Model-driven migration of monolithic applications towards a microservice-oriented architecture

Pascal Zaragoza

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR
DE L'UNIVERSITÉ DE MONTPELLIER

En Informatique

École doctorale : Information, Structures, Systèmes

Unité de recherche LIRMM

Model-driven migration of monolithic applications towards a microservice-oriented architecture

Présentée par Pascal Zaragoza
Le 18 Juillet 2022

Sous la direction de Abdelhak-Djamel Seriai,
co-encadré par Abderrahmane Seriai
et Hinde-Lilia Bouziane

Devant le jury composé de

Naouel MOHA, Professor, Université de Québec	Rapporteuse
Salah SADOU, Professor, Université Bretagne Sud	Rapporteur
David DELAHAYE, Professor, Université de Montpellier	Examinateur, Président du jury
Jannik LAVAL, Assoc. Prof., Université Lumière Lyon 2	Examinateur
Abdelhak-Djamel SERIAI, Assoc. Prof., HDR, Université de Montpellier	Directeur
Abderrahmane SERIAI, Assoc. Prof., Berger-Levrault	Co-encadrant
Hinde-Lilia BOUZIANE, Assoc. Prof., Université de Montpellier	Co-encadrante

UNIVERSITÉ DE MONTPELLIER

Berger Levrault

Acknowledgments

Une thèse n'est jamais vraiment faite toute seule. Même si la rédaction est un exercice solitaire, il reste vrai que beaucoup de gens apporte leurs idées, leurs avis, ou tout simplement leur soutien lors de cet exercice. C'est grâce à eux qu'il a été possible de réaliser ces travaux ainsi que cette thèse. Je souhaite donc remercier l'ensemble des personnes qui m'ont aidé pendant ces 3-4 ans.

Il se va de commencer par quelque part, et je souhaite donc commencer par les personnes qui ont eu un impact direct sur les travaux de cette thèse. Je souhaite donc d'abord remercier mon directeur de thèse, Djamel pour ces 3-4 dernières années d'encadrement et de soutien. Je remercie Hinde, pour m'avoir encadré et particulièrement pour ces sessions d'écriture qui n'est jamais un exercice évident. Puisque c'est une thèse Cifre, j'ai aussi eu l'opportunité d'avoir un encadrant d'entreprise. Pour cela, je souhaite remercier Abderrahmane qui a pu m'apporter une grande aide lors de cette thèse en proposant une vue scientifique mais aussi pratique sur ce sujet de recherche. Merci vous tous, pour cette confiance qui m'a permis de mener à bien mes travaux.

Je souhaite aussi remercier le jury de cette thèse pour avoir pris le temps de participer à cette soutenance. En particulier, les rapporteurs Naouel Moha et Salah Sadou pour leurs remarques pertinentes au sujet de cette thèse. Je remercie aussi Jannik Laval pour son rôle en tant qu'examinateur ainsi que David pour son rôle de président du jury.

Je souhaite remercier mes collègues à Berger-Levrault avec lequel nous avons pu passer des bons moments. Je pense à Michel, Benoit, Jimmy qui m'ont accueilli à l'entreprise et que je considère comme des bons amis. Il y a aussi Anas qui m'a apporté des conseils importants lors de la rédaction de papiers. Je souhaite aussi remercier Julien qui nous en rejoint en cours de route et qui est souvent de bon conseil. Je remercie Quentin et Bachar et je leur souhaite une belle aventure en thèse. Du côté du LIRMM, je souhaite remercier l'ensemble de l'équipe de MAREL pour m'avoir accueilli pendant cette thèse. Je me sent obligé de remercier l'équipe MAORE pour sa machine qui alimente l'ensemble des doctorants (Gabriel, Sam, Tom, Nicolas, Vincent, etc.) que j'ai pu côtoyer chaque midi et à chaque pause café pour des discussions philosophiques et scientifiques.

Je souhaite remercier tous mes amis qui ont été là pendant ces dernières années. J'en ai déjà cité plusieurs dont j'ai eu le plaisir d'aussi les avoir en tant que collègues. Je remercie aussi mes amis ERASMUS qui ne sauront pas lire cette phrase sans aide ! Je souhaite aussi remercier mes amis d'enfance de Péret qui sont comme de la famille...

Je souhaite remercier ma famille qui a toujours été là pour moi. A mes parents qui, grâce à eux, j'ai pu faire non seulement cette thèse mais toutes mes études en toute tranquillité. Je souhaite remercier mon frère pour ces 3-4 ans ensemble lors de ce parcours, en particulier pour les repas lors de la dernière année lorsque j'ai été occupé par la fin de thèse. Maintenant, cela va être à son tour de faire une thèse et j'espère qu'il arrivera à s'épanouir dans cette nouvelle étape de sa carrière. Au pire, il pourra toujours venir le weekend à Toulouse pour se plaindre de tout et de rien ! Je souhaite

remercier mes grand-parents, qui m'ont toujours soutenu et ont toujours été fiers de moi.

Enfin, à ma chérie, qui me soutien encore et toujours, j'espère être là pour toi à mon tour pour cette dernière année de thèse si difficile ainsi que l'aventure heureuse qui va la suivre. ♡

Abstract

The shift towards cloud computing has been largely promoted for its ability to decrease the deployment time and to reduce the operation costs. In turn, this shift towards cloud computing has promoted the development of new architectural styles to take advantage of its capabilities. Microservice-oriented architecture (MSA) is one of the latest style to have emerged. This architecture is organized around small services focused on specific business features, running in independent processes, and communicating through lightweight interfaces. These features, when paired with cloud computing and modern DevOps techniques, allow for easily-deployable, autonomous, and scalable applications.

Eager to take advantage of it, enterprises are slowly migrating their existing applications towards the Cloud. However, their legacy applications are oftentimes ill-adapted to the Cloud. In particular, legacy monolithic applications which are characterized by their large codebase which is generally harder to maintain, deploy, and scale. For these reasons, companies are eager to migrate their existing monoliths towards a microservice-oriented architecture. Nonetheless, the process of rewriting an application from scratch using the new architecture is undesirable, due to its costs and risks. Furthermore, companies face the need to migrate their whole software suite which are implemented in varying languages and frameworks. Therefore, not only do companies seek to automate the migration process but also repeat the process across multiple applications. Consequently, the need for a semi-automated, generic, and increasingly reusable migration process emerges.

In this thesis, we decompose the migration process into two research problems: (1) the identification of the target microservice-oriented architecture, and (2) the transformation of the source code towards valid microservice candidates. Indeed, the challenge requires understanding the monolith to extract the target microservice architecture. Secondly, once the target architecture is identified, the source code of the existing application must be transformed to conform to the MSA.

Furthermore, we propose an approach for each research problem, as well as a model-driven approach to guide the migration process from beginning to end. To evaluate our approaches, we implemented two different tools: (1) Mono2Micro, which implements our ad hoc transformation approach to migrate a JAVA-based application, and (2) MDE-Mono2Micro, which implements both our identification approach and our model-driven transformation approach into an end-to-end solution.

Résumé étendu

Les travaux présentés dans cette thèse sont le fruit de la collaboration entre le LIRMM[1] (Laboratoire d'Informatique, de Robotique, de Microélectronique de Montpellier) et la société Berger-Levrault[2]. Nous commençons donc par présenter le contexte scientifique de cette thèse, puis le contexte industriel. Ensuite, nous présentons la motivation de cette thèse et les principaux problèmes de recherche. Enfin, nous soulignons les principales contributions de cette thèse ainsi que sa structure.

Contexte Scientifique

Pour comprendre les raisons qui poussent les entreprises à vouloir moderniser leurs systèmes existants, il faut introduire plusieurs concepts. C'est pourquoi, dans cette section, nous présentons trois concepts différents : le cloud computing, le DevOps et les microservices. En outre, nous présentons également les avantages qu'ils présentent pour les entreprises qui les adoptent.

Cloud Computing

L'informatique en nuage, ou "cloud computing", est un modèle commercial permettant l'accès à un réseau à la demande à un pool partagé de ressources informatiques configurables (par exemple, réseaux, serveurs, stockage, applications et services) [MG11]. Ce pool de ressources peut être rapidement approvisionné et libéré avec un minimum de gestion ou d'interaction avec le fournisseur de services [MG11]. Ce type de modèle commercial s'oppose au serveur standard sur site, dans lequel un serveur physique sur site est géré et entretenu individuellement par son utilisateur.

Devops

DevOps (**Dev**elopment et **Op**eration**s**) est un ensemble de pratiques et de philosophie d'ingénierie logicielle qui utilise des équipes interfonctionnelles (développement,

[1] https://www.lirmm.fr/
[2] https://www.berger-levrault.com/fr/

opérations, sécurité et assurance qualité) pour construire, tester et publier des logiciels plus rapidement et de manière plus fiable grâce à l'automatisation [MB20a]. Essentiellement, chaque phase de développement et d'exploitation est placée dans un pipeline semi-automatisé initiant chaque phase l'une après l'autre. Au lieu de publier un logiciel après un long cycle de développement (par exemple, 3 à 4 semaines), les mises à jour peuvent être apportées quotidiennement à l'utilisateur final et les corrections de bogues peuvent être déployées en quelques heures.

Microservices

Les microservices, et plus particulièrement l'architecture orientée microservices (MSA), constituent la dernière tendance pour développer des applications sur le Cloud. Dans la littérature, de multiples définitions ont été proposées : [LF14b, Ric22, DGL+17]. Selon eux, le style architectural microservice est une approche visant à développer une application unique sous la forme d'un ensemble de **small** services, exécutés sur leurs propres processus, qui communiquent via des mécanismes dites "**légers**" (par exemple, gRPC, REST API, événements). Chaque microservice est construit autour de **capacités métiers** spécifiques, est déployable indépendamment (via un déploiement automatisé), autonome en termes de données, et peut être développé par de petites équipes indépendantes. Enfin, du fait qu'il s'agit de projets indépendants, chaque microservice peut être mis en œuvre à l'aide de différents langages, frameworks et technologies, ce qui rend l'architecture orientée microservices agnostique aux technologies.

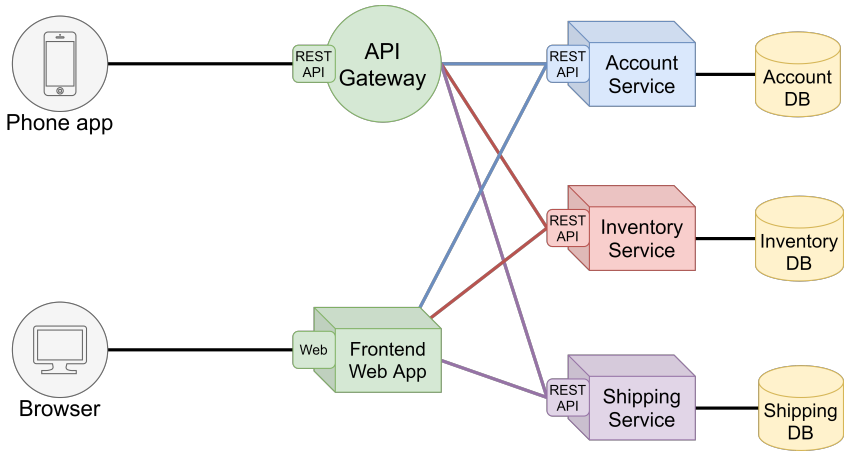

Figure 1: *Une representation simplifiée d'une architecture à base de microservices.*

Une représentation simple de l'architecture de microservices est présentée dans Figure **??**, qui illustre l'architecture comme un groupe de modules faiblement couplés communiquant via des mécanismes légers. En outre, dans cet exemple, nous illustrons la flexibilité de cette architecture, dans laquelle elle peut servir différentes applications frontales. Comme chaque microservice peut être déployé indépendamment, il peut être mis à l'échelle de façon dynamique en fonction de la demande du service [LF14b]. Associée à l'élasticité du cloud computing, l'évolutivité de l'architecture permet d'optimiser les ressources utilisées. Étant donné que chaque microservice est conçu pour être petit, le temps de démarrage est considérablement réduit par rapport aux applications traditionnelles.

En outre, l'architecture orientée microservices favorise les pratiques DevOps. En effet, de par sa conception, chaque microservice peut être développé et déployé rapidement et indépendamment les uns des autres. Ainsi, ils permettent l'intégration et le déploiement continus de l'application dans son ensemble, et permettent une plus grande rapidité d'exécution pour le développement de nouvelles fonctionnalités, et la maintenance d'urgence [Ric22].

Contexte Industriel

Berger-Levrault[3] est un éditeur de logiciels privé qui propose sa suite de solutions logicielles à une grande variété de clients. Leurs clients se trouvent aussi bien dans le secteur privé que dans le secteur public, et ils proposent plus de 150 produits dans des secteurs très variés (par exemple, la santé, l'administration publique, l'éducation). Ce large catalogue de logiciels s'explique facilement par la stratégie commerciale de l'entreprise, qui consiste à acquérir des sociétés existantes. Rien qu'en 2021, Berger-Levrault a acquis 2 sociétés différentes [4]. Cependant, à chaque acquisition, Berger-Levrault hérite des décisions architecturales et technologiques des autres. En d'autres termes, cette stratégie a un coût : ils doivent maintenir une grande variété de logiciels développés dans une multitude de langages et de frameworks avec des styles architecturaux différents.

D'une manière générale, alors que les organisations tentent de suivre les dernières tendances architecturales et d'éviter d'accumuler une dette technique, on constate une demande d'adaptation des systèmes existants au cloud computing, au DevOps et aux microservices [MB20b]. Pour lutter contre cette dette technique, profiter des dernières avancées architecturales et technologiques, et moderniser leurs systèmes existants, Berger-Levrault a mis en place une équipe de recherche et développement en génie logiciel (R & D). C'est dans ce cadre, que cette thèse a été entreprise.

En effet, Berger-Levrault comme toute autre entreprise souffre d'une dette technique. Des exemples de cette dette technique peuvent être reflétés par le temps de déploiement de certains produits. Un exemple d'application vieillissante est celle développée à Montpellier, qui nécessite une demi-journée pour réussir à déployer une mise à jour. Comme nous l'avons vu dans la section précédente, ce long temps de déploiement affecte le retour que les développeurs peuvent recevoir du système. Avec un temps de déploiement aussi long, l'entreprise est obligée de planifier à l'avance chaque version et les développeurs sont incapables de mettre en œuvre des techniques DevOps efficaces. Cela est dû au fait qu'un seul gros exécutable doit être redéployé à chaque mise à jour. Nous appelons communément ces grands exécutables des *monolithes*, et dans la section suivante, nous décrivons comment les applications monolithiques, sont structurées, leurs avantages et leurs limites inhérentes.

[3] https://www.berger-levrault.com/fr/
[4] https://www.berger-levrault.com/newsroom/

Verrous et Motivation

Le modèle économique du cloud computing réduisant le coût d'accès aux ressources informatiques, les entreprises sont impatientes de passer au cloud. En outre, les entreprises sont impatientes à l'idée d'adopter les techniques DevOps pour réduire leur délai de mise sur le marché et améliorer le cycle de développement. Avec la popularisation du cloud et de DevOps, le style architectural orienté microservices peut être considéré comme la prochaine étape logique pour tirer parti du cloud computing.

Néanmoins, les entreprises qui maintiennent des applications d'entreprise développées avant l'aube des microservices et qui souhaitent tirer parti du cloud doivent migrer leurs systèmes existants vers une architecture orientée microservices. Pour mieux comprendre pourquoi cette migration est nécessaire, il est important de comprendre l'architecture de ces anciens systèmes.

Les applications monolithiques et leurs limites

En général, les logiciels d'entreprise sont construits comme un système en trois parties : une interface utilisateur côté client, une base de données et une application côté serveur [LF14b, DGL+17]. L'application côté serveur est construite comme un exécutable logique unique et est appelée le **monolithe**. [LF14b]. Ces monolithes sont construits comme un système à un seul niveau avec plusieurs couches, à savoir : les couches de présentation, de logique métier et d'accès aux données.

Dans les premières étapes du développement d'applications monolithiques, l'architecture est simple et donc facile à développer, à tester et à déployer. Pendant la phase de conception initiale, les développeurs peuvent facilement comprendre et interagir avec l'ensemble de l'application. Cependant, à mesure que les applications monolithiques grandissent et vieillissent, elles deviennent plus complexes et plus difficiles à maintenir [DGL+17]. Avec le temps, elles tendent vers le modèle *Big Ball of Mud* proposé par [FY79], c'est-à-dire un code spaghetti structuré de manière désordonnée.

De plus, alors que les entreprises se tournent vers l'informatique en nuage, les monolithes sont mal adaptés pour en tirer parti. En effet, la mise à l'échelle d'un monolithe nécessite de multiplier l'ensemble du monolithe et de le placer derrière un équilibreur de charge. Par conséquent, l'utilisation de ses ressources ne peut se faire que par grands incréments, alors que dans une architecture orientée microservices, chaque service individuel peut être dupliqué selon les besoins. Sachant que le modèle économique du cloud computing repose sur un service mesuré, la surutilisation de ses ressources a un coût plus élevé. Par conséquent, l'architecture monolithique est fortement désavantagée par rapport à la MSA.

L'adoption des techniques DevOps nécessite de disposer d'un système qui peut être rapidement construit, testé et déployé [EGHS16]. Cependant, les monolithes ont tendance à être de grandes applications héritées, ce qui fait que de nombreux développeurs s'engagent sur la même base de code. Cela conduit à des états de construction instables fréquents de l'application et augmente le temps entre les constructions stables qui

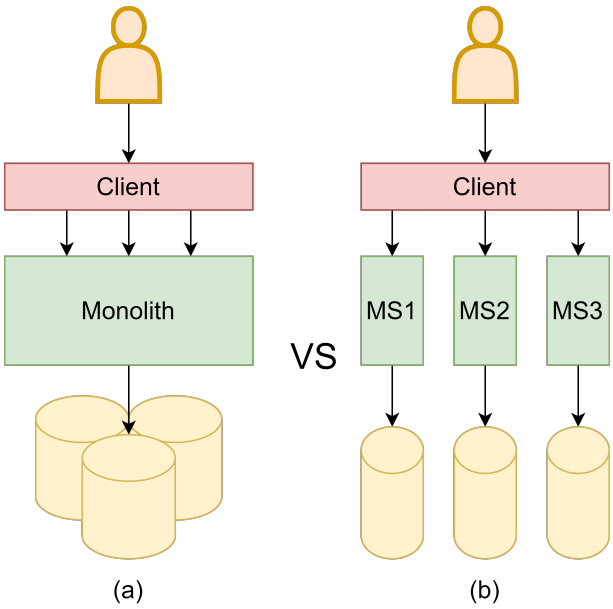

Figure 2: *(a) Illustration d'une application monolithique et (b) de l'architecture alternative basée sur les microservices.*

peuvent être testées et déployées. Ainsi, cela ralentit l'intégration du monolithe, et le retour d'information rapide attendu par les développeurs dans une méthodologie CI/CD n'est plus possible. A l'inverse, le découpage du monolithe en un ensemble de microservices permet une mise en œuvre efficace des techniques DevOps en favorisant les petites applications qui peuvent être construites plus rapidement [BHJ16]. En effet, le temps d'intégration entre les builds étant diminué, un feedback rapide aux développeurs devient possible.

Un autre aspect à considérer est le verrouillage technologique d'une application monolithique. En effet, lors de la phase de conception du monolithe, une pile technologique est choisie pour l'implémenter. Plus tard, si une nouvelle technologie apparaît, son intégration dans le monolithe peut s'avérer difficile, voire impossible. Ce n'est pas le cas avec un MSA, car les nouvelles fonctionnalités peuvent être développées sous forme de microservices distincts utilisant la nouvelle technologie, les microservices étant faiblement couplés et communiquant via des protocoles légers.

Pour ces raisons, des entreprises telles que Berger-Levrault estiment que les applications monolithiques ne sont pas adaptées au Cloud et aux pratiques DevOps. De plus, ces entreprises s'intéressent à l'architecture orientée microservices pour surmonter les limitations des applications monolithiques. Elles souhaitent donc migrer leurs applications monolithiques existantes vers une architecture orientée microservices pour bénéficier de sa plus grande flexibilité.

Migration des applications monolithiques vers un MSA

Nous avons établi que les systèmes patrimoniaux organisés sous forme d'applications monolithiques sont mal adaptés au modèle économique du cloud computing et aux techniques DevOps. De plus, les entreprises sont désireuses de tirer profit du Cloud et

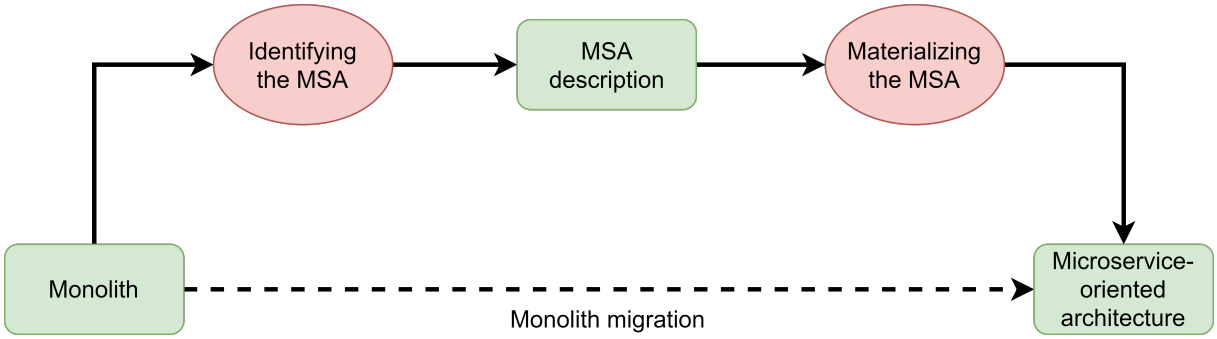

Figure 3: *Le flux de travail de la migration se divise en deux phases : la phase d'identification et la phase de matérialisation.*

des techniques DevOps, par l'adoption d'architectures orientées microservices. Elles cherchent donc à réécrire ou à faire migrer leurs systèmes existants vers une architecture orientée microservices.

L'une des options dont disposent les entreprises est de réécrire leurs systèmes *à partir de zéro*. Cependant, la réécriture complète d'une application est considérée comme une tâche risquée, coûteuse et longue, avec un risque élevé d'échec. Par conséquent, une migration est généralement une option plus sûre, avec plusieurs exemples de réussite documentés [BDD+18, FM17, SSBG20, LML20].

L'objectif principal de cette thèse est de contribuer à la migration d'applications monolithiques orientées objet vers une architecture orientée microservices. A cette fin, le problème de la migration de l'architecture d'une application existante vers une architecture orientée microservice se divise en deux problèmes de recherche différents :

1. **Ingénierie inverse, ou identification, de l'architecture orientée microservices à partir de l'application monolithique** : Pour réussir la migration d'une application vers une AMS, nous devons d'abord identifier l'architecture cible. Pour ce faire, nous pouvons analyser les artefacts du programme, afin de récupérer une description de l'architecture orientée microservice.
2. **Transformer le code existant pour matérialiser l'architecture orientée microservice** : Une fois l'architecture cible identifiée, il est possible de la matérialiser. Pour ce faire, le code source existant du monolithe doit être refactorisé pour produire des microservices valides.

Concrètement, nous divisons la migration en deux phases distinctes pour répondre à chaque problème (voir Figure 3) : la phase d'identification et la phase de transformation.

La phase d'identification est celle où l'architecture orientée microservices est récupérée à partir d'un monolithe orienté objet. Ce problème de recherche peut être réduit à un problème de clustering dans lequel le monolithe est vu comme un ensemble de classes qui doivent être partitionnées en un ensemble de clusters, chacun représentant un candidat possible de microservice [Ami18, SSB+20b]. L'objectif de la phase d'identification est de proposer une décomposition qui favorise les microservices hautement cohésifs et

faiblement couplés. De nombreuses approches ont été proposées [BGDR17, GKGZ16, Ami18, NSRS19, SRS20, ZLD+20, JLC+21] pour aborder la première phase du processus de migration en partitionnant l'implémentation OO d'une application monolithique donnée en clusters de classes qui représentent les différents microservices. Bien que les clusters résultants aident à comprendre le MSA cible, chaque cluster ne se traduit pas nécessairement par un microservice valide. En particulier, comme chaque classe est partitionnée en son propre microservice, les dépendances entre les classes appartenant à différents microservices demeurent.

L'objectif de la phase de transformation de la migration est de refactoriser le code source monolithique pour matérialiser des microservices exécutables conformes à la MSA cible, tout en préservant la logique métier de l'application. Dans le cas d'applications OO, la principale difficulté est de transformer les dépendances OO entre les clusters de classes en dépendances MSA. Ces transformations doivent respecter les principes du refactoring (c'est-à-dire préserver la logique métier) sans dégrader les performances. Cependant, malgré l'importance de la deuxième phase de la migration, ce n'est que récemment, en 2021, que des approches ont proposé de traiter ce problème : [AS20, FSC+21].

Contributions

Dans cette thèse, nous présentons 3 contributions principales au problème de la migration d'une application monolithique vers une architecture orientée microservices. Pour expliquer les contributions, nous complétons cette section avec Figure 4 pour placer chaque contribution dans le workflow de migration. Dans la figure, nous proposons en entrée le code source d'une application monolithique vers deux processus de migration différents. Le premier processus de migration est la migration ad hoc d'une application monolithique orientée objet, tandis que le second processus est piloté par le modèle.

En explorant l'état de l'art, nous avons observé que les approches ascendantes (c'est-à-dire les approches qui utilisent le code source comme entrée principale) ne prennent pas en compte l'architecture interne des applications monolithiques. En effet, les applications actuelles ne sont pas purement orientées objet et s'appuient sur un framework qui tente d'abstraire les fonctionnalités génériques, laissant l'implémentation de la logique métier à l'utilisateur. Ces frameworks s'appuient sur l'inversion du contrôle[5], qui modifie le flux de contrôle et la manière dont les classes écrites par l'utilisateur interagissent entre elles. De plus, des frameworks tels que Spring[6], les frameworks basés sur Node.JS[7], ou ASP.NET Core[8] s'appuient tous sur une architecture technique en couches pour promouvoir la séparation des préoccupations. Par conséquent, nous avons entrepris de proposer une approche pour répondre au problème de recherche lié à l'identification d'un MSA en prenant en considération l'architecture

[5] https://en.wikipedia.org/wiki/Inversion_of_control
[6] https://spring.io/
[7] https://nodejs.org/en/
[8] https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-5.0

en couches internes des applications monolithiques. C'est dans ce contexte que nous présentons **Contribution #1**, en proposant une approche qui prend en considération les frameworks qui sont inhérents à tout système moderne. De plus, nous tirons parti de l'architecture en couches internes, que ces cadres favorisent, pour identifier des candidats microservices de qualité. Dans ce contexte, nous proposons de récupérer l'architecture interne du monolithe et de la représenter via notre propre métamodèle. A partir de ce modèle extrait, nous identifions les microservices tout en préservant l'architecture en couches au sein de chaque microservice.

Dans cette thèse, nous nous sommes également concentrés sur le problème de recherche concernant la transformation du code source du monolithe vers un MSA. Notre raisonnement était que la littérature se concentrait principalement sur le problème de recherche lié à l'identification d'un MSA et qu'il manquait des travaux sur la matérialisation du code source du MSA. De plus, notre équipe avait déjà travaillé sur le problème d'identification [SSB+20b]. En particulier, ils ont proposé une approche pour identifier de manière semi-automatique les candidats microservices en regroupant un ensemble de classes qui composent le monolithe. En plus du code source, des recommandations d'experts ont été utilisées pour améliorer l'identification.

Par conséquent, notre deuxième contribution porte sur le remaniement du code source monolithique orienté objet afin de matérialiser l'architecture orientée microservices (voir **Contribution #2** dans Figure 4). Dans cette contribution, nous présentons une approche ad-hoc qui prend le code source d'un monolithe OO et le remanie pour le rendre conforme à l'architecture cible identifiée. Elle s'appuie sur un processus automatisé pour analyser le code source du monolithe par rapport à l'architecture cible, et identifier les différents points de refactoring requis. Ensuite, en fonction du type de refactoring requis détecté, un pattern de transformation est appliqué. Enfin, chaque candidat microservice est packagé et configuré dans son propre projet.

Enfin, **Contribution #2**, bien qu'applicable à tout langage orienté objet, était limité par sa mise en œuvre qui ne couvre que les systèmes basés sur Java. De plus, d'autres approches proposées pour migrer les monolithes vers un MSA se limitent également aux systèmes basés sur JAVA (par exemple, [FSC+21], [FFC21]), et ne proposent pas d'approche générique. Dans ce contexte, nous présentons **Contribution # 3**, qui utilise des techniques d'ingénierie dirigée par les modèles pour proposer une approche de migration générique et extensible, pouvant être réutilisée dans différents contextes. Dans la première partie (**Contribution # 3.1**), nous intégrons l'approche d'identification dans un workflow piloté par les modèles de bout en bout pour migrer une application monolithique orientée objet. De plus, nous présentons également un ensemble de métamodèles et de transformations de modèles (**Contribution #3.2**) pour représenter l'architecture de microservices identifiée et générer le code source cible. Dans ce flux de travail, nous utilisons spécifiquement les travaux présentés dans **Contribution #1**, cependant, d'autres approches d'identification peuvent être appliquées.

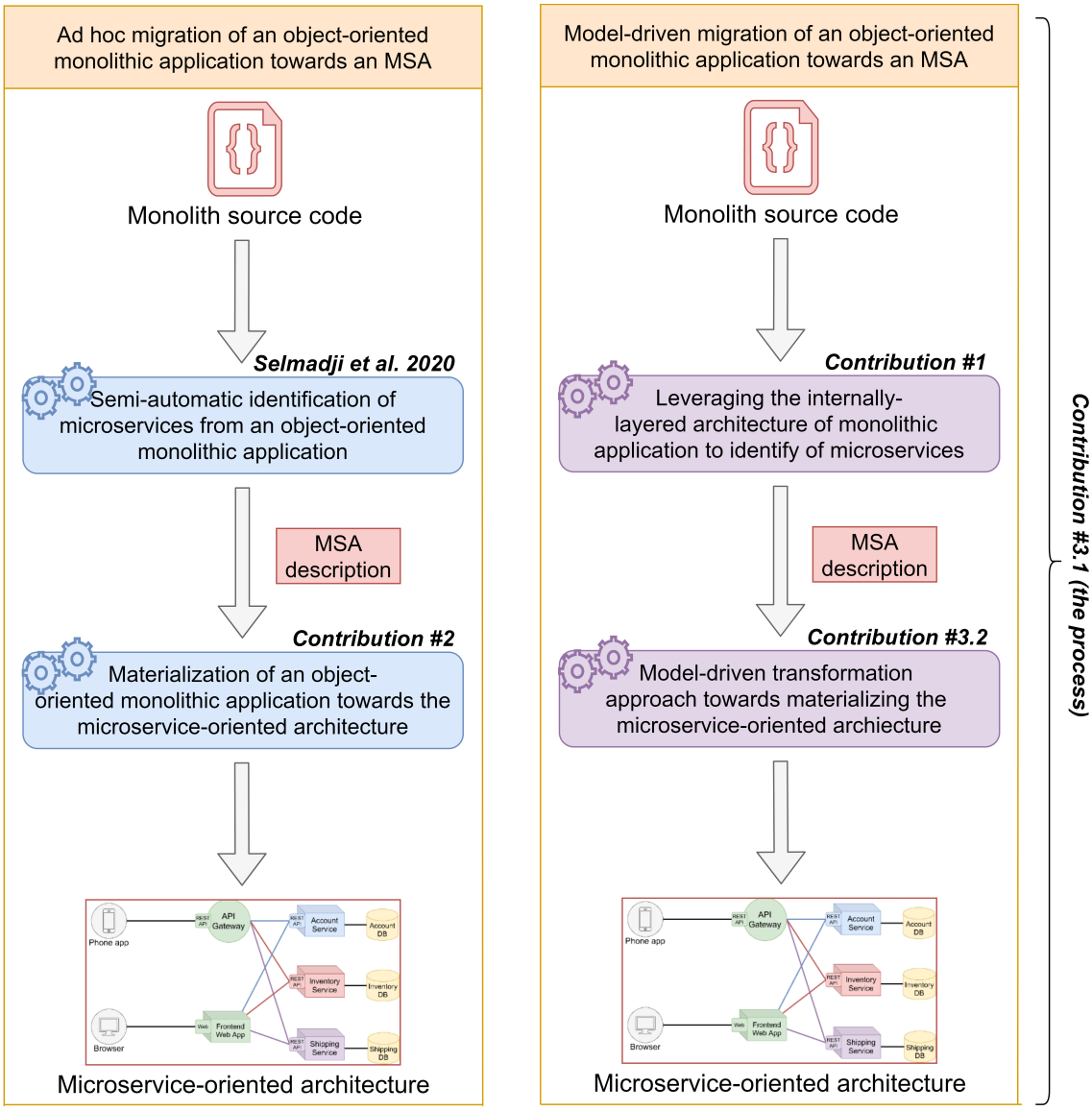

Figure 4: *Les contributions proposées dans le workflow de la migration.*

Contents

I

Introduction

Contents

The work presented in this thesis is the fruit of the collaboration between the LIRMM[1] (Laboratory of Informatics, Robotics, Microelectronics of Montpellier) and the company Berger-Levrault[2]. Therefore, we first begin by introducing the scientific context of this thesis, followed by the industrial context. Then, we present the motivation behind this thesis and the main research problems. Finally, we highlight the main contributions of this thesis as well as its structure.

[1] https://www.lirmm.fr/

[2] https://www.berger-levrault.com/fr/

1.1 Scientific Context

To understand the motivation behind companies wanting to modernize their existing systems, several concepts need to be introduced. Therefore, in this section we present three different concepts: cloud computing, DevOps, and microservices. In addition, we also present what advantages they present to companies that adopt them.

1.1.1 Cloud Computing

Cloud computing is the concept of enabling access to an on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) [Ray18, MG11]. These resources can be rapidly provisioned and released with minimal management or interaction with the service provider [MG11]. These resources are made available based on the business model "pay-as-you-go", where the usage of these resources are metered. In other words, the consumer pays for the resources they use. This type of business model contrasts with the standard on-premise server in which a physical, on-site server is managed and maintained individually by its user (see Figure 1.1). We divide cloud-computing into three types of services: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [RCL09]. In Figure 1.1, we illustrate the differences between each service and the traditional on-premise model.

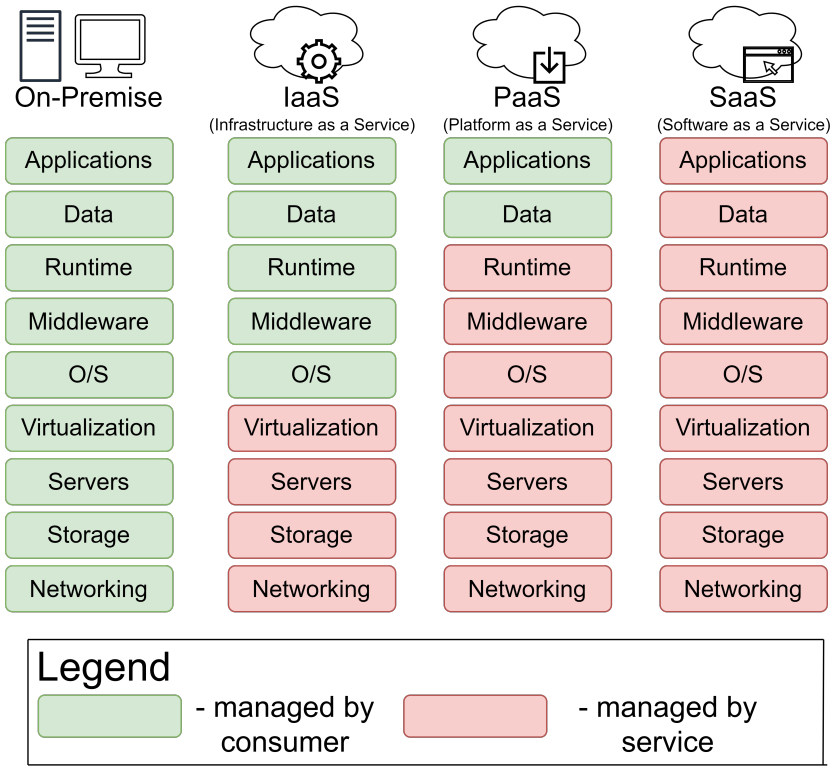

Figure 1.1: *Differences between each cloud-computing service model and the on-premise model.*

Concretely, Infrastructure as a Service (IaaS) involves providing the consumer with

the capability to provision processing, storage, networks, and other fundamental computing resources directly. The consumer is able to deploy and run arbitrary software, which can include operating systems and applications [MG11]. Examples of IaaS include Amazon's Elastic Compute Cloud[3], Google Cloud Platform[4], and Microsoft's Azure[5].

Meanwhile, Platform as a Service (PaaS) is a service which provides to the consumer the capability to deploy their own application onto the cloud infrastructure. These consumer-created applications can be created using languages, libraries, and tools supported by the provider [MG11]. In contrast with IaaS, the consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage. Instead, they can only control and manage the deployed applications, its data, and the configuration settings for the application-hosting environment.

Finally, Software as a Service (SaaS) is a service which provides to the consumer with the capability to use the provider's applications running on a cloud infrastructure. These applications are typically made accessible to various devices through either a light client interface, such as a web browser (e.g., web-based email such as Gmail) or a dedicated smartphone or desktop application [MG11]. At this level, the consumer does not manage or control the underlying cloud infrastructure, instead they can be seen as a user of the software provided by a consumer of a PaaS/IaaS.

Essentially, the cloud computing model proposes an on-demand self-service in which anyone can use to provision the required resources for their projects. These resources are provided in an abstract way, facilitating their use. Furthermore, cloud systems control and optimize resource use through automatic metering capabilities which allow for better transparency for both provider and consumer of the utilized service (i.e., cost flexibility) [MG11]. Also, these resources are set up to be quickly provisioned and released, to scale up/down rapidly based on the demand of the consumers (i.e., high elasticity) [MG11].

This contrasts with the upfront investment in an on-premise infrastructure, in which companies are locked into and cannot easily scale based on their changing requirements [JAP13]. For these reasons, companies are eager to migrate toward the cloud [JAP13]. Furthermore, when combined with efficient DevOps practice, it enables a streamline from the development to the deployment of the software towards the Cloud.

1.1.2 DevOps

DevOps (**Dev**elopment and **Op**eration**s**) is a set of software engineering practices and philosophy that utilizes cross-functional teams (development, operations, security, and quality assurance) to build, test, and release software faster and more reliably through automation [MB20a]. Essentially, each phase of development and operations is guided

[3] https://aws.amazon.com/fr/ec2/
[4] https://cloud.google.com/
[5] https://azure.microsoft.com/en-us/

by a semi-automated pipeline initiating each phase one after the other (see Figure 1.2). Instead of releasing software after a long cycle of development (e.g., 3-4 weeks), updates can be brought to the end user daily, and bug fixes can be deployed in hours.

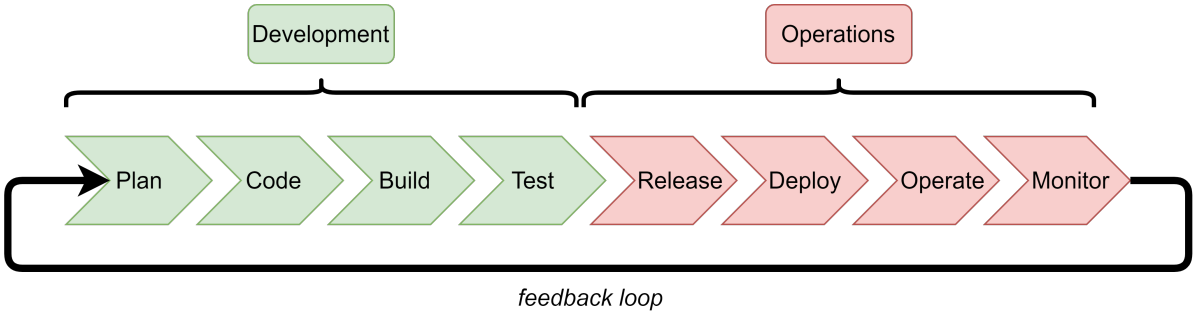

Figure 1.2: *DevOps Scheme.*

Particularly, this can be achieved by adopting the CI/CD method. Continuous Integration (CI) requires the developer to commit the code several times in a day followed by automatic build and test and immediate feedback to the developer whenever any bug is encountered [AGC18]. While, Continuous Deployment (CD)[6] is a key practice for making software development process reliable and faster. The feedbacks from the Production and Operations team are made available to the developer at frequent stages facilitating improvement and automation [AGC18, Che18]. Together, they create a virtuous circle, or positive feedback loop, by allowing developers to quickly react to effect of their changes on the stability of the software after integration and the deployment.

1.1.3 Microservices

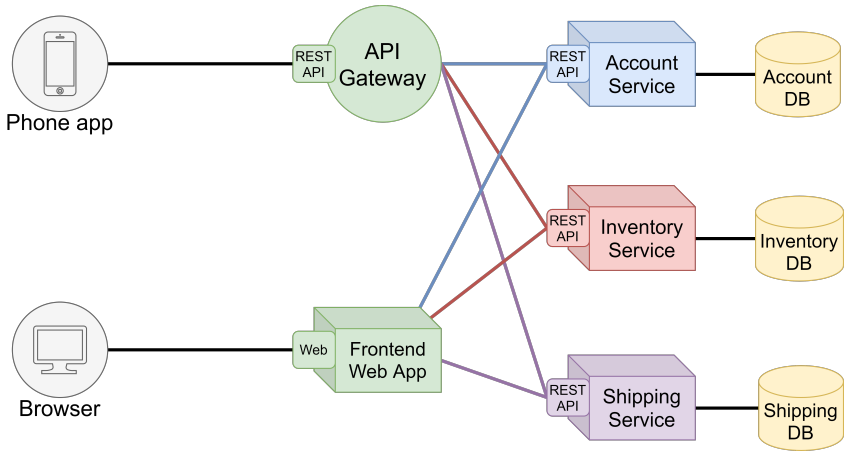

Figure 1.3: *A simple illustration of a microservice-oriented architecture.*

Microservices, and more particularly the microservice-oriented architecture (MSA), is one of the latest trends to develop applications on the Cloud. In the literature, multiple definitions have been proposed [LF14b, Ric22, DGL+17]. According to them, the microservice architectural style is an approach to developing a single application as a set of **small** services, running on their own processes, which communicate

[6] CD can also stand for Continuous Delivery, but for simplicity's sake we refer to deployment

via **lightweight** mechanisms (e.g., gRPC, REST API, events). Each microservice is built around specific **business capabilities**, is independently deployable (via automated deployment), data autonomous, and can be developed by small independent teams. Finally, by virtue of being independent projects, each microservice can be implemented using different languages, frameworks and technologies, thus making the microservice-oriented architecture language-agnostic.

A simple representation of the microservice architecture is depicted in Figure 1.3, which illustrates the architecture as a group of loosely-coupled modules communicating via lightweight mechanisms. Furthermore, in this example we show the flexibility of this architecture, in which it can serve different frontend applications. Moreover, each microservice is independently deployable, they can be independently scaled up-/down dynamically based on the demand of the service [LF14b]. When paired with cloud computing's elasticity, the scalability of the architecture allows to optimize the resources utilized. Since each microservice is made to be small, the start-up time is considerably lower than traditional applications[7].

Furthermore, the microservice-oriented architecture promotes DevOps practices. Indeed, by virtue of its design, each microservice can be developed and deployed quickly and independently of each other. Thus, they enable continuous integration and deployment of the application as a whole, and allows for a greater turn around for the development of new features, and emergency maintenance [Ric22].

1.2 Industrial Context

Berger-Levrault[8] is a private software editor offering its suite of software solutions to a wide variety of clients. Their customers can be found in both the private and the public sector, and they offer more than 150 products in a wide variety sectors (e.g., health, public administration, education). This large catalog of software can be easily explained through the company's business strategy of acquiring existing companies. In 2021 alone, Berger-Levrault has acquired 2 different companies [9]. However, with each acquisition, Berger-Levrault inherits the architectural and technological decisions of others. In other words, this strategy comes with a cost: they must maintain a wide-variety of software developed in a multitude of languages and frameworks with different architectural styles.

In general, as organizations try to keep up with the latest architectural trends, as well as avoid accumulating technical debt, there has been a demand to adapt legacy systems to cloud computing, DevOps, and microservices [MB20b]. To combat this technical debt, take advantage of the latest architectural and technological advances, and modernize their existing systems, Berger-Levrault has set up a software engineering research and development (R&D) team. It is within this framework, that this thesis was undertaken.

[7] See Section 1.3.1 on monolithic applications.
[8] https://www.berger-levrault.com/fr/
[9] https://www.berger-levrault.com/newsroom/

Indeed, Berger-Levrault like any other companies suffers from technical debt. Examples of this technical debt can be reflected by the time of deployment of certain products. An example of an aging application is one developed in Montpellier, SeditRH, which proposes a solution for the management related to Human Resources (HR). SeditRH is one of several products proposed by the team in Montpellier among Omaje which we use to validate our approach in Chapter 5. The particularity with SeditRH is that it is particularly large and requires half of a day to successfully deploy an update. As we've seen in the previous section, this long time to deploy affects the feedback that developers can receive from the system. With such long deployment time, the company is forced to plan ahead for each release and developers are unable to implement effective DevOps techniques. This is due to having one large executable that must be redeployed upon each update. We commonly refer these large executables as *monoliths*, and in the next section, we describe how monolithic application, are structured, their advantages, and their inherent limitations.

1.3 Problems & Motivation

As the cloud computing business model lowers the cost of access to computing resources, companies are eager to transition to the Cloud. Furthermore, companies are eager about the idea of adopting DevOps techniques to lower their time-to-market and improve the development cycle. With the popularization of the Cloud and DevOps, the microservice-oriented architectural style can be considered the next logical step to take advantage of cloud computing.

Nevertheless, companies that maintain enterprise applications developed before the dawn of microservices and wish to take advantage of the Cloud must migrate their existing systems towards a microservice-oriented architecture. To better understand, why this migration is necessary, it is important to understand the architecture of these legacy systems.

1.3.1 Monolithic applications and their limitations

In general, enterprise software are built as a 3-part system: a client-side user interface, a database, and a server-side application [LF14b, DGL+17]. The server-side application is built as a single logical executable and is referred to as the **monolith** [LF14b]. These monoliths are built as a single-tiered system with multiple layers, namely: presentation, business logic, and data-access layers.

In the early stages of developing monolithic applications, the architecture is straightforward and therefore easy to develop, test, and deploy. During the initial conception phase, developers can easily understand and interact with the entire application. However, as monolithic applications grow and age, they become more complex and harder to maintain [DGL+17]. With time, they tend towards the *Big Ball of Mud* pattern proposed by [FY79], which is to say a haphazardly structured spaghetti code.

Furthermore, as companies shift to using cloud computing, monoliths are ill-adapted to take advantage of it. Indeed, scaling a monolith requires multiplying the entire monolith and placing it behind a load-balancer. As a consequence, its resource utilization can only be made in large increments, whereas in a microservice-oriented architecture, each individual service can be duplicated as needed. Knowing that the cloud computing business model rests upon a metered service [MG11], over-utilization of its resources comes at a greater cost. Consequently, the monolithic architecture is greatly disadvantaged when compared to the MSA.

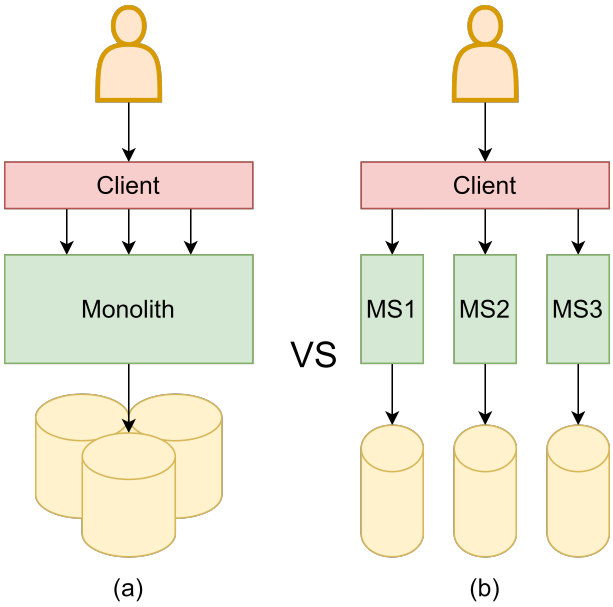

Figure 1.4: *(a) An illustration of a monolithic application, and (b) the alternative microservice-based architecture.*

The adoption of DevOps techniques requires having a system that can be quickly built, tested, and deployed [EGHS16]. However, monoliths tend to be large legacy applications which results in many developers committing to the same code base. This leads to frequent unstable build states of the application and increases the time between stable builds which can be tested, and deployed. Thus, this slows the integration of the monolith, and the rapid feedback expected by developers in a CI/CD methodology is no longer possible. In contrast, splitting the monolith into a set of microservices enables effective implementation of DevOps techniques by promoting small applications which can be built more quickly [BHJ16]. Indeed, as the integration time between builds is decreased, rapid feedback to the developers becomes possible.

Another aspect to consider is the technological lock-in of a monolithic application. Indeed, during the conception phase of the monolith a technological stack is chosen to implement it. Further down the line, if a new technology appears, integrating it into the monolith may prove difficult or even impossible. This is not the case with an MSA, as new features can be developed as a separate microservice using the new technology, as microservices are loosely-coupled and communicate via lightweight protocols.

For these reasons, companies such as Berger-Levrault believe that monolithic applications are not adapted to the Cloud and to DevOps practices. Furthermore, these companies are interested into microservice-oriented architecture to overcome the lim-

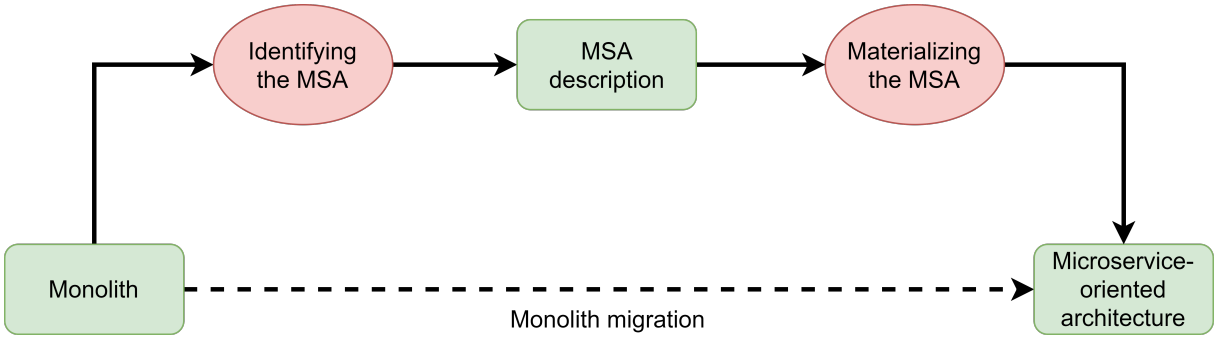

Figure 1.5: *The migration workflow divided into two phases : the identification and the materialization phase.*

itations found with monolithic application. Therefore, they wish to migrate their existing monolithic applications towards a microservice-oriented architecture to benefit from its greater flexibility.

1.3.2 Migrating monolithic applications toward an MSA

We have established that legacy systems organized as monolithic applications are ill-adapted to cloud computing business model and DevOps techniques. Moreover, companies are eager to take advantage of the Cloud and DevOps techniques, through the adoption of microservice-oriented architectures. Therefore, companies seek to rewrite or migrate legacy systems towards a microservice-oriented architecture.

One option companies have is to rewrite their systems *from scratch*. However, the complete rewrite of an application is considered to be a risky, costly, and time-consuming task with a high risk of failure [BLWG99a]. As a consequence, a migration is usually a safer option with several documented success stories [BDD+18, FM17, SSBG20, LML20].

The main goal of this thesis is to contribute to the migration of monolithic object-oriented applications towards a microservice-oriented architecture. To this end, the problem of migrating the architecture of an existing application towards a microservice-oriented one can be split into two different research problems:

1. **Reverse engineering, and identifying, the microservice-oriented architecture from the monolithic application**: To successfully migrate an application towards an MSA, we must first identify the target architecture. To do so, we can analyze program artifacts, in order to recover a description of the microservice-oriented architecture.
2. **Transforming the existing code to materialize the microservice-oriented architecture**: Once the target architecture has been identified, it can be materialized. To do so, the existing source code of the monolith must be refactored to produce valid microservices.

Concretely, the migration is divided into two distinct phases to address each prob-

lem (see Figure 1.5): the identification and the transformation phase.

The identification phase is where the microservice-oriented architecture is recovered from an object-oriented monolith. This research problem can be considered as a clustering problem in which the monolith is seen as a set of classes which must be partitioned into a set of clusters, each representing a possible microservice candidate [Ami18, SSB+20b]. The goal of the identification phase is to propose a decomposition that promotes highly-cohesive and loosely-coupled microservices. Many approaches have been proposed [BGDR17, GKGZ16, Ami18, NSRS19, SRS20, ZLD+20, JLC+21] to address the first phase of the migration process by partitioning the OO implementation of a given monolithic application into clusters of classes that represent the different microservices. Although the resulting clusters help understand the target MSA, each cluster does not necessarily translate into a properly-defined microservice. In particular, as each class is partitioned into its own microservice, any dependencies between classes belonging to different microservices remains.

The goal of the transformation phase of the migration is to refactor the monolithic source code to materialize run-able microservices that conform to the target MSA, while preserving the business logic of the application. In the case of OO applications, the main difficulty is to transform the OO dependencies between the clusters of classes into MSA ones. These transformations must adhere to refactoring principles (i.e., preserve the business-logic) without degrading the performance. However, despite the importance of the second phase of the migration, only as recently as 2021 has there been approaches that propose to address this issue [AS20, FSC+21].

1.4 Thesis contributions

In this thesis, we present 3 main contributions to the problem of migrating a monolithic application towards a microservice-oriented architecture. We illustrate our contributions in Figure 1.6 to place each contribution within the migration workflow. In the figure, we propose two different migration processes that takes as input the source code of a monolithic application. The first migration process is the ad hoc migration of object-oriented monolithic application, while the second process is the model-driven one.

1.4.1 Leveraging the internal layered architecture to identify an MSA

The first problem that we address is that of the majority of bottom-up approaches (i.e., approaches that use the source code as their main input) fail to consider the internal architecture of monolithic applications. Indeed, today's applications are not purely object-oriented and rely on a framework which attempts to abstract generic functionalities, leaving the implementation of the business logic to the user. These frameworks rely on the inversion of control[10], which changes the flow of control and how user-

[10] https://en.wikipedia.org/wiki/Inversion_of_control

written classes interact with each other. Furthermore, frameworks such as Spring[11], Node.JS-based frameworks[12], or ASP.NET Core[13] all rely on a technically-layered architecture to promote a separation of concerns. Therefore, we set out to propose an approach to address the research problem related to the identification of an MSA by taking into consideration the internally-layered architecture of monolithic applications. It is in this context that we present our first contribution (see **Contribution #1** in Figure 1.6), by proposing an approach that takes into consideration the frameworks that are inherent in any modern system. Furthermore, we leverage the internally-layered architecture, which these frameworks promote, to identify quality microservice candidates. In this context, we propose to recover the internal architecture of the monolith and represent it via our own metamodel. From this extracted model, we identify microservices while preserving the layered architecture within each microservice.

1.4.2 Microservice materialization

In this thesis, we also focused on the research problem regarding the transformation of the monolith's source code towards an MSA. Our reasoning was that the literature focused mainly on the research problem related to identifying an MSA and there lacked work in the materialization of the source code of the MSA. Moreover, our team had previously worked on the identification problem [SSB+20b]. Particularly, they proposed an approach to semi-automatically identify microservice candidates by clustering a set of classes which make up the monolith. In addition to the source code, expert recommendations were used to improve the identification.

Therefore, our second contribution is towards refactoring monolithic object-oriented source code to materialize the microservice-oriented architecture (see **Contribution #2** in Figure 1.6). In this contribution, we present an ad-hoc approach that takes the source code of an OO monolith and refactors it to conform to the identified target architecture. It relies on an automated process to analyze the source code of the monolith with regard to the target architecture, and identify the different required refactoring points. Then, based on the type of required refactoring detected, a transformation pattern is applied. Finally, each microservice candidate is packaged and configured into its own project.

1.4.3 Model-driven migration approach

Finally, while the second contribution is applicable to any object-oriented language, it was limited by it implementation which only covers Java-based systems. Furthermore, other approaches proposed to migrate monoliths towards an MSA also limit themselves to JAVA-based systems (e.g., [FSC+21], [FFC21]), and do not propose a generic approach. In this context, we present our third contribution (see **Contribution # 3** in Figure 1.6), which uses model-driven engineering techniques to propose a

[11] https://spring.io/
[12] https://nodejs.org/en/
[13] https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-5.0

generic, and extendable migration approach which can be reused in different contexts (i.e., languages). In the first part (see **Contribution # 3.1** in Figure 1.6), we integrate the identification approach in an end-to-end model-driven process to migrate a monolithic object-oriented application. Furthermore, we also present a set of metamodel and model transformation rules as **Contribution #3.2** to represent the identified microservice architecture and generate the target source code. In this workflow, we specifically use the approach proposed as our first contribution (**Contribution #1**), however, other identification approaches can be integrated.

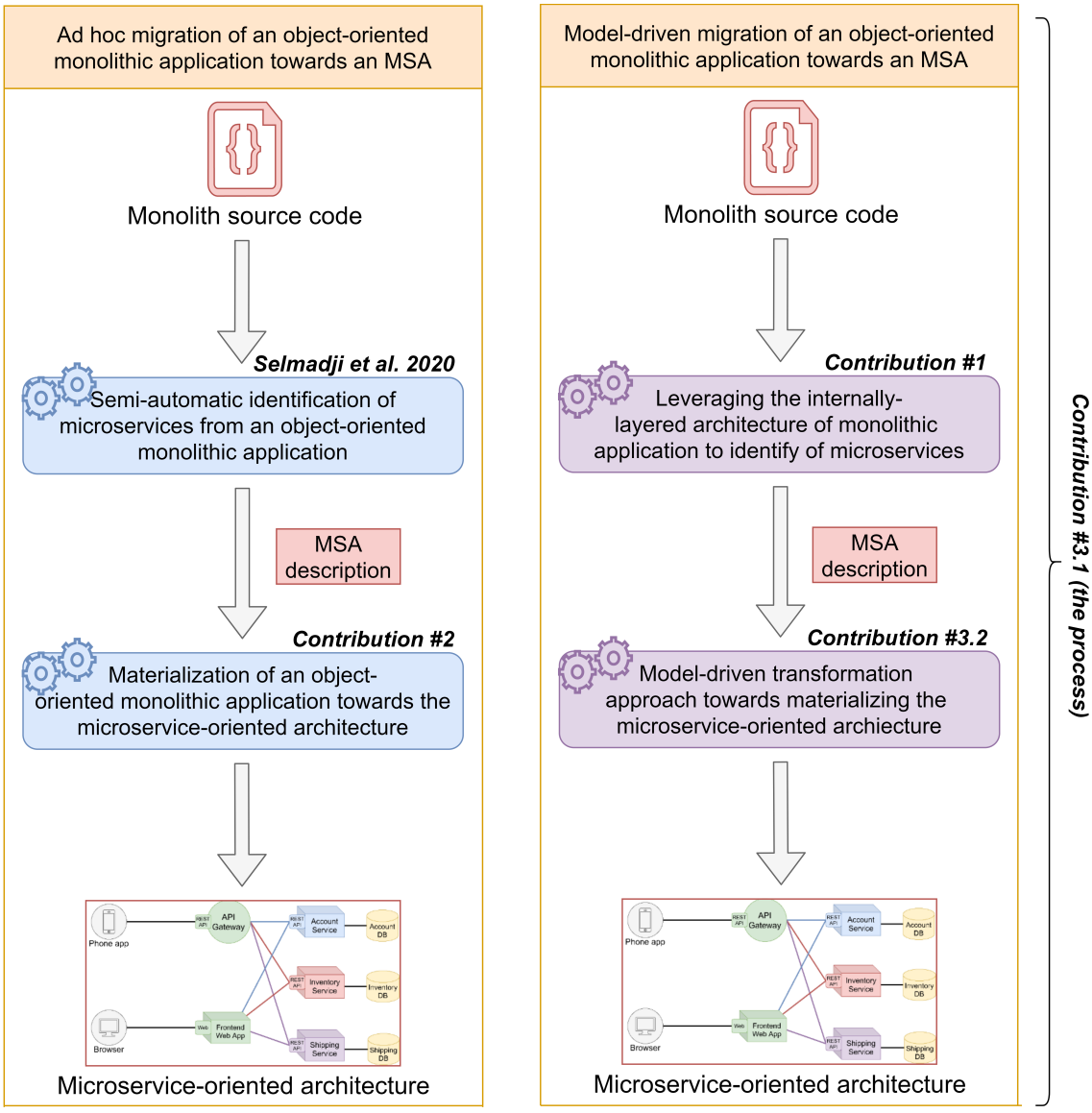

Figure 1.6: *The proposed contributions within the migration workflow.*

1.5 Structure of the Thesis

We have organized this thesis as follows:

- Chapter 2 presents the state-of-the-art concerning the migration of a monolithic application towards microservice-oriented architecture.
- Chapter 3 presents our contribution to identify a microservice-oriented architecture by leveraging the internal architecture found in modern applications.
- Chapter 4 presents our contribution for materializing the identified architecture from the source code of an object-oriented monolithic application.
- Chapter 5 presents our contribution for an end-to-end model-driven approach for the migration towards a microservice-base architecture.
- Finally, in Chapter 6 we summarize and conclude on the contributions presented in this thesis, and we provide several perspectives towards future works and challenges.

II

State of the Art

Contents

The goal of this chapter is to present the main concepts of the research domain of software migration, as well as the related work to the contributions presented in this thesis. In Section 2.1, we present the main concepts of software migration and its two main steps: reverse-engineering (incl. restructuration) and source code transformation. In Section 2.2, we present the taxonomy related to the reverse-engineering, or identification, of a microservice-oriented architecture. In Section 2.3, we present the related work for transformation of monolithic applications towards the identified architecture. In Section 2.4, we discuss the state-of-the-art of this thesis and the limitations of the literature in relation to the goal of this thesis. Finally, we conclude this chapter in Section 2.5.

2.1 Introduction to Software Migration

A legacy software system corresponds to a system built upon an obsolete language, platform, architecture, or a mix of all three. The system is still able to fulfill the needs for which it was built. However, over time its maintenance has become increasingly expensive and difficult [MB20b]. To palliate this effect, software evolution offers to modernize legacy system through what is called *software migration*. Software migration facilitates the shift of legacy systems to new environments that allow them to be more easily maintained and adapted to meet new business requirements without redeveloping it from scratch [BLWG99b].

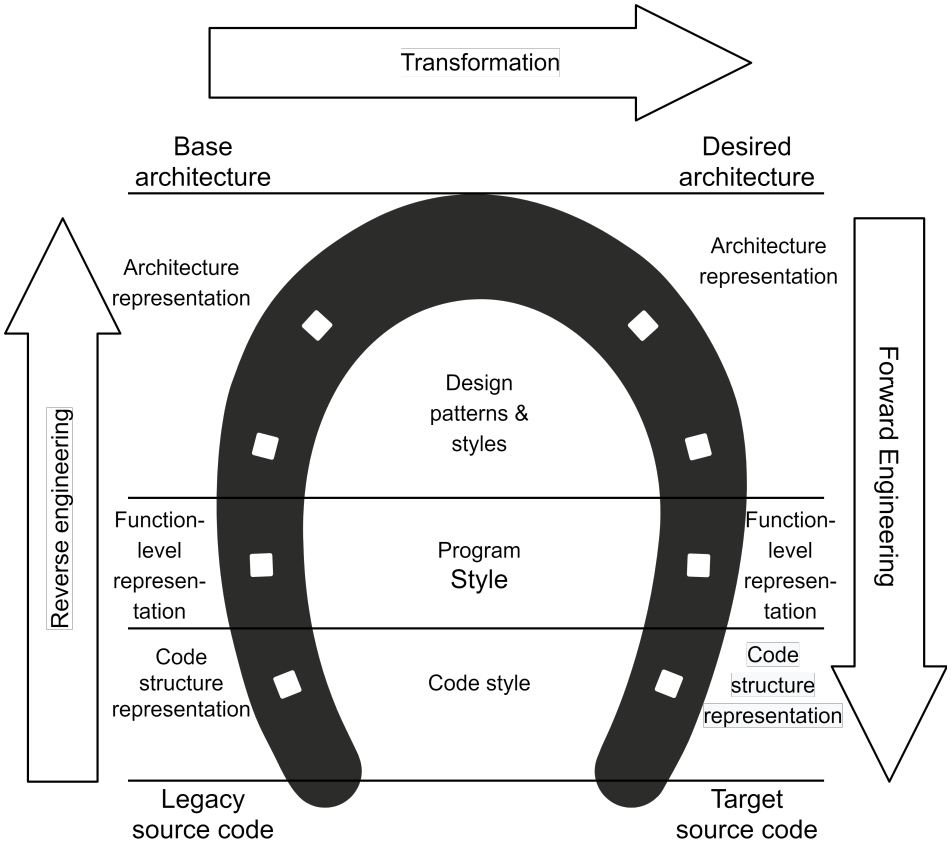

Figure 2.1: *The horseshoe model applied on architecture-driven software migration [SPL03].*

Originally proposed in [KWC98], the *Horseshoe Model* is a visual metaphor to describe the "integration of code-level and architectural reengineering views of the world". This model was later extended to describe the methodology of *architecture-driven* migration [SPL03]. Figure 2.1 illustrates the extended horseshoe model. The first step of the migration process is the *reverse-engineering* of the high-level abstract representation (i.e., base architecture) from the legacy source code. The step of reverse-engineering goes through several steps of abstraction from the source code, to the code structure representation (e.g., an AST[1] of the source code), to an architectural representation. In the next step, the base architecture is transformed using a set of transformation rules to produce the desired architecture. Finally, during the forward engineering step, the desired architecture is used to produce more concrete representations such as the target

[1] Abstract Syntax Tree

source code. Generally, during the migration process we focus on the first two steps: the reverse-engineering and the transformation, and include the code generation as technical implementation of the transformation step.

2.1.1 Software reverse-engineering

The IEEE standard for software maintenance defines reverse-engineering as a process of extracting software system (e.g., documentation) from the source code [IEE98]. Chikofsky and Cross describe reverse-engineering as a process to analyze a system to (1) identify the system's artifacts and their dependencies and (2) create representations of the system [CC90]. This representation can be a higher-level abstraction of the system, or it can be a representation of the system in another form [CC90]. Reverse-engineering has, among others, two subfields of interest: re-documentation and design recovery [CC90].

Re-documentation is the process of creating or recovering the documentation about a system. Tools for re-documentation include diagram generators, API documentation, etc. With these tools, the main goal is to visualize otherwise invisible relationships within the system. Indeed, whether an initial documentation is created during the conception of a system, as the system ages the documentation can drift until it no longer represents it. Then, it becomes necessary to extract an updated view of the system to increase its comprehension. One such design recovery approach is proposed in [GCD+17] to recover the architecture of an existing microservice-oriented application.

Design recovery goes further by recreating design abstractions from a combination of source code, design documentation, expert knowledge, and the system's domain [Big89]. This is also the initial step of the extended horseshoe model presented in Figure 2.1. Indeed, to evolve a legacy system understanding and having an updated representation is necessary.

2.1.2 Software Transformation

Reverse-engineering is a process of introspection, not a process of change or replication [BLWG99b]. In other words, it does not involve changing or creating a new system based on the reserve-engineered system. The step in the migration process after reserve-engineering is the restructuring, or transformation step. Software restructuration, is the process of changing one representation form to another at the same relative abstraction level, while preserving the functional and semantic behavior of the system [CC90]. It can take place at any abstraction level such as the source code (i.e., source-to-source transformation), or the architectural representation (i.e., model-to-model transformation). This is not to be confused with reverse-engineering which involves creating a higher-level abstraction than the initial model, nor forward-engineering which involves creating a more concrete representation of the model.

A model transformation can be defined as a set of transformation rules that specify how to change a system's model representation to a model in the target architecture

[KWB03]. Furthermore, a transformation rule is a description of how entities of the source model can be transformed into entities of the target model. The model being transformed can be the source code itself, any abstract representation of the source code, or a combination of both. Led by the reverse engineering effort, the transformation rules and patterns are applied to the initial source code representation, to produce the target source code representation. Finally, the valid source code is generated from these representations.

2.2 Taxonomy of Microservice Identification Approaches

To address the problem of identifying a microservice-oriented architecture from an existing legacy system, a significant quantity of approaches have been proposed. This section aims to extract, categorize, and discuss the goals and limitations of the different papers that present a microservice identification approach (MIA).

The classification of identification approaches is not new endeavor. Indeed, the migration of OO applications towards newer paradigms such as component-based architecture or service-oriented architecture. Several works have been produced to categorize architecture identification approaches in the field of reverse-engineering. For instance, [AS16] proposes a categorization for component identification approaches that migrate object-oriented applications towards component-based ones. In this work, the author presents a taxonomy based on 5 different categories: source, reverse engineering, transformation, target, and goal. Additionally, [Sha15] proposes a taxonomy for component identification which focuses on 4 main categories: input, process, output, and goal. More recently, [ASM+21] proposes a taxonomy on service identification approaches (SIA). Similarly to [Sha15], they focus on categorizing SIAs based on input, process, and output. However, the authors of [ASM+21] also consider the usability of the identification approaches. Finally, [Sel19] proposes a taxonomy of microservice identification approaches that, like the other taxonomies focuses on the input, process, and output of each approach. Additionally, they focus on the objective of the approaches.

To summarize, these taxonomies mainly focus on categorizing the extracted approaches based on the input of the approach, the type of algorithm used (or process), and the output produced by the approach. In the case of [Sha15] and [Sel19], they also focus on the quality metrics used to guide the identification. Additionally, [ASM+21] also take into consideration the usability of each approach. In the taxonomy presented in this thesis, we focus on categorizing each MIA based on the following 3 dimensions: input, process, and output. While we wanted to present a taxonomy on the migration of monolithic applications towards microservice-oriented architectures (like in the work of [AS16]), none of the approaches we extracted from the literature proposed an approach that covered both research problems of the migration process.

For this taxonomy, we extracted a total of 33 MIAs. Concretely, we categorize the extracted MIAs using the taxonomy schema illustrated in Figure 2.2. We divide this section into 3 parts, each covering a specific dimension, their different attributes, and the categorization of each approach within the dimension. Additionally, we conclude

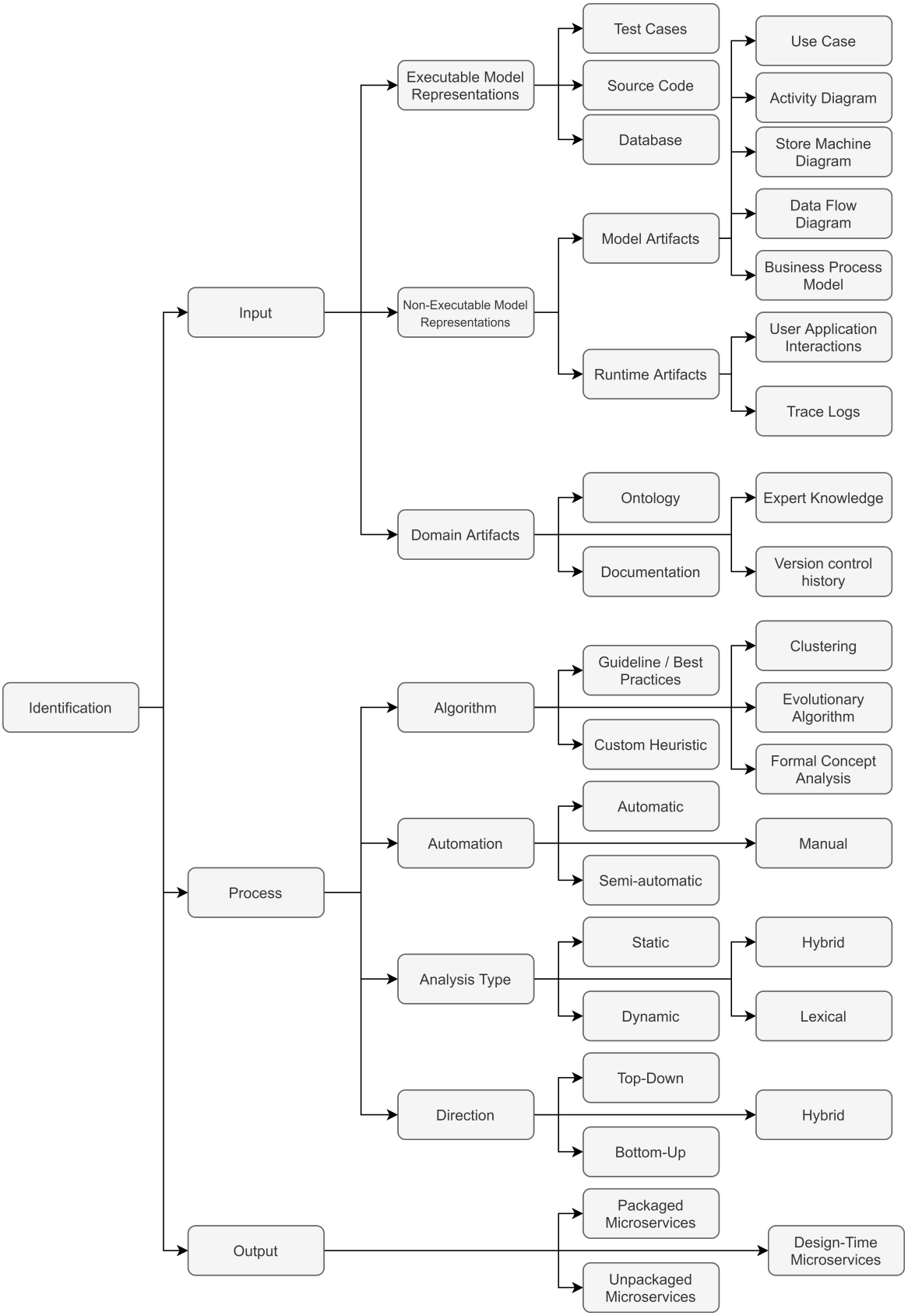

Figure 2.2: *Taxonomy of microservice identification approaches inspired by the works of [SSB+20b], [Sha15], and [ASM+21].*

this section with a discussion on the result of the taxonomy.

2.2.1 Input of Microservice Identification Approaches

The required inputs by an identification approach are categorized into 3 distinct groups: (1) executable model representations, (2) non-executable model representations, and (3) domain artifacts. Approaches can use one or more different type of inputs based on their requirements. Table 2.1 presents the extracted MIAs based on the inputs they use.

2.2.1.1 Executable Model Representations

We distinguish three different sub-categories of executable model representations, *source code*, *database*, and *test cases*.

Source Code: As a system ages, it tends to become the only reliable source of information available. This may be why it is the most used artifact from the extracted MIAs, with more than 50% of approaches using it as the primary source of information. The source code is used to extract dependencies between program artifacts (e.g., classes, methods/functions, attributes, etc.) to better understand the system and extract higher-level information. Oftentimes, MIAs partition program artifacts such classes into a set of clusters based on these dependencies. For instance, [AM21] and [KXK+21] both extract classes from the systems and partition them based on their cohesion.

Database: The database model representation includes the database schemas, stored procedures and database contents. Approaches that require the database as a source of information, use it group program artifacts based on the data they consume. This practice stems from domain-driven design which focuses on modeling a system based on the domain described by an expert [Eri03]. In [LTV16] the database schema is used, along with the interfaces and business logic, to create a 3-set of entities to represent the monolith. The 3-set is then partitioned to create a set of microservice candidates. Similarly, in [DABPF18], the database is also used as an input to guide the decomposition of the source code. Alternatively, the approach presented in [GBMM20] maps the stored procedures of the database to the system requirements through a structured process to discover the microservice candidates.

Test Cases: A test case is a description of the inputs, execution conditions, and expected outcomes that is run to meet a given software testing goal. Test cases are used in the approaches presented in [JLZ+18] and [JLC+21]. In both MIAs, test cases are used to run different features (e.g., account management) of the targeted application to generate log traces. Through these log files, they are able to map the classes and their dependencies. In [JLZ+18], they apply a clustering algorithm to group classes into highly cohesive and loosely couple clusters that become microservice candidates. Alternatively, in [JLC+21], they generate *functional atoms* by grouping highly-cohesive

Approach	Executable Model Representation	Non-Exec. Model Representation	Domain Artifacts
[LTV16]	source code, database		
[ECA+16]	source code		
[GKGZ16]			documentation, expertise
[BGDR17]			documentation (OpenAPI)
[MCL17]	source code		VCS
[CLL18]		DFD	
[DABPF18]	source code, database		
[Ami18]		BPMN	
[JLZ+18]	source code, test cases	log traces	
[KYHM18]	source code		
[AIE19]	executable	log traces	
[BSG20]	source code		
[NSRS19]	source code		
[SOMS19]	source code		
[PFM19]	source code		
[ACC+21]	source code		documentation
[ERM20]		log traces	
[GBMM20]	database (stored procedures)		documentation
[KXL+20], [KXK+21]	source code	use cases, log traces	
[KZH+20]	source code		expertise (domain analysis)
[LO20]	source code		VCS
[SRS20]	source code		expertise
[SSB+20b]	source code, database		
[TS20]	source code	log traces	
[ZLD+20]	executable	log traces	
[MCF+20]	source code	log traces	
[ADM20], [AM21]	source code		documentation (OpenAPI)
[JLC+21]	source code, test cases	log traces	
[SQMC21]	source code		expertise (domain analysis)
[BCS21]	source code		
[DMF+20], [DEF+21]		BPMN	ontology
[ASS+21]	source code, database		
[ZSS+22]	source code, database		expertise

Table 2.1: *Classification of the extracted MIAs based on their processes.*

classes, and then apply a search-based algorithm to group the atoms into a set of microservice candidates.

2.2.1.2 Non-executable Model Representations

We distinguish two different sub-categories of non-executable model representations, *model artifacts* and *runtime artifacts*.

Model Artifacts: Model artifacts serve to abstract the structure and behavior of a system. This can include business process models, dataflow diagrams, use cases, activity diagrams, or store machine diagrams. Business process models (BPM) describe a set of activities and tasks coordinated in a business environment to accomplish an organizational goal [Ami18]. In the investigated approaches, two MIAs ([Ami18] & [DEF+21]) use BPMs represented in BPMN (Business Process Model and Notation) to perform a top-down[2] decomposition of a monolith. Dataflow diagram is a type of graphical representation of data flow through an information system. It is used in [CLL18] to perform a top-down decomposition of the dataflow diagram to identify design-time microservice candidates. The use case corresponds to a set of actions performed by the system based on the interactions with an actor. This actor can be either human or an external system. In the approach proposed by [KXL+20], the use case are used to collect use-case-based execution traces on the application. We denote that none of the extracted approaches use the following: activity diagrams, store machine diagrams.

Runtime Artifacts: Runtime artifacts include any artifact that can be collected from the execution of a system. We include two types of runtime artifacts, trace logs and user-application interactions. Software tracing involves a specialized use of logging to record information about a program's execution. The output is commonly used for debugging and diagnosis purposes. In the case of microservice identification, it can be used to create a graph representation of program artifacts and their dependencies. Specifically, the MIA presented in [JLZ+18] proposes to run test cases to generate traces of the system. From those traces, they are able to build a dependency graph which is partitioned using a clustering technique. Similarly, user-application interactions can be collected and to study the relationship between the user and the system. In the MIAs extracted for this taxonomy, none propose to use user-application interactions.

2.2.1.3 Domain Artifacts

We distinguish four different sub-categories of executable model representations, *documentation*, *expert knowledge*, *ontology*, and *version control history*.

Documentation: Software documentation includes any file that attempts to document information on the system. It can include information about the system from the point of view of the end user or the developer. When available, documentation

[2]Direction of an analysis which starts from high-level artifacts such as BPMs

accompanies the source code of the system to assist the developer. One type of documentation that is often written is the specification of the API implemented by the system, which is used by developers which wish to consume its API. This type of documentation is the primary input used by the MIAs proposed in [BGDR17] and [ADM20, AM21]. In [BGDR17], they propose to cluster the system's API based on their semantic similarity by analyzing the OpenAPI specification[3]. Similarly, the MIA proposed in [ADM20, AM21] uses the OpenAPI specification to extract the operations exposed by the monolith. [GBMM20] uses the system requirements defined in the system's documentation to identify features of interests which are then mapped to the stored procedures of the system. Finally, [ACC+21] maps the documented features, to a set of program artifacts (i.e., methods) which are organized into a dependency graph to be decomposed using an evolutionary algorithm.

Ontology: An ontology can be defined as a structured set of terms and concepts representing the meaning of a field of information. The ontology of a system is used in [DMF+20, DEF+21] to establish the semantic dependency between the different activities of the BPM.

Version Control History: Version control system (VCS) is a type of system for managing changes in a project. It has become essential for modern software development as it allows for a large group of developers to collaborate on the same system while minimizing the time spent by developers to coordinate different versions of their system[4]. The changes brought to the system are recorded and attributed to specific developers by VCS tools. These records can be used to infer evolutionary coupling information [LO20]. This concept of evolutionary similarity is presented in [LO20], and it proposes that program entities that are changed together are similar. Similarly, the MIA proposed in [MCL17] uses version control history to group evolutionary similar program entities.

Expert Knowledge: Experts can also be included in the microservice identification process to include knowledge of the system that cannot be found in the documentation. It can also be used to supplement existing inputs, such is the case of [SSB+20b]. In their works, the authors propose a semi-automatic approach which can optionally accept different type of inputs from the expert on the ideal microservice architecture (e.g., the number of microservices, the center of gravity for each microservice). Similarly, both [SQMC21] & [ACC+21] identifies a set of microservice candidates based on the number of microservices provided by the expert. Furthermore, [SQMC21] also optionally accepts a set of fine-tuning parameters to function.

2.2.2 Process of Microservice Identification Approaches

The process of a microservice identification approach can be decomposed into 4 categories: (1) the type of algorithm used, (2) its degree of automation, (3) the type of input analysis used, and (4) the direction in which information is extracted from the input to propose a microservice-oriented architecture.

[3] https://swagger.io/specification/
[4] https://en.wikipedia.org/wiki/Version_control

2.2.2.1 Type of Algorithms

Algorithm	MIA	Total
Clustering	[DMF+20], [CLL18], [BGDR17], [GKGZ16], [BCS21], [ECA+16], [KXL+20, KXK+21], [NSRS19], [SRS20], [SSB+20b], [ZSS+22], [KZH+20], [SQMC21], [JLZ+18], [MCF+20], [LO20], [MCL17], [DMF+20, DEF+21], [AIE19], [DABPF18, DABFP20], [KYHM18], [ADM20, AM21]	22
Evolutionary Algorithm	[SOMS19], [CGC+20, ACC+21], [Ami18], [JLC+21], [ZLD+20]	5
Custom Heuristic	[PFM19], [LTV16], [BSG20], [ERM20]	4
Formal Concept Analysis	[ASS+21]	1
Guideline / Best Practices	[GBMM20]	1

Table 2.2: *Classification of the extracted MIAs based on their algorithm.*

Each MIA uses an algorithm to process the input and produce a description of the target microservice-oriented architecture. Table 2.2 presents the extracted MIAs based on the type of algorithm they use. Concretely, we categorize the MIA algorithms into 4 distinct categories: *clustering*, *evolutionary*, *formal concept analysis*, *custom heuristics*, and *guidelines*.

Clustering: Clustering algorithms consists in partitioning program artifacts into clusters based on a set of criteria. These clusters are partitioned in a way to optimize the cohesion between the elements within a cluster and reduce the coupling between the elements that are not in the same cluster. One example of a clustering algorithm is the hierarchical clustering, which is used in several approaches (e.g., [SSB+20b], [ZSS+22], [AM21]). A total of 21 of the 32 MIAs use a clustering algorithm to identify an MSA.

Evolutionary: Evolutionary algorithms are a family of algorithms whose principle is inspired by the theory of evolution to solve various problems. They are therefore bio-inspired methods of calculation. The idea is to evolve a set of solutions to a given problem, in order to find the best results. Several MIAs use a genetic algorithm, a sub-set of evolutionary algorithms (e.g., [ZLD+20]). This type of algorithm uses the notion of natural selection and applies it on a population of solutions to promote the best solutions over an iterative process. A total of 5 of the 32 MIAs use a clustering algorithm to identify an MSA.

Formal Concept Analysis: Formal concept analysis (FCA) is a general method of unsupervised classification and clustering. From a description of data called formal context (i.e., a set of relations between objects and their attributes), it forms concepts (i.e., a gathering of objects that shared the same common set of attributes). The concepts are then made into a hierarchy to produce a structure called a concept lattice [GW99]. FCA is used in [ASS+21] to infer the relationship between the business objects (i.e., the domain entities of the application) and the entry point of the application (i.e., the services).

Custom Heuristics: Alternatively, some MIAs propose custom heuristics that do not fall into the above categories. For instance, [LTV16] propose a 6-step methodology to extract a set of microservice candidates. This methodology lays out how to map an existing 3-part monolithic application into a triple which is decomposed using custom decomposition strategies.

Guidelines: As the name implies, approaches which only offer a set of recommendations, best practices, or recommendations to identify microservices. For instance, in [GBMM20], the authors propose a decision model to recover the microservice architecture.

2.2.2.2 Automation

Automation	MIA	Total
Automated	[ECA+16], [BGDR17], [MCL17], [JLZ+18], [DABPF18, DABFP20], [Ami18], [AIE19], [NSRS19], [SOMS19], [LO20], [SRS20], [ADM20], [ZLD+20], [BCS21], [AM21], [JLC+21], [ASS+21]	17
Semi-automated	[GKGZ16], [CLL18], [KYHM18], [PFM19], [BSG20], [DMF+20, DEF+21], [KZH+20], [MCF+20], [SSB+20b], [KXL+20, KXK+21], [SQMC21], [CGC+20, ACC+21], [ZSS+22]	12
Manual	[LTV16], [GBMM20], [TS20], [ERM20]	4

Table 2.3: *Classification of the extracted MIAs based on their automation.*

Each MIA process has a certain degree of automation. Concretely, we categorize the MIA processes into 3 distinct categories: *automated*, *semi-automated*, and *manual*. Table 2.3 presents the extracted MIAs based on those categories. An approach is considered to be automated when the process can be applied with little to no human intervention. For instance, [NSRS19] proposes an automated approach to extract the call graph from the monolith source code using *java-callgraph* tool, and using the python library *scipy* to perform a hierarchical clustering. Approaches that require a greater degree of human intervention are considered semi-automated. Approaches such as [SQMC21] require experts to tag the existing source code before an automated analysis can be performed. Finally, manual MIAs are either guidelines or custom heuristics that need to be applied by experts.

2.2.2.3 Analysis Type

Each MIA uses an algorithm to process the input and produce a microservice-oriented architecture. Table 2.4 presents the extracted MIAs based on the type analysis they use. Concretely, we categorize the MIA algorithms into 5 distinct categories: *static*, *dynamic*, *hybrid*, *lexical*, and *domain*.

Analysis Type	MIA	Total
Static	[ECA+16], [LTV16], [MCL17], [KYHM18], [NSRS19], [SOMS19], [PFM19], [BSG20], [LO20], [SRS20], [SSB+20b], [SQMC21], [ZSS+22], [DABPF18, DABFP20], [KZH+20], [CGC+20, ACC+21], [ASS+21]	17
Dynamic	[JLZ+18], [AIE19], [ZLD+20], [ERM20], [TS20], [KXL+20, KXK+21], [JLC+21]	7
Hybrid	[MCF+20]	1
Lexical	[BGDR17], [ADM20, AM21], [DABPF18, DABFP20]	3
Domain	[GKGZ16], [KMK16], [Ami18], [CLL18], [GBMM20], [CGC+20, ACC+21], [DMF+20, DEF+21]	7

Table 2.4: *Classification of the extracted MIAs based on the type of analysis they perform.*

Static: Static program analysis covers a variety of automated methods used to obtain information about the behavior of a program during its execution without actually running it. Its main advantage is that it does not require executing the code and can be applied as long as the source code is available (e.g., [NSRS19], [SSB+20b]).

Dynamic: Dynamic program analysis covers the methods used to obtain information about the behavior of a program by observing its execution. [JLZ+18] & [JLC+21] both instrument the source code and run test cases to produce program traces (i.e., association between an execution behavior and the code that implements this behavior). From these traces, dynamic analysis can take place. Other approaches, establish use cases to produce these traces [KXL+20, KXK+21].

Hybrid: Hybrid analysis combines both static and dynamic analysis. We chose to differentiate between dynamic analysis and hybrid analysis as certain MIAs use dynamic analysis in coordination with a static analysis. For instance, [KZH+20] propose to supplement static analysis with dynamic analysis to enrich the collected information.

Lexical: Lexical analysis approaches take textual similarity into account when identifying microservices. [ADM20, AM21] & [CGC+20, ACC+21] analyze the API documentation of a system to decompose it using semantic analysis.

Domain: Domain analysis relates to approaches that use domain artifacts as a main source of information. In particular, approaches such as [CLL18] use dataflow diagrams as the main input for their MIA.

2.2.2.4 Direction

MIAs can follow three different directions: *bottom-up*, *top-down*, and *hybrid*. Table 2.5 presents the extracted MIAs based on their direction.

Direction	MIA	Total
Bottom-up	[ECA+16], [LTV16], [MCL17], [JLZ+18], [KYHM18], [DABPF18, DABFP20], [NSRS19], [SOMS19], [PFM19], [BSG20], [SRS20], [TS20], [ZLD+20], [ERM20], [MCF+20], [LO20], [SSB+20b], [ADM20, AM21], [JLC+21], [CGC+20, ACC+21], [BCS21], [ZSS+22], [AIE19], [ASS+21]	24
Top-down	[GKGZ16], [BGDR17], [Ami18], [CLL18], [DMF+20, DEF+21]	5
Hybrid	[GBMM20], [KXL+20, KXK+21], [KZH+20], [SQMC21]	4

Table 2.5: *Classification of the extracted MIAs based on the direction of their process.*

Bottom-up: A bottom-up process starts with low-level artifacts (e.g., source code) to maximize code reuse and minimize changes. From these low-level artifacts, it extracts an abstraction (e.g., the architecture), which is used to identify candidate services. For instance, [LO20] proposes an approach which extracts the static coupling between the classes of a monolith, as well as semantic, and evolutionary coupling to create a weighted graph in which each node represents a class of the monolith. Each coupling information is extracted from the classes and their relationship with one another based on the static analysis, semantic similarity, and their changes over time.

Top-down: Alternatively, a top-down process starts with high-level artifacts (e.g., domain analysis) to extract a design of the target architecture. Unlike bottom-up approaches where low-level artifacts are used to extract higher-level artifacts, top-down approaches do not consider low-level artifacts to identify microservices, and they only propose a high-level decomposition of the existing system. One example of a top-down approach is proposed in [DMF+20, DEF+21]. In [DMF+20, DEF+21], the authors propose to analyze a business process model to then decompose it into a set of activities. Similarly, [Ami18] presents an approach to partition a business process model's activities based on their structural relations as well as the read and write operations they perform the monolith's data objects.

Hybrid: Finally, a hybrid process combines high-level artifacts with low-level ones to perform a hybridization of both bottom-up and top-down approaches. This is often done by mapping between low-level artifacts with high-level ones and decomposing the high-level artifacts. In the case of [GBMM20], high-level artifacts (i.e., business requirements) are used to guide the decomposition of stored procedures within a monolithic application.

2.2.3 Output of Microservice Identification Approaches

The approaches extracted for this taxonomy can also be categorized by the type of output they produce. Each MIA outputs vary in their distinction of what the finished

MSA should look like. We propose the following categories to describe the type of outputs produced by the extracted MIAs: (1) design-time microservices, (2) unpackaged microservices, (3) packaged microservices, and (4) black-box microservices. Table 2.6 presents the MIAs based on the outputs they produce.

Design-time microservices: A design-time microservice is considered design-time when non-executable model representations are decomposed, in a top-down approach, and further decomposition of the source code is necessary. Microservice-oriented architecture identified by top-down MIAs often produce *Design-time* microservices due to the direction of their process. Approaches such as [Ami18] and [DMF+20, DEF+21] decompose BPMs but do not go to lower-level artifacts such as the source code. Similarly, [AIE19] propose a decomposition of the API provided by the monolith, from there they duplicate the monolith for each partition and route the appropriate API requests to their microservices. Other MIAs, such as [BGDR17], [SRS20], partition the domain of a system, yet do not address the business logic.

Un-materialized microservices: Un-materialized microservices are the most common output produced by the extracted MIAs. MIAs that produce un-materialized microservices represent an MSA with a partition of the source code. The term *un-materialized* is used to denote that while each identified microservice is mapped to a set of program artifacts, they are not sufficient to produce a fully-functioning MSA and further refactoring is required. For instance, [SOMS19],[SSB+20b], and [JLC+21] produce a partition of classes of an object-oriented monolith that require additional refactoring.

Materialized microservices: To produce a set of materialized microservices as an output, it requires an MIA to refactor the internal code of the monolith to function as a set of microservices communicating via a set of services. Only one approach ([LTV16]) produces materialized microservice, however their approach is entirely manual. Furthermore, the authors only propose a methodology to materialize the identified architecture. For instance, in the final step of the approach, the authors prescribe to create a gateway to act as an intermediate layer between the client and the monolith and to progressively replace the services proposed by the monolith with those provided by the identified microservice candidates. They also highlight that certain microservice candidates require an *additional effort* but do not go into details on how to produce them.

2.2.4 Summary of the Taxonomy

In this section, we summarize the obtained findings of this taxonomy. The findings are organized based on the required inputs, the identification process, and the produced output.

Output Type	MIA	Total
Design-time microservices	[GKGZ16], [Ami18], [CLL18], [AIE19], [GBMM20], [DMF+20, DEF+21], [SQMC21], [BGDR17], [SRS20]	9
Unmaterialized microservices	[ECA+16], [MCL17], [JLZ+18], [KYHM18], [DABPF18, DABFP20], [NSRS19], [SOMS19], [PFM19], [BSG20], [TS20], [ZLD+20], [ERM20], [MCF+20], [LO20], [SSB+20b], [KXL+20, KXK+21], [KZH+20], [ADM20, AM21], [JLC+21], [CGC+20, ACC+21], [BCS21], [ZSS+22], [ASS+21]	23
Materialized microservices	[LTV16]	1

Table 2.6: *Classification of the extracted MIAs based on the output they produce.*

2.2.4.1 The required inputs of MIAs

More than two-thirds (72.7%) approaches rely on the source code as a required input. In fact, 21.2% of approaches rely solely on the source code to perform the identification of an MSA. The source code is often executed, along with test cases or use cases, to generate log traces. Log traces are the second most used inputs with 21.2%, and used to perform a dynamic analysis on the source code.

Data autonomy is a key microservice characteristic. Therefore, it makes sense that the database is the second most used executable model with 18.2% of MIAs use the database schema or the domain. We did not include domain analysis performed by experts in this category, however we denote that 2 out of the 4 approaches that use expert input, rely on domain analysis.

When it comes to domain artifacts, there are two main artifacts used by MIAs: documentation, and expertise. Expert knowledge is used by MIAs 12.1% of the time, while documentation is used by 15.2% of MIAs. Version control history is used only twice along with the source code to supplement the static analysis by providing coupling weights between program artifacts based on whether they are modified at the same time.

Finally, abstract model representation such as DFD, and BPMN are used by 9.1% of approaches in top-down approaches to design the microservice candidates.

2.2.4.2 The process of MIAs

The identification process can be split into 4 categories: the algorithms, the automation, the analysis type, and its direction.

In terms of algorithms, 66.7% of approaches use a clustering algorithm (e.g., hierarchical clustering, Girvan–Newman). 15.2% of MIAs used an evolutionary algorithm,

usually a genetic algorithm. Custom heuristics represent 12.1 % of approaches, and only one approach presented a set of guidelines to decompose a monolithic application.

Almost half of the approaches (51.5%) presented an automated process. Another (36.4%) managed to automate part of their process. In several cases, the semi-automated MIAs use expert inputs. Finally, 12.1% of approaches have not been automated or propose a set of guidelines or algorithms that are not easily automated.

Each MIA's process involves analyzing the inputs to extract information to propose microservice candidates. Most commonly, MIAs use either static or dynamic analysis to perform their decomposition with 51.5% of approaches relying on a static analysis, and 21.2% relying on a dynamic analysis. Additionally, 1 approach uses both static and dynamic analysis. 21.2% of MIAs primarily analyze domain artifacts. Finally, several approaches rely on the lexical analysis of the code or documentation to propose a decomposition (9.1%), occasionally on top of static analysis (e.g., [DMF+20, DEF+21]).

In terms of direction, the vast majority of approaches propose a bottom-up process (72.7%). Another 15.2% of MIAs use a top-down approach to propose design-time microservices. Finally, only 12.1% of approaches use a hybrid of both directions.

2.2.4.3 The output produced by MIAs

In terms of output produced, MIAs can be categorized into three types of categories: design-time microservices, un-materialized microservices, and materialized microservices. The majority of approaches produce un-materialized microservices (69.7%), as they are able to cluster program artifacts extracted from the source code but do not provide techniques to refactor the existing code to conform to the identified architecture. Alternatively, other approaches propose design-time microservices which are not tied to program artifacts. This type of output is generated by 27.3% of the extracted MIAs. Finally, we also included the materialized microservice category that reflect the need to materialize the identified microservice candidates into executable projects. Only one approach ([LTV16]) seems to provide a guideline towards materializing microservice candidates, however, this approach only provides a methodology to migrate a monolith with very little details on how to materialize each microservice. This highlights the lack of work done on transformation phase of the migration process.

2.3 Transformation towards an MSA

As we've seen in Section 2.1, the process of software migration can be decomposed into two phases: the reverse-engineering and the transformation. In the previous section, we covered the taxonomy of the reverse-engineering effort to recover a microservice-oriented architecture from a monolithic oriented-one. Furthermore, in the discussion we highlighted that most approaches produced un-materialized microservice candidates that were not sufficient to produce a fully-functioning MSA and that further

refactoring is required. Therefore, in this section we cover the existing works that focus on transforming monolithic application towards a microservice-oriented architecture to produce materialized microservices.

One of the first works that attempts to transform an existing monolith towards a microservice-based architectural style is presented by Chris Richardson in his book [Ric18b]. In this work, the author presents the *Strangler Fig* pattern which proposes to iteratively identify and transform individual microservices from a monolith. To create each new microservice, they propose a guideline to integrate the new microservice into the existing system. The *Strangler Fig* pattern is used in a case study to successfully migrate a set of microservices from an existing system [LML20]. While the work presented in [Ric18b] is thorough, the chapter that covers the transformation phase focuses mainly on generic architectural transformation strategies to help with the migration effort. These strategies are helpful for designing a migration process, however architects must interpret these strategies on their own on a case-by-case basis.

In [KH18], the authors propose a 5-step methodology to address the migration of monolithic application towards a microservice-based architectural style. In their paper, they assume a decomposition has already been proposed. From this decomposition, they rely on a systematic process to identify dependencies between the partitions of the monolith to define a set of external and internal service facades (i.e., interfaces). They implement the service facades by adapting the existing system, so that the facades serves an entry point to communicate with the system's artifacts. Then, the clients are modified to interact with the facade rather than directly with the system's artifact. The methodology presented in this work, presents a generic set of steps to assist in the migration of a COBOL-based application. However, the authors do not propose a set of transformation rules to facilitate the migration. Furthermore, the authors propose an approach that is validated on a mainly procedure-based application. As a result, this approach can guide architects to elaborate their own migration process but cannot be used directly to migrate an OO application.

Alternatively, a set of transformation rules is proposed in [AS20] to transform the execution flow of a monolith. In their work, the authors highlight the difficulty of redesigning monoliths which employs transactions (see Definition 2.3.1). Indeed, when a transaction is distributed across different microservices, it is difficult to redesign the transaction to respect the ACID properties of a transaction. To preserve these properties and the behavior of the application, the authors of [AS20] propose an implementation of the SAGA design pattern introduced in [GMS87]. Particularly, they propose 3 basic transformation rules, and one additional composite transformation rule, to support the redesign of a functionality's execution flow [AS20]. In this way, the authors are able to avoid creating side effects related to loss of the ACID properties of certain transactions. However, the approach focuses entirely on the preservation of the monolith's behavior in relation to transaction management, and no automation is proposed. Therefore, applying this approach on an OO application is not enough to fully transform the source code towards an MSA and further refactoring is required.

Definition 2.3.1: Transactions and Transaction Management

A transaction is an action, such as a payment, that is implemented via a sequence of access operations to modify the state of a database. Transaction management mechanisms are implemented to ensure that this sequence of operation respects the ACID properties of a transaction (i.e., that the sequence is atomic, consistent, isolated and durable).

As we have seen in [KH18], transformation approaches often rely on defining facades between microservice candidates based on the dependencies between the system artifacts of different microservices. However, the implementation of these facades can have an impact on the comprehension of the code. To reduce this, [FSC+21] propose to use aspect-oriented programming to make the refactoring transparent to the developer. Particularly, the authors use pointcuts to decouple the identified microservice to replace method calls with service calls [FSC+21]. This approach has the advantage of applying no code changes, and making it easily to enable/disable the added pointcuts based on the developer's wish to migrate. However, their approach is limited to refactoring method calls between classes of different microservices, and they do not take into consideration other object-oriented dependencies.

Finally, [FFC21] propose a methodology to automatically (via their tool *MicroReact*) transform a JAVA-based monolithic application into an MSA. Their methodology creates a REST API for each method that is invoked by a class belonging to a different microservice. The methodology also includes a database refactoring strategy by applying design patterns (e.g., *Move Foreign-Key Relationship to Code*, *Database Wrapping Service*) presented by [New19] to the object-relational mapping (ORM) entities. This transformation approach is able to promote true data autonomy by assigning the database access of a table to an individual microservice. However, as the authors highlight, they did not consider the different OO-type dependencies between microservices such as inheritance.

We also denote that in the case of [FSC+21] and [FFC21], these works were published around the same period as the work presented in this thesis (see Chapter 4 [ZSS+21]).

2.4 Discussion and Motivation

From the state-of-the-art presented, we highlight three main points that motivate this thesis:

1. In Section 2.2, we have presented a taxonomy on the existing microservice identification approaches. As a result, we observed that bottom-up approaches that use the source code as the primary input (e.g., [JLZ+18],[SSB+20b],[AM21]) often view MIAs as a clustering problem. In other words, they view the monolithic application as a set of program artifacts to be partitioned by promoting highly-cohesive and loosely-coupled structural clusters (i.e., microservice candi-

dates). However, these approaches omit to consider that enterprise applications are built upon standardized frameworks which promote a layered architecture [Ric18a]. As a result, they can fall into the trap of creating microservices that are structurally cohesive but not functionally cohesive (e.g., Wrong Cuts Antipattern) [TLP20]. Indeed, this antipattern proscribes for microservices to be split based on technical layers (presentation, business, data layers). Instead, popular decomposition patterns prescribe that microservices should be split based on business capabilities with a vertical integration of all layers within one small autonomous application [Ric18a]. Therefore, by extracting the layered architecture we can create more autonomous microservices.

2. Regarding the transformation phase of the migration process, we observed that there were significantly fewer approaches that attempted to address this research problem. Indeed, during the taxonomy of the identification approaches indicated that the vast majority of approaches only produced a description of the ideal MSA for an architect to implement. Of the approaches we presented in Section 2.3, only 3 approaches presented a set of transformation rules to assist in the migration of a monolith. Furthermore, only 2 of these approaches presented an automated approach in the last year. In these two cases, the approaches transform only few types of OO dependencies such as method invocation between classes belonging to different microservice candidates, and the ORM entities. However, when transforming an object-oriented monolith there are several OO mechanisms that must be handled to create syntactic and semantically-valid microservices. For example, object-oriented mechanisms such as inheritance, and exception handling still need to be handled.

3. Furthermore, these approaches remain limited by the language of the application's source code, obstructing its genericity and reusability across other languages and technologies. Indeed, in the case of [FSC+21], and [FFC21], both approaches limit their approach to JAVA-based applications. In other words, these approaches must be re-implemented when an application is implemented in another language. Instead, this limitation could be overcomed by adopting **Model-Driven Engineering** (**MDE**) techniques.

2.5 Conclusion

In this chapter, we presented the state-of-the-art on the modernization of existing systems and their architectures. Particularly, we focused on the migration of monolithic applications towards a microservice-based architectural style. With the goal of presenting the migration process, we divided it into a two-step process: (1) identification of the MSA, and (2) the transformation of the existing system towards the identified architecture. Furthermore, for each step we covered the existing work. In the case of the identification step, we proposed a taxonomy to categorize and summarize 33 different identification approaches in the following categories: input, process, and output. Afterward, we presented the existing state-of-the-art concerning the materialization of the MSA, in which we highlighted the limited work on the transformation phase.

As we have seen in the previous section, there are 3 issues we highlight in this thesis. The first issue deals with the identification phase, in which we have observed that most identification approaches fail to consider the monolith has an internal architecture that can be extracted and used to provided quality microservice candidates. As of a result, in Chapter 3 we present our approach that takes into consideration the layered architecture to decompose the monolith into a set of microservice candidates. Furthermore, we motivate the experimentation to evaluate the impact that the extraction of the monolith's architecture has on the quality of the identified MSA.

The second issue we wish to address, is that of the transformation phase. As we highlight in the previous section, few works have been proposed, until recently, to address the issue of materializing the identified MSA. Therefore, we set out to address this issue by proposing an ad hoc transformation approach to materialize the identified MSA while addressing the different types of OO dependencies between microservice candidates that must be resolved to create valid microservices. In Chapter 4, we present this approach.

Finally, the third issue we address is that of the limitations of an ad hoc transformation approach. While the presented approaches attempt to automate the refactoring of any OO language, their implementation are limited by the language of the application's source code. Instead, we wish to propose a generic approach. Therefore, in Chapter 5 we seek to overcome these limitations by adopting model-driven engineering techniques. Furthermore, we also seek to present an end-to-end approach that tackles both research problems by incorporating the approach presented in Chapter 3.

III

Leveraging the monolith's internally-layered architecture for the identification of microservices

Contents

3.1 Introduction

The main goal of this chapter is to propose an approach for recovering (i.e., identifying) a microservice-oriented architecture from a monolithic application. This architecture can then be used to guide the refactoring effort of the monolithic application

into a microservice-oriented one. In the state-of-the-art, we have presented several approaches which propose to identify a microservice-oriented architecture. However, these approaches omit to consider that enterprise applications are built upon standardized frameworks which promote a layered architecture [Ric18a]. Approaches such as [JLZ+18, SSB+20b, AM21] view microservice identification as clustering problem, in which the monolithic application is viewed as a set of classes to be partitioned by promoting highly-cohesive and loosely-coupled structural clusters (i.e., microservice candidates).

However, today's industrial applications are often built upon standardized framework which rely on a technically-layered architecture to promote the separation of concerns. Alternatively, microservice identification approaches that focus on identifying microservices by promoting highly-cohesive and loosely-coupled structural clusters often don't consider that enterprise applications are built upon these standardized frameworks. As a result, approaches that fail to consider these layered architectures risk falling into the trap of creating microservices that are structurally cohesive but split based on their technical layers instead of their business capabilities.

Indeed, according to experts, one of the most harmful antipatterns to consider during the identification is the Wrong Cuts [TLP20]. This antipattern proscribes for microservices to be split based on technical layers (presentation, business, data layers) [TLP20]. Instead, popular decomposition patterns prescribe that microservices should be split based on business capabilities with a vertical integration of all layers within one small autonomous application [Ric18a]. This verticality promotes functional and data autonomy in microservices which are characteristics sought out by architects. Consequently, the implementation of each microservice should be vertical or 'cross-layer' (i.e., composed of the three layers). In turn, by extracting the layered architecture we can promote vertical dependencies between the different layers to identify microservices based on their business capabilities.

This chapter proposes a semi-automatic identification approach that leverages the internally-layered architecture of legacy object-oriented applications. By doing so, it aims to avoid these common pitfalls, and promote the design of quality microservices. Our approach is divided into a two-step process. During the first step, we analyze the source code information available to extract the layered architecture artifacts present in the application. In the second step, we use the extracted artifacts to improve the partition of the monolith's classes into microservice candidates.

In the next section, we introduce the context of this chapter by presenting the typical 3-tier architecture with the web application *JPetStore*, as a motivating example to highlight the difficulties with identifying its microservices. In Section 3.3, we explain the global workflow of the proposed approach. In Section 3.4, we present the initial reverse-engineering effort performed for the identification. From there, in Section 3.5, we present how the output of the reverse-engineering effort is used to identify pertinent microservices. In Section 3.6, we use the proposed approach to answer two research questions pertaining to the extraction of the internal architecture of the monolithic application and its impact on the identification process. In Section 3.7, we conclude the chapter and highlight some possible future work.

3.2 Motivating Example: JPetStore

Today's industrial applications are often built upon standardized frameworks. By definition, frameworks provide an abstraction of a software, by providing generic non-business functionalities, that can be overloaded based on user-specific requirements [wik22]. Furthermore, these frameworks often rely on the Inversion of Control (IoC) pattern and the Dependency Injection pattern (DI) to create loosely-coupled and highly-cohesive applications. This allows companies to bypass the implementation of infrastructural needs to focus on implementing the business-logic of their software.

Frameworks such as Spring[1], Node.js-based frameworks[2], or ASP.NET Core[3] all rely on a technically-layered architecture to promote a separation of concerns. These layered architectures often take the form of a 3-layer architecture with a presentation, business, and a data-access layer. Alternatively, the Model-View-Controller (MVC) pattern also promotes a similar separation of concern between the three different elements of the pattern.

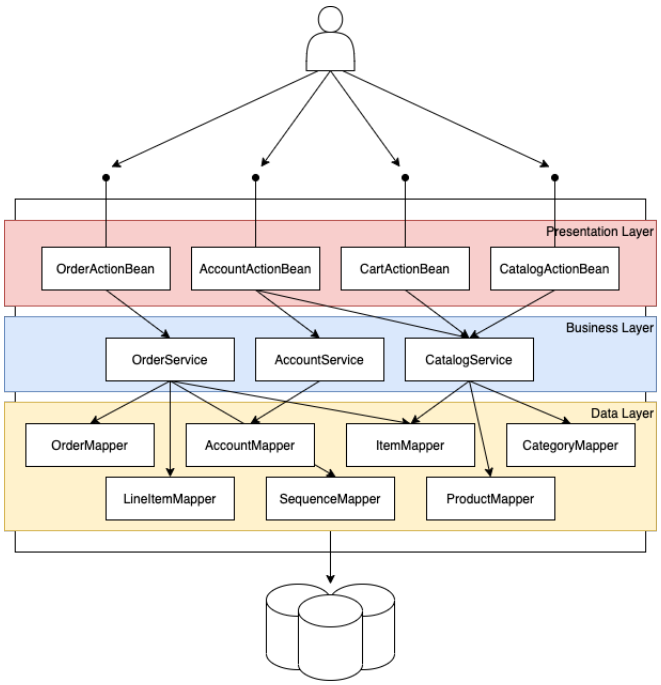

Figure 3.1: *Layered Architecture of JPetStore.*

To better illustrate the problems and solutions related to microservice identification for object-oriented monolithic applications, we introduce JPetStore[4]. JPetStore is a typical web application implementing a 3-layer architecture that acts as an online pet commerce. In essence, JPetStore contains 4 main features (functionalities): account, catalog, shopping cart, and order management. Structurally, it contains 24 classes, of which 14 are reusable structural classes and 9 data entities.

Each class can be mapped to one of the three layers described in the typical 3-layer

[1] https://spring.io/
[2] https://nodejs.org/
[3] https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-5.0
[4] https://github.com/mybatis/jpetstore-6

architecture: the presentation layer, the business layer, and the data layer. The presentation layer is the topmost level of the application. This layer is responsible for displaying the information related to JPetStore's features. The business layer acts as an intermediary between the presentation layer and the data layer. This layer is responsible for performing the business logic of the application. Finally, the data layer is responsible for applying the domain logic of the application as well as the data persistence mechanisms. In the case of JPetStore, we map the 14 structural classes to the three layers presented in Figure 3.1.

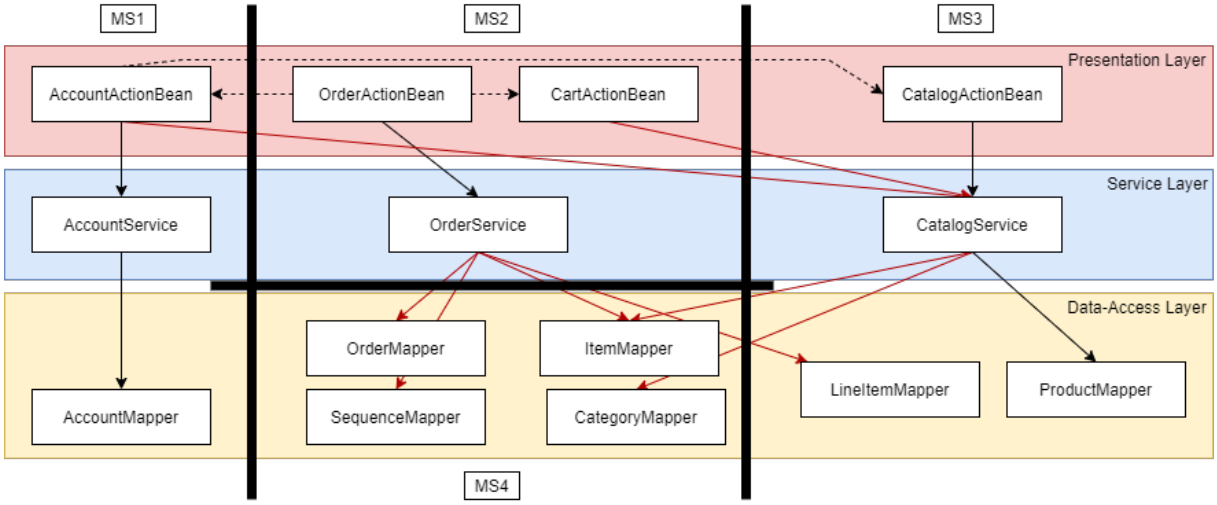

Figure 3.2: *Proposed decomposition of JPetStore by [AM21].*

Ideally, we want to recover a microservice for each feature and their underlying implementation, in the goal of making them functionally independent. This implies decomposing the application vertically to provide each microservice with a business functionality, its underlying business logic, and data access. In Figure 3.2, we illustrate the risk of identifying microservices based on the technical layers (i.e., a wrong cut) by showing a decomposition proposed in [AM21]. In this example, the authors propose 4 microservice candidates. Two microservice candidates (MS1 & MS3) offer a vertical decomposition while being business-oriented and containing some data autonomy. However, there is a horizontal slice that creates a microservice candidate (MS4) which only contains data-access classes, and a microservice candidate (MS2) with the business functionality. Horizontal slices are considered bad design as they increase the number of inter-process communications (i.e., network calls) between the client's request and its response, thus increasing the response time.

In the next section, we present our approach which leverages the underlying architecture found in industrial monoliths to propose a decomposition that prioritizes the identification of microservices based on their business capabilities first. We leverage the architecture in two ways: (1) we use the presentation layer to drive the clustering process from a usage perspective, (2) we use the data-access layer to ensure a level of data autonomy for each microservice. With regard to the presentation layer, we use its high-level business functions to guide the decomposition based on business capabilities to avoid creating the Wrong Cuts antipattern. With regard to the data-access layer, we are able to avoid dangerous antipatterns such as the Shared Persistence [TLP20] which pertains to microservices that end up using the same data, thus reducing team and service independence [TLP20].

3.3 Proposed Approach: Process and Principles

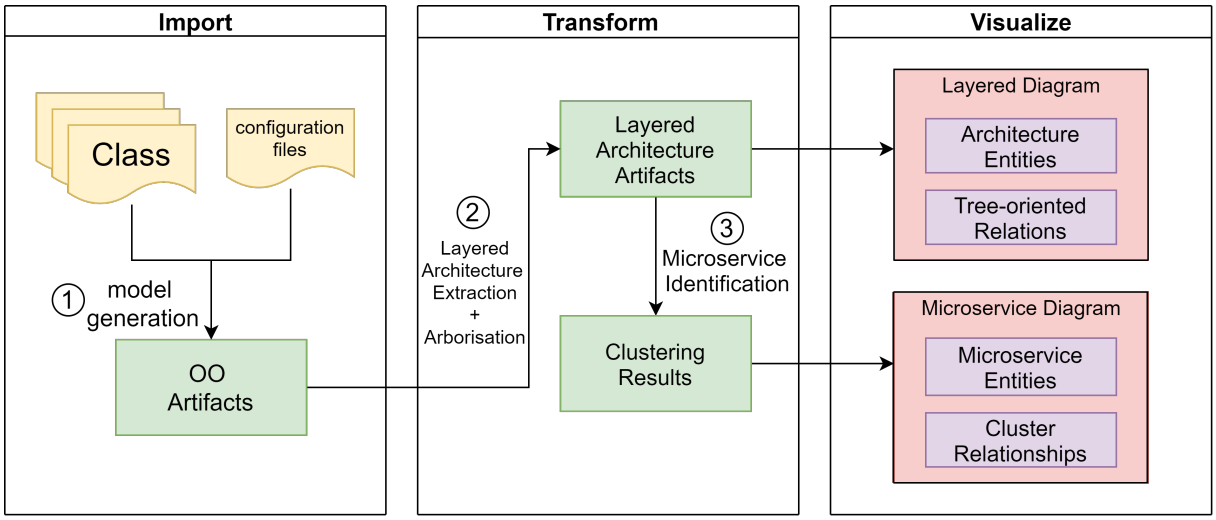

Figure 3.3: *The microservice architecture identification process.*

The goal of our approach is to leverage well-established design patterns and antipatterns to guide the process of identifying microservices. Similar to the metaphor of the carrot and stick, we motivate the identification process through observed design patterns while inhibiting it through bad decomposition patterns. Before explaining this identification approach, it is necessary to cover some design patterns proposed to design MSA [Ric18a, TLP20].

One such design pattern is the *Decompose by Business Capabilities* Pattern proposed by [Ric18a]. Richardson proposes a strategy to decompose monoliths based on the business capabilities (i.e., business-oriented functionalities) of the monolithic application. For example, in the context of an e-commerce, it can include order management, account management, and product management. It is important to note that a business capability is often focused on a particular business objects (i.e., data entity) which can also be used to guide the decomposition process [Ric18a].

Additionally, 2 of the 20 microservice antipatterns identified by [TLP20] can be ascribed to the migration's identification phase. First, The *Wrong Cuts* antipattern appears when microservices are split based on technical layers (i.e., presentation, business, data layer) instead of its business capabilities. According to [TLP20], this can increase data-splitting complexity and lead to increased communication between microservices. Second, the *Shared Persistence* antipattern occurs when different microservices access the same database. This antipattern creates a strong coupling between microservices manipulating the same data, thus reducing both development team and service independence [TLP20]. In general, this goes against one of microservices' fundamental characteristics: data autonomy. It's critical to grant data ownership to individual microservices to avoid this antipattern. One method to achieve this is to assign individual microservices to the business classes responsible for data access.

Our objective is to propose an approach which takes into consideration these design patterns and antipatterns. The process of our proposed identification approach is illustrated in Figure 3.3, and can be described as a three-step process. The initial step (i)

serves to recover the class artifacts from the object-oriented source code. Then, (ii) the artifacts are categorized into the different layers of the architecture (i.e., presentation, business, and data-access). This step also serves to extract the vertical dependencies between the different categorized artifacts. Finally, from the extracted internally-layered architecture's artifacts, (iii) the microservice candidates are identified through an automatic clustering approach.

Concretely, we present our approach in two parts. First, we present the extraction of the layered architecture and how it is represented. Second, we present the identification process that takes into consideration the extracted artifacts, as well as the design patterns and the antipatterns discussed previously. To assist in the identification process, a quality metric is proposed.

3.4 Reverse-engineering the layered architecture

The initial step in identifying microservice candidates is to extract layered architecture artifacts from the monolithic application's existing source code. This step aims at analyzing the object-oriented source code to extract OO artifacts. It involves identifying the monolithic application's structural elements (e.g., classes, methods, etc.) and the relationships between them (e.g., method calls, class inheritance, etc.) by analyzing the existing source code. Additionally, the project configurations may be analyzed to extract additional information on the structure of the source code.

From these OO artifacts, the layered architecture can be revealed through reverse-engineering techniques. In most modern frameworks, the OO artifacts are annotated based on the technical function they serve. In the case of the Spring framework, annotations such as `@RestController`, `@Service`, and `@Repository` serve to label the classes based on their responsibility. When appropriately labeled, the extraction of a layered architecture can be automatically induced. However, in the case of applications which do not use such frameworks, a manual labeling process is required. In either cases, to facilitate the labeling process and to represent the extracted layered architecture of monolithic applications, we propose the Layered Architecture Metamodel (LAMM). LAMM is illustrated in Figure 3.4, and can be divided into three viewpoints.

The first viewpoint (DI/IoC) is responsible for representing the decoupling mechanism found in most frameworks to promote business-oriented layer artifacts. The entity *LayerArtifact* is extended by the three entities found in the Layered Architecture viewpoint. It is implemented by a class entity represented by the OO artifacts, and can be described by one or more interfaces. It is important to represent this mechanism because in frameworks that use these patterns, the business-oriented code is mostly found in these layer artifacts. By ignoring this mechanism, it becomes harder to differentiate between the business and infrastructural code, and thus harder to adhere to the *Decompose by Business Capabilities* Pattern.

The second viewpoint (Layered Architecture) is responsible for representing the three layers present in the typical 3-Layer architecture: presentation, business-logic, and data-access layer. The presentation layer contains the classes responsible for inter-

acting with the user interface (UI), and it handles the requests generated by the user (i.e., the *Controller* entity). The business-logic layer contains the classes responsible for the business-oriented logic of the application (i.e, the *Service* entity). It is often described as the service layer, and it acts as the middleman that between the presentation layer and the data-access layer. Finally, the data-access layer is composed of the classes responsible for the data persistence mechanism and the data-access that encapsulates the persistence mechanism and exposes the data (i.e., the *DAO* entity). It is important to represent each layer and their interdependencies to retain the vertical dependency. By extracting the dependencies between the different layer artifacts we can minimize vertical dependencies between different candidates. Thus, we can promote a vertical decomposition of the monolith, and avoid creating *Wrong Cuts*.

The third viewpoint (Data Persistence) is responsible for representing the various data types found in web applications. Particularly, we denote two types the *Data Entity* and *DTO* (i.e, Data-Transfer Object) which specialize the *DataType* entity. *Data Entities* represent the implementation of a data table, while the *DTO* represents a data structure that can be easily serialized and transferred to the client. This layer serves to represent use of data within the monolith. Later, we use these entities to measure the similarity of data-usage between the different layer artifacts of the monolith. With this measure, we can promote data autonomy for each microservice by clustering layer artifacts with similar data-usage.

A semi-automatic process is applied to place the application's classes into one of these groups by mapping them to one of the LAMM entities. In the case of *JPetStore*, the application uses an annotation-based framework which encourages the annotation of controller, service, DAO, and data entity classes. 23 of the 24 classes in the project are categorized into four distinct categories, see figure 3.5. The remaining class is an abstract class (*AbstractActionBean*) inherited by all controller classes.

Once we have mapped the classes in the monolithic application into the appropriate layer entity, we analyze the dependencies of the classes to map them to their layer counterparts (e.g., if a Controller's implementing class references a class implementing a Service, then we can add a dependency between the Controller and the Service). This results in an oriented graph that models the architecture found in Figure 3.1. By mapping the classes of the monolith to the layered architecture, we can promote the vertical, or inter-layer dependencies, over horizontal dependencies to promote a vertical decomposition of the monolith. In doing so, we aim to avoid creating Wrong Cuts. Furthermore, by taking into account the data entities during the decomposition we aim to promote data autonomy for each microservice.

3.5 Identifying microservices using the extracted artifacts

After the reverse-engineering effort, we are left with a set of *LayerArtifacts* and *DataTypes* that we can use to identify the microservice candidates of the monolithic application. The identification problem is often reduced to a clustering problem in which a set of artifacts (e.g., classes of a system) are partitioned into a set of clusters which represent a set of microservice candidates. Multiple techniques exist which help partition a set of

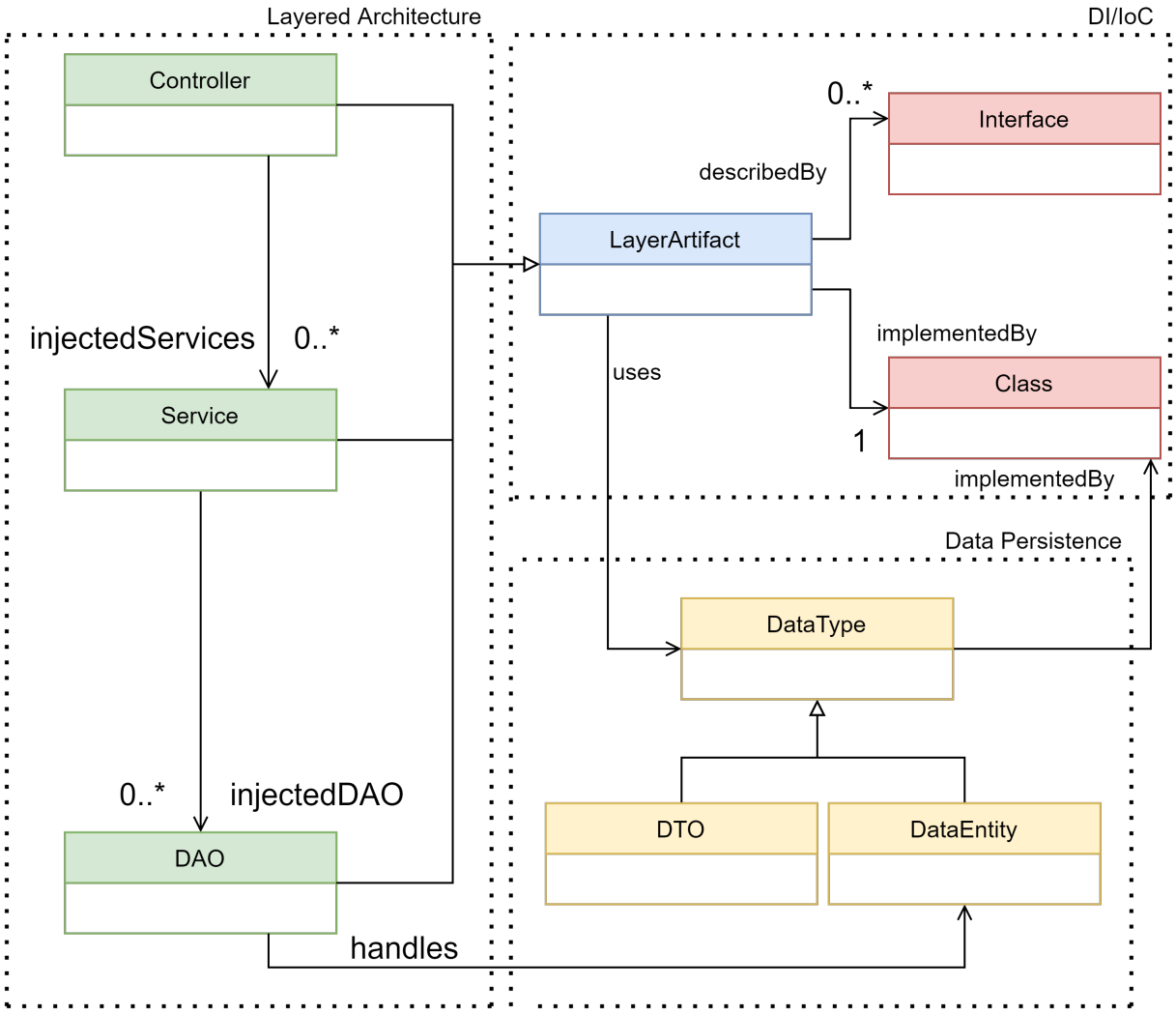

Figure 3.4: *Layered Architecture Meta-model (LAMM) that represents the layered architecture found in most frameworks.*

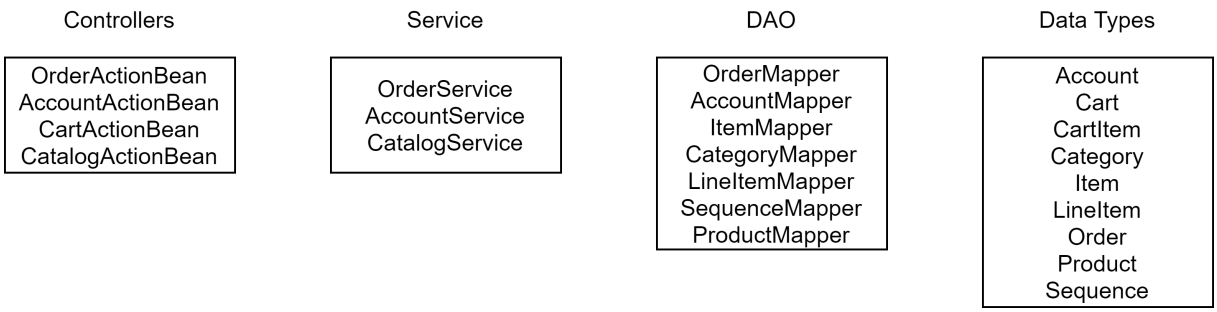

Figure 3.5: *Class categorization of JPetStore.*

artifacts based on an objective function (e.g., hierarchical clustering, k-means, genetic algorithms). In our case we use a hierarchical clustering algorithm to partition the artifacts extracted from the LAMM model instantiated during the reverse-engineering step. To guide the hierarchical clustering we propose an objective function. This objective function focuses on two aspects of quality microservices : structural and data autonomy. The first aspect serves to promote vertical microservice identifications. While the second aspect serves to promote data autonomous microservice candidates.

In the following sections, we present this objective function followed by the hierarchical clustering technique used in this approach.

3.5.1 Measuring the quality of microservice candidates

For the clustering algorithm to determine which clusters are good microservice candidates, it needs a quantifiable way to measure their quality. With this requirement in mind, we present two similarity measures to guide the clustering ($F_{Struct}(MS)$ and $F_{Data}(MS)$).

$$F_{Struct}(MS) = \frac{intradependencies(MS)}{nbPossibleDependencies(MS)} \tag{3.1}$$

The first similarity measure ($F_{Struct}(MS)$) calculates the structural cohesion between the layer artifacts of a microservice candidate. Particularly, the $intradependencies$ function calculates the number of intra-relationships between layer artifacts of the microservice candidate MS (e.g., method calls, attribute access, inheritance). While the denominator calculates the total possible incoming dependencies from the microservices. Together, they produce a similarity which promotes structural cohesion by rewarding clusters in which there are more internal relationships. Indeed, as the number of intra-relationships between layer artifacts of the same microservice candidate, so does its score.

$$F_{Data}(MS) = \frac{\sum_{(Cl_k,Cl_m)\in MS} f_{simData}(Cl_k, Cl_m)}{\frac{|MS|\times(|MS|-1)}{2}} \tag{3.2}$$

$$f_{simData}(Cl_k, Cl_m) = \frac{|Data_k \cap Data_m|}{|Data_k \cup Data_m|} \tag{3.3}$$

The second similarity measure is described in Eq. 3.2 ($F_{Data}(MS)$). It calculates the data cohesion between the layer artifacts of a microservice candidate. This is to encourage clusters with layer artifacts that manipulate the same data, and therefore promote clusters who are data autonomous. The numerator of the equation is the sum of $f_{simData}$ for each pair of layer artifact possible within the microservice. More specifically, Eq. 3.3 measures the data cohesion between two layer artifacts based on the data entities used by both layer artifacts. Eq. 3.3 returns a value close to 1 whenever the two layer artifacts manipulate the same data entities. The sum of $f_{simData}$ is divided by the total number of layer artifact pairs to produce an average value.

To better explain this measure, we present Figure 3.6, which illustrates the data dependency of two microservice candidates. In this example, MS2 contains the artifacts related to the account management (i.e., AccountActionBean, AccountService, and AccountMapper) and `D1` corresponds to the Account DataType. Table 3.1 calculates the score of $f_{simData}(Cl_k, Cl_m)$ for each pair of classes from Figure 3.6. From this table, we can calculate $F_{Data}(MS2) = 2/3$.

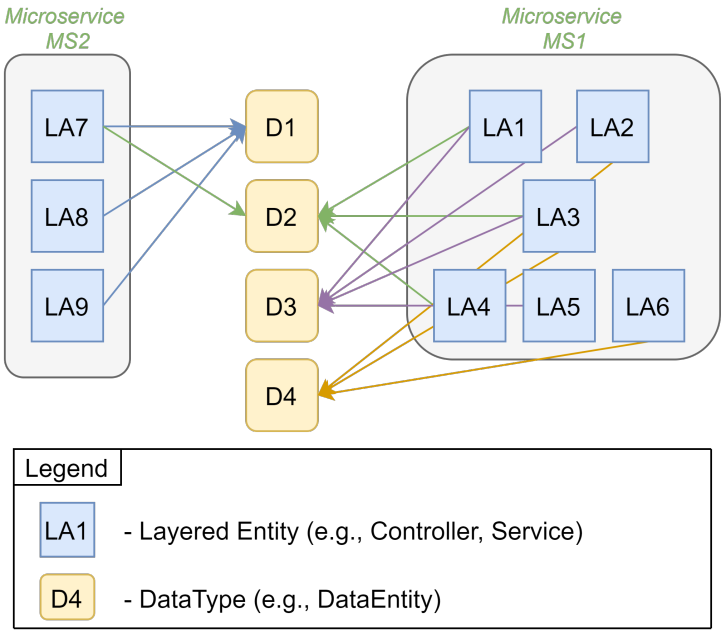

Figure 3.6: *Example of data dependency between LayeredArtifacts.*

	LA1	LA2	LA3	LA4	LA5	LA6	LA7	LA8	LA9
LA1	/	1/3	2/3	1/2	1/2	0	1/2	0	0
LA2	1/3	/	2/3	0	1/2	1/2	0	0	0
LA3	2/3	2/3	/	1/3	1/3	1/3	1/3	0	0
LA4	1/2	0	1/3	/	0	0	0	0	0
LA5	1/2	1/2	1/3	0	/	0	0	0	0
LA6	0	1/2	1/3	0	0	/	0	0	0
LA7	1/2	0	1/3	0	0	0	/	1/2	1/2
LA8	0	0	0	0	0	0	1/2	/	1
LA9	0	0	0	0	0	0	1/2	1	/

Table 3.1: *Measurement of $f_{simData}(Cl_k, Cl_m)$ between the Layered Artifacts of MS1 and MS2 from Figure 3.6.*

When the first similarity measure is applied to the Layered Architecture Metamodel, it ideally proposes a decomposition that favor vertical microservices focused on business capabilities. This is done by using a model that highlights these vertical dependencies between LayeredArtifacts instead of considering all structural dependencies as equal. Furthermore, by favoring the vertical dependencies between the business classes of the application, we limit the risk of creating Wrong Cuts.

When the second similarity measure is applied to a clustering algorithm, it ideally proposes a decomposition that favors grouping classes that manipulate the same data. Furthermore, by taking into consideration the data-access artifacts when partitioning the monolith, we limit the risk of creating Shared Persistence between microservices by creating data ownership.

$$F_{Quality}(MS) = \frac{\alpha \times F_{Struct}(MS) + \beta \times F_{Data}(MS)}{\alpha + \beta} \tag{3.4}$$

We combine both similarity measures to propose a multi-objective function (see Eq. 3.4) by proposing two coefficients (i.e., α & β) to balance their impact on the clustering process. In the next section, we use this function in the hierarchical clustering algorithm.

3.5.2 Microservice Identification using clustering algorithms

The hierarchical clustering algorithm is a clustering technique which builds a hierarchy of clusters of elements passed as input [SSB+20a]. To determine this hierarchy it uses an objective function to measure the quality of each cluster. In this chapter, we propose an agglomerative, or bottom-up approach, which creates this hierarchy of clusters starting from a set of clusters containing only one element. This hierarchy is represented as a dendrogram (i.e., a binary tree), in which each node represents a cluster (see Figure 3.7).

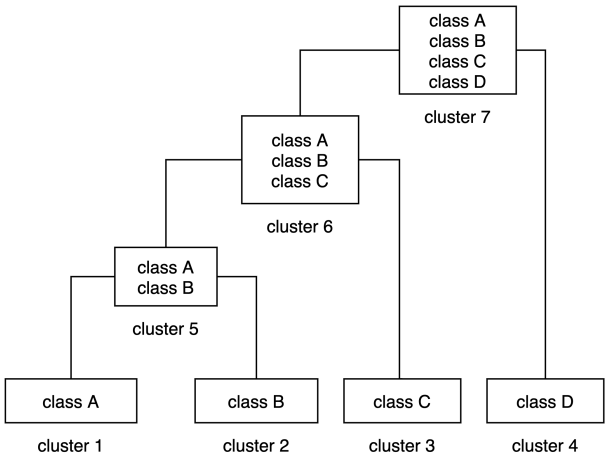

Figure 3.7: *An example of a dendrogram for clustering a set of classes.*

In Figure 3.7, we can see the hierarchy of the clusters as they are iteratively merged in a bottom-up approach, with *Cluster 7* being the last cluster. In the following section, we present the algorithm we used to generate our dendrogram. Then, in 3.5.2.2, we decompose the dendrogram to generate the final partition of set of microservice candidates.

3.5.2.1 Generating the Dendrogram

The first step towards partition the set of *LayeredArtifacts* is to generate the dendrogram. To do so, we present Algorithm 1.

To begin, we initialize a set of artifacts $S_{artifacts}$ extracted during the layered architecture recovery (line 1), and an empty set $S_{Clusters}$ which will contain the clusters of artifacts created during the algorithm (line 2). Then, for each artifact belonging to $S_{artifacts}$, a cluster is created in which the artifact is placed into, and the cluster is added to $S_{Clusters}$ (lines 3-5). Afterward, we iterate over the $S_{Clusters}$, finding the best two pairs of clusters, that when merged, return the best score using $F_{Quality}(MS)$ (lines

Algorithme 1: Hierarchical Clustering

Data: OO Source code *code*
Result: A dendrogram *dendro*
1 let $S_{artifacts}$ be the set of artifacts extracted from *code*;
2 let $S_{Clusters}$ be the set of clusters of artifacts;
3 **for *each*** $artifact \in S_{artifacts}$ **do**
4 | let *artifact* be a *cluster*;
5 | add *cluster* to $S_{Clusters}$;
6 **end**
7 **while** $size(S_{Clusters}) > 1$ **do**
8 | let $(cluster1, cluster2)$ be the closest pair of clusters based on $F_{Quality}(MS)$;
9 | let $New_{cluster} \leftarrow merge(cluster_1, cluster_2)$;
10 | remove $cluster_1$ and $cluster_2$ from $S_{Clusters}$;
11 | add $New_{cluster}$ to $S_{Clusters}$;
12 **end**
13 *dendro* $\leftarrow get(0, S_{Clusters})$;
14 return *dendro*;

7-8). Once the best pair is identified, they are merged to create a cluster containing the pair of clusters as children (line 9). Then the pair of clusters are removed from the $S_{Clusters}$, and the new cluster is added to $S_{Clusters}$ (lines 10-11). When two clusters are merged, a new cluster is created which contains the two clusters as child entities. The iteration ends when all clusters of $S_{Clusters}$ have been merged into one cluster. The end result is a binary tree (i.e., dendrogram) in which each node represents a cluster, and which the leafs are the original clusters containing one artifact.

The dendrogram in Figure 3.8 is the result of Algorithm 1 applied to *JPetStore*. In the next section, we use this dendrogram as input to propose a set of microservice candidates.

3.5.2.2 Decomposing the dendrogram

Once the dendrogram is generated, it can be used to identify the microservice candidates. In Algorithm 2, we propose an algorithm to decompose the dendrogram into the ideal set of clusters which will represent the final microservice candidates. Indeed, as the goal of Algorithm 1 is to group classes based on their functional similarity and their data-usage, the goal of Algorithm 2 is to decompose the dendrogram so that only the clusters with the strongest functional and data similarity remain.

The first step is to create an empty stack ($Stack_{Clusters}$) which will contain clusters to be decomposed, and we place the dendrogram inside (lines 1-2). Then, we initialize msa as an empty set of clusters, which will contain the final microservice candidates (line 3). We pop the first cluster from the stack, and fetch the children ($child_1, child_2$) of that cluster (lines 5-6). If the average score of the fitness function applied on both $child_1$ and $child_2$ is higher than the score of the parent cluster, then we push both children to the stack (lines 7-8). In Figure 3.8, the initial node with the score of 0.11 is lower than the

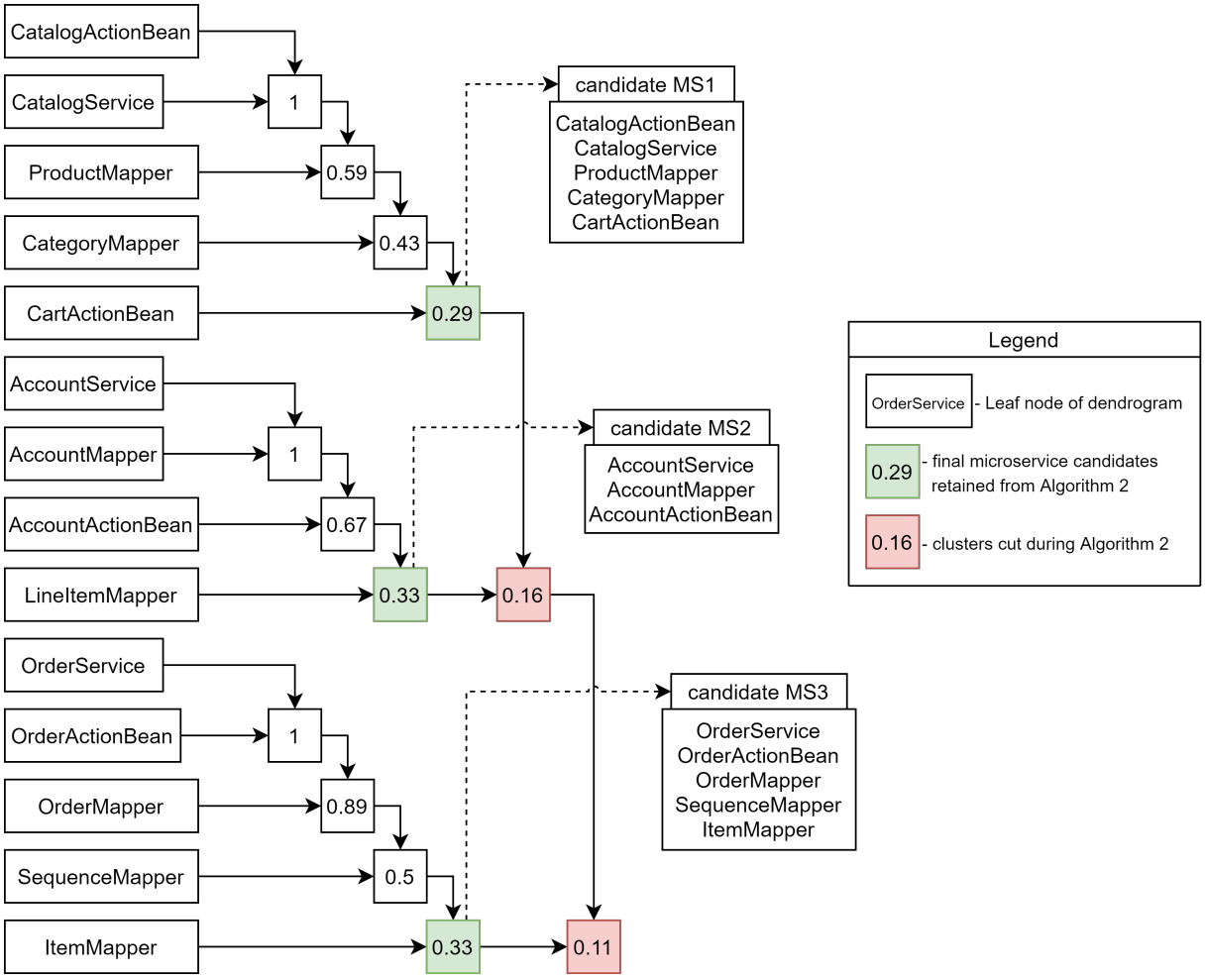

Figure 3.8: *JPetStore dendrogram generated by using Algorithm 1 and 2.*

Algorithme 2: Identifying the microservice candidates

Data: A dendrogram *dendro*
Result: A set of clusters $S_{Clusters}$

1 let $Stack_{Clusters}$ be an empty stack of clusters;
2 push *dendro* to $Stack_{Clusters}$;
3 let msa be an empty set of clusters;
4 **while** $size(Stack_{Clusters}) > 0$ **do**
5 let $cluster \leftarrow pop(Stack_{Clusters})$;
6 let $child_1, child_2 \leftarrow getChildren(cluster)$;
7 **if** $avg(F_{Quality}(child_1), F_{Quality}(child_2)) > F_{Quality}(cluster)$ **then**
8 push $child_1, child_2$ to $Stack_{Clusters}$;
9 **else**
10 add $cluster$ to msa
11 **end**
12 **end**
13 return msa;

average of its child nodes (0.245), so its children are added to the stack. Otherwise, we add the parent cluster to the final microservice candidate set (msa) (line 10). We iterate

over the previous instructions until $Stack_{Clusters}$ is empty (lines 4-11). Intuitively, the algorithm will iterate until it empties the stack or reaches the leaves of the dendrogram. In the latter case, the leaves are automatically added to the msa set. The end result is a set of clusters (or microservice candidates) which is a decomposition of the monolith (line 13).

We apply this algorithm on the dendrogram generated for *JPetStore*. This results in 3 microservice candidates (MS1, MS2, MS3).

3.6 Evaluation

3.6.1 Data Collection

We selected a set of monolithic applications of various sizes (small, medium, and large) found in the literature. In Table 3.2, 4 different applications are presented, which are commonly used in the literature.

Table 3.2: *Applications used in the following experiments.*

Application name	No of classes	Lines of Code (LOC)
FindSportMates[5]	21	4.061
JPetStore[6]	24	4.319
SpringBlog[7]	87	4.369
ProductionSSM[8]	226	31.368

3.6.2 Research Questions & Methodology

To validate our approach, we conducted a set of experiments with the goal of answering the following research questions:

- **RQ1**: Impact Study. What is the impact of the extraction of the layered architecture on the identification of microservice candidates? While we propose to extract layered architecture artifacts from the existing source code, it does not necessarily contribute to the identification of the best microservice candidates. Indeed, naive approaches using a clustering technique with an objective function may be enough to recover microservice candidates close to the ground truth. With this question, we want to measure the impact of recovering the layered architecture of the monolithic application on the identification of a microservice-oriented architecture when compared to a naive approach.
- **RQ2**: Qualitative Comparison Study. How does our approach compare to the state of the art? The goal of **RQ2** is to compare our approach in relation with other approaches found in the literature. Particularly, we wish to highlight the issues in identifying microservices pertaining to decomposing microservices as to avoid the Wrong Cuts antipattern. Furthermore, we wish to motivate the benefits of taking into consideration of the internal layer during the decomposition.

The results from our approach, the tools used for measuring our results, as well as the implementation of our approach are available in our repository[9].

[5] https://github.com/chihweil5/FindSportMates
[6] https://github.com/mybatis/jpetstore-6
[7] https://github.com/Raysmond/SpringBlog
[8] https://github.com/megagao/production_ssm
[9] https://gitlab.com/icsa2022paper22/LayeredIdentificationApproach2022

3.6.2.1 RQ1 Methodology

We perform 3 different experiments to evaluate the impact of our proposed approach:

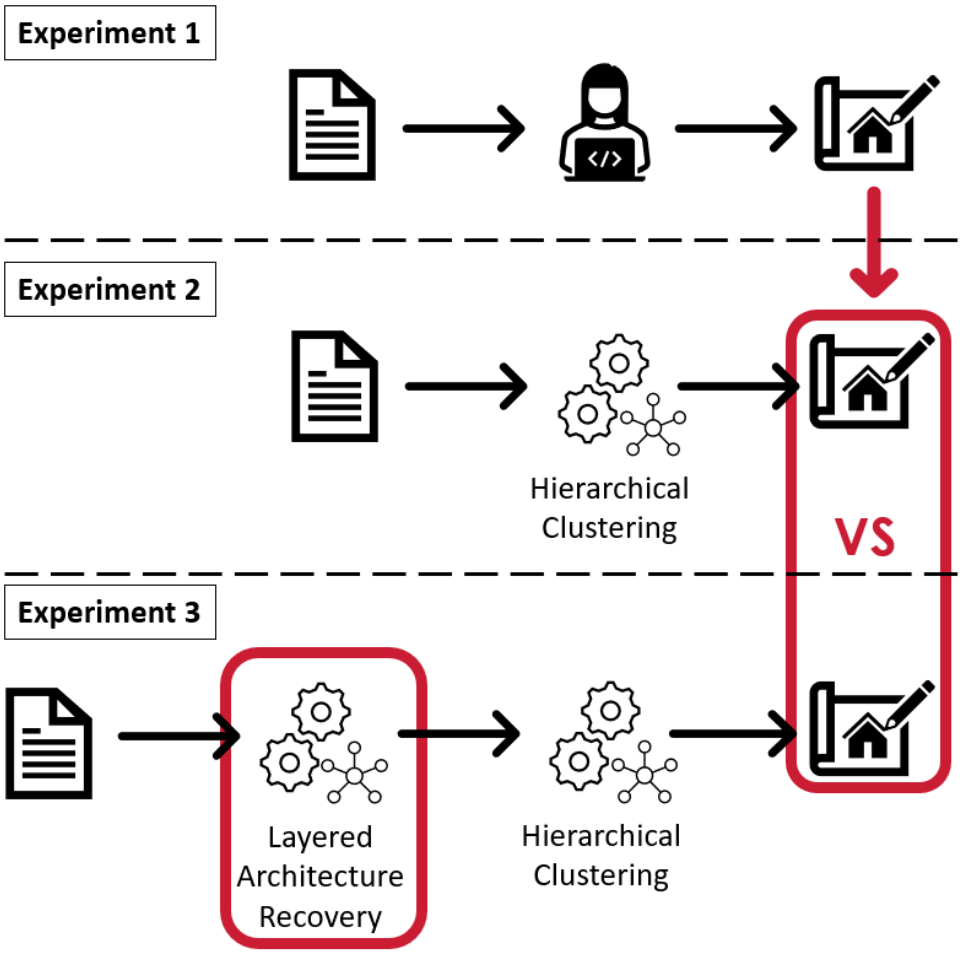

Figure 3.9: *Identification process for each experiment regarding RQ1.*

a) **Experiment 1**: Manual microservice identification. We perform a manual identification of microservice candidates. For each monolithic application, we use the source code as the primary artifacts for identifying microservice candidates. The identification was performed by 1 researcher and 3 R&D engineers. They have 12, 4, 7, and 8 years of experience in software decomposition, particularly towards microservice and component identification. We applied the pre-established identification pattern proposed in [Ric18a], namely *Decompose by Business Capabilities*. This pattern involves identifying business capabilities (and sub-capabilities) from the existing application (e.g., user, product, and catalog management), and mapping them to services. From there, the classes are placed into a service based on their relevance to the business capability and dependence between each other. We used static analysis to generate the dependency graph to help in the decision-making for classes that overlap between two different services. In the case of FindSportMates, JPetStore, and SpringBlog we followed this pattern. In the case of ProductionSSM, we were able to reuse the manual identification proposed by [ZLD+20]. [ZLD+20]'s work, they describe an identification strategy similar to the one we followed. Indeed, they

first identify a set of end-user functionalities. Then, they manually partition the set of classes belonging to the application into the different functionalities. Similarly, in our process, they use a debugging method to find which classes are related. As they did not consider java-interfaces in their partition, we supplemented their identification by adding them based on which classes implemented them. This was to have a global view of the application. All partitions are available in the aforementioned repository.

b) **Experiment 2**: MSA identification without considering the layered architecture. In this experiment, we identify microservices based on the hierarchical clustering presented in this paper. As input artifacts, we use the java-interfaces and classes of each monolithic application. These artifacts are clustered according to Algorithm 1 & Algorithm 2 and the objective function of Eq. 3.4 presented in Section 3.5. The results of this clustering algorithm are groups of java-interfaces/classes in which each group represents a candidate microservice.

c) **Experiment 3**: MSA identification by considering the layered architecture. In this experiment, we apply the proposed approach to its fullest. First, we apply the layered architecture recovery step (presented in Section 3.4) on each monolithic application, in order to map each java-interface and class to a LayerArtifact entity. Then, we apply the hierarchical clustering presented in Section 3.5 on the LayerArtifact entities to produce groups of LayeredArtifact entities. Finally, for each group, we replace the LayerArtifact entities with their implementation (i.e., java-interfaces and classes). The partition resulting from this replacement is then used to evaluate the impact of our approach.

All partitions are available on the provided repository, as well as the code used to perform Experiment 2 & 3. We propose a comparative study between the results of the proposed approach (Experiment 3) and the results obtained without considering the layered architecture recovery (Experiment 2). By comparing both experiments, we aim to highlight that the added layered architecture recovery has a positive impact on the identification of a microservice architecture. To illustrate this study, we provide Figure 3.9. To measure the impact of the layered architecture recovery, we measure the similarity of each recovered architecture (Experiment 2 & 3) with regard to the proposed ground-truth (Experiment 1). Inspired by the experimental method proposed in [LCG+15], we use two different accuracy measures to evaluate the identification approach. Particularly, we apply MoJoFM [WT04], and $c2c_{cvg}$ [LBG+15], proposed to evaluate modularization techniques by measuring the similarity of a recovered architecture with regard to the proposed ground-truth. We then can compare the similarity measures of Experiment 2 & 3 to see which performs better. We reason that by using two different metrics, if Experiment 3 consistently outperforms Experiment 2, we reduce the bias of selecting a metric that favors one particular experiment.

The MoJoFM metric is used to evaluate different modularization techniques [WT04]. This metrics is commonly used in the literature to evaluate architecture recovery approaches [LCG+15, PIIL21]. During the evaluation, the identified microservice-oriented architectures are compared with the ones prepared by experts. The MoJoFM is calculated by Eq. 3.5, where $mno(A, B)$ is the minimum number of operations (e.g., move or join) required to transform the architecture proposed by the approach (A) into the

ground truth (B) [WT04]. In addition, $max(mno(\forall A, B)$ calculates the actual maximum distance to partition B.

$$MoJoFM(M) = (1 - \frac{mno(A, B)}{max(mno(\forall A, B))}) \times 100\% \tag{3.5}$$

The higher the value of MoJoFM, the more similar the identified architecture is to the ground truth. Inversely, a lower score indicates that the identified architecture is further from the ground truth.

The cluster-to-cluster coverage ($c2c_{cvg}$) [LBG+15] is a metric used to measure the degree of overlap of the implementation-level entities between two clusters, using the following equation as a base:

$$c2c(c_i, c_j) = \frac{|entities(c_i) \cap entities(c_j)|}{max(|entities(c_i), entities(c_j)|)} \times 100\% \tag{3.6}$$

Eq. 3.6 is used in Eq. 3.7 to evaluate the similarity between two clusters. More precisely, Eq. 3.7 calculates the best similarity for each cluster of the recovered architecture with regard to the ground truth. It is then compared to a pre-determined overlap threshold (th_{cvg}) to count the cluster as covered. The authors of [LCG+15], define the thresholds values of 50%, 33%, and 10%. The first value depicts $C2C_{cvg}$ for $th_{cvg} = 50\%$ which is referred to as the majority match [LCG+15]. This threshold is used to measure the clusters produced by the proposed approach which mostly resemble the clusters proposed by the ground truth [LCG+15]. While, the remaining threshold highlight the moderate (33%) and weak (10%) matches [LCG+15].

$$simC(A_1, A_2) = \{c_i | (c_i \in A_1, \exists c_j \in A_2) \wedge (c2c(c_i, c_j) > th_{cvg})\} \tag{3.7}$$

$$c2c_{cvg}(A_1, A_2) = \frac{|simC(A_1, A_2)|}{|A_2.C|} \times 100\% \tag{3.8}$$

The final count of the number of covered clusters is used in Eq. 3.8 (i.e., $c2c_{cvg}$) to calculate the total coverage of the recovered architecture with regard to the ground truth architecture. The result of this metric is highly dependent on the threshold used. The higher the threshold, the higher the extent to which the clusters produced by the approaches mostly resemble the clusters in the ground truth.

We use these two metrics to compare the similarity score of Experiment 2 and Experiment 3. If the results of Experiment 3 consistently outperform the results of Experiment 2, then the intermediary step of recovering the layered architecture of the monolith has a positive impact on the identification of microservices.

3.6.2.2 RQ2 Methodology

To highlight the issues of identifying microservices without creating Wrong Cuts, we have extracted the class partitions of JPetStore from different approaches to compare them. First, we have recovered the partitions presented, as is, in [AM21, JLZ+18]. In addition, we have contacted the first author of [SSB+20b] to apply their identification approach on JPetStore. For the approach proposed by Selmadji et al., they reported that their approach with the quality metric's ($FMicro(M)$) coefficients calibrated to $\alpha = 1$ and $\beta = 3$ [SSB+20b]. Finally, we use the partition of the proposed approach (Experiment 3) on JPetStore. We compare all these partitions with the expert partition proposed in Experiment 1. We analyze each proposed architecture and compare them with the expert architecture to highlight the common pitfalls of microservice identification.

3.6.3 Results and Discussion

3.6.3.1 RQ1

Table 3.3 and 3.4 respectively show the results of MoJoFM and $C2C_{cvg}$ to evaluate the overall accuracy of the different experiments with regard to the ground truth. Experiment 2 corresponds to the proposed identification approach without the use of the layered architecture artifacts. On the other hand, Experiment 3 corresponds to the proposed identification approach with the use of the layered architecture artifacts. In both experiments, we applied the approach to the same set of applications presented in Table 3.2. Generally, our results indicate that the use of the layered architecture artifacts improved the accuracy of the identification approach. According to the results of MojoFM, the use of layered architecture artifacts increased the accuracy of the identification approach **by at least 14%** (in the case of SpringBlog). On average, **Experiment 3 performed 23.98% better than Experiment 2**.

Table 3.4 shows $C2C_{cvg}$ for three different values of th_{cvg} (i.e., 10%, 33%, and 50%) for each application under both Experiment 2 & 3. The first value depicts $C2C_{cvg}$ for $th_{cvg} = 50\%$ which is referred to as the majority match [LCG+15]. The scores in which one experiment performed better than the other are in bold. We denote that generally, **the use of layered architecture artifacts increases the quality of the microservice architecture**. For FindSportMates, the use of layered artifacts was able to produce perfect matches which explains the 100% majority matches. With JPetStore, we notice a drop of matches once we reach 50% threshold. Indeed, in Experiment 2 the approach was

Table 3.3: *MojoFM results (in %age).*

Applications	Experiment 2	Experiment 3
FindSportMates	70.0%	**100.0%** ($\uparrow 30\%$)
JPetStore	55.0%	**89.47%** ($\uparrow 34,5\%$)
SpringBlog	58.06%	**72.09%** ($\uparrow 14\%$)
ProductionSSM	57.73%	**75.16%** ($\uparrow 17,4\%$)

unable to create pertinent microservice candidates. In the case of SpringBlog, the use of layered artifacts produced better moderate and weak matches but failed to produce strong matches. However, without the use of these artifacts, **Experiment 2 produced few majority matches** (7.41%). This seems to indicate that the low performance is due to the clustering technique. Indeed, we notice a similar drop in score with the MoJoFM, which seems to support that hypothesis. With regard to the largest application (ProductionSSM), we denote that the use of layered architecture artifacts greatly outperformed by producing more than 55% of majority matches, while without these artifacts, Experiment 2 produced 1.92% majority matches. In other words, **the use of the layered architecture produced 53% more majority matches**. Furthermore, when we increased the threshold to 66% Experiment 3 faced a small drop in the majority matches for ProductionSSM and remained at 48% while Experiment 2 dropped to 0%. This seems to indicate that the identified clusters of Experiment 3 are of even higher quality.

Summary RQ 1

The overall conclusion from the results of these metrics (MoJoFM & $C2C_{cvg}$) is that the use of the layered architecture artifacts does indeed increase the accuracy of the proposed approach for all systems, in almost every case, independent of the accuracy measure chosen.

Table 3.4: *Results of $C2C_{cvg}$ (in %age).*

Applications		Experiment 2	Experiment 3
FindSportMates	th10	100%	100%
	th33	100%	100%
	th50	25%	**100%** ($\uparrow 75\%$)
JPetStore	th10	100%	100%
	th33	30%	**100%** ($\uparrow 70\%$)
	th50	0%	**66.66%** ($\uparrow 66,6\%$)
SpringBlog	th10	77.78%	**86.67%** ($\uparrow 8,9\%$)
	th33	48.15%	**53.33%** ($\uparrow 5,2\%$)
	th50	**7.41%**	0.0% ($\downarrow 7,4\%$)
ProductionSSM	th10	**90.38%**	86.21% ($\downarrow 4,1\%$)
	th33	30.77%	**72.41%** ($\uparrow 41,7\%$)
	th50	1.92%	**55.17%** ($\uparrow 53,3\%$)

3.6.3.2 RQ2

Figures 3.13, 3.10, 3.11, 3.12, and 3.14 illustrate five different decomposition of the JPetStore application: one decomposition proposed by [AM21], another by [JLZ+18], and one by [SSB+20b], the decomposition proposed by our approach, and a manual expert decomposition. The manual expert decomposition proposes 3 different microservices based on functionality and vertical decomposition. The first microservice is related to order management, the second microservice provides account management, and

the third microservice relates to the product catalog. The decomposition tries to promote functional autonomy by limiting the inter-dependency between a microservice and the service layer of another microservice. This is the case except for the Account Microservice (MS2) which uses *CatalogService*. Furthermore, this decomposition tries to promote data autonomy by limiting the inter-dependency between a microservice and the data-access layer of another microservice. The expert decomposition is able to promote this except for the data-access related to the *Item* data-entity which is used by Order Microservice (MS1) to generate an order of items. In both cases, the decomposition limits the inter-microservice access to read-only features. For instance, the Account Microservice consults *CatalogService* to read the user's favorite category, and the Order Microservice to read the items being bought.

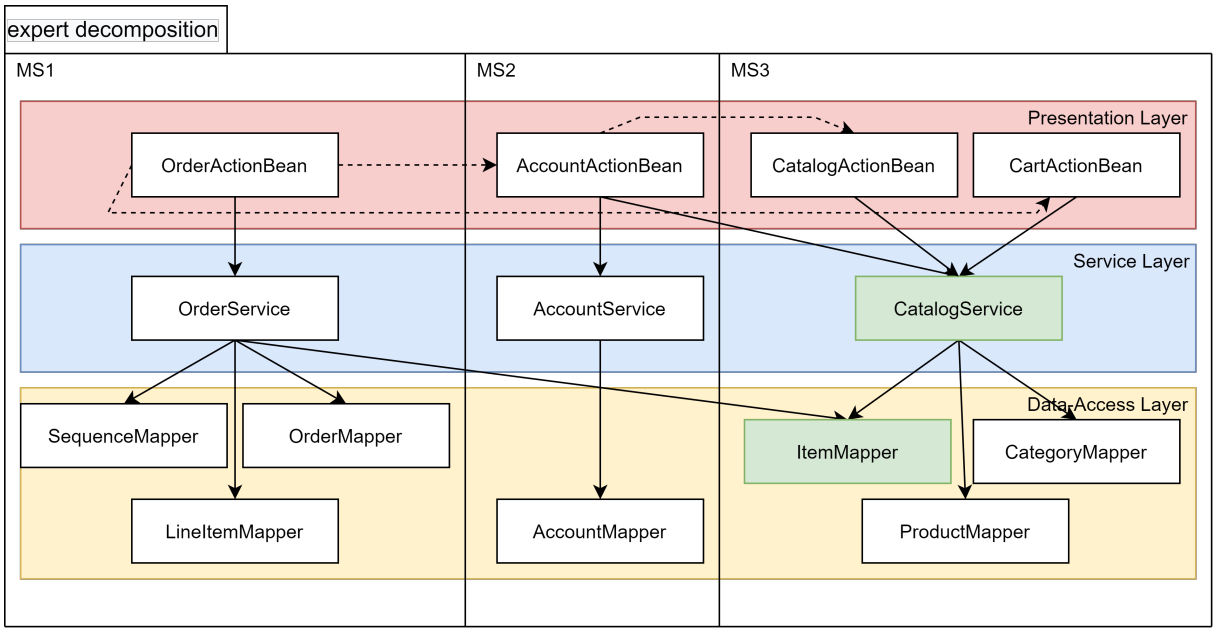

Figure 3.10: *Expert decomposition of JPetStore.*

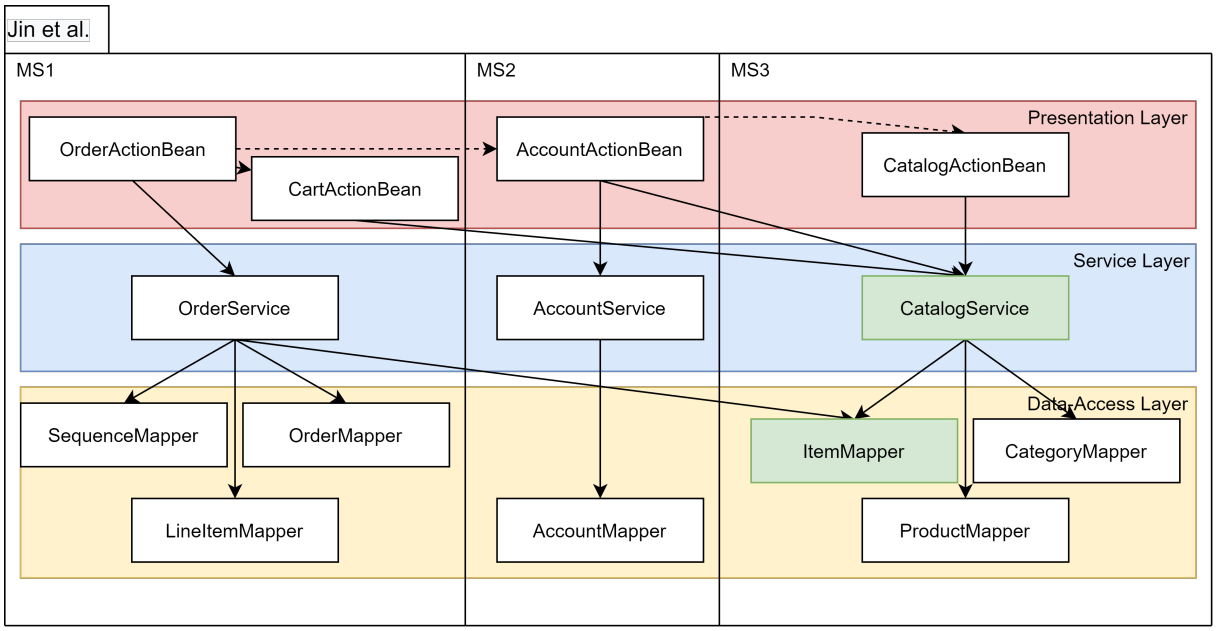

Figure 3.11: *Decomposition of JPetStore proposed by [JLZ+18]*

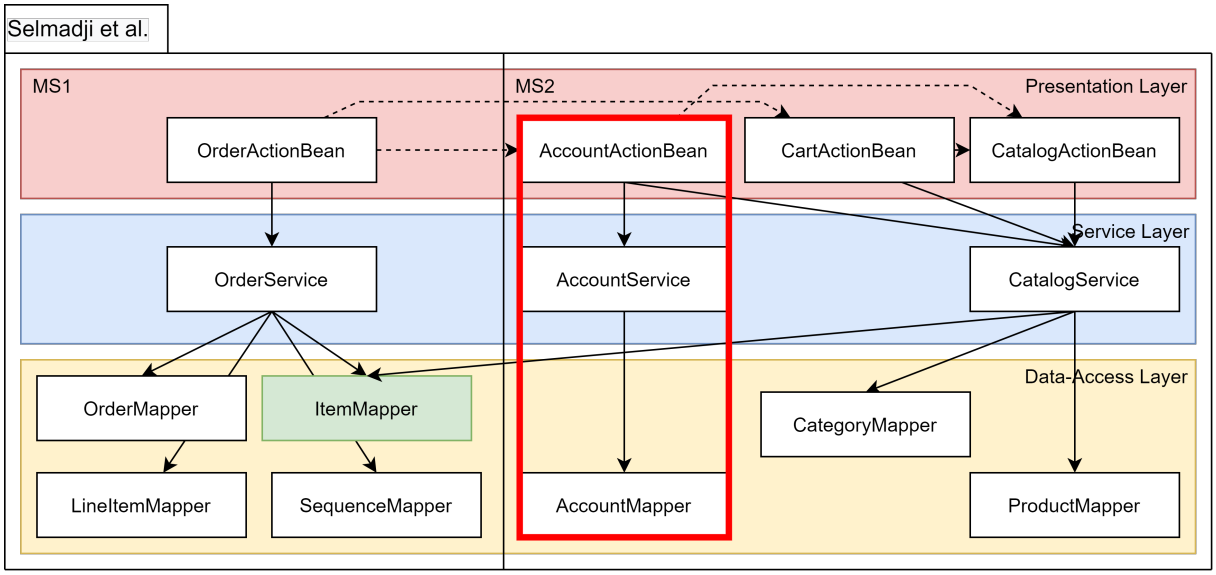

Figure 3.12: *Decomposition of JPetStore proposed by [SSB+20b]*

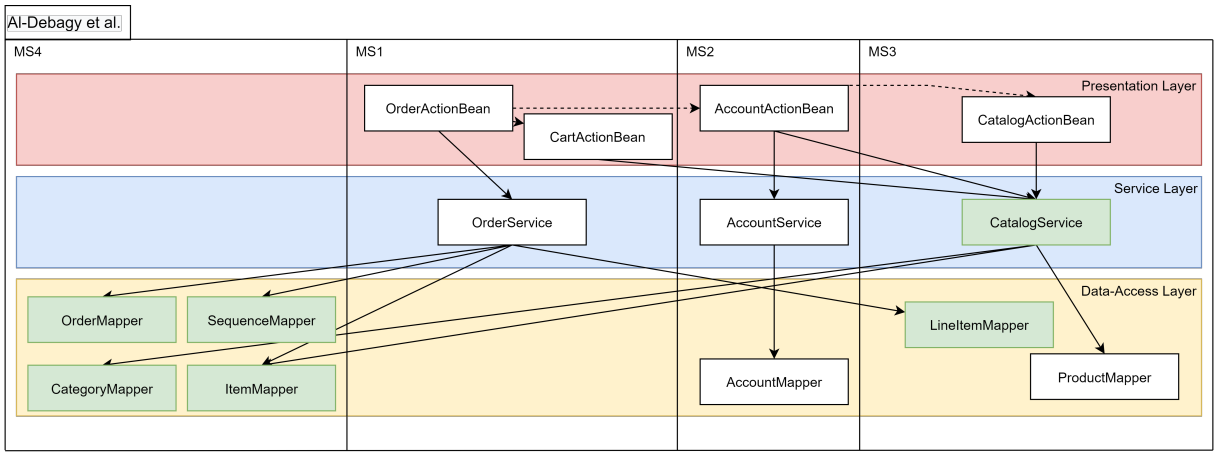

Figure 3.13: *Decomposition of JPetStore proposed by [AM21]*

Regarding the decomposition proposed by Al-Debagy et al. [AM21], they propose 4 microservice candidates of which 2 offer a vertical decomposition (MS2 & MS3). The other two microservice candidates show a horizontal decomposition along its technical layers. This is the typical example of a Wrong Cuts antipattern, where MS4 serves a technical need (that of data-access) to two different microservices, which may lead to increased network communication. The decomposition produced by Selmadji et al. [SSB+20b], generates two microservices. While, they manage to avoid creating Wrong Cuts, they were unable to differentiate between the account and the catalog service. Regarding the decomposition of Jin et al. [JLZ+18], we observe a decomposition that respects the vertical decomposition and is similar to the expert decomposition. However, the approach places *CarActionBean* in a different microservice candidate. This is likely because of the dependency between the *OrderActionBean* and the *CartActionBean*. However, this decomposition creates another vertical dependency between MS1 and MS3. In contrast, our approach generated another decomposition that is similar to the expert decomposition. However, it placed *CartActionBean* in its own microservice. This is likely because our approach relies on data-oriented objective function as *CartActionBean* uses data exclusive to its function (e.g. Cart, CartItem). Since these data types

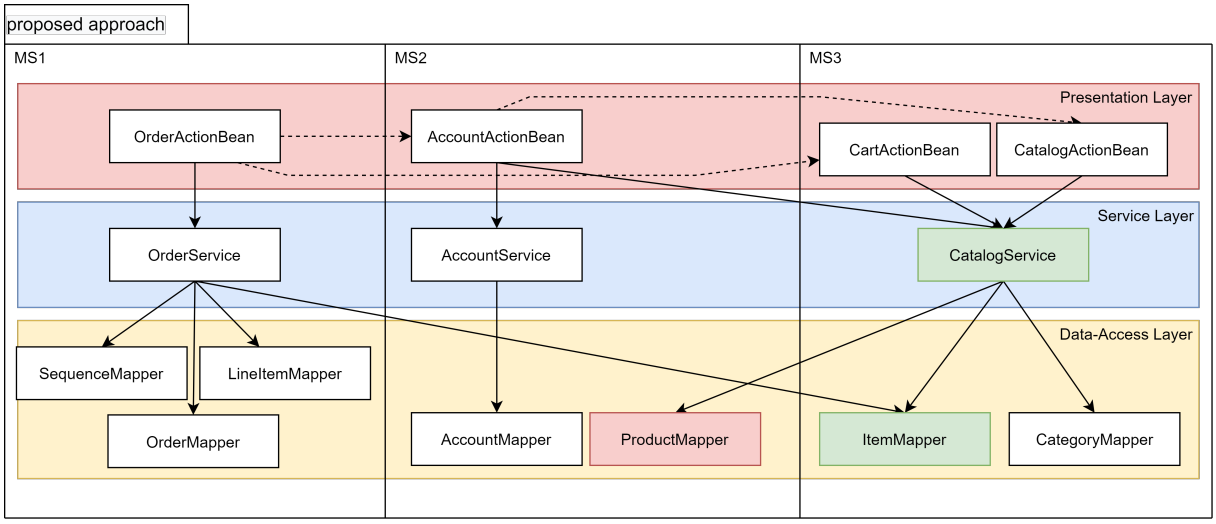

Figure 3.14: *Decomposition of JPetStore of the proposed approach*

were only used in this class it is likely that the approach opted to propose a separate microservice candidate.

Both the proposed approach and Jin et al.'s approach recovered a similar architecture to the expert decomposition while avoiding creating Wrong Cuts. However, Al-Debagy et al.'s approach decomposes JPetStore along technical layers, which usually results in highly-dependent microservices.

Summary RQ 2

In this experiment we set out to highlight the challenge of identification approaches which involves decomposing a monolith vertically as to avoid the harmful Wrong Cuts antipattern. In the case of Al-Debagy et al.'s decomposition, we highlight that their approach was unable to avoid this antipattern by creating microservices based on their technical layers. Furthermore, Selmadji et al.'s decomposition failed to identify all 3 microservice candidates. This was not the case for our approach. As a result, we conclude that our approach was able to avoid these antipatterns and provide a higher quality decomposition.

3.6.4 Threats to Validity

3.6.4.1 Internal Validity

First, there is not necessarily a unique architecture for a system [GKMM13]. However, we base our results on one ground-truth architecture. To reduce this threat, we used two different metrics when comparing the ground truth with the identified architectures which both indicate a positive impact for the proposed approach. In the future, we intend to include other metrics to measure different aspects of the recovered architecture, such as system-level dependencies which is independent of the ground truth. Secondly, to limit the bias related to constructing the ground truth for the Experiment 1, we have performed the experiment using 4 people with different profiles. By doing

so, we limit the influence of an individual's point of view. Furthermore, the approach and its results were not presented or produced until after the manual identification. Finally, we use the expert decomposition of `ProductionSSM` proposed by the authors of [ZLD+20]. In this instance, we are able to completely remove the bias by using the expert decomposition of a team which is not involved in this experiment.

3.6.4.2 External Validity

Another threat to the validity of our experimentation is that we focused our selection of experimental candidates to applications implemented in JAVA. While, JAVA is a popular language in the industry, there are over languages that are not covered. We attempt to mitigate this threat by selecting applications of varying size and from different studies. However, the current literature does not provide enough applications with a recovered architecture. In future works, we would like to extend the number of applications with a publicly available decomposition, and an evaluation framework to facilitate the evaluation of other identification approaches.

3.7 Conclusion

In this chapter we have highlighted the importance of the internally-layered architecture of the monolith when identifying microservices. However, most approaches do not take into consideration the internal architecture of the application when making their decomposition. Indeed, according to a study by Taibi et al. [TLP20], there are several antipatterns (Wrong Cuts & Shared Data Persistence) that can be directly tied to the identification phase. Particularly, during the evaluation of the comparison study, we noted that approaches do fall into this trap.

In turn, we have proposed an approach which leverages the internally-layered architecture of monolith to propose a microservice-oriented architecture that avoids these antipatterns while also promoting good design patterns. In addition to the internal architecture, we also take into consideration the domain of the application to promote data autonomy. Concretely, we were able to demonstrate that taking into consideration the internal architecture of the monolithic application has a positive impact on the identification process (**RQ1**) when compared to the same identification process that did not take the architecture into consideration. Furthermore, we highlighted that there is a recurring antipattern found in several approaches that attempt to identify microservice candidates based on the cohesion at the class-level (**RQ2**).

In a practical sense, we highlight two takeaways (1) that extracting the internal architecture is essential towards understanding the monolith and making better microservice identification decisions, and (2) existing approaches that do not take this into consideration risk falling into known antipatterns.

For the experimentation we have limited ourselves to object-oriented systems. However, we believe that this architectural design can be found in other types of systems.

This validation can be extended by taken into consideration multiple languages as well as different evaluation metrics. Otherwise, the proposed approach identifies a quality microservice-oriented architecture that can be used to guide the migration effort of the application. However, a refactoring of the monolithic source code is still required for it conform to the identified architecture. In the next chapter, we address this issue by proposing an approach to automate the transformation of the monolith's source code.

IV

Microservice materialization

Contents

4.1 Introduction

In the previous chapter, we presented an approach that identifies a microservice architecture from an existing monolithic application. After the target architecture is validated by an expert, it must be materialized into a source code that will be hereafter used by the development team. However, this materialization process requires that developers re-write (i.e., transform) the existing source code to conform to the target architecture. In an industrial context, this may require re-writing large applications which could take months, or even years, to accomplish. Furthermore, while the developers are tasked with evolving the existing application they must also continue fulfilling the clients' needs. This makes long manual re-writes difficult. Therefore, the materialization process must be automated to facilitate the migration.

During the identification process, an architecture description is generated to describe the decomposition of an OO application. These microservice identification approaches often propose an architecture description that describes each microservice as a cluster of classes from the monolithic application (see Chapter 2). However, despite their best efforts these approaches can only reduce OO dependencies between clusters, and not completely eliminate them. As each identified cluster encapsulated into their respective microservice project, these dependencies are no longer possible as they reference classes defined in different projects. Indeed, the inter-cluster dependencies prevent the proper encapsulation of the identified microservices and are defined as microservice encapsulation violations that must be resolved.

Definition 4.1.1: microservice encapsulation violation

A microservice encapsulation violation is an object-oriented dependency between two clusters that is no longer possible after they have been encapsulated into their microservices.

Indeed, as we shift from an OO paradigm to an MSA one, we need to replace these OO-type dependencies between microservices into MSA-type dependencies. To do so, in this chapter we propose a set of refactoring patterns to resolve these OO-type dependencies into MS-type dependencies. More precisely, we aim to answer the following questions:

1. What are the different types of encapsulation violations?
2. What are the refactoring patterns to resolve each type?
3. If they cannot be resolved directly, how can we reduce the violation into a resolvable violation?

In this chapter, we present our semi-automated approach towards materializing the microservice-oriented architecture. This approach uses the target architecture generated by identification approaches such as the one presented in Chapter 3 and the source code of the existing monolithic application to materialize the target source code. In the next section, we highlight the challenges behind encapsulating the identified

clusters of classes into separate microservice projects through a motivating example. In Section 4.3, we present a 4-step Workflow of our approach for materializing the MSA. Then, in Section 4.4, we focus on the first step by defining and detecting the different types of encapsulation violations. In Section 4.5, we focus on the second step by presenting the transformation patterns for resolving the detected encapsulation violations. In Section 4.6, we apply our approach on a set of monolithic applications by transforming their architecture into a microservice-oriented one and evaluate the quality of the resulting transformation, as well as their performance. Finally, in Section 4.7 we conclude this chapter with our observations and our perspectives.

4.2 Challenges towards materializing an MSA

To tackle the problems we face during the materialization phase of the migration process, we use an illustrative example of a display screen management system (e.g., an information panel in an airport). It is composed of the *DisplayManager* class that is responsible for handling the information to be displayed on the *Screen* class (see Fig 4.1). It does so through a *ContentProvider* class that implements methods for stacking content such as incoming messages (i.e., *Message* instances) or the current time (i.e., *Clock* instances). Finally, the *Clock* class uses an instance of the *Timezone* class to get the time based on its GPS location.

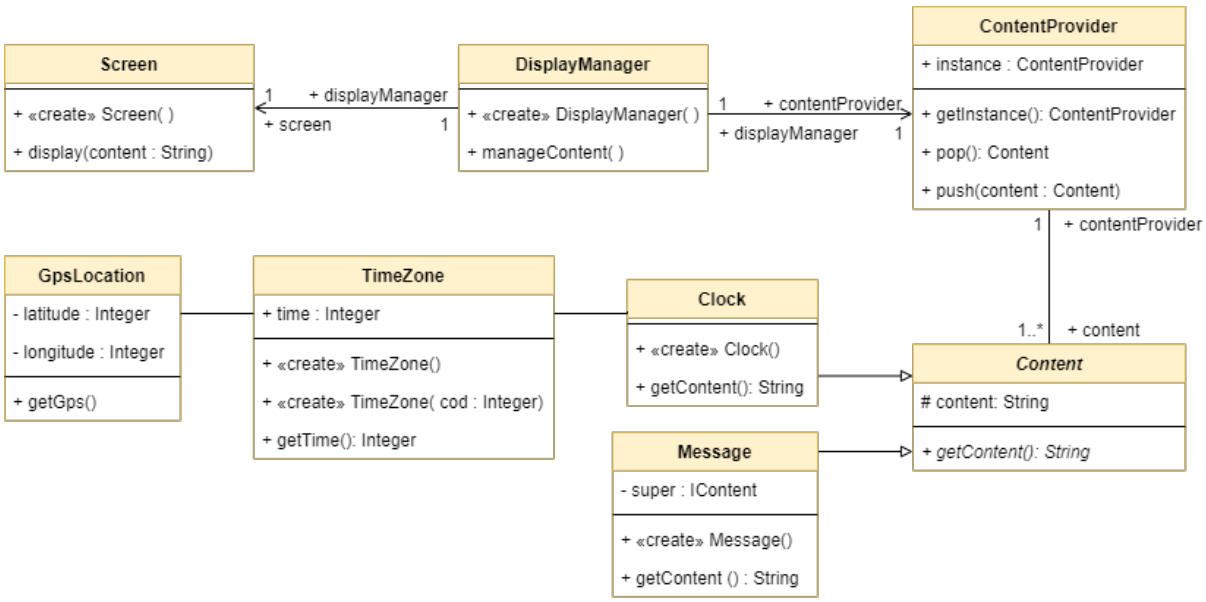

Figure 4.1: *Information Screen class diagram*

During the identification phase of the migration clusters of classes are extracted from the monolithic application. Each cluster forms the basis of a structurally and behaviorally-valid microservice candidate. Existing approaches propose different clustering techniques to maximize the quality of the identified microservice candidates. However, as these clusters are encapsulated into their own microservice project, inter-cluster dependencies are revealed. Indeed, despite efforts to extract independent microservices, identification approaches are unable to extract microservices without creating inter-cluster dependencies. In the case of *Information Screen*, every class is depen-

dent on one or more classes to function. This makes it impossible to extract completely independent clusters of classes.

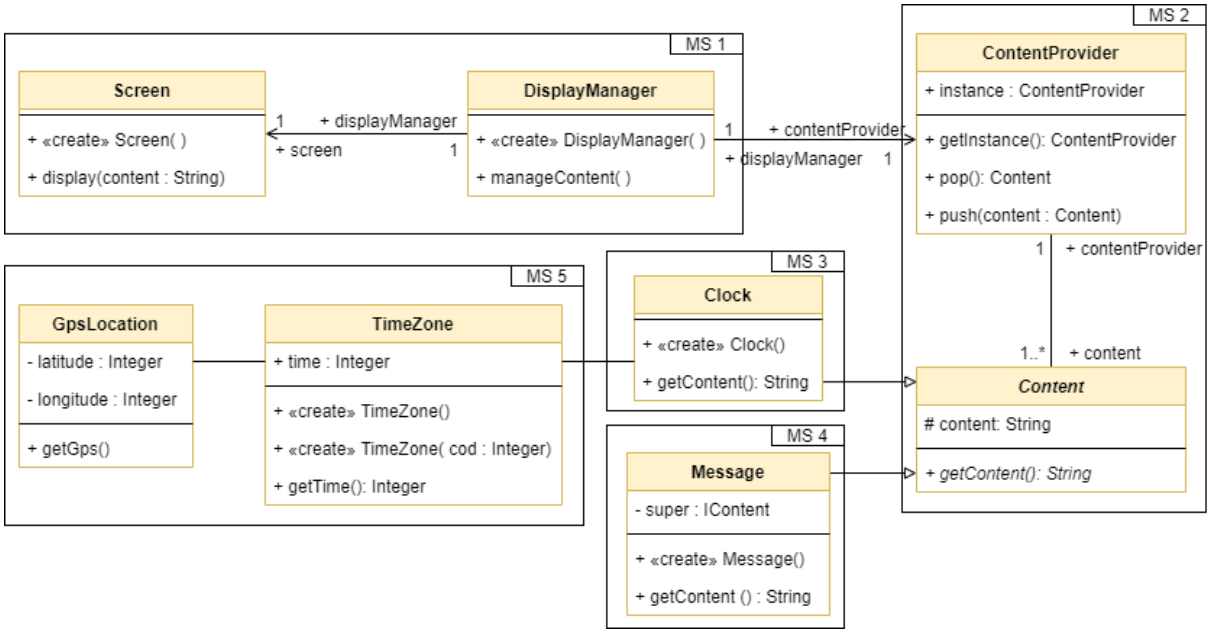

Figure 4.2: *The Transformation Process using the MonoToMicro tool.*

In Figure 4.2, when *Information Screen* is partitioned into 5 microservice candidates, there still exists inter-cluster OO-type dependencies that must be resolved before the candidates can be fully encapsulated. Indeed, during the extraction process of *Information Screen*, 4 different inter-cluster dependencies can be seen. In particular, the association between the *DisplayManager* and *ContentProvider*, as well as the association between *TimeZone* and *Clock* are no longer possible. Furthermore, the inheritance properties between the parent class *Content* and the child classes *Clock* and *Message* are also exposed. Other underlying dependencies that do not appear in the diagram can also be found in the source code, such as *Clock* accessing an attribute of *TimeZone* or *ContentProvider* throwing a stack overflow exception that must be handled by *DisplayManager*.

However, before we can materialize the identified microservices, we must transform these inter-cluster dependencies in the goal of healing the resulting encapsulation violations. To do, we must first establish a *systematic* and *automated* way to detect and transform each violation. Indeed, the first obstacle towards materializing an MSA is to explicitly define the different encapsulation violations that are possible, and establish detection rules to systematically reveal all encapsulation violations. After establishing detecting the various encapsulation violations, the second obstacle involves defining a set of transformation patterns to *systematically* resolve the different violations. In the next section, we present the global workflow of our approach to solve these two obstacles and properly package the materialized MSA.

4.3 Global Workflow of Macro2Micro

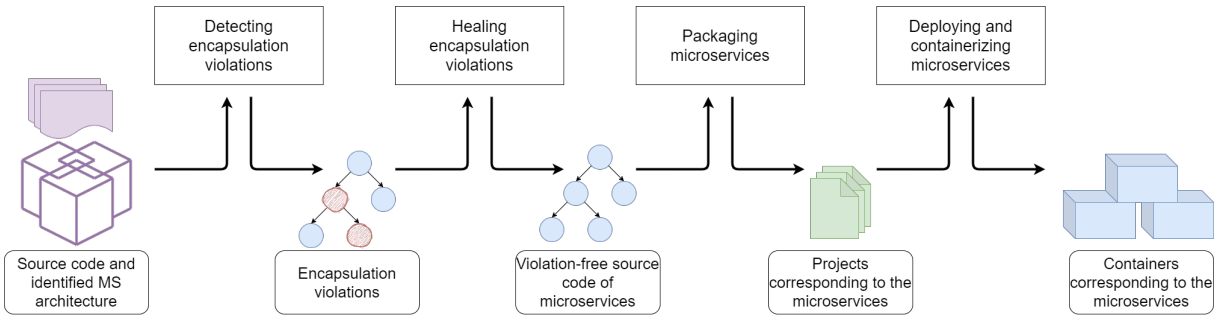

Figure 4.3: *The Transformation Process using the Macro2Micro tool.*

To handle the challenges identified in the previous section, we propose a systematic way of transforming a monolithic OO application into an MSA application through the use of a set of transformation patterns. The purpose of this approach is to generate the source code of the MSA by encapsulating the clusters discovered during the initial migration step. To do so, we define a process composed of four steps as presented in Figure 4.3 which consist of: (1) detecting encapsulation violations, (2) healing encapsulation violations, (3) packaging microservices, and (4) deploying and containerizing microservices.

4.3.1 Detecting Encapsulation Violations

Each recovered cluster of classes is encapsulated in its own microservice project to materialize the recovered microservice candidates from the source of object-oriented software. Upon encapsulation, OO dependencies between clusters are no longer permitted as they cause encapsulation violations. Therefore, they must be transformed into MS-type dependencies. However, before the transformation can take place these encapsulation violations must be detected. To facilitate the detection of these encapsulation violations, a set of encapsulation violation rules are proposed to analyze the monolith:

Definition 4.3.1: Microservice encapsulation detection rules

(**Rule 1**) Method Invocation: if a cluster's method invokes a method belonging to a class from another cluster then it is a method invocation violation.
(**Rule 2**) Attribute Access: if a cluster's method accesses an attribute belonging to a class from another cluster then it is an access violation.
(**Rule 3**) Instance Handling: if a cluster's class contains a reference targeting a class from another cluster then it is an instance violation.
(**Rule 4**) Inheritance: if a cluster's class inherits a class belonging to another cluster then it is an inheritance violation.
(**Rule 5**) Exception Handling: if a cluster's method throws, catches or declares an exception defined in another cluster then it is a thrown exception violation.

However, to use these detection rules they must be formalized and applied systematically. To do so, in Section 4.4 we propose a formalization of these rules and apply them on the AST[1] representation of the OO source code. Initially, the target architecture description is used to partition the AST nodes that represent the classes into clusters. Then, each node is parsed for references towards class nodes belonging to another cluster using the aforementioned detection rules. To facilitate the formalization of the detection rules, a representative model of the references between class nodes are proposed. Then, these detection rules are applied on each cluster. After all the violations have been detected, they can be healed.

4.3.2 Healing Encapsulation Violations

The violations detected in the preceding step must be healed using transformation rules in order to properly encapsulate microservices. These transformations must either fully heal or reduce a violation to a solvable one. Indeed, later when we propose our set of transformation patterns, some patterns end up creating more violations. To this effect, we consider a transformation pattern only reduces a specific violation if it creates another violation.

Previously, the detection of encapsulation violation covered method invocation, attribute access, instance handling, inheritance, and exception handling. In Section 4.5, we present a set of transformation rules to systematically heal the microservice encapsulation violations identified in the first step. Concerning the transformation patterns that only reduce violations, we also present a transformation order to heal all violations efficiently (see Section 4.5.6).

4.3.3 Packaging and Deployment of an MSA

Once the MSA source code has been healed, it must be packaged and made deployable. In step (3), the violation-free microservices are packaged. To accomplish this, we generate a project for each microservice, its file structure, and library dependencies. Furthermore, the healed AST class nodes are printed into their respective microservice projects. In step (4), the microservice-oriented architecture is made deployable to the cloud by creating the necessary configuration files (e.g., dockerfiles) for each microservice. Furthermore, a composition file is also created, which organizes all the microservices when deployed together.

In this chapter, we concentrate on the first two steps of the transformation phase, which comprise the major scientific roadblocks previously highlighted, and leave the last two steps for the implementation as they comprise more technical roadblocks. Next, we present a systematic approach towards detecting microservice encapsulation violation.

[1] Abstract syntax tree

4.4 Detecting Microservice Encapsulation Violations

4.4.1 Detection Process and Model

The first step towards materializing the identified MSA, is to detect the microservice encapsulation violations that occur when we place each identified cluster into their own microservice project. However, the task of detecting all violations can be difficult, therefore it requires providing an automated and systematic way of detecting them. To detect these violations, we apply a static analysis on the source code of the monolithic application.

Concretely, we propose to parse and extract the source code's abstract syntax tree (AST) representation. Within the AST, there are nodes that represent the classes, attributes, and methods. Furthermore, dependencies between classes are also parsed and represented within the AST as references which are typed based on the type of dependencies. We use these references to guide the detection of microservice encapsulation violations.

To facilitate the detection of microservice encapsulation violations we need to formalize, or define, each violation. By formalizing these violations, we can also formalize the rules to detect them. We propose an object-to-microservice mapping model that maps the object-oriented elements from the generated AST to the identified microservice-oriented architecture (see Figure 4.4).

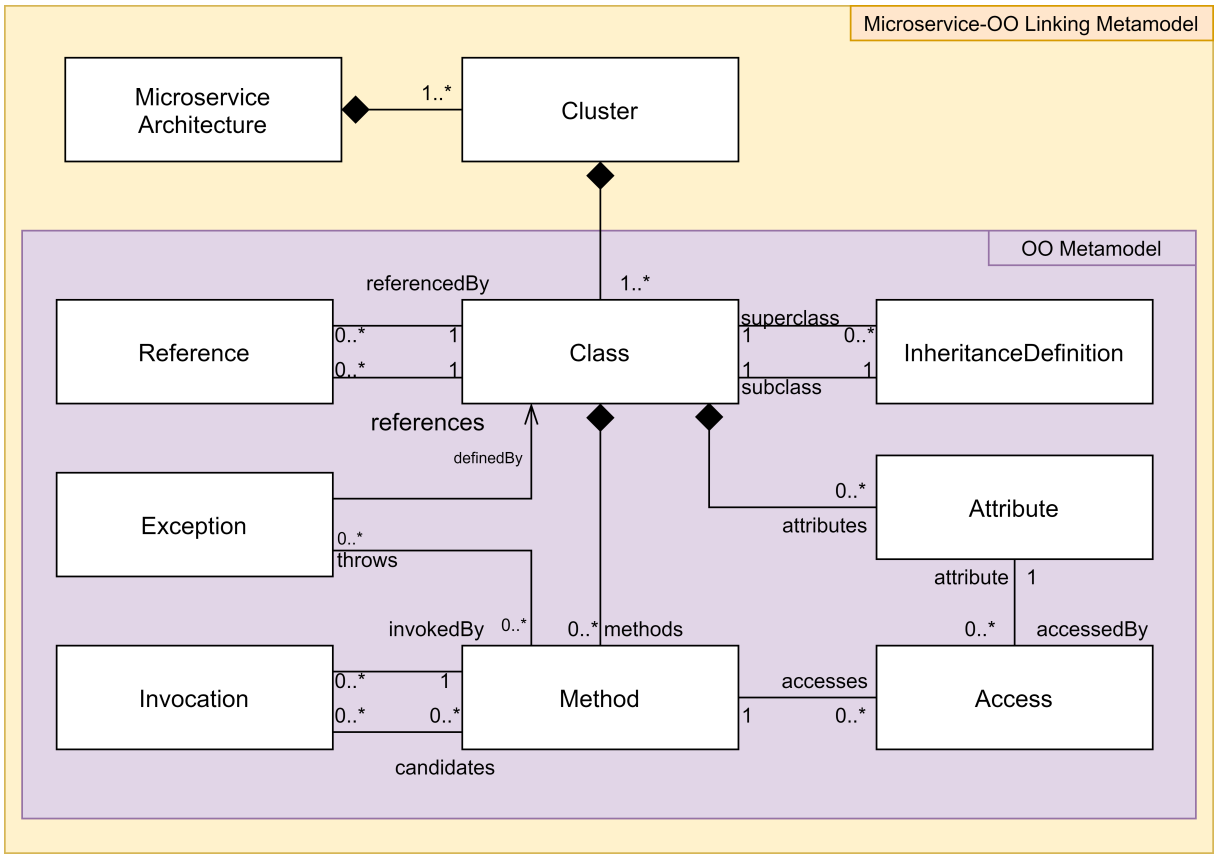

Figure 4.4: *Object-to-microservice model.*

This mapping model allows us to formally represent the different relationships between classes in addition to the clustering results during the identification phase (see Chapter 3). In the mapping model, we define a `Microservice Architecture` as being composed of one or more `Clusters`. In turn, a `Cluster` contains a set of `Classes`. Furthermore, the associations between the different classes are represented. For instance, we represent any class references between two classes with the entity `Reference`. Inheritance between two classes are represented by the `InheritanceDefinition` entity. In the case of attribute access and method invocation, they are both represented by their respective entities. Finally, thrown exceptions are taken are formalized by the association between the `Exception` entity and the `Method` entity.

4.4.2 Microservice Encapsulation Violation Detection Rules

From the model presented, we can now define a set of rules to detect specific violations. Particularly, we present 5 detection rules concerning the 5 type of encapsulation violations presented previously.

The first rule (**Rule 1**: Method Invocation) towards detecting encapsulation violations relates the method invocation violation. Particularly, this violation defines that a class invoking a method belonging to a class from another cluster should not be allowed. The detection of this violation is formalized in Listing 4.1 to be applied on the OO-to-Microservice model.

```
rule MethodInvocationViolation(c1: class):
    when:
        c1.methods.forAll(Method m1:
        m1.candidates.forAll(Invocation invoc:
        invoc.invokedMethod.class.cluster != c1.cluster))
```

Listing 4.1: *Violation Detection Rule 1: Method Invocation Violation.*

Secondly, if a cluster's method accesses an attribute belonging to a class from another cluster then it is considered an access violation. To formalize this definition, we propose the following violation detection rule (**Rule 2**: Attribute Access):

```
    rule AttributeAccessViolation(c1: class):
        when:
            c1.methods.forAll(Method m1:
            m1.accesses.forAll(Access access:
            access.attribute.class.cluster != c1.cluster))
```

Listing 4.2: *Rule 2: Attribute Access Violation.*

Thirdly, we consider the instance violation where one cluster's class contains a reference targeting a class from another cluster. Particularly, we mean to detect the act of instancing a class defined in another cluster. We use references to detect instances being created and handled. To formalize this definition, we propose the following violation detection rule (**Rule 3**: Instance Handling):

```
    rule InstanceReferenceViolation(c1: class):
        when:
            c1.references.forAll(Reference ref:
```

```
4            ref.references.cluster != c1.cluster))
```

Listing 4.3: *Rule 3: Instance Violation.*

Another encapsulation violation that we have to consider is the inheritance between classes belonging to different clusters (**Rule 4**: Inheritance). The detection of this violation is formalized in Listing 4.4, and applied on the Object-to-Microservice model.

```
1 rule InheritanceViolation(c1: class):
2     when:
3         c1.inheritanceDefinition.superclass.cluster != c1.cluster
```

Listing 4.4: *Violation Detection Rule 4: Inheritance Violation.*

Finally, if a method throws or catches an exception defined in another cluster then it is considered to be a exception throwing & handling violation (**Rule 5**: Thrown & Caught Exceptions). We define the detection rule relating to exception throwing & handling violations in Listing 4.5. Particularly, we detect that a method either throws or catches an exception defined in another cluster.

```
1     rule ExceptionViolation(c1: class):
2         when:
3             c1.methods.forAll(Method m1:
4             m1.throws.definedBy.cluster != c1.cluster)
5
```

Listing 4.5: *Violation Detection Rule 5: Thrown & Caught Exception Violation.*

We apply the detection rules on the AST of the monolith to generate a set of encapsulation violations. From this set, we can proceed to heal each violation systematically. Indeed, in the next section, we present a transformation pattern for each defined violations.

4.5 Healing Microservice Encapsulation Violations

After fragmenting the monolithic code into different microservices (e.g., clusters of classes), some classes are instanced in one microservice and used (i.e., invoked, referenced, accessed) in others. To remove these type of violations, it is necessary to provide adequate answers to the following questions to properly heal all violations:

1. How do we invoke a method existing in a class belonging to another microservice? (i.e., **Rule 1**)
2. How do we access attributes of objects belonging to another microservice? (i.e., **Rule 2**)
3. How do we create an instance of a class belonging to another microservice? When a given instance is referenced in several microservices, how do we ensure the

sharing of this instance while preserving the business logic of the application? (i.e., **Rule 3**)

4. How can a class inherit from a class defined in another microservice? (i.e., **Rule 4**)

5. How can we handle exception thrown from another microservice? (i.e., **Rule 5**)

Particularly, as we move from an object-oriented paradigm to a microservice-oriented one, the OO-type dependencies between microservices needs to be replaced by microservice-type dependencies (i.e., web services). In the following subsections we provide a set of transformation patterns to address the aforementioned questions and to transform the encapsulation violations detected in the previous section.

4.5.1 Method Invocation

The first encapsulation violation we address is the one related to (**Rule 1**) method invocation violation. Let's present a case from the motivating example provided in Section 3.2 where the class *DisplayManager*, placed in the microservice candidate 1 (MS1), invokes a method from the class *ContentProvider* which is placed in MS2 (see Figure 4.5). Particularly in Listing 4.6, *DisplayManager* invokes the method `pop()` of the class *ContentProvider*.

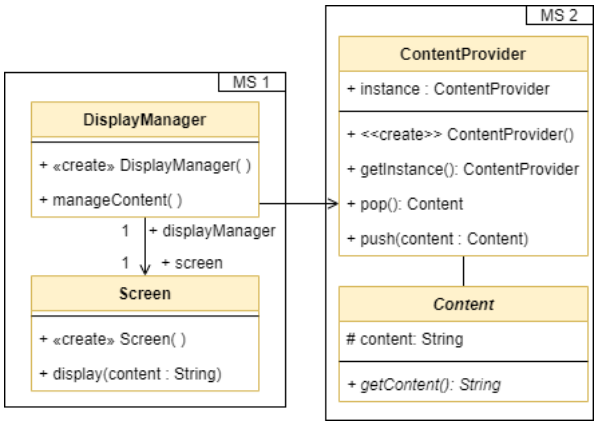

Figure 4.5: *ContentProvider's method pop() invoked by DisplayManager.*

```
public class DisplayManager{
    private ContentProvider contentProvider;
    public DisplayManager(){}
    public void manageContent(){
        ...
        contentProvider.pop();
        ...
    }
}
public class ContentProvider {
    private Stack<Content> contents;
    public ContentProvider(){...}
    public ContentProvider getInstance(){return self;}
    public Content pop(){
        return contents.pop();
    }
```

```
17      }
```

Listing 4.6: *An example of a method invocation between DisplayManager and ContentProvider.*

To remove this encapsulation violation, the set of methods from the invoked class (i.e., *ContentProvider*) are extracted into a set of required and provided interfaces that are placed in MS1 and MS2 respectively (see Figure 4.6). Then, the outgoing method calls from the invoker (e.g.,*DisplayManager*) are refactored to invoke the required interface (i.e., *IContentProvider*). This transformation decouples the two classes while providing a set of required and provided interfaces that will be used to establish for future communication.

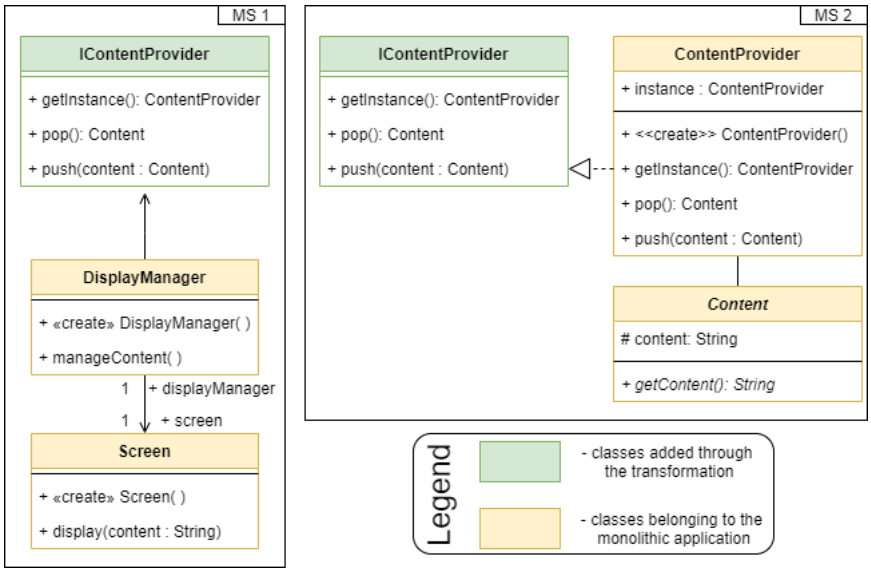

Figure 4.6: *Decoupling method invocations between MS1 & MS2 through interface-based calls.*

Nevertheless, after encapsulating the microservice, the invoked class (e.g., *ContentProvider*) cannot be reached by the invoker (e.g., *DisplayManager*) via the required interface. Indeed, as microservice communicate exclusively through lightweight mechanism (e.g., RPC or messages), a technological layer must be implemented. Therefore, the provided interfaces must be implemented, or exposed, as a web service in the microservice containing the invoked class (e.g., *ContentProvider*). In Figure 4.7, a *WebService* class is generated to expose the methods of *ContentProvider*. To achieve this goal, a method is created in the *WebService* for every public method of *ContentProvider*, to act as a proxy to receive a request. The proxy method then calls the appropriate method and returns its result. From the invoking microservice, a *WebConsumer* class is generated to implement the required interface and handle the network calls to its corresponding *WebService class*.

The transformation pattern for refactoring the set of required/provided interfaces into a web service is the only pattern proposed in this chapter that completely resolves an OO-type dependency into an MS-type one. Particularly, we differentiate transformation patterns that resolve encapsulation pattern with those that reduce them. Indeed, in the following sections we propose transformation patterns that reduce (and not resolve) a violation into another violation. In Section 4.5.6, we will discuss how by chaining transformation patterns we are able to transform all OO-type dependencies

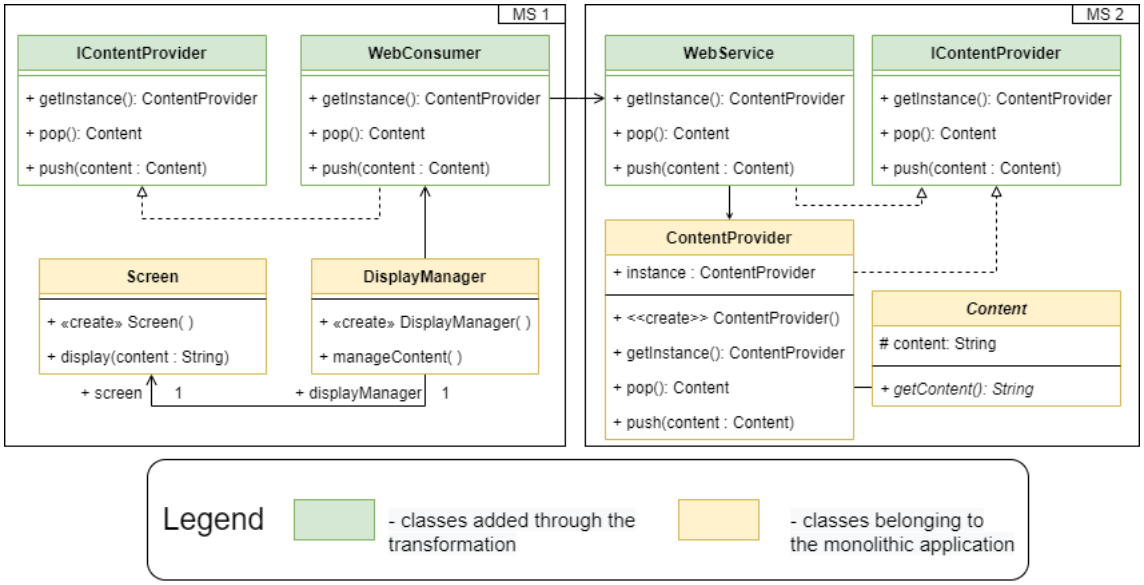

Figure 4.7: *Implementing the provided/required interfaces as a set of web service & web consumer.*

into MS-type ones. For now, let's accept that the following violation reductions are sufficient towards resolving them.

4.5.2 Attribute access

The attribute access violation (**Rule 2**) involves a cluster accessing the attribute of a class belonging to another cluster. This violation can be reduced by creating getter/setter methods, and limiting the access of the attribute to its class. Finally, the existing accesses within the internal code are refactored into the appropriate method invocation.

```
1  public class Clock extends Content{
2      public TimeZone timezone;
3
4      public Clock() {}
5      public String getContent() {
6          return "Current time is " + timezone.time;
7      }
8  }
9  public class TimeZone {
10   public int time;
11   public GpsLocation gpsLocation;
12
13   public TimeZone() {}
14   public TimeZone(int cod) {...}
15 }
```

Listing 4.7: *An example of an attribute access between Clock and TimeZone.*

In Listing 4.7, *Clock* directly accesses the attribute `time` of the class *TimeZone*. To remove this particular violation, the attribute `time` is made private (see Listing 4.8). Then, the method `getTime()` is created which accesses and returns the value of time. Finally, the method `getContent()` from the class *Clock* is modified to invoke the method `getTime()` instead.

```
1   public class Clock extends Content{
2       public TimeZone timezone;
3
4       public Clock() {}
5       public String getContent() {
6           return "Current time is " + timezone.getTime();
7       }
8   }
9   public class TimeZone {
10      private int time;
11      public GpsLocation gpsLocation;
12
13      public TimeZone() {}
14      public TimeZone(int cod) {...}
15      public int getTime(){
16          return time;
17      }
18  }
```

Listing 4.8: *Internal code refactoring to reduce the access attribute to a method invocation violation.*

However, this refactoring pattern leads to the creation of method invocation violations, which must also be resolved. For this, we can apply the method invocation transformation pattern presented in the previous subsection to completely resolve this violation.

4.5.3 Instance creation, handling, and sharing

We have addressed how an object's methods are invoked and attribute are accessed, however we have not addressed how classes can be instanced, handled and share across multiple microservices. Indeed, when we move from a centralized to a decentralized application, objects cannot be easily passed. In this subsection, we address the questions surrounding the creation and sharing of instances of a class between multiple microservices and how to recreate them in an MSA (**Rule 3**).

4.5.3.1 Instance creation

The first challenge involves recreating the instance creation that happens when a constructor method (e.g., *ContentProvider*'s) is called by a class (e.g., *DisplayManager*) belonging to another cluster. To resolve this violation, we must decouple the constructor call. More specifically, we apply a Factory pattern to decouple the creation of instances between classes belonging to different clusters. By applying a factory pattern, we create a class containing a method that is responsible for creating and returning the expected object.

For instance, we replace the instantiation of *ContentProvider* by the class *DisplayManager* with a class *Factory* acting as an object factory. For simplicity and ease of use, the same provided/required interfaces used to decouple method invocations between microservices also define these object factory methods. In Figure 4.8, this requires

adding a factory method in the required/provided interface (i.e., method *createContentProvider()*), and implementing the corresponding methods in the *WebConsumer* & *WebService*.

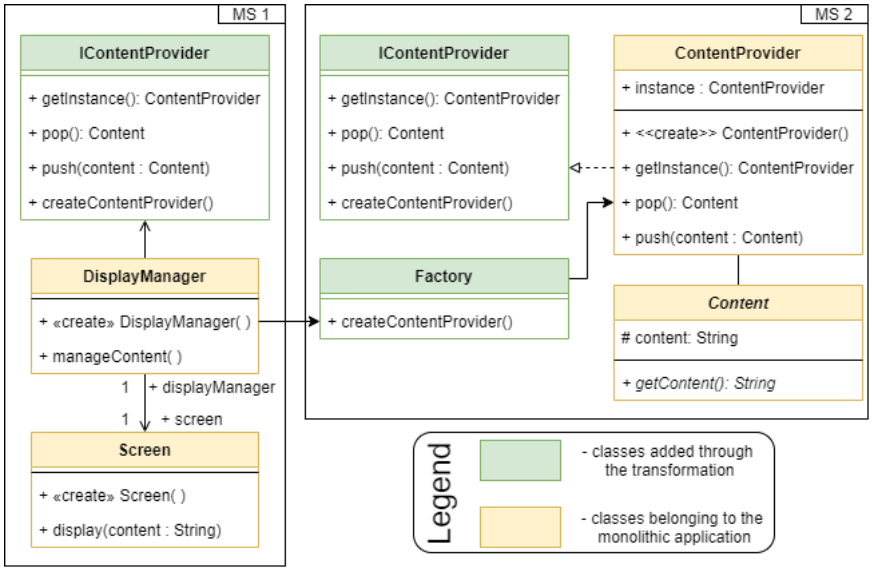

Figure 4.8: *Decoupling class instantiation with the Factory Pattern.*

In essence, this allows microservices to create objects through a web service. After the constructor is called by the invoker, there needs to be a way for the invoker to handle the object upon further use. However, since each microservice runs on its own process [LF14b], it is not easy to pass an object from one process to another. Therefore, the mechanism of passing objects through a method needs to be reproduced in the context of two microservices communicating.

4.5.3.2 Instance handling

Since objects cannot be easily passed between two microservices, there needs to be a way for microservices to manipulate classes defined and instanced within other microservices. To reproduce this mechanism, we propose a transformation pattern based on the Proxy Pattern. Generally, the intent of the proxy pattern is to provide a surrogate or placeholder for another object to control access to it [GHJV95].

Figure 4.9 illustrates the proxy pattern applied on the class *ContentProvider* to propose a surrogate (*ContentProviderProxy*) and handle all method invocations from *DisplayManager*. In this scenario, the proxy class acts to decouple the object referenced in one microservice (i.e., MS1) which is defined in another microservice (i.e., MS2). Therefore, a proxy class is created for any class referenced in one microservice and defined in another. This proxy class will have the same public methods and the same public constructors. However, the proxy class implementation is rewritten to use the *WebConsumer* class to interact with the real class definition.

Furthermore, upon the instantiation of the proxy class, the real class' instance is created. To differentiate, between the proxy class and the real class, the instances of the proxy class are called proxy instances, and instances of the real class are called

concrete instances. However, after instantiating a proxy instance, there needs to be a mechanism to link the proxy instance to the concrete instance. Indeed, a proxy instance should reference its concrete instance. Therefore, we propose that a proxy instance references its concrete instance via the same unique reference, and any operation on a proxy instance is transferred to its concrete instance. Finally, whenever the concrete instance is exchanged between microservices, the unique reference is passed instead of the concrete instance.

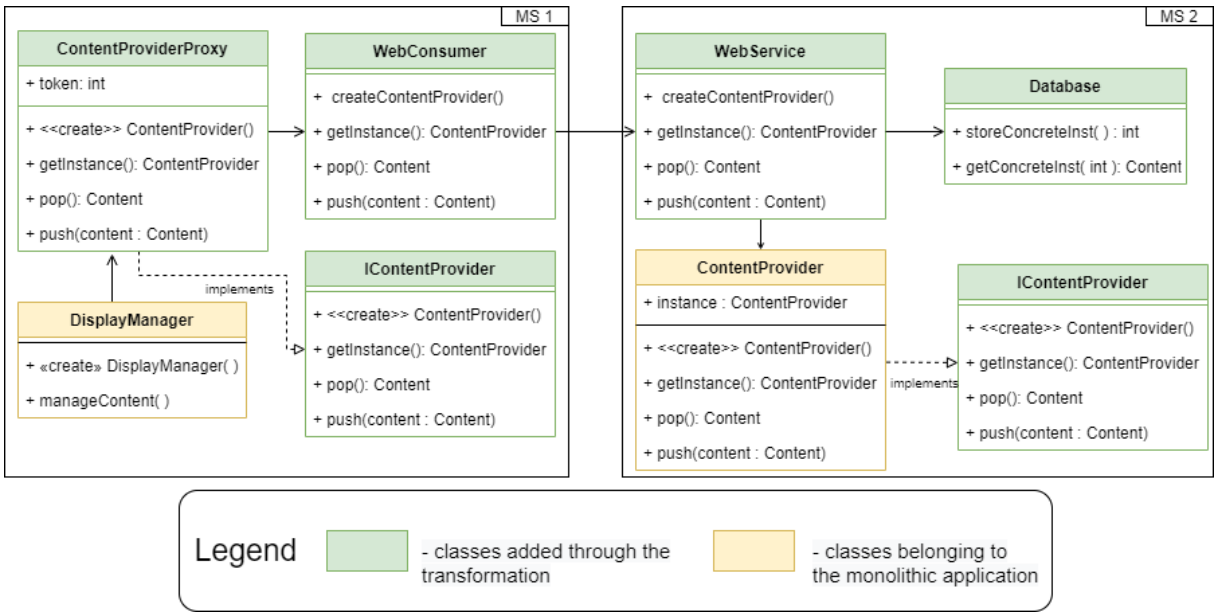

Figure 4.9: *Replacing access to an object with the Proxy Pattern.*

Concretely, there needs to be a mechanism to keep the state of the concrete instance between methods calls. Indeed, in the current implementation, shortly after the instantiation, a garbage collector can destroy the concrete instance. Therefore, we implement a class to store and manage all concrete instances created in a microservice. Whenever a factory is called to create a concrete instance, it sends the object to a storage class (e.g., the *Database* class in Figure 4.9) to preserve it. In turn, the storage class returns a token for accessing the object at a later date. The factory method returns the token via its web service implementation to the proxy instance which stores it for later method invocations. Later, when a proxy-instance method is invoked, it transfers the request along with its token to the appropriate web service method. The token adds the required context for the web service to load the concrete instance and invoke the correct method. From there, the result of the method invocation is returned by the web service.

4.5.3.3 Instance sharing

The final aspect to consider, when transforming the instance handling violation, is that complex objects may be passed as a parameter. However, as we shift from an OO paradigm to a microservice-oriented one, only certain serializable objects can be passed from one microservice to another. Indeed, while primitives or data classes can be easily serialized, certain objects and their states cannot be serialized without losing information. As microservice-type communication is limited to passing simple data

types such as strings and integers, we need to transform this type of exchange between microservices to be able to preserve the consistency of the application's business logic.

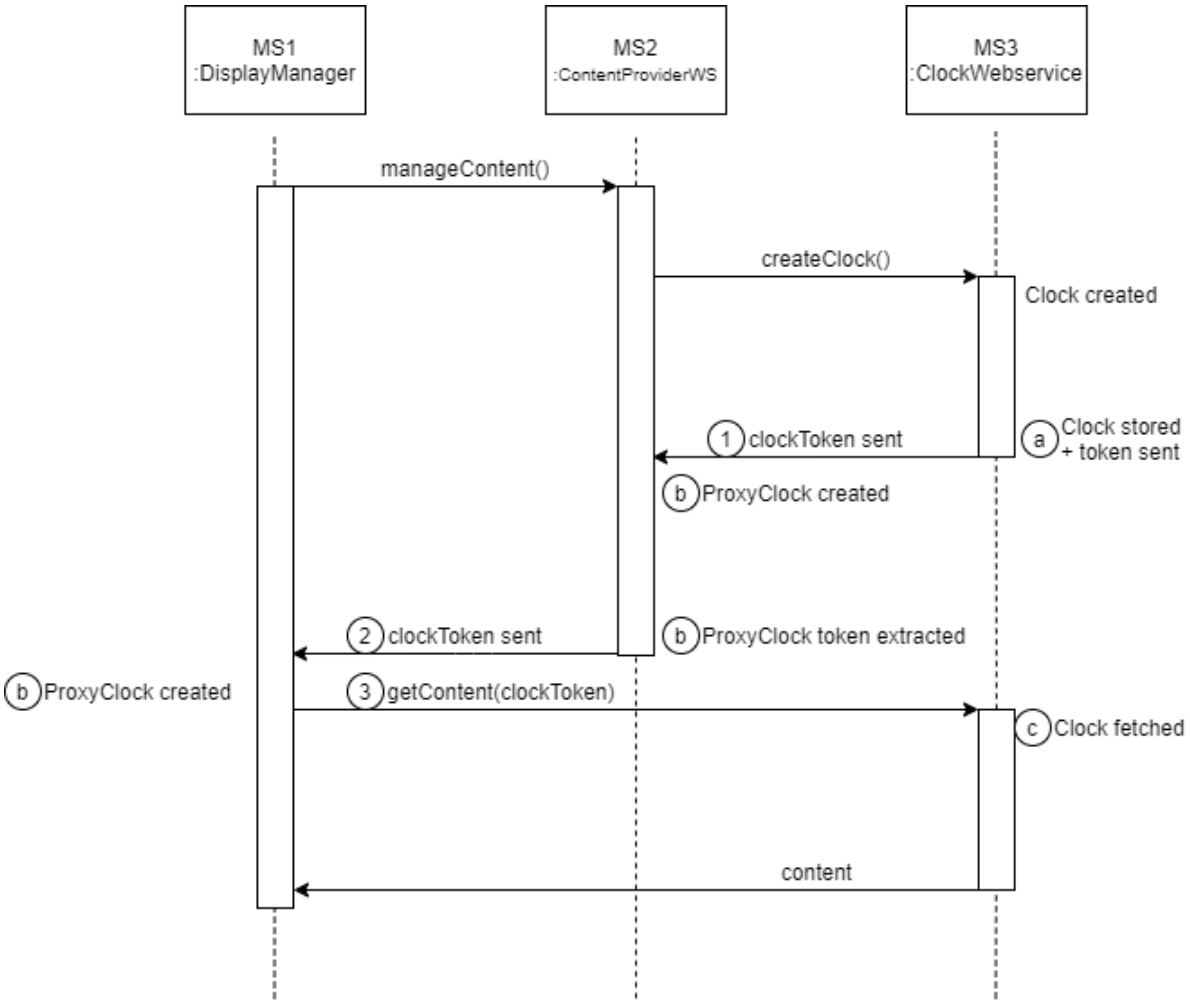

Figure 4.10: *Sequence diagram of instance sharing between three microservices.*

For instance, a microservice may receive or send an object of a class which it does not define. Furthermore, Whether the sender holds a proxy or a concrete instance, it must be able to produce a token to represent it. In the case of the receiver, it must be able to handle a token whether the receiver holds the concrete instance or not. With this token mechanism, complex objects can be passed between microservices as tokens, while the owner of the class manages the instances. A microservice is able to instantiate the proxy instance whenever it receives a token. When a microservice receives a token, it is able to create the appropriate proxy instance to access the concrete instance.

To illustrate the possible configurations the implementation handles, we propose an example based on the *Information Screen* application. In such application, the *DisplayManager* is able to invoke the `manageContent()` method of *ContentProvider*. In turn, *ContentProvider* creates a *Clock* instance and returns it to the *DisplayManager*. From there, the *DisplayManager* fetches the content of the *Clock* instance through its method `getContent()`.

However, during the migration *DisplayManager*, *ContentProvider*, and *Clock* are placed in different microservices. Furthermore, the *Clock* instance is passed between MS1 and

MS2, while being defined in MS3. Using the token system, several scenarios need to be handled (for each scenario a number indicates where in Figure 4.10 this scenario happens):

1. A microservice may receive an object of a class that does not belong to it (1 & 2).
2. A microservice may send an object of a class that belongs to it (1).
3. A microservice may receive an object of a class that belongs to it (3).
4. A microservice may send an object of a class that does not belong to it (2 & 3).

For each scenario, a token is passed between the microservices, but depending on the scenario the token is handled differently (each scenario is labeled in Figure 4.10):

1. When a microservice receives an object of a class that does not belong to it, the microservice creates the relevant proxy and stores the token within it (a).
2. When a microservice sends an object of a class that does not belong to it, the microservice must extract the token from its proxy and send it (b).
3. When a microservice receives an object of a class that belongs to it, the microservice uses the token to fetch it from its storage class (c).
4. When a microservice sends an object of a class that belongs to it, it must store the object and send the token (d).

In conclusion, this approach enables the passing of complex objects between microservices, while the owner of the class handles the instances for other microservices. Specifically, if a microservice does not define a class that is handles, then it uses the appropriate proxy class and the token mechanism to interact with the concrete instance.

4.5.4 Inheritance Relationship

Whenever a class inherits from another class belonging to a different microservice, it is considered an inheritance violation. To heal this encapsulation violation, inheritance must be decomposed into its different mechanisms and then transformed as to preserve all the mechanisms. This includes (a) the extension of the child class definition through the parent class, (b) the method overriding mechanism, and (c) polymorphic assignment. To do so, we propose a three-step transformation inspired from [AST+15]: (i) Uncoupling the child/parent inheritance with a double proxy pattern, (ii) Recreating method overriding via proxy inheritance, and (iii) recreating the polymorphic assignment through interface inheritance. We apply this three-step transformation on the inheritance between *Content* and *Message* presented in Figure 4.11.

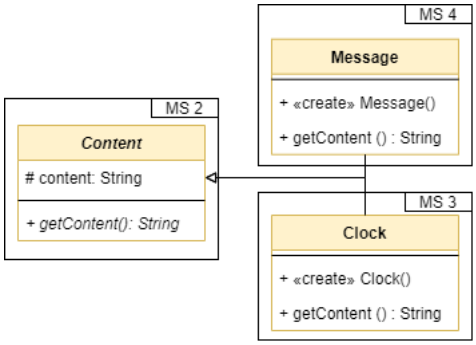

Figure 4.11: *Inheritance relationship between Content and the child classes Message and Clock.*

4.5.4.1 Child/Parent definition extension

The first mechanism that must be transformed is the extension of the definition of the parent by the child. A child class has access to the parent's attributes and methods. Furthermore, it may override the parent's methods. Finally, both child and parent method definitions may access each other's methods through the use of reference variables to the parent object or itself.

In their paper, the authors propose a double delegate pattern to preserve the inheritance between class placed in different components [AST+15]. As part of their solution, whenever a child object is created, a parent object is also created as an attribute within the child object. Furthermore, the child class is refactored to implement any parent method that is not redefined in the child class. These redefined methods delegate any invocation to the parent method through the stored parent object to recreate the access to the parent class methods. Inversely, the parent object store the child object and acts as a delegate to preserve the dynamic calling of overridden methods. This transforms the inheritance encapsulation violation into a set of instance creation violations and a method invocation violations which can be healed using the transformation patterns proposed previously. In the case of an abstract parent class, [AST+15] also apply a proxy pattern so that the proxy class inherits from the parent class, and it can be instantiated by the child class.

This transformation pattern requires refactoring the internal code to add attributes and modify the existing methods of both the parent and the child classes. All this internal code refactoring requires informing the developer to use the delegate pattern instead of the native inheritance implementation which can reduce readability and thus increase the maintenance cost. Instead, we proposed a revised version that treats inheritance as a service and which also minimizes the refactoring of the internal code.

Concretely, we propose a double-proxy pattern to reproduce the inheritance link between the child and parent classes without significantly refactoring the child/pattern classes. Figure 4.12 illustrates the transformation of the inheritance link between the child (e.g., *Message*) and the parent class (e.g., *Content*). First, a parent proxy class (e.g., *ContentProxy*) is created to implement the methods defined by the interface extracted from the parent class (e.g., *IContent*). Then, the child class (e.g., *Message*) is refactored to extend the parent proxy (e.g., *ContentProxy*). Finally, child proxy class (e.g., *MessageProxy*) is defined to extend the parent class, and acts as the child proxy for the

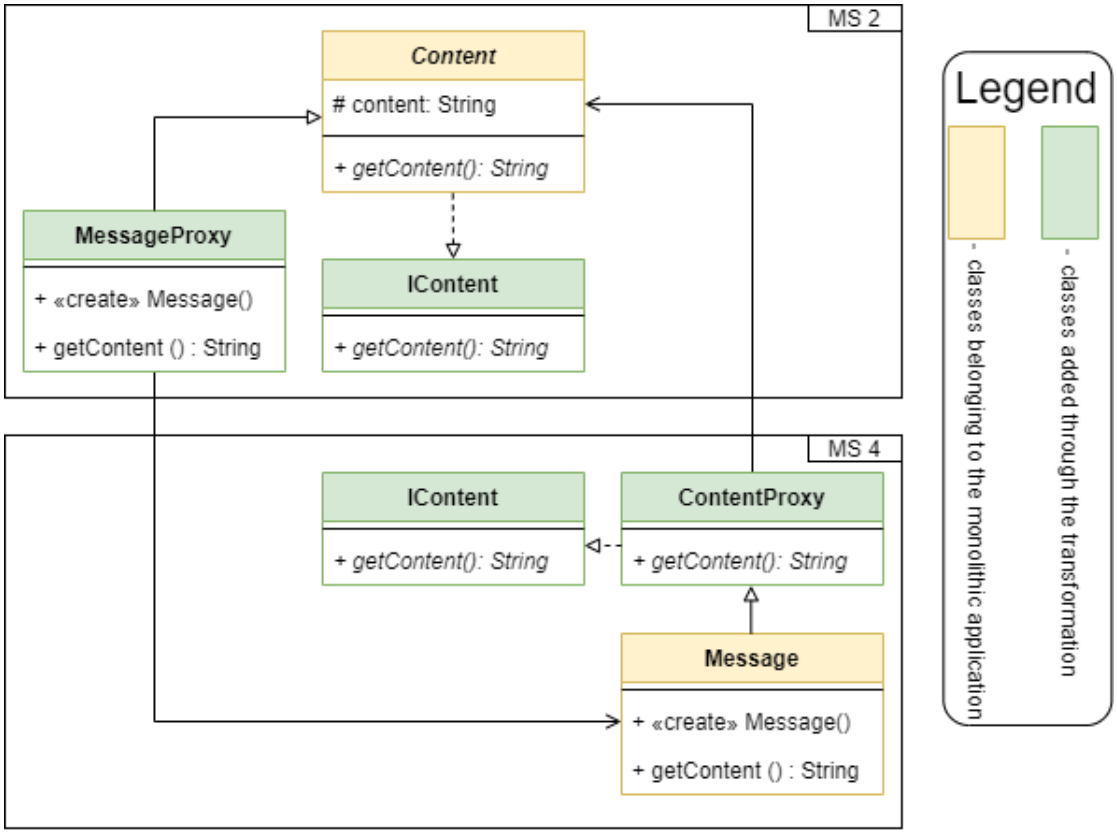

Figure 4.12: *Redefining the inheritance between Content and Message through a double-proxy pattern.*

parent class. In total, the refactoring of the internal code is limited to modifying which class the child class extends (e.g., in our case *Message* now extends *ContentProxy*). Furthermore, this transformation pattern can also be applied in the context of an abstract parent class.

4.5.4.2 Recreating Method Overriding

Method overriding is an OO mechanism that allow a child class to rewrite the implementation of a method defined in its parent class. Concretely, the version of the method that is executed is determined by the object that is used to invoke it. This mechanism is recreated through the double-proxy pattern, by overriding the same methods of the parent class with the proxy child. In turn, the overriding method calls the appropriate method of the child class. However, this creates a set of instance creation & method invocation violations that needs to be transformed. This is also the case regarding the relationship from the child class to the parent class.

To do so, we add two web services (see Figure 4.13). Upon the creation of a child object (e.g., *Message*), the parent proxy's constructor (e.g., *ContentProxy*) is called to consume the Parent web service. This has the effect of initializing the child proxy (e.g., *MessageProxy*) that inherits naturally from the parent class (e.g., *Content*). Whenever a method defined by the parent class (e.g., *Content*) is invoked by the child object (e.g., *Message*), the parent object will be invoked via the parent web service. Furthermore,

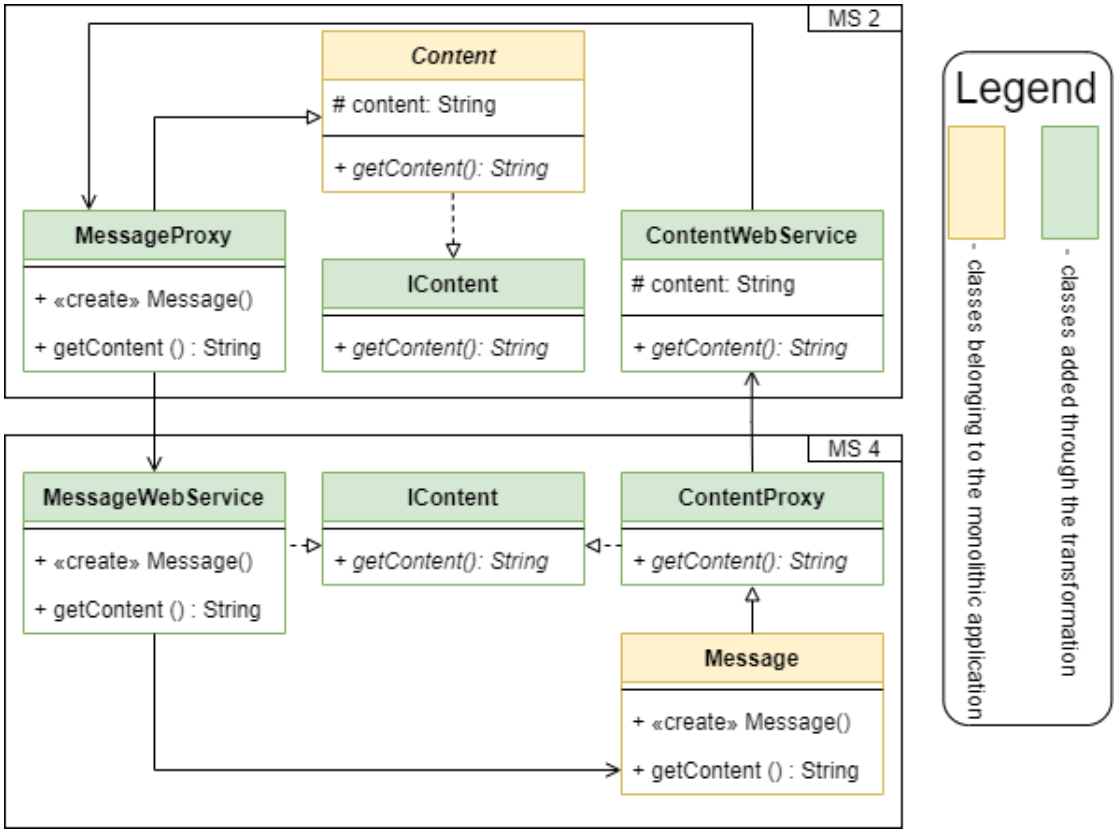

Figure 4.13: *Inheritance as a service.*

when the parent class (e.g., *Content*) references the instance, it will invoke the child object through the child proxy (e.g., *MessageProxy*) object.

4.5.4.3 Recreating Polymorphic Assignment through Interface Inheritance

In a language supporting polymorphic assignment, a variable of a parent class can be assigned an instance of a subclass type. However, as we transform the inheritance we replace the assignment compatibility (i.e., subtyping). To recreate the polymorphic mechanism, a child interface (e.g., *IMessage*) is defined to extend the parent interface (e.g. *IContent*). The child class implements the child interface, allowing for the polymorphic assignment of the child objects (see Figure 4.14).

4.5.5 Resolving Exception Throwing & Catching Violations

The exception handling encapsulation violation involves create, throwing, and catching exception objects across microservices. To ensure a well-contained throwing & handling of exceptions we propose a two-step transformation process: wrapping the exception response and transforming the exception-handling source code.

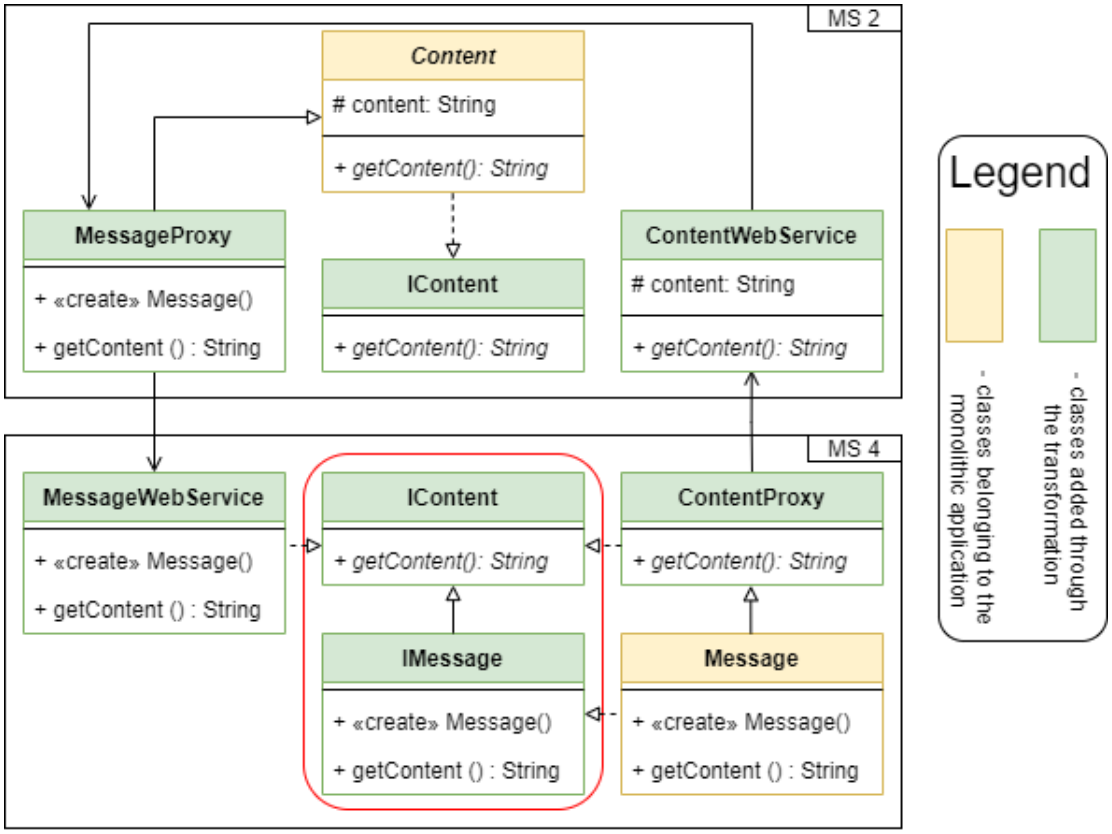

Figure 4.14: *Polymorphic assignment can be recreated by applying an interface inheritance between the Parent Interface and the Child Interface.*

4.5.5.1 Wrapping the exception response

Normal methods have two different types of responses. They may return the normal intended type response, or an exception response. However, Web service methods are not intended to throw exception objects. When a method is exposed as a service, this limitation must be circumvented by introducing a class that acts as a wrapper return type which can hold either the normal response type, or an exception response type. Every method's return type is replaced by this wrapper class.

4.5.5.2 Transforming the exception-handling source code

To prepare the wrapper type, a web service operation surrounds the method invocation with a try and catch. When the method returns the normal response type, it safely adds the value in a dictionary. When the method returns an exception response type, it safely captures the exception object, stores it for later use, and adds its corresponding access token to access the dictionary. Listing 4.9 illustrates an example of a web service method wrapping the normal *IContent* response type, or catching either an *EmptyContentStackException* or a *FullContentStackException* object. Either way, the object is stored, and its token is placed in a JSON node and returned.

```
1 public class ContentProviderWebService {
2     public JsonNode pop(int proxy_id){
3         JsonNode return_node = new JsonNode();
```

```
4           IContentProvider contentprovider = InstanceDB.getContentProvider
    (proxy_id);
5           try{
6               return_node.put("return", InstanceDB.addContent(
    contentprovider.pop()));
7           } catch(EmptyContentStackException e){
8               return_node.put("EmptyContentStackException", InstanceDB.
    addEmptyContentStackException(e));
9           }
10          return return_node
11      }
12  }
```

Listing 4.9: *Surrounding the method which throws an error with a try and catch.*

Upon receiving the response from the service, the proxy must check the response with a series of if/else. If the wrapper contains the normal response then it returns it. If, on the other hand, it contains one of the exception responses, then it extracts the token corresponding to the exception response, associates it with a new proxy exception object, and finally throws the latter. Listing 4.10 illustrates how the JSON sent in Listing 4.9 is handled. If the JSON contains a value designating any of the keys that correspond to an exception type, then the corresponding exception proxy is created. Otherwise, it is assumed that the normal response was stored in the return key of the JSON.

```
1  public class ContentProviderProxy {
2      public IContent pop() throws EmptyContentStackExceptionImpl {
3          JsonNode return_node = getProxy().pop(contentprovider_id);
4          if(return_node.get("EmptyContentStackException" != null){
5              throw new EmptyContentStackExceptionImpl(return_node.get("
    EmptyContentStackException"));
6          } else {
7              return new IContentImpl(return_node.get("return").asInt());
8          }
9      }
10 }
```

Listing 4.10: *Surrounding the network call with an if/else statement to unwrap either the normal response or the exception response.*

4.5.6 Violation Resolution Order

For every type of encapsulation violation identified in this approach, transformation rules have been proposed. However, some transformation rules produced additional violations. Such is the case with the inheritance violation which creates additional instance creation & method invocation violations. Therefore, to systematically resolve all encapsulation violations in one iteration, we propose a violation resolution order which is presented in Figure 4.15.

The order is as follows:

1. The attribute access violation is reduced as it adds public methods to its class that may be further refactored by inheritance violation.

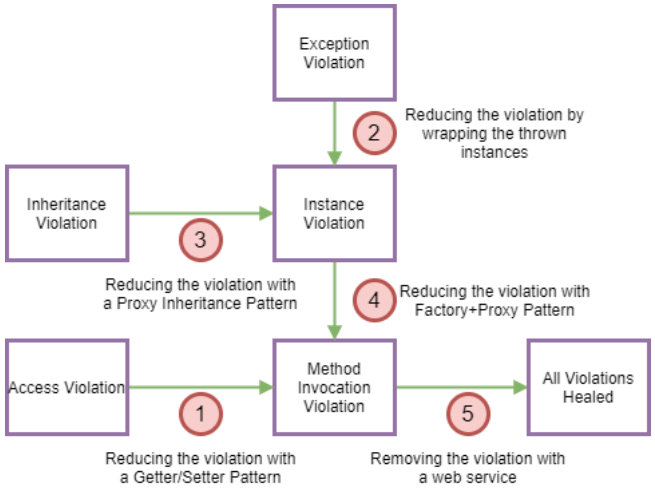

Figure 4.15: *The transformation order of each microservice encapsulation violation.*

2. The thrown exceptions are reduced to instance violations.
3. The inheritance violations are reduced to an instance violation, so all instance violations can be healed together.
4. The instance violations are reduced to method invocation violations.
5. The remaining method invocations violations are transformed into a set of web services.

Once all violations have been resolved, we are left with a set of web services in each microservice candidates. Finally, the source code of each microservice candidate can be generated and packaged into their own project. In the next section, we discuss the implementation of our approach to evaluate it on a set of applications.

4.6 Evaluation

The main goal of this evaluation is to determine whether our approach is able to properly encapsulate microservice candidates from the identified MSA. To evaluate our approach we implemented a tool to apply our approach. Furthermore, we migrated to various degrees a set of monolithic applications of various sizes.

4.6.1 Data Pre-processing: Microservice Identification

For this experiment, we extracted an initial list of 19 monolithic applications that were used in nine different articles in the field of MSA recovery. From this initial list, we selected 6 applications based on whether the source code was open-source and object-oriented. The seventh application (*Omaje*), is a closed-source legacy application by Berger-Levrault, an international software editor. This application was designed over 10 years ago by a team of 4 developers to handle the distribution of software licenses

between Berger-Levrault and its clients. Metrics on the seven applications are available in Table 4.1.

Table 4.1: *Applications on which the experiment was conducted.*

Application name	No of classes	Lines of Code (LOC)
FindSportMates[2]	21	4.061
JPetStore[3]	24	4.319
PetClinic[4]	44	2.691
SpringBlog[5]	87	4.369
IMS[6]	94	13.423
JForum[7]	373	60.919
Omaje	1.821	137.420

Furthermore, we must recover the microservice architecture to evaluate the transformation approach. To do so, we use the semi-automatic approach proposed in [SSB+20b] to recover an MSA as a cluster of classes, but other identification approaches can be used. These clusters along with the source code of the applications are used as input for our approach.

4.6.2 Research Questions and their Methodologies

We conduct an experiment with the goal of answering the following research questions regarding our approach.

4.6.2.1 RQ1: Is our approach able to preserve the behavior of the migrated application when materializing its microservice-oriented architecture?

Goal The goal of this research question is to evaluate the functional/behavioral correctness of the microservice architecture. In other words, we aim to demonstrate that we are able to transform the source code of a monolithic application while preserving its business logic.

Method To answer this RQ, we want measure the precision and recall of our approach based on the syntactic and semantic correctness of the transformed microservices. It stands to reason that if the resulting MSA applications behaves in the same way as the monolithic applications then the business logic was preserved.

[2] https://github.com/chihweil5/FindSportMates
[3] https://github.com/mybatis/jpetstore-6
[4] https://github.com/spring-petclinic/spring-framework-petclinic
[5] https://github.com/Raysmond/SpringBlog
[6] https://github.com/gtiwari333/java-inventory-management-system-swing-hibernate-nepal
[7] https://github.com/rafaelsteil/jforum2/

We consider that microservices have a correct syntax if there is no compilation errors. To measure the semantic correctness, we rely on whether the transformed microservices produce the same results compared to the functionalities of the original the monolithic applications at run-time. To do so, we identify a set of execution scenarios that can be used in both applications. We compare the outputs of the monolithic application with its microservice counterpart for each execution scenario. We consider that the transformation has a semantic correctness when the outputs generated by the monolith and the MSA are identical based on the same inputs.

When possible, the identification of execution scenarios is based on test cases defined by the developers of the monolithic applications (e.g. JPetStore). When test cases are not available, we identify a set of features and sub-features for each monolithic application (e.g. FindSportmates, IMS). From these features, we establish a set of user scenarios that cover all features of each application. These user scenarios are performed on the monolithic application and the results are saved. Then, these user scenarios are performed on the MSA, and the results are compared with those of the monolithic application.

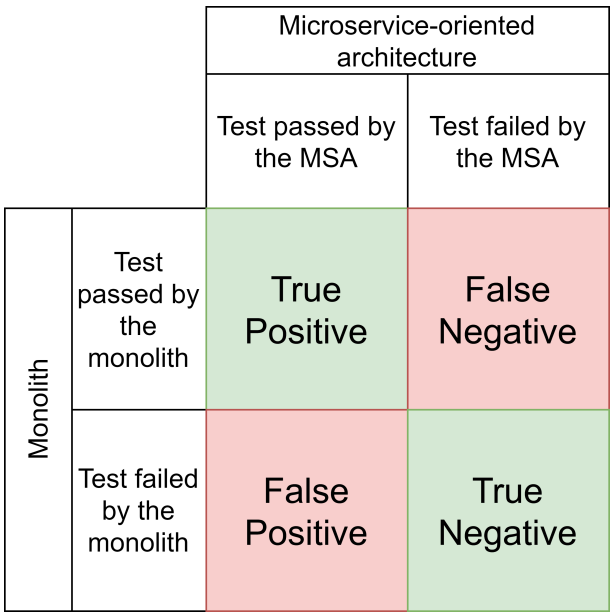

Figure 4.16: *The confusion matrix for evaluating the materialization of the MSA in comparison with the monolith.*

To calculate the precision of the materialization approach, we use the confusion matrix presented in Figure 4.16. Particularly, the precision is calculated by taking the number of tests passed by both architectures and dividing by the number of the tests passed by the MSA. Furthermore we calculate the recall by taking the number of tests passed by both architectures and dividing it by the number of tests passed by the monolith.

Due to time constraints related to the application packaging that is highly dependent on the technology of the monolith working with Spring, we study this research question with the *FindSportMates*, *JPetStore*, and *InventoryManagementSystem* applications. For *JPetStore*, we ran the Selenium tests provided with the monolithic application. For *FindSportmates* and *Inventory Management System*, we manually ran these user scenarios.

4.6.2.2 RQ2: What are the impacts of Mono2Micro on the performance?

Goal The overall goal of our approach is to migrate while preserving the semantic behavior of an application. Moreover, an important aspect of the migration is that it must preserve the semantic without degrading drastically the runtime performance of the application. Therefore, the primary goal of this RQ is to evaluate whether the performance impacts resulting from the migration of the monolithic application to microservices are negligible when compared to the original application.

Method To answer RQ2, we rely on the execution time of user requests. The execution time measures the delay between the time when the request is sent and the time when the response is received by the user. We compare the execution time of both the monolith and the MSA.

We establish a user scenario using *Omaje* to compare the performance of the monolithic application with its microservice counterpart. For this evaluation, we chose *Omaje* because its business logic is the most complex of all 7 applications. To evaluate the performance, we simulate an increasing number of users connecting to both the MSA and the monolith, using JMeter[8] to simulate user load. As the number of user increases, we increase the number of instances of the microservice for both the monolith and the MSA. For the monolith, this involves duplicating the application. For the MSA, this involves duplicating the microservices involved in the current scenario. We consider that the refactoring results improve or maintain the quality and performance of the original code if the execution time difference between both architectures is negligible for the average user while the resource utilization is optimized. For our test we use a computer with an i7-6500U @ 2.5GHz and 16 GB of ram.

4.6.3 Results

Table 4.2: *Data on the applications being transformed.*

Application	No. MSs	No. data classes	No. violations
Findsportmates	3	2	9
JPetStore	4	9	21
PetClinic	3	7	26
SpringBlog	4	8	104
IMS	5	18	113
JForum	8	37	1031

[8] https://jmeter.apache.org/

Table 4.3: *Type of violations caused by OO-type dependencies between microservices.*

Application	No. Instances	No. Inheritances	No. Exceptions
Findsportmates	9	0	0
JPetStore	20	0	0
PetClinic	24	2	0
SpringBlog	95	7	2
IMS	110	3	0
JForum	1013	16	2

Table 4.4: *Number of tests performed for each application and the resulting precision and recall from these tests.*

Application	No. Test	Precision	Recall
Findsportmates	7	100%	100%
JPetStore	34	100%	100%
IMS	36	100%	100%

4.6.3.1 RQ1: Is our approach able to preserve the behavior of the migrated application when materializing its microservice-oriented architecture?

Table 4.4 shows the results of **RQ1**. The results show that our approach has both 100% precision, and 100% recall for *FindSportMates*, *JPetStore* and *InventoryManagementSystem* in terms of syntactic and semantic correctness. Therefore, our approach is able to preserve the business logic with a high precision.

The proposed transformation did not create a side-effect that was detected by failed functional tests that otherwise passed for the monolith. Therefore, our approach is able to preserve the business logic with a high recall. However, it should be noted that for *JPetStore* the Selenide test "testOrder" failed for both the monolithic version and the MSA version, as both checked the pricing notation using a period as a decimal separator while the testing was performed on a computer which defaults to using a comma instead. Furthermore, while the results indicate a perfect precision/recall, it is important to remember that these tests evaluate the functional aspect of the applications and other types of tests (e.g., unit tests) should be performed to assure that no side-effect is introduced during the refactoring.

4.6.3.2 RQ2: What are the impacts of Mono2Micro on the performance?

Figure 4.17 illustrates the number of users per scenario with the different architecture configurations. We can see, there is a small gain in performance upon the introduction of scaling for the microservice-oriented architecture.

The proposed transformations from Mono2Micro does not negatively affect the per-

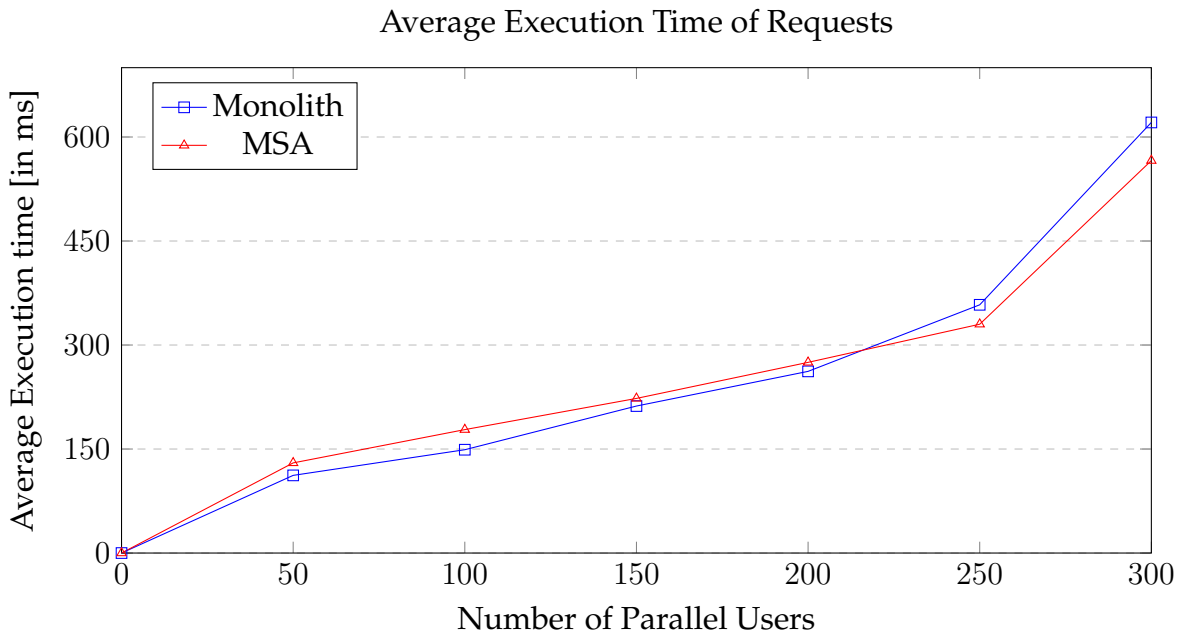

Figure 4.17: *Average execution time of Omaje based on the number of users and the corresponding architecture.*

formance of the application. Our expectations were that by introducing additional network calls the performance of the migrated application would be affected negatively. However, in this scenario it was not the case. This was likely due to the parallelization aspect of scaling the requested service. By adapting the number of instances of microservices, the MSA was able to handle the increased requests and compensate for the additional network layer. In fact, as the number of parallel requests increased, the MSA performed better (on average) compared to its monolith counterpart.

4.6.4 Threats to Validity

Our study may be concerned by internal and external threats to validity. We discuss below these two kinds of possible threats:

4.6.4.1 Internal Threat to Validity

The first threat to validity is that our transformation approach uses static analysis to detect and transform the existing source code. Indeed, static analysis cannot detect dynamic binding and polymorphism when identifying instance encapsulation violations. However, this can be avoided by taking into consideration the worst case by creating an instance dependency for every subtype. Another risk is that static analysis, unlike dynamic analysis, cannot detect unused source code. This results in detecting more dependencies than necessary. However, this can be mitigated in well-maintained applications. Another solution is to perform an hybrid analysis during the detection step. However, dynamic analysis requires instrumenting and providing a thorough set of test cases, which is not always available or feasible in a large industrial code-base.

Also, we consider our approach to be adequate for source code that is not reliant on a strong framework (e.g. Spring for JAVA). We do not consider, dependency injection which is one of the properties of this type of framework. Finally, our approach does not consider the reflexivity of certain languages, thus in our experiment we identified and manually resolved these types of encapsulation violation.

4.6.4.2 External Threat to Validity

One external threat of validity we considered is the use of a specific architecture recovery approach (i.e., the approach presented in [SSB+20b]) to have an impact of the transformation phase. Indeed, the number of identified dependencies and the overall performance are highly dependent on the results of the architecture recovery phase. However, our goal was not to analyze the impact of our transformation on the produced architecture, but whether we are able to migrate applications while preserving the intended behavior (business-wise and performance-wise) of the application. Another threat we considered is that our monolithic applications are all implemented in JAVA. However, the obtained results can be generalized for any OO language. We argue, just as most architecture recovery approaches, that generalization is possible since all OO languages (e.g., C++, C#) are structured in terms of classes and their relationships are realized through the same general mechanisms (e.g. method invocations, field access, inheritance, etc.).

4.7 Conclusion

In this chapter, we focused on the materialization phase of the migration phase. We have established that after identifying a microservice-oriented architecture, there are several challenges towards materializing the MSA. Particularly, we have revealed encapsulating each microservice candidate into its own project exposes OO-type dependencies. This requires refactoring of the source code to create semantically-valid microservices. In turn, we have proposed a semi-automated approach towards transforming the existing OO source code to conform to the identified MSA. In the next chapter, we address the limitations of the ad hoc transformation approach by proposing a generic end-to-end migration approach.

V

Model-driven end-to-end migration approach

Contents

5.1 Introduction

In the previous chapter, we propose an ad-hoc approach to refactor object-oriented code. While the approach attempts to automate the refactoring of any OO language, its implementation is limited by the language of the application's source code, obstructing its genericity and reusability across other languages and technologies. We seek to overcome the limitations of the previous ad-hoc refactoring approach by adopting **Model-Driven Engineering** (**MDE**) techniques.

Indeed, MDE has already been adopted for software modernization [FBB+07], and has been recognized as an efficient, reliable, and flexible approach for software migration [Sch06]. An MDE-based approach enables the migration process to be independent of the application's source code by leveraging **Platform-Independent Models** (**PIMs**) which can be reused across different migration efforts to represent monoliths of different types. Once the generic model has been extracted from a monolith, the same transformation process can be applied to then generate a microservice-oriented architecture.

Furthermore, we wish to introduce an end-to-end approach that encompasses both phases of the migration process. To this end, we integrate the identification as part of the transformation process of the monolith. Concretely, the goal of this chapter is to present a generic and reusable end-to-end migration approach. First, we use MDE techniques to implement a generic and reusable approach. Second, we incorporate the identification and transformation phase together in all-in-one solution.

In the next section, we introduce the global workflow of our approach, and describe each of its steps. In Section 5.3, we present the metamodels used in our approach. In Section 5.4, we present the transformations rules between the different models. In Section 5.6, we apply our approach on a case study to evaluate its feasibility. Finally, we present our conclusions and perspectives in Section 5.7.

5.2 The Global Migration Workflow

In this section, we introduce a broad overview of the different phases which constitutes our global MDE-based migration workflow. In short, our workflow (see Figure Figure 5.1) encompasses four important steps:

1. Extracting a generic model from the source application.
2. Identifying the application's candidate MSA and incorporating it into an intermediary pivot model.
3. Transforming the pivot model to obtain the target MSA model, consisting of multiple microservices and their associated entities.
4. Generating and packaging the target source code from the target MSA model to make it deployable.

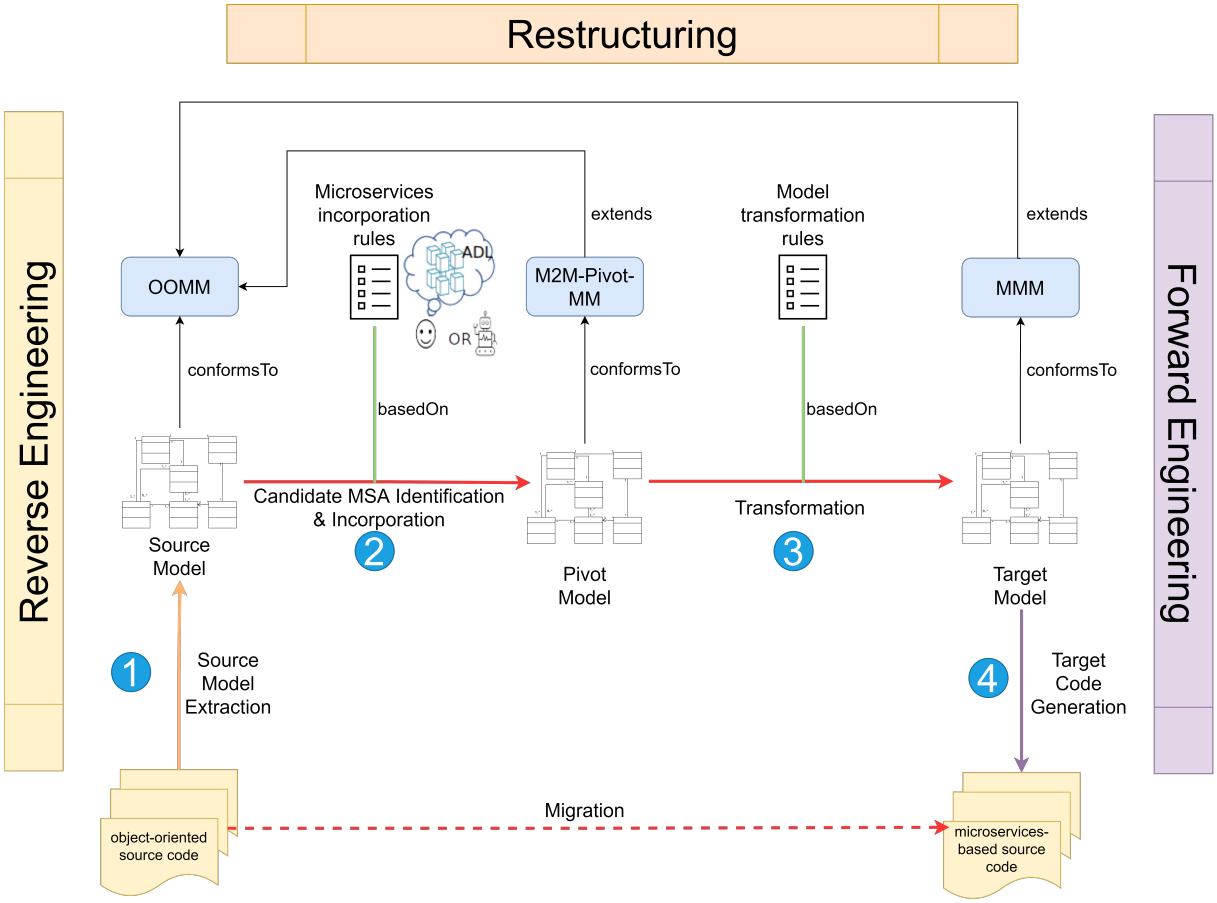

Figure 5.1: *The global MDE-based migration process with the models and metamodels used therein.*

This workflow follows the principles of the *Horseshoe Model* [KWC98] of extracting an architectural representation of the monolith to perform the transformation towards the desired architecture (i.e., MSA). We propose a workflow that is primarily PIM-oriented instead of being **Platform-Specific Model** (**PSM**)-oriented, as is the case with its ad-hoc counterparts. In other words, by relying on PIMs, we aim to increase the genericity of the overall approach.

In our approach, we use 3 different platform-independent metamodels: `OOMM`, `M2M-Pivot-MM`, and `MMM`. The `OOMM` metamodel serves to represent the monolith through a generic representation of its classes and their relationships. The `M2M-Pivot-MM` metamodel serves as an intermediary representation between the source and the target model, in which higher-level abstraction of the source model is stored. Finally, the `MMM` metal-model represents the target architecture and is used to generate its source code. More details on each metamodel can be found in Section 5.3.

Transitioning from one source model to another target model requires a set of model transformation rules, mapping one/many elements in the source to one/many elements in the target. These transformations occur at a domain-level, and are thus less constrained by their applications' technologies and platforms, rendering them consequently more reusable for object-oriented applications in general. Once the target model is complete, we can generate the corresponding code.

In the following subsections, we describe each major step of this workflow.

5.2.1 Model Extraction

The first phase in the MDE-based migration process consists of extracting a generic model (i.e., an OOMM model) to represent the monolith. Concretely, it consists in parsing the project's source code and extracting its model from its corresponding AST. Depending on the project's source code, a corresponding parser can be used to extract its AST. For instance, in the case of the implementation of our approach, we use VerveineJ[1] to extract the FAMIX model from JAVA source code. One advantage of the model-driven approach is that we can reuse existing parsers to extract the source model of a monolith.

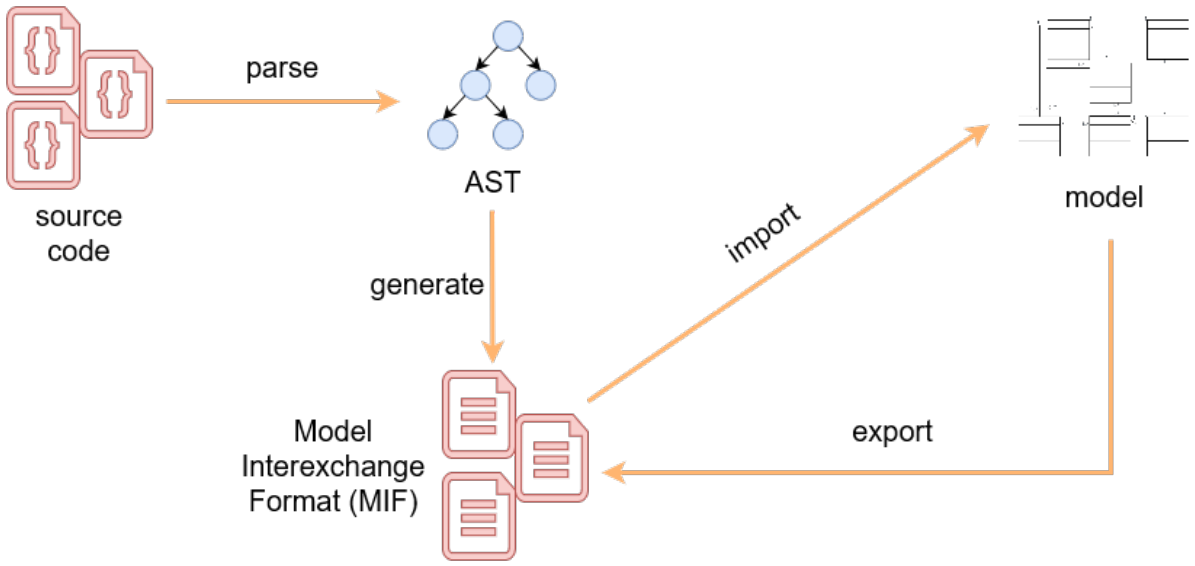

Figure 5.2: *Model extraction process.*

After the extraction of the model, it can be exported or imported to and from the disk based on a **MIF** (**Model Interexchange Format**), independently of its original project (see Figure 5.2). An MIF designates any model interexchange format used to represent abstract models, such as **Moose interexchange format** (**MSE**), a model interexchange format used to represent FAMIX models. Concretely, this allows for a greater flexibility when manipulating the models (e.g., adding versioning).

From the generic model we can more to the next step of identifying the microservice-oriented architecture, and incorporating it into a higher-level abstract representation of the monolith.

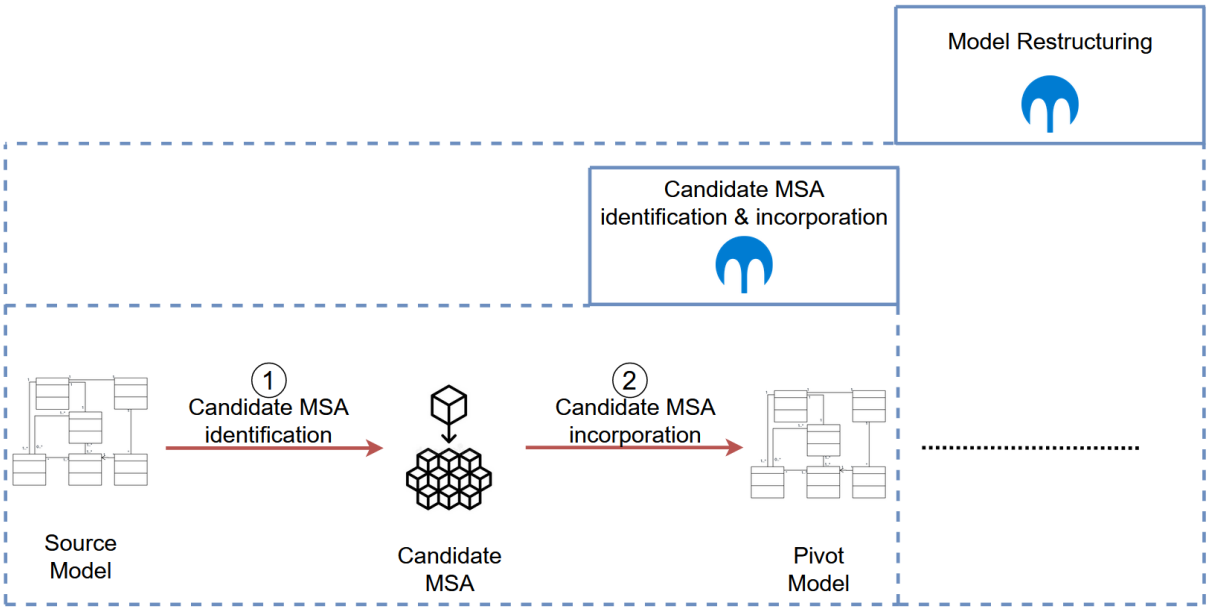

Figure 5.3: *The identification and incorporation step of the model-driven migration.*

5.2.2 Candidate MSA Identification & Incorporation

The second phase in the MDE-based migration process consists of extracting the MSA from the source model and incorporating it into a pivot model (see Figure 5.3). Concretely, we divide this phase into two steps: (1) the identification of a candidate MSA, and (2) its incorporation into a pivot model.

5.2.2.1 Candidate MSA Identification

Microservices identification is a software engineering task occurring at the architectural level of a monolith and aims to apply reverse engineering techniques on its software artifacts to identify the corresponding microservice candidates and their description within an MSA. The identification process is based on a set of patterns/strategies, constraints, and quality attributes shaping and guiding its course [WLM+21].

In the context of this chapter, we focus on the identification of microservices based on the source code of an object-oriented monolithic application. One common way to achieve this is a graph-based approach that employs graph clustering and visualization techniques to identify the candidate microservices from the monolith's source code [WLM+21]. In particular, we apply the approach proposed in Chapter 3 which extracts the layered architecture before partitioning its artifacts using a clustering algorithm to represent the candidate MSA as a set of LayerEntity clusters.

The initial step of reverse-engineering the layered architecture serves an important task towards the overall migration process. In Chapter 3, we saw its impact on the identification process. Furthermore, in the model-driven transformation it also serves the function of structuring the internal architecture of each identified microservice.

[1] https://github.com/moosetechnology/VerveineJ

Once the candidate MSA is obtained from the identification step, it will be forwarded to the software architect for validation. Accordingly, the architect can also interact with the candidate MSA to modify it. Otherwise, the identification process ends, and the candidate MSA can be incorporated into the new model.

5.2.2.2 Candidate MSA Incorporation

The MSA identification process only provides an incomplete framework describing the general outlines of the target MSA model in the form of a **candidate MSA**. Hence, to further complete this process, we need to describe the candidate MSA by a properly dedicated metamodel, followed by the application of the necessary transformations on its model to actually reflect the identified MSA description. We incorporate the candidate MSA yielded by the MSA identification process into an intermediary pivot metamodel, namely the **Monolithic-to-Microservices Pivot Metamodel** (**M2M-Pivot-MM**).

Moreover, our workflow deals with candidate MSAs extracted through identification processes that use graph-based clustering algorithms. This yields MSA descriptions with a specific interface that exposes the candidate MSAs' class clusters and their associated entities. This interface may vary from one identification approach to another. As a result, our incorporation rules are conceptually dependent on this interface.

Nevertheless, we aim to make our approach as reusable and generic as possible to reduce migration efforts. In other words, we wish to make these incorporation rules independent of the MSA description's interface. This proves necessary to implement them once and reuse them across different workflows employing different identification processes that yield MSA descriptions with different interfaces.

Consequently, the incorporation mechanism employs the `Adapter` design pattern. Particularly, a candidate MSA adapter maps the API of a candidate MSA to the API expected by the pivot metamodel (see Figure 5.4). In other words, the candidate MSA adapter enforces any identified candidate MSA to implement its expected interface, namely to provide means of access to the MSA, its candidate microservices, their business, data, utility classes, and their provided and required interfaces.

Once the candidate MSA implements the adapter's required interface, the adapter maps every entity retrieved from the candidate MSA to its corresponding entity in the pivot model.

5.2.3 Model Transformation

The third phase in the MDE-based migration process is the model transformation (*see Figure 5.5*). It consists of transforming the pivot model, described by the M2M-Pivot-MM, to obtain the target model, described by the **Microservices Architecture Model-Driven Migration Metamodel** (**MMM**). An important aspect of the model transformation is that we limit our transformation rules to be applied to the model itself. Since,

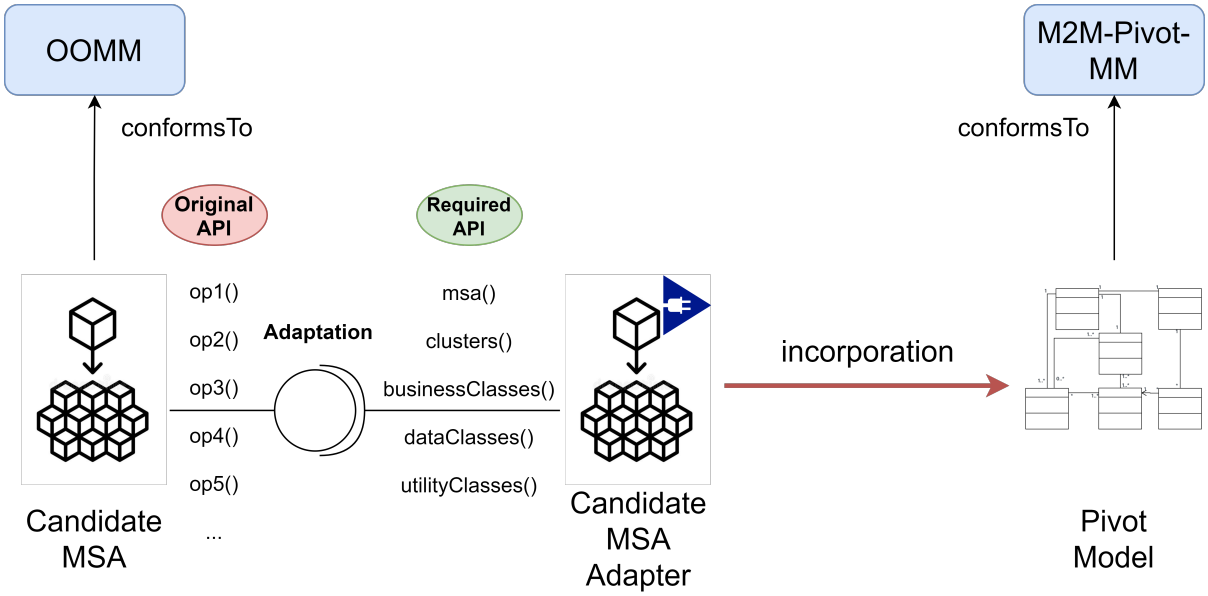

Figure 5.4: *Adaptation of the identified candidate MSA and its incorporation into the pivot model.*

we only manipulate platform-independent models, we are able to ensure that each step is applicable in any context as long as we are able to map the different platforms to a common OOMM.

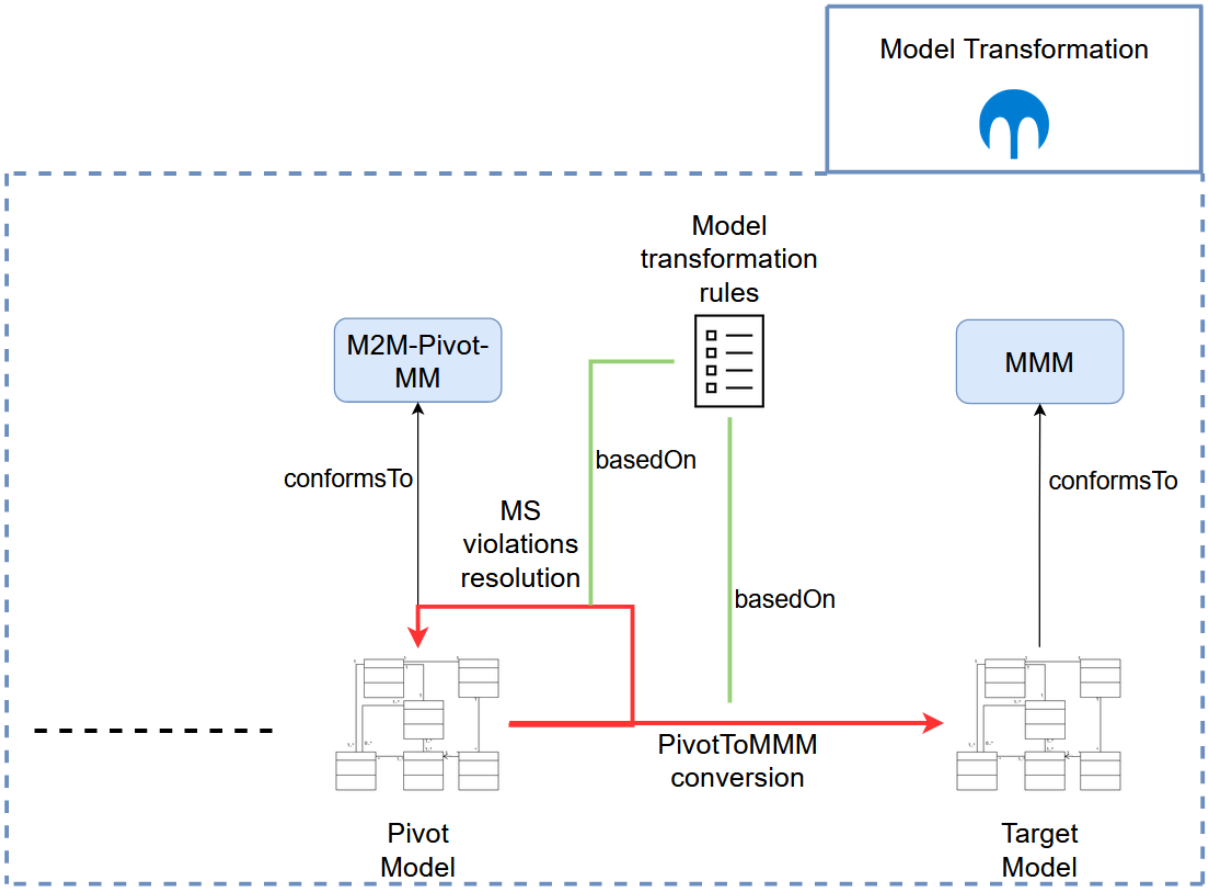

Figure 5.5: *The model transformation phase of the global MDE-based migration process.*

For this transformation phase, we distinguish between two different types of model transformations, each of which accomplishes a well-defined objective in the course of the migration process: (1) the microservice encapsulation violations resolution and (2) the `Pivot2MMM` conversion.

To understand the reasons behind the distinction between these two types of model transformations, we have to start by examining their input, namely the pivot model. As previously mentioned, the pivot model is obtained following the candidate MSA identification and incorporation phases of the migration process. In particular, the pivot model incorporates the identified candidate MSA, consisting of all candidate microservices, such that each candidate microservice is represented as a cluster of classes obtained from the original source model.

The act of creating microservice candidates from a monolith's set of classes is defined as **microservice encapsulation**. In this context, each microservice is its own application. Therefore, classes residing in one microservice should have their access restricted from classes belonging to other microservices. In other words, a microservice **encapsulates** its own set of classes. In Chapter 4 we defined a **microservice encapsulation violation** as a class encapsulated by one microservice which depends upon a class encapsulated by another microservice. These class-level dependencies include method invocations, class instantiations, public attribute accesses, class inheritances, class implementations of interfaces, among others. As such, we observe that class-level dependencies still exist between classes belonging to different microservice candidates in the pivot model (*see Figure* 5.6), despite the microservice identification step. Indeed, the microservice identification task cannot completely eliminate said dependencies, but at best may try to minimize them instead.

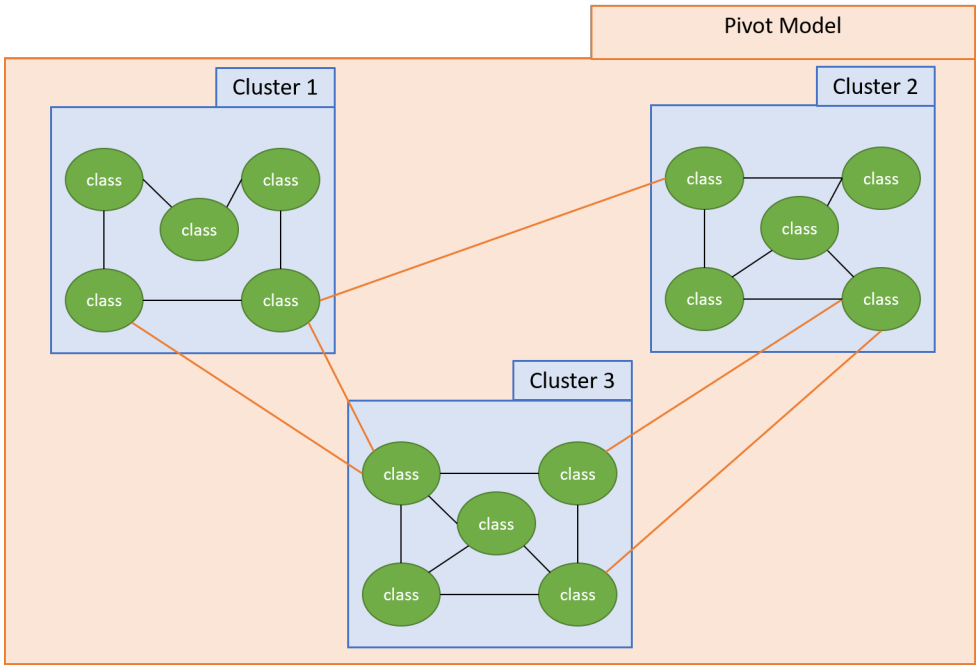

Figure 5.6: *The class-level dependencies in the microservice candidate clusters of the pivot model.*

Nonetheless, a properly crafted MSA should be devoid of any such violations. Thus, prior to the pivot model's transformation into the target model, we need to iden-

tify these violations in the pivot model and duly resolve them. Consequently, we decompose our model transformation phase into two steps accomplishing the following objectives respectively: (1) identifying and resolving the microservice encapsulation violations in the pivot model, and (2) converting the violation-free pivot model into the target model.

5.2.3.1 Microservice Encapsulation Violations Resolution

The first step in the model transformation phase of the migration process consists of identifying and resolving the microservice encapsulation violations in the pivot model.

The resolution mechanism for each identified encapsulation violation type is based on transformation rules and patterns defined in Chapter 4. These patterns consist of transforming the underlying OO-type dependencies for a given encapsulation violation type into corresponding MS-type dependencies.

For example, **method invocation violations** are healed by refactoring method invocations between classes belonging to different microservices into interface-based calls [ZSS+21]. Concretely, given a class A, encapsulated by a microservice MS_1, that invokes a method declared by a class B, encapsulated by a microservice MS_2, an **interface** IB is extracted from B and contains all its declared public methods. IB is set as a **required interface** by MS_1 and a **provided interface** by MS_2. Afterwards, all references to B in A are replaced with IB references. Moreover, given that microservices can only interact through **Inter-Process Communication** (**IPC**) protocols, an additional technological layer must be added by defining a **web service** and a **web service consumer** in the providing (MS_2) and requiring (MS_1) microservices, respectively. As such, IB's methods are implemented by the web service in MS_2 and exposed for consumption by other microservices, such that each implemented method acts as an intermediary to receive a request, forward it to the real method, and return the invocation result. In addition, IB's methods are implemented in MS_1 by the web service consumer which prepares the network calls necessary to consume the corresponding web service. Concretely, each created interface is represented in the Pivot model, and during the generation phase the code corresponding to web service and web service consumer is generated.

Once all violations have been identified and resolved, our pivot model becomes violation-free. As a result, we can proceed to the second step of the model transformation phase, namely the `Pivot2MMM` conversion.

5.2.3.2 `Pivot2MMM` Conversion

The second step in the model transformation phase of the migration process consists of converting the violation-free pivot model into the target MSA model. As previously mentioned, the violation-free pivot model conforms to `M2M-Pivot-MM`, while the target model conforms to `MMM`. The conversion consists of mapping each element in the pivot model to its corresponding element(s) in the target model. In particular, each ele-

ment in the pivot model will have its own transformation entity, defining its mapping mechanism. For example, every *M2MPivot-Cluster* entity in the Pivot model will be mapped to a `Microservice` entity in MMM. Once all transformations have been applied to all pivot model elements, the violation-free pivot model would be completely converted into its target model, and the transformation phase of the migration process comes to an end.

5.2.4 Model Exportation

The fourth and last phase in the MDE-based migration process consists of generating and packaging the project's target code based on its corresponding target MSA model. To generate the source code, we require both the target MSA model and the source code of the monolith. Indeed, until now we have manipulated a representation of the source code. However, since this approach aims to be reusable in different contexts we have limited our transformations to its PIM. To generate the source code we need to move from a platform-independent model to the platform itself. Therefore, we need to define a generic but extendable exporter that can be extended for each different platform.

Before the generation however, the target model can be configured by an expert to choose the desired artifacts to be generated. In a nutshell, configurations include choosing the target framework technologies, project builders, dependency managers, and other technologies relevant for microservices, such as the containerization technology, circuit breakers, service discovery, API clients, or communication protocols. Once the configuration complete, the **model exporter** takes over to generate the code.

5.3 Metamodels

Metamodels constitute a cornerstone in any MDE-based workflow, as they define the semantics of the models they describe [Sch06]. Indeed, without metamodels, models cannot be used as first class entities, and therefore cannot be subjected to model transformations. Consequently, a clear definition of an MDE-based workflow must be accompanied by a vivid description of the metamodels it uses. Metamodels can be either created from scratch or reused when they already exist. Metamodels that we intend to create from scratch should be formalized properly, while grouping their elements into different viewpoints. The usage of viewpoints allows us to visualize different aspects of a given metamodel, thus facilitating the comprehension of its structure and its associated domain. In fact, each viewpoint describes its metamodel at a specific level of abstraction, highlighting the necessary concepts required to make sense of its encapsulated aspect.

In this section, we provide a detailed description for all metamodels used in our MDE-based migration process. We start by introducing `OOMM`, our source metamodel. Afterwards, we define `M2M-Pivot-MM`, our intermediary metamodel. Finally, we discuss `MMM`, our target metamodel. For each metamodel, we start by indicating the motivation behind its conception, then describe its relevant viewpoints, and finally explicit

the reasons underlying its integration into our workflow.

5.3.1 The OOMM metamodel

The goal of the OOMM metamodel is to be able to describe a set of object-oriented systems developed in different languages. In particular, we denote the need to represent two types of entities: structural and behavioral. For the implementation of our approach, we use the FAMIX metamodel as it contains the majority of structural and behavioral entities we require. For reference, FAMIX denotes an extensible family of language-independent metamodels for the representation of various facets of object-oriented software systems [DTD01]. The viewpoint of FAMIX that is most relevant to our approach is its **core** viewpoint (*see Figure 5.7*). This viewpoint defines the most common structural entities (*e.g., classes, methods, attributes*) and behavioral entities *(e.g., inheritance, invocations, accesses*) used in object-oriented software design and implementation, and their relative relationships.

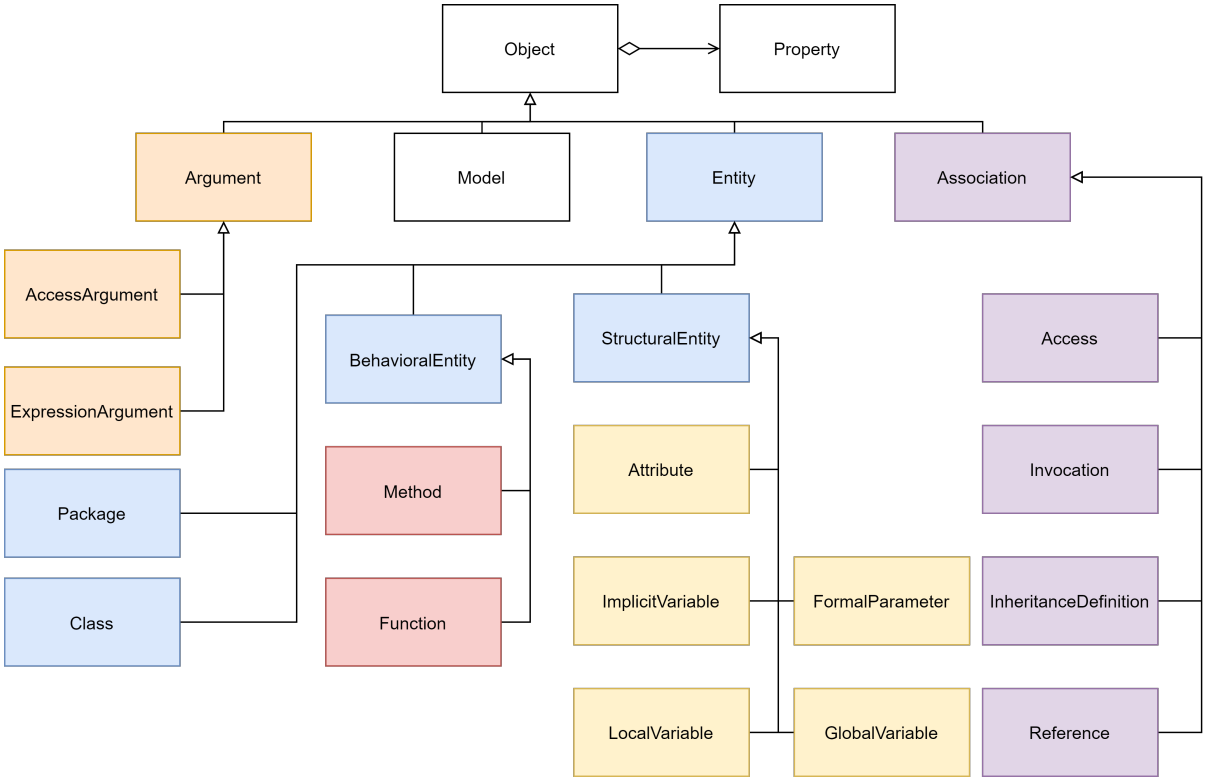

Figure 5.7: *FAMIX's core viewpoint (based on [DDT99]).*

In the context of this approach, we chose FAMIX as the metamodel of our monolith's corresponding model. Indeed, FAMIX appears to be a decent candidate for MDE-based workflows as illustrated by the research projects that had used it for MDE-based analysis and re-engineering tasks [DDT99, TDD00, DLT00, DAB+12]. Furthermore, FAMIX is generic enough and easily extensible to be used as a basic support for defining domain-specific metamodels. As such, we can extend it to define the other metamodels used in our workflow. Finally, many importers exist already for the extraction of FAMIX models, namely for Java, C# and C++ applications, and can be reused by our approach for different source monoliths, thus increasing its reusability and genericity.

5.3.2 The M2M-Pivot-MM metamodel

Monolithic-to-Microservices Pivot Metamodel (**M2M-Pivot-MM**) is an intermediary minimalist metamodel that we introduce as an extension of `FAMIX`. It includes abstractions of object-oriented entities, and intermediary entities, obtained following the application of our identification approach which extracts the layered architecture of the monolith.

Besides the entity inherited from the OOMM (i.e., classes), the core viewpoint of the metamodel introduces the notion of clusters and the entities associated to them (*see Figure 5.8*). A `M2MMicroserviceArchitecture` designates an identified candidate MSA, while a `M2MPivotCluster` designates a candidate microservice, belonging to the identified candidate MSA. The cluster maintains both `LayerArtifact` and `UtilityClass` entities. `LayerArtifact` classes encapsulate the business functionalities of their containing cluster and are extracted from the identification approach presented in Chapter 3. As for the `UtilityClass` entities, they represent the classes of the monolith that are not categorized into either the *Data Persistence* nor the *Layered Architecture*. Concretely, `UtilityClass` entities represent classes in a cluster that are responsible for orthogonal concerns such as logging and security. These concerns remain in the microservices created, thus they are included in each microservice as necessary. Any remaining entity referenced by the cluster and that doesn't pertain to any of the aforementioned categories is a utility class.

Definition 5.3.1: Bounded Context

In Domain-Driven Design (DDD), the context of an application relates to its underlying domain. In the context of microservices, this domain is often divided into different *bounded* contexts for each microservice. Furthermore, the interrelationships between each context is made explicit [LF14a].

`DataType` entities define data models introduced by the containing cluster. Inspired by the metamodel described in Chapter 3, we extend the notion of `DataType` into two types. The first, `DTO` (Data-Transfer Object), is a data type that is used to transfer data from one service to another. Oftentimes, in a monolithic paradigm, it used to pass certain data to the back-and-forth between the frontend and the backend. The second, `DataEntity`, is another data type that is used to represent the data stored in databases. In this metamodel we chose to differentiate the two, as `DataEntities` are more closely associated to the database, and it allows us to establish the **bounded context** of each candidate microservice.

Additionally, a cluster **requires** or **provides** a specific set of **interfaces** through which it can communicate with other clusters in their pertaining candidate MSA. Namely, we define the `M2MPivotRequiredInterface` entity to represent the required interfaces, and the `M2MPivotProvidedInterface` entity for provided ones. Both entities are tied to the `LayerArtifact` that it represents Initially, when this model is initialized, neither interface entity is instantiated. It is during the model transformation phase (see Section 5.2.3.1), and more specifically the MS encapsulation violation resolution step that any existing dependencies between clusters are resolved into a set of required/provided interfaces.

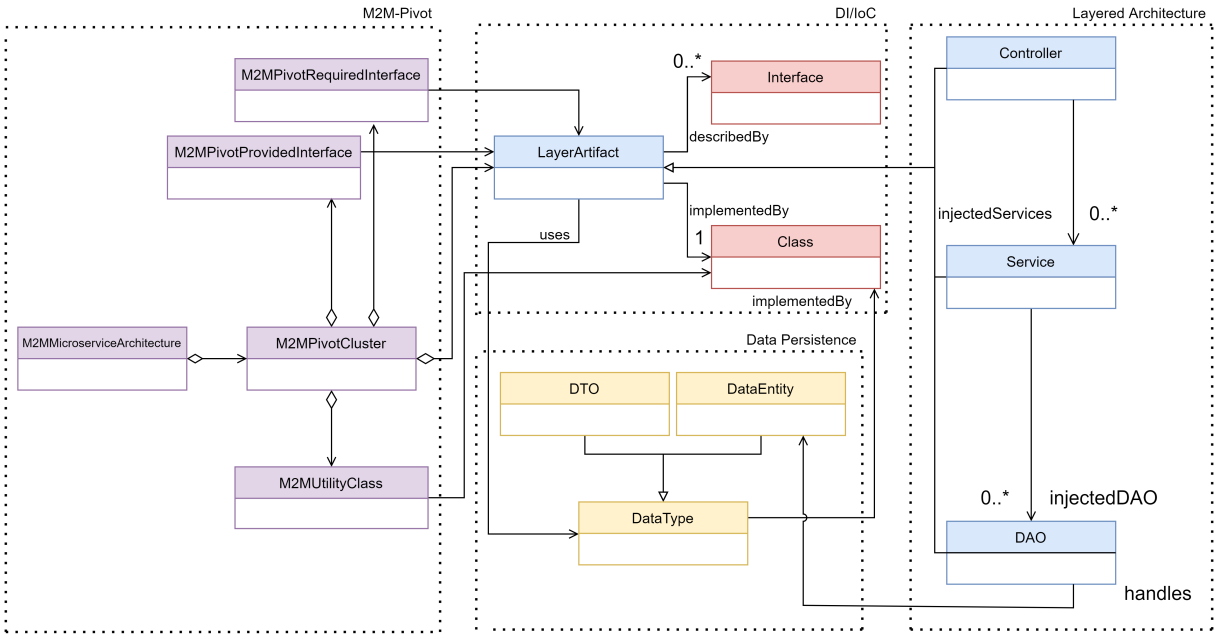

Figure 5.8: *The M2M-Pivot Meta-Model.*

The main reason behind using this metamodel is to describe an intermediary model that can incorporate a candidate MSA. This step is crucial to facilitate the resolution of the class-level dependencies causing microservice encapsulation violations. Indeed, it is easier to resolve these violations in an intermediary model where object-oriented entities and candidate MSA entities are combined, rather than resolving them in an MSA-based model. Secondly, when we transition to the target `MMM` model, we want focus on MSA-type dependencies.

Finally, we decided to define and implement the metamodel ourselves because, to the best of our knowledge, there are no dedicated metamodels that exists to represent a candidate MSA recovered from a monolithic object-oriented application. As such we adopted a minimalist definition approach that could be reused and/or further extended for other similar use cases beyond the scope of this contribution.

5.3.3 The MMM metamodel

The primary goal of this chapter is to design and implement the necessary model representation to facilitate the migration towards an MSA. As such, the metamodel we adopted is based on the **Microservices Architecture Model-Driven Development Metamodel** (**MMDDM**) proposed by [RSSZ18]. MMDDM has been deduced from SOA modeling approaches, and is primarily interested in supporting concepts for service design and operation modeling to support DevOps in **Microservices-Architecture-Model-Driven-Development** (**MSA-MDD**). As such, it targets the development of an MSA through a model-driven approach.

Concretely, MMDDM can be understood through three viewpoints: `Data`, `Service` and `Operation`. The `Data` viewpoint encapsulates concepts that can be used by service developers and domain experts to define domain-specific models for a microser-

vice [RSSZ18]. As such, the viewpoint provides concepts designating data structures for service interaction (i.e., the bounded context). The `Service` viewpoint is the core viewpoint of MMDDM. It encapsulates concepts that can be used by service developers and domain experts to define microservices, interfaces, and contracts [RSSZ18]. Finally, the `Operation` viewpoint encapsulates concepts that can be used by service developers and operators to specify microservices' implementation and deployment technologies [RSSZ18].

However, this metamodel focuses on model-driven development concepts which do not necessarily align with our migration goal. In other words, we require a target metamodel to support the concepts surrounding the migration towards MSA-based applications. Therefore, we propose **Microservices Architecture Model-Driven Migration Metamodel** (**MMM**). It extends MMDDM in that it inherits many of its concepts, but it additionally introduces new concepts concerned, for instance, with capturing business logic and data model aspects from the source application, such as business classes and data entities. Furthermore, it modifies some associations between some of its inherited concepts. For example, we specialize establish an aggregation relationship between the specialized entities of `ServiceContract` (i.e., `ProvidedServiceContract` and `RequiredServiceContract`). Also, a single relationship between `ServiceContract` and `ServiceInterface`.

Concretely, MMM can also be understood through three distinct viewpoints: Service, Business, and Configuration/Operation.

5.3.3.1 The Service Viewpoint

The **Service** viewpoint (*see Figure 5.9*) is the core viewpoint of this metamodel. It encapsulates concepts that can be used to define the microservice architecture, its microservices, and their interactions.

Microservices interact through **choreography** which requires no central entity dictating their interactions. Instead, interactions occur directly between them, where **service interfaces** enable them to interact through **operations**, while **service contracts** define the scope of operations they're allowed to expose and/or consume.

Concretely, for a given microservice M, a service interface can be provided by a service contract to expose a subset of M's operations that can be consumed by other microservices. Alternatively, M might require the consumption of operations exposed by service interfaces provided by other microservices through service contracts. In a nutshell, M provides service interfaces that are required by other microservices and requires service interfaces that are provided by other microservices. As such, a `Microservice` M exposes at least one `ServiceInterface` composed of at least one `Operation`. Each `ServiceInterface` requires/exposes one of M's `LayerArtifact` entities and introduces at least one `Operation`.

Each `Operation` has a name, can have zero/more `Parameter` entities, and references one of the exposed `LayerArtifact`'s methods. A `Parameter` has a type that references a `DataType`, can be optional, and possibly has an alias. Furthermore, it

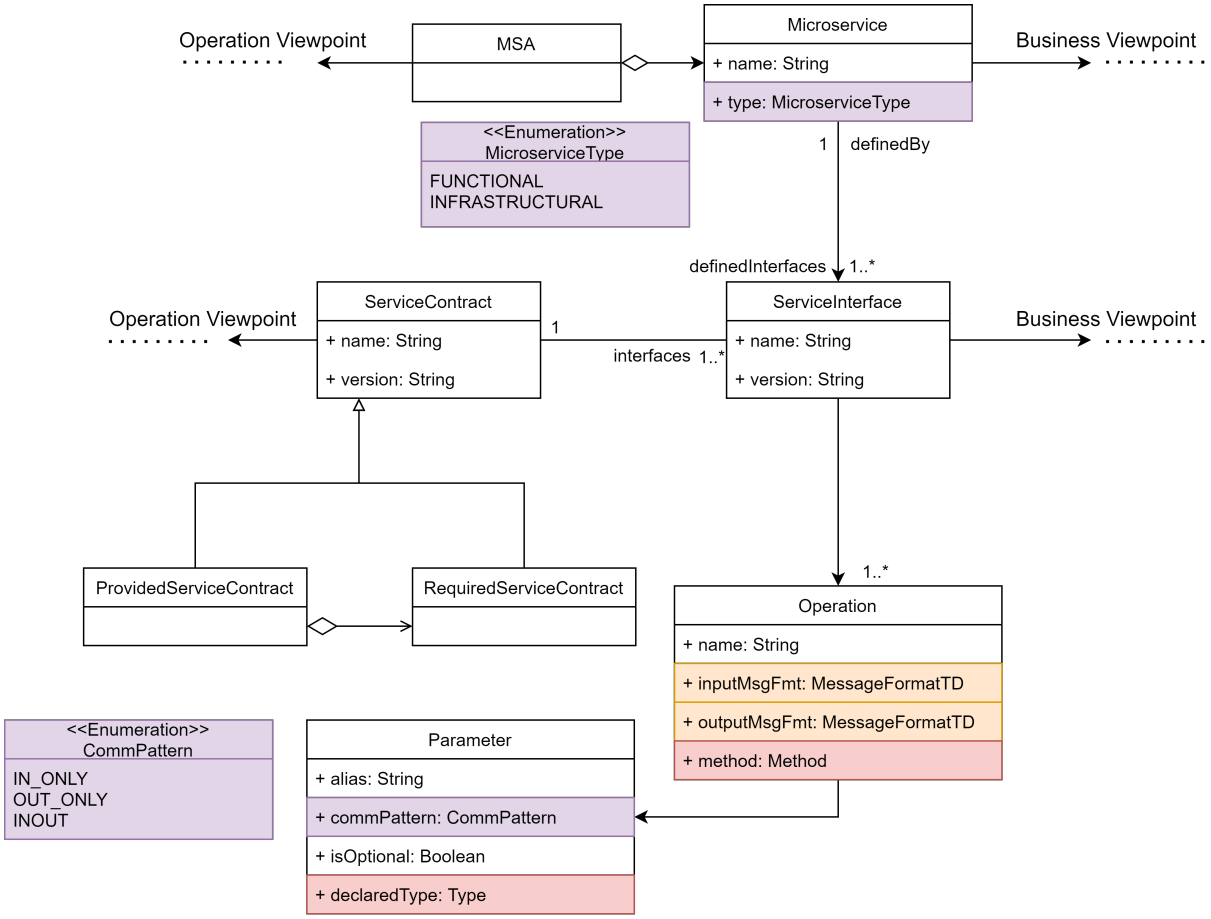

Figure 5.9: *MMM's* `Service` *viewpoint based on [RSSZ18].*

must specify the direction of information exchange. In particular, for a given `Operation`, a `Parameter` can be an input, an output, or both, as defined respectively by the enumerated values `IN_ONLY`, `OUT_ONLY`, and `INOUT` of the `CommPattern` enumeration entity.

Moreover, a `ServiceContract` aggregates a set of `ServiceInterface` entities. If the `ServiceInterface` entities are provided by a microservice, then the contract entity is a `ProvidedServiceContract`. Alternatively, if they are required from other microservices, then the `ServiceContract` is a `RequiredServiceContract`. Furthermore, each provided service interface is associated to the collection of its consumers. This is materialized by having a `ProvidedServiceContract` entity aggregate the set of its consumers, namely the `RequiredServiceContract` entities.

5.3.3.2 The Business Viewpoint

The **Business** viewpoint (*see Figure 5.10*) focuses on the business-logic of the application. As we represent a microservice-oriented application, this viewpoint encapsulates concepts that describe the internal structure, the business-logic, and the bounded context of each microservice.

Concretely, an `MSA` entity is composed of a set of `Microservice` entities, where

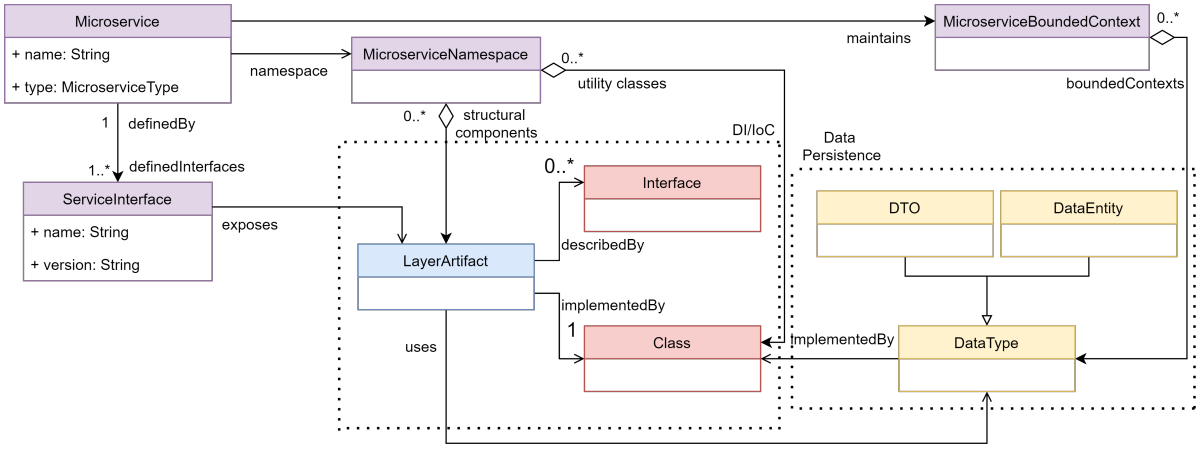

Figure 5.10: *MMM's* `Business` *viewpoint.*

each `Microservice` has a name and a type defined by the `MicroserviceType` enumeration entity. In particular, a `Microservice` either implements a business logic of its encapsulating MSA (i.e., has a `FUNCTIONAL` type) or plays a supporting infrastructural role therein (i.e., has a `INFRASTRUCTURAL` type).

Moreover, a `Microservice` is associated to a `MicroserviceNamespace` entity, and a `MicroserviceBoundedContext` entity. The `MicroserviceNamespace` is where it maintains its structural components. A microservice's structural components are represented by `LayerArtifact` entities, where each entity uses a `DataType` entity, either for data storage (`DataEntity`) or for data transportation purposes (`DTO`[2]).

All `DataEntity` entities introduced by a `Microservice`, and used by its associated `MicroserviceNamespace`, are maintained in its `MicroserviceBoundedContext`. Not pictured in Figure 5.10, the `LayerArtifact` is specialized by the three entities presented in Figure 5.8: Controller, Service, and DAO.

5.3.3.3 The Configuration/Operation Viewpoint

The final viewpoint is the **Configuration/Operation** viewpoint (*see Figure 5.11*). It encapsulates concepts that can be used by experts to configure the generation of an MSA-based application.

A central entity in this viewpoint is the `TechnologyDescriptor` (**TD**) entity, which allows the modeling of service implementation, deployment, and communication technologies. Each technology can have a characteristic, described by a name/-value pair.

Furthermore, endpoints specify an address, a communication protocol (e.g., REST APIs, AMQP, ...), and message formats (e.g., XML, JSON, ...). An endpoint can be associated with one/many protocols, while a protocol can use one/many message formats. In addition, an endpoint can be associated with a microservice's service contracts or with individual operations.

[2] An implementation of the **Data Transfer Object** design pattern.

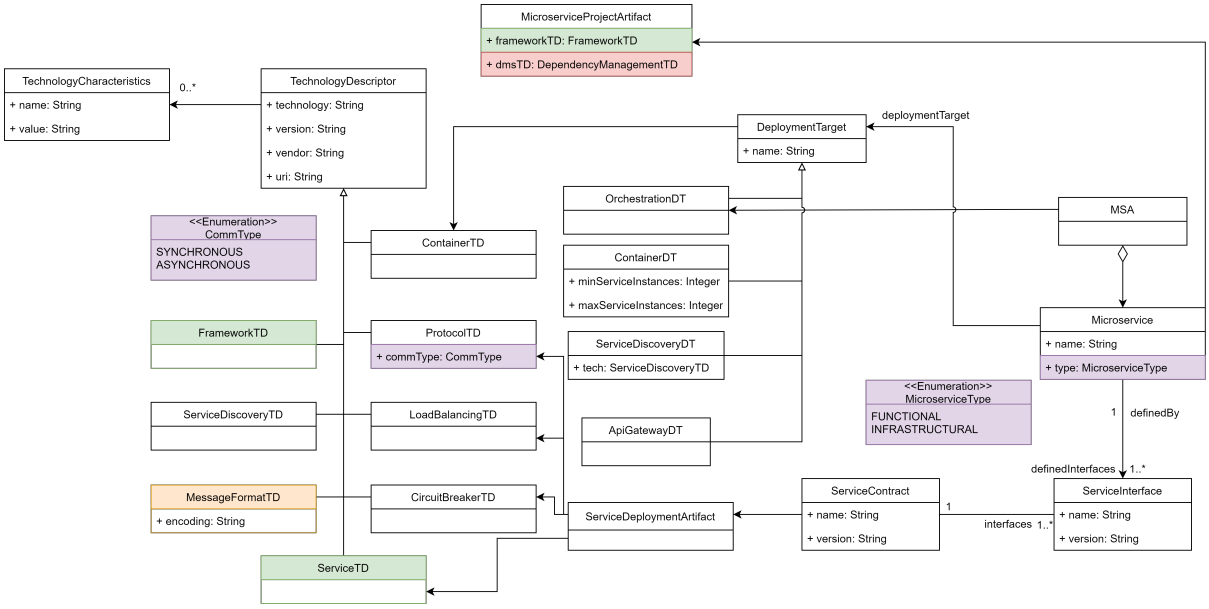

Figure 5.11: *MMM's* `Operation` *viewpoint based on [RSSZ18]).*

Moreover, to deploy a microservice, `ServiceDeploymentArtifact` entities are used to bundle the microservice's service contracts prior to their deployment. This approach allows to selectively deploy specific service interfaces that are provided by the microservice, but also to limit its interactions to microservices whose services are required by the bundled service contracts. In addition, each artifact is associated to a `ServiceTD` entity that specifies its implementation technology (e.g., Java used with the Spring framework). A microservice can also have technological descriptors for load balancing and fault mitigation associated to its deployable artifacts, if it performs service-internal infrastructure functionalities.

MMM also focuses on container-based MSA deployment. As such, a deployable microservice artifact is associated to a `ContainerDT` entity, describing its corresponding deployment container. The container can specify the minimum/maximum number of the deployed service's instances.

Additionally, the actual technology implementation of a container is provided by associating it to a `ContainerTD`. Consequently, a container's operating environment is described by its associated `ContainerTD`, in conjunction with one of the associated deployable artifacts `ServiceTD` entities. This allows a container's operating environment to be proactively modified by service operators, to designate any combination of container and service implementation technologies. In particular, the operating environment's name points to a container image, designating the combination of specific container and service implementation technologies. For example, deploying a Spring microservice in a Docker container can employ the `openjdk`[3] Docker image. Finally, to allow deployable microservice artifacts to be discovered internally and/or externally, their containers are associated with service discovery and/or API gateway entities.

[3] https://hub.docker.com/_/openjdk

5.4 Model Transformation Rules

The second cornerstone in any MDE-based workflow is the transformation engine, as it analyzes aspects of a model and synthesizes an alternative model representation which can provide more information about the initial system [Sch06]. Indeed, it is through these transformations that we are able to navigate through the different metamodels introduced in the previous section. Consequently, a clear definition of an MDE-based workflow must be accompanied by a set of transformation rules that formalize the transitions between the different models.

In this section, we define model transformations as either **adaptations** or **conversions**. In other words, when transforming a model we can either alter an existing model to redefine it to conform to the same metamodel, or we can convert it to conform to a new metamodel.

Additionally, all transformations are first class citizen entities and conform to a common model-transformation-centric framework, which simplifies their definition and allows them to be reused and integrated across different workflows.

This section provides a detailed description for all model transformations used in the MDE-based migration process. We start by introducing the transformations between the OOMM model and the Pivot model generated during the extraction process (see Section 5.4.1). Afterwards, we define and differentiate the transformation rules between the Pivot model and the MMM model. Particularly, we differentiate the adaptations of the Pivot model (Section 5.4.2) that must be applied before the model can be converted to the MMM model (see Section 5.4.3). For each model transformation, we start by indicating the input entities, the desired output entities, and finally describe the mapping rules.

5.4.1 Candidate MSA Incorporation Transformations

The initial phase of our MDE-based migration workflow is the model extraction. During this phase, the OOMM model is extracted from the monolithic OO source code. This model serves as an initial representation of the state of the application.

After the model extraction phase, the MSA identification and incorporation phase begins. Generally, the identification step results in a microservice architecture description which describes the partition of class entities from the monolithic application (e.g., [SSB+20b, LTV16, CLL18, GKGZ16, MCL17]). Each partition becomes a microservice candidate that is used to generate a microservice. During the identification step, we identify the candidate MSA with the approach from Chapter 3 using the OOMM model as input.

Furthermore, the candidate MSA is represented as a set of `LayerArtifact` clusters which needs to be incorporated into an intermediary pivot metamodel (M2M-Pivot-MM). During the incorporation step, we use the extracted Layered Architecture model and the microservice architecture description to generate the M2M-Pivot-

MM model. We generate an instance of M2M-Pivot-MM (*see Figure 5.6*) with an instance of `M2MMicroserviceArchitecture` to represent the identified microservice architecture. Then, for each identified cluster a `M2MPivotCluster` is created. Each `LayerArtifact` of an identified cluster is added to the `M2MPivotCluster` entity.

Any class entity referenced by a `LayerArtifact` that is not attributed as a `DataType`, nor a `LayerArtifact`, is mapped to a `UtilityClass`. This is usually applied to classes that serve infrastructural or cross-cutting concerns (e.g, logging, security) and are not tied to the business-logic of the application. Once the identified MSA is incorporated into the pivot model, we can move to the model transformation phase.

5.4.2 Microservice Encapsulation Violations Resolution Adaptations

In section 5.2.3, we highlighted the difficulty of transitioning from an object-oriented system to a microservice-oriented one. Particularly, we explained that during the microservice identification phase, most approaches partition classes in an attempt to reduce the inter-cluster dependencies but cannot completely eliminate them. Furthermore, in a microservice-oriented architecture, microservices can only communicate through web services. Therefore, we must adapt these OO-type dependencies into MS-type dependencies before we convert the Pivot Model into the MMM model (see figure 5.5). We must propose and apply transformation rules defined in [ZSS+21] to transform these OO-type dependencies into MS-type dependencies.

Concretely, it requires replacing the existing OO-type dependencies with a set of required and provided interfaces. During this step, we identify the symbol dependencies between the `LayerArtifact` entities belonging to different `M2MPivotCluster` entities. These dependencies can be detected by analyzing the OOMM entities that are tied to the `LayerArtifact` entities. For each identified dependency, a `M2MPivotRequiredInterface` is instantiated in the `M2MPivotCluster` from which the dependency originates. Also, a `M2MPivotProvidedInterface` is instantiated in the `M2MPivotCluster` which contains the used `LayerArtifact`.

5.4.3 Pivot2MMM Conversion

Once all OO-type dependencies have been resolved, the pivot model can be converted to an MMM model. Therefore, we propose a Pivot-to-MMM mapping model that maps the pivot elements to a microservice-oriented model (see Figure 5.12).

The `M2MMicroserviceArchitecture` is mapped to its MMM equivalent. Each `M2PivotCluster` is transformed into a `Microservice` entity with a namespace and a bounded context. Additionally, any `M2MUtilityClass` is mapped to microservice's namespace.

The `M2MPivotCluster`'s `M2MLayerArtifact` entities are mapped to the microservice's namespace. To establish the bounded context of each microservice, we observe the `DataType` entities used by each `LayerArtifact`. From this dependency,

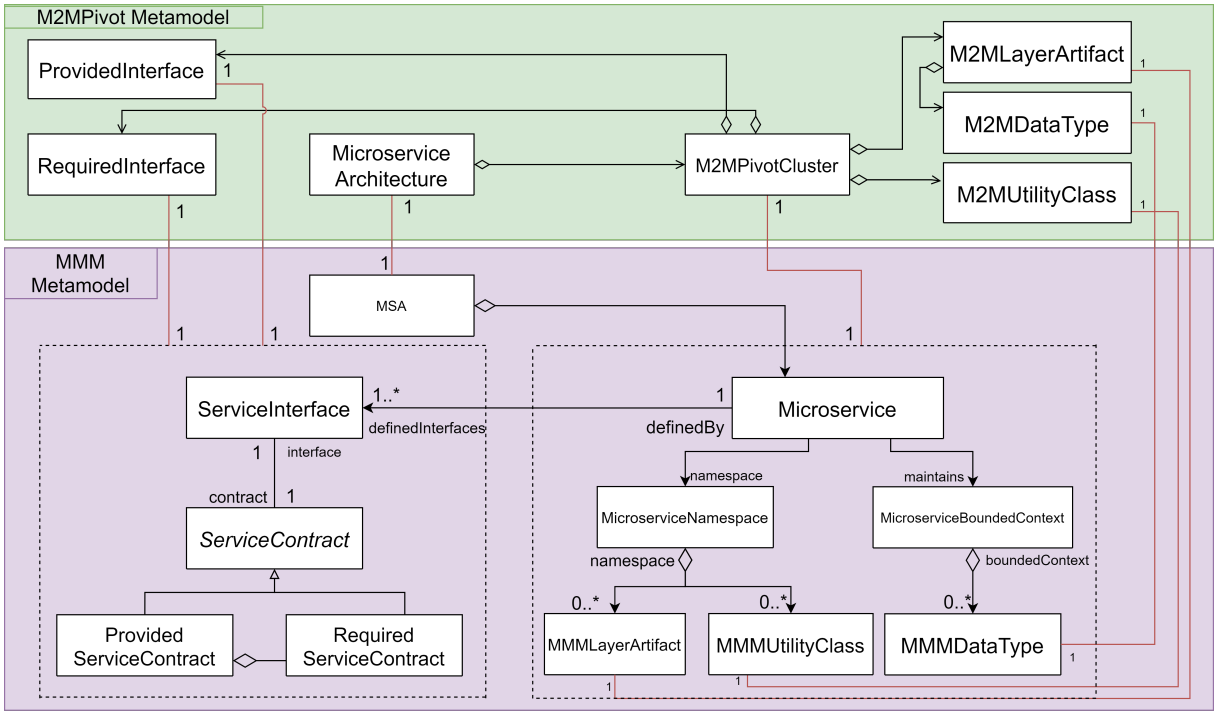

Figure 5.12: *Model conversion rules between M2M-Pivot-MM and MMM*

the `M2MDataType` entities used by the `LayerArtifact` are mapped to the microservice's bounded context.

Finally, for each `M2MPivotCluster`'s interface, a `ServiceContract` is defined and linked with its respective interface. In other words, each provided interface is mapped to a `ProvidedServiceContract`. In turn, this entity aggregates all the `RequiredServiceContract` entities that consume the service. These mapping rules cover the generation of both the business viewpoint and the service viewpoint. However, they do not cover the `Configuration/Operation` viewpoint. We leave this viewpoint for the expert to complete as it contains the design choices pertaining to the technologies used in the new architecture. Particularly, this allows the expert to configure technologies relevant to microservices, such as the containerization technology (e.g., Docker, Kubernetes) circuit breakers (e.g., resilience4j), service discovery (e.g., Consul or Zuul), or REST clients (e.g., Feign or RetroFit). The configurability of this viewpoint is essential for the provision of a packaged microservice architecture that is deployable based on the technologies known by the expert. Once the expert is done configuring, the target MSA's source code can be generated.

5.5 Target Code Generation

The final phase of the migration process is the target code generation of the identified architecture. Throughout this approach the main objective has been to propose a generic and reusable approach towards the migration of any application. In turn, we employed a set of platform-independent models to describe the source code, identify the target MSA, and transform the existing model to conform to a complete target

model which is specific to the MSA paradigm. Since, we rely solely on PIMs we require additional information to generate the code of the target architecture. Thus, along with the MMM model we use the source code of the monolith to generate the MSA's source code.

To ensure that we provide a generic approach towards generating the MSA's source code we rely on several design patterns. In particular, we rely on the *factory* pattern to generate the appropriate exporter based on an entity's specification. For instance, when exporting a microservice, we must know the implementation language to generate the proper project structure and to generate its namespace and bounded context. Therefore, the factory method is called to determine the appropriate microservice exporter based on its language implementation. Furthermore, when generating the source code the visitor pattern is applied.

To initiate the export, we initialize the generic Microservice Architecture Exporter and pass it the `MSA` entity of the MMM model. From there, the exporter calls the appropriate microservice exporter using the previously-described factory method. The microservice exporter initializes the project structure. Furthermore, it must ensure that the packages of the newly generated projects properly import the necessary dependencies. This is often done using the project builders and dependency managers (*e.g., Maven, Gradle, etc.*) used with the source monolith. In turn, the build management exporter is used. In addition, the microservice exporter must generate an image description for each microservice in the target MSA model, describing the configuration details necessary for the creation and deployment of its corresponding container. For example, if the microservice is configured to deploy in a Docker[4] container, then a corresponding Dockerfile must be created to describe its Docker image using the image description generator extended for the Docker configuration.

To summarize, for every MMM entity a generic exporter is required. Furthermore, for every configuration the exporter is extended. Once the MSA has been generated, it up to the architects to configure, and integrate each microservice.

5.6 Validation of the model-driven migration approach

In this chapter, we proposed an end-to-end approach that facilitates the migration of a monolithic application towards a microservice-oriented architecture. Throughout this thesis our goal as been to facilitate the modernization of an application's architecture in an industrial setting. In this section, we validate our approach by migrating an industrial application found at Berger-Levrault.

First, in Section 5.6.1, we present the application we wish to migrate, and describe the workflow we adopted for the migration. In Section 5.6.2, we present the research questions and evaluation methods. In Section 5.6.3, we present our results, and in Section **??** we discuss our approach. Finally, we conclude our work on the end-to-end approach to migrating monolithic applications.

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

5.6.1 Omaje : A case study

The goal of this case study is to evaluate our model-driven migration approach on an industrial application. To do so, we apply our approach on *Omaje*, a Spring-based application developed at Berger-Levrault. We first present Omaje in its monolithic architecture, and then we present the migration results of Omaje. To assist in the explanation of the migration, Figure 5.13 highlights the architecture of Omaje before, and after, the migration. Furthermore, during the execution of both applications we use a save of the database to simulate use with real data.

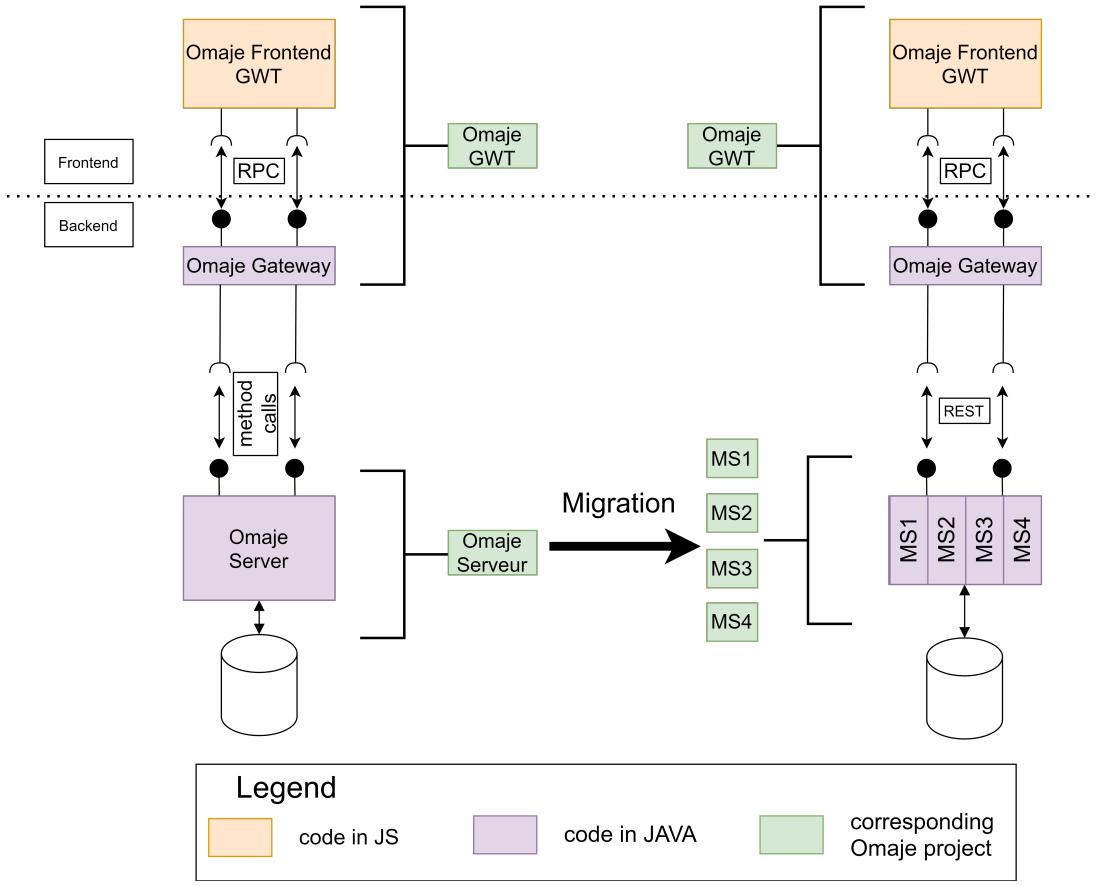

Figure 5.13: *(left) Omaje as a monolith, and (right) the result of migrating Omaje towards an MSA.*

5.6.1.1 Omaje as a monolith

Omaje is a license management system used internally to track the license distribution of Berger-Levrault's software. This application uses the same technology stack as several other applications proposed by Berger-Levrault. Particularly, Omaje adheres to the typical 3-tier application with a server-side application organized as a monolith. The 3-tier architecture can be viewed on the left side of Figure 5.13. Structurally, the application is organized around two projects: *OmajeServer* and *OmajeGWT*. *OmajeGWT* contains the code for the web client. While *OmajeServer* contains the server-side code which is compiled and packaged with *OmajeGWT* to form the monolithic web application. Whenever a request is made by the web client the gateway of the client

receives the RPC request and invokes the corresponding method in *OmajeServer*. The application is launched with its database of more than 50 tables with over 12 GB of real production data.

For the sake of the migration we focus on the OmajeServer project, which contains the server-side business logic. Concretely, the project contains 364 classes with a total of 37.7 KLOC. Using the Layered Architecture metamodel from Chapter 3, these classes can be categorized into 45 controllers, 38 services, 40 data-access objects. Furthermore, Omaje has 53 different data entities, and 33 data-transfer objects.

5.6.1.2 Omaje as an MSA

We automated our approach, `MDE-Mono2Micro`, using Moose, a platform for software and data analysis [DLT00]. We applied the automated approach to identify and transform the existing application into a set of microservices. The migration resulted in the identification and subsequent materialization of 4 microservices to replace the existing monolith (see Figure 5.13).

To finalize the migration, the developers must configure, test, and eventually manually fix each identified microservice. This task can be considerable when migrating large applications in one shot, which can impact the continued development of the application. Therefore, to limit this impact, the integration of the generated microservices was done incrementally. To do so, each microservice one configured, test, and manually fixed one by one. More information on the configuration of the microservices can be found in the discussion section of this experiment.

5.6.2 Research questions & Methodology

RQ1 (Validity): Can `MDE-Mono2Micro` support the migration of a monolithic object-oriented application to an MSA-based equivalent one?

The goal of this research question is to verify that the migration process is able to preserve the behavior of the application. In other words, we seek to demonstrate that MDE-Mono2Micro is capable of migrating a monolithic OO application to an MSA-based equivalent, while preserving its business logic.

To answer this RQ, we applied a set of user scenarios. These scenarios are described in the user manual provided by the development team, which resulted in a total of 56 user scenarios that cover all the described scenarios. We apply these scenarios on the web application twice, once with the monolithic backend and once with the microservice-oriented one. We compare the results after the migration with the results of the monolithic version to verify that the application produced the same results after being migrated.

RQ2 (Runtime Performance): Does the migration impact the overall responsiveness of the application?

The goal of this research question is to measure the impact of the changes to the architecture of the backend on the response time between the moment the client sends a request and the moment it receives a response. Indeed, as we migrate from a monolithic architecture towards a distributed one, we replace method calls with network calls. Furthermore, for each network call, the data passed must be serialized and deserialized which can also increase the response time.

To evaluate this RQ, we execute a set of user scenarios that requires a communication between the frontend and the backend. Each scenario uses different types of data (primitives, data entities, collections, etc.), of different sizes, and cycled references. We ran the scenarios using Firefox Version 99.0.1 on a laptop with 16 Go RAM and the Intel Core i7-6500u CPU. To limit the impact on the results, no other applications was running on the computer. To measure the response time, we modified the web client to measure the time spent waiting for the response from the server. Finally, each scenario is run a 1,000 times to reduce the wrong sample size bias, and the distribution is collected and presented as a box plot.

RQ3 (Build Performance): Does the migration impact the overall build time of the application?

As we migrate from a monolithic architecture to a distributed one, we modify the way the application is built. Furthermore, the build time of large applications can be prohibitively long. Therefore, the goal of this research question is to evaluate the build time of the application to see how it affects the development of Omaje. Indeed, as we move towards a distributed architecture we increase the number of projects that must be built.

To evaluate this RQ, we measure the time required to build the monolithic application (OmajeServer + OmajeGWT), and the microservice-oriented architecture (OmajeGWT + OmajeMS1 + OmajeMS2 + OmajeMS3 + OmajeMS4). To do so, we built each project independently.

5.6.3 Validation Results

In this section we present the results to the previously stated research questions.

RQ1 (Validity): Can `MDE-Mono2Micro` support the migration of a monolithic object-oriented application to an MSA-based equivalent?

To answer this RQ, we executed the user scenarios over the web client. The scenarios are described in the user manual provided by the development team. In total, 56 user scenarios were performed. Throughout the process, we did not uncover any bugs in the migrated Omaje application.

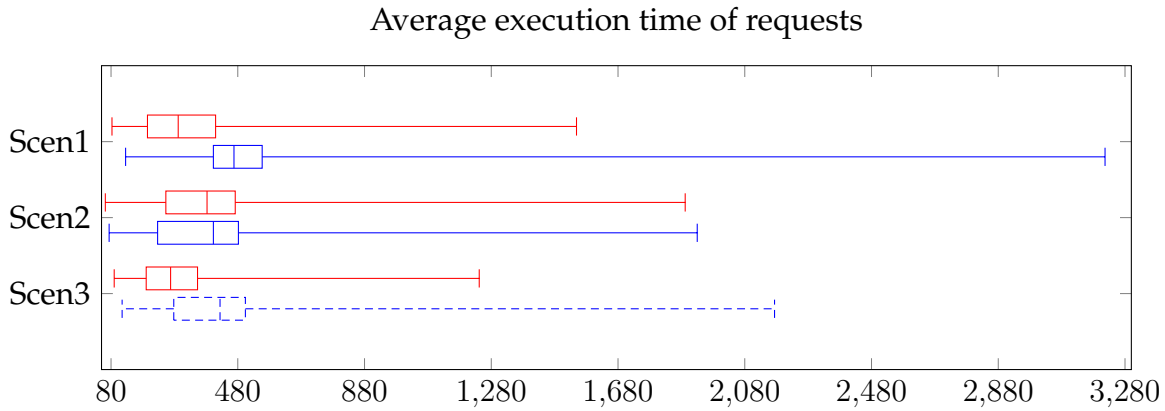

Figure 5.14: *Average execution time (in ms) of a scenario in both the monolith (red) and the MSA (blue).*

Summary RQ 1

To evaluate the business logic of the migrated application, we performed 56 different user scenarios. As each user scenario was run successfully on the migrated application approach, we were able to conclude that migration of Omaje was performed in a way as to preserve the business logic of the application.

RQ2 (Runtime Performance): Does the migration impact the overall responsiveness of the application?

To check the performance of the migrated application at runtime, we performed several user scenarios to measure the responsiveness between the client and the server. We performed each scenario with both the monolith and the MSA, and the results are displayed in Figure 5.14.

For scenario 1, we automated the consulting of the clients based on the software licenses they use. We note a displacement of 150-200 ms for the first quartile, the median and the third quartile between the monolith and the MSA. Overall, there is an increase in response time after the migration towards the MSA. Scenario 2 involves a search of the clients based on a set of filters. In this scenario, we did not experience the same increase between the monolith and the microservice. In fact, the relative remains relatively the same with an increase of 20 ms between the medians. Finally, in scenario 3 we automate the search of active users of Omaje. In this scenario we denote an increase of 90-160 ms between the monolith's response time and the MSA's response time.

Overall, the migration does impact the overall responsiveness of the application. This can be explained by the fact that we replace method calls in the monolith with network calls in the microservice-oriented architecture. However, the increase in response time remains imperceptible for the end-user.

Summary RQ 2

The passage from a monolithic architecture to a distributed one does increase the response time of the application. The increase can be considered negligible from the viewpoint of the user experience, however the number of scenarios does not cover enough of the application to determine that the application is not negatively impacted, and we can only conclude that from these scenarios, the features covered are not impacted.

RQ3 (Build Performance): Does the migration impact the overall build time of the application?

To answer this RQ, we measured the compilation time to build each application. For the monolithic application, this consists in building both *OmajeServer* and *OmajeGWT*. To build the microservice-oriented architecture, this consists in building *OmajeGWT* (which contains the client and routing to the other microservices), as well as the 4 microservices (i.e., *MS1*, *MS2*, *MS3*, *MS4*).

Table 5.1: *Build time for the monolithic version of Omaje.*

	Omaje GWT	Omaje Server
Build Time	6m50s	20.225s

Table 5.2: *Build time for the microservice-oriented version of Omaje.*

	Omaje GWT	MS1	MS2	MS3	MS4
Build Time	6m41s	16.244s	14.072s	12.620s	13.642s

Table 5.1 presents the compilation time of each project for the monolith, and Table 5.2 presents the build time of each project for the MSA. As we can see, the build time of the web client is unaffected by the migration of the monolith. For the server build time, we move from having one server compilation of about 20 seconds to 4 different compilations that take the same range of build time. However, the migration allowed for the decoupling of the *OmajeServer* project from the *OmajeGWT* project. In other words, when a change is required in the backend, the developers no longer need to compile the whole of the application. Instead, they can focus on compiling the microservice in question.

Summary RQ 3

The build time of the Omaje application remains relatively the same with the web client taking most of the time. However, by decoupling the frontend from the backend, we are able to reduce the need to build the OmajeGWT when only changes to the backend are required. This can save some considerable time during the development of the application, which we were able to experience during the configuration phase of the migration process, as it allowed to configure manually each microservice without rebuilding the client between configurations.

5.6.4 Threats to validity

In this section, we discuss the threats to the validity of our case study using the definition proposed in [WRH+12]. Specifically, we present the construct validity, the internal validity, and the external validity.

5.6.4.1 Construct Validity

The purpose of this study is to determine the ability of our approach to migrate a real-world application, and its impact on the continued development of the application. In this subsection, we evaluate whether the measures used really represent what we are trying to investigate with the proposed research questions.

With regard to the end-users, we wanted to validate the application's behavior and usability. For the application's behavior, we used the user scenarios proposed by the user manual to determine whether the application still behaved as expected. This user manual is available on the front page of the web application to help any new user to navigate through its different features. Therefore, it is likely to cover the important features of the application. However, we acknowledge that these user scenarios do not necessarily cover all the application's requirements. In terms of usability, we wanted to validate that the migrated application remained responsive. For this we measured and compared the response time of the migrated application with its monolithic version in different scenarios. To measure the response time, we evaluate the execution time with several user scenarios. However, since there are no functional tests, we have to automate each scenario. Due to time constrictions, we limited the number of scenarios that were used to evaluate the performance of the application. There is a likely risk that we do not cover enough of the application to determine the responsiveness across the application.

5.6.4.2 Internal Validity

The threats to the internal validity of this experiment include whether external factors impacted the results. We cover the threats related to both RQ1 and RQ2:

(RQ1) Validity For the different user scenarios we asked the practitioner who were not involved with the migration to evaluate the behavior of the migrated application. Thus, we consider the reported results unbiased.

(RQ2) Runtime Performance We measure the usability of the migrated application by compare its responsiveness with the original application. During the migration we did not seek to optimize the identification phase to produce performance-optimal microservice. Furthermore, results were computed using the internal tools available with the web client technology, using the same computer, and without other applications other than the web application and the server.

5.6.4.3 External Validity

In this experiment, we validated our approach by migrating an industrial application. Since it is not open-source, we cannot share the results and this makes replication difficult. Furthermore, we limit our validation to one application, therefore we cannot validate the approach's genericity. However, our approach is designed to be language-agnostic and was implemented using open-source technologies, therefore we are confident that the proposed approach can be reused to a high degree in other cases.

5.7 Conclusion

In this chapter, we have proposed an MDE-based monolithic object-oriented to MSA migration approach. This approach aims to be as generic as possible and extendable based on the migration context. To do so, we have proposed a set of platform-independent models that can be reused throughout the migration process. Particularly, this migration process consists of extracting the source application's model, identifying its candidate MSA and incorporating it into an intermediary pivot model. From this pivot model, we identify and resolve the microservice encapsulation violations, and convert the violation-free pivot model into the MMM model. Finally, we generate the source code for the target platform using the monolith's source code, and the MMM model to guide the generation process.

We validated our approach by migrating an industrial application. The migration was performed using the tool `MDE-Mono2Micro`. To evaluate the results of our migration, we performed three different experiments to evaluate the behavior, the usability, and the build time of the microservice-oriented architecture. The findings of our experiment conclude that the behavior, and the usability of the application is not affected by the migration. Furthermore, the build time of the application does not have negative consequences on the development of the application. Based on the evaluation of these results, we conclude that the migration of Omaje was successful.

VI

Conclusion & Future Works

Contents

This chapter provides an overview of the contributions proposed in this thesis. In it, we also highlight the limitations of the proposed contributions, and identify future research directions. Finally, we present our publications.

6.1 Summary of Contributions

The goal of this thesis is to contribute to the migration of monolithic applications towards a microservice-oriented architecture. Throughout this thesis, we highlighted two research problems:

1. **Reverse engineering, and identifying, the microservice-oriented architecture**: the identification relies on analyzing program artifacts, in order to extract a decomposition of the source code (e.g., clusters of classes) that will constitute the MSA.
2. **Transforming the existing code to materialize the microservice-oriented architecture**: from the identified architecture, the existing source code of the monolith must be refactored to produce valid microservices.

In this thesis, we proposed several contributions to address these two research problems:

- An identification approach which leverages the internal architecture of monolithic applications to propose an MSA (Chapter 3): Our approach addresses the first research problem by partitioning a set of classes of an object-oriented application into a set of clusters, each cluster representing a microservice. Unlike most existing approaches, our clustering strategy relies on an intermediate extraction phase which seeks to extract information about the internal architecture of the application. By identifying the structural components of the application, we can propose a decomposition approach which takes into consideration common decomposition antipatterns. We then apply a clustering algorithm on the structural components to generate the microservice candidates.

- A semi-automated approach to materialize the identified architecture through a set of transformation rules (Chapter 4): This contribution addresses the second research problem by proposing a semi-automated systematic transformation approach towards materializing microservice candidates. This approach takes as input an identified MSA description and the source code of the monolith. Using the architecture description as a guide, we generate the microservice's source code. However, the act of separating the monolith's source code into separate projects reveals object-oriented dependencies between them that must be resolved. Thus, we propose a set of transformation rule to resolve the different OO dependencies into a set of required/provided interfaces. During the generation process, these interfaces are implemented as web services to permit the communication between the different microservices.

- An end-to-end migration approach driven by model engineering (Chapter 5): our proposed approach combines both previous contributions to address both research problems. The goal of this approach is to generalize both contributions using model-driven engineering techniques. Particularly, this approach aims to be generic enough to be applied to different languages, frameworks, and technologies. We propose several generic metamodels to represent the application throughout the different phases of the migration. Each metamodel aims to be generic to promote their reusability throughout the migration of different platforms. This allows for a greater flexibility towards adapting the final result based on the expert's choices.

6.2 Limitations

We denote several limitations with the proposed approaches:

In the case of our contribution to the identification problem, we denote several points. First, our approach is limited to the static analysis of the source code. During this thesis, we made the choice to limit the accepted input of our identification approach to the source code of the application. Our reasoning was that the source code was the most commonly available source of information. However, it is not the only source of information that is available. In particular, we relied on static analysis techniques to determine the dependencies between the classes of an application. This means we are unable to address polymorphism and dynamic binding dependencies.

However, during the extraction of the internal architecture we take into consideration the dependency injection found in most popular frameworks (e.g., Spring). However, another problem with static analysis is that it treats dead/unused code with the same importance as used code. This may create some noise during the identification process by promoting certain dependencies other others. Dynamic analysis addresses all of these limitations by highlighting dependencies between classes based on their real use. However, the challenge with dynamic analysis is that it requires collecting traces during the use of the application which can be difficult to acquire.

Also, the proposed identification approach clusters the classes of an application using the hierarchical clustering algorithm. While this can provide a near-optimal solution, other clustering techniques exist that may provide more accurate solutions (e.g., graph neural networks, search-based techniques, evolutionary algorithms).

With regard to the approaches that address the transformation problem, we denote that our experimentation is limited to JAVA-based applications. While Java is a popular object-oriented language, it is not the only OO language used to implement monolith. However, the transformation approach we present attempts to generalize its transformation to pure object-oriented mechanisms.

6.3 Future Directions

Several directions have been identified throughout this thesis that could not be explored due to constraints. We split these directions into two distinct categories.

The first category consists of directions that relate to the identification of microservices. In particular, we have highlighted that our approach is limited to the static analysis of code. In future works, we would like to integrate a dynamic analysis on top of the current static analysis. For instance, we could identify microservice through the elaboration of a set of use cases. These use cases could be executed on an instrumented version of the monolith to identify the classes used to serve these use cases.

Another challenge we encountered during our work identifying microservices is the lack of an evaluation framework for identification approaches. While there are more than 30 approaches presented in the literature, few approaches provide the full data-set of their identification. Instead, they provide the quantified results, which makes comparing approaches using other metrics difficult. Furthermore, there are few approaches that offer to quantify their results to make comparison between approaches possible. In a future work, we would like to expand the evaluation methodology presented in Chapter 3 to consider other types of evaluation metrics (e.g. the metrics proposed in [LCG+15], and [JLC+21]) as well as include a quantitative comparison between different approaches.

The second category involves the transformation problem. In the work proposed in Chapter 4 and Chapter 5, we focused on the transformation of object-oriented applications. However, during the experimentation we limited ourselves to JAVA applications. In Chapter 4, our approach attempts to cover systems implemented in any OO

language, however, there are several OO concepts (e.g., reflexivity, friends) that we did not cover. In future works, we would like to address these OO concepts. Regarding the model-driven migration approach, we aim to propose a generic approach. However, we limited our experiment to a single application. In future works, we would like to extend our experimentation to see whether our approach is generic. In particular, we would like to address this by migrating applications written in C#, and SmallTalk. Furthermore, our tool `MDE-Mono2Micro` is implemented in Pharo using the Moose platform for code analysis. However, Pharo is a language that few developers are comfortable using. To overcome this limitation, we would like to develop a visual environment to enable developers/architect to make high-level decision regarding the migration of their monolithic application. This visual environment is current focus of an internship at Berger-Levrault.

Finally, from an industrial perspective, the case study presented Chapter 5 has resulted in the successful migration of Omaje. In a future work, Berger-Levrault is interested in applying this work in the context of two additional applications developed using the same technological stack as Omaje.

6.4 Publications

This PhD thesis started in November 2018. During this period we have worked the following research papers (in chronological order):

- Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde-Lilia Bouziane, Rahina Oumarou Mahamane, **Pascal Zaragoza**, Christophe Dony. *From Monolithic Architecture Style to Microservice one Based on a Semi-Automatic Approach*. ICSA 2020: 157-168.
- **Pascal Zaragoza**, Abdelhak-Djamel Seriai, Abderrahmane Seriai, Hinde-Lilia Bouziane, Anas Shatnawi, Mustapha Derras. *Refactoring Monolithic Object-Oriented Source Code to Materialize Microservice-oriented Architecture*. ICSOFT 2021: 78-89. (**nominated for best student paper**)
- **Pascal Zaragoza**, Abdelhak-Djamel Seriai, Abderrahmane Seriai, Hinde-Lilia Bouziane, Anas Shatnawi, Mustapha Derras. *Materializing Microservice-Oriented Architecture from Monolithic Object-Oriented Source Code*. In: Software Technologies. ICSOFT 2022. Communications in Computer and Information Science. (**invited to ICSOFT extension & accepted**)
- **Pascal Zaragoza**, Abdelhak-Djamel Seriai, Abderrahmane Seriai, Anas Shatnawi, Mustapha Derras: Leveraging the Layered Architecture for Microservice Recovery. ICSA 2022: 135-145
- **Pascal Zaragoza**, Bachar Rima, Abdelhak-Djamel Seriai, Abderrahmane Seriai, Mustapha Derras. *Model-Driven Engineering Migration of an Object-Oriented Monolithic Application to a Microservices Architecture*. (**to be submitted**)

Bibliography

[ACC+21] Wesley K. G. Assunção, Thelma Elita Colanzi, Luiz Carvalho, Juliana Alves Pereira, Alessandro F. Garcia, Maria Julia de Lima, and Carlos Lucena. A multi-criteria strategy for redesigning legacy features as microservices: An industrial case study. In *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*, pages 377–387. IEEE, 2021. 19, 21, 22, 23, 24, 25, 27

[ADM20] Omar Al-Debagy and Peter Martinek. Extracting Microservices' Candidates from Monolithic Applications: Interface Analysis and Evaluation Metrics Approach. In *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)*, pages 289–294. IEEE, jun 2020. 19, 21, 22, 23, 24, 25, 27

[AGC18] Aayush Agarwal, Subhash Gupta, and Tanupriya Choudhury. Continuous and integrated software development using devops. In *2018 International Conference on Advances in Computing and Communication Engineering (ICACCE)*, pages 290–293, 2018. 4

[AIE19] Muhammad Abdullah, Waheed Iqbal, and Abdelkarim Erradi. Unsupervised learning approach for web application auto-decomposition into microservices. *Journal of Systems and Software*, 151(February):243–257, may 2019. 19, 22, 23, 24, 25, 26, 27

[AM21] Omar Al-Debagy and Peter Martinek. A microservice decomposition method through using distributed representation of source code. *Scalable Comput. Pract. Exp.*, 22(1):39–52, 2021. 18, 19, 21, 22, 23, 24, 25, 27, 30, 34, 36, 51, 52, 54

[Ami18] Mohammad Javad Amiri. Object-Aware Identification of Microservices. In *2018 IEEE SCC*, pages 253–256. IEEE, jul 2018. x, xi, 9, 19, 20, 22, 23, 24, 25, 26, 27

[AS16] Zakarea Al Shara. *Migration des applications orientées-objet vers celles à base de composants*. PhD thesis, 2016. Thèse de doctorat dirigée par Dony, Christophe Informatique Montpellier 2016. 16

[AS20] João Franscisco Almeida and António Rito Silva. Monolith migration complexity tuning through the application of microservices patterns. In Anton Jansen, Ivano Malavolta, Henry Muccini, Ipek Ozkaya, and Olaf

Zimmermann, editors, *Software Architecture*, pages 39–54, Cham, 2020. Springer International Publishing. xi, 9, 29

[ASM+21] Manel Abdellatif, Anas Shatnawi, Hafedh Mili, Naouel Moha, Ghizlane El Boussaidi, Geoffrey Hecht, Jean Privat, and Yann-Gaël Guéhéneuc. A taxonomy of service identification approaches for legacy software systems modernization. *Journal of Systems and Software*, 173:110868, 2021. 16, 17

[ASS+21] Shivali Agarwal, Raunak Sinha, Giriprasad Sridhara, Pratap Das, Utkarsh Desai, Srikanth Tamilselvam, Amith Singhee, and Hiroaki Nakamuro. Monolith to microservice candidates using business functionality inference. In *2021 IEEE International Conference on Web Services (ICWS)*, pages 758–763, 2021. 19, 22, 23, 24, 25, 27

[AST+15] Zakarea Alshara, Abdelhak-Djamel Seriai, Chouki Tibermacine, Hinde-Lilia Bouziane, Christophe Dony, and Anas Shatnawi. Migrating large object-oriented applications into component-based ones: instantiation and inheritance transformation. In *International Conference on Generative Programming: Concepts and Experiences, GPCE 2015*, pages 55–64. ACM, 2015. 75, 76

[BCS21] Miguel Brito, Jácome Cunha, and João Saraiva. *Identification of Microservices from Monolithic Applications through Topic Modelling*, page 1409–1418. Association for Computing Machinery, New York, NY, USA, 2021. 19, 22, 23, 25, 27

[BDD+18] Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Stephan T. Larsen, and Manuel Mazzara. From monolithic to microservices: An experience report from the banking domain. *IEEE Software*, 35(3):50–55, 2018. x, 8

[BGDR17] Luciano Baresi, Martin Garriga, and Alan De Renzis. Microservices identification through interface analysis. In Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen, editors, *Service-Oriented and Cloud Computing*, pages 19–33, Cham, 2017. Springer International Publishing. xi, 9, 19, 21, 22, 23, 24, 25, 26, 27

[BHJ16] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016. ix, 7

[Big89] T.J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, 1989. 15

[BLWG99a] J. Bisbal, D. Lawless, Bing Wu, and J. Grimson. Legacy information systems: issues and directions. *IEEE Software*, 16(5):103–111, 1999. 8

[BLWG99b] Jesus Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. Legacy information systems: Issues and directions. *IEEE Softw.*, 16(5):103–111, 1999. 14, 15

[BSG20] Antonio Bucchiarone, Kemal Soysal, and Claudio Guidi. A model-driven approach towards automatic migration to microservices. In Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 15–36, Cham, 2020. Springer International Publishing. 19, 22, 23, 24, 25, 27

[CC90] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, 1990. 15

[CGC+20] Luiz Carvalho, Alessandro Garcia, Thelma Elita Colanzi, Wesley K. G. Assuncao, Juliana Alves Pereira, Baldoino Fonseca, Marcio Ribeiro, Maria Julia de Lima, and Carlos Lucena. On the Performance and Adoption of Search-Based Microservice Identification with toMicroservices. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 569–580. IEEE, sep 2020. 22, 23, 24, 25, 27

[Che18] Lianping Chen. Microservices: Architecting for continuous delivery and devops. In *IEEE International Conference on Software Architecture, ICSA 2018, Seattle, WA, USA, April 30 - May 4, 2018*, pages 39–46. IEEE Computer Society, 2018. 4

[CLL18] Rui Chen, Shanshan Li, and Zheng Li. From Monolith to Microservices: A Dataflow-Driven Approach. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, pages 466–475, 2018. 19, 20, 22, 23, 24, 25, 27, 106

[DAB+12] Stéphane Ducasse, Nicolas Anquetil, Muhammad Bhatti, Andre Hora, Jannik Laval, and Tudor Girba. Mse and famix 3.0: an interexchange format and source code model family. Research report, Laboratoire d'Informatique Fondamentale de Lille, 05 2012. 99

[DABFP20] Adambarage Anuruddha Chathuranga De Alwis, Alistair Barros, Colin Fidge, and Artem Polyvyanyy. Remodularization analysis for microservice discovery using syntactic and semantic clustering. In Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *Advanced Information Systems Engineering*, pages 3–19, Cham, 2020. Springer International Publishing. 22, 23, 24, 25, 27

[DABPF18] Adambarage Anuruddha Chathuranga De Alwis, Alistair Barros, Artem Polyvyanyy, and Colin Fidge. Function-splitting heuristics for discovery of microservices in enterprise systems. In Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu, editors, *Service-Oriented Computing*, pages 37–53, Cham, 2018. Springer International Publishing. 18, 19, 22, 23, 24, 25, 27

[DDT99] Serge Demeyer, Stéphane Ducasse, and Er Tichelaar. Why famix and not uml? uml shortcomings for coping with round-trip engineering. In *In Proceedings of UML'99, Fort Collins*. Citeseer, 09 1999. 99

[DEF+21] Mohamed Daoud, Asmae El Mezouari, Noura Faci, Djamal Benslimane, Zakaria Maamar, and Aziz El Fazziki. A multi-model based microservices identification approach. *Journal of Systems Architecture*, 118:102200, 2021. 19, 20, 21, 22, 23, 24, 25, 26, 27, 28

[DGL+17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017. vi, viii, 4, 6

[DLT00] Stéphane Ducasse, Michele Lanza, and Er Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, volume 4. Citeseer, 04 2000. 99, 111

[DMF+20] Mohamed Daoud, Asmae El Mezouari, Noura Faci, Djamal Benslimane, Zakaria Maamar, and Aziz El Fazziki. Automatic microservices identification from a set of business processes. In Mohamed Hamlich, Ladjel Bellatreche, Anirban Mondal, and Carlos Ordonez, editors, *Smart Applications and Data Analysis*, pages 299–315, Cham, 2020. Springer International Publishing. 19, 21, 22, 23, 24, 25, 26, 27, 28

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. Famix 2.1 - the famoos information exchange model. Technical report, University of Bern, 01 2001. 99

[ECA+16] Daniel Escobar, Diana Cardenas, Rolando Amarillo, Eddie Castro, Kelly Garces, Carlos Parra, and Rubby Casallas. Towards the understanding and evolution of monolithic applications as microservices. In *2016 XLII Latin American Computing Conference (CLEI)*, pages 1–11. IEEE, oct 2016. 19, 22, 23, 24, 25, 27

[EGHS16] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *IEEE Software*, 33(3):94–100, 2016. viii, 7

[Eri03] Evans Eric. Domain-driven design: tackling complexity in the heart of software. *Addison-Wesley Professional*, 2003. 18

[ERM20] Fola-Dami Eyitemi and Stephan Reiff-Marganiec. System Decomposition to Optimize Functionality Distribution in Microservices with Rule Based Approach. In *2020 IEEE International Conference on Service Oriented Systems Engineering (SOSE)*, pages 65–71. IEEE, aug 2020. 19, 22, 23, 24, 25, 27

[FBB+07] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jézéquel. Model-driven engineering for software migration in a large industrial context. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, pages 482–497, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. 90

[FFC21] Francisco Freitas, André Ferreira, and Jácome Cunha. *Refactoring Java Monoliths into Executable Microservice-Based Applications*, page 100–107. Association for Computing Machinery, New York, NY, USA, 2021. xii, 10, 30, 31

[FM17] Chen-Yuan Fan and Shang-Pin Ma. Migrating monolithic mobile application to microservice architecture: An experiment report. In *2017 IEEE International Conference on AI Mobile Services (AIMS)*, pages 109–112, 2017. x, 8

[FSC+21] Augusto Flávio A. A. Freire, Américo Falcone Sampaio, Luis Heustakio L. Carvalho, Otávio Medeiros, and Nabor C. Mendonça. Migrating production monolithic systems to microservices using aspect oriented programming. *Software: Practice and Experience*, 51(6):1280–1307, 2021. xi, xii, 9, 10, 30, 31

[FY79] Brian Foote and Joseph Yoder. Big ball of mud. *Pattern languages of program design*, 4:654–692, 1997/9. viii, 6

[GBMM20] Marx Haron Gomes Barbosa and Paulo Henrique M. Maia. Towards identifying microservice candidates from business rules implemented in stored procedures. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 41–48, 2020. 18, 19, 21, 22, 23, 24, 25, 27

[GCD+17] Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. Towards Recovering the Software Architecture of Microservice-Based Systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 46–53. IEEE, apr 2017. 15

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995. 72

[GKGZ16] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. Service cutter: A systematic approach to service decomposition. In Marco Aiello, Einar Broch Johnsen, Schahram Dustdar, and Ilche Georgievski, editors, *Service-Oriented and Cloud Computing*, pages 185–200, Cham, 2016. Springer International Publishing. xi, 9, 19, 22, 23, 24, 25, 27, 106

[GKMM13] Joshua Garcia, Ivo Krka, Chris Mattmann, and Nenad Medvidovic. Obtaining ground-truth software architectures. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 901–910, 2013. 55

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, dec 1987. 29

[GW99] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis - Mathematical Foundations*. Springer, 1999. 22

[IEE98] IEEE. Ieee standard for software maintenance. *IEEE Std 1219-1998*, pages 1–56, 1998. 15

[JAP13] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. Cloud migration research: A systematic review. *IEEE Transactions on Cloud Computing*, 1(2):142–157, 2013. 3

[JLC+21] Wuxia Jin, Ting Liu, Yuanfang Cai, Rick Kazman, Ran Mo, and Qinghua Zheng. Service candidate identification from monolithic systems based on execution traces. *IEEE Transactions on Software Engineering*, 47(5):987–1007, 2021. xi, 9, 18, 19, 22, 23, 24, 25, 26, 27, 119

[JLZ+18] W. Jin, T. Liu, Q. Zheng, D. Cui, and Y. Cai. Functionality-oriented microservice extraction based on execution trace clustering. In *2018 IEEE ICWS*, pages 211–218, July 2018. 18, 19, 20, 22, 23, 24, 25, 27, 30, 34, 51, 52, 53, 54

[KH18] Holger Knoche and Wilhelm Hasselbring. Using Microservices for Legacy Software Modernization. *IEEE Software*, 35(3):44–49, 2018. 29, 30

[KMK16] Gabor Kecskemeti, Attila Csaba Marosi, and Attila Kertesz. The entice approach to decompose monolithic services into microservices. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 591–596, July 2016. 24

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA explained - the Model Driven Architecture: practice and promise*. Addison Wesley object technology series. Addison-Wesley, 2003. 16

[KWC98] Rick Kazman, Steven S. Woods, and S. Jeromy Carrière. Requirements for integrating software architecture and reengineering models: CORUM II. In *5th Working Conference on Reverse Engineering, WCRE '98, Honolulu, Hawai, USA, October 12-14, 1998*, pages 154–163. IEEE Computer Society, 1998. 14, 91

[KXK+21] Anup K. Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. *Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices*, page 1214–1224. Association for Computing Machinery, New York, NY, USA, 2021. 18, 19, 22, 23, 24, 25, 27

[KXL+20] Anup K. Kalia, Jin Xiao, Chen Lin, Saurabh Sinha, John Rofrano, Maja Vukovic, and Debasish Banerjee. *Mono2Micro: An AI-Based Toolchain for Evolving Monolithic Enterprise Applications to a Microservice Architecture*, page 1606–1610. Association for Computing Machinery, New York, NY, USA, 2020. 19, 20, 22, 23, 24, 25, 27

[KYHM18] Manabu Kamimura, Keisuke Yano, Tomomi Hatano, and Akihiko Matsuo. Extracting Candidates of Microservices from Monolithic Application Code. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 571–580. IEEE, dec 2018. 19, 22, 23, 24, 25, 27

[KZH+20] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kröger. Microservice decomposition via static and dynamic analysis of the monolith. In *2020 IEEE ICSA-C*, pages 9–16, 2020. 19, 22, 23, 24, 25, 27

[LBG+15] Duc Minh Le, Pooyan Behnamghader, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. An empirical study of architectural

change in open-source software systems. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 235–245, 2015. 49, 50

[LCG+15] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic, and Robert Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 69–78, 2015. 49, 50, 51, 119

[LF14a] James Lewis and Martin Fowler. Bounded context, 2014. 100

[LF14b] James Lewis and Martin Fowler. Microservices: a definition of this new architectural term, 2014. vi, viii, 4, 5, 6, 72

[LML20] Chia-yu Li, Shang-Pin Ma, and Tsung-wen Lu. Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System. In *2020 International Computer Symposium (ICS)*, pages 519–524. IEEE, dec 2020. x, 8, 29

[LO20] Jakob Löhnertz and Ana Maria Oprescu. Steinmetz: Toward automatic decomposition of monolithic software into microservices. *CEUR Workshop Proceedings*, 2754:1–8, 2020. 19, 21, 22, 23, 24, 25, 27

[LTV16] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. Towards a technique for extracting microservices from monolithic enterprise systems. *CoRR*, abs/1605.03175, 2016. 18, 19, 22, 23, 24, 25, 26, 27, 28, 106

[MB20a] Ruth W. Macarthy and Julian M. Bass. An empirical taxonomy of devops in practice. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 221–228, 2020. vi, 3

[MB20b] Ben D. Monaghan and Julian M. Bass. Redefining legacy: A technical debt perspective. In Maurizio Morisio, Marco Torchiano, and Andreas Jedlitschka, editors, *Product-Focused Software Process Improvement*, pages 254–269, Cham, 2020. Springer International Publishing. vii, 5, 14

[MCF+20] Tiago Matias, Filipe F. Correia, Jonas Fritzsch, Justus Bogner, Hugo S. Ferreira, and André Restivo. Determining microservice boundaries: A case study using static and dynamic software analysis. In Anton Jansen, Ivano Malavolta, Henry Muccini, Ipek Ozkaya, and Olaf Zimmermann, editors, *Software Architecture*, pages 315–332, Cham, 2020. Springer International Publishing. 19, 22, 23, 24, 25, 27

[MCL17] Genc Mazlami, Jurgen Cito, and Philipp Leitner. Extraction of Microservices from Monolithic Software Architectures. In *2017 IEEE ICWS*, pages 524–531. IEEE, jun 2017. 19, 21, 22, 23, 24, 25, 27, 106

[MG11] Peter Mell and Tim Grance. The nist definition of cloud computing, 09 2011. v, 2, 3, 7

[New19] Sam Newman. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Incorporated, 2019. 30

[NSRS19] Luís Nunes, Nuno Santos, and António Rito Silva. From a monolith to a microservices architecture: An approach based on transactional contexts. In Tomas Bures, Laurence Duchien, and Paola Inverardi, editors, *Software Architecture. ECSA 2019.*, pages 37–52, Cham, 2019. Springer International Publishing. xi, 9, 19, 22, 23, 24, 25, 27

[PFM19] Ilaria Pigazzini, Francesca Arcelli Fontana, and Andrea Maggioni. Tool support for the migration to microservice architecture: An industrial case study. In Tomás Bures, Laurence Duchien, and Paola Inverardi, editors, *Software Architecture - 13th European Conference, ECSA 2019, Paris, France, September 9-13, 2019, Proceedings*, volume 11681 of *Lecture Notes in Computer Science*, pages 247–263. Springer, 2019. 19, 22, 23, 24, 25, 27

[PIIL21] Babak Pourasghar, Habib Izadkhah, Ayaz Isazadeh, and Shahriar Lotfi. A graph-based clustering algorithm for software systems modularization. *Information and Software Technology*, 133:106469, 2021. 49

[Ray18] Partha Pratim Ray. An introduction to dew computing: Definition, concept and implications. *IEEE Access*, 6:723–737, 2018. 2

[RCL09] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A taxonomy and survey of cloud computing systems. In *2009 Fifth International Joint Conference on INC, IMS and IDC*, pages 44–51, 2009. 2

[Ric18a] C. Richardson. *Microservices Patterns: With examples in Java*. Manning Publications, 2018. 31, 34, 37, 48

[Ric18b] Chris Richardson. *Microservices Patterns*. O'Reilly Media, 2018. 29

[Ric22] Chris Richardson. Pattern: Microservice architecture, 2022. vi, vii, 4, 5

[RSSZ18] Florian Rademacher, Jonas Sorgalla, Sabine Sachweh, and Albert Zündorf. Towards a viewpoint-specific metamodel for model-driven development of microservice architecture, 04 2018. 101, 102, 103, 105

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, February 2006. 90, 98, 106

[Sel19] Anfel Selmadji. *From monolithic architectural style to microservice one : structure-based and task-based approaches*. PhD thesis, 2019. Thèse de doctorat dirigée par Seriai, Abdelhak-DjamelDony, Christophe et Bouziane, Hinde Lilia Informatique Montpellier 2019. 16

[Sha15] Anas Shatnawi. *Supporting Reuse by Reverse Engineering Software Architecture and Component from Object-Oriented Product Variants and APIs*. PhD thesis, 2015. Thèse de doctorat dirigée par Seriai, Abdelhak-Djamel et Sahraoui, Houari A. Informatique Montpellier 2015. 16, 17

[SOMS19] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Aymen Saied. Towards automated microservices extraction using muti-objective evolutionary search. In Sami Yangui, Ismael Bouassida Rodriguez, Khalil Drira, and Zahir Tari, editors, *Service-Oriented Computing*, pages 58–63, Cham, 2019. Springer International Publishing. 19, 22, 23, 24, 25, 26, 27

[SPL03] R.C. Seacord, D. Plakosh, and G.A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. SEI series in software engineering. Addison-Wesley, 2003. 14

[SQMC21] Simone Staffa, Giovanni Quattrocchi, Alessandro Margara, and Gianpaolo Cugola. Pangaea: Semi-automated monolith decomposition into microservices. In Hakim Hacid, Odej Kao, Massimo Mecella, Naouel Moha, and Hye-young Paik, editors, *Service-Oriented Computing*, pages 830–838, Cham, 2021. Springer International Publishing. 19, 21, 22, 23, 24, 25, 27

[SRS20] Nuno Santos and António Rito Silva. A complexity metric for microservices architecture migration. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 169–178, 2020. xi, 9, 19, 22, 23, 24, 25, 26, 27

[SSB+20a] Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde-Lilia Bouziane, R. Mahamane, Pascal Zaragoza, and C. Dony. From monolithic architecture style to microservice one based on a semi-automatic approach. *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 157–168, 2020. 43

[SSB+20b] Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde-Lilia Bouziane, Rahina Oumarou Mahamane, Pascal Zaragoza, and Christophe Dony. From monolithic architecture style to microservice one based on a semi-automatic approach. In *2020 IEEE International Conference on Software Architecture, ICSA 2020, Salvador, Brazil, March 16-20, 2020*, pages 157–168. IEEE, 2020. x, xii, 9, 10, 17, 19, 21, 22, 23, 24, 25, 26, 27, 30, 34, 51, 52, 54, 82, 87, 106

[SSBG20] Heimo Stranner., Stefan Strobl., Mario Bernhart., and Thomas Grechenig. Microservice decompositon: A case study of a large industrial software migration in the automotive industry. In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE,*, pages 498–505. INSTICC, SciTePress, 2020. x, 8

[TDD00] Sander Tichelaar, Stéphane Ducasse, and Serge Demeyer. Famix and xmi. In *Proceedings Seventh Working Conference on Reverse Engineering*, pages 296 – 298. IEEE, 02 2000. 99

[TLP20] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. *Microservices Antipatterns: A Taxonomy*, pages 111–128. Springer International Publishing, Cham, 2020. 31, 34, 36, 37, 56

[TS20] Davide Taibi and Kari Systä. A decomposition and metric-based evaluation framework for microservices. In Donald Ferguson, Víctor Méndez Muñoz, Claus Pahl, and Markus Helfert, editors, *Cloud Computing and Services Science*, pages 133–149, Cham, 2020. Springer International Publishing. 19, 23, 24, 25, 27

[wik22] *Software framework*. 01 2022. 35

[WLM+21] Muhammad Waseem, Peng Liang, Gastón Márquez, Mojtaba Shahin, Arif Ali Khan, and Aakash Ahmad. A decision model for selecting patterns and strategies to decompose applications into microservices, 2021. 93

[WRH+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering*. Springer, 2012. 115

[WT04] Zhihua Wen and Vassilios Tzerpos. An effectiveness measure for software clustering algorithms. In *12th International Workshop on Program Comprehension (IWPC 2004), 24-26 June 2004, Bari, Italy*, pages 194–203. IEEE Computer Society, 2004. 49, 50

[ZLD+20] Yukun Zhang, Bo Liu, Liyun Dai, Kang Chen, and Xuelian Cao. Automated Microservice Identification in Legacy Systems with Functional and Non-Functional Metrics. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 135–145. IEEE, mar 2020. xi, 9, 19, 22, 23, 24, 25, 27, 48, 56

[ZSS+21] Pascal Zaragoza, Abdelhak-Djamel Seriai, Abderrahmane Seriai, Hinde-Lilia Bouziane, Anas Shatnawi, and Mustapha Derras. Refactoring monolithic object-oriented source code to materialize microservice-oriented architecture. In Hans-Georg Fill, Marten van Sinderen, and Leszek A. Maciaszek, editors, *Proceedings of the 16th International Conference on Software Technologies, ICSOFT 2021, Online Streaming, July 6-8, 2021*, pages 78–89. SCITEPRESS, 2021. 30, 97, 107

[ZSS+22] P. Zaragoza, A. Seriai, A. Seriai, A. Shatnawi, and M. Derras. Leveraging the layered architecture for microservice recovery. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pages 135–145, Los Alamitos, CA, USA, mar 2022. IEEE Computer Society. 19, 22, 23, 24, 25, 27