UNIVERSITÉ DU QUÉBEC À MONTRÉAL

CARACTÉRISATION DES DÉPENDANCES LOGICIELLES ARCHITECTURALES INTER-SERVICES : UNE ANALYSE EMPIRIQUE SUR 13 ARCHITECTURES OPEN-SOURCE

RAPPORT DE PROJET DE SYNTHÈSE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAITRISE EN GÉNIE LOGICIEL

PAR

MOHAMED DRAMANE JEAN-PHILIPPE KOÏTA

MAI 2022

REMERCIEMENTS

Ce projet de synthèse n'aurait pu se réaliser sans la contribution de nombreuses personnes qui, chacune de leur manière, m'ont aidé à le mener à terme.

J'aimerais en premier lieu remercier M Sébastien Mosser, professeur au département d'informatique de l'UQAM qui, dès le début a accepté de m'accompagner dans ce projet. Sa disponibilité en tout temps, sa patience, ses vastes connaissances et sa passion pour le domaine m'ont donné un fort intérêt pour ce sujet et ont fait de lui une personne clé dans la réussite de ce projet.

Deuxièmement, j'aimerais remercier ma famille, spécialement mes parents, Mana et Nobala qui ont consenti d'énormes sacrifices pour me payer des études à l'UQAM, qui m'ont toujours démontré un soutien sans failles et m'ont toujours conseillé dans mes choix. Je remercie également mes frères et sœurs, oncles et tantes qui m'ont également été d'un grand soutien.

Ensuite, je tiens également à remercier tous mes professeurs et chargés de cours de l'UQAM qui m'ont apporté beaucoup de connaissances qui ont été nécessaires à la réalisation de ce travail, ont développé en moi l'envie de toujours apprendre plus et de toujours dépasser mes limites.

Finalement, je tiens à remercier un ancien patron, devenu pour moi un frère, qui m'as appris la rigueur dans le travail, le sens du travail bien fait et qui est toujours là quand j'ai besoin de lui, merci Erwin et merci à tous ceux que je n'ai pu citer qui ont contribué de leur manière à ce document.

DÉDICACE

À ma tante Saly, partie trop tôt.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ABRÉVIATIONS, DES SIGLES ET DES ACRONYMES

ACU Asynchronous **C**ommunication **U**tilization

API Application **P**rogramming **I**nterface

CDD Circular **D**ependencies **D**etection

DSS Direct **S**ervice **S**haring

DTU Database **T**ype **U**tilization

REST REprensentational **S**tate **T**ransfer

RPC Remote **P**rocedure **C**all

SDBI Service **D**ata**B**ase **I**nteraction

SIC Service **I**nteraction **C**omponent

SOA Service **O**riented **A**rchitecture

TSS Transitively **S**ervice **S**haring

RÉSUMÉ

Les bonnes pratiques dues au besoin de toujours avoir des logiciels décomposables en des parties indépendantes les unes des autres et réutilisables, ont conduit l'industrie vers les architectures microservices. L'architecture microservices est une approche de système distribué dans laquelle les briques logicielles, appelées microservices, sont des entités indépendantes, capables de fonctionner et d'évoluer de manière isolée.

Le couplage constitue un réel défi dans les architectures microservices, car celui-ci empêche les services d'évoluer seuls, rend les dépendances entre les services plus coûteuses en termes de ressources à cause de leur caractère distribué, ceci entravant la maintenance et à l'évolution du système. Il est alors nécessaire de pouvoir évaluer le couplage et comprendre comment se matérialisent les dépendances entre les services dans les architectures microservices.

Plusieurs travaux académiques ont étudié le couplage dans les différents paradigmes de programmation, cependant l'étude du couplage et la caractérisation des dépendances logicielles entre les services dans les architectures microservices demeurent peu explorées.

Nous concevons alors un cadre d'évaluation du couplage dans les architectures microservices qui étudie le couplage au niveau des bases de données, le couplage au niveau des communications asynchrones entre les services et le couplage au niveau du partage des services. Nous faisons par la suite une évaluation de 13 architectures microservices à partir de ce cadre afin de mieux comprendre les dépendances logicielles entre les services pour mieux les caractériser.

Mots clés : Microservices, architecture logicielle, couplage, dépendances logicielles, évaluation du couplage, communication asynchrone

CHAPITRE 1

INTRODUCTION

1.1 Contexte

Depuis la création du logiciel, le couplage a toujours été un grand défi à relever. Les acteurs de l'industrie du génie logiciel ont toujours essayé de découpler les logiciels en des unités indépendantes les unes des autres afin de mieux les maintenir et d'en réutiliser les différentes parties. Au fil des ans sont apparus plusieurs paradigmes de programmation dont le plus marquant, qui est à la base de la majorité des paradigmes de nos jours, la programmation orientée objet qui est caractérisée par des messages, des objets et l'échange de messages entre ces objets (Rentsch, 1982).

Depuis lors, la majorité des paradigmes architecturaux qui ont émergé utilisent la notion de programmation orientée objet. Ces dernières décennies, on a vu l'apparition de plusieurs paradigmes architecturaux qui proposaient des approches qui comportaient de moins en moins de couplage telle que l'architecture en couches dans laquelle l'application est décomposée en plusieurs couches logicielles, généralement la couche de présentation, la couche de traitement de données et la couche d'accès aux données, l'architecture à composants dans laquelle on a des composants plus modulaires, par exemple un composant pour chaque fonctionnalité, l'architecture à services dans laquelle les unités qui la composent appelées services, ont des responsabilités plus ciblées et des mécanismes de communication plus avancés, puis récemment l'architecture microservices, décrite comme une évolution de l'architecture orientée service(Raj *et al.*, 2018) qui fait aujourd'hui l'objet de notre étude.

Aujourd'hui, nous constatons un réel essor de ce paradigme qui fait l'objet de nombreuses études et dans lequel chaque microservice ne remplit qu'une seule fonctionnalité, gère ses propres données, contient

tout ce dont il a besoin pour fonctionner en son sein et peut être déployé et fonctionner seul par des processus automatisés (Newman, 2015).

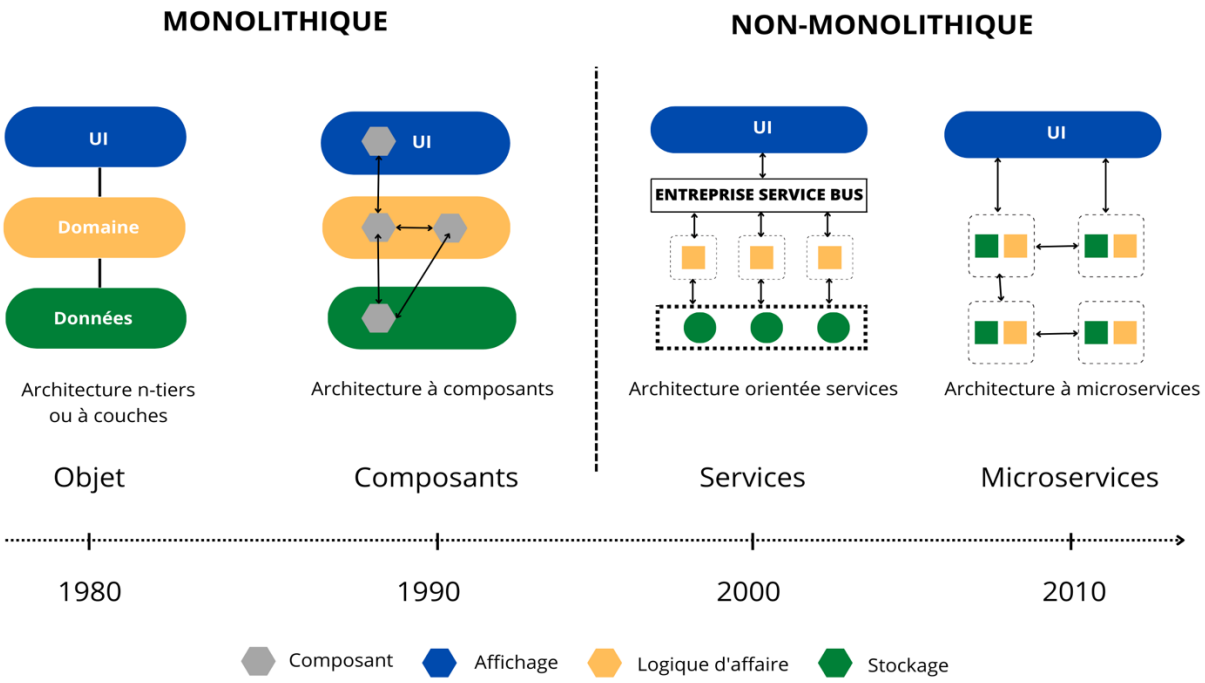

Figure 1.1 - Évolution des paradigmes architecturaux des applications au fil du temps

1.2 Problématique

Le caractère distribué des architectures microservices, bien que permettant d'utiliser un microservice dans plusieurs systèmes différents, rend le couplage plus coûteux en termes de ressources et de temps car dans les systèmes monolithiques, les communications entre les services se résumaient à des appels de méthode d'un objet à un autre, mais dans les systèmes microservices, les services communiquent entre eux par HTTP ou RPC (Wan *et al.*, 2020), ce qui augmente donc les chances d'avoir de la latence et des échecs lors des communications entre les services. Le couplage dans les architectures microservices a donc plusieurs enjeux qui doivent être maîtrisés et c'est cela qui nous a amenés à nous intéresser à la notion de couplage

dans les microservices, à savoir comment se matérialise le couplage dans les architectures microservices, comment celui-ci est mesuré et comment on peut évaluer des architectures de référence en termes de couplage afin de mitiger les impacts que peut avoir le couplage dans les architectures microservices.

1.3 Objectifs du projet

Par nos travaux, nous souhaitons proposer un cadre d'évaluation des architectures microservices sur la base des dépendances qui existent entre leurs différents microservices. Pour ce faire nous allons d'abord analyser manuellement 13 architectures microservices *open source* de référence, faire une revue de la littérature dans le but de chercher des mesures pertinentes qui permettent de quantifier le couplage dans ces architectures et évaluer ces architectures avec les mesures sélectionnées.

1.4 Organisation du rapport

Le présent rapport de projet est structuré comme suit, le chapitre 2 décrit les concepts préliminaires c'est-à- dire une présentation des architectures microservices, de leurs caractéristiques, des techniques et outils nécessaires pour leur mise en œuvre ainsi que des concepts qu'ils ont apportés afin d'effectuer une mise en contexte. Le chapitre 3 est consacré à l'état de l'art des attributs de qualité des architectures microservices, du couplage dans les architectures microservices et des patrons et antipatrons dans les architectures microservices nécessaires à l'entame de notre projet. Le chapitre 4 nous décrit les différentes mesures qui serviront à évaluer le couplage dans les systèmes à l'étude et le chapitre 5 quant à lui décrit la mise en œuvre de notre projet commençant par le choix des architectures de références jusqu'à la modélisation des données et la présentation de l'outil qui soutiendra notre étude. Le chapitre 6 nous présente les résultats obtenus à la suite de notre étude et leur analyse avant de passer à la conclusion.

CHAPITRE 2

CONCEPTS PRÉLIMINAIRES

Dans l'objectif de clarifier tous les concepts qui seront abordés et de faciliter la compréhension de ce rapport, nous allons dans ce chapitre, aborder les concepts fondamentaux liés à notre étude. Nous commencerons par définir les microservices, discuter de leurs principaux avantages et inconvénients, ensuite décrirons les différentes caractéristiques principales des microservices et par la suite énoncerons les différents outils nécessaires pour la mise en œuvre d'architectures microservices.

2.1 Les architectures microservices

2.1.1 Définition des architectures microservices

L'architecture microservices est une approche de système distribué selon laquelle on développe une application en une suite de petits services élastiques appelés microservices, composables, chacun ayant en lui-même tout ce dont il a besoin pour fonctionner correctement, ayant leur propre cycle de vie et qui communiquent par des protocoles légers, généralement des *API REST* en *HTTP*. (Newman, 2015).

Cette approche n'a pas été inventée, mais a progressivement émergé à la suite de bonnes pratiques dans le domaine et dans la recherche constante de pouvoir construire des applications en des entités toujours plus petites, maintenables, indépendantes et réutilisables.

Certains chercheurs du domaine du génie logiciel comme Raj et al. définissent aussi les microservices comme une amélioration de l'architecture orientée service ou *SOA* grâce à laquelle il est possible de

développer des exigences métier avec des services faiblement couplés, auto-déployables et évolutifs (Raj *et al.*, 2018).

Les architectures microservices doivent aussi leur succès aux grandes compagnies telles que Netflix ou Amazon, qui dès les débuts de leur apparition ont fait évoluer leurs applications vers les microservices (Di Francesco *et al.*, 2017).

2.1.2 Principaux avantages des architectures microservices

Que ce soit pour la conception d'une nouvelle application ou la migration d'une application monolithique, les architectures microservices offrent de nombreux avantages parmi lesquels :

- Diversité de technologies :

Les architectures microservices permettent d'utiliser pour chaque microservice différent, un langage de programmation et une technologie de stockage de données différentes. Ce qui donne la liberté aux concepteurs de l'application de choisir, en fonction du microservice, le langage le mieux approprié aux besoins, pareil pour le choix de la technologie de stockage (Fowler, 2015).

- Déploiement indépendant :

L'autonomie des microservices diminue l'incidence qu'a un nombre élevé de déploiements sur le fonctionnement global de l'application, car les déploiements sont destinés à un service à la fois contrairement aux systèmes monolithiques où l'entièreté de l'application est de nouveau déployée à chaque modification de l'application (Fowler, 2015).

- Frontières solides entre les différents composants :

Bien que l'architecture microservices ne soit pas la première à proposer l'implémentation d'un système distribué, elle est pour le moment, sa meilleure implémentation avec des microservices capables de fonctionner seuls, dans plusieurs environnements simultanément. La composition de l'application en plusieurs unités indépendantes les unes des autres donne la possibilité de la faire évoluer et modifier sans avoir d'incidences sur le reste de l'application. Cela facilite également la prise en main de l'application lorsqu'elle doit passer entre les mains d'une nouvelle équipe de développement (Fowler, 2015).

2.1.3 Principaux inconvénients des architectures microservices

Bien qu'offrant de nombreux avantages, il faut, avant de choisir de migrer une application existante aux microservices ou de choisir cette approche pour une future application à concevoir, prendre en compte plusieurs points importants :

- La complexité du découpage en microservices :

L'un des principaux défis du modèle d'architecture de microservices est de déterminer le niveau de granularité correct pour les composants de service. Si les composants de service n'ont pas la bonne taille, notre architecture risque d'apporter plus d'inconvénients que d'avantages (Richards, 2015). Pour cela, le *Domain Driven Design ou DDD (Evans, 2003),* conception pilotée par domaine qui fournit les moyens d'isoler les concepts de domaine et d'identifier les relations entre eux, est approprié pour la conception des microservices qui sont articulés autour des besoins d'affaires (Rademacher *et al.*, 2018).

- La complexité d'opération et de maintenance :

Compte tenu du nombre de microservices dans l'application, la complexité de l'infrastructure, le caractère distribué du système et les différents langages de programmation qui peuvent être présents, la maintenance et l'opération d'une application à microservices impliquent plus de complexité, il faut donc des équipes matures pour s'en occuper (Fowler, 2015).

2.2 Les caractéristiques des architectures microservices

Bien que les microservices n'ont pas de caractéristiques formellement définies, Lewis et Fowler ont défini des concepts communs, présents dans la plupart des architectures à microservices, nécessaires pour pouvoir tirer le maximum de bénéfices des architectures microservices et rester dans la philosophie pour laquelle elle a émergé (Lewis et Fowler, 2014). Ce sont :

- La composition par des services

L'architecture microservice doit être composée d'unités indépendantes et surtout remplaçables afin de faciliter son évolution et que la modification d'une partie de l'application n'affecte pas les autres.

- La conception autour des besoins du métier

Un microservice est conçu pour effectuer une tâche précise et doit avoir des frontières bien définies. Cela facilite les évolutions dans le système et le choix de technologies de programmation et de stockage pour des problématiques précises.

- La notion de produit plutôt que projet

Les microservices sont voués à une constante évolution au cours de leur cycle de vie, ainsi ils sont considérés comme un produit en non comme un projet dont la fin de l'évolution est bien définie. Ceci a pour objectif de toujours fournir la valeur métier désirée, faire toujours mieux ce pour quoi il a été conçu. Ainsi il est conseillé de confier le cycle de vie d'un microservice à la même équipe qui saura le faire évoluer.

- *"Smart endpoints and dumb pipes"*

Dans la conception du microservice, il faut se concentrer sur l'implémentation de la logique d'affaires plutôt que se focaliser sur les moyens de communication qui doivent rester très simples.

- La gouvernance décentralisée

Le caractère découplé de l'architecture microservices permet à celle-ci d'utiliser pour chaque microservice, un langage de programmation différent donc mieux adapté aux tâches que celui-ci aura à effectuer. Ainsi la gestion de l'architecture sera décentralisée, car chaque service aura différents besoins.

- La gestion des données décentralisée

Comme pour le choix de différents langages pour chaque service, il est également recommandé de tout d'abord choisir un moyen de stockage différent pour chaque service dans le but d'éviter le couplage, il sera donc possible d'utiliser plusieurs types de stockage différents compte tenu des besoins de chaque microservice.

- Automatisation de l’infrastructure

Avec le nombre de microservices présents dans une architecture et le nombre de déploiements qui pourront être effectués, il est alors nécessaire d'automatiser tout ce qui peut l’être dans l’infrastructure, de l’allocation d’espace dans le serveur aux déploiements et tests.

- Conçu pour échouer

Une architecture microservice doit être conçue de sorte à tolérer les pannes, pour éviter que la panne d’un microservice n’occasionne une succession de pannes, due aux autres microservices qui pourraient l’utiliser, ce qui pourrait causer un arrêt total du système. Les autres microservices doivent plutôt gérer la situation aussi gracieusement que possible, par des messages d’indisponibilité du service par exemple. Le système doit être également capable de relancer automatiquement des microservices lorsque nécessaire.

- Conception évolutive

Les microservices doivent être conçus de sorte à faciliter les changements fréquents, rapides puis à toujours garder le contrôle sur ces changements. Ceci permet une évolution rapide du système, car il sera possible d'ajouter rapidement des fonctionnalités sans apporter des modifications dans le reste du système.

2.3 Technologies, outils et concepts nécessaires à la mise en œuvre des architectures microservices

L’architecture microservices est un paradigme complexe à mettre en place. Pour la plupart des critères que doivent rencontrer les microservices, cités précédemment, l’industrie a eu besoin de mettre en place ou d’utiliser différents outils et technologies afin de rendre le développement et les opérations des microservices plus efficients. Pour créer un environnement propice au développement et à l’évolution des microservices, il est nécessaire de disposer d’un certain nombre d’outils et de technologies qu’on va énumérer dans ce chapitre.

2.3.1 Intégration et déploiement continu

L'évolutivité est capitale dans les architectures microservices, c'est-à-dire la capacité pour l'application d'évoluer rapidement afin de toujours garder le maximum de valeur métier.

Avec le rythme d'évolution des microservices et leur grand nombre dans les applications, le déploiement et l'intégration de nouvelles fonctionnalités deviennent un véritable défi, car toutes ces actions doivent se faire sans interruption du fonctionnement de l'application. Il devient alors nécessaire d'automatiser toutes ces actions, ce qui conduit à l'utilisation du déploiement et de l'intégration continue.

L'intégration continue est une pratique de développement de logiciels où les membres d'une équipe intègrent fréquemment leur travail au dépôt principal (Fowler, 2006). Elle est composée d'actions systématiques telles que des tests de régressions pour s'assurer que l'ajout fonctionne en harmonie avec le code déjà présent.

La livraison continue est une discipline de développement logiciel dans laquelle le logiciel est construit de telle sorte qu'il puisse être mis en production à tout moment. Il peut être mis en œuvre par l'utilisation de tunnels de déploiement dans lesquels les incréments seront mis ensemble et testés sur des environnements proches de l'environnement de production pour s'assurer de son fonctionnement optimal (Fowler, 2013).

Le déploiement continu nécessite l'intégration continue, car ils reposent sur le même principe sous-jacent de fractionnement du travail en petits incréments. Toutes les modifications de code sont alors automatiquement créées, testées et préparées pour la version de production (Railić *et al.*, 2021).

Parmi les outils d'intégration et de déploiement continus, on peut citer Jenkins[1] qui est l'un des plus populaires (Maayan, 2021).

2.3.2 Conteneurs

Un système conçu avec le paradigme des microservices doit permettre un passage à l'échelle à la demande en cas d'augmentation du trafic. Le passage à l'échelle signifie l'augmentation des instances des

[1] https://www.jenkins.io/

microservices présents dans l'application. Avec les systèmes d'infrastructures traditionnels où chaque instance de service est directement installée sur une seule machine virtuelle, le passage à l'échelle est très coûteux en termes de ressources et de temps de mise en œuvre des instances supplémentaires.

La solution qui paraît la mieux adaptée dans ce cas de figure est la virtualisation par conteneurs.

La virtualisation basée sur les conteneurs est plus légère et économe en ressource que la virtualisation basée sur les machines virtuelles classiques (Al-Dhuraibi *et al.*, 2018). Les conteneurs isolent les processus au niveau central du système d'exploitation et utilisent le système d'exploitation de la machine hôte, ce qui permet de gérer efficacement les ressources et d'avoir plus d'instances sur le même serveur (Al-Dhuraibi *et al.*, 2018) donc de faciliter le passage à l'échelle.

Docker[2] est l'outil de virtualisation par conteneurs le plus populaire (Heusser, 2022).

2.3.3 Découverte de services

Le passage à l'échelle des microservices nécessite la création de plusieurs instances des microservices dans des conteneurs différents. Dans un contexte où les localisations des services peuvent changer dynamiquement, un service doit acquérir à tout moment l'emplacement d'autres services afin de communiquer avec eux. Ce problème peut être résolu par un processus appelé découverte de service qui est essentiel lorsque les microservices sont sur des conteneurs (Long *et al.*, 2017).

La découverte de service consiste à trouver l'emplacement des instances correctes qui fournissent les services requis (Long *et al.*, 2017) et peut se faire côté serveur où le routeur qui joue le rôle d'un équilibreur de charge, redirige directement la requête d'un service sur un service disponible ou côté client où il y a présence d'un registre de services sur lequel le client va effectuer des requêtes pour connaître la localisation de toutes les instances des services. On peut citer Eureka[3] et ZooKeeper[4] parmi les outils de découverte de services les plus utilisés (g2.com, 2022).

[2] https://www.docker.com/

[3] https://github.com/Netflix/eureka

[4] https://cloud.spring.io/spring-cloud-zookeeper/

2.3.4 Descripteurs d'interfaces de services

Une application à base de microservices est composée de plusieurs microservices commis à différents usages qui comportent différentes interfaces. Maintenir la documentation des services synchronisée et à jour avec la mise en œuvre réelle des services est une tâche difficile et peu automatisée (Casas *et al.*, 2021).

Afin de combler cette lacune, différents outils permettant de spécifier et décrire les interfaces sont apparus tels que les descripteurs d'interfaces de services afin d'aider les développeurs à mieux comprendre les capacités du service avec lequel ils travaillent sans accès préalable au code de mise en œuvre du service ni d'inspection du réseau (Casas *et al.*, 2021).

Ils s'installent généralement sous forme de bibliothèques dans le code de l'application et suivent généralement deux approches. La première, ***code first*** génère de la documentation à partir du code source de l'application et la deuxième, ***design first*** dans laquelle la documentation est écrite en avance et utilisée par le descripteur de services pour générer le code source.

Le descripteur d'interface de services le plus populaire et utilisé est **SWAGGER** qui est désormais considéré comme un standard de facto (Casas *et al.*, 2021).

2.3.5 Disjoncteur de services

Dans les architectures microservices, les microservices communiquent entre eux par des appels réseaux pour s'échanger des informations. Les requêtes effectuées peuvent échouer ou se bloquer sans réponse jusqu'à ce qu'un certain délai d'expiration soit atteint, et lorsqu'un service qui reçoit de nombreuses requêtes est dysfonctionnel, cela peut entraîner des pannes en cascade et ralentir le système (Fowler, 2014).

Pour pallier ce problème, il faut implémenter un disjoncteur de services qui empêche les défaillances en cascade, où la défaillance d'un seul microservice peut entraîner l'échec de l'application globale (Jagadeesan *et al.*, 2020). Ce disjoncteur va permettre de router les requêtes des services entre eux. À l'image d'un disjoncteur électrique, il sera à l'état fermé lorsque toutes les requêtes passent avec succès. Dans le cas d'une requête échouée, il va permettre au service qui émet la requête de réessayer jusqu'à la valeur seuil fixée avant de passer à l'état ouvert et marquer le service indisponible. Après un certain temps dans l'état ouvert, il passe à un état semi-ouvert afin de laisser passer quelques requêtes pour tester l'état

de fonctionnement du service; si les requêtes passent avec succès, il passe à l'état fermé, sinon reste ouvert. Istio[5] en est un exemple très connu qui est recommandé à l'installation de Docker.

2.3.6 DevOps

Les architectures microservices sont composées de microservices généralement écrits avec plusieurs langages de programmation ayant des besoins différents qui sont dans un environnement au rythme d'évolution très rapide. La tâche devient alors plus difficile pour les équipes des opérations gérant les infrastructures qui devront désormais gérer une multitude de serveurs avec des configurations différentes, les configurations d'intégration et de déploiement continu ou la distribution du système.

C'est ainsi qu'est né le besoin d'intégrer la philosophie DevOps qui permet de réduire l'impact des défis liés au développement et aux opérations des architectures microservices (Knoche et Hasselbring, 2019).

DevOps est une approche qui favorise la collaboration du personnel de développement et des opérations afin de développer des logiciels de qualité de manière continue (Kim *et al.*, 2016). Les équipes chargées du développement d'un microservice sont alors également chargées de s'occuper de tâches relatives à l'infrastructure de ce service en particulier telles que le déploiement et la surveillance du trafic.

Dans un environnement avec des conteneurs, l'équipe de développement aura la responsabilité de former des paquets logiciels composés du code source et de tout le nécessaire de l'infrastructure dans des fichiers exécutables appelés images qu'elle va déployer sur les différents environnements.

2.3.7 Infonuagique

L'utilisation de conteneurs dans les architectures microservices et le besoin d'avoir des ressources à la demande pour faciliter le passage à l'échelle des microservices nécessite l'utilisation de l'infonuagique dans les architectures microservices qui sont considérés comme cloud-native (Microsoft, 2022) qui signifie conçu pour fonctionner avec l'infonuagique.

[5] https://istio.io

L’infonuagique est la fourniture de services informatiques tels que les serveurs, le stockage, l’infrastructure, les logiciels permettant d’offrir aux clients des ressources flexibles en fonction des besoins et facturés à la consommation (Microsoft, 2022)

L’infonuagique et l’utilisation des conteneurs permettent aux organisations de réduire les coûts de développement des logiciels, d’augmenter le niveau de fiabilité des microservices et de créer puis d'exécuter des applications évolutives dans des environnements dynamiques modernes (Li, 2020).

Amazon Web Services ou AWS[6] est un exemple assez populaire de plateforme infonuagique rassemblant la majorité des outils cités plus haut afin de mettre en place des systèmes microservices *cloud native*.

2.4 Conclusion

Dans ce chapitre nous avons défini les termes et concepts relatifs aux architectures microservices qui sont nécessaires à la compréhension de notre rapport de projet. Nous avons commencé par définir les architectures microservices, ensuite donné leurs caractéristiques, enfin donné les technologies et outils nécessaires à la mise en œuvre des microservices sans oublier les concepts qu’ils ont apportés. Nous présenterons dans le prochain chapitre l’état de l’art sur les attributs de qualité des architectures microservices, le couplage dans les architectures microservices et les patrons et antipatrons dans les architectures microservices.

[6] https://aws.amazon.com/

CHAPITRE 3

ÉTAT DE L'ART

Le couplage a toujours suscité beaucoup d'intérêt en général dans l'industrie du logiciel. Plusieurs études ont été identifiées dans lesquelles il a été défini les différents types de couplages, d'autres dans lesquelles on a tenté des approches de détection. Mais dans cet état de l'art, nous nous intéresserons particulièrement au couplage dans les architectures microservices, qui lui aussi suscite beaucoup d'intérêt.

En effet, ce qui porte l'intérêt sur le couplage dans les architectures microservices, c'est que là où dans les logiciels monolithiques les communications entre les services s'effectuent par des appels de méthodes, dans les architectures microservices, les communications entre les services s'effectuent via des protocoles comme HTTP et RPC (Wan *et al.*, 2020) ce qui peut augmenter le temps de latence et les chances d'échecs des communications, d'où la nécessité d'étudier ce couplage.

Ensuite, l'émergence des architectures microservices de ces derniers temps a poussé les chercheurs à s'intéresser aux attributs de qualité des microservices, car leur compréhension actuelle dans l'architecture des microservices était déficiente et incomplète (Li *et al.*, 2021). Il y a eu alors plusieurs études en ce sens telle que celle de Li (Li *et al.*, 2021) ayant pour objectif de construire un corpus de connaissances sur les attributs de qualité en architecture microservices à travers une revue systématique de la littérature.

Enfin, l'émergence des architectures microservices a également conduit les chercheurs à s'intéresser aux patrons et antipatrons dans ces architectures, notamment par des études dans lesquels ils ont été définis et des approches de détection d'antipatrons dans les architectures.

Dans les sections qui suivent, nous aborderons en détail les différents travaux qui ont été conduits concernant les attributs de qualité des architectures microservices, le couplage dans les architectures microservices ainsi que les patrons et les antipatrons dans les architectures microservices.

3.1 Attributs de qualité des architectures microservices

Avec le succès qu'ont rencontré les architectures microservices ces dernières années, les migrations des systèmes monolithiques vers les microservices des géants de la technologie, les chercheurs ont fait plusieurs études ayant pour objectif de rechercher et définir les attributs de qualité des microservices afin de mieux les évaluer. C'est ainsi que Li et al. dans leur revue de littérature portant sur 72 études, avec pour objectif de faire l'état de l'art des attributs de qualité des architectures microservices ont identifié les six attributs de qualité les plus réputés dans les architectures à microservices qui sont le passage à l'échelle (*scalability*) qui est la capacité à augmenter les ressources pour gérer une quantité de demandes pouvant atteindre des sommets, la performance (*performance*) qui est la capacité à répondre efficacement aux demandes dans les temps impartis, la disponibilité (*availability*) qui est la capacité de réduire considérablement le nombre de pannes d'où le temps d'indisponibilité du microservice, la capacité de surveillance (*monitorability*) qui est la capacité à être surveillée pendant son exécution, la sécurité (*security*) est la capacité du système à protéger les données et les informations contre tout accès non autorisé et permettre l'accès aux ayants-droit et la testabilité (*testability*) qui est la capacité à démontrer ses défauts lors de l'exécution des tests (Li *et al.*, 2021). Les auteurs nous donnent également pour chaque attribut de qualité, des outils à utiliser pour les mettre en œuvre dans les architectures. Pour conclure, les auteurs indiquent que mettre trop l'emphase sur l'un des attributs de qualité peut impliquer à faire des compromis avec les autres. Mettre en œuvre tous les attributs de qualité simultanément est une chose complexe, voire impossible. Il est donc toujours important de faire des choix en fonction de la nature de l'application à concevoir.

3.2 Le couplage dans les architectures microservices

Bien avant l'avènement de l'architecture à base de microservices, le couplage a toujours été un grand défi dans l'industrie du génie logiciel. Déjà en 2001, dans leur étude, Allen et al. s'intéressent aux mesures du couplage dans les systèmes modulaires, qui causaient alors des problèmes de performance de celles-ci (Allen *et al.*, 2001).

Aujourd'hui avec les microservices, le couplage demeure un défi c'est ainsi que Ma et al. nous proposent un outil nommé GMAT (*Graph-based Microservice Analysis and Testing*) chargé d'analyser et de visualiser les relations de dépendances entre les microservices (Ma *et al.*, 2018). Pour recueillir les données des services nécessaires à son bon fonctionnement telles que les *endpoints* et les appels de service, GMAT se sert du mécanisme de réflexion chez les langages compatibles ou peut se coupler à SWAGGER pour

récupérer les données dont il a besoin. Une fois les informations recueillies, GMAT s'en sert pour établir avec Neo4J[7], une base de données orientée graphe, un graphe avec tous les services et les relations entre elles. Avec le graphe ainsi modélisé avec Neo4j, il est désormais possible d'effectuer plusieurs opérations telles que retrouver des chaînes d'invocation de services afin de détecter des dépendances cycliques. Comme mentionné par les auteurs, le premier et le plus important objectif du GMAT est d'extraire les services et les invocations de services afin d'obtenir un graphe qui peut permettre d'observer visuellement les relations entre les microservices. Les utilisateurs pourront, en fonction de leurs besoins, écrire des requêtes Neo4j pour avoir des mesures plus spécifiques.

Apolinário et al. quant à eux, nous proposent l'outil SYMBIOTE pour surveiller l'évolution du couplage au cours de l'évolution du système dans un système à base de microservices (Apolinário *et al.*, 2021). SYMBIOTE collecte les informations sur l'architecture durant l'exécution du programme idéalement pendant que les tests sont exécutés pour recueillir le maximum d'appels entre les services, en utilisant un *service mesh* ou un détecteur de paquets *HTTP* dans un cluster *kubernetes*[8] qui est aussi capable de détecter les communications asynchrones entre les services. Avec les données recueillies, un graphe est généré avec le langage DOT, sur lequel seront effectuées les différentes mesures. SYMBIOTE collecte quatre mesures, deux pour chaque service qui sont AIS(*Absolute Importance of the Service*) l'importance absolue du service qui est le nombre de services qui invoquent au moins une fois une opération du service mesuré et ADS (*Absolute Dependence of the Service*) qui est le nombre de services que le service mesuré invoque au moins une fois, puis deux mesures sur l'architecture complète qui sont le SCF(*Service Coupling Factor*)facteur de couplage de service qui est la mesure de la densité de la connectivité du graphe et le ADCS (*Average number of Directly Connected Services*) moyenne de services directement connectés qui est la moyenne des ADS dans l'architecture.

SYMBIOTE est un outil d'analyse dynamique du code et est conçu pour être déployé dans la phase de production ou de *staging* du logiciel, conserve et compare toutes les mesures prises au cours du temps pour alerter les utilisateurs lorsque des changements significatifs seront détectés. SYMBIOTE dispose d'un tableau de bord qui affiche un graphe de l'évolution des différentes mesures, le dernier graphe de dépendances et les mesures de chaque service.

[7] https://neo4j.com

[8] https://kubernetes.io/fr/

3.3 Les patrons et antipatrons dans les architectures microservices

La forte implantation des architectures microservices a poussé les chercheurs du génie logiciel à mener plusieurs études sur les bonnes et les mauvaises pratiques architecturales, car comme tous les styles architecturaux, l'utilisation amène toujours de nouvelles problématiques. C'est ainsi que Akbulut et al. dans leurs travaux destinés à analyser la performance de plusieurs architectures microservices, ont identifié le patron de conception passerelle d'API (*API gateway design pattern*) qui consiste en une passerelle à partir de laquelle tous les microservices de l'application sont accessibles, constituant un unique point d'entrée aux microservices se chargeant d'appeler le service approprié et peut jouer le rôle de disjoncteur en cas de défaillance du service, le patron chaîne de responsabilité (*chain responsibility design pattern*) qui consiste en un ensemble de services conçus pour travailler ensemble pour exécuter une requête, les services sont liés les uns aux autres séquentiellement de telle sorte que la sortie d'un service constitue directement l'entrée du service suivant et le patron de conception avec messagerie asynchrone (*asynchronous messaging design pattern*) dans lequel les microservices s'envoient des messages entre eux pour effectuer des actions et que les réponses qu'ils doivent recevoir ne doivent nécessairement venir immédiatement, un agent de message (*message broker*) doit être introduit dans l'architecture afin de traiter les messages et les diriger vers les bons services, ce patron permet de traiter graduellement un grand nombre d'échanges entre les services et la capacité de traitement de l'agent de message n'est limitée que par l'espace mémoire qui lui est attribué (Akbulut *et al.*, 2019). En conclusion de cette étude, les auteurs affirment qu'il n'y a aucun patron architectural meilleur qu'un autre. Il faut juste prendre le temps d'analyser le besoin afin de choisir un patron architectural qui correspondra le mieux aux besoins du projet.

Gamage *et al.* quant à eux, nous proposent dans leur étude un outil qui détecte les antipatrons dans les architectures à partir d'un graphe de dépendance et de la théorie des graphes (Gamage *et al.*, 2021). L'outil va recueillir les données via la collecte des traces d'exécutions par un outil de traçage qui va collecter les noms des services, les points de terminaison (*endpoints*) des services et les relations entre les services puis les enregistrer dans une base de données orientée graphe. Ainsi les données structurées dans la base de données, les algorithmes de graphe tels que la centralité (*Degree centrality*), le coefficient de clustering (*clustering coefficient*) et le degré de composants fortement connectés (*strongly connected components*) vont aider à détecter les goulots d'étranglement dans le système, les nœuds, les dépendances cycliques, les chaînes de services et les nano services. Quant aux valeurs seuils des mesures, les chercheurs laissent leurs choix à l'équipe de développement.

3.4 Conclusion

Dans ce chapitre, nous avons fait l'état de l'art des différents travaux académiques concernant les différents axes de notre étude. Nous avons d'abord présenté les travaux traitants des attributs de qualité dans les architectures microservices, ensuite les travaux traitant de la détection du couplage entre les services et enfin les travaux concernant l'identification des patrons et antipatrons dans ces architectures. La plupart des études sur le couplage dans les architectures microservices portent sur la détection du couplage dans le système et l'identification de mesures pour l'évaluer mais peu d'entre elles nous permettent de caractériser des architectures en fonction des dépendances qui existent entre ses microservices.

Nous avons donc relevé quelques études donnant des mesures pour évaluer le couplage et d'identifier ses différentes formes dans une architecture et nous allons, sur 13 architectures de références, effectuer ces mesures, identifier les différents types de couplage et évaluer ces architectures afin de caractériser les dépendances qui existent entre les services.

CHAPITRE 4

MESURE DU COUPLAGE

Dans ce chapitre, nous aborderons les principaux types de couplage qu'on peut rencontrer dans des architectures microservices. Pour chaque type de couplage, nous effectuerons une revue de la littérature dans le but d'obtenir des mesures qui nous permettront d'évaluer les architectures à l'étude.

Le couplage dans une architecture microservice est causé par l'existence de dépendances entre les services (Ntentos et al., 2020). Le couplage peut se manifester de plusieurs façons dans les architectures microservices, mais nous avons décidé de nous attarder dans le cadre de notre projet sur le couplage par les bases de données, le couplage par les communications asynchrones entre les services et le couplage par les dépendances partagées entre les services.

4.1 Couplage au niveau des bases de données

Le caractère distribué des microservices rend complexe la gestion des données car la conservation d'un seul point d'entrée aux données à l'instar des applications monolithiques dans les architectures microservices peut réellement impacter les performances des microservices (Munonye et Martinek, 2020). Les microservices doivent gérer eux-mêmes leurs données, il faut donc dédier à chaque service un moyen de persistance de données qui lui est propre (Newman, 2015).

Dans leur étude portant sur l'évaluation des patrons de stockage de données dans les architectures microservices, Munonye et Martinek (Munonye et Martinek, 2020) ont identifié cinq patrons de stockage de données dans les architectures microservices qui sont le ***database-per-service pattern***, le ***schema-per-service pattern***, le ***private-table-per-service pattern***, le ***shared-database pattern*** et le ***CQRS / Event Sourcing pattern***.

Le *per-service pattern* est une approche selon laquelle les données de chaque microservice restent privées et accessibles uniquement via son API (Chris Richardson, Microservices.io, 2021). Les différents patrons de cette catégorie sont le *database-per-service* dans lequel chaque service dispose d'un serveur de base de

données qui est le patron le plus recommandé (Munonye et Martinek, 2020), ensuite le *schema-per-service* dans lequel chaque service possède son propre schéma de données qui n'est accessible que par lui seul sur un serveur de base de données partagé; ce patron est valable que sur un certain type de bases de données comme les bases de données relationnelles, enfin le *private-table-per-service* dans lequel une ou plusieurs tables ne sont dédiées qu'à un microservice qui a l'exclusivité de lecture et écriture (Munonye et Martinek, 2020).

Le *shared-database pattern* est une approche selon laquelle une base de données unique est partagée par plusieurs microservices et dans laquelle chaque service accède librement aux données détenues par d'autres services à l'aide de transactions ACID locales (Chris Richardson, Microservices.io, 2021). C'est le patron le moins recommandé car il renforce le couplage dans les architectures microservices (Chris Richardson, Microservices.io, 2021).

Le *CQRS/Event Sourcing pattern* qui est une approche basée sur les événements dans laquelle l'accès aux données est divisé en deux parties distinctes : les commandes qui représentent les opérations qui modifient l'état des données et les requêtes qui représentent une demande de données en lecture seule (Munonye et Martinek, 2020). Il est alors défini une base de données de vues, qui est un répliqua en lecture seule conçue pour prendre en charge ces requêtes. L'application maintient le répliqua à jour en s'abonnant aux événements de domaines publiés par le service propriétaire des données. Les événements peuvent concerner l'état d'une entité commerciale, qui est changé à chaque nouvel événement le concernant (Chris Richardson, Microservices.io, 2021).

Pour quantifier le couplage par les bases de données dans notre projet, nous allons utiliser deux mesures que proposent Ntentos *et al.* dans leur étude, à savoir le ***Database Type Utilization Metric*** et le ***Shared Database Interaction Metric*** qui seront décrits dans les sections suivantes (Ntentos *et al.,* 2020).

4.1.1 Database Type Utilisation

Le *Database Type Utilization Metric* (DTU) est une mesure qui permet d'évaluer la proportion de microservices reliés à des bases de données unique dans une architecture, en d'autres mots la présence du patron *database-per-service dans* un système.

Son résultat se situe entre 0 et 1, 1 est la valeur idéale, ce qui signifie alors que chaque service du système possède sa propre base de données sur laquelle il est le seul à pouvoir faire des opérations.

$$DTU = \frac{\text{Database per Service Links}}{\text{Total Service} - \text{to} - \text{Database Links}}$$

- *Database per service links* constitue le nombre de liens entre les services qui ont le patron database-per-service et leurs bases de données respectives.
- *Total Service-to-Database Links* représente tous les liens qui existent entre les services et des bases de données dans le système

4.1.2 Shared Database Interaction

Le Shared Database Interaction Metric (SDBI) est une mesure qui permet d'évaluer la proportion de microservices utilisant un même serveur de base de données dans le système, qu'il y ait partage de données ou non, en d'autres termes, on mesure la présence de *Shared database pattern*, *schema-per-service pattern* et de *private-table-per-service pattern*. Bien que le *schema-per-service pattern* et le *private-table-per-service pattern* sont classés dans la catégorie des *per-service pattern*, il n'en demeure pas moins qu'ils utilisent le même serveur de base de données que les autres microservices, ils produisent certes moins de couplage mais ils sont quand même à la base de couplage.

Son résultat se situe entre 0 et 1 et vu que les bases de données partagées constituent un antipatron, la valeur idéale de cette mesure est 0, ce qui signifie alors que le système ne comporte aucune base de données partagée.

$$SDBI = \frac{\text{Service Interconnections with Shared Database}}{\text{Total Service Interconnections}}$$

- *Service interconnections with Shared Database* représente le nombre de liens qu'il existe entre les services ayant des bases de données partagées et leurs bases de données respectives.
- *Total Service Interconnections* représente le nombre de liens de communication qu'il existe entre les services, que ce soit les liens par appels directs entre les services ou les liens d'échange d'informations par base de données partagée.

4.2 Couplage au niveau des communications asynchrones entre les services

L'une des principales raisons des dépendances inter services est la présence de dépendances entre les services nécessaires pour l'échange et le partage de données entre eux. Pour réduire l'impact de ces dépendances, l'utilisation de communications asynchrones entre les services est recommandée (Ntentos *et al.*, 2020).

En effet les communications asynchrones permettent de réduire considérablement l'impact des dépendances inter services sur le couplage et augmentent la fiabilité du système (Ntentos et *al.*, 2020).

Les patrons de communications asynchrones les plus fréquents dans les microservices selon Ntentos *et al.* sont le *Publish/Subscribe Pattern* dans lequel les informations sont envoyées du service émetteur (*producer*) au service souscripteur (*consumer*), le tout géré par un agent de message qui est garant de la bonne transmission des messages de part et d'autre (Kul et Sayar, 2020), le *Data Polling* dans lequel les services se sondent périodiquement entre eux pour se propager des informations lorsque de nouvelles informations sont disponibles et *l'Asynchronous Direct Invocation* qui est la communication asynchrone directe entre les services qui se fait nativement en fonction des langages de programmation dans lesquels sont développés les services (Ntentos *et al.,* 2020).

Pour évaluer la présence des communications asynchrones dans notre projet, nous utiliserons les deux mesures suivantes : le ***Service Interaction via Intermediary Component (SIC)*** et le ***Asynchronous Communication Utilization (ACU)*** qui seront décrites dans les sections suivantes (Ntentos *et al.,* 2020).

4.2.1 Service Interaction via Intermediary Component

Le Service Interaction via Intermediary Component Metric (SIC) est une mesure qui permet d'évaluer la proportion d'interconnexions entre les services par des relais asynchrones tels que les *Message Brokers* ou agent de messages, le *Publish/Subscribe* et le *Stream Processing*.

Sa valeur se situe entre 0 et 1 dans le meilleur des cas, ce qui correspond à un système dans lequel toutes les connexions entre les services se font par le moyen de communications asynchrones.

$$SIC = \frac{\text{Service Interconnections via [Message Brokers | Pub/sub | Stream]}}{\text{Total Service Interconnections}}$$

- *Service Interconnections via [Message Brokers | Pub/Sub | Stream]* représente les liens entre les services qui s'effectuent par des agents de messages, par des processus pub/sub et par des stream processing.
- *Total Service Interconnections* représente le nombre de liens de communication entre les services, que ce soit les liens par appels directs entre services ou les liens d'échange d'informations par base de données partagée.

4.2.2 Asynchronous Communication Utilization

Le Asynchronous Communication Utilization Metric (ACU) est une mesure qui permet d'évaluer la proportion de la somme des interconnexions asynchrones de services via *API Gateway*/*HTTP Polling*/Appels directs/Base de données partagée par rapport au nombre total d'interconnexions de services.

Sa valeur se situe entre 0 et 1 dans le meilleur des cas, ce qui correspond à un système dans lequel toutes les connexions entre les services se font par le moyen de communications asynchrones.

$$ACU = \frac{\text{Asynchronous Service Interconnections via [API | Polling | Direct Calls | Shared DB]}}{\text{Total Service Interconnections}}$$

- *Asynchronous Service Interconnections via [API | Polling | Direct Calls | Shared DB]* représente les liens asynchrones entre les services par API, Polling, appels directs et bases de données partagées.
- *Total Service Interconnections* représente le nombre de liens de communication entre les services, que ce soit les liens par appels directs entre services ou les liens d'échange d'informations par base de données partagée.

4.3 Couplage au niveau du partage des services

Dans les architectures microservices, certains microservices se partagent des données entre eux afin d'effectuer certaines requêtes. Ils sont alors dépendants d'autres services pour fonctionner correctement, on parle alors de (micro)services partagés. Certaines requêtes, qui nécessitent une succession d'appels entre plusieurs services pour s'exécuter, forment des chaînes d'invocations qui peuvent être problématiques dans le cas où un service appelle un autre dont il est dépendant. Cela conduit à la formation de chaînes de dépendances transitives qui mènent elles aussi à des dépendances cycliques entre

les services, d'où l'importance de prendre en considération les dépendances d'un service avant de l'impliquer dans une chaîne d'appels de services (Ntentos *et al.,* 2020).

Il est alors évident que les services partagés et les chaînes d'appels entre eux constituent une source de couplage dans les architectures microservices, c'est pourquoi Ntentos *et al*. définissent dans leur étude trois cas d'antipatrons qui leur sont liés. Le *Directly Shared Service* ou service directement partagé qu'ils définissent comme un service directement requis et lié à plus d'un autre service, le *Transitively Shared Service* qui est un service qui est lié aux autres services par au moins un intermédiaire et la dépendance cyclique qui est lorsqu'une chaîne d'invocations de services ramène à son origine comme si le service en question est dépendant de lui-même.

Pour quantifier la présence de ces antipatrons dans notre projet, Ntentos *et al*. nous proposent trois mesures qui sont le ***Direct Service Sharing Metric (DSS)***, le ***Transitively Shared Services Metric (TSS)*** et le ***Cyclic Dependencies Detection Metric (CDD).***

4.3.1 Direct Service Sharing

Le *Direct Service Sharing Metric (DSS)* est une mesure qui permet d'évaluer la proportion d'éléments directement partagés dans le système. Sa valeur se situe entre 0 et 1, plus la valeur est proche de 0 mieux c'est. Cela indique moins de couplage par services partagés dans le système.

$$DSS = \frac{\frac{\text{Shared Services}}{\text{Total Services}} + \frac{\text{Shared Services Connectors}}{\text{Total Services Interconnections}}}{2}$$

- *Shared Services* représente le nombre de services partagés dans le système.
- *Total Services* représente le nombre total de services dans le système.
- *Shared Services Connectors* représente le nombre de liens que possèdent les services partagés du système.
- *Total Service Interconnections* représente le nombre de liens de communication entre les services, que ce soit les liens par appels directs entre services ou les liens d'échange d'informations par base de données partagée.

4.3.2 Transitively Shared Services

Le *Transitively Shared Services Metric (TSS)* est une mesure qui permet d'évaluer la proportion d'éléments transitivement partagés dans le système. Sa valeur se situe entre 0 et 1, plus la valeur est proche de 0 mieux c'est. Cela indique moins de couplage par services transitivement partagés dans le système.

$$\mathrm{TSS} = \frac{\frac{\text{Transitively Shared Services}}{\text{Total Services}} + \frac{\text{Transitively Shared Services Connectors}}{\text{Total Services Interconnections}}}{2}$$

- *Transitively Shared Services* représente le nombre de services transitivement partagés dans le système.
- *Total Services* représente le nombre total de services dans le système.
- *Transitively Shared Services Connectors* représente le nombre de liens que possèdent les services transitivement partagés du système.
- *Total Service Interconnections* représente le nombre de liens de communication entre les services, que ce soit les liens par appels directs entre services ou les liens d'échange d'informations par base de données partagée.

4.3.3 Cyclic Dependencies Detection

Le *Cyclic Dependencies Detection Metric (CDD)* est une mesure qui permet d'évaluer la présence de dépendance cyclique dans le système. Elle a une valeur qui est égale à 0 quand il n'y a aucune dépendance cyclique dans le système et elle est égale à 1 lorsqu'il y a au moins une dépendance cyclique dans le système.

CDD = 1 si ServicesCycles > 0

CDD = 0 si ServicesCycles = 0

ServicesCycles représente le nombre de dépendances cycliques dans le système.

4.4 Conclusion

Afin d'évaluer nos différentes architectures en termes de couplage, nous avons ainsi choisi 6 mesures qui seront effectuées sur chacune d'entre elles. Ces six mesures sont distinguées en trois groupes; concernant

le couplage au niveau de la base de données, on a le ***Database Type Utlisation*** et le ***Shared Database Interaction***, concernant le couplage au niveau des communications synchrones on a le ***Service Interaction via Intermediary Component*** et le ***Asynchronous Communication Utilization*** et en ce qui concerne le couplage au niveau du partage de services on a ***Direct Service Sharing, Transitively Shared Service*** et ***Cyclic Dependencies Detection.*** Après le choix de ces mesures, nous passons à l'implémentation de celles-ci dans un outil qui va permettre de supporter notre étude.

CHAPITRE 5

MISE EN ŒUVRE DU PROJET

Dans ce chapitre il sera question de la présentation des différentes architectures à l'étude dans le cadre de notre projet et de l'outil que nous avons développé pour effectuer les mesures du couplage sur ces architectures.

5.1 Architectures étudiées

Dans l'objectif de constituer une banque de données de projets *open source* issus de migrations d'applications monolithiques vers les applications microservices pouvant servir à la recherche ou à l'apprentissage du développement des microservices, Taibi a établi une liste d'architectures microservices qui ont été développés de zéro (*from scratch*) (Taibi, 2019). De cette première ressource, Klewerton et Mosser ont effectué une analyse de chaque architecture en termes de taille, nombre de contributeurs pour retenir une liste d'architectures qui rencontrent les critères des architectures microservices et qui sont raisonnables à étudier. Suite à cette analyse départagée par Kruger, les projets listés ci-dessous ont été désignés et feront l'objet de notre étude.

TeaStore est une application de référence et de test de microservices à utiliser dans les benchmarks et les tests qui émule une boutique en ligne de base pour le thé et les fournitures de thé générés automatiquement. Elle est composée de 5 microservices REST tous écrits en JAVA et dispose d'un interface utilisateur écrit en JavaScript.

SiteWhere est une plate-forme d'activation d'applications IoT open source de force industrielle qui facilite l'ingestion, le stockage, le traitement et l'intégration des données des appareils IoT à grande échelle. La plate-forme est composée de 17 microservices gRPC écrits en JAVA qui s'exécutent sur des technologies de pointe telles que Kubernetes, Istio et Kafka afin de s'adapter efficacement aux charges attendues dans les grands projets IoT.

Spring Petclinic Microservices est une application de gestion d'une clinique vétérinaire, composée de plusieurs services, chacun responsable d'un domaine métier de la clinique vétérinaire : les animaux et leurs propriétaires, leurs visites à la clinique et les vétérinaires. Ces microservices sont tous écrits en Java et communiquent sur HTTP via une API REST.

eShopOnContainers est un exemple d'application de référence .NET Core, optimisée par Microsoft, basée sur une architecture de microservices simplifiée et des conteneurs Docker. L'application représente un site de commerce en ligne composé de microservices tous écrits en C#, s'exécutant sur des conteneurs Docker, et d'une application frontale écrit avec JavaScript.

Microservices Demo est une application de démonstration de microservices cloud native. La boutique en ligne consiste en une application de microservices écrits dans des langages divers tels que Python, Go, Java ou JavaScript communiquant par gRPC et REST. C'est une application de commerce électronique basée sur le Web où les utilisateurs peuvent parcourir les articles, les ajouter au panier et les acheter.

Microservices reference Implementation est une application de référence destinée à gérer un service de livraison par drones. L'application est composée de plusieurs microservices essentiellement écrits en Java, C# et JavaScript, communiquant par HTTP REST et par un relais asynchrone. Cette implémentation de référence présente un ensemble de bonnes pratiques pour créer et exécuter une architecture de microservices sur Microsoft Azure, à l'aide de Kubernetes.

Piggy Metrics est une application de conseiller financier conçue pour démontrer le modèle d'architecture de microservice à l'aide de Spring Boot, Spring Cloud et Docker. Elle est composée de 4 microservices écrits en JAVA et qui communiquent par des API REST.

PitStop est une application basée sur un système de gestion de garage pour Pitstop - un garage fictif. L'objectif principal de cet exemple est de démontrer plusieurs concepts d'architecture logicielle tels que : Microservices, CQRS, Event Sourcing, Domain Driven Design (DDD). L'application est composée de microservices essentiellement écrits en C#, communiquant par des API REST et des relais asynchrones.

RobotShop est une application de commerce électronique qui dispose de microservices écrits dans des langages variés tels que Python, Go, Java et JavaScript, communiquant par des API REST et des relais

asynchrones. L'application est destinée à servir de base pour tester et apprendre les techniques d'orchestration et de surveillance des applications conteneurisées.

Digota est une application à microservices de commerce électronique conçu pour être la norme moderne pour les systèmes de commerce électronique. Il est composé de microservices tous écrits en Go et communicants uniquement par gRPC. Il fournit une interface RPC propre, puissante et sécurisée.

Parts Unlimited MRP Microservices est une application fictive de planification des ressources de fabrication (MRP) externalisée à des fins de formation basée sur la description des chapitres 31 à 35 du projet Phoenix de Gene Kim, Kevin Behr et George Spafford. Cette application comporte 5 microservices écrits en JAVA, JavaScript et C# qui communiquent par des API REST.

Blueprint Microservices est une application de microservice de micro-boutique développée avec Vert.x. Ce référentiel est destiné à illustrer la conception d'une architecture de microservices et le développement d'applications de microservices à l'aide de Vert.x. Elle est composée de 7 microservices écrits en JAVA et communicant avec des API REST et des relais asynchrones.

Microservices Demo est la partie accessible à l'utilisateur d'une boutique en ligne qui vend des chaussettes. Elle est destinée à faciliter la démonstration et les tests de microservices et de technologies cloud natives. L'application est composée de 7 microservices écrits à l'aide de Spring Boot, Go kit et Node.js qui s'exécutent dans des conteneurs Docker.

Table 5.1 - Architectures microservices évaluées dans le cadre de notre projet

Nom du projet	**Lien github**	**Nombre de microservices**	**Qté de lignes de code**
Teastore	https://github.com/DescartesResearch/TeaStore	5	27.422
Sitewhere	https://github.com/sitewhere/sitewhere	17	38.885
Petclinic	https://github.com/spring-petclinic/spring-petclinic-microservices	3	12.212
eShopOnContainers	https://github.com/dotnet-architecture/eShopOnContainers	4	123.018
Microservices Demo	https://github.com/GoogleCloudPlatform/microservices-demo	9	18.187
Microservices reference Imp	https://github.com/mspnp/microservices-reference-implementation	5	10.215
PiggyMetrics	https://github.com/sqshq/PiggyMetrics	4	19.095
PitStop	https://github.com/EdwinVW/pitstop	8	57.146
Robot Shop	https://github.com/instana/robot-shop	7	4.513
Digota	https://github.com/digota/digota	4	19.358
PartsUnlimited	https://github.com/microsoft/PartsUnlimitedMRPmicro	5	82.265
Blueprint Microservices	https://github.com/sczyh30/vertx-blueprint-microservice	7	9.558
Microservices demo	https://github.com/microservices-demo	7	41.388

5.2 Technologies utilisées

Dans l'objectif de supporter notre étude, nous avons conçu un programme dans lequel on a chargé tous nos projets et implémenté toutes nos mesures dans le but de faire une meilleure analyse de nos architectures.

Le langage de programmation **Python** a été utilisé pour la conception du programme qui servira à effectuer les mesures sur les différentes architectures étudiées.

NetworkX qui est une bibliothèque python servant à l'étude des graphes et des réseaux sera utilisée pour représenter les différentes architectures étudiées sous forme de graphes pour y effectuer des opérations.

Notre programme est composé de trois classes, à savoir la classe **Archi** représentant une architecture, la classe **Service** qui représente un composant, notion qui sera expliquée dans les prochaines sections, qui peut être soit un microservice ou une entité de stockage de données puis la classe **Relation** qui représente un lien entre des composants.

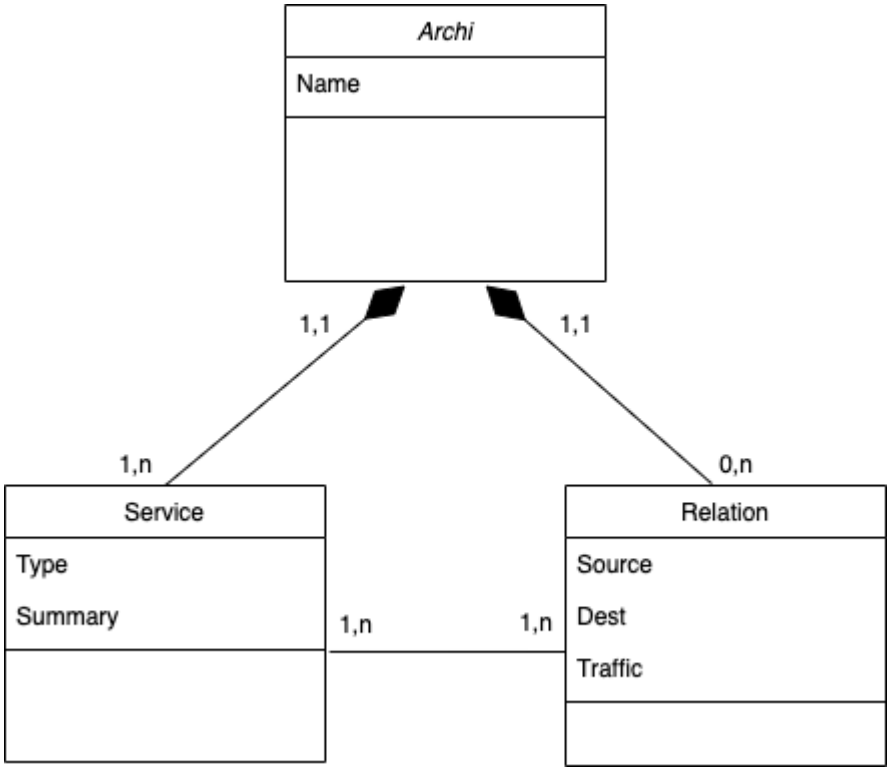

Figure 5.1 - Diagramme de classe du programme d'analyse des architectures

5.3 Analyse manuelle du code source des architectures étudiées

Afin de recueillir les données nécessaires des architectures à l'étude, nous avons effectué une analyse manuelle des codes sources de chacune d'elles.

Notre analyse avait pour premier objectif d'identifier les différents microservices qui composent chaque projet, le langage de programmation dans lequel il a été écrit, ainsi que leur entité de stockage, ensuite d'identifier les interfaces de communications de chaque microservice et le type de communication de ces interfaces, soit synchrone ou asynchrone, le protocole de communication des microservices et enfin d'identifier les appels qui s'effectuent entre les différents microservices d'une architecture dans le but de dresser un graphe qui pourra être analysé par le programme qui sera mis en place. Le tableau suivant nous montre un aperçu de ce qu'on peut obtenir après l'analyse du code source d'un projet.

Table 5.2 - Tableau à remplir suite à l'analyse du code source des projets

Nom Projet	Microservices	Langage	Protocole de communication	Emplacement dans le projet	MS1	MS2	MS3
	MS1	JAVA	REST	Src/services/ms1			x
	MS2	JAVA	grpc	Src/services/ms2	_		
	MS3	JAVA	REST	Src/services/ms3		x	

- X : Lien synchrone
- _ : Lien asynchrone

5.4 Modélisation des données recueillies

Une fois l'analyse manuelle des projets terminée, les données recueillies sont formatées selon le modèle de décomposition de services que nous proposent Zdun *et al*. (Zdun *et al.*, 2017) dans leur étude. Selon eux, les architectures microservices sont constituées de composants qui peuvent être les services ou les interfaces et de connecteurs qui lient les composants entre eux. En fonction des données recueillies à la suite de notre analyse, nous avons obtenu les éléments qui suivent.

Nous avons obtenu les types de composants suivants :

- **FACADE** représente un composant qui n'est pas un microservice mais qui communique avec les microservices, tel un *API GATEWAY*.
- **SERVICE** représente un composant de type microservice.
- **DATABASE** représente les entités de stockage telles que les bases de données.
- **EVENT_BUS** représente un composant de type Event BUS qui reçoit et propage les événements dans le système.
- **MESSAGE_BROKER** représente un composant de type message broker qui reçoit les messages et les propage aux consommateurs.

Nous avons obtenu les types de connecteurs suivants :

- **GET | POST | PUT | DELETE** représente des connecteurs décrivant des appels *RESTFULL*
- **grpc** qui représentent des appels de fonctions grpc entre deux composants. Quand il est suivi de ASYNC alors l'appel est asynchrone
- **TABLE** représente les connecteurs qui lient des composants de type SERVICE à des composants de type DATABASE lorsque le service écrit dans des tables précises d'une base de données relationnelles**.**
- **SCHEMA** représente les connecteurs qui lient des composants de type SERVICE à des composants de type DATABASE lorsque le service écrit dans un SCHEMA précis, qui est une structure propre aux bases de données relationnelles.
- **COLLECTION** représente les connecteurs qui lient des composants de type SERVICE à des composants de type DATABASE lorsque le service écrit dans une collection précise d'une base de données MongoDB.
- **FULL_DATABASE** représente les connecteurs qui lient des composants de type SERVICE à des composants de type DATABASE lorsque le service a accès à toute la base de données sans restriction.

Les données d'une architecture peuvent alors se présenter comme suit dans notre programme :

```
Projet = { "name" : "Test",
"COMPOSANTS" : [
[0, 'FACADE', 'WEB UI'],
[1, 'SERVICE', 'Registry'],
[2, 'SERVICE', 'Authentication'],
[3, 'SERVICE', 'ImageProvider'],
[4, 'SERVICE', 'Persistence'],
[5, 'SERVICE', 'Recommender'],
[6, 'DATABASE', 'MYSQL']],


"CONNECTEURS" : [
[0, 1, {"link": ["POST /{name}/{location} ", " DELETE /{name}/{location}"]}],
[0, 2, {"link": ["POST /cart/add/{id} ", " POST /cart/remove/{id} ", " PUT /cart/{id}
", " POST /useractions/login ", " POST /useractions/logout ", " POST
/useractions/placeorder ", " POST /useractions/isLoggedIn"]}],
[0, 3, {"link": ["POST /getProductImages ", " POST /getWebImages"]}],
[0, 4, {"link": ["GET /products ", " GET /categories ", " GET /users ", " GET
/orders"]}],
[0, 5, {"link": ["POST /recommend/{uid}"]}],
[5, 4, {"link": ["GET /orders ", " GET /ordersitems"]}],
[2, 4, {"link": ["POST /products ", " POST /orders ", " POST /ordersitems"]}],
[3, 4, {"link": ["GET /generatedb ", " /GET products ", " GET/categories"]}],
```

```
[4, 6, {"link": ["FULL_DATABASE MYSQL"]}],
] }
```

5.5 Présentation de l'outil

L'outil[9] que nous avons conçu pour supporter notre étude nous permet entre autres d'afficher les différentes architectures sous forme de graphes, d'effectuer chacune des mesures choisies sur chaque architecture individuellement et de les calculer globalement puis de les exporter dans une feuille de calculs pour une exploitation future.

Table 5.3 - Rendu des mesures effectuées sur une architecture dans une feuille de calcul

Projet	DTU	SDBI	SIC	ACU	DSS	TSS	CDD
Example	0.29	0.5	0	0	0.39	0	0

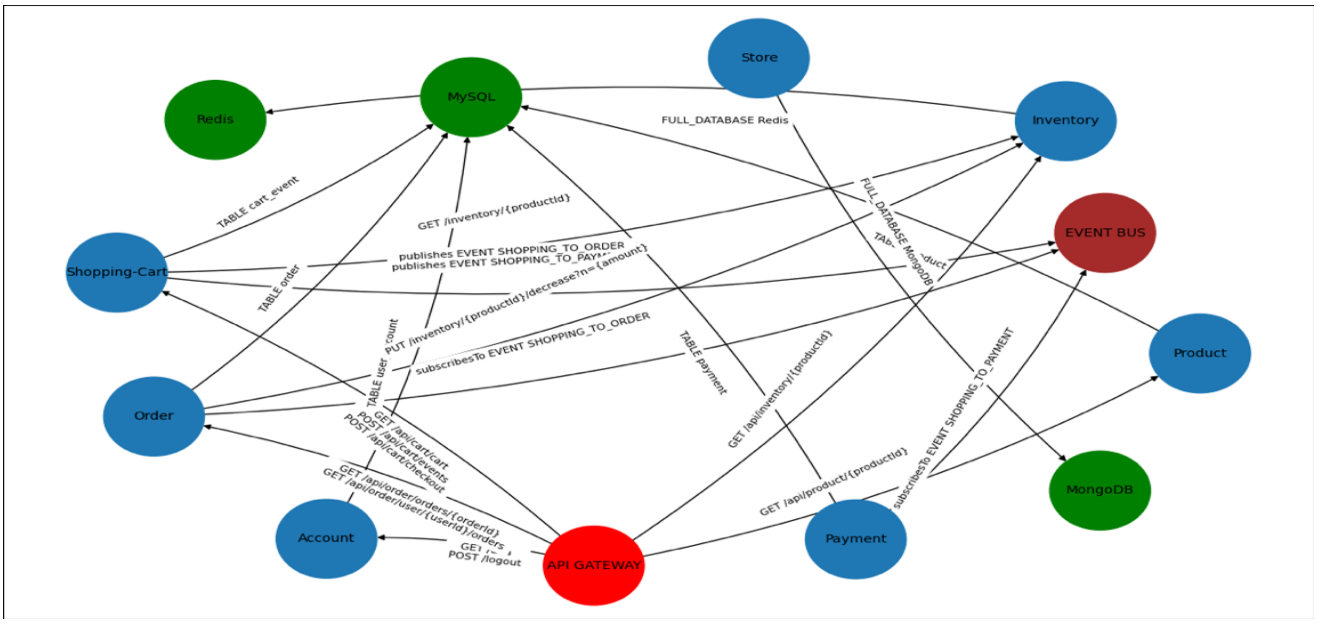

Figure 5.2 - Affichage du projet Blueprint sous forme de graphe grâce à la bibliothèque NetworkX

[9] https://github.com/mohamedkoita/graphpython

5.6 Conclusion

Après avoir expliqué le choix des architectures à étudier et décrit les technologies nécessaires à la mise en place de l'application servant à supporter notre étude, nous allons effectuer les différentes mesures sur les architectures et les analyser dans le chapitre suivant.

CHAPITRE 6

RÉSULTATS ET ANALYSE

6.1 Résultats obtenus

Cette section présente les résultats obtenus après l'évaluation du couplage avec notre outil sur les 13 systèmes microservices *open source* qui ont fait l'objet de notre étude.

Table 6.1 - Résultats des mesures du couplage sur les 13 architectures

Nom du projet	DTU	SDBI	SIC	ACU	DSS	TSS	CDD
Teastore	1	0	0	0	0.6	0	0
Sitewhere	0	0.11904762	0.35714286	0	0.45728291	0.33753501	0
Petclinic	0	1	0	0	0	0	0
eShop	1	0	1	0	0.625	0	0
GoogleCloudPlatform	1	0	0	0	0.1984127	0	0
Mspnp	1	0	0.4	0	0	0	0
Piggymetrics	0	0.625	0	0	0	0.1875	0
PitStop	0.71428571	0.2	0.8	0.2	0.4625	0	0
RobotShop	1	0	0.28571429	0	0.42857143	0.21428571	0
Digota	0	0.5	0	0	0.25	0.4375	1
PartsUnlimited	0	1	0	0	0	0	0
Blueprint	0.28571429	0.5	0.3	0	0.39285714	0	0
MicroservicesDemo	1	0	0.5	0	0.32142857	0.19642857	0

6.2 Analyse des résultats

6.2.1 Le couplage par les bases de données

Le couplage par la base de données est présent dans 7 projets sur 13 étudiés, soit 53,8% de taux de présence.

Le DTU ne prend en compte que les services reliés à des moyens de stockage, mais pas les services qui ont réellement besoin de stockage, chose qui peut créer des lacunes lors de cette mesure. Dans le cas du projet **Teastore**, le DTU à une valeur égale à 1, ce qui signifie en théorie que tous les services reliés à des bases de données sont reliés à une base de données unique sur laquelle ils ont tout le contrôle.

En analysant de plus près l'architecture, on constate qu'il existe une seule base de données dans le système, reliée au service **Persistance**. Le service **Persistance** quant à lui est relié aux services

Authentication, **ImageProvider** et **Recommander**. Après Analyse en profondeur du service **Persistance** et de la base de données Mysql, on constate que nous sommes en face d'un cas de base de **données partagée masquée par un service**. En effet, les informations que les services Authentication, ImageProvider et Recommander ont besoin de stocker et d'utiliser pour leur fonctionnement passent obligatoirement par le service P**ersistance** qui a la gestion totale de l'unique base de données du système, ce qui rend cette base de données partagée entre tous les services du système. Cette configuration empêche alors le passage à l'échelle d'un seul service, car celui-ci devra être impérativement suivi du service Persistance pour gérer le flux de données.

Revenant au DTU, pour que celui-ci soit plus efficace, il est important que soit pris en compte le nombre de services ayant besoin de stocker et d'utiliser des informations et le nombre différents de moyens de stockage dans le système. De plus, on doit également rechercher la présence d'un service relié à plusieurs services et qui lui communique avec la base de données pour s'assurer qu'on n'est pas en présence d'un antipatron de bases de données partagées masquées par un service de persistance.

Le SDBI quant à lui, varie également d'une architecture à l'autre, il est généralement élevé lorsque le DTU est bas et vice versa. Cependant, il pourrait aussi gagner en efficacité si le calcul faisait la distinction entre les cas où il y a partage de données ou non. En effet, dans plusieurs types de systèmes de gestion de bases de données, il est possible d'allouer à un service un groupe de tables qui lui est dédié sur lequel il a l'entière maitrise, c'est le cas des schémas dans les bases de données relationnelles. Ainsi, les services qui auront des schémas comme moyen de stockage des données, seront plus proches du patron *database-per-service* que du patron *shared-database* à la seule différence que les données de tous les services seront stockées

ensemble et devront passer à l'échelle ensemble. Il y a certes toujours du couplage, mais ce couplage sera moindre.

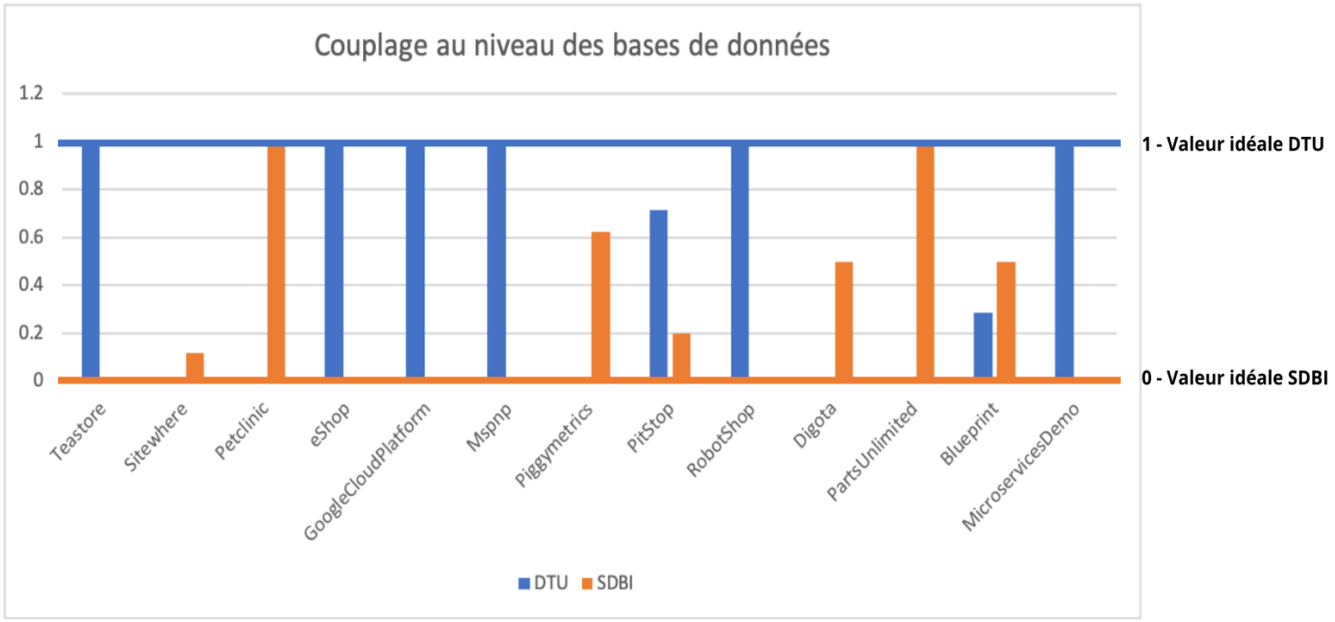

Figure 6.1 - Histogramme du couplage au niveau de la base de données

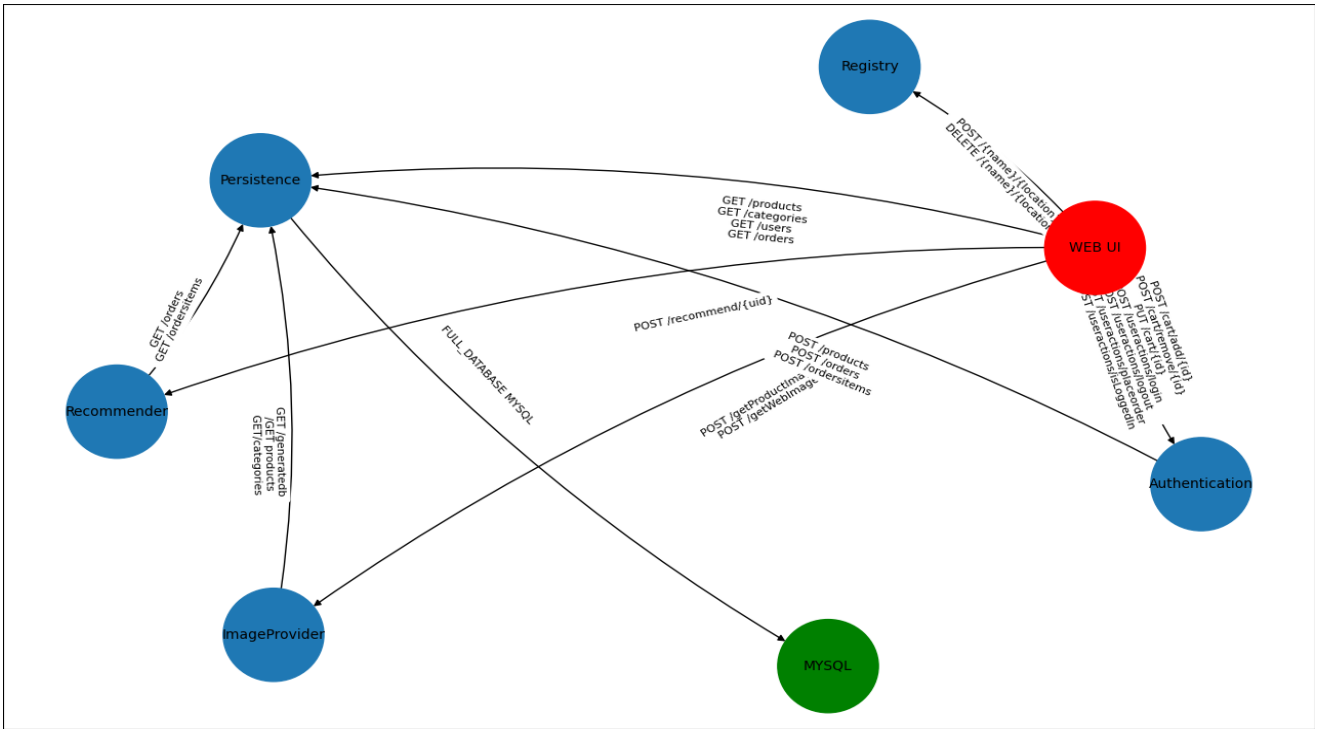

Figure 6.2 -Graphe de services Teastore

6.2.2 Le couplage au niveau de la communication asynchrone

Sur les 13 architectures étudiées dans le cadre de notre projet, la mesure du ACU est nulle dans 12 architectures sur 13, donc 92,3% des cas.

Le SIC quant à lui varie d'une architecture à l'autre. Ce qu'on peut comprendre en comparant ces mesures, c'est que dans la majorité des cas, lorsqu'il y a des communications asynchrones entre deux services de la même architecture, elles sont effectuées de services relais asynchrones tels que les agents de messages, *pub/sub* et *stream processing*, plutôt que par passerelle API qui concerne généralement les appels de services inter architecturaux, ou les appels asynchrones directs qui sont intra services. Cela s'explique par les caractères agnostique et distribué de ces outils qui s'intègrent facilement d'un service à un autre et parce qu'ils permettent de gérer simultanément un grand nombre de messages simultanément et limitent les pertes de messages.

Les communications asynchrones sont utilisées dans environ 54% des systèmes étudiés et représentent un moyen efficace de limiter le couplage dans les architectures microservices.

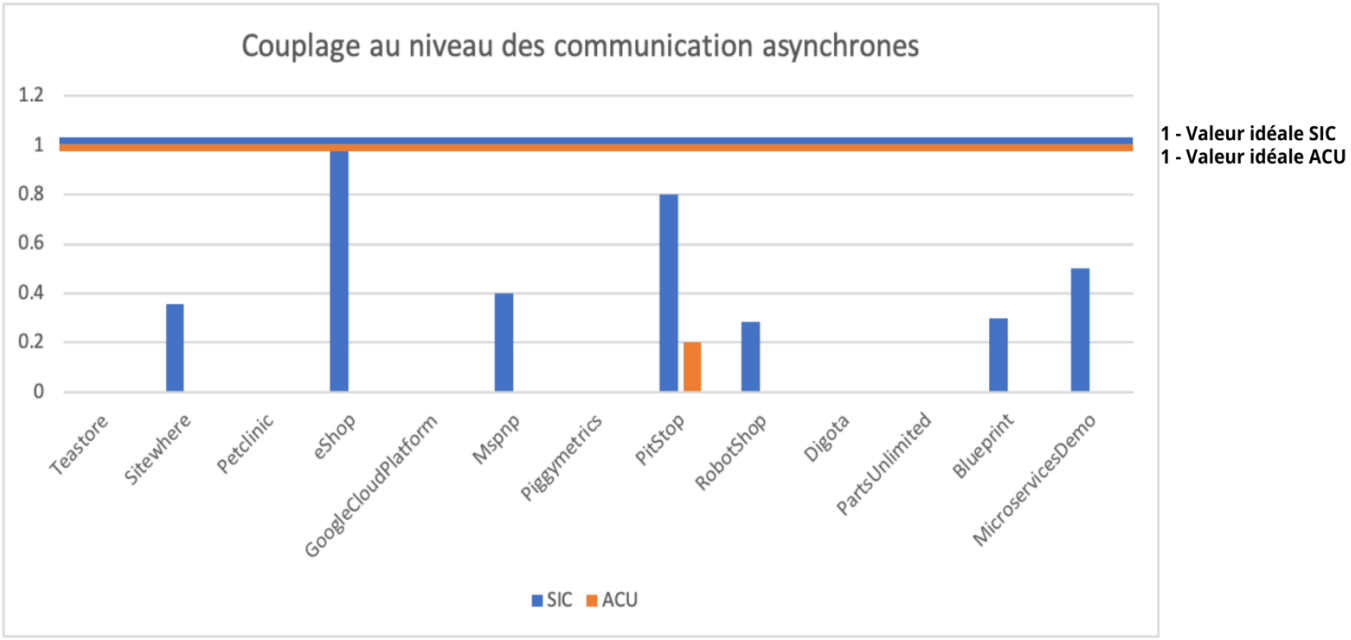

Figure 6.3 - Histogramme du couplage au niveau des communications asynchrones

6.2.3 Le couplage au niveau du partage des services

On constate la présence de services directement partagés dans 9 architectures sur 13 étudiées, soit 69%. Cet antipatron est plutôt fréquent et dû aux besoins des services d'échanger des données entre eux.

Le TSS quant à lui, présent dans 5 projets sur 13 étudiés, soit 38% des projets, indique la présence de services transitivement partagés qui conduisent à des dépendances cycliques.

Nous constatons que seulement 1 système étudié sur 13 contient au moins une dépendance circulaire, ce qui signifie que cet antipatron est bien connu des concepteurs qui l'évitent dans la majorité des cas.

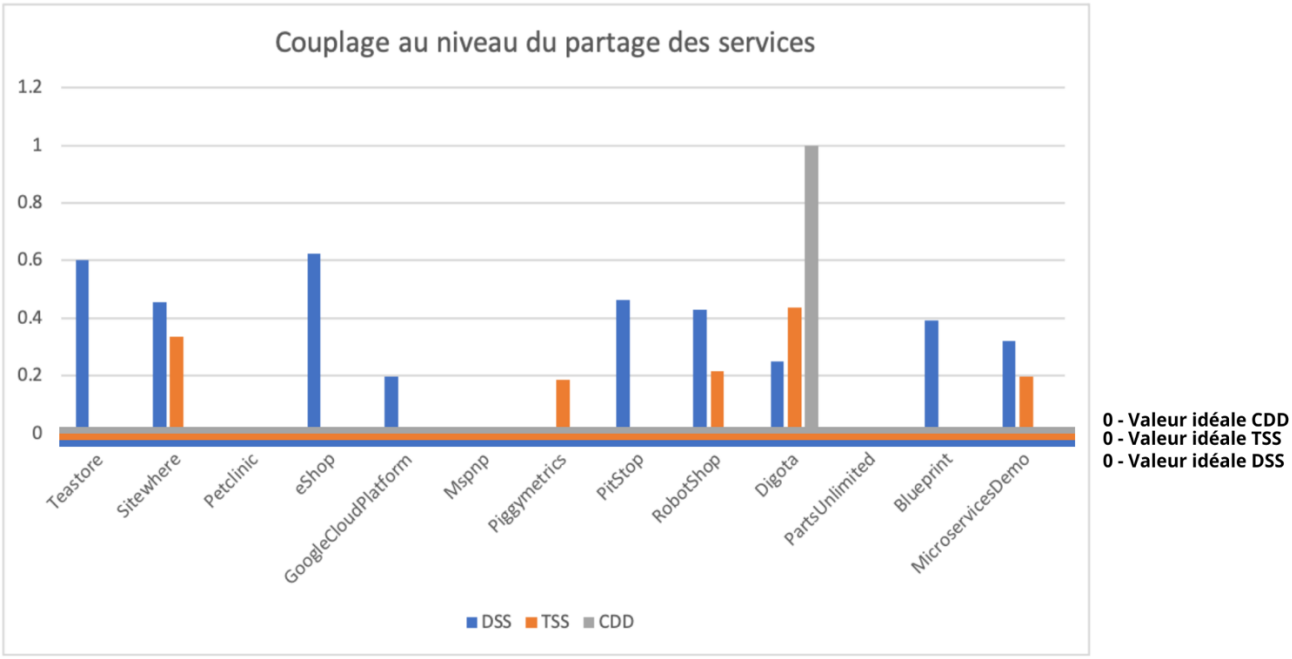

Figure 6.4 - Histogramme du couplage au niveau du partage des services

6.2.4 Architecture avec le moins de couplage

Après le calcul des mesures sur toutes les architectures étudiées dans le cadre de notre projet, nous avons décidé de choisir l'architecture ayant les meilleures mesures concernant le couplage, qui peut être considérée comme architecture modèle.

L'architecture choisie est le **Projet 6 - *Microservices Reference Implementation*** qui a obtenu des mesures de couplage satisfaisantes. Le DTU de cette architecture est égal à 1, c'est à -dire de tous les services ayant besoin de stockage, soit le service **Delivery**, le service **Package** et le service **Ingestion** ont chacun leurs propres moyens de stockage qu'ils gèrent entièrement et ont alors des SDBI est nuls.

Concernant le couplage par communication asynchrone entre les services, le SIC est égal à 0,4, ce qui signifie que 40% des interactions entre les services de l'architecture s'effectuent par le biais de relais asynchrones, dans ce cas précis un bus d'événements ou EVENT BUS.

Quant au dernier aspect du couplage étudié, soit le couplage par services partagés, toutes les mesures sur celui-ci sont nulles, c'est-à-dire pas de présence de services directement partagés, de services transitivement partagés et de dépendances cycliques.

Au vu de tout ce qu'on a cité, on peut donc affirmer que l'architecture détaillée plus haut est l'une des meilleures architectures étudiées en termes de couplage.

Table 6.2 - Mesures de couplage projet Microservice Reference Implementation

Projet	DTU	SDBI	SIC	ACU	DSS	TSS	CDD
MSPNP	1	0	0,4	0	0	0	0

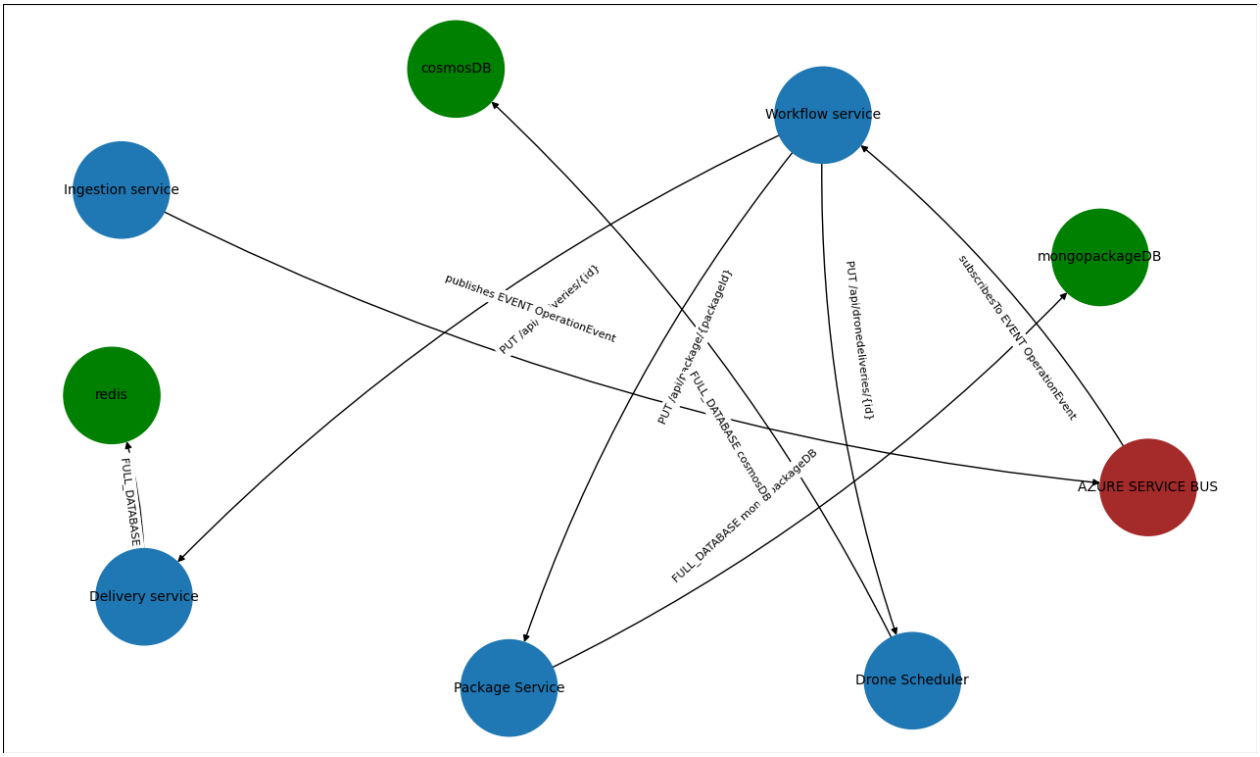

Figure 6.5 - Graphe de services Microservices Reference Implementation

6.2.5 Architecture avec le plus de couplage

Après le calcul de toutes les mesures sur les projets étudiés, nous avons également décidé de choisir l'architecture qui détient les mesures de couplage les moins satisfaisantes, qui est **le projet 10 - Digota**.

Le DTU de cette architecture est égal à 0, c'est-à-dire que parmi les services de l'architecture ayant besoin de stockage, soit **Payment**, **Sku**, **Order** et **Product,** aucun n'a un service de stockage qui lui appartient totalement sur lequel il a le contrôle entier. Ces services possèdent chacun **une collection** dans MongoDB, qui est l'équivalent des schémas dans les bases de données relationnelles, qui confèrent un groupe de tables à usage exclusif pour chaque service, mais qui demeure sur le même serveur de bases de données. Cela a certes un effet moins grave sur le couplage que si chaque service pouvait écrire librement dans les tables de l'autre, mais rend le passage à l'échelle d'un service qui est l'un des principes fondamentaux des architectures microservices plus coûteux, car la base de données de l'application entière aura besoin d'être répliquée à chaque fois ; le service ne sera pas donc totalement indépendant. Le SDBI quant à lui est égal à 0.5 ce qui signifie que la moitié de l'échange de données entre les services est susceptible de passer par la base de données commune qu'ils utilisent.

Concernant les communications asynchrones, le SIC et le ACU sont nuls, c'est -à -dire qu'il n'existe aucune communication asynchrone entre les différents services de l'architecture.

Concernant le couplage par partage de services, le DSS est égal à 0,25 et le TSS égal à 0,4375, l'architecture comporte donc bon nombre de services directement partagés et transitivement partagés.

Nos mesures ont également détecté la présence d'une dépendance cyclique, notamment entre les services **Sku** et **Product**, qu'il serait conseillé de fusionner pour faire disparaitre la dépendance cyclique.

Table 6.3 - Mesures du couplage Projet Digota

Projet	DTU	SDBI	SIC	ACU	DSS	TSS	CDD
0	0,5	0	0	0	0,25	0,4375	1

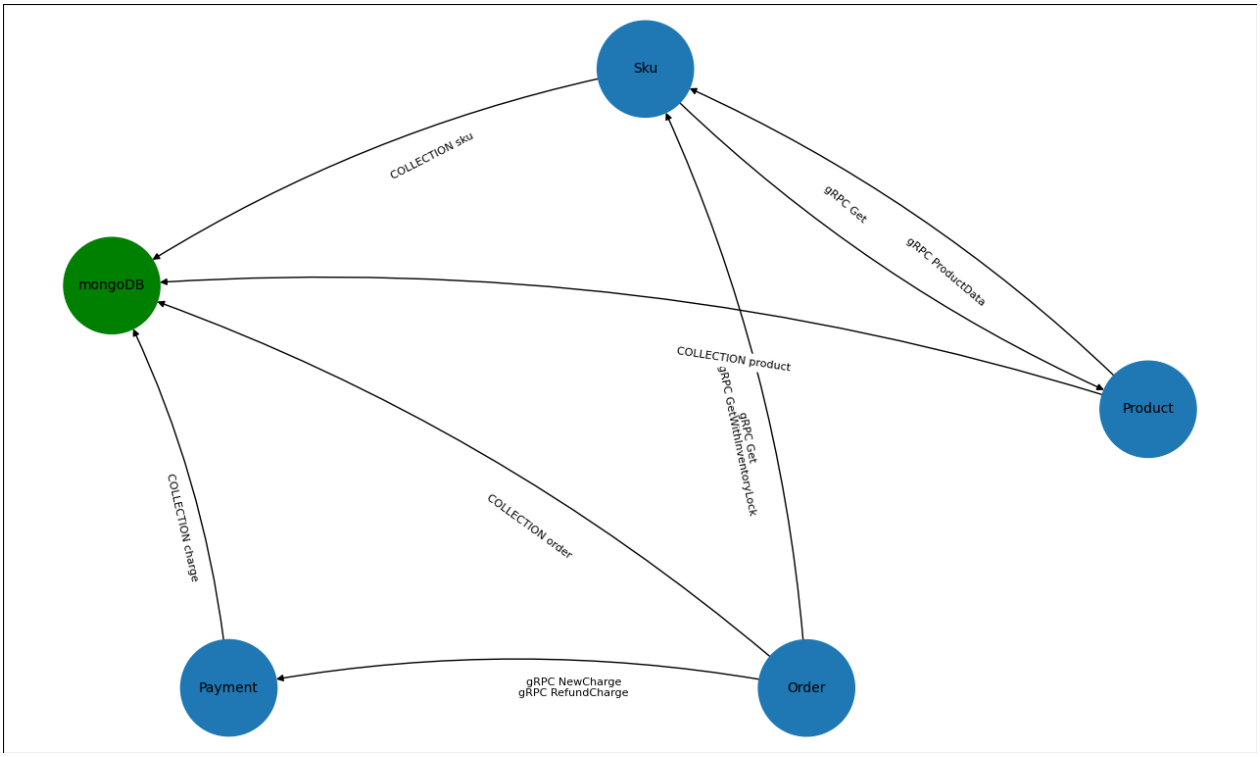

Figure 6.6 - Graphe des services Projet Digota

6.3 Conclusion

Dans ce chapitre nous avons présenté et analysé les résultats obtenus à la suite de notre étude. Le couplage par base de données partagée est présent dans 53,8% des projets étudiés et dans 61,5% des cas on avait la présence de services qui possédaient des bases de données uniques par service. Parmi les systèmes possédant des bases de données uniques par services, dans 75% des cas, cela concernait tous les services de l'application. Nous avons remarqué tout de même qu'il faut prendre en compte dans le calcul du DTU, le nombre de services ayant besoin de stockage et le nombre de moyens de stockage présent dans le système pour éviter de faire face aux cas de base de données partagée masquée par un seul service qui est chargé d'effectuer toutes les persistances en base de données du système.

La communication asynchrone entre les services d'une même architecture présente dans 53,85% des systèmes se fait dans 87,5% des cas par le biais de relais asynchrones tels que les agents de messages et dans 12,5% des cas par appels asynchrones directs entre les services. Les agents de messages sont les plus utilisés, car ils s'intègrent facilement entre plusieurs langages de programmation et possèdent des

mécanismes qui leur permettent de recevoir et traiter un grand nombre de messages avec de faibles taux de perte.

Le couplage par partage de service est présent dans 69,2% des systèmes étudiés et, bien que difficile à éradiquer dû aux besoins des services de s'échanger des données, peut se remplacer par des échanges asynchrones entre les services ou des fusions de plusieurs services en un seul selon les cas. On constate néanmoins la faible présence de dépendances circulaires, soit 7,7% dans les systèmes étudiés.

Nous avons par la suite présenté l'architecture la mieux conçue en termes de couplage qui pourra servir de modèle de conception et l'architecture présentant le plus de couplage, montrant les points auxquels il faut faire attention lors de la conception de systèmes à base de microservices.

CONCLUSION

Les architectures microservices sont un réel sujet d'actualité ces derniers temps. Les géants de la technologie tels que Netflix et Amazon en sont devenus de grands évangélistes. Beaucoup d'entreprises décident alors de migrer leurs systèmes monolithiques vers ces architectures dans l'objectif d'avoir des systèmes qui passent à l'échelle plus facilement et plus résilients.

Cet engouement révèle, à l'instar des paradigmes qui l'ont précédé, de réels défis liés au couplage qui, à cause de la nature distribuée et dynamique des microservices, entraine plus de conséquences. Il est donc important de pouvoir évaluer ce couplage afin de le comprendre et prendre des décisions pour le réduire dans les systèmes.

Nous avons proposé à cet effet un cadre d'évaluation du couplage des architectures microservices qui s'intéresse à trois aspects du couplage à savoir, le couplage par la base de données, le couplage par les communications asynchrones et le couplage par partage de services. Pour supporter notre étude, nous avons conçu un outil qui nous a permis de calculer le couplage sur les architectures microservices, duquel nous nous sommes servis pour effectuer une étude empirique des dépendances interservices sur 13 architectures *open source*.

Nous avons alors constaté que le couplage par base de données partagée est présent dans 53,8% des projets étudiés, 61,5% des cas étudiés possédaient des bases de données uniques et parmi ceux-ci 75% concernaient tous les services de l'application. Pour éviter de faire face à des bases de données partagées masquées par un service chargé de la persistance, il faut prendre en compte le nombre de services ayant besoin de stockage et le nombre de moyens de stockage du système.

La communication asynchrone entre les services d'une même architecture présente dans 53,85% des systèmes se fait dans 87,5% des cas par le biais de relais asynchrones tels que les agents de messages et dans 12,5% des cas par appels asynchrones directs entre les services.

Le couplage par partage de service est présent dans 69,2% des systèmes étudiés, peut se remplacer par des échanges asynchrones entre les services ou des fusions de services selon les cas. On a aussi constaté une faible présence de dépendances circulaire, soit dans 7,7% des systèmes étudiés.

La première option à envisager pour réduire le couplage dans une application à microservices, est la fusion des services. Il s'agira alors de reconsidérer les frontières entre les microservices pour réduire les dépendances entre eux. Si la première option ne marche pas, il faut envisager de remplacer le plus possible de communications synchrones par des communications asynchrones qui ont moins d'impacts sur le fonctionnement de l'application. Enfin pour réduire le couplage par les bases de données, il faut dédier à chaque microservice, une base de données unique sur laquelle chacun aura la gestion exclusive.

Nos travaux pourront être utiles aux concepteurs de systèmes ou aux chercheurs aux fins de mesures de ces différents types de couplage dans leurs systèmes.

Nous considérons comme perspectives de nos travaux, l'augmentation du nombre de mesures sur différents types de couplages présents dans les architectures microservices, l'évolution de notre analyse qui se fait actuellement statiquement vers une analyse en temps réel du couplage, avec des marqueurs qui pourront s'insérer dans le code sous forme d'annotations, afin de permettre aux concepteurs de mieux contrôler l'évolution du couplage dans leurs systèmes, ce qui aura un grand impact sur l'effort et les coûts de développement à court et long terme des applications à microservices.

RÉFÉRENCES

A. Akbulut and H. G. Perros (2021). "Performance Analysis of Microservice Design Patterns," in IEEE Internet Computing, vol. 23, no. 6, pp. 19-27, 1 Nov.-Dec. 2019, doi: 10.1109/MIC.2019.2951094.

Apolinário, D.R., de França, B.B. A method for monitoring the coupling evolution of microservice-based architectures. J Braz Comput Soc 27, 17. https://doi.org/10.1186/s13173-021-00120-y
Consulté : Mai 2022

E. B. Allen, T. M. Khoshgoftaar and Y. Chen (2001). "Measuring coupling and cohesion of software modules: an information-theory approach," Proceedings Seventh International Software Metrics Symposium, 2001, pp. 124-134, doi: 10.1109/METRIC.2001.915521.

Evans, E. (2003). Domain-Driven Design. Pearson Education

F. Rademacher, J. Sorgalla and S. Sachweh (2018). "Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective," in IEEE Software, vol. 35, no. 3, pp. 36-43, May/June 2018, doi: 10.1109/MS.2018.2141028.

F. Wan, X. Wu and Q. Zhang (2020). "Chain-Oriented Load Balancing in Microservice System," 2020 World Conference on Computing and Communication Technologies (WCCCT), 2020, pp. 10-14, doi: 10.1109/WCCCT49810.2020.9169996.

Fowler, M (2006). Continuous Integration.
https://martinfowler.com/articles/continuousIntegration.html
Consulté : Mars 2022

Fowler, M (2013). Continuous Delivery.
https://martinfowler.com/bliki/ContinuousDelivery.html
Consulté : Mars 2022

Fowler, M (2014). Circuit Breaker
https://martinfowler.com/bliki/CircuitBreaker.html
Consulté : Mars 2022

Fowler, M (2015). Microservices Trade-Offs.
https://martinfowler.com/articles/microservice-trade-offs.html
Consulté : Mars 2022

G2.com (2022). Best Service Discovery Software in 2022
https://www.g2.com/categories/service-discovery
Consulté : Avril 2022.

Heusser, M (2022). 30 essential container technology tools and resources
https://techbeacon.com/enterprise-it/30-essential-container-technology-tools-resources-0
Consulté : Avril 2022

I. U. P. Gamage and I. Perera (2021). "Using dependency graph and graph theory concepts to identify anti-patterns in a microservices system: A tool-based approach," 2021 Moratuwa Engineering Research Conference (MERCon), 2021, pp. 699-704, doi: 10.1109/MERCon52712.2021.9525743.

J. Li (2021). "Online Experiment Platform: A Microservices-based Cloud Native Application," 2021 IEEE International Conference on Computer Science, Electronic Information Engineering and Intelligent Control Technology (CEI), 2021, pp. 435-439, doi: 10.1109/CEI52496.2021.9574601.

K. B. Long, H. Yang and Y. Kim (2017). "ICN-based service discovery mechanism for microservice architecture," 2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN), 2017, pp. 773-775, doi: 10.1109/ICUFN.2017.7993899.

K. Munonye and P. Martinek (2020). "Evaluation of Data Storage Patterns in Microservices Architecture," 2020 IEEE 15th International Conference of System of Systems Engineering (SoSE), 2020, pp. 373-380, doi: 10.1109/SoSE50414.2020.9130516.

Kim, G., Humble, J., Debois, P., Willis, J. (2016). The DevOps Handbook. IT Revolution Press, LLC

Knoche, H. & Hasselbring, W., (2019). Drivers and Barriers for Microservice Adoption – A Survey among Professionals in Germany. Enterprise Modelling and Information Systems Architectures (EMISAJ) – International Journal of Conceptual Modeling: Vol. 14, Nr. 1. Berlin: Gesellschaft für Informatik e.V.. (S. 1-35). DOI: 10.18417/emisa.14.1

L. J. Jagadeesan and V. B. Mendiratta (2020). "When Failure is (Not) an Option: Reliability Models for Microservices Architectures," 2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2020, pp. 19-24, doi: 10.1109/ISSREW51248.2020.00031.

Lewis, J et Fowler, M (2014). Microservices
https://martinfowler.com/articles/microservices.html
Consulté : Mars 2022

Maayan, D. (2021) Best of 2021 – 7 Popular Open Source CI/CD Tools
https://devops.com/7-popular-open-source-ci-cd-tools/
Consulté : Avril 2022.

Microsoft (2022). Cloud-Native
https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition
Consulté : Avril 2022

Mohammad Imranur Rahman, Sebastiano Panichella, Davide Taibi (2019). A curated Dataset of Microservices-Based Systems. Joint Proceedings of the Summer School on Software Maintenance and Evolution. Tampere, 2019 Rahman, Mohammad Imranur and Panichella, Sebastiano and Taibi, Davide. "A curated Dataset of Microservices-Based Systems" Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution. CEUR-WS, vol. 2520, pp.1-9. Tampere, Finland. 2019

N. Railić and M. Savić (2021). "Architecting Continuous Integration and Continuous Deployment for Microservice Architecture," 2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH), 2021, pp. 1-5, doi: 10.1109/INFOTEH51037.2021.9400696.

Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Inc.
P. D. Francesco, I. Malavolta and P. Lago, "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption," 2017 IEEE International Conference on Software Architecture (ICSA), 2017, pp. 21-30, doi: 10.1109/ICSA.2017.24.

Richards, M. (2015). Software Architecture Patterns. O'Reilly Media, Inc.

S. Casas, D. Cruz, G. Vidal and M. Constanzo (2021). "Uses and applications of the OpenAPI/Swagger specification: a systematic mapping of the literature," 2021 40th International Conference of the Chilean Computer Science Society (SCCC), 2021, pp. 1-8, doi: 10.1109/SCCC54552.2021.9650408.

S. Kul and A. Sayar (2021). "A Survey of Publish/Subscribe Middleware Systems for Microservice Communication," 2021 5th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT), 2021, pp. 781-785, doi: 10.1109/ISMSIT52890.2021.9604746.

S. Li (2017). "Understanding Quality Attributes in Microservice Architecture," 2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW), 2017, pp. 9-10, doi: 10.1109/APSECW.2017.33.

S. Ma, C. Fan, Y. Chuang, W. Lee, S. Lee and N. Hsueh (2018). "Using Service Dependency Graph to Analyze and Test Microservices," 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), 2018, pp. 81-86, doi: 10.1109/COMPSAC.2018.10207.

Taibi, D (2019). A curated list of Open Source projects developed with a microservices architectural style
V. Raj and S. Ravichandra, "Microservices: A perfect SOA based solution for Enterprise Applications compared to Web Services," 2018 3rd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT), 2018, pp. 1531-1536, doi: 10.1109/RTEICT42901.2018.9012140.
https://github.com/davidetaibi/Microservices_Project_List

Y. Al-Dhuraibi, F. Paraiso, N. Djarallah and P. Merle (2018). "Elasticity in Cloud Computing: State of the Art and Research Challenges," in IEEE Transactions on Services Computing, vol. 11, no. 2, pp. 430-447, 1 March-April 2018, doi: 10.1109/TSC.2017.2711009.

Zdun, U., Navarro, E., Leymann, F. (2017). Ensuring and Assessing Architecture Conformance to Microservice Decomposition Patterns. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (eds) Service-Oriented Computing. ICSOC 2017. Lecture Notes in Computer Science(), vol 10601. Springer, Cham. https://doi.org/10.1007/978-3-319-69035-3_29