UNIVERSITÉ DU QUÉBEC À MONTRÉAL

ASSISTANCE À LA MAINTENANCE ET À L'ÉVOLUTION DES LOGICIELS VIA L'IDENTIFICATION AUTOMATIQUE D'ANTI-PATRONS DANS LES ARCHITECTURES À BASE DE MICROSERVICES

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

SAID AMINE RAFIK TIGHILT

JUIN 2021

REMERCIEMENTS

J'adresse mes remerciements à ma directrice de recherche, Naouel Moha, pour le temps qu'elle m'a accordé, la patience dont elle a fait preuve et l'implication sans failles dans ces travaux.

Je remercie aussi le professeur Yann-Gaël Guéhéneuc pour sa présence, ses conseils et toutes les interrogations qui, sans lui, seraient restées sans réponse.

Je tiens à remercier particulièrement Manel. Sa présence durant toute cette aventure et son aide précieuse ont fait de ce travail, ce qu'il est aujourd'hui.

Je remercie toutes les personnes extraordinaires qui m'ont soutenu et accompagné durant cette maîtrise, mon frère Reda, mon cousin Mehdi, mes amis Raouf, Nail, Karim et Achraf. Sans ces personnes, je n'en serais jamais où j'en suis maintenant.

J'adresse un remerciement particulier à Amina, mon âme sœur, pour tout le stress qu'elle a absorbé et qu'elle m'a permis d'évacuer, pour tous ces mots d'encouragements, et pour tous ces moments qu'elle a su rendre agréables.

Je tiens à remercier aussi tous ceux, que je ne peux citer ici, qui ont contribué à leur façon à ce document.

Pour finir, ce travail est dédié à mes parents, mes modèles et source d'inspiration. Pour tous vos sacrifices, votre soutien et votre éducation : merci *'tichou*, merci *d'da vrah* !

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

ACRONYMES

AMQP **A**dvanced **M**essage **Q**ueuing **P**rotocol.

API **I**nterface de **P**rogrammation d'**A**pplication.

CD **C**ircular **D**ependencies.

ESB **E**ntreprise **S**ervice **B**us.

HE **H**ardcoded **E**ndpoints.

HTTP **H**yperText **T**ransfer **P**rotocol.

IM **I**nsufficient **M**onitoring.

LL **L**ocal **L**ogging.

MARS **M**icroservice **A**ntipatterns **R**esearch **S**oftware.

MC **M**anual **C**onfiguration.

MQTT **M**essage **Q**ueuing **T**elemetry **T**ransport.

MS **M**ega **S**ervice.

MSIH **M**ultiple **S**ervice **I**nstances per **H**ost.

NAG **N**o **A**PI **G**ateway.

NAV **N**o **A**PI **V**ersioning.

NCI **N**o **CI**/CD.

NHC **N**o **H**ealth**C**kech.

NS **N**ano **S**ervice.

ORM **M**apping **O**bjet-**R**elationnel.

POO **P**rogrammation **O**rientée **O**bjet.

REST **RE**presentational **S**tate **T**ransfer.

SL **S**hared **L**ibraries.

SOA **A**rchitecture **O**rientée **S**ervices.

SP **S**hared **P**ersistence.

TO **T**ime**O**uts.

URL **L**ocalisateur **U**niforme de **R**essource.

WC **W**rong **C**uts.

RÉSUMÉ

L'industrie logicielle est présentement en pleine migration depuis les architectures monolithiques vers les architectures à base de microservices qui impliquent des services indépendants, réutilisables et faiblement couplés. Cependant, le manque de compréhension des concepts fondamentaux de ce nouveau type d'architectures peut mener à l'introduction de solutions mal conçues à des problèmes récurrents, aussi appelées « anti-patrons ».

Les anti-patrons peuvent grandement affecter la qualité de service, et entraver la maintenance et l'évolution des systèmes à base de microservices. De ce fait, la spécification et la détection des anti-patrons peuvent aider à l'estimation et à l'évaluation de la qualité de conception dans de tels systèmes.

Plusieurs travaux académiques ont étudié les patrons et anti-patrons dans les systèmes à base de microservices. Cependant, le sujet de la détection automatique des anti-patrons en est toujours à ses prémices. De ce fait, nous proposons MARS (**M**icroservice **A**ntipatterns **R**esearch **S**oftware), une approche outillée entièrement automatisée, supportée par un cadriciel, permettant la détection des anti-patrons dans les systèmes à base de microservices.

En utilisant MARS, nous spécifions et détectons 16 anti-patrons dans 24 systèmes à base de microservices. Nos résultats montrent que MARS peur détecter ces anti-patrons avec une précision moyenne supérieure à 68% et un rappel moyen supérieur à 78%.

Mots-clés : Maintenance logicielle, microservices, architecture logicielle, patrons et anti-patrons, approche outillée, identification automatique d'anti-patrons, architectures orientées services, systèmes à base de microservices.

CHAPITRE I

INTRODUCTION

1.1 Contexte

Les architectures à base de microservices sont un style architectural et une approche de développement d'applications sous forme d'un ensemble de services indépendants et faiblement couplés. Chacun de ces services fonctionne sur son propre processus et communique avec les autres services par le biais de mécanismes légers, souvent des APIs (**I**nterfaces de **P**rogrammation d'**A**pplication) dites REST (**RE**presentational **S**tate **T**ransfer) (Lewis et Fowler, 2015).

Ce type d'architecture connaît actuellement un grand succès dans l'industrie et est adopté par plusieurs acteurs majeurs (Netflix, Amazon, Riot Games, etc.) car il permet le développement de services indépendants, réutilisables et faiblement couplés. Ce type d'architecture est aussi de nature très dynamique et distribuée par essence, ce qui en fait un excellent choix pour une agilité accrue, une efficacité opérationnelle plus importante pour les entreprises, une mise à l'échelle plus simple et des cycles de déploiement plus courts par rapport aux architectures dites « monolithiques » (Newman, 2015). Ces bénéfices sont d'autant plus importants qu'il est admis que la maintenance logicielle est l'une des phases les plus coûteuse en terme de ressources financières et humaines (Hanna, 1993).

Les architectures à base de microservices ont aussi montré être un paradigme très intéressant pour migrer et moderniser des applications monolithiques. En effet, elles permettent aux développeurs de décomposer leurs monolithes en petits services indépendants, où chaque service (1) est géré par une seule équipe, (2) représente une seule fonctionnalité d'affaires et (3) peut être développé, déployé et maintenu sans affecter les autres services du système.

Chaque microservice d'un système à base de microservices ne remplit qu'une seule fonctionnalité métier, maintient ses propres données, fonctionne sur son propre processus, est géré par une seule et même équipe de développement, et n'est pas lié au reste du système pour son évolution, sa maintenance et son déploiement. Les microservices sont faiblement couplés et construits autour des besoins d'affaires. Ils sont déployés via des mécanismes automatisés avec le minimum de gestion centralisée (Newman, 2015).

1.2 Problème

La nature dynamique des architectures à base de microservices ainsi que leur intégration et déploiement continus peuvent mener à l'introduction de problèmes de conception ou d'implémentation. Ces problèmes de conception, qui sont issus de solutions inadéquates à des problématiques récurrentes, sont communément appelés « anti-patrons » (Taibi et Lenarduzzi, 2018). Les anti-patrons peuvent considérablement affecter la maintenance et dégrader la qualité de conception et d'opération d'un système où ils sont présents (Palma, 2013). Ainsi, les architectures à base de microservices, tout comme n'importe quel autre style architectural, font face à des problématiques de maintenance et d'évolution introduites par les anti-patrons.

Contrairement aux anti-patrons du paradigme de programmation « orientée ob-

jet », la spécification et la définition des anti-patrons des systèmes à base de microservices en sont toujours à leurs prémices. Il n'existe que quelques ouvrages et publications académiques qui traitent ce sujet. Ces anti-patrons sont souvent combinés ou confondus avec ceux des architectures orientées services (SOA) et ne sont pas toujours décrits complètement et minutieusement dans la littérature. De plus, malgré la démocratisation et la popularité des architectures à base de microservices, il n'existe à ce jour que très peu d'approches pour détecter automatiquement des occurrences d'anti-patrons dans les systèmes à base de microservices.

1.3 Contribution

À travers nos travaux, nous souhaitons contribuer à la maintenance et l'évolution des systèmes à base de microservices (Tighilt *et al.*, 2019). Dans cette optique, nous proposons (1) une définition générique, concise et consensuelle des anti-patrons de systèmes à base de microservices sous la forme d'un catalogue des anti-patrons les plus importants (Tighilt *et al.*, 2020b) et (2) une approche outillée entièrement automatisée pour l'identification de ces anti-patrons dans les architectures à base de microservices (Tighilt *et al.*, 2020a).

Notre catalogue d'anti-patrons est construit autour de deux aspects principaux : (1) une revue systématique de la littérature relative aux architectures à base de microservices et (2) l'analyse de plusieurs systèmes à code source ouvert, développés sous forme de microservices, afin de mieux comprendre l'implémentation des systèmes à base de microservices et de visualiser concrètement la manifestation des anti-patrons dans des systèmes en production. Le catalogue d'anti-patrons de systèmes à base de microservices fournit une description uniforme et structurée de chaque anti-patron, ainsi que des solutions potentielles pour les corriger.

En se basant sur notre catalogue d'anti-patrons, nous proposons ensuite MARS

(**M**icroservices **A**ntipatterns **R**esearch **S**oftware), une approche outillée entièrement automatisée permettant de déterminer la présence d'occurrences des anti-patrons de notre catalogue dans les systèmes à base de microservices.

La nature des architectures à base de microservices les rend très dynamiques (multi-langages et multi-environnements par exemple (Newman, 2015)), de ce fait, nous avons dû surpasser quelques défis afin d'implementer MARS :

1. Les microservices sont par définition indépendants (Newman, 2015). Les systèmes à base de microservices peuvent être déployés sur plusieurs plate-formes différentes en utilisant une multitude d'outils et de configurations.
2. Les systèmes à base de microservices peuvent être développés en utilisant plusieurs langages de programmation (Lewis et Fowler, 2015). Cela rend le processus de détection d'anti-patrons plus difficile par rapport aux systèmes développés avec un seul langage.

De ce fait, MARS se base sur un méta-modèle indépendant des plate-formes et des langages couvrant l'ensemble des données nécessaires au processus de détection. Cela permet de rendre MARS plus évolutif et plus extensible.

1.4 Organisation du mémoire

Le présent mémoire est structuré de la manière suivante. Le chapitre 2 décrit certains concepts préliminaires associés à notre thématique de recherche et fournit une mise en contexte. Le chapitre 3 est dédié à l'état de l'art réalisé tout au long de notre recherche. Les chapitres 4 et 5 décrivent respectivement notre catalogue d'anti-patrons relatifs aux systèmes à base de microservices ainsi que MARS, notre approche outillée et automatisée pour la détection de ces anti-patrons. Dans le chapitre 6, nous décrivons notre étude empirique sur des systèmes à base de

microservices et nous présentons les résultats obtenus. Nous terminons ce mémoire par une synthèse et une description de travaux futurs possibles.

CHAPITRE II

CONCEPTS PRÉLIMINAIRES

Dans le but de faciliter la compréhension et la lecture de ce mémoire et afin de clarifier certains aspects de notre problématique de recherche, nous avons jugé indispensable de revenir sur certains principes fondamentaux liés à notre thématique d'étude (afin de comprendre les notions préliminaires qui y sont associées).

Ce chapitre commence par détailler les principes de base des architectures à base de microservices, puis illustre certaines différences entre les architectures à base de microservices et les architectures orientées services (SOA). Enfin, il détaille certaines technologies et méthodes usuellement utilisées dans les architectures à base de microservices afin de résoudre des problématiques récurrentes.

2.1 SOA vs. architectures à base de microservices

Les architectures à base de microservices sont pour beaucoup issues de l'architecture orientée services. En effet, plusieurs sources, e.g., (Zimmermann, 2016; Lewis et Fowler, 2015), les considèrent comme descendant direct de SOA.

Les architectures à base de microservices et les SOA se basent toutes les deux sur le principe de « service » comme unité de base. Cependant, il existe un certain nombre de différences notoires qui font des architectures à base de microservices

un style architectural à part entière et un sujet de recherche en pleine expansion.

Les principales différences entre ces deux styles architecturaux sont ci-après.

D'abord, les architectures SOA prônent le partage et la réutilisation de fonctionnalités au sein de leur code source, tandis que les architectures à base de microservices sont des architectures qui découragent le partage de code et de fonctionnalités à travers le principe de « contexte borné ».

Ensuite, la gouvernance dans les architectures SOA est centralisée, et les équipes de développement sont homogènes (responsables des bases de données, responsables des fonctionnalités métier ou responsables de la présentation, par exemple) contrairement aux architectures à base de microservices où la gouvernance est complètement décentralisée et où chaque équipe dispose de toutes les compétences nécessaires au développement, au déploiement et à la maintenance du service.

Enfin, les architectures SOA utilisent des *Enterprise Service Buses* (ESB) pour communiquer, et supportent plusieurs protocoles de messagerie, alors que les architectures à base de microservices se basent sur des protocoles très légers tels que les API REST et les protocoles de messagerie légers tels que MQTT (**M**essage **Q**ueuing **T**elemetry **T**ransport) ou AMQP (**A**dvanced **M**essage **Q**ueuing **P**rotocol).

Il est important de noter aussi que l'automatisation (de la configuration, des tests ou du déploiement), bien que de plus en plus populaire dans les SOA, n'en est pas un principe fondamental contrairement aux architectures à base de microservices.

2.2 Les architectures à base de microservices

Le terme « microservice » est apparu pour la première fois en 2011 lors d'un atelier organisé par un groupe d'architectes logiciels tenu à Venise. Ce terme visait à décrire ce que les participants considéraient comme un style architectural qu'ils

commençaient à explorer. Ce même groupe décida en 2012 que le terme « microservices » (au pluriel) serait un terme plus approprié à la description de ce style architectural (Lewis et Fowler, 2015).

Le but premier des architectures à base de microservices tel que discuté initialement est de permettre le remplacement ou l'évolution d'un composant d'un système logiciel sans impliquer ses composants collaborateurs. De ce fait, les microservices répondent à un certain nombre de principes de base (Lewis et Fowler, 2015) énumérés ci-dessous :

1. **Composition via des services :** un composant d'un système à base de microservices doit être une unité de base indépendamment remplaçable et évolutive.
2. **Articulés autour de fonctionnalités d'affaires :** Il est nécessaire dans les architectures à base de microservices de décomposer le système autour des fonctionnalités d'affaires et non autour des couches techniques de développement.
3. **Livraison de produits et non de projets :** En suivant ce principe, chaque microservice reste sous la responsabilité d'une seule et même équipe au cours de tout son cycle de vie (développement, déploiement, maintenance, évolution, etc.)
4. **Smart endpoints and dumb pipes**[1] **:** Dans les architectures à base de microservices, il est nécessaire de se focaliser sur les microservices eux-mêmes plutôt que sur leurs moyens de communication.
5. **Gouvernance décentralisée :** Dans les architectures à base de microservices, chaque microservice doit être géré par une seule et même équipe, qui a libre choix sur les technologies adaptées à son microservice.

1. La traduction de ce terme ne rend pas justice à l'idée de fond.

6. **Gestion décentralisée des données :** Dans la même logique que la gouvernance décentralisée, chaque microservice d'un système à base de microservices doit gérer ses propres données. Cela inclut la possession de ses propres bases de données et de ne pas les partager avec d'autres microservices.
7. **Automatisation de l'infrastructure :** Dans les architectures à base de microservices, tout ce qui peut être automatisé doit être automatisé. Cela inclut la configuration, les tests et le déploiement.
8. **Conçus pour échouer :** Les microservices d'un système à base de microservices doivent être développés de sorte à tolérer les pannes et à ne pas influencer le reste du système. Un microservice ne doit pas propager une panne et les microservices collaborant doivent répondre à cette panne aussi gracieusement que possible.
9. **Conception évolutive :** Les microservices doivent être conçus de sorte à permettre leur évolution constante et à toujours garder le contrôle sur d'éventuels changements dans le système. Cela offre la possibilité aux développeurs de déployer rapidement de nouvelles fonctionnalités sans devoir s'inquiéter des répercussions sur les autres microservices.

2.3 Technologies et outils dans les microservices

L'arrivée des architectures à base de microservices a soulevé un certain nombre de problématiques jusque-là inconnues ou peu importantes. En effet, la nature indépendante et volatile des microservices rend difficiles des tâches telles que la communication, la coordination, la configuration ou le déploiement. De ce fait, plusieurs outils, technologies et méthodes ont émergé afin de proposer des solutions et des implémentations répondant aux besoins particuliers des architectures à base de microservices.

Certains de ces mécanismes sont devenus des standards dans les architectures à base de microservices, notamment ceux exposés dans le reste de cette section.

2.3.1 Intégration et déploiement continus (CI / CD)

L'automatisation est au coeur des architectures à base de microservices, et ce, à tous les niveaux et à toutes les étapes du cycle de vie (développement, déploiement, maintenance, etc.). C'est pour cela que les outils d'intégration et de déploiement continus sont indispensables aujourd'hui dans la plupart des systèmes à base de microservices.

L'intégration continue permet aux développeurs d'apporter régulièrement des changements au code source de leurs systèmes à base de microservices, de les tester, puis de les fusionner (et donc de les intégrer) au code source déjà existant dans un dépôt partagé.

Le déploiement continu quant à lui, est une continuité du processus d'intégration continue, puisqu'il consiste à déployer automatiquement les changements apportés au code source vers les environnements de production, en s'assurant que le code mis à jour passe avec succès les tests prédéfinis par les développeurs.

2.3.2 Passerelle d'API (API Gateway)

Il est courant qu'un système à base de microservices dispose de plusieurs interfaces (interface Web, application mobile Android, application mobile iOS, etc.). Une passerelle d'API permet à toutes ces interfaces de n'avoir qu'un seul point d'entrée vers le système, et abstrait ainsi le découpage interne du système.

La passerelle d'API se place entre les applications consommatrices du système (*frontend*) et le système à base de microservices lui même (*backend*). La figure 2.1

illustre le positionnement d'une passerelle d'API.

En utilisant une passerelle d'API, l'authentification, les autorisations, la validation des requêtes et bien d'autres aspects relatifs aux performances ou à la sécurité notamment se trouvent facilités puisqu'ils peuvent êtres gérés depuis un seul endroit.

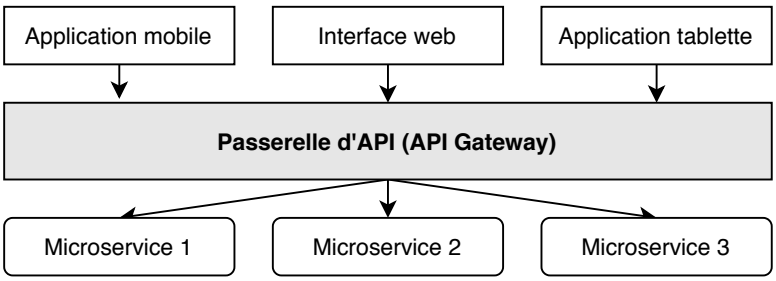


Figure 2.1: Passerelle d'API

2.3.3 Disjoncteurs (Circuit breakers)

L'un des principes fondamentaux des microservices est d'être conçu pour échouer. Cela veut dire qu'une panne ou un microservice hors d'état de service ne doit pas impacter le reste du système.

Une panne dans un microservice peut avoir une durée indéterminée, ou causer l'utilisation intensive de ressources dans le cas où les autres microservices ne sont pas au courant de cette panne. De ce fait, il est important d'implémenter un mécanisme permettant d'interrompre temporairement les requêtes vers le microservice hors d'état de service, et ce, jusqu'à la résolution de la panne ou le rétablissement de la connexion. C'est dans ce contexte qu'intervient le concept de disjoncteur, puisqu'il agit de la même manière qu'un disjoncteur dans un circuit électrique.

Lorsque le système est en état de fonctionnement, le disjoncteur se trouve dans un état dit « fermé », et les requêtes sont acheminées normalement entre les microser-

vices. Cependant, si un problème survient dans l'un des microservices, le disjoncteur se met dans un état « ouvert » et renvoie systématiquement une exception lors d'une requête vers le microservice problématique. Un état intermédiaire dit « entre-ouvert » se met en place au bout d'un certain temps après une erreur, cet état a pour but de vérifier si la panne est toujours présente. En laissant passer une seule requête, le disjoncteur vérifie le succès de celle-ci pour se mettre en état « fermé ». Si par contre, la requête échoue, il se remet dans l'état « ouvert ».

2.3.4 Conteneurs (Containers)

Le déploiement des microservices est une tâche difficile et constitue un défi en soi. En effet, les microservices peuvent utiliser plusieurs types de technologies et nécessiter des infrastructures très différentes. De ce fait, le déploiement traditionnel sur un simple serveur relève de l'impossible, et l'utilisation de machines virtuelles devient très coûteux en utilisation de ressources.

Afin de résoudre cette problématique, les conteneurs permettent de partager le système d'exploitation et l'infrastructure physique d'un système, mais d'isoler complètement leur propre écosystème. De ce fait, il est beaucoup plus facile de déployer plusieurs conteneurs sur un même système d'exploitation que de devoir embarquer ce dernier sur une multitude de machines virtuelles, ce qui en fait un choix léger et flexible pour les développeurs.

Un autre avantage des conteneurs est leur consistance et leur fiabilité. En effet, il n'est plus nécessaire de déplacer ou de dupliquer la pile technologique entre le poste du développeur et l'environnement de production puisque les conteneurs peuvent être déployés tels quels, incluant l'écosystème sous-jacent et le code source.

2.3.5 Découverte de services (Service Discovery)

Dans les architectures à base de microservices, les microservices se voient très souvent assigner des adresses et des numéros de ports dynamiques, rendant très difficile voire impossible de les connaître à l'avance. Il existe toutefois un mécanisme permettant de remédier à cette problématique : la découverte de services.

Ce mécanisme permet la communication entre les microservices sans connaître à l'avance leurs adresses. Son principe de fonctionnement consiste à faire enregistrer les informations de connexion dans un registre par les microservices eux-mêmes. Ainsi, lorsqu'un autre microservice veut communiquer, il devient aisé de consulter le registre pour découvrir l'adresse de destination.

Il existe deux sortes de découverte de services : (1) découverte côté client (illustrée sur la figure 2.2a), et (2) découverte côté serveur (illustrée sur la figure 2.2b). Dans la découverte côté client, le client envoie une requête au registre, puis choisit une instance de service disponible et enfin, procède à sa requête. Dans la découverte côté serveur, le client envoie sa requête au registre à travers un « routeur » qui se charge de choisir une instance de service disponible afin de transmettre la requête.

(a) Découverte de services côté client (b) Découverte de services côté serveur

Figure 2.2: Découverte de services

CHAPITRE III

ÉTAT DE L'ART

De nombreuses études relatives à l'étude des anti-patrons ont été réalisées par le passé, et ce, dans plusieurs domaines et pour plusieurs paradigmes de programmation. DECOR (Moha *et al.*, 2010) par exemple permet d'identifier quatre anti-patrons du paradigme de programmation orienté objet. Paprika (Hecht *et al.*, 2015) et aDoctor (Di Nucci *et al.*, 2017) quant à eux offrent la possibilité de détecter des anti-patrons dans les applications mobiles Android. Compte tenu du nombre considérable de travaux sur la détection d'anti-patrons en général, nous nous focaliserons dans le présent état de l'art spécifiquement sur les travaux traitant des anti-patrons dans les systèmes à base de microservices.

Les architectures à base de microservices ainsi que les bonnes et mauvaises pratiques qui leurs sont associées sont des sujets d'actualité qui suscitent un grand intérêt pour la communauté de chercheurs en génie logiciel et provoquent un engouement certain dans le monde de l'industrie logicielle. En effet, depuis l'apparition des architectures à base de microservices, un bon nombre de travaux traitant de multiples axes de ces architectures ont été publiés, notamment relatifs aux anti-patrons, nous alimentant ainsi en ressources académiques, et posant des bases solides pour nos travaux.

Le fait que les architectures à base de microservices soient un sujet de recherche

plutôt récent comparativement à la programmation orientée objet ou aux architectures orientées services, par exemple, pousse les chercheurs à d'abord organiser la recherche et à clarifier l'état de l'art. Nous pouvons citer Zimmerman qui, dans son papier d'aperçu et de vision (Zimmermann, 2016), tente de passer en revue les différentes définitions des microservices afin d'en identifier les principes. Il compare ensuite les définitions des microservices et celles des architectures orientées service et en conclut que les microservices sont une implémentation spéciale du paradigme SOA.

Aussi, Pahl et Jamdi (Pahl et Jamshidi, 2016) ont cherché dans leurs travaux à identifier, classifier en une taxonomie et comparer systématiquement le corpus académique disponible sur les microservices et sur leur utilisation dans l'informatique infonuagique. Ils ont proposé dans ce papier une cartographie systématique de 21 publications académiques publiées entre 2014 et 2016 en définissant et en utilisant un cadriciel de caractérisation. Dans leur contribution, ils rapportent un manque d'outils d'étude supportant les systèmes à base de microservices et concluent que la recherche dans ce domaine en est toujours à ses prémices. Cette étude amène aussi une discussion des préoccupations architecturales des microservices et les positionne plus proche des technologies infonuagiques et de conteneurs.

3.1 Patrons et pratiques dans les architectures à base de microservices

Depuis leur démocratisation, les architectures à base de microservices sont sujettes à une multitude d'études relatives aux bonnes et mauvaises pratiques de leur conception et de leur développement. En effet, comme dans n'importe quel autre style architectural, plusieurs problématiques ont vu le jour à travers l'utilisation des architectures à base de microservices. Des technologies, méthodes et techniques sont apparues pour les résoudre. Plusieurs chercheurs se sont intéressés

aux patrons et anti-patrons dans les architectures à base de microservices, notamment Marquez et Astudillo (Marquez et Astudillo, 2018), qui ont étendu leurs travaux préalablement réalisés avec Osses (Osses *et al.*, 2018) afin de déterminer si des patrons architecturaux sont utilisés dans les systèmes à base de microservices. Ils proposent dans ces travaux (1) un catalogue de patrons architecturaux dans les systèmes à base de microservices issu des travaux académiques et de l'industrie, (2) une corrélation entre des attributs qualitatifs et ces patrons, (3) une liste de technologies et de méthodologies utilisées pour la conception de systèmes à base de microservices dans le respect des patrons identifiés et (4) une analyse comparative entre les SOA et les architectures à base de microservices.

Taibi et al. (Taibi *et al.*, 2019) quant à eux ont introduit un catalogue et une taxonomie des anti-patrons les plus communs dans les systèmes à base de microservices résultant de la migration depuis des architectures monolithiques vers des architectures à base de microservices. Afin de construire leur catalogue, les auteurs ont procédé à des entrevues auprès de 27 développeurs. Ils ont identifié une taxonomie de 20 anti-patrons, incluant des anti-patrons organisationnels et techniques. Ils ont aussi estimé le degré de nocivité de chaque anti-patron à travers un questionnaire. Les travaux de Taibi et al. concluent que la décomposition est la problématique la plus critique dans la migration d'un système monolithique vers un système à base de microservices.

Soldani et al. (Soldani *et al.*, 2018) de leur côté ont identifié et comparé les bénéfices et les limitations des architectures à base de microservices en étudiant la littérature industrielle. Ils ont aussi étudié les pratiques de conception et de développement de microservices afin de rapprocher le monde académique et le monde industriel sur cette thématique.

Garriga (Garriga, 2018) a défini dans ses travaux un cadriciel d'analyse prélimi-

naire des systèmes à base de microservices sous forme d'une taxonomie de concepts incluant l'ensemble du cycle de vie des microservices, ainsi que les aspects organisationnels. L'auteur décrit ce cadriciel comme nécessaire afin de rendre plus efficaces l'exploration, la compréhension, l'évaluation, la comparaison et la sélection des modèles, langages, techniques, plate-formes et outils dans les systèmes à base de microservices. Il analyse ensuite les approches de l'état de l'art relatives au développement de systèmes à base de microservices en utilisant cette taxonomie de concepts, et propose une perspective sur les solutions disponibles. De plus, cet article identifie un certain nombre de défis afin d'alimenter la recherche à partir d'analyses de la littérature.

3.2 Détection d'anti-patrons dans les architectures à base de microservices

La recherche sur les bonnes et mauvaises pratiques liées au développement de systèmes à base de microservices est de plus en plus riche et liste de plus en plus de pratiques et de recommandations. Cependant, une problématique majeure a vu le jour au sein de la communauté de chercheurs du domaine : comment spécifier et détecter automatiquement ces pratiques dans un système à base de microservices en production ou en cours de développement ? La réponse à cette question permettrait certainement d'accélérer les processus de développement, de faciliter le déploiement et la gestion, et de réduire les coûts associés à la maintenance et à l'évolution des systèmes à base de microservices. C'est ainsi que les premières études s'intéressant à la détection automatique d'anti-patrons dans les systèmes à base de microservices sont apparues.

Borges and Khan (Borges et Khan, 2019) dans leurs travaux ont sélectionné cinq anti-patrons bien connus dans la littérature, et relatifs aux systèmes à base de microservices, et ont proposé un algorithme pour les détecter. Leur algorithme se

base sur deux métriques que les auteurs ont définies : `Closeness centrality` et `Betweenness centrality`. Les auteurs pensent que leur solution peut permettre d'éviter plusieurs erreurs communes lors du déploiement de système à base de microservices et peut aider les gestionnaires de projets à avoir une vision plus générale de leurs systèmes. Les auteurs ont testé leur algorithme sur un système à base de microservices connu et ont discuté des limitations de leur approche, en révélant des améliorations potentielles.

Walker et al. (Walker *et al.*, 2020) se sont intéressés à la détection automatique des défauts de code dans les systèmes à base de microservices. L'approche proposée par les auteurs permet de détecter onze défauts de code en se basant sur la reconstruction d'architecture logicielle (Rademacher *et al.*, 2020; Alshuqayran *et al.*, 2018; Granchelli *et al.*, 2017). Les microservices dans ces travaux sont d'abord analysés individuellement, puis regroupés afin de construire un réseau sur lequel l'analyse sera réalisée. Les auteurs ont testé leur approche sur deux systèmes à base de microservices et ont montré qu'il était possible de détecter automatiquement des défauts de code dans les systèmes à base de microservices en utilisant l'analyse statique de code.

Dans des travaux qui s'intéressent à la même problématique, Pigazzini et al. (Pigazzini *et al.*, 2020) proposent de détecter automatiquement trois défauts de code dans les systèmes à base de microservices en proposant un produit minimum viable permettant de tester et de valider leur approche. Les auteurs se basent dans leurs travaux sur un outil déjà existant permettant la détection de défauts architecturaux appelé `Arcan`, qu'ils ont fait évoluer pour y inclure leurs stratégies de détection. L'approche des auteurs a été validée sur cinq systèmes à base de microservices, sans toutefois calculer de valeurs de précision ni de rappel. Les auteurs visaient à démontrer la possibilité de détecter automatiquement des défauts de code dans les systèmes à base de microservices.

Dans ce chapitre, nous avons présenté de multiples travaux académiques traitant des différents axes de notre travail de recherche. Nous avons présenté des travaux traitant des patrons et anti-patrons dans les systèmes à base de microservices, ainsi que des travaux relatifs à l'identification et la détection automatiques de ces anti-patrons dans les systèmes à base de microservices.

Nous constatons qu'à ce jour très peu de travaux se sont intéressés à la détection automatique d'anti-patrons dans les systèmes à base de microservices. Notre travail de recherche a pour but de consolider les études déjà existantes et de fournir une nouvelle approche dans le domaine de la détection automatique d'anti-patrons dans les systèmes à base de microservices.

CHAPITRE IV

ANTI-PATRONS DE MICROSERVICES

4.1 Méthodologie de construction

Notre approche de détection d'anti-patrons dans les systèmes à base de microservices repose sur deux phases. La première phase consiste à spécifier et définir un catalogue de patrons et anti-patrons dans les systèmes à base de microservices. (Tighilt *et al.*, 2020b) présente les travaux qui nous ont permis d'aboutir au catalogue détaillé dans ce chapitre.

Comme illustré sur la figure 4.1, la première phase de nos travaux (objectif 1) prend en entrée les publications académiques constituant l'état de l'art, afin d'en extraire les anti-patrons les plus commun, et une liste de systèmes à base de microservices à code source ouvert afin d'en extraire les pratiques industrielles les plus courantes. Cela nous permet d'extraire une taxonomie des anti-patrons et une liste de pratiques industrielles, de les regrouper, de les homogénéiser et de les présenter sous forme de catalogue uniforme.

Afin de proposer notre catalogue d'anti-patrons pour les systèmes à base de microservices, nous avons effectué une revue systématique de la littérature. Nous avons étudié un total de 27 travaux relatifs à l'étude de patrons et d'anti-patrons de microservices et généré un catalogue de 16 anti-patrons. La méthodologie de cette

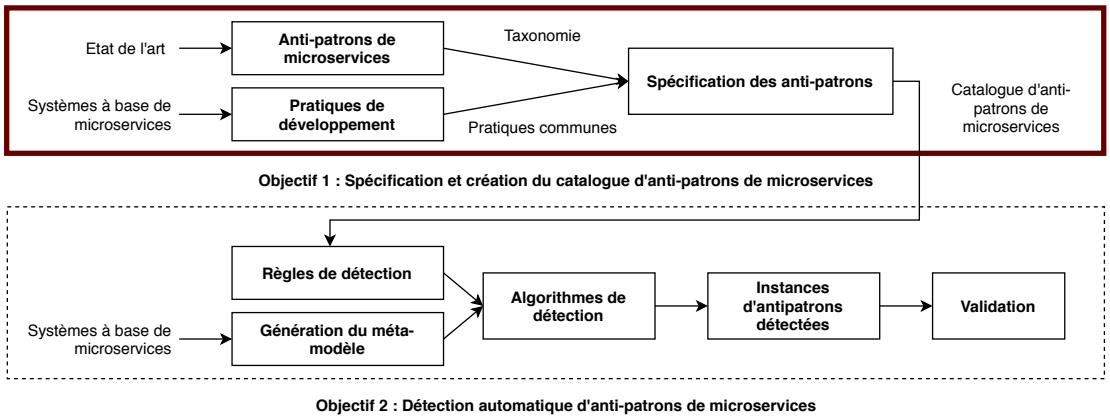


Figure 4.1: Méthodologie des travaux - Objectif 1

revue systématique de littérature ainsi que le catalogue sont décrits dans les sections suivantes. Nous pensons que cette méthodologie permet de répliquer notre étude dans les mêmes conditions et peut être étendue pour des études futures visant à étudier d'autres patrons et anti-patrons.

4.1.1 Revue systématique de littérature

Nous avons suivi la procédure décrite par Kitchenham et al. (Kitchenham, 2004) pour réaliser une revue systématique de littérature. Nous avons premièrement collecté plusieurs publications académiques selon des mots-clés bien définis qui sont reliés à notre sujet d'étude. Nous avons commencé par l'identification des mots-clés pertinents relatifs aux architectures à base de microservices. Ensuite, pour chaque mot-clé, nous avons établi une liste de termes semblables ou synonymes en utilisant « *Synonym Finder* », un outil en ligne [1] permettant d'obtenir des synonymes pour un mot donné. Notre liste de mots-clés nous a permis de définir les termes de recherche suivants à utiliser pour trouver des publications académiques pertinentes :

1. https ://www.synonym-finder.com/

> (microservice OU micro-service OU service) ET (antipattern OU anti-pattern OU bad smell OU pitfall OU poor design OU code smell OU bad practice)

Une fois notre requête de recherche établie, nous l'avons exécutée sur plusieurs moteurs de recherches scientifiques : ACM Digital Library, Engineering Village, Google Scholar, IEEE Xplore Digital Library. Cette recherche a retourné un total de 1,195 références.

Nous avons ensuite trié et filtré les publications obtenues selon (1) le titre, (2) le résumé et (3) le contenu de la publication. Deux personnes ont indépendamment et manuellement analysé la liste des publications obtenues, puis ont discuté des potentielles différences afin d'arriver à un consensus. Nous avons exclu de notre liste initiale les publications répondant à un ou plusieurs des critères ci-dessous :

- Publications écrites dans une langue autre que l'anglais.
- Publications non relatives aux microservices.
- Publications non relatives aux anti-patrons dans les systèmes à base de microservices.

De ce fait, nous avons réduit notre liste initiale à un total de 21 publications académiques relatives à la conception des systèmes à base de microservices.

Enfin, nous avons appliqué la technique de « *forward and backward snowballing* » (Wohlin, 2014; Felizardo *et al.*, 2016) afin de minimiser les risques de manquer des publications importantes. Le « *forward snowballing* » désigne l'utilisation de la bibliographie des publications académiques afin d'identifier de nouvelles publications. Le « *backward snowballing* » consiste à identifier les nouvelles publications citant les publications déjà identifiées.

Nous avons itéré ce processus et appliqué nos critères d'exclusion sur chaque nou-

velle publication identifiée. Nous avons arrêté le processus d'itération lorsque nous n'étions plus en mesure de trouver de nouvelles publications. Au total, nous avons réalisé cinq itérations et ajouté six nouvelles publications à notre liste initiale. De ce fait, nous avons obtenu 27 publications académiques décrivant les patrons et anti-patrons des système à base de microservices présentées dans le tableau 4.1.

Référence	**Titre**
(Newman, 2015)	Building Microservices : Designing Fine-Grained Systems
(Pahl et Jamshidi, 2016)	Microservices : A Systematic Mapping Study
(Garriga, 2018)	Towards a Taxonomy of Microservices Architectures
(Soldani *et al.*, 2018)	The Pains and Gains of Microservices : A Systematic Grey Literature Review
(Marquez et Astudillo, 2018)	Actual Use of Architectural Patterns in Microservices-Based Open Source Projects
(Osses *et al.*, 2018)	Exploration of Academic and Industrial Evidence about Architectural Tactics and Patterns in Microservices
(Salah, 2016)	Microservices Antipatterns
(Bogard, 2017)	Avoiding Microservices Megadisaster
(Shadija *et al.*, 2017)	Towards an Understanding of Microservices
(Taibi *et al.*, 2019)	Microservices Anti Patterns : A Taxonomy
(Taibi *et al.*, 2018)	Architectural Patterns for Microservices : A Systematic Mapping Study
(Taibi et Lenarduzzi, 2018)	On the Definition of Microservice Bad Smells
(Neri *et al.*, 2019)	Design Principles, Architectural Smells and Refactorings for Microservices : A Multivocal Review
(Bogner *et al.*, 2019)	Towards a Collaborative Repository for the Documentation of Service-Based Antipatterns and Bad Smells

(Carrasco *et al.*, 2018)	Migrating Towards Microservices : Migration and Architecture Smells
(Alshuqayran *et al.*, 2016)	A Systematic Mapping Study in Microservice Architecture
(Balalaie *et al.*, 2018)	Microservices Migration Patterns
(Carnell, 2017)	Spring Microservices in Action
(Di Francesco *et al.*, 2019)	Architecting with Microservices : A Systematic Mapping Study
(Di Francesco *et al.*, 2017)	Research on Architecting Microservices : Trends, Focus, and Potential for Industrial Adoption
(Dragoni *et al.*, 2017)	Microservices : Yesterday, Today, and Tomorrow.
(Ghofrani et Lübke, 2018)	Challenges of Microservices Architecture : A Survey on the State of the Practice.
(Nadareishvili *et al.*, 2016)	Microservice Architecture : Aligning Principles, Practices, and Culture
(Pautasso *et al.*, 2017)	Microservices in Practice, Part 1 : Reality Check and Service Design
(Richards, 2016)	Microservices Antipatterns and Pitfalls
(Stocker *et al.*, 2018)	Interface Quality Patterns — Communicating and Improving the Quality of Microservices APIs.
(Wolff, 2016)	Microservices : Flexible Software Architecture

Tableau 4.1: Publications académiques relatives à la conception des systèmes à base de microservices

4.1.2 Revue de systèmes à code source ouvert

En plus de notre revue de la littérature, et afin de mieux comprendre les anti-patrons de systèmes à base de microservices, nous avons analysé manuellement

67 systèmes disponible en code source ouvert [2] afin d'identifier les potentielles violations des bonnes pratiques de conception, et donc de probables anti-patrons.

Nous avons cherché à déterminer les occurrences de chaque anti-patron de notre catalogue dans le code source de ces systèmes et nous avons examiné leurs manifestations. Nous avons ensuite décrit textuellement les symptômes et indices nous ayant permis d'identifier les occurrences de chaque anti-patron afin d'étoffer notre catalogue. Cette analyse nous a aussi permis d'identifier des solutions concrètes de refactorisation des anti-patrons et des pratiques à appliquer pour prévenir l'anti-patron détecté.

À la suite de la revue de littérature et de l'étude de systèmes à base de microservices à code source ouvert, nous avons établi un catalogue de 16 anti-patrons. Nous avons tenté de généraliser la définition et la description de chaque anti-patron, de décrire ses symptômes d'apparition ainsi que les solutions potentielles pour les prévenir.

La liste des anti-patrons constituant le catalogue est décrite ci-après.

---

2. `https://github.com/davidetaibi/Microservices_Project_List`

4.2 Catalogue

Cette section présente les anti-patrons constituant notre catalogue d'anti-patrons dans les systèmes à base de microservices. Pour chaque anti-patron, afin de proposer un catalogue uniforme, les informations suivantes sont décrites :

- **Aussi connu comme :** D'autres noms sous lesquels un anti-patron apparaît dans la littérature.
- **Nom de la solution :** Le nom de la méthodologie, technique ou outil permettant de retirer ou de minimiser l'impact de l'anti-patron.
- **Contexte :** Le contexte général dans lequel cet anti-patron se manifeste.
- **Forme générale :** L'aspect le plus commun que prend un anti-patron.
- **Symptômes :** Indices permettant de déduire la potentielle présence de cet anti-patron dans un système.
- **Conséquences :** Les impacts potentiels de la présence d'un anti-patron dans le système.
- **Solution refactorisée :** Détails concernant l'implémentation d'une solution à un anti-patron.
- **Avantages de la refactorisation :** Problématiques résolues et avantages apportés par la suppression de l'anti-patron.

4.2.1 Wrong Cuts (WC)

Aussi connu comme : Architecture de microservices en couches
Nom de la solution : Décomposer autour de fonctionnalités d'affaires

Contexte : Un microservice devrait encapsuler un groupe de fonctionnalités pour fournir une seule et unique fonctionnalité d'affaires. Un microservice doit être géré, développé et déployé par une seule équipe au sein de l'organisation. De ce fait, un microservice doit être axé autour des fonctionnalités d'affaire et ne répondre qu'à

un seul besoin.

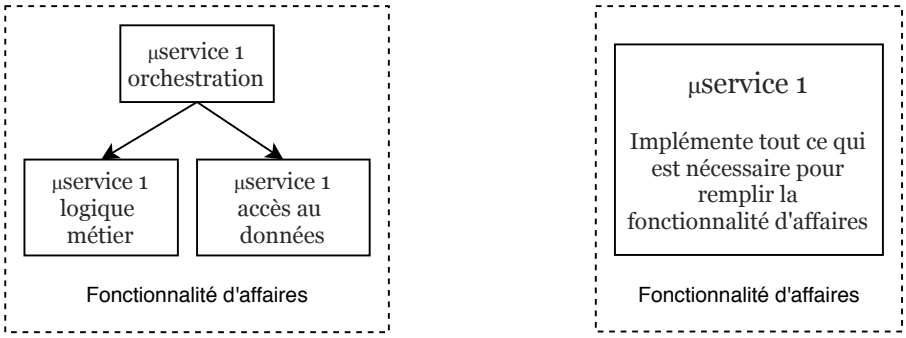
Forme générale : Le système à base de microservices est développé en suivant une architecture en couches, en suivant une décomposition technique (couche de présentation, couche métier et couche de données) comme sur la figure 4.2a.

Symptômes : Certains des aspects suivants peuvent indiquer la présence d'un découpage erroné dans un système à base de microservices : (1) les microservices du système sont fortement couplés entre eux, (2) des appels récurrents entre plusieurs microservices, (3) la présence de microservices uniquement responsables de la présentation, (4) la présence de microservices uniquement responsables de la communication avec les bases de données, ou (5) des dépendances de déploiement.

Conséquences : La livraison de fonctionnalités d'affaires devient plus difficile car elle implique plusieurs équipes qui doivent se coordonner. Les microservices deviennent aussi fortement couplés et de ce fait, rendent la détection et la correction de bogues plus difficiles.

Solution refactorisée : Les microservices doivent être décomposés en concordance avec les fonctionnalités d'affaires et les équipes au sein de l'organisation. Chaque microservice doit représenter une entité d'affaire atomique et implémenter toutes les caractéristiques lui permettant d'offrir cette fonctionnalité (voir la figure 4.2b).

Avantages de la refactorisation : La décomposition en fonctionnalités d'affaires permet à chaque équipe de l'organisation de se concentrer sur une seule responsabilité. Les microservices deviennent alors plus simples à maintenir, les déploiements peuvent être plus fréquents et les frontières entre les services deviennent plus claires.



(a) Microservices en couches (b) Microservice en fonctionnalité d'affaires

Figure 4.2: Découpage erroné (a) vs. bon découpage (b)

4.2.2 Circular Dependencies (CD)

Aussi connu comme : N/A

Nom de la solution : Regrouper les microservices interdépendants

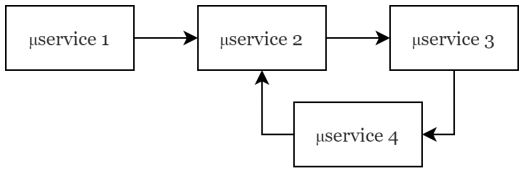
Contexte : Un microservice est une unité de base entièrement indépendante et communiquant avec les autres microservices par le biais de protocoles de communications légers (ex. les API REST). Cette indépendance a pour but d'éviter de tomber dans le piège du « monolithe distribué » (Taibi et Lenarduzzi, 2018).

Forme générale : Plusieurs microservices sont dépendants les uns des autres de façon circulaire, créant ainsi une boucle comme illustré sur la figure 4.3a.

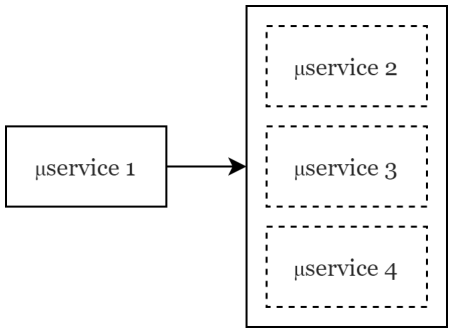
Symptômes : Les dépendances circulaires peuvent se manifester par le biais de l'un des symptômes suivants : (1) appels directs entre les microservices ou (2) communications fréquentes entre les microservices.

Conséquences : Les microservices ne sont plus indépendants les uns des autres car le déploiement d'un microservice nécessitera le déploiement des microservices qui lui sont couplés. Aussi, un dysfonctionnement de l'un des microservices causera fatalement un dysfonctionnement dans les autres microservices.

Solution refactorisée : Il est possible de corriger cet anti-patron en rassemblant en un même service tous les microservices qui dépendent les uns des autres de manière circulaire (voir la figure 4.3b).



(a) Dépendances circulaires entre microservices



(b) Microservices interdépendants regroupés

Figure 4.3: Dépendances circulaires vs. microservices regroupés

Avantages de la refactorisation : Regrouper les microservices permet d'éviter les dépendances, et transforme les dépendances circulaires en une seule unité plus simple à maintenir et déployable de façon indépendante.

4.2.3 Mega service (MS)

Aussi connu comme : N/A

Nom de la solution : Décomposer autour de fonctionnalités d'affaires

Contexte : Les microservices doivent être les unités de base d'un système à base de microservices et être totalement indépendants. Le déploiement des microservices doit se faire en totale autonomie (sans dépendre d'autres microservices) et ils ne doivent répondre qu'a un seul besoin d'affaires (Lewis et Fowler, 2015).

Forme générale : Un microservice d'un système à base de microservices qui répond à plusieurs besoins d'affaires.

Symptômes : Un mega service potentiel a un nombre de lignes de code et un nombre de fichiers relativement plus élevés que le reste des microservices composant le système à base de microservices global.

Conséquences : Avoir un mega service dans son système à base de microservices génère les mêmes inconvénients qu'un système monolithique. Il peut provoquer des problèmes de maintenance, réduire les performances du système, rendre les procédures de test plus complexes, et tout cela, en plus de la couche de complexité apportée par les architectures à base de microservices elles-mêmes.

Solution refactorisée : Pour éviter cet anti-patron, il est nécessaire de décomposer le ou les microservice(s) incriminé(s) en plus petites unités répondant chacune à un besoin d'affaires unique.

Avantages de la refactorisation : Décomposer un mega service en plus petits microsevices permet de simplifier son processus de développement, de déploiement et de maintenance. Cela permet aussi d'isoler les fonctionnalités d'affaires dans des microservices clairement définis.

4.2.4 Nano service (NS)

Aussi connu comme : Casser la tirelire (*Break the piggy bank*)
Nom de la solution : Décomposer autour de fonctionnalités d'affaires

Contexte : Refactoriser un monolithe en un système à base de microservices est une tâche très complexe. Les microservices doivent répondre à un seul besoin d'affaires, et la décomposition doit être étudiée et réalisée minutieusement afin d'éviter les nano services.

Forme générale : Un système est décomposé en un grand nombre de microservices.

Symptômes : L'anti-patron nano service est potentiellement présent lorsque (1) le système à base de microservices a un nombre très élevé de microservices, (2) les microservices d'un système échangent beaucoup d'informations entre eux ou (3) le système contient des dépendances circulaires.

Conséquences : Lorsque la décomposition vers un système à base de microservices est réalisée sans prendre en considération le domaine d'affaires ou la granularité, les services finissent souvent par être fortement couplés et doivent être déployés et maintenus ensemble.

Solution refactorisée : Décomposer le système en se basant sur les fonctionnalités d'affaires et considérer les microservices comme une unité atomique de base pour chaque entité d'affaires.

Avantages de la refactorisation : Refactoriser les nano services augmente l'indépendance des microservices et permet d'accroître l'efficacité générale du système, de faciliter la maintenance, et d'améliorer la durabilité d'un système à base de microservices.

4.2.5 Shared Libraries (SL)

Aussi connu comme : On m'a appris à partager (*I was taught to share*)
Nom de la solution : Partager aussi peu que possible

Contexte : Les architectures à base de microservices sont composés de microservices indépendants (Richards, 2016). De ce fait, les microservices ne devraient pas partager de code ou de bibliothèques entre eux.

Forme générale : Plusieurs microservices partagent des fichiers de code source ou des bibliothèques entre eux.

Symptômes : La présence de code exécutable dans un système à base de microservices et leur utilisation par plusieurs microservices, ainsi que les références à des bibliothèques au sein du code source peuvent indiquer la présence de cet anti-patron.

Conséquences : Cet anti-patron augmente le couplage des microservices entre eux et il brise le principe de « contexte borné » (*bounded context*) des microservices. Il engendre aussi des problèmes d'évolution, de test et de déploiement dans les systèmes à base de microservices (Richards, 2016).

Solution refactorisée : La solution à cet anti-patron est de partager aussi peu de code que possible, même au détriment du principe « DRY » (*Don't Repeat Yourself*) (Intelliware-Development, 2018), car ce principe ne s'applique que très rarement aux architectures à base de microservices. La figure 4.4 illustre ce procédé. Une autre solution est de créer un microservice responsable de l'encapsulation du code partagé et d'y faire appel par le biais de requêtes HTTP (voir figure 4.5).

Avantages de la refactorisation : Retirer cet anti-patron d'un système à base de microservices permet d'isoler les microservices et d'accroître leur indépendance.

Cela préserve aussi le principe de contexte borné des microservices.

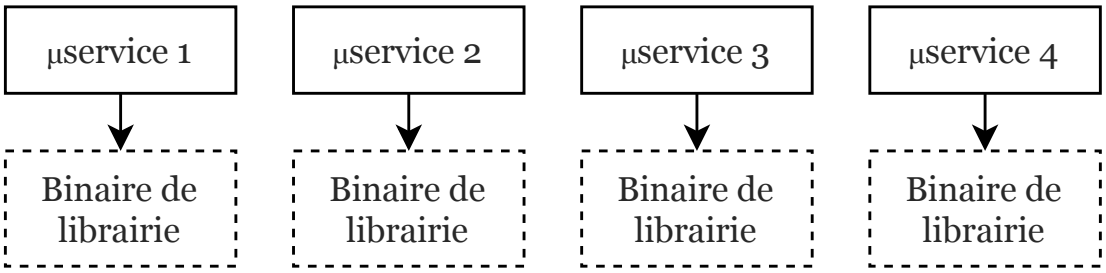


Figure 4.4: Chaque microservice dispose de son propre binaire de librairie

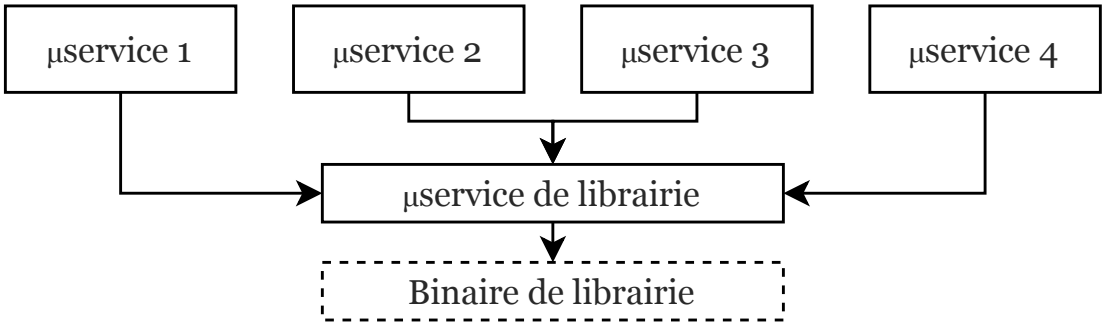


Figure 4.5: Un microservice est responsable d'encapsuler une librairie

4.2.6 Hardcoded Endpoints (HE)

Aussi connu comme : IPs et ports codés en dur

Nom de la solution : Service Discovery (Découverte de services)

Contexte : Les microservices ont besoin d'échanger des informations. Du fait de leur déploiement indépendant, et qu'ils communiquent via des APIs REST, n'importe quel microservice peut communiquer avec les autres en connaissant l'adresse et le numéro de port correspondants.

Forme générale : Les adresses Web, IP et les numéros de port sont explicitement écrits dans le code source du système à base de microservices . La figure 4.6 illustre

un exemple de cet anti-patron [3].

```
module.exports = {
  catalogueUrl:  util.format("http://catalogue%s", domain),
  tagsUrl:       util.format("http://catalogue%s/tags", domain),
  cartsUrl:      util.format("http://carts%s/carts", domain),
  ordersUrl:     util.format("http://orders%s", domain),
  customersUrl:  util.format("http://user%s/customers", domain),
  addressUrl:    util.format("http://user%s/addresses", domain),
  cardsUrl:      util.format("http://user%s/cards", domain),
  loginUrl:      util.format("http://user%s/login", domain),
  registerUrl:   util.format("http://user%s/register", domain),
};
```

Figure 4.6: Adresses codées en dur dans le code source

Symptômes : Cet anti-patron peut se manifester lorsque des adresses Web, des adresses IP ou des numéros de port sont contenus dans les fichiers de code source, de configuration ou d'environnement.

Conséquences : Lorsque le nombre de microservices d'un système à base de microservices est important, il est très difficile de garder une trace de toutes les URLs (**L**ocalisateur **U**niforme de **R**essource) et de toutes les adresses des différents microservices. De plus, lancer plusieurs instances d'un même microservice en parallèle dans le cadre d'un équilibrage de charge par exemple devient impossible. Enfin, si l'adresse d'un microservice change, il serait nécessaire de mettre à jour et de déployer de nouveau tous les microservices qui lui font appel.

Solution refactorisée : Utiliser la découverte automatique de services permet d'éviter de coder en dur les différentes adresses des microservices, et facilite grandement leurs communications. Deux stratégies de découverte de services existent :

---

3. https ://github.com/microservices-demo

découverte de service côté client tel qu'illustré sur la figure 2.2a et *découverte de service côté serveur* tel qu'illustré sur la figure 2.2b.

Avantages de la refactorisation : Lorsque la découverte de services est implémentée, la localisation des microservices peut changer sans en affecter les consommateurs. L'utilisation de cette technologie facilite aussi le déploiement des microservices sur des serveurs dont les adresses ne sont pas fixes. De plus, cela permet d'avoir un registre central de toutes les adresses des microservices.

4.2.7 Manual Configuration (MC)

Aussi connu comme : N/A
Nom de la solution : Configuration automatisée

Contexte : L'efficacité des architectures à base de microservices se base beaucoup sur l'automatisation. Dans ces architectures, tout ce qui peut être automatisé doit être automatisé.

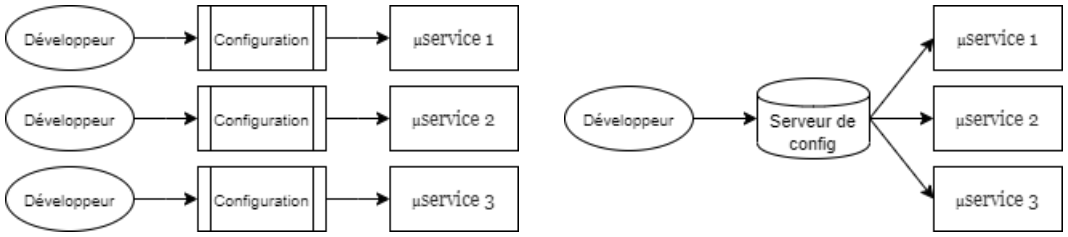
Forme générale : La configuration des instances, des microservices et des hôtes se fait manuellement par les développeurs comme illustré sur la figure 4.7a

Symptômes : Des fichiers de configuration présents au niveau de chaque microservice ainsi que l'utilisation intensive de variables d'environnement peuvent être des indices de la présence de cet anti-patron.

Conséquences : Le processus de configuration des microservices est un processus chronophage et peut engendrer des erreurs. En effet, pour chaque microservice, il peut être nécessaire de gérer plusieurs configurations différentes en fonction des environnements (développement, test, production, etc.). De ce fait, la configuration manuelle peut entraîner un cycle de développement plus long et des pannes du système.

Solution refactorisée : Afin d'éviter cet anti-patron, il est possible d'utiliser des serveurs de configuration dans le but d'automatiser cette tâche tel que décrit sur la figure 4.7b. Cela peut également être fait au travers d'outils de déploiement continu.

Avantages de la refactorisation : L'utilisation d'outils de configuration automatisée permet aux développeurs de gérer les configurations de leurs systèmes à base de microservices de façon centralisée, rendant ainsi plus simple de partager des éléments entre les différentes configurations et, plus généralement, permettant de maintenir un haut niveau de cohérence entre les configurations de plusieurs microservices.



(a) Configuration manuelle (b) Configuration automatisée

Figure 4.7: Configuration manuelle vs. Configuration automatisée

4.2.8 No CI/CD (NCI)

Aussi connu comme : Pas d'outils de DevOps
Nom de la solution : Utilisation de CI/CD

Contexte : Le déploiement indépendant des microservices permet à des équipes relativement petites —au sein de l'entreprise— d'appliquer des processus itératifs et continus de développement et d'opérations (DevOps) et, de ce fait, d'accroître l'agilité du système.

L'intégration du DevOps et le déploiement continu permettent de (1) réduire les

délais de livraison, (2) améliorer l'efficacité des processus de livraison, (3) réduire les délais entre les versions, et (4) maintenir la qualité du système (Newman, 2015).

Forme générale : Aucun mécanisme automatisé pour gérer les processus de test et de déploiement.

Symptômes : Un système à base de microservices dont les versions ne sont pas gérées à l'aide d'un dépôt distant (ex. Git), ne disposant pas de tests unitaires, fonctionnels ou d'intégration, n'utilisant pas d'outils de déploiement automatisés ou n'ayant pas d'environnement de pré-production peut indiquer la présence en son sein de cet anti-patron.

Conséquences : Les processus de test et de déploiement du système deviennent plus lents et sujets à des bogues (Duvall *et al.*, 2007). Les problèmes et erreurs de compilation deviennent plus difficiles à détecter. Aussi, les équipes au sein de l'organisation doivent faire preuve de plus de coordination pour assurer la cohérence des déploiements des microservices.

Solution refactorisée : Utiliser des outils d'intégration et de déploiement continus afin de construire un pipeline de test et de déploiement efficace.

Avantages de la refactorisation : Lorsque des outils d'intégration et de déploiement continus sont mis en place, les mises à jour de code peuvent être plus petites, ciblées, et déployées aussi vite qu'elles sont prêtes. Les bogues dans les nouvelles versions sont plus faciles à détecter et à suivre. Les retours en arrière sur une version sont simplifiés car la cause d'un problème peut être identifiée rapidement. De plus, les tests deviennent plus efficaces puisqu'ils sont lancés automatiquement à chaque nouveau déploiement.

4.2.9 No API Gateway (NAG)

Aussi connu comme : N/A

Nom de la solution : Utiliser une passerelle d'API

Contexte : Lors du développement de systèmes à base de microservices, les applications les consommant ont besoin de moyens de communication avec les microservices. Chaque application consommatrice a besoin de son propre ensemble d'informations.

Forme générale : Les microservices d'un système à base de microservices sont exposés et les applications consommatrices communiquent directement avec ces derniers. La figure 4.8 illustre ce procédé.

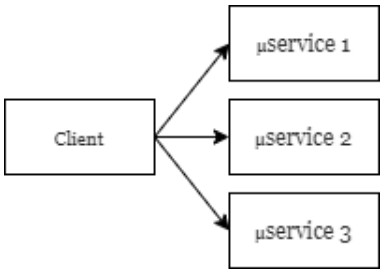


Figure 4.8: Communication directe avec les microservices

Symptômes : Des microservices recevant plusieurs requêtes, ou des systèmes disposant de plusieurs points d'entrée (page Web, application mobile, application de montre connectée) peuvent indiquer la potentielle présence de cet anti-patron.

Conséquences : Les applications consommatrices doivent connaître à l'avance le découpage du système afin de communiquer avec le bon microservice. Ces applications doivent aussi gérer et maintenir les adresses des différents microservices dont ils ont besoin. Enfin, l'authentification et la gestion des autorisations doivent se faire au niveau de chaque microservice.

Solution refactorisée : Utiliser une passerelle d'API comme point d'entrée unique pour tous les microservices comme illustré sur la figure 2.1

Avantages de la refactorisation : En tant que point d'entrée unique, la passerelle d'API deviendra responsable de (1) transmettre toutes les requêtes entrantes au microservice correspondant et (2) gérer les tâches communes telles que l'authentification, la gestion des autorisations, la validation des valeurs d'entrée et la transformation des réponses.

4.2.10 Timeouts (TO)

Aussi connu comme : Dog piles
Nom de la solution : Utiliser des disjoncteurs

Contexte : La disponibilité d'un service fait référence à la possibilité pour un service consommateur d'envoyer une requête à un service. La réactivité d'un service quant à elle correspond au temps mis par un service à envoyer une réponse à cette requête (Richards, 2016). Une pratique courante dans les systèmes distribués est de définir des délais d'attentes dans les applications consommatrices pour gérer l'indisponibilité ou le manque de réactivité d'un service.

Forme générale : Des valeurs de délais d'attente sont mises en place par les développeurs lors de l'envoi de requêtes ou de l'attente de réponses.

Symptômes : Un code responsable de renvoyer des requêtes échouées ou des valeurs de délais d'attente écrites en dur dans le code sont de bons signes de la présence de cet anti-patron.

Conséquences : Trouver la bonne valeur de délai d'attente est très difficile. Une valeur trop courte tombera vite dans l'exception causée par le délai dépassé, ne laissant pas aux services « occupés » suffisamment de temps pour traiter la re-

quête. Une valeur trop grande laissera le microservice consommateur en attente longtemps avant de se rendre compte de l'indisponibilité d'un service.

Solution refactorisée : Utiliser un système de disjoncteurs, qui vérifie a intervalle régulier l'état de santé des services et refuse les requêtes automatiquement dans le cas de l'indisponibilité d'un service.

Avantages de la refactorisation : Les microservices consommateurs ne doivent plus attendre avant de savoir qu'un service est indisponible. Cela évite aussi des tentatives de connexion inutiles. Les disjoncteurs se referment automatiquement lorsque les services sont de nouveau disponibles.

4.2.11 Multiple Service Instances per Host (MSIH)

Aussi connu comme : N/A
Nom de la solution : Utiliser un hôte par service

Contexte : Plusieurs stratégies de déploiement sont disponibles lors du développement de systèmes à base de microservices. Un développeur peut choisir de déployer chaque microservice sur son propre hôte (conteneur, serveur, machine virtuelle, etc.) ou bien de déployer plusieurs microservices sur un seul hôte.

Forme générale : Plusieurs instances de microservices sont déployées sur un seul et même hôte (voir la figure 4.9a)

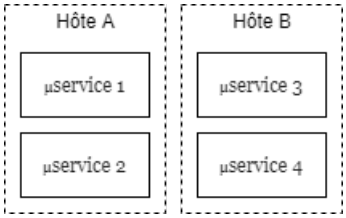
Symptômes : Des indices de la présence de cet anti-patron pourraient être l'utilisation de : (1) une seule plate-forme de déploiement dans le système, (2) un seul dépôt de versions (ex. Git), ou (3) un script de déploiement global n'utilisant pas de technologie de déploiement distribué.

Conséquences : Les microservices sont contraints de partager les ressources dis-

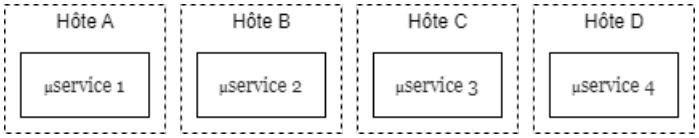
ponibles sur un seul hôte. De plus, mettre à l'échelle (augmentation ou réduction de ressources) un service donné implique de mettre à l'échelle tous les autres services qui partagent l'hôte. Enfin, des conflits de technologies peuvent avoir lieu entre des microservices partageant un hôte (différentes versions d'un langage de programmation, bibliothèques incompatibles, etc.)

Solution refactorisée : Chaque microservice devrait être déployé indépendamment sur son propre hôte, comme illustré sur la figure 4.9b.

Avantages de la refactorisation : Un seul microservice par hôte permet l'utilisation indépendante et la mise à l'échelle des ressources pour chacun d'entre eux. Cela offre également une couche d'isolation supplémentaire entre microservices et prévient de potentiels conflits entre les technologies utilisées par les microservices.



(a) Plusieurs instances par hôte



(b) Une seule instance par hôte

Figure 4.9: Plusieurs instances vs Une seule instance par hôte

4.2.12 Shared Persistence (SP)

Aussi connu comme : Possession des données
Nom de la solution : Base de données par service

Contexte : Les architectures à base de microservices prônent la décomposition d'un système en services indépendants (Lewis et Fowler, 2015). Chacun de ces services peut avoir besoin d'enregistrer et d'accéder à des données. Cependant, afin de bénéficier de tous les avantages des architectures à base de microservices,

les architectes logiciel doivent gérer le problème d'accès aux données où chaque microservices dispose de ces données sans affecter les autres (Microsoft, 2019).

Forme générale : Une seule base de données est accédée par plusieurs microservices (voir la figure 4.10)

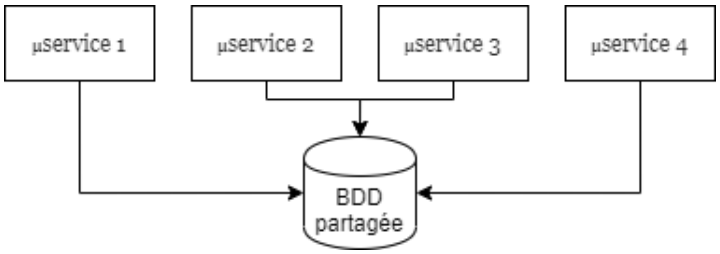


Figure 4.10: Base de données partagée

Symptômes : Cet anti-patron est caractérisé par un ou plusieurs symptômes parmi les suivants : (1) plusieurs microservices partagent les mêmes fichiers de configuration et environnements de déploiement, (2) les tables au sein des bases de données sont préfixées, ou (3) les bases de données du système contiennent un nombre important de schémas.

Conséquences : Les microservices deviennent fortement couplés (Microsoft, 2019), et la maintenance devient plus difficile. De plus, les données de chaque microservice doivent être adaptées afin de correspondre à un seul type d'entrepôt de données. L'indépendance des microservices est aussi affectée puisqu'ils partagent tous la même base de données.

Solution refactorisée : Les données doivent être séparées en prenant en considération la façon dont elles sont utilisées. Il est nécessaire de choisir le type de base de données approprié à chaque type de données (ex. persistance polyglotte (Fowler, 2011)) et d'utiliser une base de données unique pour chaque microservice. La figure 4.11 illustre la solution à cet anti-patron.

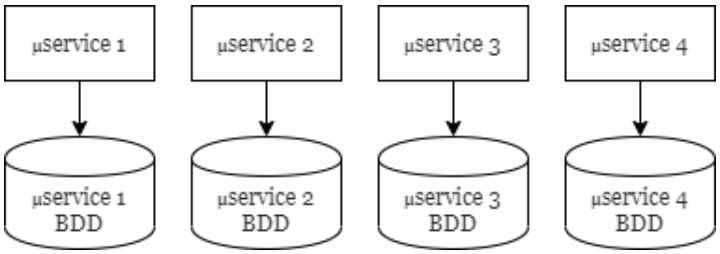


Figure 4.11: Base de données par service

Avantages de la refactorisation : Avoir une base de données par microservice offre les avantages suivants : (1) la séparation des préoccupations, puisque chaque microservice possède ses propres données, (2) la gestion intégrale d'un microservice par une seule équipe et (3) la flexibilité des technologies de stockage de données.

4.2.13 No API Versioning (NAV)

Aussi connu comme : Contrat statique
Nom de la solution : Gérer les version des APIs

Contexte : Dans un système à base de microservices, plusieurs versions d'un même microservice peuvent cohabiter. Cela pourrait être pour tester de nouvelles fonctionnalités ou pour assurer la rétro-compatibilité d'un service.

Forme générale : Les adresses et les URLs des microservices et des points d'entrée ne contiennent pas de numéros de version, et une seule version d'un microservice est disponible à un instant donné.

Symptômes : Certains points peuvent indiquer la présence de cet anti-patron : (1) les adresses au sein des microservices ne contiennent pas de numéros de version, (2) il n'y a pas d'information d'en-tête précisant la version dans les requêtes des microservices, et (3) plusieurs microservices ont des noms similaires.

Conséquences : Les changements dans un microservice peuvent causer l'annula-

tion de la rétro-compatibilité et, de ce fait, il devient nécessaire de mettre à jour tous les services consommateurs ou de déployer la nouvelle version vers une autre destination. La figure 4.12 (Alagarasan, 2016) illustre ce problème.

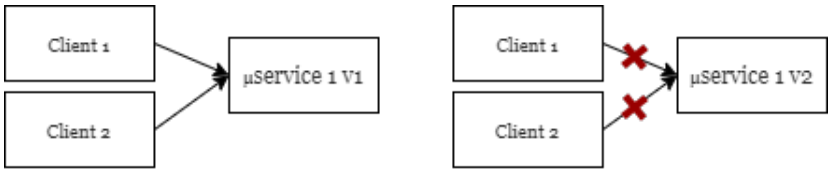


Figure 4.12: Pas de version d'API

Solution refactorisée : Utiliser un système de versions dans les APIs permet aux mêmes microservices de fournir plusieurs versions. Comme résultat, les microservices consommateurs ne sont plus affectés par le développement d'un changement majeur et le déploiement d'une nouvelle version.

Avantages de la refactorisation : Les microservices peuvent évoluer sans introduire des problèmes de rétro-compatibilité. Les consommateurs peuvent continuer à utiliser des versions précédentes en évoluant graduellement vers les nouvelles versions comme illustré par la figure 4.13 (Alagarasan, 2016).

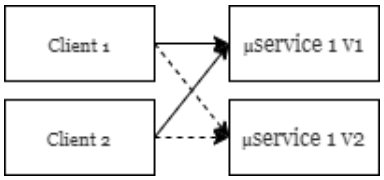


Figure 4.13: Utilisation des versions d'API

4.2.14 No Healthcheck (NHC)

Aussi connu comme : N/A
Nom de la solution : Utiliser une adresse de vérification de santé

Contexte : Les systèmes à base de microservices sont volatiles. Un microservice

peut être déployé n'importe où et peut être indisponible pendant un certain temps dans un contexte particulier.

Forme générale : Aucune adresse n'est exposée pour permettre la vérification de l'état de santé d'un microservice donné.

Symptômes : Aucune requête périodique vers un microservice, pas de passerelles d'API ni de découverte de service peuvent indiquer la présence de cet anti-patron.

Conséquences : Les consommateurs d'un microservice peuvent subir des temps d'attente importants sans recevoir de réponse. Les délais d'attente peuvent aussi être dépassés sans que le client ne sache que le microservice n'est pas disponible.

Solution refactorisée : Utiliser des adresses permettant la vérification périodique de l'état de santé d'un microservice et le fait qu'il est en mesure d'accepter de nouvelles requêtes.

Avantages de la refactorisation : Lorsque les consommateurs sont avertis qu'un microservice n'est pas disponible, ils n'envoient plus de requête. Cela évite des temps d'attente inutiles et des délais de réponse trop longs.

4.2.15 Local Logging (LL)

Aussi connu comme : N/A
Nom de la solution : Journalisation distribuée

Contexte : Chaque microservice produit une grande quantité de données journalisées dans plusieurs systèmes de fichiers. Ces informations sont très utiles dans un contexte de suivi et devraient être facilement accessibles.

Forme générale : Chaque microservice enregistre ses propres fichiers de journalisation.

Symptômes : Certaines indications de la présence de cet anti-patron sont (1) la présence de fichiers de journalisation dans les microservices eux-mêmes, (2) l'écriture de fichiers par les microservices, (3) l'utilisation de bases de données temporelles au sein des microservices ou (4) l'utilisation d'outils de journalisation locale par les microservices.

Conséquences : Les fichiers de journalisation locaux peuvent être difficiles à agréger et à analyser. Cela ralentit le processus de suivi proportionnellement au nombre de microservices et à la taille des fichiers de journalisation.

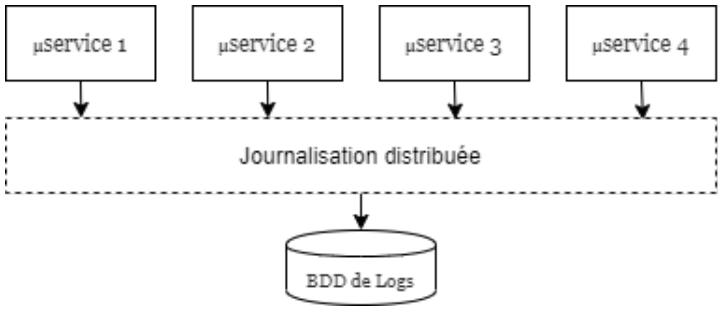


Figure 4.14: Journalisation distribuée

Solution refactorisée : Les systèmes de journalisation distribués permettent aux microservices d'utiliser un système qui sera responsable de l'agrégation des journaux de tous les microservices (voir la figure 4.14)

Avantages de la refactorisation : La journalisation distribuée permet d'avoir un seul entrepôt de journalisation, oblige les microservices à utiliser le même format de journaux et simplifie les processus d'analyse et de suivi.

4.2.16 Insufficient Monitoring (IM)

Aussi connu comme : Manque de suivi (*Lack of Monitoring*)
Nom de la solution : Métriques d'application

Contexte : Comprendre le comportement d'une application et dépanner les problèmes sont deux aspects cruciaux dans les systèmes à base de microservices (Richardson, 2020).

Forme générale : Les performances du système à base de microservices ne sont pas suivies par des outils regroupant toutes les informations relatives aux activités des microservices.

Symptômes : Un suivi insuffisant peut être introduit par l'utilisation de la journalisation locale, le manque de vérification de l'état de santé des microservices ou l'utilisation d'outils de suivi peu adaptés aux architectures à base de microservices.

Conséquences : Le manque de suivi peut affecter les activités de maintenance d'un système à base de microservices. Les pannes deviennent ainsi plus difficiles à éviter et à comprendre. Le suivi des performances générales du système devient plus complexe.

Solution refactorisée : Cet anti-patron peut être retiré en introduisant des outils de suivi récoltant des statistiques et des métriques pour chaque microservice dans le système à base de microservices. Ces informations essentielles sont relatives aux performances, aux activités et aux opérations individuelles des microservices.

Avantages de la refactorisation : Suivre chaque aspect d'un système à base de microservices permet de le comprendre dans son ensemble et d'anticiper le comportement des différents microservices le composant. Cela permet aussi de mettre en place des alertes et des rapports plus efficaces.

CHAPITRE V

DÉTECTION AUTOMATIQUE D'ANTI-PATRONS

5.1 Approche de détection

La seconde phase de nos travaux consiste à détecter automatiquement les occurrences d'anti-patrons dans des systèmes à base de microservices. Pour ce faire, nous prenons en entrée le résultat de la première phase de nos travaux, c'est-à-dire le catalogue d'anti-patrons, afin d'en extraire des règles de détection pour chaque anti-patron, ainsi qu'une liste de systèmes à base de microservices à code source ouvert pour créer notre méta-modèle (Tighilt *et al.*, 2020a).

À partir de ces règles de détection et de ce méta-modèle, nous appliquons notre outil de détection afin de détecter les pratiques, que nous validons manuellement afin de calculer les métriques de précision et de rappel.

L'objectif 2 de la figure 5.1 illustre cette phase de nos travaux.

Pour chaque anti-patron, nous avons défini un ensemble de règles de détection afin de détecter ses occurrences dans un système à base de microservices donné. Ci-dessous un rappel des anti-patrons accompagné d'une description textuelle des règles de détection permettant d'identifier leur présence dans un système à base de microservices.

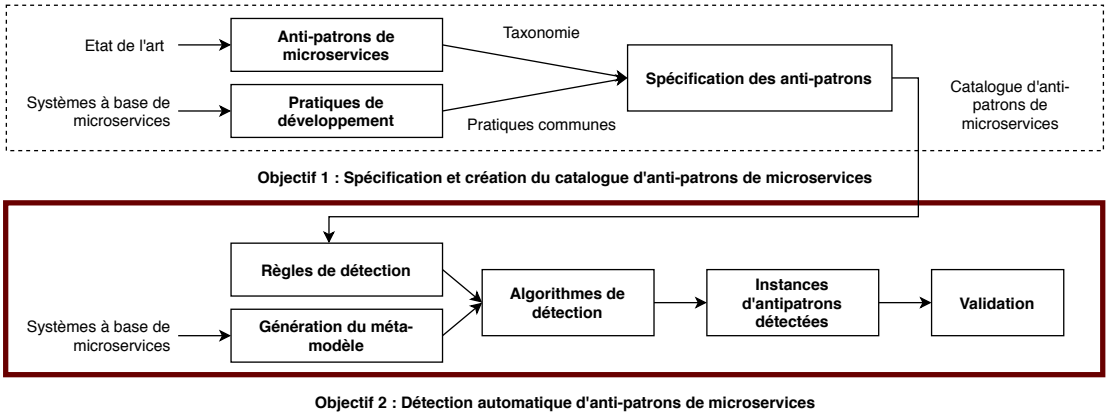


Figure 5.1: Méthodologie des travaux - Objectif 2

1. **Wrong Cuts (WC) :**

   Rappel : cet anti-patron signifie qu'un système à base de microservices est découpé selon un aspect technique et pas selon les fonctionnalités d'affaires (ex. couche présentation, couche métier, couche de données.) Voir section 4.2.1.

   Pour détecter cet anti-patron, nous considérons les microservices qui ne contiennent qu'un seul type de fichier de code source et font appel à des microservices ne contenant eux aussi qu'un seul type de fichier de code source. Un exemple serait un microservice ne contenant que du code présentation faisant appel à un microservice ne contenant que du code métier. Nous nous basons sur l'extension des fichiers, leurs contenus et le langage de programmation utilisé pour détecter cet anti-patron.

2. **Circular Dependencies (CD) :**

   Rappel : cet anti-patron implique que plusieurs microservices d'un système à base de microservices sont dépendants les uns des autres de façon circulaire. Voir section 4.2.2.

Pour détecter cet anti-patron, nous utilisons la pile d'appels du système à base de microservices afin d'extraire les appels circulaires entre les différents microservices composant le système.

3. **Mega Service (MS) :**

   > **Rappel :** cet anti-patron signifie qu'un ou plusieurs microservices ne se limitent pas à une seule fonctionnalité d'affaires. Voir section 4.2.3.

   Pour identifier cet anti-patron, nous détectons les microservices qui disposent de plus de lignes de code et de nombre de fichiers que le reste des microservices du système à base de microservices. Pour cela, nous avons établi les valeurs seuils pour cette règle de détection en utilisant une boîte à moustache sur l'ensemble de données des systèmes étudiés. Ces valeurs pour notre ensemble de données correspondent à 17 fichiers et 570 lignes de code.

4. **Nano Service (NS) :**

   > **Rappel :** cet anti-patron signifie qu'un ou plusieurs microservices ne répondent qu'à une partie d'une fonctionnalité d'affaires. Voir section 4.2.4.

   Ce type de microservices dispose de moins de lignes de code et de nombre de fichiers que le reste des microservices du système à base de microservices. Pour cet anti-patron aussi, nous avons aussi utilisé la boîte à moustache mentionnée dans les règles de détection des mega services. Ces valeurs pour notre ensemble de données correspondent à 4 fichiers et 95 lignes de code.

5. **Shared Libraries (SL) :**

> **Rappel :** cet anti-patron implique le partage de librairies (sous forme de fichiers binaires par exemple) par plusieurs microservices d'un système à base de microservices. Voir section 4.2.5.

Afin d'identifier cet anti-patron, nous procédons à l'extraction des bibliothèques qui sont utilisées par plusieurs microservices du système à base de microservices. Nous ne considérons ici que les bibliothèques internes du système et non celles provenant de dépôts externes (Maven par exemple).

6. **Hardcoded Endpoints (HE) :**

> **Rappel :** cet anti-patron consiste en l'écriture d'URLs, de numéros de ports et d'adresses IP directement dans le code source des microservices d'un système à base de microservices. Voir section 4.2.6.

Pour identifier la présence de cet anti-patron, nous considérons des appels d'APIs, des URLs, des adresses IP ou des numéros de ports qui sont présents dans le code source des microservices, dans les fichiers de configuration, dans les fichiers de déploiement ou encore dans les fichiers d'environnement. De plus, nous considérons le fait qu'aucun mécanisme de découverte automatique de services ne soit implémenté dans le système.

7. **Manual Configuration (MC) :**

> **Rappel :** cet anti-patron signifie que la configuration de chaque microservice se fait manuellement par les développeurs, sans système de gestion de configuration. Voir section 4.2.7.

Nous détectons pour cet anti-patron les microservices possédant leurs propres fichiers de configuration. De plus, nous vérifions qu'aucun outil de gestion de configuration centralisée ne soit présent dans les dépendances du sys-

tème à base de microservices.

8. **No CI/CD (NCI) :**

   > **Rappel :** cet anti-patron signifie qu'aucune pipeline d'intégration et de déploiement continus n'est présente dans le système à base de microservices. Voir section 4.2.8.

   Afin d'identifier cet anti-patron, nous vérifions si (1) les fichiers de configuration et le dépôt de contrôle de version ne contiennent aucune information relative à l'intégration et au déploiement continu et (2) aucun fichier de bibliothèque d'intégration et de déploiement continus n'est présent parmi les fichiers du système à base de microservices. Nous nous basons sur une liste extensible d'outils d'intégration et de déploiement continus afin de détecter cet anti-patron.

9. **No API Gateway (NAG) :**

   > **Rappel :** cet anti-patron consiste en la communication directe entre les clients du système et les microservices le composant. Aucune passerelle d'API n'est implémentée afin de servir de point d'entrée unique. Voir section 4.2.9.

   Nous déterminons la présence de cet anti-patron en vérifiant si le code source des microservices du système à base de microservices ne contient pas des implémentations d'outils de passerelle d'API communes (ex. Netflix Zuul). De plus, nous vérifions si aucune bibliothèque de passerelle d'API n'est présente dans la liste de dépendances du système à base de microservices.

10. **Timeouts (TO) :**

> **Rappel :** cet anti-patron signifie qu'aucun mécanisme de disjoncteur n'est implémenté dans le système à base de microservices afin d'éviter les temps de réponse trop long lors d'une erreur. Voir section 4.2.10.

Nous détectons cet anti-patron en cherchant des valeurs de *timeout* qui soient présentes dans les appels d'API au sein du code source des microservices. De plus, nous vérifions qu'aucune implémentation d'outils de *circuit breaker* communs (ex. Netflix Hystrix) ne soit présente dans le code source des microservices ou dans les dépendances du système à base de microservices.

11. **Multiple Service Instances per Host (MSIH) :**

> **Rappel :** cet anti-patron signifie que plusieurs microservices d'un système à base de microservices sont déployés sur un seul hôte. Voir section 4.2.11.

En présence de cet anti-patron, le système à base de microservices n'utilise pas généralement des technologies de déploiement telles que "docker-compose". Nous détectons cet anti-patron en analysant les fichiers de déploiement et nous vérifions que le système à base de microservices possède un seul fichier de déploiement qui déploie tous les microservices sur un seul hôte.

12. **Shared Databases (SD) :**

> **Rappel :** cet anti-patron veut dire que plusieurs microservices d'un système à base de microservices partagent une ou plusieurs sources de données. Voir section 4.2.12.

Nous identifions cet anti-patron en détectant les microservices qui par-

tagent des URLs de sources de données. Aussi, nous vérifions si une seule base de données est créée dans le système et est utilisée par l'ensemble des microservices.

13. **No API Versioning (NAV) :**

    > **Rappel :** cet anti-patron signifie qu'aucune information relative à la version d'un microservice n'est présente dans ses URLs ou dans les informations d'en-tête. Voir section 4.2.13.

    Nous détectons la présence de cet anti-patron en vérifiant si les adresses et les URLs internes (par oppositions aux adresses externes, c-à-d, adresses distantes) du système contiennent des informations sur la version du microservice. Aussi, nous vérifions qu'il n'y a pas d'informations relatives aux versions des APIs qui soient présentes dans les fichiers de configuration.

14. **No HealthCheck (NHC) :**

    > **Rappel :** cet anti-patron veut dire que l'état de fonction d'un microservice n'est pas vérifié périodiquement de façon automatique. Voir section 4.2.14.

    Nous identifions cet anti-patron en cherchant s'il n'y a pas `/healthcheck` ou `/health` dans la liste des adresses et des URLs du système à base de microservices. De plus, nous vérifions s'il n'y a pas d'implémentation d'outils de vérification de *healthcheck* communs dans le code source des microservices (ex. Springboot Actuator). Finalement, nous vérifions si aucun mécanisme de vérification de *healthcheck* dans les technologies de déploiement n'est utilisé (ex. `Docker HEALTHCHECK`) dans le système.

15. **Local Logging (LL) :**

> **Rappel :** cet anti-patron signifie que chaque microservice d'un système à base de microservices enregistre ses propres fichiers de journalisation, et qu'aucun mécanisme de journalisation distribuée n'est utilisé dans le système. Voir section 4.2.15.

Nous présumons la présence de cet anti-patron si aucune bibliothèque de journalisation distribuée n'est présente dans les dépendances du système ou que chaque microservice dispose de son propre système de journalisation.

16. **Insufficient Monitoring (IM) :**

> **Rappel :** cet anti-patron consiste en un manque de suivi et de récolte de métriques au sein de chaque microservice d'un système à base de microservices. Voir section 4.2.16.

Nous détectons cet anti-patron en cherchant si des bibliothèques de suivi communes (ex. Prometheus) sont implémentées dans le code source du système à base de microservices.

5.2 Méta-modèle de la solution

Nous avons établi un méta-modèle afin d'encapsuler toutes les informations nécessaires à l'application de nos algorithmes de détection. Il inclut les informations relatives aux éléments suivants : le système à analyser, son dépôt distant Git, les microservices composant le système, les dépendances, le code source, les fichiers d'environnement, les fichiers de configuration, les fichiers de déploiement, les images de conteneurs Docker, les plate-formes de déploiement, les requêtes HTTP, les bases de données et les imports.

Ce méta-modèle permet à nos algorithmes d'accéder aux informations pertinentes

tout en étant indépendants de la source de provenance de ces informations. Cela évite aussi d'avoir à réinterpréter le code source des systèmes à base de micro-services et permet d'assurer l'évolution de MARS en introduisant de nouveaux composants dans le méta-modèle ainsi que les algorithmes de détection correspondant à la détection d'un nouvel anti-patron. Cela permet aussi aux algorithmes de détection d'être indépendants des technologies du système, par exemple, en faisant abstraction des dépendances à travers le composant `Dependency`, et ce, peu importe leur provenance (ex. Maven, Gradle, NPM, etc.)

La figure 5.2 illustre les différents composants de notre méta-modèle ainsi que leurs relations. Chaque composant du méta-modèle est utilisé dans le processus de détection d'un ou de plusieurs anti-patrons. Par exemple, le composant `Configuration` est utilisé pour la détection de l'anti-patron *Hardcoded Endpoints* en cherchant les URLs écrites dans les fichiers de configuration, en même temps que le composant `Code` et le composant `Dependency`.

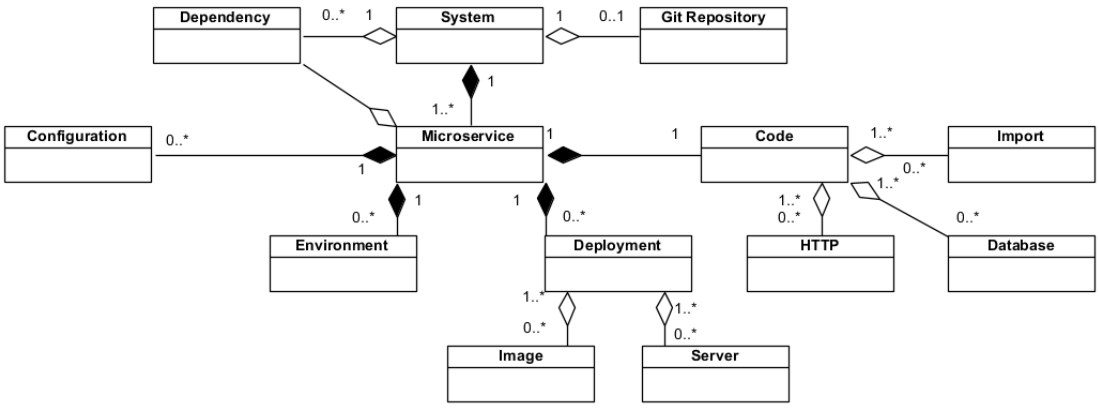


Figure 5.2: Méta-modèle de MARS

Nous décrivons ci-dessous les différents composants de notre méta-modèle.

Le composant `System` constitue la racine d'un modèle, il est construit soit en important un dépôt distant `Git` (qui est un composant facultatif du modèle) ou

en important un système à base de microservices depuis le système de fichiers local de l'utilisateur. Un `System` dispose d'informations à propos de deux composants : (1) `Microservice`(s) et (2) `Dependency`(s).

Le composant `Microservice` représente un seul microservice dans le système donné en entrée. Il contient des informations telles que le nombre de fichiers et le nombre de lignes de code du microservice (informations utilisées pour détecter les anti-patrons *Mega Service* et *Nano service*).

Le composant `Dependency` est référencé à la fois par le composant `System` et le composant `Microservice` car il est commun pour un système à base de microservices d'avoir des dépendances (ex. Maven, Gradle) aux deux niveaux de la hiérarchie du système. Ce composant contient des informations relatives aux dépendances du système ou d'un microservice en particulier.

Le composant `Configuration` sauvegarde des données récoltées depuis les différents fichiers de configuration d'un microsevice. Cela permet de rechercher des informations seulement dans les fichiers de configuration, par exemple les variables relatives aux cadriciels utilisés dans le système ou l'activation/désactivation de fonctionnalités.

Le composant `Environment` contient les informations relatives aux variables d'environnement, typiquement leur nom et la valeur associée, car ces variables sont communément utilisées dans le but d'injecter dynamiquement des variables dans un système.

Le composant `Deployment` fait référence aux configurations et mécanismes de déploiement du système. Ce composant permet d'abstraire les Dockerfiles, les fichiers de configuration docker-compose et les fichiers de déploiement personnalisés. Il référence aussi deux composants du méta-modèle : `Image` (ex. pour la technologie de

déploiement Docker) ou `Server` (ex. pour Amazon ECS) en encapsulant les données relatives à ces systèmes de déploiement. Ce composant permet par exemple de faciliter l'identification du type d'un microservice en examinant l'`Image` depuis laquelle il est dérivé (ex. base de données).

Le composant `Code` est le plus utilisé au sein de MARS. En effet, ce composant contient des données relatives au code source des microservices du système afin d'en extraire les informations pertinentes sans avoir à traverser le code source plusieurs fois. Ce composant inclut les éléments suivants :

1. Fichiers sources : Un dictionnaire de noms de fichiers et leurs chemins. Cela va servir à filtrer les fichiers de test par exemple.
2. Fichiers filtrés : Un dictionnaire de chemins vers les fichiers de code source filtrés (sans commentaires par exemple).
3. Imports : Une liste des *imports* réalisés dans les fichiers de code source.
4. Méthodes : Une liste des noms de méthodes contenues dans le code source.
5. Annotations : Une liste d'annotations contenues dans le code source.
6. HTTP : Une liste de tous les appels HTTP contenus dans le code source.
7. BDD : Une liste de requêtes vers les bases de données ainsi qu'une liste de requêtes de création de tables contenues dans le code source.
8. Pile d'appels : La pile d'appels du code source générée via le cadriciel MoDisco KDM [1].

En plus de ces sous-composants, le composant `Code` définit aussi quelques informations importantes telles que le langage utilisé dans un microservice ou le type de fichier dans lequel il se trouve.

---

1. `https://wiki.eclipse.org/MoDisco/Components/KDM`

5.3 Implémentation et technologies

Nous implémentons MARS en utilisant une variété de cadriciels et de bibliothèques qui sont les plus adaptés à chacune des tâches nécessaires. Nous utilisons MoDisco KDM, un cadriciel de réingénierie logicielle pour l'environnement de développement Eclipse, afin d'extraire la pile d'appels du système et de créer un premier modèle. Nous identifions les bibliothèques et les dépendances du système et de chacun des microservices en utilisant *Bibliothecary*[2], une bibliothèque développée en Ruby pour la plate-forme *libraries.io*[3]. Pour l'analyse syntaxique des dépendances provenant de Gradle, nous utilisons un service NodeJS particulier nommée *Gradle Parser*[4]. Enfin, nous combinons toutes ces données pour créer un modèle de systèmes à base de microservices conforme à notre méta-modèle en utilisant le langage de programmation Python. Nous utilisons aussi Python pour implémenter les règles de détection appliquées à ce méta-modèle. Nous distribuons MARS comme un projet à code source ouvert disponible sur notre dépôt Github[5].

MARS est conçu pour être extensible. Nous l'avons construit en nous basant sur un méta-modèle indépendant des technologies des systèmes à base de microservices afin de supporter plusieurs langages de programmation, technologies de déploiement et outils. Par exemple, la recherche de bibliothèques dans MARS est assistée par un fichier de configuration textuel dans lequel les développeurs peuvent spécifier les bibliothèques qu'ils désirent considérer dans un système à base de microservices. Une méthode similaire est appliquée aux langages de pro-

---

2. `https://github.com/librariesio/bibliothecary`

3. `https://libraries.io`

4. `https://github.com/librariesio/gradle-parser`

5. `https://github.com/rtighilt/MARS`

grammation : pour supporter un nouveau langage de programmation, tel que Go ou JavaScript (langages très populaires pour le développement de systèmes à base de microservices), il suffirait d'ajouter le nom du langage dans un fichier de configuration et d'implémenter les quelques méthodes nécessaires à l'interprétation du code source.

La figure 5.3 résume le processus de détection des anti-patrons dans les systèmes à base de microservices et ses différentes étapes.

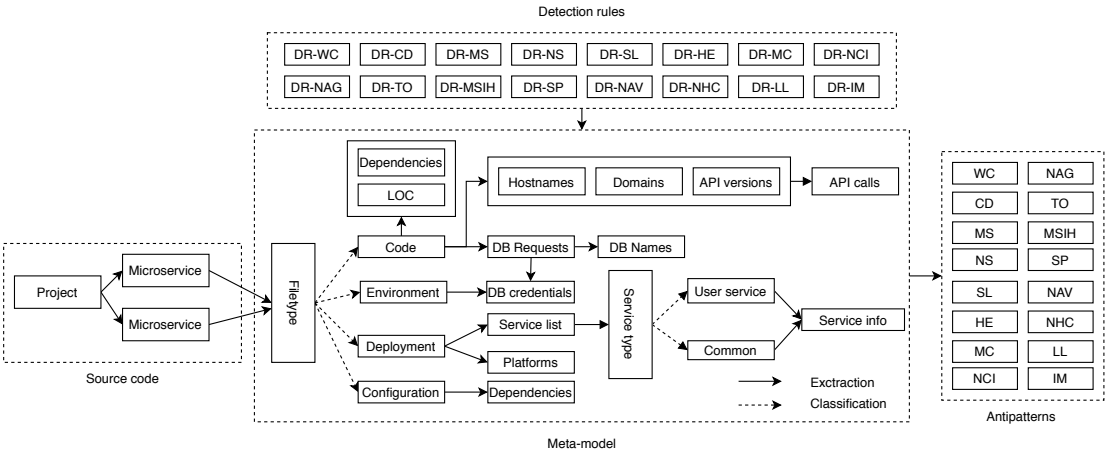


Figure 5.3: Microservice Antipatterns Research Software (MARS)

MARS prend en entrée un projet à analyser, sous forme de dépôt distant Git ou de répertoire sur le système de fichier local de l'utilisateur. Le méta-modèle de ce système est ensuite construit en extrayant les informations requises (Code source, fichiers de configuration, requêtes aux bases de données, etc.). Les algorithmes de détection sont ensuite appliqués sur ces informations à travers les règles de détection de chaque anti-patron. MARS retourne ensuite une liste d'occurrences trouvées pour chaque anti-patron.

5.4 Procédure d'exécution

Afin d'exécuter MARS, quelques étapes préliminaires sont nécessaires. Nous présumons dans la présente section que le système à base de microservices à analyser est un système présent sur le système de fichier local de l'utilisateur et que l'ensemble des librairies et outils nécessaires au fonctionnement de MARS sont installés et prêts à être utilisés.

Tout d'abord, il est important de préparer le système à être analysé par MARS. Cette procédure consiste à placer l'ensemble du code source du système à base de microservices dans un sous-répertoire nommé `source`, puis à générer la pile d'appels grâce au cadriciel « MoDisco KDM » d'Eclipse.

Une fois la pile d'appel générée, il faut l'interpréter pour en extraire les appels entre les différentes classes composant le système à base de microservices. Cette étape se réalise grâce à un outil développé dans le cadre d'un projet de recherche connexe au nôtre. Ainsi, nous obtenons un fichier de type `csv` contenant les relations entre les classes du système à base de microservices. Ce fichier doit être renommé `callgraph.csv` et se trouver à la racine du projet (au même niveau que le sous-répertoire `source`).

Ensuite, il faut spécifier à MARS les différents dossiers ne représentant pas de réels microservices dans le système. Pour ce faire, il faut créer un fichier appelé `exclude.txt`. Ce fichier doit aussi se trouver à la racine du projet.

Lorsque la procédure de préparation du système à base de microservices est effectuée, il suffit simplement d'exécuter MARS (avec la commande `python main.py`) en se situant dans le dossier `Analyzer` et en spécifiant, en premier argument, le chemin vers la racine du projet à analyser.

MARS va alors générer dans un premier temps un modèle conforme au méta-modèle décrit à la section 5.2 relatif au système fourni en entrée, puis appliquer tour-à-tour les algorithmes de détection de chaque anti-patron et renvoyer une liste d'occurrences pour chaque anti-patron détecté.

CHAPITRE VI

VALIDATION ET RÉSULTATS

6.1 Validation

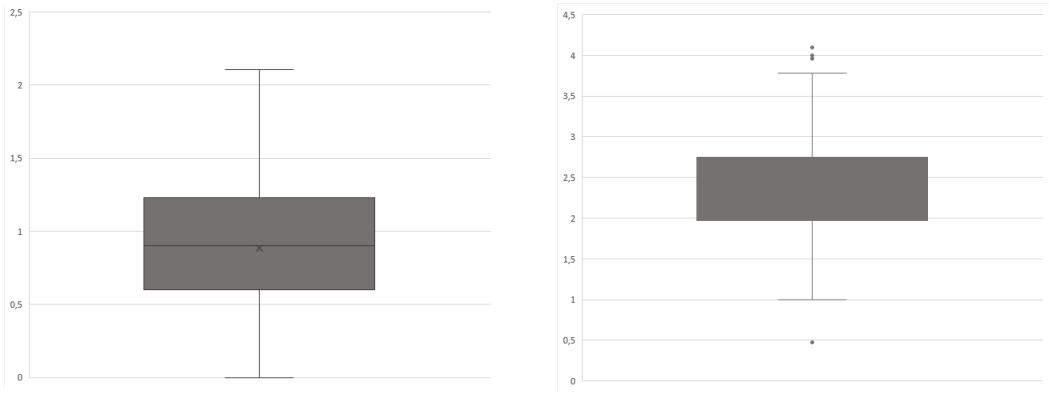
Afin de valider notre approche, nos règles de détection et notre outil, nous avons appliqué MARS sur 24 systèmes à base de microservices écrits dans le langage de programmation Java [1]. Ces 24 systèmes sont issus d'un ensemble de données de systèmes à base de microservices disponible en ligne (Rahman *et al.*, 2019). Cet ensemble de données est décrit dans le tableau 6.1. Les figures 6.1a et 6.1b montrent la répartition du nombre de fichiers et du nombre de lignes de code dans les systèmes. Les valeurs ont été normalisées en utilisant la fonction `log` par souci de clarté. Les axes des ordonnées dans les figures 6.1a et 6.1b correspondent à la fonction `log(valeur)`.

Le code source de tout système à base de microservices contient du code écrit par son/ses développeur(s) mais aussi d'autres artefacts et du code issu de bibliothèques utilisées dans le système. Inclure du code tiers dans notre analyse aurait probablement faussé les résultats. De ce fait, nous avons exclu du code source des systèmes à base de microservices composant notre ensemble de données tout le

1. La version actuelle de MARS ne supporte que la rétro-ingénierie de systèmes en JAVA, cependant la prise en charge d'autres langages de programmation est prévue dans les travaux futurs.

System	# Microservices	# Files	LOCs
Spring Netflix OSS	3	17	443
FTGO	9	257	8239
LakeSide Mutual	13	55	2170
Spring Petclininc	3	25	795
Freddy's BBQ	6	35	1752
Spring cloud Movie	4	33	885
Piggymetrics	4	88	3176
Tap And Eat	6	31	576
E-commerce	3	24	756
Consul	3	38	1750
Microservice Demo	3	38	1766
Qbike	5	77	2057
Spring cloud Microservice	9	21	673
CQRS Microservice Sampler	3	26	1028
Spring boot Microservices	2	4	116
CAS microservice architecture	5	29	871
Cloud Strangler example	3	30	932
Micro company	13	55	2170
MicroService	13	42	1052
MicroService Kubernetes	3	38	1640
TeaStore	3	62	5073
Warehouse Microservice	6	222	4623
Apollo	9	68	29510
Delivery System	2	14	537

Tableau 6.1: Nombre de microservices, lignes de code et nombre de fichiers par système de l'ensemble de données



(a) Nombre de fichiers des systèmes (b) Lignes de code des systèmes

Figure 6.1: Lignes de code et nombre de fichiers dans les systèmes analysés

code relatif aux bibliothèques et dépendances, et ce, en ne prenant pas en compte les dossiers de dépendances (ex. Gradle, mvn, composer vendor, node modules, etc.).

Nous avons ensuite appliqué notre outil sur les systèmes à base de microservices filtrés et comparé les résultats de la détection à notre ensemble d'instances vérifiées, c-à-d., les instances trouvées manuellement dans les systèmes.

Afin de constituer notre ensemble d'instances d'anti-patrons vérifiées, trois évaluateurs (deux professeurs et une doctorante) ont procédé indépendamment à une analyse des systèmes à base de microservices de l'ensemble de données. Chaque évaluateur avait une liste spécifique d'anti-patrons à identifier dans les systèmes.

Un quatrième évaluateur (candidat à la maîtrise) a analysé tous les systèmes en cherchant tous les anti-patrons. De cette manière, nous étions certains d'obtenir au moins deux avis par anti-patron identifié.

Après avoir indépendamment collecté les instances d'anti-patrons, les quatre évaluateurs en charge de la validation ont procédé à la discussion des résultats trouvés afin de lever toute ambiguïté et de résoudre tout résultat contradictoire. De ce

fait, la validation manuelle de notre outil s'est faite grâce à quatre intervenants distincts.

Anti-patron	**Valeur d'accord**
Wrong Cuts	0.91
Circular dependencies	1
Mega Service	0.96
Nano Service	0.67
Shared Libraries	1
Hardcoded Endpoints	0.79
Manual Configuration	0.92
No CI / CD	1
No API Gateway	0.88
Timeouts	0.88
Multiple Service Instances per Host	0.79
Shared Persistence	0.83
No API versioning	0.88
No Health Check	0.75
Local Logging	0.83
Insufficient Monitoring	0.83
Moyenne	**0.87**

Tableau 6.2: Aperçu des valeurs d'accord inter-évaluateurs

Nous avons calculé les valeurs d'accord inter-évaluateurs (en utilisant la métrique de Cohen Kappa (Cohen, 1960)) par anti-patron, détaillées dans le tableau 6.2 et nous avons obtenu une valeur moyenne de 0.87, ce qui correspond à un accord « Excellent » ou « Presque parfait » (Cicchetti et Sparrow, 1981; Landis et Koch, 1977) selon les sources.

6.2 Résultats

Cette section présente les résultats et observations obtenus après avoir appliqué MARS sur notre ensemble de données de 24 systèmes à base de microservices.

Nous avons calculé les valeurs de précision et de rappel pour chaque anti-patron détecté en utilisant les formules suivantes (AP correspondant à « Anti-patron ») :

$$Precision = \frac{|\{APs\ existants\} \cap \{APs\ detectes\}|}{|\{APs\ detectes\}|} \tag{6.1}$$

$$Rappel = \frac{|\{APs\ existants\} \cap \{APs\ detectes\}|}{|\{APs\ existants\}|} \tag{6.2}$$

La précision de notre outil de détection est satisfaisante puisqu'elle varie entre 35% et 100%, avec une moyenne de 68.5%. Le rappel est lui aussi satisfaisant, il varie entre 33.33% et 100% avec une moyenne de 78.22%.

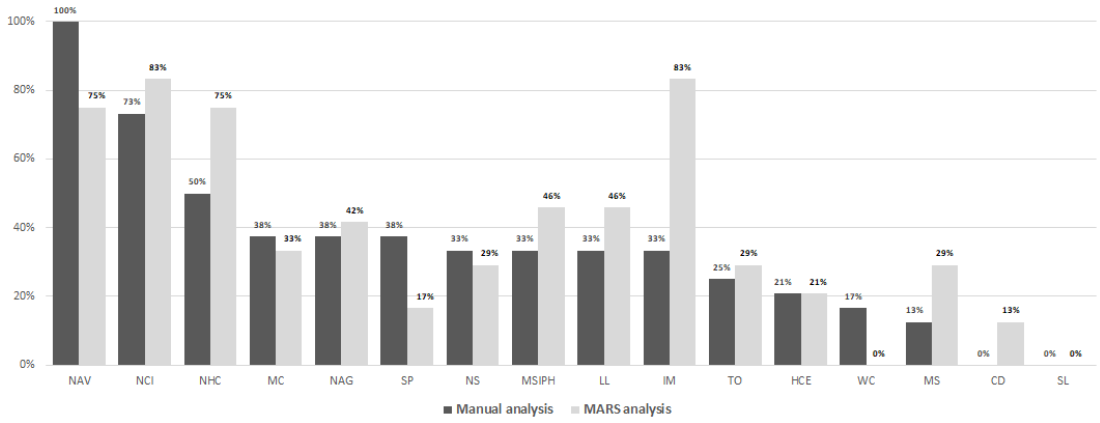


Figure 6.2: Pourcentage de systèmes contenant chaque anti-patron

Les valeurs de précision et de rappel obtenues confirment l'efficacité de MARS à détecter des occurrences d'anti-patrons de notre catalogue dans les systèmes à base de microservices.

Les résultats de notre analyse sont décrits dans le tableau 6.3 et sur la Figure 6.2.

System	Antipattern															
	WC	CD	MS	NS	SL	HE	MC	NCI	NAG	TO	MSIH	SP	NAV	NHC	LL	IM
Spring Netflix OSS	-	-	-	-	-	-	-	✓	-	-	-	-	✓	(✓)	(✓)	(✓)
FTGO	-	-	✓	(-)	-	✓	✓	-	-	✓	-	-	(-)	-	-	-
LakesideMutual	(-)	-	(✓)	(✓)	-	(-)	✓	✓	✓	-	(-)	-	-	-	(✓)	(✓)
Spring Petclinic	-	-	-	-	-	-	-	-	-	-	-	✓	-	(✓)	-	-
Freddys BBQ	-	-	-	-	-	-	-	✓	✓	-	✓	(-)	✓	✓	(✓)	(✓)
Spring cloud movie	-	(✓)	-	-	-	-	-	✓	-	-	✓	-	✓	✓	-	(✓)
Piggymetrics	-	-	(✓)	-	-	-	-	-	-	-	-	(-)	✓	-	-	(✓)
Tap And Eat	-	-	-	-	-	-	-	✓	✓	-	-	-	✓	✓	(✓)	(✓)
E-commerce	-	-	-	-	-	-	-	✓	(✓)	(✓)	-	-	✓	✓	✓	✓
Consul	-	-	-	-	-	(✓)	✓	✓	(✓)	✓	-	-	✓	-	(-)	-
ms demo	-	-	-	-	-	-	✓	✓	(-)	-	-	-	✓	-	(✓)	(✓)
Qbike	-	-	(✓)	(-)	-	-	(-)	✓	-	-	-	✓	✓	✓	-	(✓)
Spring cloud ms	-	-	-	✓	-	-	-	✓	-	-	(✓)	(-)	✓	✓	-	(✓)
Cqrs ms sampler	(-)	-	-	-	-	-	-	✓	-	-	-	-	✓	✓	✓	✓
Spring boot ms	-	-	-	✓	-	-	✓	✓	-	-	✓	-	✓	✓	✓	✓
Cas ms archi.	-	-	-	-	-	✓	-	✓	✓	✓	✓	-	✓	✓	✓	✓
Cloud Strangler	-	-	-	(✓)	-	-	-	✓	-	-	✓	(-)	✓	(✓)	-	(✓)
Micro company	(-)	-	-	✓	-	-	-	(✓)	-	-	-	(-)	✓	(✓)	(✓)	(✓)
MicroService	-	-	-	(✓)	-	-	-	✓	✓	-	✓	-	✓	(✓)	-	(✓)
Microservice K8s	-	-	-	-	-	✓	✓	✓	✓	(✓)	(✓)	-	(-)	-	✓	✓
TeaStore	(-)	(✓)	✓	(-)	-	(✓)	✓	✓	✓	✓	(✓)	✓	(-)	✓	-	✓
Warehouse ms	-	-	(✓)	-	-	-	-	✓	-	(-)	(✓)	-	✓	(✓)	-	(✓)
Apollo	-	(✓)	✓	(✓)	-	-	(✓)	-	✓	✓	-	✓	(-)	✓	(-)	(-)
Delivery system	-	-	-	-	-	-	-	✓	-	-	✓	-	✓	✓	(-)	✓
Précision	(NA)	(NA)	(3/7)	(3/7)	(NA)	(3/5)	(7/8)	(19/20)	(8/10)	(5/7)	(7/11)	(4/4)	(18/18)	(12/18)	(5/11)	(7/20)
	(NA)	(NA)	42.86%	42.86%	(NA)	60%	87.5%	95%	80%	71.4%	63.64%	100%	100%	66.67%	45.45%	35%
Rappel	(NA)	(NA)	(3/3)	(3/8)	(NA)	(3/5)	(7/9)	(19/19)	(8/9)	(5/6)	(7/8)	(4/9)	(18/24)	(12/12)	(5/8)	(7/7)
	(NA)	(NA)	100%	37.5%	(NA)	60%	77.78%	100%	88.89%	83.3%	87.5%	44.44%	75%	100%	62.5%	100%

Tableau 6.3: Résultats de détection de MARS : ✓ Vrai positif, (✓) Faux positif, - Vrai négatif, (-) Faux négatif

Le restant de cette section présente les détails de l'analyse pour chaque anti-patron, ainsi que les observations tirées de cette analyse.

1. Wrong Cuts (WC)

Décomposer un système en microservices doit prendre en considération les fonctionnalités d'affaires et non les aspects techniques. De ce fait, un microservice doit contenir tous les artefacts dont il a besoin pour répondre à un besoin d'affaires (présentation, code métier, bases de données, etc.).

Lors de l'exécution de MARS sur les systèmes à base de microservices de notre ensemble de données, nous n'avons détecté aucune occurrence de cet anti-patron. Cependant, nous avons trouvé quatre occurrences lors de la validation manuelle. Cette différence est due à la difficulté d'obtenir les relations entre les microservices en utilisant seulement une analyse statique du code (Kitajima et Matsuoka, 2017).

Observation 1 : Nous observons que quelques systèmes (4/24) contiennent des occurrences de découpage erroné. La majorité des systèmes à base de microservices analysés sont décomposés de manière adéquate.

2. Circular Dependencies (CD)

Les microservices doivent être indépendants et faiblement couplés. La communication entre les microservices doit se faire via des mécanismes légers, (ex. en utilisant les APIs REST) afin d'éviter de créer un « monolithe distribué » (Taibi et Lenarduzzi, 2018).

MARS a détecté trois occurrences de cet anti-patron dans les systèmes à base de microservices analysés. Cependant, lors de la validation manuelle, nous n'avons

trouvé aucune occurrence. Cela est du à l'analyse manuelle des piles d'appels qui est une tâche complexe à réaliser manuellement. Nous avons vérifié les résultats de MARS et nous avons confirmé que les trois occurrences trouvées sont bel et bien valides. Puisque la validation manuelle n'a retourné aucun résultat, nous n'avons pas pu calculer les valeurs de précision et de rappel pour cet anti-patron.

Observation 2 : La majorité des systèmes à base de microservices analysés ne contiennent pas de dépendances circulaires au sein de leurs microservices, ce qui constitue une bonne pratique et l'un des piliers de ce style d'architectures.

3. Mega Service (MS)

Les microservices doivent être des unités logicielles indépendantes fournissant une seule fonctionnalité d'affaires. Un mega service dans un système à base de microservices peut entraîner des problèmes de maintenance, augmenter la complexité des tests et réduire les performances générales du système. MARS a détecté les occurrences de cet anti-patron avec une précision de 42.85% et un rappel de 100%. La subjectivité de cet anti-patron peut être la cause de la valeur de la précision. En effet, un mega service dépend grandement du contexte du système et des fonctionnalités d'affaires représentées.

Observation 3 : Nous observons que seulement 12.5% (3/24) des systèmes à base de microservices analysés contiennent des occurrences de mega services (voir la figure 6.2). Éviter les mega services est une bonne pratique et est largement répandue parmi les développeurs.

4. Nano Service (NS)

La granularité des microservices est sujette à interprétation et est fortement liée au contraintes d'affaire. Cependant, les développeurs doivent éviter d'avoir plusieurs microservices pour une seule et même fonctionnalité d'affaires, car cela peut conduire à un fort couplage et à des problèmes de déploiement et de maintenance.

MARS a détecté cet anti-patron avec une précision de 42.86% et un rappel de 37.5%. Ces résultats s'expliquent par le fait que les nano services sont subjectifs : ils nécessitent souvent l'avis d'un expert afin de déterminer si un découpage de microservices est trop granuleux ou pas.

Observation 4 : Nous observons que 33.33% (8/24) des systèmes à base de microservices analysés contiennent des nano services. Cet anti-patron est plus répandu que les mega services. La nature des systèmes dans l'ensemble de données à notre disposition peut aussi expliquer ce résultat car il s'agit souvent de projets non-commerciaux/non-industriels.

5. Shared Libraries (SL)

Les architectures à base de microservices sont dites « sans partage » car on y déconseille fortement le partage de bibliothèques et autres artefacts entre les microservices. Nous n'avons trouvé aucune occurrence de bibliothèques partagées entre les microservices dans notre ensemble de données, que ce soit en utilisant MARS ou en validant manuellement. De ce fait, nous n'avons pas pu calculer les valeurs de précision et de rappel pour cet anti-patron.

Notre (manque de) résultat montre que les développeurs sont conscients de l'importance de séparer les artefacts qu'ils utilisent au sein de leurs systèmes à base

de microservices.

Observation 5 : Les développeurs utilisent des librairies et artefacts séparés pour chacun de leurs microservices, ce qui constitue une bonne pratique pour la gestion et le déploiement indépendants de ces microservices.

6. Hardcoded Endpoints (HE)

Les microservices doivent communiquer en utilisant des mécanismes légers sans avoir à coder en dur les adresses des autres microservices. La découverte de services doit être utilisée afin de communiquer sans avoir à connaître à l'avance les adresses des autres microservices.

Notre outil a trouvé cinq occurrences d'adresses codées en dur dans l'ensemble de données, avec une précision de 60%. MARS n'a pas détecté quelques instances car le format des URLs varie et ces dernières peuvent être construites dynamiquement dans le code. Il est compliqué pour un outil de détecter toutes les adresses en utilisant seulement une analyse statique.

Observation 6 : Les développeurs utilisent généralement la découverte de services lorsqu'ils développent des systèmes à base de microservices. Seulement 20.83% des systèmes analysés contiennent des occurrences de cet anti-patron.

7. Manual Configuration (MC)

La configuration automatique est un principe de base des systèmes à base de microservices, c-à-d., charger dynamiquement la configuration des microservices afin d'accélérer le processus de développement.

MARS détecte correctement les occurrences de configuration manuelle avec une précision de 87.5% et un rappel de 77.78%. Nous constatons que moins de 50% des systèmes mettent en place une configuration automatisée.

Observation 7 : Même s'il existe des cadriciels pour simplifier la mise en place de configuration automatique, 54% des systèmes analysés continuent de compter une configuration manuelle de leurs microservices.

8. No CI/CD (NCI)

L'intégration et le déploiement continus sont fondamentaux dans les systèmes à base de microservices afin d'assurer leur déploiement rapide et de raccourcir leurs cycles de développement.

Nous constatons que MARS détecte avec précision les occurrences de cet anti-patron. Nous observons que seulement 20.83% des systèmes de l'ensemble de données utilisent l'intégration et le déploiement continus dans leurs chaînes de développement.

Observation 8 : Malgré l'importance de l'intégration et du déploiement continus dans les architectures à base de microservices, ce n'est pas une pratique très répandue dans les systèmes analysés. Cela peut être dû à la nature de notre ensemble de données, constitué principalement de projets non-commerciaux/non-industriels.

9. No API Gateway (NAG)

Les passerelles d'API permettent à plusieurs clients de se connecter à un système en utilisant un seul point d'entrée, ce qui augmente la sécurité et le potentiel de

mise à l'échelle du système et simplifie l'authentification et la gestion des autorisations.

MARS détecte correctement les occurrences de cet anti-patron avec une précision de 80% et un rappel de 88.89%.

Nous observons que seulement 9 systèmes sur 24 dans notre ensemble de données utilisent une passerelle d'API.

Observation 9 : Malgré les avantages des passerelles d'API, certains développeurs continuent d'ignorer cette pratique. Cela peut être dû à la complexité additionnelle introduite par sa mise en place ou par la nature des systèmes de notre ensemble de données (projets non-commerciaux/non-industriels).

10. Timeouts (TO)

Traiter chaque requête comme échouée lorsqu'aucune réponse n'a été reçue après un délai d'attente dépassé peut accroître la latence du système et réduire considérablement ses performances.

Spécifier des délais d'attente dans les microservices au lieu d'utiliser des *circuit breakers* est une pratique non recommandée pour les systèmes à base de microservices.

MARS détecte à tort deux occurrences de cet anti-patron car les systèmes concernés utilisent une méthode d'implémentation de gestion de requêtes échouées qui n'est pas encore supportée par notre cadriciel. Pour cet anti-patron, nous avons une précision de 71.4% et un rappel de 83.3%.

Nous observons que cet anti-patron est présent dans 25% des systèmes de notre ensemble de données. Les développeurs ont donc tendance à utiliser des disjoncteurs dans leurs systèmes à base de microservices pour gérer les potentiels échecs suite aux appels aux autres microservices.

Observation 10 : Les développeurs mettent rarement en place des délais d'attente dans leurs systèmes à base de microservices. Cela constitue une bonne pratique pour la tolérance aux pannes et la résilience. Les disjoncteurs sont souvent utilisés afin de protéger les systèmes de pannes potentielles.

11. Multiple Service Instances per Host (MSIH)

Chaque microservice doit être déployé sur son propre hôte afin de faciliter sa mise à l'échelle et d'éviter les potentiels conflits de technologies au sein d'un même hôte.

Les résultats de la détection de MARS sont satisfaisants malgré le fait que certaines occurrences sont faussement détectées.

Nous observons que 11 systèmes de notre ensemble de données (45.83%) sont déployés sur un hôte unique.

Observation 11 : Près de la moitié des systèmes à base de microservices de notre ensemble de données utilisent un hôte de déploiement commun à tous leurs microservices. Cette pratique doit être évitée afin d'améliorer la mise à l'échelle des systèmes et de permettre de faire fonctionner en même temps plusieurs versions d'un même microservice.

12. Shared Persistence (SP)

Chaque microservice d'un système à base de microservices devrait être responsable de ses propres données et ne pas partager de bases de données avec d'autres microservices.

Pour cet anti-patron, MARS obtient une précision de 100% et un rappel de 44%. Cela s'explique par le fait que plusieurs systèmes utilisent des technologies et des ORM (**M**apping **O**bjet-**R**elationnel) non pris en charge pour le moment par notre outil.

Nous observons que cet anti-patron apparaît dans au moins 37.5% des systèmes de l'ensemble de données.

Observation 12 : Les développeurs sont conscients de l'importance d'éviter de partager les bases de données entre les différents microservices de leurs systèmes.

13. No API Versioning (NAV)

La gestion des versions des APIs permet le déploiement de multiples versions d'un microservice et de maintenir la rétro-compatibilité avec les applications consommatrices. Cependant, implémenter ce patron est une tâche relativement difficile et ajoute de la complexité au système.

Nous avons obtenu une précision de 100% avec MARS pour cet anti-patron et un rappel de 75%.

Aucun système de notre ensemble de données n'implémente la gestion des versions d'APIs. Cela est probablement dû à la nature de notre ensemble de données, qui

est constitué principalement de petits systèmes non-commerciaux/non-industriels.

Observation 13 : Les développeurs ont tendance à ignorer la gestion des versions de leurs APIs. Cela peut être dû au ratio de la complexité ajouté par cette implémentation par rapport au gain obtenu pour des petits systèmes.

14. No Health Check (NHC)

Vérifier l'état de santé et la disponibilité des microservices d'un système à base de microservices est essentiel afin d'en assurer la tolérance aux pannes.

Nous constatons que MARS détecte efficacement les occurrences de cet anti-patron avec une précision de 66.67% et un rappel de 100%.

En se basant sur les résultats de notre analyse, nous remarquons que la moitié des systèmes de notre ensemble de données ne procèdent pas à la vérification périodique de l'état de santé de leurs microservices.

Observation 14 : NHC est un anti-patron très commun dans les systèmes à base de microservices. Sa présence peut causer la panne ou l'indisponibilité des systèmes où il est présent.

15. Local Logging (LL)

La journalisation distribuée facilite la gestion et l'agrégation des journaux d'évènements et aide à rapidement identifier les potentiels problèmes d'un système.

Nous n'avons pas obtenu de résultats satisfaisants pour cet anti-patron, puisque nous avons obtenu une précision de 45.45% et un rappel de 62.5%. Basé sur

notre analyse manuelle, nous observons que la journalisation locale est largement répandue dans les systèmes à base de microservices. Seulement huit systèmes sur 24 utilisent des solutions de journalisation distribuées.

Observation 15 : 66.67% des systèmes à base de microservices analysés utilisent des mécanismes de journalisation locale, pratique qui devrait être évitée car elle empêche le suivi efficace des pannes parmi les microservices.

16. Insufficient Monitoring (IM)

Le suivi est très important dans les architectures à base de microservices afin de retracer les éventuelles pannes le plus tôt possible.

MARS a généré plusieurs faux-positifs à la suite de l'analyse mais n'a manqué aucune occurrence de cet anti-patron (précision de 35% et un rappel de 100%). Ces valeurs s'expliquent notamment par la nature subjective de cet anti-patron. En effet, le suivi d'un système est relatif à la nature du projet, sa taille ainsi que l'industrie dans laquelle il opère.

Nous remarquons que les développeurs sont conscients de l'importance du suivi. En effet, 70.83% des systèmes de l'ensemble de données utilisent des outils et mécanismes de suivi.

Observation 16 : Le suivi des microservices est une pratique bien établie parmi les développeurs de systèmes à base de microservices. 17 systèmes sur 24 incluent des outils de suivi pour les microservices.

6.3 Discussions et recommandations

Nous avons abordé notre problématique de recherche avec le plus grand recul possible, cependant, nous savons que certains points peuvent constituer des menaces à la validité de nos résultats. Nous discutons ces points dans la section suivante, puis nous proposons des recommandations aux développeurs désirant minimiser l'impact des anti-patrons dans leurs systèmes à base de microservices et aux chercheurs désirant s'appuyer sur nos travaux afin de contribuer à la recherche dans le domaine des systèmes à base de microservices, et des architectures à base de microservices de façon plus générale.

6.3.1 Menaces à la validité

Malgré notre étude intensive et systématique de la littérature, nous ne prétendons pas proposer un catalogue entièrement exhaustif et incluant tous les anti-patrons existants dans les systèmes à base de microservices. En effet, nos critères de recherche ainsi que les mots-clés que nous avons utilisés peuvent ne pas couvrir l'intégralité des travaux dans le domaine. C'est pourquoi, nous avons tenté d'utiliser les termes les plus courants relatifs à notre sujet de recherche, et avons appliqué le *forward* et *backward snowballing* (Wohlin, 2014; Felizardo *et al.*, 2016).

Aussi, les règles de détection que nous avons établies pour détecter les anti-patrons de notre catalogue sont sujettes à notre interprétation des anti-patrons. L'anti-patron `Mega Service` par exemple est un anti-patron subjectif dont la définition et détection peuvent être discutées. Cependant, nous avons tenté de minimiser cette menace en considérant chaque microservice comme faisant partie du système et non comme un élément seul. De ce fait, nous pouvons considérer un service comme `Mega Service` *relativement* à son système d'appartenance.

En ce qui concerne le processus de détection des anti-patrons, il est important de noter que les systèmes à base de microservices sont très volatiles. En effet, ils peuvent être conçus en utilisant une large variété de langages de programmation et peuvent être déployés sur plusieurs environnements. Bien que nous ayons tenté d'identifier les technologies les plus communes de conception des systèmes à base de microservices, nous pouvons en avoir manqué certaines. Nous minimisons ce risque en concevant MARS de sorte à être extensible à d'autres technologies et à d'autres environnements de déploiement.

De plus, nous considérons, pour la détection de certains anti-patrons, des listes de bibliothèques et outils. Nous avons prédéfini ces listes en choisissant les technologies les plus utilisées dans chaque domaine. Cependant, nous ne prétendons pas être exhaustifs. C'est pourquoi, nous avons conçu MARS de sorte à ce que ces listes puissent être étendues pour intégrer d'autres bibliothèques et outils.

Les travaux de Walker et al. (Walker *et al.*, 2020) sont très semblables aux nôtres. En effet, l'approche des auteurs permet la détection automatique de 11 anti-patrons dans les systèmes à base de microservices. Cependant, bien que nous en partagions certains, nous détectons des anti-patrons que leur approche ne permet pas de détecter (Mega Service, Manual configuration, No CI/CD, Timeouts, Multiple Service Instances per Host, No Healthcheck, Local logging et Insufficient Monitoring) et inversement (ESB Usage, Too Many Standards et Inappropriate Service Intimacy). Nous appliquons aussi des approches relativement similaires quant aux anti-patrons que nous identifions dans nos deux travaux. De ce fait, nous considérons ces travaux comme complémentaires et comme un gage de viabilité et une preuve de l'intérêt de recherche que représente le sujet des anti-patrons dans les architectures à base de microservices.

6.3.2 Recommandations

À la suite de nos travaux, plusieurs points importants ont été notés et méritent d'être soulevés, à la fois pour les développeurs de systèmes à base de microservices que pour les chercheurs dans le domaine des architectures à base de microservices ou de leurs anti-patrons.

Nous constatons d'abord que la majorité des systèmes que nous avons analysés ne contiennent pas de Mega Services, tandis que les Nano Services sont présents dans un nombre plus important de systèmes et en plus grand nombre. Cela est principalement causé par l'idée générale que les microservices doivent être très petits. Cependant, il est très souvent problématique d'avoir beaucoup de Nano Services, car chaque microservice dans un système apporte une couche de complexité supplémentaire. De ce fait, nous recommandons aux développeurs de faire appel à un expert du domaine *avant* d'entamer le processus de décomposition et de réaliser cette tâche de concert. Cela évitera bon nombre de problèmes de coordination et de découpage lors de la maintenance et de l'évolution du système.

Aussi, plusieurs techniques très répandues dans les microservices peuvent s'avérer plus complexes que le problème qu'elles résolvent dans un certain contexte. C'est le cas par exemple des passerelles d'API (API Gateways). En effet, il est commun dans l'industrie de considérer ce patron de conception comme une complexité inutile en-dessous d'un certain nombre de microservices. Il serait intéressant pour les chercheurs d'étudier l'impact de la complexité ajoutée par une passerelle d'API dans un système de taille réduite comparativement à une communication directe avec les microservices. Cela étant, il est toujours préférable de concevoir un système à base de microservices en suivant un maximum de bonnes pratiques si ce dernier est indéniablement amené à évoluer dans le futur.

En ce qui concerne les procédés de développement des systèmes à base de microservices, il est très fortement recommandé d'automatiser le plus de tâches possible. En effet, l'automatisation est un principe fondamental des architectures à base de microservices et c'est là-dessus que se base une grande partie de leurs avantages. De ce fait, nous recommandons l'automatisation des tests, de la configuration, des déploiements et du suivi pour une agilité accrue et une réactivité plus importante.

Enfin, nous recommandons la journalisation et le suivi de toutes les informations et évènements émis par les microservices, non seulement pour améliorer la découverte d'erreurs et de pannes, mais aussi leur correction et la constitution de bases de connaissances permettant d'éviter leur reproduction future.

CONCLUSION

Les architectures à base de microservices sont en voie de devenir un style architectural majeur dans l'industrie. Plusieurs acteurs principaux de l'industrie logicielle tels que Amazon ou Netflix ont adopté ce style architectural avec un très grand succès. Un microservice est une unité indépendante, fournissant une seule fonctionnalité d'affaires, s'exécutant et se déployant de façon indépendante, et communiquant avec les autres services à travers des mécanismes simples et légers tels que les API REST. Chaque microservice d'un système à base de microservicesest géré par une seule équipe disposant de toutes les compétences nécessaires à son cycle de vie (développement, déploiement, maintenance, etc.)

L'utilisation des microservices progresse chaque jour grâce à leur nature dynamique et distribuée qui (1) offre une grande agilité et une efficacité opérationnelle importante et (2) réduit à la fois les coûts et la complexité de la mise à l'échelle et du déploiement.

Cependant, un manque de compréhension des concepts fondamentaux et principes de base de ce style architectural cause l'introduction de « mauvaises » solutions à des problématiques récurrentes, aussi appelées « anti-patrons », qui impactent négativement la qualité du système. Contrairement aux systèmes orientés objet, la nature dynamique et distribuée des microservices rend difficile la détection de tels anti-patrons.

Nous avons proposé MARS, une approche outillée entièrement automatisée permettant la spécification et la détection des anti-patrons dans les systèmes à base de microservices. Nous avons proposé un méta-modèle permettant d'abstraire les

informations d'un système à base de microservices, des anti-patrons et des règles de détection associées.

Nous avons spécifié 16 anti-patrons dont nous avons détecté et validé les occurrences dans 24 systèmes à base de microservices. Une validation manuelle des occurrences détectées a permis de constater que MARS a pu détecter les anti-patrons dans les systèmes à base de microservices avec une précision moyenne de 68.5% et un rappel moyen de 78.22%.

Nos travaux sont utiles à la fois pour les développeurs et les chercheurs. En effet, ils fournissent une approche complète de spécification et de détection d'anti-patrons dans les systèmes à base de microservices.

Nous considérons comme travaux futurs la spécification de plus d'anti-patrons et l'étude de leur prévalence dans les systèmes à base de microservices. Ainsi, nous pourrions recommander aux développeurs et aux chercheurs des bonnes pratiques à adopter et des mauvaises à éviter lors du développement de systèmes à base de microservices. Nous voulons aussi étudier empiriquement et quantitativement la présence des anti-patrons dans un ensemble de systèmes plus grand et étudier l'impact de ces anti-patrons sur la maintenance des systèmes à base de microservices. Enfin, nous désirons étendre MARS à d'autres langages de programmation et technologies de déploiement, afin de supporter un maximum de technologies utilisées dans les systèmes à base de microservices.

RÉFÉRENCES

Alagarasan, V. (2016). Microservices antipatterns. https ://www.youtube.com/watch ?v=uTGIrzzmcv8. Consulté : Janvier 2020.

Alshuqayran, N., Ali, N. et Evans, R. (2016). A systematic mapping study in microservice architecture. Dans *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, 44–51. IEEE.

Alshuqayran, N., Ali, N. et Evans, R. (2018). Towards micro service architecture recovery : An empirical study. Dans *2018 IEEE International Conference on Software Architecture (ICSA)*, 47–4709.

Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D. A. et Lynn, T. (2018). Microservices migration patterns. *Software : Practice and Experience*, *48*(11), 2019–2042.

Bogard, J. (2017). Avoiding microservices megadisaster.

Bogner, J., Boceck, T., Popp, M., Tschechlov, D., Wagner, S. et Zimmermann, A. (2019). Towards a collaborative repository for the documentation of service-based antipatterns and bad smells. Dans *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 95–101. IEEE.

Borges, R. et Khan, T. (2019). Algorithm for detecting antipatterns in microservices projects. Dans *Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution*, 21–29. CEUR-WS.

Carnell, J. (2017). *Spring microservices in action*. Manning Publications Co.

Carrasco, A., Bladel, B. v. et Demeyer, S. (2018). Migrating towards microservices : migration and architecture smells. Dans *Proceedings of the 2nd International Workshop on Refactoring*, 1–6.

Cicchetti, D. et Sparrow, S. (1981). Developing criteria for establishing interrater reliability of specific items : Applications to assessment of adaptive behavior. *American journal of mental deficiency*, *86*, 127–37.

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and psychological measurement*, *20*(1), 37–46.

Di Francesco, P., Lago, P. et Malavolta, I. (2019). Architecting with microservices : A systematic mapping study. *Journal of Systems and Software*, *150*, 77–97.

Di Francesco, P., Malavolta, I. et Lago, P. (2017). Research on architecting microservices : Trends, focus, and potential for industrial adoption. Dans *2017 IEEE International Conference on Software Architecture (ICSA)*, 21–30. IEEE.

Di Nucci, D., Palomba, F., Panichella, A., Zaidman, A. et Lucia, A. (2017). Lightweight detection of android-specific code smells : the adoctor project.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. et Safina, L. (2017). Microservices : yesterday, today, and tomorrow. In *Present and ulterior software engineering* 195–216. Springer.

Duvall, P., Matyas, S. et Glover, A. (2007). *Continuous Integration, Improving Software Quality and Reducing Risk*. Pearson.

Felizardo, K. R., Mendes, E., Kalinowski, M., Souza, É. F. et Vijaykumar, N. L. (2016). Using forward snowballing to update systematic reviews in software engineering. Dans *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 1–6. ACM.

Fowler, M. (2011). Polyglot persistence. https ://martinfowler.com/bliki/PolyglotPersistence.html. Consulté : Janvier 2020.

Garriga, M. (2018). Towards a taxonomy of microservices architectures. In *Software Engineering and Formal Methods* 203–218. Springer International Publishing.

Ghofrani, J. et Lübke, D. (2018). Challenges of microservices architecture : A survey on the state of the practice. Dans *ZEUS*, 1–8.

Granchelli, G., Cardarelli, M., Francesco, P., Malavolta, I., Iovino, L. et Salle, A. D. (2017). Towards recovering the software architecture of microservice-based systems. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 46–53.

Hanna, M. (1993). Maintenance burden begging for remedy. *Software magazine Westborough*, *13*, 53–53.

Hecht, G., Rouvoy, R., Moha, N. et Duchien, L. (2015). Detecting antipatterns in android apps.

Intelliware-Development (2018). Microservices : Drawing the line on dry. https ://www.intelliware.com/microservices-drawing-the-line-on-dry/. Consulté : Décembre 2019.

Kitajima, S. et Matsuoka, N. (2017). Inferring calling relationship based on external observation for microservice architecture. Dans *Service-Oriented Computing*, 229–237. Springer International Publishing.

Kitchenham, B. (2004). Procedures for performing systematic reviews. *Keele, UK, Keele University*, *33*(2004), 1–26.

Landis, J. R. et Koch, G. G. (1977). The measurement of observer agreement for categorical data. *Biometrics*, *33*(1), 159–174.

Lewis, J. et Fowler, M. (2015). Microservices a definition of this new architectural term. https ://martinfowler.com/articles/microservices.html. Consulté : Janvier 2020.

Marquez, G. et Astudillo, H. (2018). Actual use of architectural patterns in microservices-based open source projects. Dans *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE.

Microsoft (2019). Microservices : Data considerations. https ://docs.microsoft.com/en-us/azure/architecture/microservices/design/data-considerations. Consulté : Janvier 2020.

Moha, N., Gueheneuc, Y., Duchien, L. et Le Meur, A. (2010). Decor : A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, *36*(1), 20–36.

Nadareishvili, I., Mitra, R., McLarty, M. et Amundsen, M. (2016). *Microservice architecture : aligning principles, practices, and culture*. « O'Reilly Media, Inc. ».

Neri, D., Soldani, J., Zimmermann, O. et Brogi, A. (2019). Design principles, architectural smells and refactorings for microservices : a multivocal review. *SICS Software-Intensive Cyber-Physical Systems*, 1–13.

Newman, S. (2015). *Building Microservices : Designing Fine-Grained Systems*. O'Reilly Media, Inc.

Osses, F., Marquez, G. et Astudillo, H. (2018). Exploration of academic and industrial evidence about architectural tactics and patterns in microservices. Dans *Proceedings of the 40th International Conference on Software*

Engineering Companion Proceeedings - ICSE. ACM Press.

Pahl, C. et Jamshidi, P. (2016). Microservices : A systematic mapping study. Dans *Proceedings of the 6th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and and Technology Publications.

Palma, F. (2013). Detection of soa antipatterns. Dans A. Ghose, H. Zhu, Q. Yu, A. Delis, Q. Z. Sheng, O. Perrin, J. Wang, et Y. Wang (dir.). *Service-Oriented Computing - ICSOC 2012 Workshops*, 412–418. Springer Berlin Heidelberg.

Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J. et Josuttis, N. (2017). Microservices in practice, part 1 : Reality check and service design. *IEEE Software*, 91–98.

Pigazzini, I., Fontana, F. A., Lenarduzzi, V. et Taibi, D. (2020). *Towards Microservice Smells Detection*, Dans *Proceedings of the 3rd International Conference on Technical Debt*, (p. 92–97). Association for Computing Machinery : New York, NY, USA.

Rademacher, F., Sachweh, S. et Zündorf, A. (2020). A modeling method for systematic architecture reconstruction of microservice-based software systems. Dans S. Nurcan, I. Reinhartz-Berger, P. Soffer, et J. Zdravkovic (dir.). *Enterprise, Business-Process and Information Systems Modeling*, 311–326., Cham. Springer International Publishing.

Rahman, M. I., Panichella, S. et Taibi, D. (2019). A curated dataset of microservices-based systems.

Richards, M. M. (2016). Microservices antipatterns and pitfalls.

Richardson, C. (2020). Microservices—pattern : Application metrics. https ://microservices.io/patterns/observability/application-metrics.html. Consulté : Janvier 2020.

Salah, T. (2016). Microservices antipatterns. https ://www.youtube.com/watch ?v=I56HzTKvZKc.

Shadija, D., Rezai, M. et Hill, R. (2017). Towards an understanding of microservices. Dans *2017 23rd International Conference on Automation and Computing (ICAC)*, 1–6.

Soldani, J., Tamburri, D. A. et Heuvel, W.-J. V. D. (2018). The pains and gains of microservices : A systematic grey literature review. *Journal of Systems and Software*, *146*, 215–232.

Stocker, M., Zimmermann, O., Zdun, U., Lübke, D. et Pautasso, C. (2018). Interface quality patterns : Communicating and improving the quality of microservices apis. Dans *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, 1–16.

Taibi, D. et Lenarduzzi, V. (2018). On the definition of microservice bad smells. *IEEE Software*, *35*(3), 56–62.

Taibi, D., Lenarduzzi, V. et Pahl, C. (2018). Architectural patterns for microservices : a systematic mapping study. In *Proceedings of the 8th International Conference on Cloud Computing and Services Science*. SCITEPRESS.

Taibi, D., Lenarduzzi, V. et Pahl, C. (2019). *Microservices Anti Patterns : A Taxonomy*, Dans *Microservices : Science and Engineering*, (p. 111–128). Springer International Publishing.

Tighilt, R., Abdellatif, M., Moha, N. et Guéhéneuc, Y.-G. (2020a). Towards a tool-based approach for microservice antipatterns identification. Dans *Proceedings of The Twelfth International Conference on Advanced Service Computing (SERVICE COMPUTATION 2020), Nice, France*, p. pp. To appear.

Tighilt, R., Abdellatif, M., Moha, N., Mili, H., Boussaidi, G. E., Privat, J. et Guéhéneuc, Y.-G. (2020b). On the specification of microservices antipatterns. Dans *European Conference on Pattern Languages of Programs 2020 (EuroPLoP '20), Kloster Irsee, Bavaria, Germany*, p. pp. To appear.

Tighilt, R., Abdellatif, M., Saad, N. A., Moha, N. et Guéhéneuc, Y.-G. (2019). Collection and identification of microservices patterns and antipatterns. Dans *Proceedings of the Conference Francophone sur les Architectures Logicielles, Hammamet, Tunisie*, p. pp. To appear.

Walker, A., Das, D. et Cerny, T. (2020). Automated code-smell detection in microservices through static analysis : A case study. *Applied Sciences*, *10*(21), 7800.

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. Dans *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, 1–10. ACM.

Wolff, E. (2016). *Microservices : flexible software architecture*. Addison-Wesley Professional.

Zimmermann, O. (2016). Microservices tenets. *Computer Science - Research and Development*, *32*(3-4), 301–310.