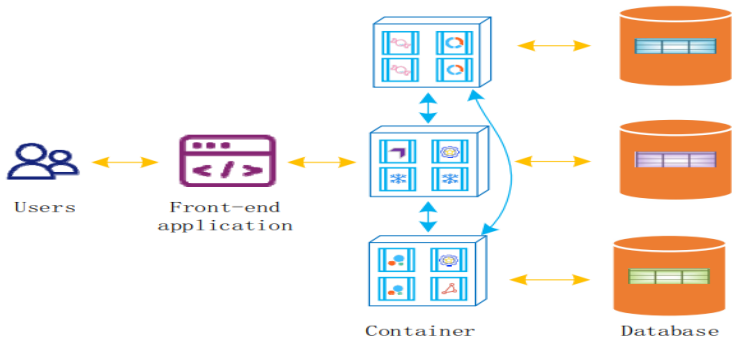

FIGURE 1.3 – Architecture de microservices [1]

En outre, les microservices utilisent deux stratégies principales pour gérer les interactions entre les services : l'orchestration et la chorégraphie.

1.2.3.1 Architecture d'orchestration

Dans l'orchestration des microservices, un composant central, souvent appelé l'orchestrateur, agit comme un chef d'orchestre qui dirige chaque service comme un musicien dans un orchestre. Chaque microservice exécute une tâche spécifique attribuée par l'orchestrateur, qui coordonne également la collecte des résultats et la gestion des états de service. Cela permet d'exécuter des transactions logicielles ou des requêtes de manière efficace, en s'assurant que chaque partie du système travaille en harmonie.

La figure 1.1 détaille l'architecture de microservices orchestrée :

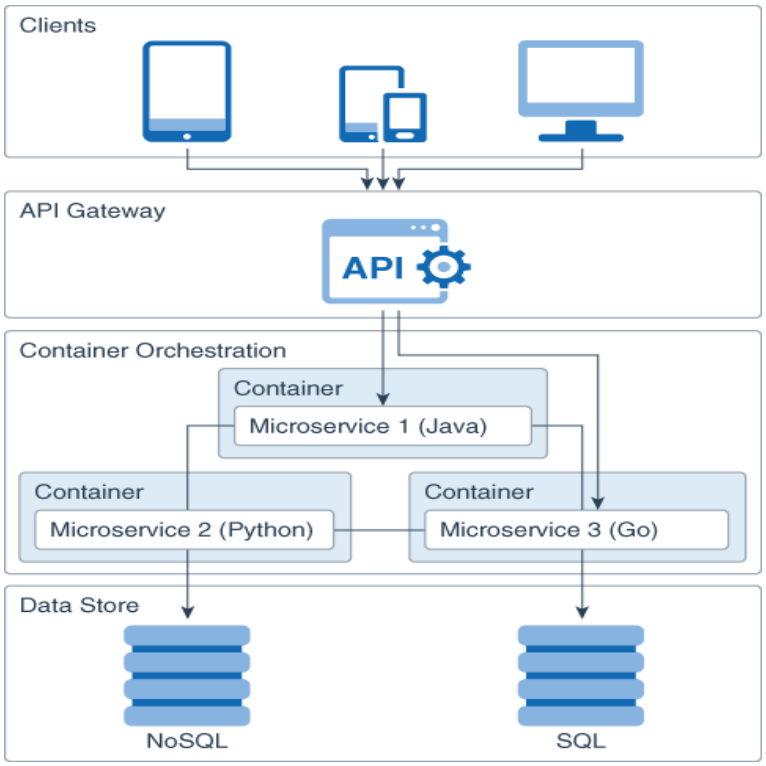

FIGURE 1.4 – L'architecture de microservices orchestréé [2]

1 **La couche client**

Cette couche représente les points d'interaction entre l'utilisateur et le système. Elle inclut divers types de dispositifs tels que les ordinateurs de bureau, les ordinateurs portables, et les appareils mobiles.[2]

2 **La couche de passerelle API**

Dans une architecture microservices AMS, la décomposition des applications en services indépendants soulève des défis tels que la complexité de la communication entre Services et l'inefficacité dans la gestion des fonctionnalités communes. Pour résoudre ce problème, l'utilisation d'une API Gateway est proposée comme solution.[14]

L'API Gateway agit comme un intermédiaire entre les clients et les microservices, offrant un point d'entrée unique pour toutes les demandes externes.[14]

De plus, l'API Gateway prend en charge les fonctionnalités transversales (comme l'authentification, l'équilibrage de charge, etc.), ce qui allège la charge sur les microservices individuels et assure une gestion centralisée et plus efficace de ces aspects critiques. [14]

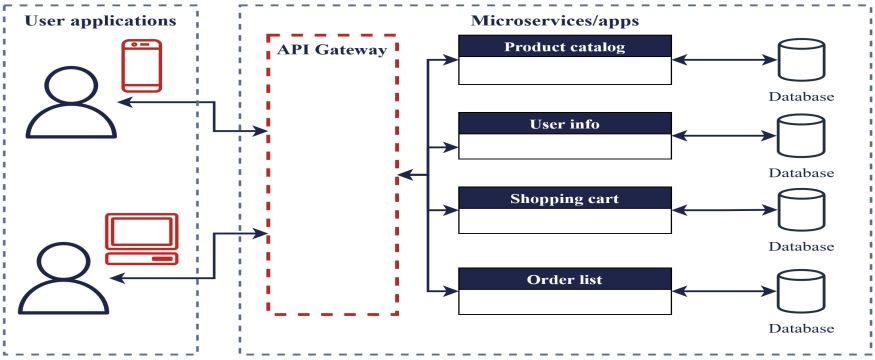

FIGURE 1.5 – API Gateway [3]

En complément, il est important de souligner que l'API Gateway facilite une communication efficace entre différentes applications ou services, en interne comme en externe. Elle traite les requêtes entrantes, les dirige vers le bon service, puis renvoie les réponses appropriées. Cette approche centralisée permet d'éviter la complexité et les risques liés à des appels directs entre services. [3]

3 **La couche d'orchestration de conteneurs**

Un système qui encapsule les microservices dans des conteneurs distincts. Cette approche facilite la communication entre les microservices et le monde extérieur via la passerelle API, tout en maintenant une organisation et une gestion efficaces des services.[2]

Chaque service fonctionne dans son propre conteneur, permettant l'exécuter plusieurs instances de microservices, chaque instance étant hébergée dans son propre conteneur. Les conteneurs, de par leur nature légère et éphémère, rendent plus aisée l'ajout ou le retrait de services selon les besoins, tout en gardant chaque service bien séparé des autres.[15]

Il existe plusieurs outils d'orchestration de conteneurs sur le marché, parmi lesquels Kubernetes, Docker Swarm, et Apache Mesos se distinguent. Kubernetes est notamment l'outile open source le plus populaire pour orchestrer des conteneurs, conçue initialement par des ingénieurs chez Google.[16]

4 **Entrepôt de données**

La composante finale de l'architecture est dédiée à la gestion des données. Chaque microservice est associé à sa propre base de données, optant soit pour un système NoSQL afin de bénéficier d'une plus grande flexibilité, soit pour un système SQL

qui offre une structure plus rigide. Cette approche individualisée permet à chaque service de gérer ses données de manière indépendante, facilitant ainsi la maintenance et l'évolution de l'ensemble du système.[2]

1.2.3.2 Architecture de chorégraphie

La chorégraphie des microservices se distingue de l'orchestration par son mécanisme décentralisé. En effet, contrairement à l'approche centralisée de l'orchestration qui implique un orchestrateur, la chorégraphie n'en nécessite pas. Dans ce modèle, les microservices opèrent de manière asynchrone, réagissant aux événements déclencheurs sans qu'un agent central coordonne leurs actions.

Cette approche asynchrone permet aux microservices de collaborer pour réaliser des tâches en réponse à des événements spécifiques. Les demandes des clients sont placées dans une file d'attente d'événements (ou file de messages), et seuls les microservices concernés reçoivent et traitent ces messages. Une fois la tâche accomplie, les microservices renvoient un feedback de succès ou d'échec. Ce mode de communication permet une coordination efficace entre les microservices sans recourir à un orchestrateur.

La chorégraphie offre aux microservices une autonomie opérationnelle, réduisant ainsi les problèmes liés aux dépendances. Cela facilite la parallélisation des tâches et des événements, améliorant ainsi les performances globales du système.[17]

1.2.3.3 Choix entre chorégraphie et orchestration dans les architectures de microservices

Pour aborder la question de l'utilisation conjointe des deux architectures, on pourrait commencer ainsi,la chorégraphie des événements offre rapidité et efficacité dans les architectures de microservices simples, mais elle peut devenir complexe et difficile à gérer avec l'augmentation du nombre de services et d'interactions. L'orchestration, bien que plus lente, fournit un contrôle centralisé essentiel dans les transactions complexes, réduisant les risques d'erreurs.

Le choix entre ces deux approches dépend donc des spécificités du projet, notamment la complexité des interactions et les exigences de performance. Pour des systèmes simples et rapides, la chorégraphie est avantageuse, tandis que pour des contextes exigeant une coordination minutieuse, l'orchestration est préférable.[18]

1.3 Les avantages des microservices

Pour répondre aux besoins croissants de flexibilité et de scalabilité, les microservices se révèlent être une solution efficiente. Cette architecture divise les applications en services autonomes, offrant ainsi divers avantages que nous examinerons dans ce chapitre [19] [20] :

A. **Scalabilité**

Les microservices permettent de déployer et de dimensionner chaque service indépendamment,ce qui accélère la scalabilité des applications complexes.

B. **Isolation des fautes**

En cas de défaillance d'un service, elle ne se propage pas à l'ensemble du système, ce qui améliore la résilience éde l'application.

C. **Productivité des équipes**

Les équipes peuvent se concentrer sur des services spécifiques, ce qui simplifie les cycles de développement et permet une spécialisation plus poussée.

D. **Déploiement rapide**

Chaque microservice peut être développé, testé et déployé indépendamment, ce qui réduit le risque et les édélais associés aux mises à jour de l'application.

E. **Efficacité des coûts**

La gestion ciblée des ressources et la maintenance lo-calisée des services réduisent les coûts globaux de édéveloppement et de système.

F. **Diversité technologique**

Les microservices permettent d'utiliser différents langages et technologies adaptés à chaque service, optimisant ainsi l'efficacité et la productivité du édéveloppement.

1.4 Les défis des microservices

Les microservices,présentent également des défis significatifs. Comprendre ces défis est essentiel pour une mise en œuvre efficace et une gestion réussie de ces architectures distribuées[19] [20] [21] :

A. **Complexité accrue**

La gestion de la communication entre les services distribués peut s'avérer compliquée, nécessitant souvent un codage supplémentaire pour assurer une interaction fluide entre les modules.

B. **Gestion des déploiements et des version**

Coordonner les déploiements et gérer le contrôle des versions à travers de multiples services peut entraîner des problèmes de compatibilité et compliquer l'administration.

C. **Surcharge du réseau**

Comme les microservices fonctionnent de manière indépendante, ils peuvent générer un trafic réseau élevé, ce qui requiert des techniques robustes pour maintenir la performance et la disponibilité.

D. **Développement dispersé**

Le passage d'une architecture monolithique à des microservices augmente la complexité organisationnelle, rendant difficile la visualisation des relations entre les différents composants et la gestion des dépendances.

1.5 Développement et Utilisation des Microservices

1.5.1 Développement des microservices

Les phases de développement de chaque microservice au sein d'une architecture basée sur les microservices. [4] :

A. **Création de microservices**

Chaque microservice est conçu et développé de manière autonome par une équipe distincte, en utilisant un langage de programmation et une base de données spécifiques.

B. **Intégration régulière**

Le code du microservice en cours de développement est synchronisé et fusionné quotidiennement par chaque équipe.

C. **Tests automatisés**

Suite à l'intégration du code, une série de tests (unitaires, fonctionnalités, charge, etc) est exécutée automatiquement pour assurer la qualité du code développé.

D. **Identification et découverte**

Ce processus permet aux consommateurs de microservices de localiser et d'établir une communication avec ces derniers.

E. **Déploiement continu**

Si les tests sont concluants, le code est automatiquement déployé sur le serveur de test ou de production.

F. **Intégration et composition**

Une fois les microservices répondant aux besoins de l'application identifiés, nous procédons à leur intégration et à la composition de l'application d'entreprise. Dans certains cas, il peut être nécessaire de modifier certains microservices pour répondre aux exigences de l'application, ou même d'en créer un nouveau à partir de zéro.

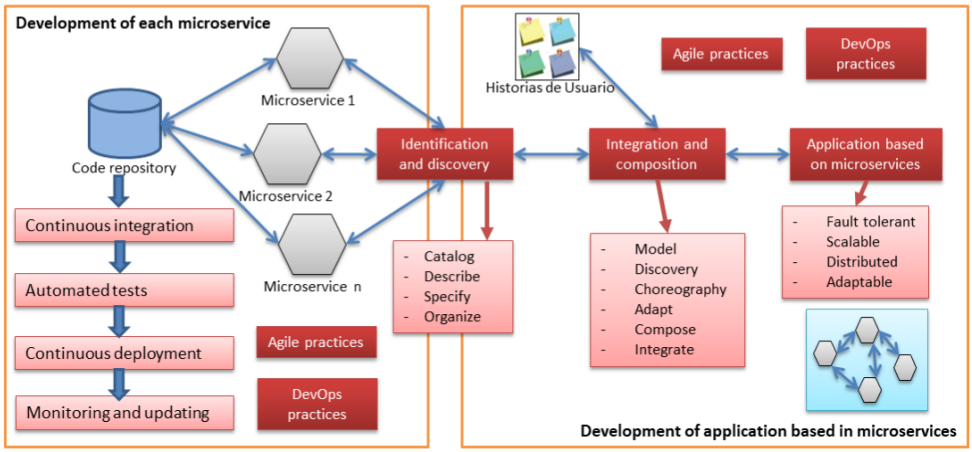

FIGURE 1.6 – Principales caractéristiques du processus de développement des microservices [4]

G. **Surveillance et mise à jour**

Chaque microservice est surveillé de manière indépendante, permettant d'identifier les échecs ou les besoins de mise à l'échelle ou d'augmentation des ressources informatiques. Les mises à jour sont effectuées de manière indépendante, permettant de modifier un microservice sans perturber ses consommateurs.

DevOps est un paradigme en pleine expansion qui vise à intégrer l'équipe de développement avec le personnel des opérations, d'où son acronyme Dev (Développeurs) - Ops (Opérations). Leurs pratiques permettent des mises à jour rapides et fréquentes. DevOps est un terme récent qui émerge avec la convergence de deux nouvelles tendances, l'infrastructure agile ou les opérations agiles, et la collaboration entre le personnel de développement et d'exploitation à toutes les étapes du cycle de vie du développement, du codage au déploiement en passant par la production. [4]

1.5.2 Déploiement et maintenance des microservices dans des environnements de production

Dans le cadre de la gestion des microservices, il est essentiel de prendre en compte non seulement leur création et leur développement, mais aussi leur maintenance. Chaque microservice a des exigences spécifiques en raison de la diversité de ses ressources et de sa technologie. Plusieurs options de déploiement sont disponibles pour répondre à ces besoins. [22]

A. **Modèle d'hôte à services multiples**

Ce modèle utilise plusieurs hôtes, physiques ou virtuels, pour déployer les microservices. Plusieurs instances de service partagent le même serveur et le même système d'exploitation. Ce modèle permet une utilisation efficace des ressources et peut améliorer l'évolutivité et les performances. Cependant, il peut être difficile de surveiller l'utilisation des ressources par chaque instance de service.

B. **Modèle d'hôte à service unique**

Dans ce modèle, chaque instance de service est déployée sur un hôte spécifique et ne partage aucune ressource. L'hôte peut être une machine virtuelle ou un conteneur. Les technologies de conteneurisation comme Docker peuvent être utilisées pour le déploiement. Ce modèle facilite le suivi des ressources, mais peut affecter l'utilisation des ressources, les performances et l'évolutivité.

C. **Déploiement sans serveur**

C'est une autre méthode de déploiement de microservices qui peut être adaptée à certaines applications.

1.5.3 L'impact des mécanismes de communication et de la surveillance sur le déploiement des microservices

Les mécanismes de communication et la surveillance jouent un rôle crucial dans le déploiement efficace des microservices.[22]

A. **Mécanismes de communication**

Les microservices, étant un ensemble de petits services, nécessitent un canal de communication sophistiqué pour la communication entre services ou entre processus.
Les mécanismes de communication peuvent être classés en deux catégories : synchrone et asynchrone.

* **Communication synchrone**

  Ce type de communication, qui fonctionne sur le principe de la demande et de la réponse, est essentiel pour une communication fluide entre les processus. Cependant, il peut introduire des dépendances temporelles entre les services, ce qui peut affecter le déploiement et l'évolutivité des microservices.

* **Communication asynchrone**

  Dans ce type de communication, le client ne attend pas de réponse du service. Cela permet une meilleure isolation entre les services et améliore l'évolutivité, mais peut rendre le suivi des interactions plus complexe.

B. **Surveillance**

La surveillance et le suivi sont essentiels pour assurer le bon fonctionnement des microservices dans un environnement de production. Ils permettent d'assurer la stabilité du système, d'optimiser l'utilisation des ressources et de maintenir un haut niveau de performance et de sécurité.

* **Surveillance des performances**

  Le suivi des performances des microservices peut aider à identifier les goulots d'étranglement et à optimiser le déploiement des services. Cela peut également aider à planifier l'évolutivité et à garantir que les services répondent aux exigences de performance.

* **Surveillance de l'état du système**

  Le suivi de l'état du système peut aider à détecter les problèmes avant qu'ils n'affectent les utilisateurs, ce qui peut faciliter le déploiement et la maintenance des microservices.

Les mécanismes de communication et la surveillance ont un impact significatif sur le déploiement des microservices. Ils influencent la manière dont les services sont déployés et gérés, et peuvent aider à optimiser l'utilisation des ressources, les performances et la sécurité. Ils sont donc des éléments clés à prendre en compte lors de la conception et du déploiement de microservices.

1.6 Cas d'usage et exemples

Le tableau ci-dessous présente une vue d'ensemble des cas d'usage des microservices, en expliquant comment ils peuvent être appliqués pour résoudre divers problèmes et défis dans le développement de logiciels. Chaque cas d'usage est illustré par un exemple concret d'une entreprise qui a réussi à mettre en œuvre l'architecture de microservices pour répondre à ces défis spécifiques. Ces exemples illustrent la flexibilité et la puissance des microservices, et comment ils peuvent être utilisés pour construire des systèmes robustes et évolutifs.[23]

Cas d'usage	**Description**	**Exemples**
Plateformes de commerce électronique	Les microservices peuvent gérer diverses fonctionnalités telles que le catalogue de produits, le panier d'achat, la gestion des commandes, le traitement des paiements, les avis des utilisateurs et les moteurs de recommandation.	Amazon, eBay
Services de streaming multimédia	Les microservices permettent une diffusion efficace du contenu, des recommandations personnalisées, la gestion des utilisateurs et la facturation.	Netflix, Hulu, Spotify

Applications de voyage et d'hôtellerie	Les microservices peuvent gérer des fonctions telles que la gestion des réservations, la recherche et le filtrage, les profils d'utilisateurs, le traitement des paiements et les avis.	Booking.com, Airbnb
Services financiers et bancaires	Les microservices peuvent gérer des fonctionnalités telles que la gestion des comptes, le traitement des transactions, la détection des fraudes, l'évaluation des risques et la conformité.	PayPal, Stripe
Internet des objets (IoT)	Les microservices peuvent fournir des services pour l'ingestion de données, le traitement en temps réel, l'analyse, la gestion des appareils et l'intégration avec des systèmes externes.	Smart Home Systems, Industrial IoT
Plateformes de médias sociaux	Les microservices peuvent gérer diverses fonctionnalités telles que les profils d'utilisateurs, les publications, les chronologies, les notifications, la messagerie et la modération du contenu.	Facebook, Twitter
Jeux et divertissement	Les microservices peuvent gérer le matchmaking, la logique de jeu, la gestion des joueurs, les classements, la monnaie virtuelle et la distribution de contenu.	Fortnite, PUBG
Soins de santé et télémédecine	Les microservices peuvent gérer des fonctionnalités telles que la gestion des patients, la planification des rendez-vous, l'analyse des données, la communication sécurisée et l'intégration avec des dispositifs médicaux.	Doctolib, Teladoc

TABLE 1.1 – Cas d'usage et exemples

1.7 Conclusion

Après avoir parcouru ce chapitre, nous avons une meilleure compréhension des microservices. Nous avons vu comment ils sont structurés, comment ils sont utilisés et les défis qu'ils présentent. Les exemples pratiques ont permis de mettre en lumière leur utilisation dans différents contextes. Ainsi, nous avons pu constater que malgré certains défis, les microservices offrent une flexibilité et une puissance qui peuvent être très bénéfiques dans le développement et le déploiement d'applications.

Chapitre 2

Benchmarks des Microservices

2.1 Intoduction

Dans le vaste domaine des technologies de l'information, les benchmarks sont des outils standardisés utilisés pour évaluer et comparer la performance de divers systèmes et composants. Ils permettent de quantifier de manière objective la capacité d'un système à exécuter des tâches spécifiques, facilitant ainsi les choix technologiques et l'amélioration des performances. Plusieurs benchmarks sont proposés dans le domaine des microservices. Ces derniers, en raison de leur nature modulaire et distribuée, nécessitent des méthodes d'évaluation spécifiques pour mesurer avec précision leur performance et leur fiabilité.

Ce chapitre se concentrera sur ces outils dédiés, explorant les critères de performance essentiels et les meilleures pratiques en matière de benchmarking spécifique aux microservices.

2.2 Benchmarks

Un benchmark est une série de tests conçus pour évaluer les performances d'un système, d'un composant, ou d'une application[24]

2.3 Vue d'Ensemble des Benchmarks

2.3.1 *µBench*

µBench se présente comme un outil open source innovant destiné à la génération et à l'exécution d'applications de microservices de référence, offrant ainsi une plateforme essentielle pour la recherche dans ce domaine. En utilisant *µBench*, les chercheurs disposent d'un contrôle précis sur des aspects critiques de l'application produite. Cela inclut la détermination du nombre de microservices à intégrer, ainsi que la spécification des besoins en ressources - qu'il s'agisse du CPU, du stockage ou du réseau. [5]

µBench permet également de configurer le réseau de dépendances entre les microservices, une caractéristique fondamentale connue sous le nom de maillage de services, tout en offrant la possibilité de choisir entre HTTP et gRPC pour la communication interservices.[5]

Une fois l'application de microservice conceptualisée, *µBench* prend en charge son déploiement sur un cluster Kubernetes, facilitant ainsi l'exportation de métriques clés par microservice via Prometheus. Ces données, incluant la latence et le débit, sont cruciales pour l'évaluation des performances. Les benchmarks réalisés avec *µBench* peuvent englober divers scénarios, faisant intervenir un nombre défini ou aléatoire de microservices par requête, basés soit sur des configurations prédéterminées, soit sur des traces réelles.[5]

Les microservices de *µBench* sont conçus en Python à l'origine. Toutefois, ils ne supportent pas les applications multilingues dans leur configuration de base. C'est là que les choses deviennent intéressantes : un microservice µBench peut être couplé à un conteneur side-car qui est sollicité à chaque requête. Ce side-car a la capacité d'exécuter des programmes définis par l'utilisateur dans n'importe quelle langue, ou même d'être une application réelle (comme MongoDB, par exemple). Alors que la plupart des autres applications de référence sont multilingues par nature, *µBench* offre cette flexibilité.[5]

En plus de leur flexibilité linguistique, les microservices *µBench* sont autonomes dans l'exportation de leurs mesures de performances. Chaque microservice envoie indépendamment ses métriques vers un serveur Prometheus. De plus, pour le traçage, ils peuvent être intégrés aux capacités de traçage fournies par Istio et Jaeger.[5]

2.2.1.1 L'architecture de *µBench*

L'architecture de *µBench* est structurée autour de la création, de la configuration et du déploiement d'applications basées sur des microservices, visant à faciliter la recherche et l'enseignement des systèmes distribués. Voici les éléments clés de cette architecture, divisée en plusieurs composants principaux [5] :

1. **Modèle d'Application**

   Microservices et API Gateway : Les applications générées par *µBench* sont composées de microservices, dont le nombre et le type peuvent être configurés. Ces microservices sont accessibles via une API Gateway, qui dirige les requêtes HTTP entrantes vers les microservices internes concernés.

   Modèle de Travail des Microservices : Chaque microservice suit un modèle de travail synchrone simple en trois phases : une phase de service interne axée sur l'utilisation des ressources locales (CPU, disque, mémoire), une phase de services externes pour l'appel et l'attente des résultats des microservices en aval, et une phase de réponse.

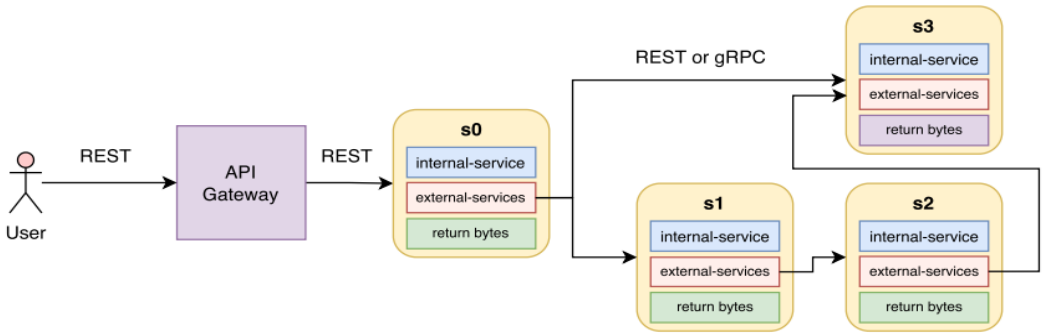

FIGURE 2.1 – Une application de microservice *µBench* [5]

2. **Chaîne d'Outils Logiciels**

   Service Mesh Generator : Crée la topologie du maillage de services et définit une stratégie pour les appels de services (parallèles ou séquentiels), en se basant sur un fichier de paramètres pour produire un fichier servicemesh.json.

   Générateur de Modèle de Travail : Définit le modèle de travail de chaque microservice, y compris les détails des services internes et externes et la quantité de données à renvoyer. Il génère un fichier workmodel.json à partir d'un fichier de paramètres et du fichier servicemesh.json.

K8s Deployer : Déploie les applications de microservices sur un cluster Kubernetes en créant les ressources nécessaires, utilisant les fichiers de paramètres et workmodel.json.

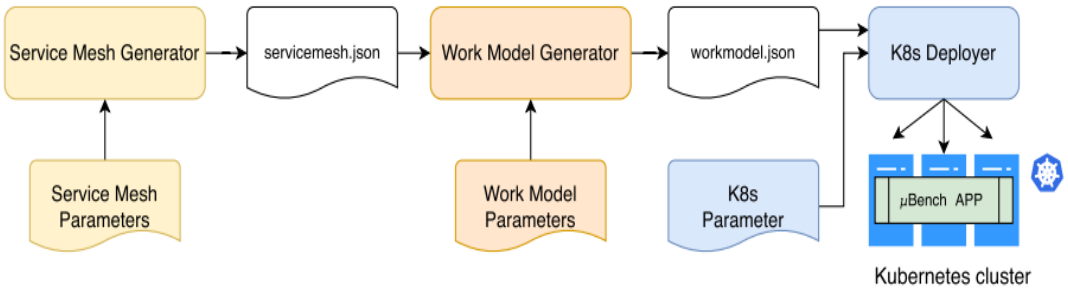

FIGURE 2.2 – Chaîne d'outils *µBench* [5]

3. **Maillage de Services**

   Configuration : *µBench* permet de définir comment les microservices interagissent au sein de l'application, en utilisant soit un modèle stochastique pour des appels de services aléatoires ou basés sur des probabilités, soit un modèle basé sur des traces pour des séquences spécifiques de services.

   Générateur de Maillage de Services : Pour les configurations complexes, un outil génère des topologies aléatoires ou spécifiques en utilisant des algorithmes comme Barabási-Albert, permettant la création de différentes structures de maillage (centralisées ou décentralisées).

4. **Modèle de Travail**

   Définit précisément le rôle et le comportement de chaque microservice au sein de l'application, incluant la nature du service interne, les appels aux services externes (pour les configurations stochastiques), la quantité de données retournées, et d'autres paramètres systèmes.

2.2.1.2 Domaines d'application et utilité

Pour illustrer concrètement l'apport de *µBench* dans le domaine des applications microservices, examinons des exemples significatifs [5] :

-Dans une étude, *µBench* a comparé les avantages et les inconvénients des architectures microservices par rapport aux architectures monolithiques.

-Ils ont analysé l'impact des choix architecturaux clés, tels que la topologie du service mesh et aussi Évolutivité horizontale (l'utilisation de la réplication).

2.2.1.3 Les avantages de *µBench*

Utiliser *µBench* offre aux chercheurs une suite d'avantages significatifs pour la modélisation et l'analyse d'applications microservices, grâce à sa capacité à [5] :

Contrôler Finement l'Architecture d'Application : Les utilisateurs peuvent spécifier avec précision le nombre de microservices dans une application, permettant des simulations de systèmes allant de simples à extrêmement complexes.

Adapter les Besoins en Ressources : La possibilité de déterminer la consommation de ressources (CPU, disque, réseau) pour chaque microservice permet de créer des scénarios réalistes et de mesurer l'impact de différentes configurations de ressources sur les performances globales.

Personnaliser le Réseau de Microservices : En configurant le maillage de services, ou le réseau de dépendances entre microservices, les chercheurs peuvent explorer l'efficacité des communications et la tolérance aux pannes dans divers arrangements architecturaux.

Choisir le Protocole de Communication : La flexibilité de choisir entre HTTP ou gRPC comme protocole de communication interservices offre la possibilité d'évaluer l'impact du protocole de communication sur la latence, le débit, et d'autres métriques de performance.

2.3.2 DeathStarBench

Un benchmark open source pour les microservices cloud. Composée de six services complets qui représentent un éventail varié de services cloud et edge couramment utilisés : un réseau social, un service multimédia , un site de e-commerce, la réservation d'hôtels, un système bancaire sécurisé et Swarm ; un service IoT destiné à la gestion et au contrôle de groupes de drones, avec ou sans support cloud. Au moment de la rédaction de ce mémoire, trois applications sur six sont disponibles Réseau social ,Service média , Réservation d'hôtel. [6]

Ces services sont bâtis à l'aide de nombreux microservices utilisant divers langages et modèles de programmation tels que Node.js, Python, C/C++, Java, JavaScript, Scala et Go, et intègrent des applications open source comme NGINX, memcached, MongoDB,

Cylon et Xapian. Pour assembler ces services de manière cohérente, des API RPC et RESTful personnalisées ont été développées en utilisant des cadres open source de premier plan tels qu'Apache Thrift et gRPC.

En outre, pour tracer le parcours des requêtes utilisateurs à travers les microservices, un système de traçage distribué, léger et transparent, similaire à Dapper et Zipkin, a été conçu. Ce système suit les requêtes au niveau RPC, associe les RPC qui font partie de la même requête de bout en bout et enregistre les traces dans une base de données centralisée. L'analyse couvre tant le trafic provenant d'utilisateurs réels que celui généré artificiellement par des simulateurs de charge en boucle ouverte.

2.2.2.1 Règles de conception

DeathStarBench suit des principes de conception clés pour garantir sa qualité et son efficacité [6] :

Représentativité : La suite est assemblée avec des applications open source couramment utilisées par les fournisseurs de cloud, telles que NGINX, memcached, MongoDB, RabbitMQ, MySQL, et le serveur http Apache, ainsi que des microservices spécifiques comme ceux de Sockshop de Weave. L'essentiel du nouveau code sert à connecter ces services via des interfaces comme Apache Thrift, gRPC, ou des requêtes HTTP.

Fonctionnement complet : Contrairement à certains composants open source qui opèrent isolément, DeathStarBench recrée l'ensemble du parcours d'un service, de la requête initiale du client jusqu'à sa résolution finale, en passant par le backend. Ceci permet de mesurer l'impact réel des interactions entre services sur la performance globale.

Hétérogénéité : La diversité des langages de programmation dans la suite reflète à la fois les défis et les opportunités posés par les microservices. Avec des langages allant du C/C++ au Python, en passant par Java, JavaScript, Node.js, Ruby, Go, et Scala, DeathStarBench explore comment différentes technologies influencent les performances, la synchronisation, et le développement.

Modularité : Inspiré par la loi de Conway, DeathStarBench adopte une structure où l'architecture logicielle des services reflète l'organisation de ceux qui les ont conçus. Cela aide à minimiser les dépendances fortes et à favoriser des composants indépendants et faiblement couplés, facilitant ainsi la communication et l'intégration.

Reconfigurabilité : Un des grands avantages des microservices est la facilité de mise à jour ou de remplacement des composants. DeathStarBench utilise des API RPC/HTTP flexibles permettant de substituer des microservices ou d'apporter des ajustements mineurs sans perturber l'ensemble du système.

2.2.2.2 Composants et architecture

le benchmark est composé de 5 applications suivantes [6] :

A. **Réseau social**

Objectif : Le service de bout en bout met en œuvre un modèle de réseau social avec des relations de suivi unidirectionnelles.

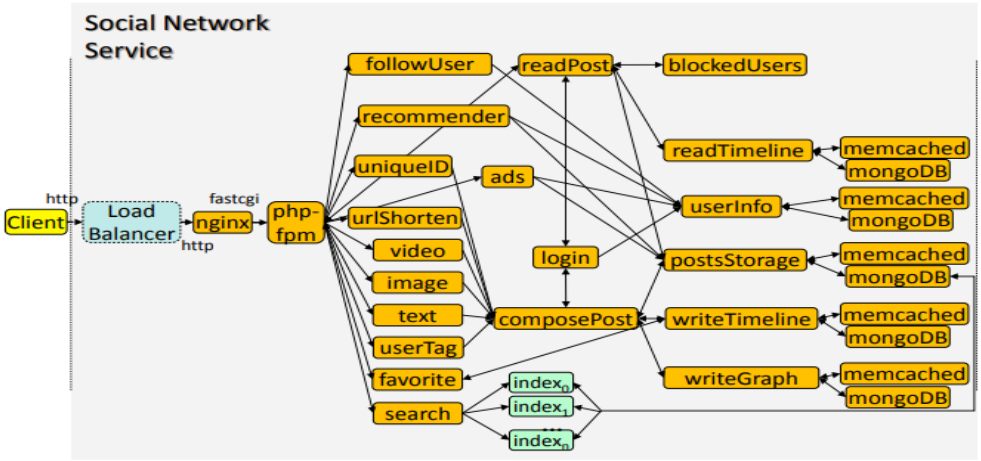

FIGURE 2.3 – L'architecture (graphe de dépendance des microservices) du réseau social [6]

Fonctionnalités : La figure 2.3 illustre l'architecture complète du service. Les utilisateurs (clients) envoient des requêtes via le protocole HTTP, qui sont d'abord dirigées vers un équilibreur de charge implémenté avec Nginx. Une fois qu'un serveur Web spécifique est sélectionné (également géré par Nginx), ce dernier communique avec les responsables des microservices. Ces microservices sont responsables de la rédaction et de l'affichage des publications, ainsi que de la gestion des publicités, des moteurs de recherche, etc. Tous les messages transitant par le module PHP-FPM sont des appels de procédure distante (RPC) via Apache Thrift .

Les utilisateurs peuvent créer des publications intégrant du texte, des médias, des

liens et des balises pour d'autres utilisateurs. Ces messages sont ensuite diffusés à tous leurs abonnés. Les utilisateurs ont également la possibilité de lire, de marquer comme favori et de republier des publications. Ils peuvent également répondre publiquement ou envoyer des messages directs à d'autres utilisateurs.

L'application intègre également des plugins d'apprentissage automatique, tels que des systèmes de publicité et des moteurs de recommandation d'utilisateurs . De plus, un service de recherche basé sur Xapian est disponible, ainsi que des microservices pour enregistrer et afficher les statistiques des utilisateurs (par exemple, le nombre d'abonnés) et pour gérer les abonnements, les désabonnements et le blocage d'autres comptes. Le backend du service utilise Memcached pour la mise en cache et MongoDB pour le stockage persistant des publications, des profils, des médias et des recommandations. Enfin, le service est équipé d'un système de traçage distribué , qui enregistre la latence de chaque requête réseau et de chaque traitement par microservice. Ces traces sont stockées dans une base de données centralisée.

B. **Service Média**

Objectif : L'application met en œuvre un service complet pour explorer les informations sur les films, permettant aux utilisateurs de consulter, noter, louer et diffuser des films .

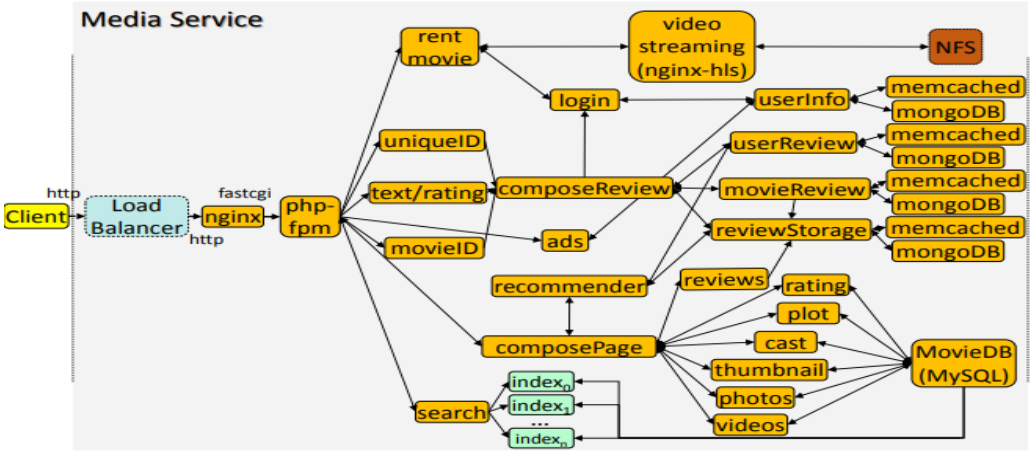

FIGURE 2.4 – L'architecture du Media Service pour la révision,location et streaming de films [6]

Fonctionnalités : La figure 2.4 illustre l'architecture du service. Tout comme pour le réseau social, les requêtes clients arrivent à l'équilibreur de charge, qui les répartit

entre plusieurs serveurs Web Nginx. Les utilisateurs peuvent rechercher et parcourir des informations sur les films, notamment leur intrigue, leurs photos, leurs vidéos, leur distribution, ainsi que consulter les critiques existantes. Ils ont également la possibilité d'ajouter de nouvelles critiques pour des films spécifiques en se connectant à leur compte.

De plus, les utilisateurs peuvent choisir de louer un film, ce qui implique un paiement. Pour vérifier que l'utilisateur dispose de suffisamment de fonds, un module d'authentification est utilisé. Le streaming vidéo est géré par Nginx-HLS, un module de production Nginx pour le streaming en direct via HTTP. Les fichiers vidéo actuels sont stockés dans NFS pour éviter la latence et la complexité liées à l'accès à des enregistrements fragmentés provenant de bases de données non relationnelles. Les critiques de films sont conservées dans Memcached et des instances MongoDB. Les informations sur les films sont stockées dans une base de données MySQL fragmentée et répliquée.

L'application propose également des recommandations de films et des publicités. En plus de cela, quelques services auxiliaires pour la maintenance et la découverte des services sont disponibles, bien qu'ils ne soient pas représentés sur la figure. Nous déployons actuellement le service Media de la même manière que le site d'hébergement pour le projet de démonstration à Cornell, où les membres de la communauté peuvent parcourir et rédiger des critiques.

C. **Service de e-commerce**

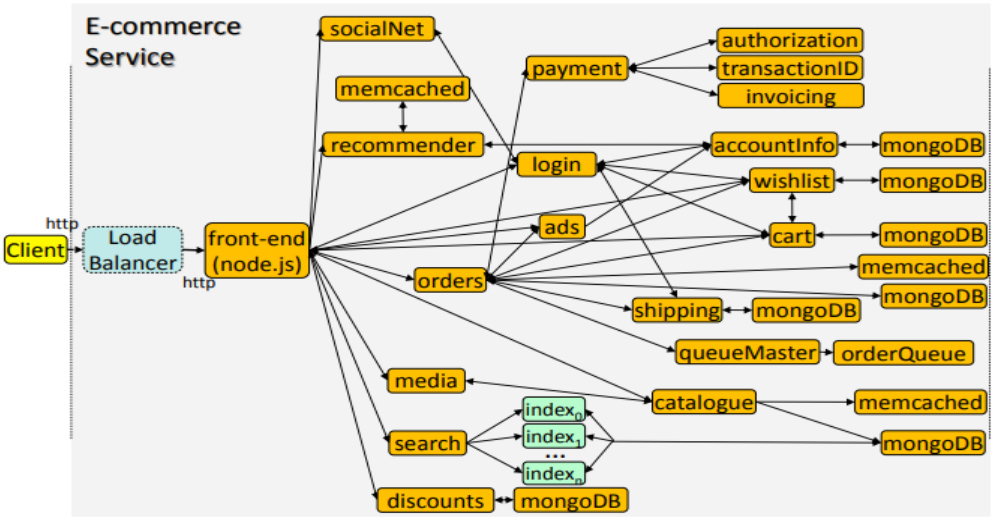

FIGURE 2.5 – L'architecture du service E-commerce [6]

Objectif : Le service de commerce électronique met en place un site dédié à la vente de vêtements. Son architecture s'inspire et utilise plusieurs composants de l'application open source Sockshop .

Fonctionnalités : La figure 2.5 illustre l'architecture complète du service. Dans ce cas, le frontal de l'application est un service Node.js. Les clients peuvent utiliser ce service pour explorer l'inventaire via le catalogue, qui est géré par un microservice Go. Ce dernier communique avec les instances back-end de Memcached et MongoDB, qui contiennent les informations sur les produits. Les utilisateurs ont la possibilité de passer des commandes (implémentées en Go) en ajoutant des articles à leur panier (développé en Java). Une fois connectés à leur compte, ils peuvent sélectionner les options d'expédition (Java), traiter leur paiement (Go) et obtenir une facture (Java) pour leur commande. Les commandes sont sérialisées et validées à l'aide de QueueMaster (Go). Enfin, le service intègre un moteur de recommandation pour les produits suggérés, ainsi que des microservices pour créer une liste de souhaits d'articles (Java) et afficher les réductions en cours.

C'est ainsi que le service de commerce électronique fonctionne, en offrant aux utilisateurs une expérience fluide pour l'achat de vêtements en ligne.

D. **Système bancaire**

Objectif : Ce service propose une plateforme bancaire fiable, offrant aux clients la possibilité de réaliser des transactions, de solliciter des crédits ou de gérer l'équilibre de leur carte de crédit en toute sécurité.

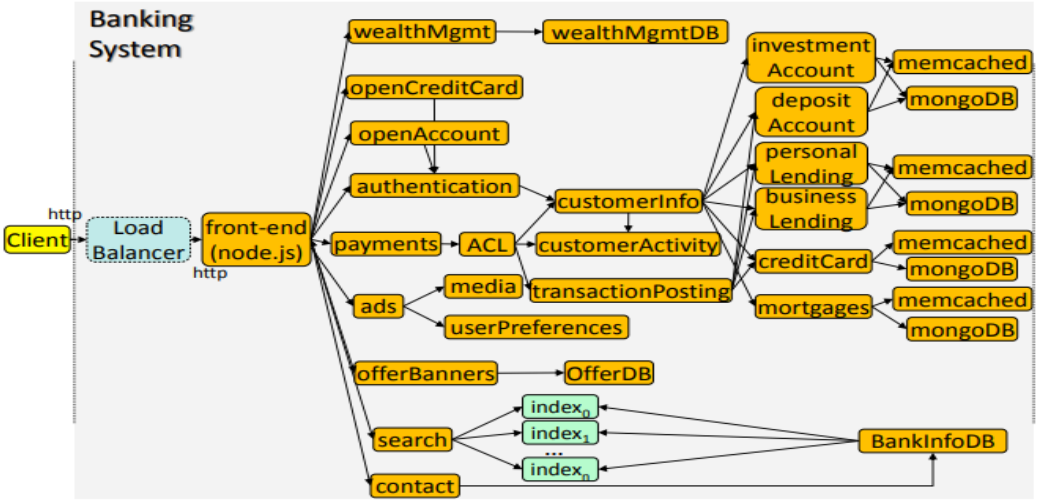

FIGURE 2.6 – L'architecture du service bancaire de bout en bout [6]

Fonctionnalités : Pour l'interface, on s'est inspiré des sites d'e-commerce avec un front-end élaboré en node.js, offrant une expérience utilisateur fluide pour se connecter à son espace personnel, s'informer sur les services bancaires ou entrer en contact avec un conseiller. Une fois connecté, le client a la possibilité de faire des paiements directement depuis son compte, de gérer le solde de sa carte de crédit, d'explorer et demander des prêts, ainsi que de découvrir des solutions pour optimiser ses investissements.

Sur le plan technique, le cœur de l'application repose principalement sur des microservices développés en Java et Javascript. Pour ce qui est du stockage des données, nous avons opté pour une combinaison de caches memcachés et de bases de données MongoDB pour une persistance efficace. De plus, le système intègre une base de données relationnelle, BankInfoDB, regroupant toutes les informations essentielles sur la banque, ses offres et les interlocuteurs disponibles.

E. **Swarm Coordination**

Objectif :Cet aspect du projet se penche sur une configuration innovante pour le déploiement de microservices, impliquant une exécution simultanée dans le cloud et sur des dispositifs périphériques. Il s'agit de superviser le pilotage d'un groupe de drones programmables, capables de réaliser des tâches telles que l'analyse d'images et la navigation à l'écart d'obstacles.

Fonctionnalités : Nous explorons deux approches pour ce service.

1. Première Approche (Figure 2.7a)

o La majorité des calculs sont effectués sur les drones eux-mêmes, notamment la planification des mouvements, la reconnaissance d'images et l'évitement d'obstacles.

o Le cloud se limite à construire l'itinéraire initial pour chaque drone (via le service Java ConstructRoute) et à conserver des copies persistantes des données des capteurs.

o Cette architecture réduit la latence élevée entre le cloud et les drones, mais elle est limitée par les ressources embarquées.

o Les contrôleurs (Controller et MotionController) sont implémentés en JavaScript, tandis que le module ImageRecognition utilise la bibliothèque jimp (un module Node.js) pour la reconnaissance d'images. L'évitement d'obstacles est géré en C++.

o Les services sur les drones communiquent nativement entre eux via IPC, tandis

que le cloud et les drones échangent des données via HTTP pour éviter d'installer des dépendances lourdes comme Thrift sur les appareils Edge.

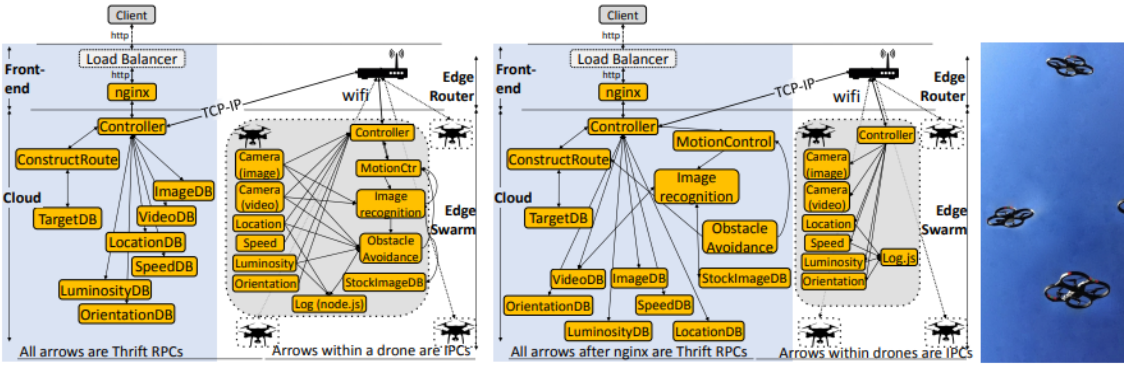

FIGURE 2.7 – Le service Swam s'exécutant (a) sur des appareils périphériques et (b) sur le cloud [6]

2. Deuxième Approche (Figure 2.7b)

o Dans cette version, le cloud prend en charge la majeure partie des calculs. o Il gère le contrôle des mouvements, la reconnaissance d'images et l'évitement d'obstacles pour tous les drones.

o Pour cela, il utilise les bibliothèques ardrone-autonomy et Cylon , respectivement implémentées en OpenCV et en JavaScript.

o Les appareils Edge se concentrent uniquement sur la collecte des données des capteurs et leur transmission vers le cloud.

o Ils enregistrent également certains diagnostics à l'aide d'un service de journalisation local (implémenté en Node.js).

o Dans ce scénario, presque toutes les actions subissent la latence du réseau entre le cloud et les appareils Edge, mais les services bénéficient des ressources supplémentaires du cloud.

2.3.3 TeaStore

TeaStore,est une plateforme de test et de benchmarking fondée sur l'architecture des microservices, basé sur un site web pour vendre des articles de thé. Cette application offre une flexibilité considérable, permettant aux utilisateurs de modifier divers paramètres pour ajuster leurs analyses. le benchmark est de cinq services distincts, qui met en œuvre un site Web de commerce électronique vendant des produits liés au thé ,chaque élément de

TeaStore propose des défis uniques en termes de performance et de gestion des contraintes, rendant la plateforme idéale pour l'exploration des différentes facettes des performances logicielles et l'optimisation des stratégies de modélisation.[7]

Grâce à sa diversité fonctionnelle et à sa conception modulaire, TeaStore se prête aussi bien aux environnements distribués qu'aux déploiements locaux, soulignant sa polyvalence. L'architecture de l'application est pensée pour être dynamiquement scalable, permettant l'ajout, la suppression, et la réplication de services et d'instances de service en temps réel. Cette capacité à s'adapter dynamiquement facilite une optimisation poussée et une gestion efficace des ressources, où la prise de décision concernant le déploiement de services et de ressources devient un exercice complexe mais enrichissant.

2.2.3.1 Architecture et fonctionnalités

L'image ci-dessous montre l'architecture et les fonctionnalités de TeaStore, qui est structurée autour de cinq services principaux, accompagnés d'un service de registre. Ce système garantit une communication fluide entre tous les composants via le service de registre. Notamment, l'interface utilisateur web (WebUI) effectue des requêtes auprès de divers services, notamment ceux dédiés à l'image, l'authentification, la persistance et la recommandation .[7]

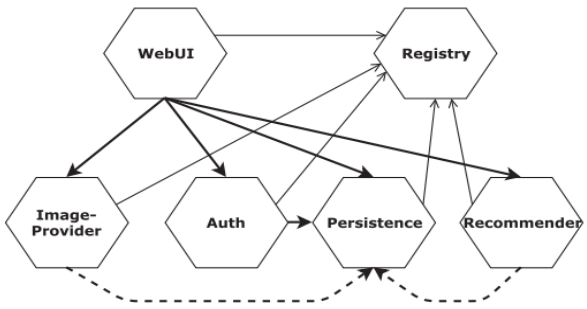

FIGURE 2.8 – Architecture du TeaStore [7]

Au démarrage, le service de fourniture d'images et le service de recommandation établissent une connexion initiale avec le service de persistance, illustrée par des lignes pointillées, pour leurs besoins spécifiques : le service d'images pour générer des visuels pour chaque produit et le service de recommandation pour compiler les données historiques des commandes en tant que matériel d'apprentissage. Une fois cette étape initiale complétée,

seuls les services d'authentification et l'interface WebUI interagissent activement avec le service de persistance pour la création, modification, et accès aux données.

Les interactions entre ces services se font via des API REST, un choix dicté par la prédominance de cette approche dans l'écosystème des microservices. Ces services sont hébergés sur des serveurs web Apache Tomcat, bien qu'ils puissent être déployés sur tout serveur d'applications Java compatible. Pour faciliter le déploiement, des images Docker contenant l'ensemble de l'infrastructure Tomcat sont également fournies, chaque service étant encapsulé dans son propre fichier war ou image Docker.

TeaStore intègre l'équilibreur de charge Ribbon pour gérer la réplication des instances de service, distribuant ainsi les requêtes REST entre les instances disponibles d'un service donné. Contrairement à certaines solutions comme Netflix Eureka, TeaStore mise sur son propre service de registre pour l'identification des instances de service disponibles, permettant ainsi une gestion dynamique des instances en cours d'exécution.

La fiabilité est assurée par un système de battements de cœur envoyés au registre par chaque service. En cas de surcharge ou de panne d'un service, ce mécanisme permet de le retirer automatiquement de la liste des instances disponibles, évitant ainsi l'envoi de requêtes vouées à l'échec. En tant qu'outil destiné aux benchmarks et aux tests, TeaStore est un projet open-source, compatible avec différentes solutions de monitoring, notamment grâce à des images Docker préconfigurées avec l'application de surveillance Kieker. Cette approche permet une instrumentation flexible et non intrusive, adaptable en temps réel.

L'interface WebUI joue un rôle central dans le traitement des requêtes, s'assurant que les données nécessaires sont toujours récupérées depuis le service de persistance. Les images nécessaires à la présentation des produits sont obtenues du service de fourniture d'images, et l'ensemble est compilé en une page Java Server Pages (JSP) avant d'être renvoyé à l'utilisateur. Cette méthodologie assure une expérience utilisateur cohérente, même pour des navigateurs simplifiés ou des générateurs de charge ne récupérant pas les images par défaut.

Enfin, pour une requête utilisateur concernant des détails de produit, l'interface WebUI suit un processus précis comme le montre la figure 2.9 : vérification de l'état de connexion via le service d'authentification, récupération des informations produit depuis le service de persistance, demande de recommandations personnalisées au service de recommandation, et intégration des images produits et recommandations en utilisant le service de fourniture

d'images. Toutes les données visuelles sont intégrées en encodage base64 dans la réponse HTML, assurant une présentation complète et détaillée du produit demandé.

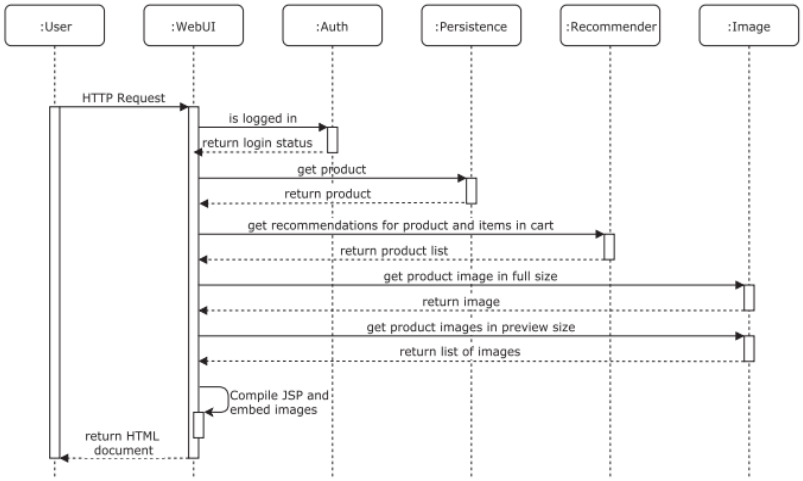

FIGURE 2.9 – Appels de service lors de la demande de page produit [7]

2.2.3.2 Prestations de service

Le TeaStore est une application complexe composée de plusieurs services, chacun jouant un rôle spécifique pour assurer le fonctionnement de la boutique en ligne [7] :

Interface Utilisateur (WebUI) : Pensez à cela comme le visage du TeaStore, l'endroit où les utilisateurs interagissent avec le site. Ce service s'occupe de montrer les produits, les catégories, les images et les recommandations. Il vérifie aussi les informations fournies par les utilisateurs avant de les envoyer à d'autres services pour traitement. La vitesse à laquelle les pages se chargent dépend grandement du type de contenu affiché, en particulier des images.

Fournisseur d'Images : Ce service est comme un photographe qui ajuste les images pour qu'elles s'affichent parfaitement sur la page web, en fonction de leur taille. Il stocke les images les plus demandées dans un cache pour accélérer l'affichage, ce qui signifie que la rapidité d'accès à une image dépend si elle est déjà dans le cache ou non.

Authentification : Imaginez ce service comme un gardien qui vérifie l'identité de chaque utilisateur à l'entrée. Il gère les informations de connexion et les sessions des utilisateurs, en utilisant des techniques de cryptage pour assurer la sécurité. La performance de ce service peut varier en fonction de la quantité d'informations stockées dans chaque session utilisateur.

Recommender : C'est le conseiller personnel de shopping du TeaStore, qui suggère des produits aux utilisateurs en fonction de leurs goûts et des achats d'autres clients. Ce service apprend à partir d'un ensemble de données pour offrir des recommandations pertinentes, et son efficacité dépend de l'algorithme de recommandation utilisé.

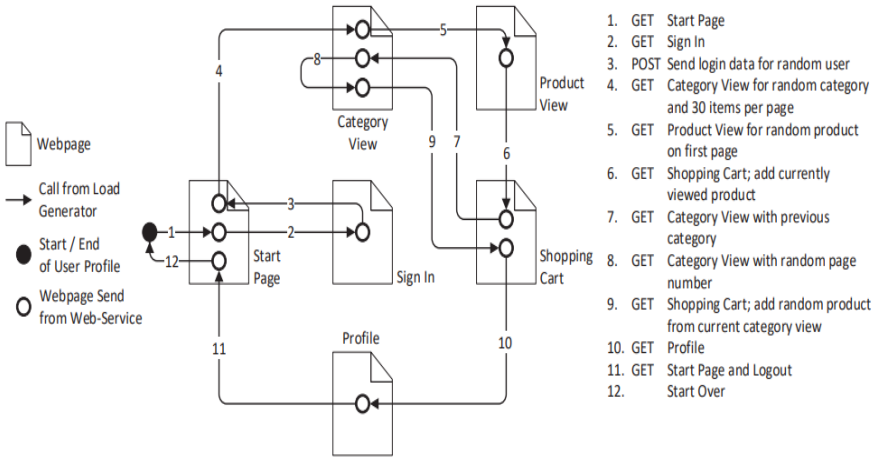

FIGURE 2.10 – Profil utilisateur «Parcourir» configuré dans le générateur de charge HTTP pour nos cas d'utilisation, y compris les pages Web livrées et HTTP type de demande. Les pages Web inutilisées sont omises pour plus de clarté [7]

Persistance : Ce service agit comme une bibliothèque qui stocke et gère toutes les informations sur les produits, les catégories, les achats, et les utilisateurs. Il utilise un système de cache pour rendre l'accès à ces données plus rapide et efficace, réduisant ainsi la charge sur la base de données principale.

Registre : Bien qu'il ne fasse pas directement partie de l'application testée, le registre est essentiel car il tient à jour une liste de tous les services en cours d'exécution, facilitant ainsi leur découverte et communication.

TraceRepository : Lorsque la surveillance est activée, ce service collecte et stocke les données sur l'utilisation et la performance des différents services du TeaStore. Cela aide à analyser et à optimiser le fonctionnement de l'application.

2.3.4 Online Boutique

Online Boutique est une application web de démonstration conçue pour montrer les capacités des microservices dans le cloud. Elle simule un site e-commerce permettant aux utilisateurs de sélectionner des produits, les ajouter à un panier et effectuer des achats.[8]

Développée par Google, elle sert d'exemple pratique pour montrer comment utiliser des technologies avancées telles que Kubernetes, GKE (Google Kubernetes Engine), Istio, Stackdriver et gRPC. Compatible avec tout environnement Kubernetes, y compris GKE, Online Boutique peut être mise en place facilement et rapidement, nécessitant minimal ou aucun ajustement préalable.[8]

2.2.4.1 L'architecture de boutique en ligne

Online Boutique est construite autour de 11 microservices comme montre la figure 2.11, chacun développé dans un langage de programmation spécifique, ils communiquent entre eux en utilisant : le gRPC [8]

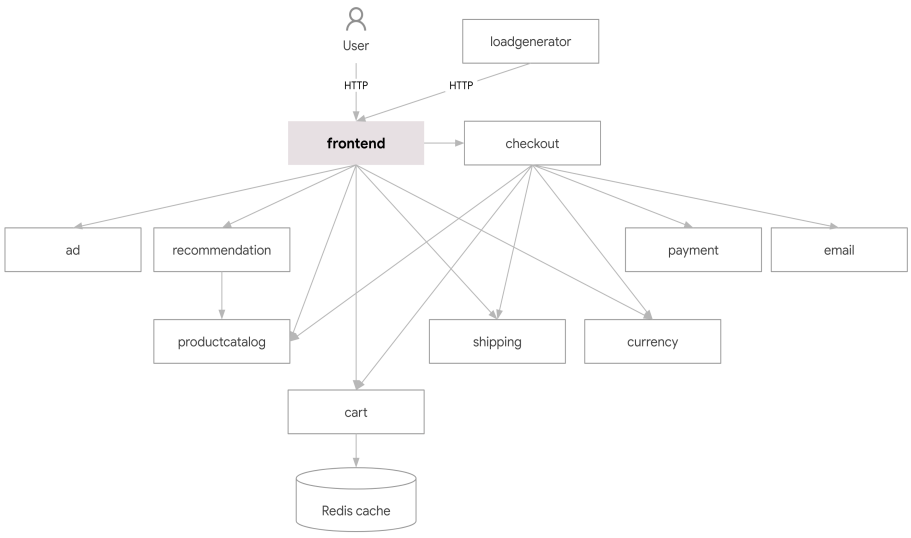

FIGURE 2.11 – L'architecture de boutique en ligne [8]

Le service Frontend : est ecrit en Go et Sert le site web sans nécessiter d'inscription, en attribuant des identifiants de session automatiques.

Le Cartservice : en C#, gère le panier d'achats en sauvegardant et récupérant les articles via Redis.

Le Productcatalogservice : en Go, offre un catalogue de produits et supporte la recherche et l'affichage de détails produits.

Le Currencyservice : écrit en Node.js, convertit les montants dans différentes devises, basé sur les taux de la Banque centrale européenne.

Le Paymentservice : également en Node.js, simule le traitement des paiements par carte de crédit, retournant un identifiant de transaction.

Le Shippingservice : en Go, estime les frais de port et simule l'envoi des commandes.

Le Emailservice : en Python, envoie des emails de confirmation de commande.

Le Checkoutservice : en Go, finalise les achats en orchestrant le panier, le paiement, la livraison et l'envoi de notifications.

Le Recommendationservice : en Python, suggère des produits basés sur le contenu du panier.

L'Adservice : en Java, affiche des annonces textuelles adaptées au contexte. Enfin,

le Loadgenerator : en Python/Locust, génère un trafic artificiel pour tester le système.

2.3.5 Bookinfo

2.2.5.1 Définition ISTIO

Istio est un maillage de services open source qui s'intègre de manière fluide avec les applications distribuées existantes.Les capacités avancées d'Istio fournissent une approche cohérente et optimisée pour sécuriser, connecter et surveiller les services de manière efficace.Istio facilite l'équilibrage de

charge, l'authentification de service à service et la surveillance, nécessitant peu ou pas de modifications du code de service. Parmi les applications qui tirent parti d'Istio en raison de ses avantages, on trouve Bookinfo.[25]

2.2.5.2 Définition Bookinfo

L'application Bookinfo simule une librairie en ligne, présentant chaque livre avec sa description, ses caractéristiques détaillées comme le nombre de pages, ainsi que les avis et commentaires des utilisateurs.Bookinfo est constitué de quatre microservices autonomes créés avec des langages de programmation différents,Ces services fonctionnent de manière indépendante grace à l'Application Service Mesh (ASM). [26] [27]

L'application BookInfo est divisée en quatre microservices distincts :

Page produit :page produit :Le microservice productpage fait des appels au microservice details et inspecte les microservices pour remplir le contenu de la page.

Détails : Le microservice details conserve des informations sur le livre.

Commentaires : Le microservice de critiques inclut des avis sur les livres, et il communique également avec le microservice de notation.

Évaluations : Le microservice de notation établit des classements de livres en fonction des critiques associées à chaque livre.

2.2.5.3 Architecture Bookinfo

L'image ci-dessous (2.12) montre l'architecture de l'application BookInfo avec Istio, expliquant comment les requêtes sont acheminées et traitées à travers les différents microservices. Lorsqu'un utilisateur déclenche une requête, celle-ci est tout d'abord acheminée vers le microservice Productpage, qui joue le rôle de premier point d'accès dans le système. Ensuite, Istio assure la gestion du cheminement de la requête à travers les divers microservices comme Details, Reviews, et Ratings en utilisant des règles de trafic configurées. Les proxies Envoy, inclus dans chaque conteneur grâce à Istio, facilitent les échanges entre les microservices. Ils sont responsables de la surveillance des erreurs et de la mise en œuvre de stratégies de sécurité telles que l'authentification et l'autorisation.

À la fin du processus, les microservices génèrent la réponse qui est ensuite renvoyée au client d'origine. Cela assure un traitement des demandes efficace, sécurisé et sous surveillance au sein de l'application BookInfo.[9]

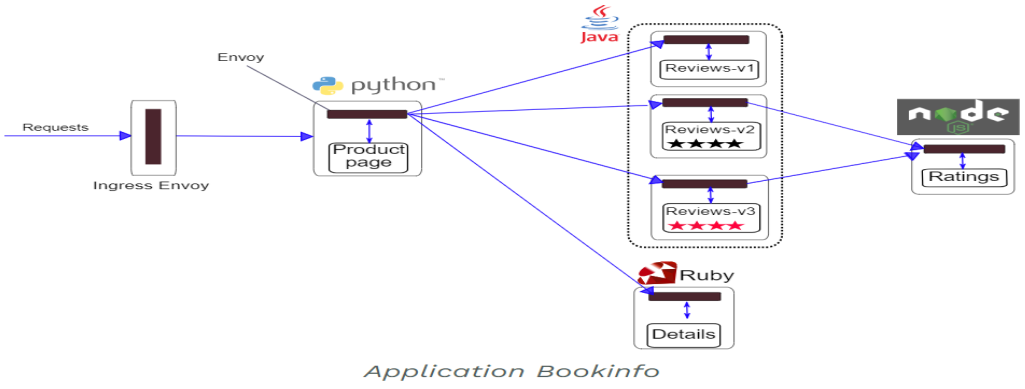

FIGURE 2.12 – Architecture BOOKINFO [9]

2.3.6 Petclinic

Petclinic est une application de gestion de clinique vétérinaire conçue comme un exemple d'application d'entreprise utilisant le Spring Framework. Elle permet de gérer les informations des propriétaires d'animaux de compagnie, les vétérinaires, les visites et les traitements des animaux. En combinant simplicité et efficacité, Petclinic démontre l'utilisation du Spring Framework pour développer des applications orientées bases de données.[28]

2.2.6.1 Histoire de Petclinic (LES ORIGINES)

En 2003, Kren Krebs et Juergen Hoeller ont créé la Spring PetClinic pour démontrer les capacités du Spring Framework, sorti en 2004. En 2013, Keith Donald, Michael Isvy et Costin Leau ont transféré le projet sur GitHub avec Spring Framework 3. En 2016, PetClinic est passé à Spring Boot sous la direction de Dave Syer et Stéphane Nicoll. Actuellement, la communauté entretient 9 forks de Spring PetClinic.[29]

2.2.6.2 Architecture de PetClinic

L'architecteure est la suivante (figure 2.13) :[30]

A. **Microservices de back end**

L'architecture de Petclinic comprend trois microservices back (Arrière-plan)qui offrent les fonctionnalités de l'application via une API REST.

Service Clients (Customers) : est responsable de la gestion des données et des actions associées aux clients, notamment l'ajout, la modification et la suppression des informations client.etc

Service Vétérinaires (Vets) : gère les données relatives à la disponibilité des vétérinaires, y compris leurs horaires de travail et leurs domaines de spécialisation.etc

Service Visites (Visits) : gère les rendez-vous et les consultations des clients avec les vétérinaires, incluant les plannings, les détails des rendez-vous, etc.

Ces microservices peuvent démarrer simultanément plusieurs instances du même microservice pour garantir la scalabilité horizontale et la tolérance aux pannes. Pour assurer son autonomie, chaque microservice possède sa propre base de données.

En revanche, toutes les instances d'un microservice donné utilisent la même base de

données.

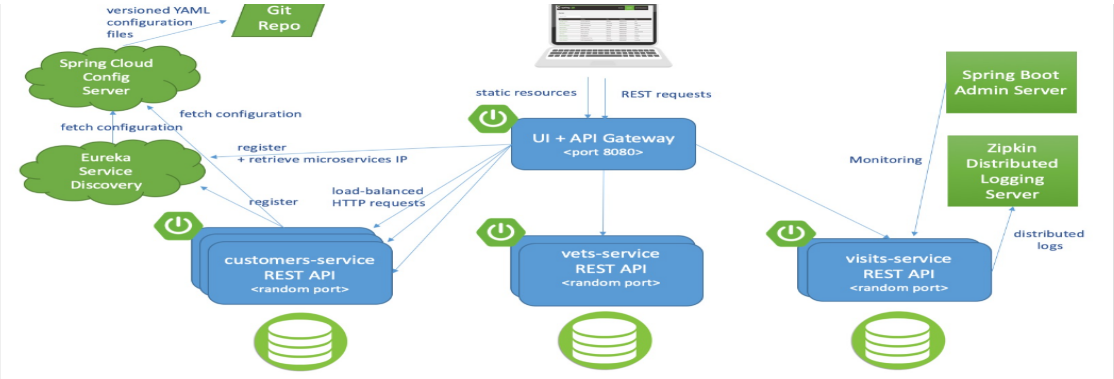

FIGURE 2.13 – Architecture de Petclinic [10]

B. **Microservice de frontend :** le front-end désigne la partie de l'application que les utilisateurs voient et avec laquelle ils interagissent directement, telle que l'interface utilisateur graphique (GUI) via un navigateur web.Le front-end ne communique pas directement avec les trois microservices principaux. Au lieu de cela, il passe par le microservice Front API Gateway, qui agit comme une passerelle pour gérer les requêtes et les réponses entre le front-end et les microservices back-end.

   En plus de cela, le Front API Gateway s'occupe également de servir les ressources statiques, comme les fichiers CSS, JavaScript et les images, pour l'interface utilisateur.

C. **Les microservices d'infrastructures**

   L'annuaire de service Eureka :permet aux microservices de s'enregistrer dynamiquement et de communiquer entre eux sans avoir besoin de connaître à l'avance les adresses IP des autres microservices.

   Serveur de Config : Le Serveur de Configuration fournit les réglages nécessaires aux microservices et aux serveurs pour leur fonctionnement. Cela simplifie la gestion des configurations dans un système distribué.

   Zipkin :permet d'enregistrer les traces des appels entre les microservices dans son serveur de logs distribués. Cela permet d'analyser les performances et le flux des opérations spécifiquement au sein des microservices dans une application distribuée.

   Spring Boot Admin : joue un rôle essentiel dans la supervision et l'administration des microservices. Il fournit aux administrateurs une interface centralisée pour

surveiller et gérer efficacement ces services, simplifiant ainsi la gestion opérationnelle dans un environnement distribué.

2.3.7 Sock Shop

Le benchmark Sock Shop offre une plateforme complète pour explorer les fonctionnalités et les bénéfices des microservices, des conteneurs, et des technologies cloud natives.

2.2.7.1 Architecture et Conception

Sock Shop simule un site de vente en ligne spécialisé dans les chaussettes, adoptant une architecture de microservices. Chaque service gère une fonctionnalité spécifique, ce qui facilite leur développement,déploiement et mise à l'échelle indépendants.

Cette approche modulaire améliore l'isolation des erreurs,simplifie le suivi des bugs et permet une évolutivité accrue par rapport aux architectures monolithiques classiques.

L'application se compose de plusieurs microservices clés, à savoir :

.Service Utilisateur : Gère les informations des utilisateurs et assure l'authentification.

.Service Catalogue : Gère les listes de produits et la gestion des stocks.

.Service Panier : Gère les opérations liées au panier d'achat.

.Service Commande : Traite les commandes en intégrant des services de paiement et de livraison.

Les microservices sont rédigés dans différents langages de programmation tels que Go, Java et Node.js,ce qui souligne la flexibilité de ces services vis-à-vis des langages. Cette approche polyglotte offre aux développeurs la possibilité de choisir le langage le plus approprié pour chaque service, ce qui permet d'optimiser les performances, la fiabilité et la scalabilité de chaque composant.[31]

2.2.7.2 Sock shop comme outil educatif

Le Sock Shop n'est pas seulement un outil de démonstration, mais il sert également de ressource éducative précieuse. Il offre aux développeurs et aux architectes système un exemple concret de la structure et de l'interaction des microservices dans un environnement natif de cloud. En explorant le code source et l'architecture, les utilisateurs peuvent obtenir des idées sur les meilleures pratiques pour construire et déployer des microservices.

En outre, Sock Shop est fréquemment employé pour expérimenter et comparer diverses plateformes et technologies cloud. Les développeurs s'en servent pour évaluer la performance et les fonctionnalités de différentes mises en œuvre de service mesh comme Istio et Linkerd, ainsi que pour explorer les configurations Kubernetes. Cela permet de mieux saisir l'impact des différentes technologies sur les performances et le fonctionnement des microservices dans des situations concrètes.

2.2.7.3 Technologies utilisées et déploiement

Le Sock Shop est conteneurisé avec Docker, ce qui encapsule chaque microservice dans son propre conteneur, garantissant que chaque composant a ses dépendances satisfaites sans conflit avec les autres. Cette conteneurisation est cruciale pour créer des environnements de développement, de test et de production fiables et reproductibles. Les conteneurs sont gérés et orchestrés avec Kubernetes, qui automatise le déploiement, le dimensionnement et les opérations de ces applications conteneurisées à travers des clusters de machines hôtes.

Kubernetes rend le déploiement plus simple tout en renforçant la résistance et la capacité de mise à l'échelle des applications. Il propose des fonctionnalités telles que l'autoréparation, les déploiements et les retours automatiques en arrière, la découverte de services et l'équilibrage de charge. Ces fonctionnalités combinées garantissent que le Sock Shop peut s'adapter dynamiquement à la demande et récupérer automatiquement des pannes, ce qui améliore considérablement la disponibilité et la fiabilité globales du service.[31]

2.3.8 JPetStore

JPetStore représente un exemple remarquable d'application de microservices qui illustre les pratiques modernes de développement logiciel en s'intégrant avec des technologies avancées cloud- native. Elle a été convertie d'une application web Java classique en une architecture de microservices dynamique en utilisant Docker, Kubernetes et des services d'IA de IBM Cloud. Cet exemple complet montre comment les anciennes applications ont été transformées en systèmes modernes et évolutifs, adaptés aux besoins numériques actuels. Voici une analyse détaillée de ses divers composants et approches.

2.2.8.1 Architecture et Composants

À l'origine construit sur la pile technologique J2EE, JPetStore était une application monolithique utilisant les technologies Java standard. L'application a été revue dans une architecture de microservices, fragmentant le monolithe en services plus petits et autonomes. Chaque microservice est désormais conteneurisé, permettant un développement, un test et un déploiement isolés, renforçant ainsi l'agilité et la scalabilité de l'application.[32] [33] Les microservices inclus sont :

Service de Catalogue : Gère les listings de produits, les catégories et l'inventaire.

Service de Compte : Gère les comptes utilisateurs, l'authentification et l'autorisation.

Service de Commande : Traite les commandes, incluant le traitement des paiements et l'expédition.

Ces services sont développés en utilisant Spring Boot et exploitent Netflix OSS pour les modèles de microservices, ainsi que MyBatis pour la gestion relationnelle des données. Ils interagissent via des protocoles légers tels que REST et utilisent JSON pour l'échange de données.

2.2.8.2 Processus de développement et de déploiement

La modernisation de JPetStore comprend des instructions détaillées de configuration et de déploiement qui couvrent :

Configuration de l'environnement : Préparation des services IBM Cloud, création de clusters Kubernetes et configuration des services nécessaires comme Watson Visual Recognition et Twilio.

Construction et publication d'images Docker : Chaque microservice est transformé en une image Docker et publié dans un registre de conteneurs, à partir duquel ils sont déployés dans le cluster Kubernetes.

Gestion des secrets : Gestion des informations sensibles via les secrets Kubernetes, assurant que les clés d'API et autres informations d'identification sont sécurisées et gérées de manière centralisée.

2.2.8.3 L'intégration avec l'IA et les services externes

L'intégration de IBM Watson Visual Recognition dans JPetStore constitue une amélioration significative. Ce service d'intelligence artificielle permet à l'application de réaliser des recherches à partir d'images et de reconnaître des attributs spécifiques. Cette intégration enrichit considérablement l'expérience utilisateur en autorisant des recherches visuelles de produits et des recommandations fondées sur le contenu des images.

De plus, l'application intègre les APIs de messagerie de Twilio pour faciliter la communication via SMS et MMS. Cette fonctionnalité permet à JPetStore d'envoyer des notifications et d'interagir directement avec les clients sur leurs appareils mobiles, rendant ainsi l'expérience d'achat plus interactive et réactive.[32]

2.2.8.4 Modernisation avec Conteneurs et Kubernetes :

La transformation de JPetStore en une application conteneurisée implique d'encapsuler chaque microservice dans des conteneurs Docker. Docker fournit un environnement isolé pour chaque service, garantissant que les dépendances sont contenues et les conflits évités. Ceci est essentiel pour maintenir une cohérence entre les environnements de développement, de test et de production. Kubernetes est utilisé pour orchestrer ces conteneurs, gérant leur cycle de vie depuis le déploiement jusqu'à l'évolutivité et la réparation. Kubernetes propose des fonctionnalités avancées telles que :[32]

Auto-scaling : Ajuste automatiquement le nombre de conteneurs actifs en fonction de la charge.

Équilibrage de charge : Répartit le trafic entrant entre les conteneurs pour garantir une utilisation optimale.

Autoréparation : Remplace automatiquement les conteneurs défaillants pour maintenir l'état désiré de l'application.

2.4 Comparaison des Benckmarks pour les microservices

le tableau ci-dessus 2.1 propose une comparaison détaillée entre les applications de benchmarking de microservices les plus populaires, illustrant leurs fonctionnalités et performances pour guider les utilisateurs dans leur sélection.[34]

	µBench	DeathStar Bench	TeaStore	Online Boutique	BookInfo	SockShop	JPetStore	PetClinic
Objectifs	Evaluer, Comparer	Evaluer, Comparer, Demo	Evaluer, Comparer, Demo	Evaluer, Comparer, Demo	Evaluer, Comparer, Demo	Evaluer, Comparer, Demo	Evaluer, Comparer, Demo	Evaluer, Comparer, Demo
Nombre de microservices	configurable	19, 31, 33	6	11	4	14	4	5
Nombre d'applications	configurable	3	1	1	1	1	1	1
languages	python	Node.js, C/C++, Java, JavaScript, Scala, Go, Python	Java	Go, C#, Node.js, Python	Java, Python, Node.js, Ruby	Java, Go, Node.js	Java	Java

Appeler une méthode	REST, gRPC	REST	REST	gRPC	gRPC	REST	REST	REST
Modèle de travail	Configurable	Dépendant de l'application	Dépendant de l'application	Dépendant de l'application	Dépendant de l'application	Dépendant de l'application	Dépendant de l'application	Dépendant de l'application
Maillage de services	Configurable	Dépendant de l'application	Dépendant de l'application	Dépendant de l'application	Dépendant de l'application	Dépendant de l'application	Dépendant de l'application	Dépendant de l'application
Modèle exhaustif	stochastique, basé sur des traces	guidé par l'utilisateur	guidé par l'utilisateur	guidé par l'utilisateur	guidé par l'utilisateur	guidé par l'utilisateur	guidé par l'utilisateur	guidé par l'utilisateur
Plateformes	Kubernetes	Kubernetes, Docker compose, Openshift	Kubernetes, Docker compose	Kubernetes	Kubernetes	Kubernetes, Docker compose	Kubernetes	Kubernetes, CloudFoundry
performance exportée	métrique, traces	métrique, traces	métrique, traces	métrique, traces	métrique, traces	métrique	métrique, traces	métrique

TABLE 2.1 – Tableau Comparatif des Performances des Applications de Benchmarking

2.5 Critères de Performance Importants pour les Microservices

Les principaux critères de performance pour évaluer l'efficacité d'une architecture basée sur les microservices incluent le temps de réponse, le débit, la scalabilité, la disponibilité, et la résilience. Voici une discussion détaillée de chaque critère [35] [36] [37] [38] [39] :

1. **Temps de réponse :** Le temps de réponse est critique dans les microservices car il mesure combien de temps il faut pour qu'une requête soit traitée et une réponse soit renvoyée. Un système de microservices bien optimisé devrait minimiser ce temps pour améliorer l'expérience utilisateur.
2. **Débit :**Le débit, souvent mesuré en transactions par seconde, est un indicateur de la quantité de travail qu'un système peut traiter dans un intervalle de temps donné. Pour les microservices, il est important d'avoir un haut débit pour gérer de grandes volumes de requêtes simultanées sans dégradation des performances.
3. **Scalabilité :**La capacité d'un système à gérer une augmentation de la charge de travail en ajoutant des ressources (horizontalement ou verticalement) est cruciale. Les microservices doivent être conçus pour être facilement scalables pour répondre à des charges variables.
4. **Disponibilité :**La disponibilité mesure la capacité d'un système à rester opérationnel et accessible. Les microservices devraient garantir une haute disponibilité même en cas de panne de l'un des services.
5. **Résilience :**La résilience est la capacité d'un système à récupérer rapidement d'échecs et à continuer à fonctionner même en conditions dégradées. Les microservices doivent être conçus avec des mécanismes de tolérance aux pannes pour maintenir le service opérationnel.

Ces critères sont interdépendants et souvent, améliorer l'un peut affecter les autres. Par exemple, augmenter le débit peut parfois augmenter le temps de réponse si les ressources ne sont pas suffisamment scalées. Ainsi, il est crucial de considérer tous ces aspects lors de la conception et de l'optimisation des systèmes basés sur les microservices.

2.6 Outils et Méthodes couramment utilisés pour le Benchmarking des Microservices

Le benchmarking des microservices est une pratique essentielle pour évaluer les performances, la fiabilité et la scalabilité des architectures basées sur les microservices. Voici quelques outils et méthodes couramment utilisés pour le benchmarking des microservices [40] [41] [42] [43] [44][45] [46][47] :

A. **Les Outils**

JMeter : Un outil open source pour tester la charge et la performance des services. Il est utilisé pour simuler un grand nombre de requêtes simultanées pour tester la force et la performance des microservices.

Gatling : Un autre outil de test de performance puissant, conçu pour les applications web modernes. Il est particulièrement utile pour simuler des scénarios d'utilisateurs complexes avec des microservices.

Locust : Un outil de test de charge distribué écrit en Python. Il permet aux développeurs de écrire des scripts de test de charge en Python, ce qui est utile pour tester des architectures de microservices complexes.

Taurus : Un outil d'automatisation de tests qui permet de créer des tests de performance pour divers outils de test de charge comme JMeter, Gatling et Selenium. Il est idéal pour l'intégration dans des pipelines CI/CD.

Prometheus et Grafana : Prometheus, utilisé pour la surveillance et l'alerte, combiné avec Grafana pour la visualisation, offre une solution puissante pour monitorer les performances des microservices en temps réel.

Kubernetes et Istio : Kubernetes permet de gérer des clusters de conteneurs, ce qui est crucial pour déployer des microservices à grande échelle. Istio, un service mesh pour Kubernetes, aide à gérer la communication inter-services, offrant des métriques précieuses pour le benchmarking.

Chacun de ces outils apporte une valeur ajoutée spécifique dans le cadre du benchmarking des microservices, permettant aux équipes de développement de mieux comprendre et optimiser leurs architectures.

B. **Les Méthodes**

Pour réaliser le benchmarking des microservices, plusieurs méthodes sont utilisées pour évaluer leur performance et leur efficacité :[48] [49] [50]

Définition d'objectifs clairs et documentation rigoureuse : Avant de commencer le benchmarking, il est crucial de définir clairement les objectifs et de documenter le processus dans une feuille de calcul ou un système similaire. Cela inclut la définition des métriques clés, la planification des expériences, et la documentation des configurations de l'environnement de test pour garantir la reproductibilité.

Utilisation d'outils de génération de charge synthétique : Pour simuler des conditions de test réalistes, des outils comme JMeter ou Locust sont souvent utilisés pour générer un trafic synthétique qui imite le comportement des utilisateurs réels. Cela permet de mesurer la performance des microservices sous diverses charges.

Tests continus et monitoring : Il est conseillé de réaliser des benchmarks régulièrement et de monitorer les performances en continu pour détecter rapidement les dégradations ou les améliorations suite aux modifications du code ou de l'architecture. L'utilisation d'outils de surveillance en temps réel comme Grafana ou InfluxDB aide à visualiser et analyser les performances de manière proactive.

Analyse des résultats et ajustements : Après la collecte des données de performance, il est important d'analyser les résultats pour comprendre l'impact des différentes configurations et ajustements. Des analyses statistiques comme le calcul de la déviation standard des résultats peuvent aider à évaluer la fiabilité des tests et à identifier les besoins de modification pour optimiser la performance.

Ces méthodes combinées offrent une approche complète et efficace pour le benchmarking des microservices, permettant ainsi d'assurer des performances optimales et une meilleure compréhension des interactions entre services.

2.7 Conclusion

Ce chapitre a exploré divers benchmarks et outils conçus pour évaluer les architectures de microservices. À travers une analyse comparative, nous avons identifié les critères de performance cruciaux, mettant en lumière la modularité, l'évolutivité, et la résilience des microservices. Les outils et méthodes discutés illustrent les meilleures pratiques pour un benchmarking efficace et précis. Cette exploration renforce notre compréhension des outils adaptés à chaque besoin spécifique, contribuant à optimiser les performances des systèmes de microservices.

Chapitre 3

Tests et Résultas

3.1 introduction

Ce chapitre présente une étude comparative approfondie de la performance des microservices en utilisant l'application Social Network de DeathStarBench. L'objectif est de mesurer et d'analyser la réponse du système sous différentes charges, en observant des métriques clés telles que la latence, l'utilisation du CPU et la mémoire. Ces tests permet d'évaluer les performances de cette application basée sur les microservices.

3.2 Environnement de travail

3.2.1 Environnement matériel

Pour réaliser ce mémoire, nous avons utilisé un ordinateur HP pavilion laptop 15-eh1xxx , Intel Core AMD Ryzen 5 5500U 2.10 GHz, RAM 16Go, Système d'exploitation Windows 11, 64 bit. Nous avons installé une machine virtuelle Ubuntu version 20.04 LTS sur VMware. Cette configuration nous permet de travailler dans un environnement Linux tout en utilisant Windows. Dans notre cas, nous avons alloué 8 Go de RAM et 2 vCPUs à notre machine virtuelle Ubuntu 64 bits.

3.3 Architecture du social network

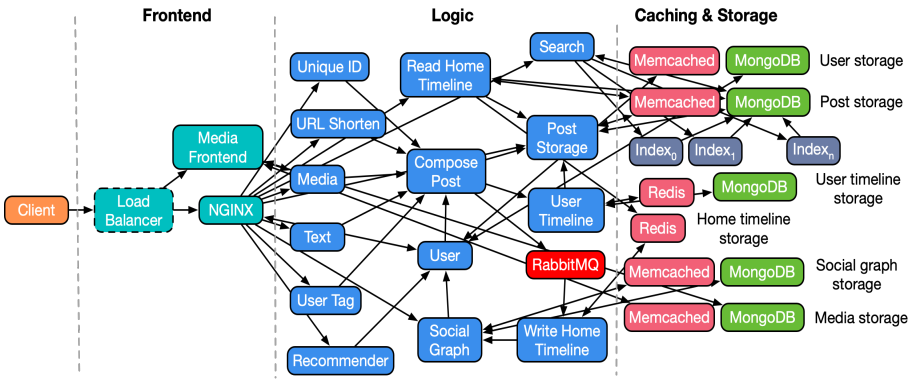

FIGURE 3.1 – Architecture du social network [11]

L'architecture du réseau social représentée dans la figure 3.1 est divisée en plusieurs sections principales : le Frontend, la Logique, et le Caching & Storage. Concentrons-nous sur les services spécifiques "Compose Post", "Read Home Timeline", et "Read User Timeline", ainsi que sur leurs relations et interactions.

3.3.1 Relations et Interactions

Les trois services "Compose Post", "Read Home Timeline" et "Read User Timeline" sont étroitement liés et interagissent de manière fluide pour garantir une expérience utilisateur cohérente.

1. **Compose Post and Timelines :** Le service "Compose Post" est essentiel pour la mise à jour des timelines. Chaque nouveau post créé est propagé via RabbitMQ aux services responsables des timelines, assurant que les nouvelles informations sont rapidement disponibles pour lecture.
2. **Storage Interaction :** Tous ces services dépendent des systèmes de stockage (MongoDB, Redis, Memcached) pour stocker et récupérer les données efficacement.
3. **Flow of Information :** Le flux d'information commence généralement avec une action utilisateur (comme composer un post), passe par les services de logique (comme "Compose Post"), et se termine par des mises à jour de stockage et la lecture des timelines via "Read Home Timeline" et "Read User Timeline".

   En résumé, ces services et leurs interactions forment le cœur de la fonctionnalité de réseau social, assurant que les utilisateurs peuvent créer des posts, et voir les mises

à jour de leurs propres timelines ainsi que celles de leurs amis en temps réel.

3.4 Méthodologie des tests

La méthodologie suivie pour cette étude comprend plusieurs étapes :

1. **Installation et Configuration :** Nous avons installé Docker et Docker Compose (figure 3.2), et vérifié la disponibilité des ports nécessaires pour l'application.

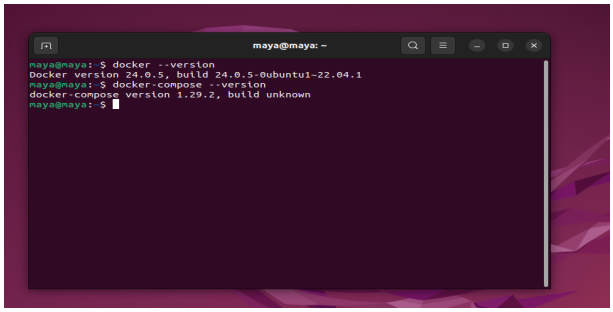

FIGURE 3.2 – Vérification de l'installation de Docker et Docker Compose.

2. **Démarrage des Conteneurs Docker :** Les conteneurs ont été démarrés à l'aide de Docker Compose.
3. **Enregistrement des Utilisateurs et Construction du Graph Social :** Les utilisateurs ont été enregistrés et le graphe social a été construit en utilisant un script Python(init_social_graph.py)pour simuler différents types de réseaux sociaux.
4. **Lancement des tests :** Pour lancer le test,nous avons compilé et utilisé l'outil wrk2 pour générer des requêtes HTTP vers l'application Social Network.

   wrk2 est un outil de génération de charge HTTP de haute performance, dérivé de l'outil original wrk. Il est conçu pour tester les performances des systèmes web en simulant un nombre élevé de requêtes par seconde (RPS), avec une durée de 30 minutes pour chaque test. Avant de lancer les tests, nous vidons les différentes bases de données.
5. **Visualisation des Traces Jaeger :**

   Jaeger est une plateforme open-source pour le traçage distribué, développée initialement par Uber Technologies. Elle est utilisée pour surveiller et dépanner les transactions dans les systèmes microservices distribués. Jaeger permet de contrôler la performance des applications en analysant les requêtes qui passent à travers différents

services, en fournissant une vue d'ensemble des temps de latence et des dépendances entre services.

FIGURE 3.3 – Jaeger [12]

Accès à Jaeger via `http://localhost:16686` pour visualiser les traces des requêtes.les figures 3.4, 3.5 et 3.6 représentant respectivement exemples de trace Jaeger pour composition de Posts, lecture des timelines d'Accueil et lecture des timelines des utilisateurs.

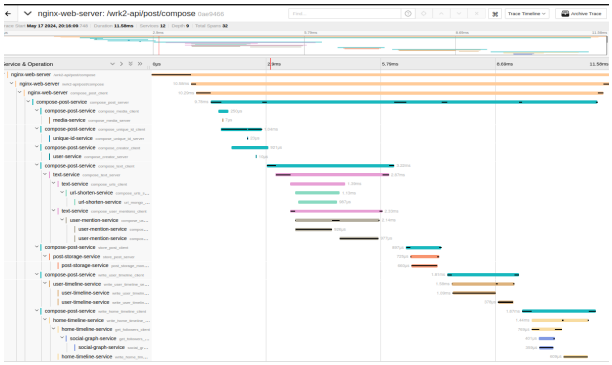

FIGURE 3.4 – trace Jaeger pour composition de Posts.

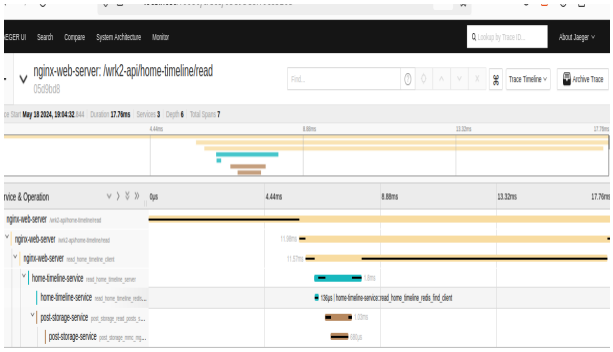

FIGURE 3.5 – trace Jaeger pour lecture des timelines d'Accueil.

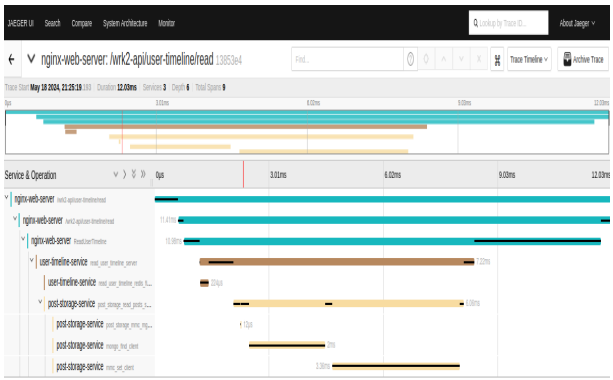

FIGURE 3.6 – trace Jaeger pour lecture des timelines des utilisateurs.

6. **Mesurer le temps de reponse**

   Pour évaluer la performance de l'application de réseau social, nous avons utilisé l'outil wrk2.

   Après l'exécution de chaque test, wrk2 fournit un rapport détaillé incluant les statistiques de latence.

7. **Mesurer l'Utilisation des Ressources Système**

   Pour mesurer l'utilisation du CPU et de la RAM pendant nos tests de charge, nous avons utilisé la bibliothèque psutil en Python. Nous avons développé un script qui surveille ces métriques toutes les secondes durant le test . Le script enregistre l'utilisation du CPU et la mémoire utilisée du processus wrk. Les données collectées sont sauvegardées dans des fichiers JSON pour une analyse ultérieure.

3.5 Tests et résultats

3.5.1 But des tests

le but de ces tests est d'utiliser l'application réseau social du benchmark deathstar est d'évaluer quelques performances des architectures Microservices.

Pour cela, trois expériences ont été réalisées : premièrement, en lançant les tests sur chaque service individuellement, deuxièmement, en exécutant des tests en parallèle sur les trois services, et troisièmement, en comparant les résultats des deux premières expériences.

3.5.2 Expérience 1 : Tests Individuels

L'objectif de cette expérience est de mesurer les performances de chaque microservice de manière isolée. En testant chaque service indépendamment des autres, nous pouvons comprendre leur comportement de base sans l'influence des interactions avec d'autres microservices. Cette approche permet de simuler le comportement d'une application monolithique où tous les composants sont intégrés et fonctionnent ensemble dans un seul environnement.

Pour chaque test, nous lancons le nombre de requêtes pendant 30 minutes, et nous récupérons les différents métriques, nous calculons la moyenne des valeurs récupérées pour tracer les graphes ci-dessous. Les tests incluent la composition de posts, la lecture des

timelines d'accueil, et la lecture des timelines des utilisateurs. Les résultats sont rapportés pour des taux de requêtes variant de 10 à 100 requêtes par seconde (RPS).

3.4.1.1 Le service Compose Post

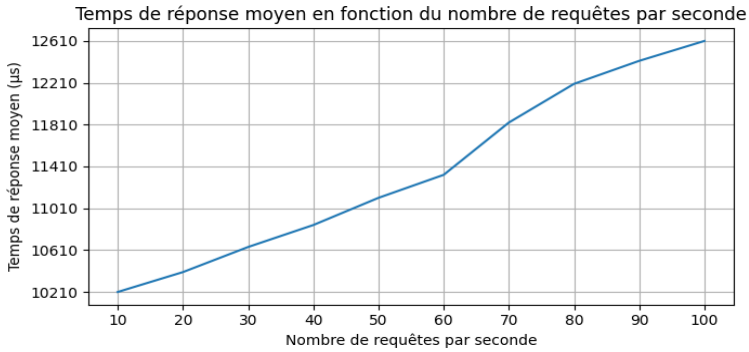

FIGURE 3.7 – Temps de réponse moyen en fonction du nombre de requêtes pour le service compose post.

La figure 3.7 montre la variation du temps de réponse moyen en fonction du nombre de requêtes par seconde. On remarque qu'à 10 requêtes par seconde, le temps de réponse moyen est de 10210 μs. Avec l'augmentation du nombre de requêtes, le temps de réponse moyen augmente progressivement : il est de 10850 μs à 40 requêtes par seconde et de 11330 μs à 60 requêtes par seconde. Jusqu'à ce point, l'augmentation est relativement linéaire, ce qui suggère que le système peut gérer la charge supplémentaire sans dégradation significative des performances. Cependant, au-delà de 60 requêtes par seconde, une inflexion notable est observée dans la pente de la courbe : à 70 requêtes par seconde, le temps de réponse moyen atteint 11830 μs. Cette inflexion indique que le système commence à atteindre ses limites de capacité, entraînant une saturation des ressources. Avec une charge accrue, le temps de réponse moyen continue d'augmenter de manière plus prononcée, atteignant 12610μs à 100 requêtes par seconde. Ce graphique met en évidence que 60 requêtes par seconde constituent un seuil critique au-delà duquel le système commence à se dégrader en termes de temps de réponse.

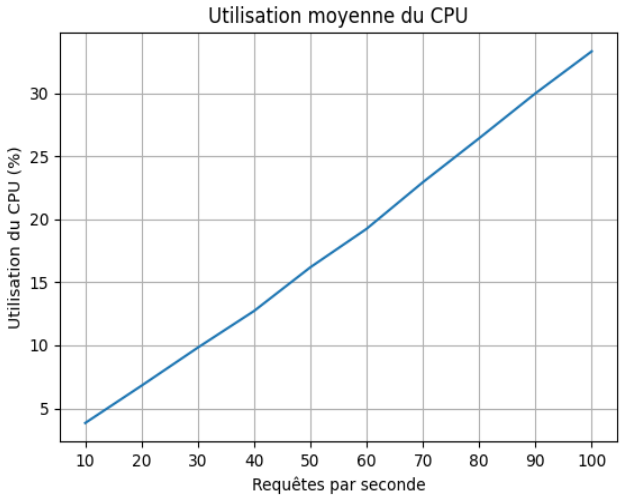

FIGURE 3.8 – Utilisation moyenne du CPU.

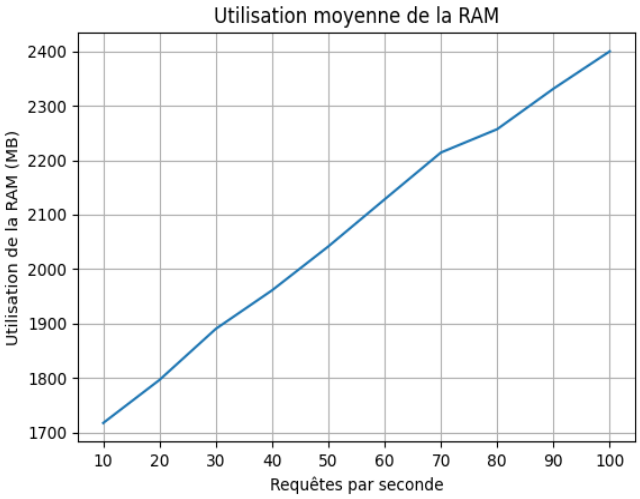

FIGURE 3.9 – Utilisation moyenne de la RAM.

La figure 3.8 illustre l'utilisation moyenne du CPU en fonction du nombre de requêtes par seconde. Il démontre une relation linéaire positive, où l'utilisation du CPU augmente de manière proportionnelle avec le nombre de requêtes. À partir de 10 requêtes par seconde, l'utilisation est de 3.61 %, et elle s'élève à 33.71 % pour 100 requêtes par seconde. Cette progression indique une capacité de l'application à gérer une charge croissante sans atteindre une saturation du CPU.

La figure 3.9 illustre une relation directe entre le nombre de requêtes par seconde et l'utilisation de la RAM. À 10 requêtes par seconde, l'utilisation de la RAM est de 1749,5 MB. Cette observation initiale marque le point de départ pour évaluer l'impact des requêtes supplémentaires sur la consommation de ressources. Lorsque le système gère 50 requêtes par seconde, l'utilisation de la RAM monte à 2020 MB, montrant une augmentation proportionnelle et gérable. À 100 requêtes par seconde, la consommation de la RAM culmine à 2399,9 MB, confirmant ainsi une tendance linéaire d'accroissement en fonction de la charge de travail. Ces données soulignent l'importance d'une gestion optimisée de la mémoire pour maintenir des performances constantes et efficaces face à une demande croissante.

3.4.1.2 le service de lecture des timelines d'accueil

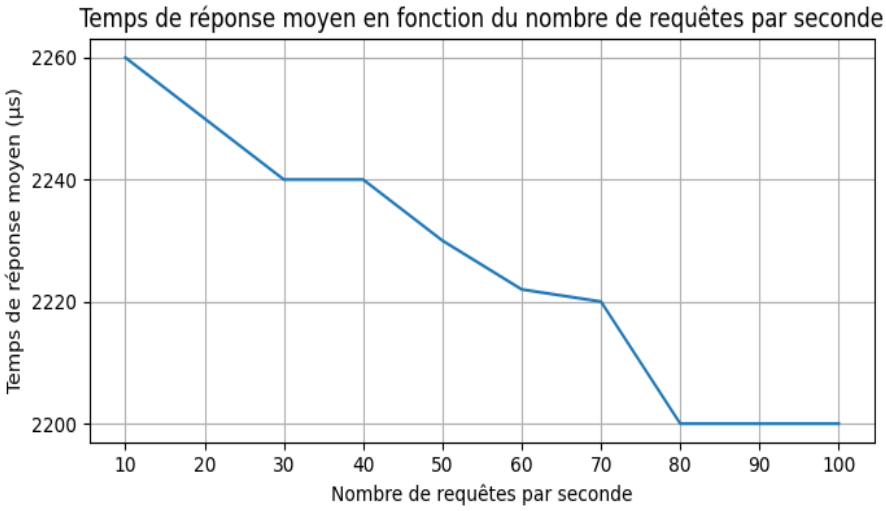

FIGURE 3.10 – Temps de réponse moyen en fonction du nombre de requêtes.

La figure 3.10 montre une tendance générale à la diminution du temps de réponse moyen avec l'augmentation du nombre de requêtes par seconde. En remarques, à 10 requêtes par seconde, le temps de réponse moyen est de 2260 μs. À 50 requêtes par seconde, ce temps diminue légèrement à 2230 μs. Enfin, à 100 requêtes par seconde, le temps de réponse moyen atteint 2200 μs. Cette tendance indique que le système gère efficacement l'augmentation de la charge jusqu'à un certain point, atteignant finalement un état d'équilibre où les augmentations supplémentaires de requêtes n'affectent plus significativement le temps de réponse.

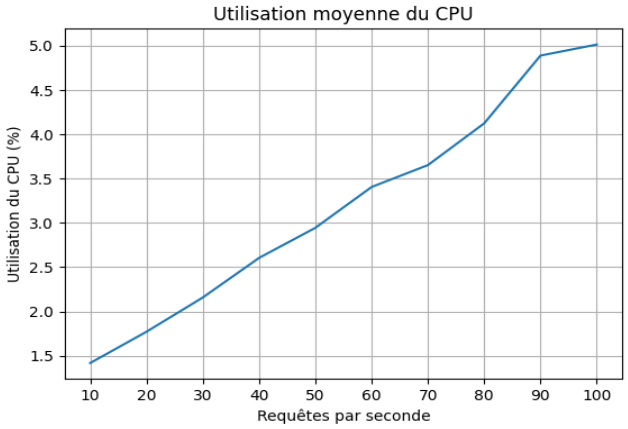

FIGURE 3.11 – Utilisation moyenne du CPU.

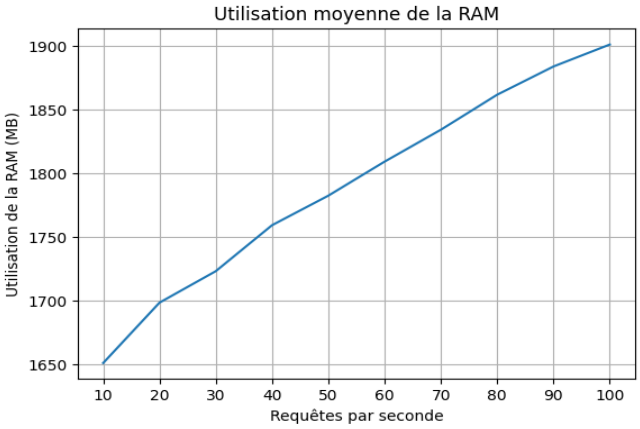

FIGURE 3.12 – Utilisation moyenne de la RAM.

L'utilisation du CPU pour ce service est plus faible comparée à la composition de posts comme le montre la figure 3.11, démarrant à 1.39% pour 10 RPS et atteignant 5.01% à 100 RPS. Cela reflète la moindre complexité de ce service. La légère augmentation de l'utilisation du CPU s'explique par le traitement requis pour gérer l'augmentation du nombre de requêtes.

L'utilisation de la RAM montrée sur la figure 3.12 augmente légèrement, de 1649.9 MB à 1899.8 MB, indiquant une gestion efficace de la mémoire pour cette tâche. L'augmentation négligeable de l'utilisation de la RAM s'explique par la nature moins intensive en mémoire de la lecture des timelines d'accueil.

2.4.1.3 le service de lecture des timelines des utilisateurs

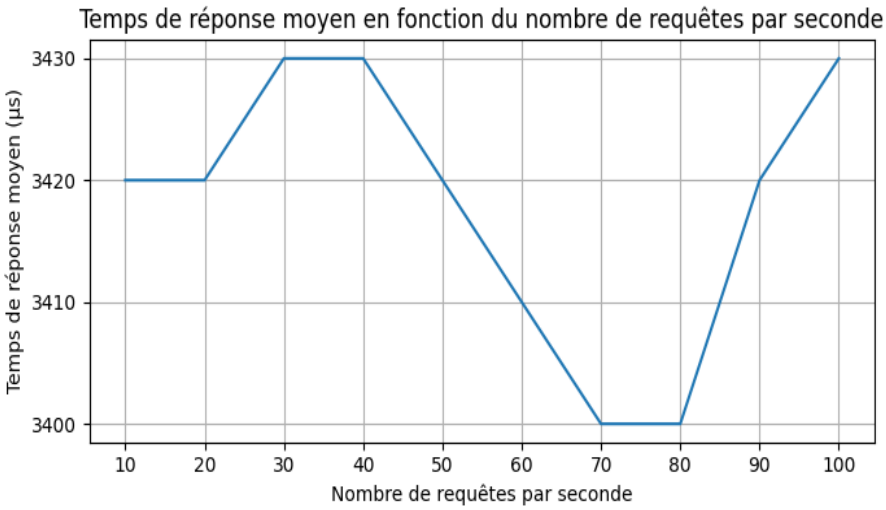

FIGURE 3.13 – Temps de réponse moyen en fonction du nombre de requêtes.

Le graphique du temps de réponse moyen dans la figure 3.13 montre des variations avec l'augmentation du nombre de requêtes par seconde. À 10 et 20 requêtes par seconde, le temps de réponse est de 3420 μs. Il augmente légèrement à 3430 μs pour 30 et 40 requêtes par seconde. À 50 requêtes par seconde, il revient à 3420 μs et continue de diminuer à 3410 μs pour 60 requêtes par seconde, puis à 3400 μs pour 70 et 80 requêtes par seconde. Le temps de réponse remonte à 3420 μs pour 90 requêtes par seconde et atteint 3430 μs à 100 requêtes par seconde. Ces variations indiquent une légère fluctuation des performances du système sous différentes charges.

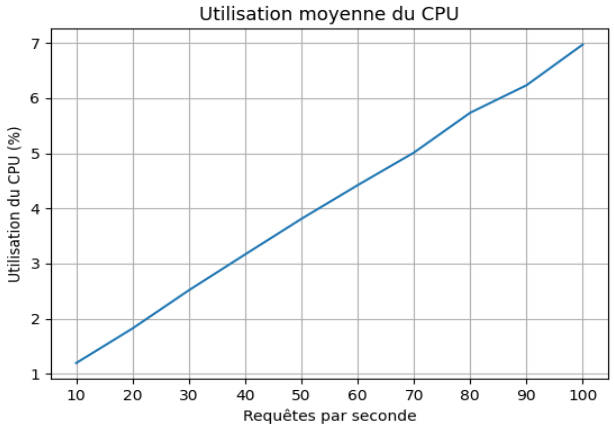

FIGURE 3.14 – Utilisation moyenne du CPU.

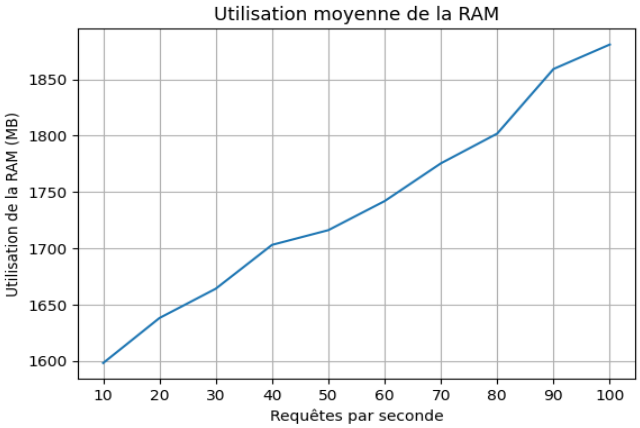

FIGURE 3.15 – Utilisation moyenne de la RAM.

la figure 3.14 montre l'utilisation du CPU augmente progressivement de 1.14% à 6.96% à mesure que le taux de requêtes augmente. Cette tendance est similaire à celle observée pour la lecture des timelines d'accueil, bien que légèrement plus élevée. La légère augmentation de la complexité des requêtes pour les timelines des utilisateurs par rapport à celles des timelines d'accueil est la cause principale.

la figure 3.15 montre l'utilisation de la RAM augmente de 1598.6 MB à 1881 MB, suivant une tendance similaire à celle des autres services testés individuellement. L'augmentation négligeable de l'utilisation de la RAM s'explique par la nature relativement légère des opérations de lecture des timelines des utilisateurs.

En plus, la comparaison des résultats des tests individuels montrent les variations de performance en fonction du nombre de requêtes par seconde pour chaque microservice.

Temps de Réponse :

Pour le temps de réponse, le service de composition de posts présente des résultats notables. À 10 requêtes par seconde (RPS), le temps de réponse moyen est de 10 210 µs, et il augmente progressivement pour atteindre 12 610 µs à 100 RPS. En comparaison, le service de lecture des timelines d'accueil affiche un temps de réponse moyen de 2 260 µs à 10 RPS, diminuant légèrement à 2 200 µs à 100 RPS. Le service de lecture des timelines des utilisateurs montre un temps de réponse moyen de 3 420 µs à 10 RPS, restant relativement stable autour de 3 430 µs à 100 RPS. Les services de lecture ont des

temps de réponse plus courts que le service d'écriture, ce qui indique que les opérations de lecture sont généralement moins complexes et plus rapides à exécuter que les opérations d'écriture.

Utilisation du CPU :

En ce qui concerne l'utilisation du CPU, le service de composition de posts consomme considérablement plus de ressources. L'utilisation du CPU passe de 3,61 % à 33,71 % entre 10 et 100 RPS, montrant une augmentation significative en fonction de la charge de requêtes. En revanche, le service de lecture des timelines d'accueil varie de 1,39 % à 5,01 % entre 10 et 100 RPS, et le service de lecture des timelines des utilisateurs passe de 1,14 % à 6,96 % dans le même intervalle de requêtes. Cette consommation plus élevée de CPU par le service d'écriture s'explique par la complexité des opérations d'insertion et de mise à jour des données, nécessitant plus de cycles de traitement par le processeur.

Utilisation de la RAM :

L'utilisation de la RAM montre également des différences significatives entre les services de lecture et d'écriture. Le service de composition de posts utilise plus de RAM, passant de 1 749,5 MB à 2 399,9 MB entre 10 et 100 RPS. En comparaison, le service de lecture des timelines d'accueil varie de 1 649,9 MB à 1 899,8 MB, et le service de lecture des timelines des utilisateurs de 1 598,6 MB à 1 881 MB sur la même plage de requêtes. L'utilisation plus élevée de la RAM par le service d'écriture est due aux besoins accrus en mémoire pour le cache et les opérations d'insertion de données, tandis que les services de lecture nécessitent moins de mémoire, reflétant des opérations moins intensives en termes de ressources.

3.5.3 Expérience 2 : Tests en Parallèle

L'objectif de cette deuxième expérience est d'évaluer les performances d'une application basée sur des microservices lorsqu'elle est soumise à des charges de travail concurrentes. Pour cela, nous avons exécuté simultanément des requêtes vers les trois services : services de composition de posts, de lecture des timelines d'accueil, et de lecture des timelines des utilisateurs. En mesurant les métriques de performance telles que le temps de réponse, l'utilisation du CPU et de la RAM. Cette expérience permet d'analyser les différences de comportement entre les services de lecture et d'écriture dans un contexte concurrentiel.

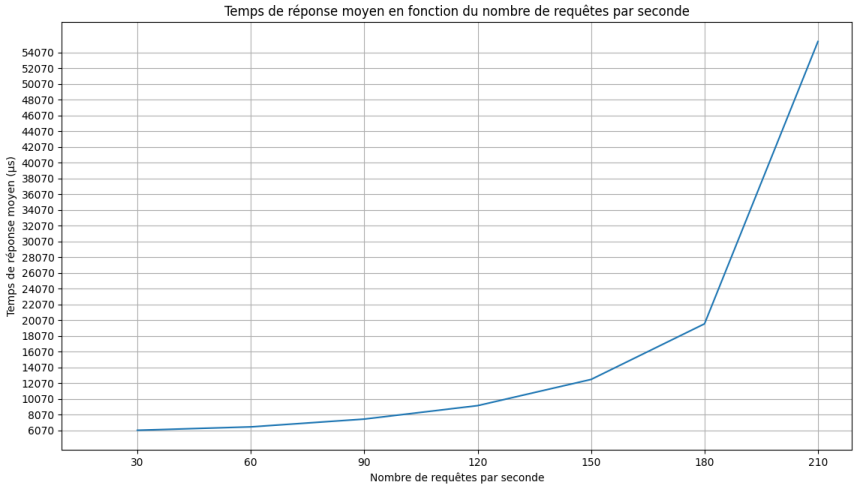

FIGURE 3.16 – Temps de réponse moyen en fonction du nombre de requêtes.

Les résultats des tests présentée dans la figure 3.16 montrent une augmentation progressive du temps de réponse moyen en fonction du nombre de requêtes jusqu'à 210 requêtes. Initialement, avec 30 requêtes, le temps de réponse moyen est de 6070 μs, et il augmente de manière graduelle, atteignant 55480μs pour 210 requêtes. Cette augmentation linéaire indique que le système est capable de gérer la charge croissante jusqu'à ce point. Après avoir terminé avec 210 requêtes, les tests ont été poursuivis avec 240 ,270 , et 300 requêtes : pour 240 requêtes, le temps de réponse est de 4.09 secondes ; pour 270 requêtes, il passe à 33.37 secondes ; et pour 300 requêtes, il atteint 70.37 secondes. Il est important de noter que l'échelle de mesure du temps de réponse change de millisecondes à secondes à partir de 240 requêtes, indiquant une augmentation exponentielle du temps de réponse. Cela suggère que le système atteint un point de saturation, où les ressources

telles que le CPU et la mémoire sont fortement sollicitées, entraînant une dégradation significative des performances.

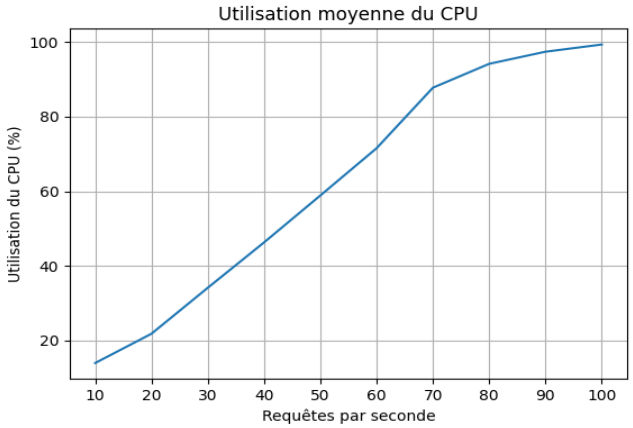

FIGURE 3.17 – Utilisation moyenne du CPU.

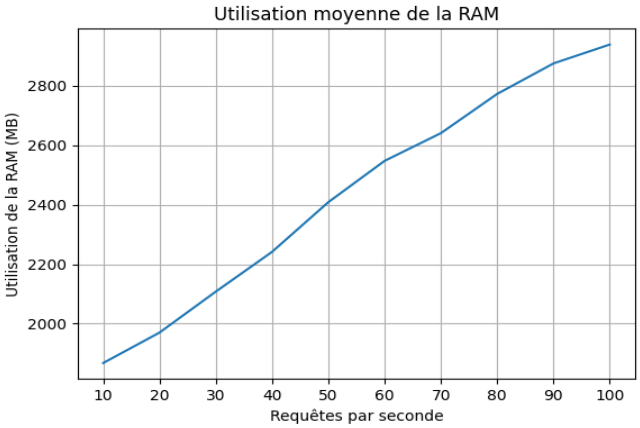

FIGURE 3.18 – Utilisation moyenne de la RAM.

Les tests simultanés ont révélé une évolution cohérente mais progressive de l'utilisation CPU en fonction du nombre croissant de requêtes(figure 3.17). Les taux d'utilisation CPU élevés observés jusqu'à 70 requêtes sont attribuables à la multiplication des opérations résultant de la simultanéité des trois tests, à la concurrence pour les ressources CPU entre les tests et à la nature des opérations effectuées. Cette augmentation souligne l'importance de la gestion des ressources et de l'optimisation des performances pour maintenir des niveaux acceptables d'utilisation CPU dans des environnements. multi-tests, garantissant ainsi des performances stables et évolutives de l'application. Après avoir atteint 70 requêtes, on a poursuivi les tests en augmentant le nombre de requêtes à 80, 90 et 100. Les résultats montrent que pour 80 requêtes, l 'utilisation CPU était de 93,1%, pour 90 requêtes elle est passée à 96,9%, et pour 100 requêtes, elle a atteint 98,9%. Cette augmentation progressive de l'utilisation CPU met en lumière la nécessité d' une gestion efficace des ressources pour maintenir les performances.

L'exécution simultanée des trois tests a conduit une augmentation linéaire de la demande en mémoire avec l'augmentation du nombre de requêtes(figure 3.18). Initialement, à 10 requêtes, l'utilisation de la RAM était mesurée à 1798 MB, et cette charge a progressé régulièrement jusqu'à atteindre 2938 MB à 100 requêtes. Cette augmentation proportionnelle indique que l'exécution des trois tests en même temps a contribué de manière

significative à l'augmentation de la charge en mémoire. Il est crucial de gérer efficacement la mémoire RAM pour maintenir des performances optimales et éviter les goulets d'étranglement, en particulier lors de l'exécution simultanée de multiples tests.

3.5.4 Expérience 3 : Comparaison des Charges

Dans la troisième expérience, nous avons comparé les performances observées lors des deux premières expériences. L'objectif était de comprendre les différences de comportement des microservices lorsqu'ils sont testés individuellement par rapport à des conditions de charge parallèle. Nous avons analysé les métriques de temps de réponse, d'utilisation du CPU et de la RAM pour les services de composition de posts, de lecture des timelines d'accueil, et de lecture des timelines des utilisateurs.

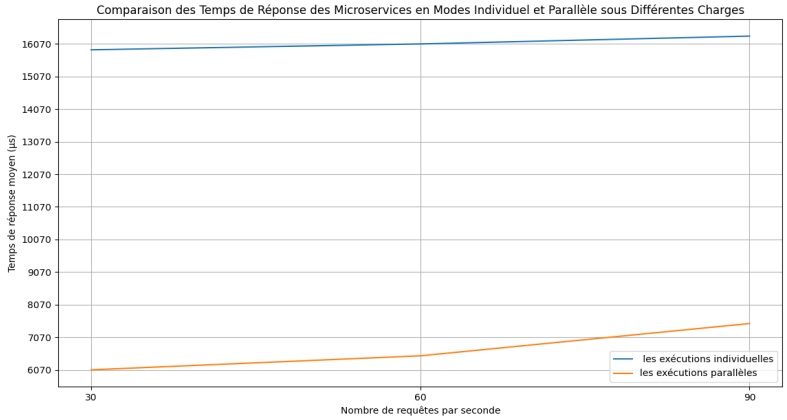

FIGURE 3.19 – Comparaison des temps de réponse des microservices en mode individuel et parallèle sous différentes charges.

À 30 requêtes par seconde (RPS), le temps de réponse moyen en mode individuel (moyenne de 10 requêtes par service pour chaque microservice) était de 15890 μs. En mode parallèle (10 requêtes pour chaque microservice exécuté simultanément), le temps de réponse moyen était de 6070 μs. Cette réduction significative du temps de réponse en mode parallèle indique une meilleure gestion des ressources et une efficacité accrue lorsque les services sont exécutés simultanément à ce niveau de charge. De plus, l'utilisation de la RAM et du CPU en mode parallèle était optimisée, offrant une consommation plus équilibrée par rapport au mode individuel.

À 60 RPS, le temps de réponse moyen en mode individuel (20 requêtes par service pour chaque microservice) était de 16070 μs. En mode parallèle (20 requêtes pour chaque microservice exécuté simultanément), le temps de réponse moyen était de 6500 μs. Cette amélioration en mode parallèle par rapport au mode individuel montre que le système peut encore gérer efficacement les charges accrues lorsqu'elles sont distribuées parmi les services. L'utilisation du CPU et de la RAM était plus efficace en mode parallèle, réduisant la charge sur les ressources et améliorant la performance globale.

À 90 RPS, le temps de réponse moyen en mode individuel (30 requêtes par service pour chaque microservice) était de 16310μs. En mode parallèle (30 requêtes pour chaque microservice exécuté simultanément), le temps de réponse moyen était de 7490 μs. Bien que le temps de réponse augmente en mode parallèle sous une charge plus élevée, il reste inférieur au temps de réponse en mode individuel, indiquant une performance relativement meilleure en parallèle.L'utilisation de la RAM et du CPU, bien que plus élevée en mode parallèle sous cette charge, demeure plus efficace que celle observée en mode individuel.

En résumé, bien que les microservices montrent une performance stable lorsqu'ils sont exécutés individuellement, leur exécution parallèle sous des charges croissantes offre des avantages en termes de réduction du temps de réponse. Cette comparaison met en évidence l'importance de l'optimisation des microservices pour une exécution parallèle efficace, ce qui peut améliorer considérablement les performances et la scalabilité du système dans des environnements à haute charge, , tout en optimisant l'utilisation des ressources telles que le CPU et la RAM.

3.6 Résultats des tests

Aprés une analyse minutieuse des résultats des tests pour différentes expériences, nous avons conclu que les microservices présentent des performances robustes lorsqu'ils sont exécutés individuellement, pouvant gérer jusqu'à 100 requêtes par seconde pour chaque type de requête. Cependant, l'exécution simultanée de plusieurs services sous charge (comme observé dans l'expérience de comparaison des charges) entraîne une augmentation significative du temps de réponse global, soulignant ainsi l'importance de l'optimisation des microservices pour une exécution parallèle efficace. Les tests parallèles ont démontré la capacité du système à supporter jusqu'à 300 requêtes par seconde, mettant en avant les avantages de la scalabilité et de la gestion des charges des microservices. De plus, la comparaison des performances sous différentes charges a révélé une réduction significative du temps de réponse moyen en mode parallèle par rapport au mode individuel, indiquant une gestion plus efficace des ressources et une meilleure efficacité. Ces résultats soulignent l'importance de l'optimisation des microservices pour une exécution parallèle efficace, ce qui peut considérablement améliorer les performances et la scalabilité du système dans des environnements à haute charge.

3.7 Conclusion

Cette étude met en lumière l'impact significatif de l'exécution parallèle des microservices sur l'efficacité et la performance globale du système. Les tests réalisés sur l'application SocialNetwork de DeathStarBench montrent que les architectures de microservices peuvent supporter des charges importantes tout en maintenant des temps de réponse acceptables. En termes d'utilisation du CPU, l'exécution parallèle permet une répartition plus stable de la charge de travail, réduisant les pics de consommation observés en mode individuel. De plus, la gestion de la RAM est optimisée, maintenant une utilisation plus constante et prévisible de la mémoire. Ces résultats soulignent les avantages des architectures de microservices pour la scalabilité et la gestion des charges importantes dans les applications modernes, offrant une solution robuste et efficace pour les environnements de production.

Conclusion générale

Ce mémoire réalise une étude des caractéristiques des benchmarks spécifiques aux microservices. Ce type d'architecture est de plus en plus adopté dans les environnements de développement modernes en raison de sa flexibilité et de sa capacité à faciliter la mise à l'échelle des applications.

Notre étude commence par une revue détaillée des benchmarks existants, visant à identifier et comparer les méthodologies et métriques utilisées pour évaluer les performances des microservices.

La partie expérimentale de notre travail utilise l'application de réseau social du benchmark DeathStarBench. Cette application est composée de trois services : le service d'authentification, le service de profil utilisateur et le service de messagerie. Nous avons effectué plusieurs tests de charge dans divers scénarios et nous avons évalué les réponses du système à travers les métriques suivantes : temps de réponse, utilisation du CPU et consommation de la RAM.

La partie expérimentale est composée de trois expériences. Dans la première expérience, nous avons testé chaque service individuellement ; chaque service simule alors une application monolithique. Cette expérience nous a permis d'étudier les performances des applications monolithiques. Les tests ont montré que chaque service pouvait supporter jusqu'à 100 requêtes par seconde sans dégradation significative des performances, avec des augmentations linéaires de l'utilisation du CPU et de la RAM. Dans la deuxième expérience, nous avons testé les trois services en parallèle, constituant ainsi une application basée sur des microservices. Cette expérience nous a permis d'étudier les performances des applica-

tions microservices. Les résultats ont démontré que le système pouvait gérer une charge combinée de 300 requêtes par seconde, réparties équitablement entre les trois services, tout en maintenant des temps de réponse acceptables. Dans la troisième expérience, nous avons comparé les performances observées lors des deux premières expériences. Cette comparaison a permis de mettre en évidence l'efficacité de l'exécution parallèle des microservices par rapport à leur exécution individuelle. En mode parallèle, les temps de réponse moyens étaient significativement réduits, même à des niveaux élevés de requêtes par seconde, soulignant les avantages de la scalabilité et de l'efficacité des microservices dans des environnements de charge importante.

Comme perspectives futures pour ce travail, nous envisageons d'exploiter le benchmark utilisée pour évaluer d'autres aspects cruciaux tels que la sécurité des microservices et la consistance des données. Par ailleurs, l'utilisation de matériel plus performant est prévue afin de renforcer la fiabilité de nos résultats. Nous projetons également de comparer notre approche avec d'autres benchmarks afin d'enrichir notre analyse et de mieux situer notre étude dans le contexte de la recherche actuelle.

Bibliographie

[1] Guozhi Liu et all. " microservices : architecture, container, and challenges". *IEEE*, page 630, 2020.

[2] docs.oracle. https://docs.oracle.com/fr/solutions/learn-architect-microservice/index.html . Consulté le : [15-02-2024].

[3] AWS Presales Consultant Adam Novotný. https://www.stormit.cloud/blog/api-gateway/. Consulté le : [15-02-2024].

[4] F H Vera-Rivera. "a development process of enterprise applications with microservices". *Journal of Physics*, 1126(1) :3–4, 2018.

[5] Ludovico Funari Andrea Detti and Luca Petrucci. "µbench : An open-source factory of benchmark microservice applications". *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 34(3) :969–970–974–977, 2023.

[6] Yu Gan et al. "an open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems". *ASPLOS '19 : Architectural Support for Programming Languages and Operating Systems*, pages 6–7–8–9, 2019.

[7] N.Schmitt A.Bauer J.Grohmann J.von Kistowski, S.Eismann and S.kounev. "teastore : A micro-service reference application for benchmarking, modeling and resource management research". *Conference Paper*, pages 1–2–4–5–6, 2018.

[8] Online boutique. https://github.com/GoogleCloudPlatform/microservices-demo. Consulté le : [20-03-2024].

[9] istio.io. https://istio.io/latest/docs/examples/bookinfo/. Consulté le : [12-04-2024].

[10] https://github.com/spring-petclinic/spring-petclinic-microservices/. Consulté le : [09-04-2024].

[11] https:https://github.com/delimitrou/DeathStarBench/tree/master/socialNetwork/. Consulté le : [02-05-2024].

[12] https://www.jaegertracing.io/. Consulté le : [02-05-2024].

[13] Pamela Flores Victor Velepucha. " a survey on microservices architecture : Principles, patterns and migration challenges". *IEEE*, 11, 2023.

[14] S Y JING J T ZHAO and L Z Jiang. "management of api gateway based on microservice architecture". *Journal of Physics*, 1087(3) :1, 2018.

[15] learn.microsoft. https://learn.microsoft.com/fr-fr/azure/architecture/microservices/design/orchest . Consulté le : [17-02-2024].

[16] www.redhat. https://www.redhat.com/fr/topics/containers/what-is-container-orchestration . Consulté le : [17-02-2024].

[17] architect.io. https://www.architect.io/blog/2022-06-30/microservices-orchestration-primer/. Consulté le : [20-04-2024].

[18] Chaitanya Rudrabhatla. "comparison of event choreography and orchestration techniques in microservice architecture". *International Journal of Advanced Computer Science and Applications*, 9(8) :18–19–20–21, 2018.

[19] atlassian.com. https://www.atlassian.com/microservices/cloud-computing/advantages-of-microservices/. Consulté le : [20-04-2024].

[20] camunda.com. https://camunda.com/blog/2023/02/benefits-of-microservices-advantages-disadvantages/. Consulté le : [20-04-2024].

[21] cloudacademy.com. https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/. Consulté le : [19-04-2024].

[22] Pravin S. Game Nupura Torvekar. "microservices and it's applications : An overview". *International Journal of Computer Sciences and Engineering*, 7(4) :805, 2019.

[23] Sardar Mudassar Ali Khan. https://dev.to/sardarmudassaralikhan/use-cases-of-microservices-architecture-4lck. Consulté le : [15-02-2024].

[24] Jr. Henry Lucas. "performance evaluation and monitoring ". *ACM Computing Surveys*, 3 :83, 1971.

[25] istio.io. https://istio.io/latest/about/service-mesh/. Consulté le : [12-04-2024].

[26] huaweicloud.com. https://support.huaweicloud.com/intl/en-us/qs-asm/asm$_q s_0$001.*html*. Consulté le : [12-04-2024].

[27] istio.io. https://istio.io/v0.1/docs/samples/bookinfo.html. Consulté le : [12-04-2024].

[28] javaetmoi.com. https://javaetmoi.com/2017/02/spring-framework-petclinic-presentation/. Consulté le : [09-04-2024].

[29] spring petclinic.github. https://spring-petclinic.github.io/. Consulté le : [09-04-2024].

[30] javaetmoi.com. https://javaetmoi.com/2017/02/spring-framework-petclinic-presentation/. Consulté le : [09-04-2024].

[31] github.com. https://github.com/microservices-demo/microservices-demo. Consulté le : [12-04-2024].

[32] github.com. https://github.com/IBM-Cloud/jpetstore-kubernetes. Consulté le : [12-04-2024].

[33] mybatis.org. https://mybatis.org/jpetstore-6/index.html. Consulté le : [12-04-2024].

[34] Ludovico Funari Andrea Detti and Luca Petrucci. "µbench : An open-source factory of benchmark microservice applications". *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 34(3) :970, 2023.

[35] Nicola Dragoni et al. "microservices : yesterday, today, and tomorrow". page 9, 2017.

[36] Sam Newman. *"Building Microservices"*. O'Reilly Media, 1st edition edition, 2015.

[37] Martin Kleppmann. *"Designing Data-Intensive Applications"*. O'Reilly Media, 1st edition edition, 2017.

[38] Michael T. Nygard. *"Release It ! Design and Deploy Production-Ready Software"*. Pragmatic Bookshelf, 1st edition edition, 2007.

[39] Betsy Beyer et al. *"Site Reliability Engineering : How Google Runs Production Systems"*. O'Reilly Media, 1st edition edition, 2016.

[40] Apache JMeter. https://jmeter.apache.org/. Consulté le : [14-04-2024].

[41] gatling.io. https://gatling.io/. Consulté le : [14-04-2024].

[42] locust.io. https://locust.io/. Consulté le : [14-04-2024].

[43] gettaurus.org. https://gettaurus.org/. Consulté le : [15-04-2024].

[44] prometheus.io. https://prometheus.io/. Consulté le : [15-04-2024].

[45] grafana.com. https://grafana.com/. Consulté le : [15-04-2024].

[46] kubernetes.io. https://kubernetes.io/. Consulté le : [15-04-2024].

[47] istio.io. https://istio.io/. Consulté le : [15-04-2024].

[48] STEFAN ZIER. https://devops.com/operational-approach-benchmarking-microservices/. Consulté le : [17-04-2024].

[49] Niranjan Limbachiya. https://dzone.com/articles/performance-testing-of-microservices/. Consulté le : [17-04-2024].

[50] Martin Grambow et al. "benchmarking microservice performance : A pattern-based approach". pages 2–3, 2020.

[51] I. Santamaria R. Colomo-Palacios Larrucea, X. and C. Ebert. " microservices". *IEEE Software*, 35(03) :1, 2018.

[52] martinfowler.com. https://martinfowler.com/tags/microservices.html . Consulté le : [24-05-2024].

[53] learn.microsoft.com. https://learn.microsoft.com/fr-fr/azure/architecture/microservices/. Consulté le : [13-02-2024].

[54] jpetstore. https://jpetstore.aspectran.com/. Consulté le : [12-04-2024].

[55] ibm.com. https://www.ibm.com/blog/modernize-apps-containers kubernetes-ai/. Consulté le : [12-04-2024].