# Décomposition des systèmes monolithiques à base d'intelligence artificielle en microservices

par

Hakim GHLISSI

MÉMOIRE PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION DE LA MAÎTRISE
AVEC MÉMOIRE EN GÉNIE LOGICIEL
M. Sc. A.

MONTRÉAL, LE "09 MARS 2026"

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

**PRÉSENTATION DU JURY**

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

Mme. Manel Abdellatif, directrice de mémoire
Département de génie logiciel et des TI à l'École de Technologie Supérieure

Mme. Naouel Moha, codirectrice
Département de génie logiciel et des TI à l'École de Technologie Supérieure

Mr. Sègla Jean-Luc Kpodjedo, président du jury
Département de génie logiciel et des TI à l'École de Technologie Supérieure

Mme. Ghizlane El Boussaidi, membre du jury
Département de génie logiciel et des TI à l'École de Technologie Supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE "25 FÉVRIER 2026"

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

## REMERCIEMENTS

Je souhaite avant tout exprimer ma profonde gratitude et ma reconnaissance envers ALLAH, le Tout-Miséricordieux, pour Ses innombrables bienfaits et pour la guidance qu'Il m'a accordée tout au long de mon parcours. C'est grâce à Sa bienveillance et à Sa sagesse infinie que j'ai eu le privilège d'entreprendre cette aventure. Chaque étape franchie, chaque réussite obtenue, n'a été possible que par Sa grâce et Sa volonté.

Je tiens à adresser mes plus sincères remerciements à mes professeurs et encadrants universitaires, dont la rigueur scientifique, la bienveillance et le dévouement ont profondément marqué mon parcours. Leur accompagnement constant, leurs précieux conseils et leur confiance ont nourri ma curiosité et renforcé mon engagement envers la recherche et l'apprentissage.

À mes directrices, Prof. Manel Abdellatif et Prof. Naouel Moha, je vous exprime ma reconnaissance la plus sincère pour votre patience, votre disponibilité et la richesse de vos orientations. Vos réflexions, votre exigence intellectuelle et vos encouragements m'ont permis d'élargir mes horizons et d'enrichir mes compétences. Ce travail porte l'empreinte de votre encadrement et de vos précieux conseils.

Je souhaite également remercier chaleureusement mes collègues et collaborateurs pour leur esprit d'équipe, leur soutien constant et leurs échanges constructifs. Vos contributions, votre enthousiasme et votre professionnalisme ont rendu cette expérience enrichissante.

À mes amis les plus proches et à ma famille, je vous remercie pour votre soutien indéfectible, votre patience et vos encouragements continus. Votre confiance en moi, même dans les moments de doute, a été une source de motivation inestimable.

Je voudrais adresser un remerciement tout particulier à mon frère Mohamed Wissem et à ma sœur Farah. Votre présence constante, vos encouragements et votre bienveillance ont accompagné chaque étape de ce chemin. Vous avez été un pilier essentiel de ma force et de ma persévérance.

Enfin, je dédie ce travail à mes parents bien-aimés, pour leurs sacrifices, leur amour inconditionnel et leur foi en mes capacités. Vous avez été ma première source d'inspiration et de courage. Rien de ce que j'ai accompli n'aurait été possible sans vos prières, votre soutien et les valeurs que vous m'avez transmises. Cette réussite est avant tout la vôtre.

**Décomposition des systèmes monolithiques à base d'intelligence artificielle en microservices**

Hakim GHLISSI

## RÉSUMÉ

L'essor rapide de l'Intelligence Artificielle (IA) a profondément transformé les architectures logicielles, en plaçant les composants de l'IA au cœur des systèmes modernes. Toutefois, les architectures monolithiques, longtemps dominantes, atteignent rapidement leurs limites face aux besoins de mise à l'échelle, de maintenabilité et d'évolution continue inhérents aux pipelines IA, qui englobent des étapes critiques telles que le prétraitement des données, l'entraînement et le déploiement des modèles. Les approches classiques de migration vers les microservices, initialement conçues pour des systèmes traditionnels dépourvus de composants intelligents, se révèlent inadaptées à ces spécificités, car elles échouent à capturer la complexité et l'interdépendance des flux de données et des processus spécifiques à l'apprentissage automatique.

Pour répondre à ces limites, nous proposons une approche de décomposition des systèmes IA monolithiques en microservices. Notre approche comporte deux volets complémentaires : (1) d'une part, une recherche de la littérature qui met en évidence les forces et les limites des approches existantes de décomposition de systèmes monolithiques en microservices, (2) d'autre part, une approche automatisée de décomposition qui combine l'usage de patrons architecturaux et les modèles de langage (LLMs) afin de guider la décomposition.

Nous validons notre approche sur trois systèmes IA monolithiques et comparons les résultats de décomposition obtenus à deux approches de référence issues de la littérature. Les résultats démontrent l'efficacité de notre méthode pour produire des décompositions modulaires et sensibles aux spécificités des traitements en IA, avec une précision de 84% et un rappel de 65%, surpassant les approches de référence.

**Mots-clés:** Système IA monolithique, architecture en microservices, ingénierie logicielle, décomposition assistée par LLM, décomposition en microservices

**Migration of Machine Learning-Based Systems to Microservices**

Hakim GHLISSI

## ABSTRACT

The rapid rise of artificial intelligence (AI) has profoundly transformed software architectures, placing machine learning components at the core of modern systems. However, monolithic architectures, long considered the standard, quickly reach their limits when faced with the demands of scalability, maintainability, and continuous evolution inherent to AI pipelines, which include critical stages such as data preprocessing, training, and model deployment. Traditional migration approaches toward microservices, originally designed for systems without intelligent components, prove inadequate in this context, as they fail to capture the complexity and interdependencies of data flows and learning processes.

To address these limitations, we propose a decomposition approach explicitly tailored to monolithic AI-based systems. It brings together two complementary aspects : first, a review that highlights the strengths and limitations of existing approaches for transforming monolithic systems into microservices ; and second, an original decomposition approach that combines the use of architectural patterns with the assistance of large language models to identify and group AI pipeline components into cohesive and loosely coupled microservices.

We validate our approach on three monolithic AI-based systems and compare our decomposition results with two baseline approaches from the literature. The results demonstrate the effectiveness of our method in producing modular and AI-aware decompositions, with a precision of 84% and a recall of 65%, outperforming the baseline approaches.

**Keywords:** Monolithic AI system, Microservices architecture, Software engineering, LLM-assisted decomposition, Microservice decomposition

## TABLE DES MATIÈRES

Page

## LISTE DES TABLEAUX

Page

## LISTE DES FIGURES

Page

## LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

| | |
|---|---|
| ÉTS | École de Technologie Supérieure |
| IA | Intelligence Artificielle |
| AST | Arbre Syntaxique Abstrait |
| LLM | Grands Modèles de Langage |
| RNN | Réseau Neuronal Récurrent |
| LSTM | Réseaux de Mémoire à Long Terme |
| CNN | Réseau de Neurones Convolutifs |
| SMQ | Structural Modularity Quality (Qualité de modularité structurelle) |
| CMQ | Conceptual Modularity Quality (Qualité de modularité conceptuelle) |
| IFN | Number of Interfaces (Nombre d'interfaces) |
| NED | Non-Extreme Distribution (Distribution non extrême) |
| ICP | Inter Call Percentage (Pourcentage d'appels inter-services) |
| BCP | Business Context Purity (Pureté du contexte métier) |

# INTRODUCTION

L'Intelligence Artificielle (IA) a connu une expansion fulgurante, transformant en profondeur le paysage des systèmes logiciels. De la recommandation personnalisée aux agents autonomes, les fonctionnalités pilotées par l'IA ne sont plus accessoires : elles constituent désormais le cœur de nombreuses applications, qu'elles soient destinées aux entreprises ou aux utilisateurs finaux. Cette intégration massive de l'intelligence dans les logiciels s'accompagne d'une accélération sans précédent de leur évolution, tant en termes d'échelle que de complexité (Dragoni *et al.*, 2017). Or, cette évolution met en tension les paradigmes traditionnels de programmation et les architectures classiques, initialement conçus pour des environnements plus stables et moins exigeants. Face à la nature dynamique, fortement consommatrice de ressources et en perpétuelle évolution des composants d'apprentissage automatique (en anglais Machine Learning, ML), ces approches atteignent rapidement leurs limites. Les enjeux deviennent alors clairs : assurer la mise à l'échelle, préserver la maintenabilité, garantir des performances fiables et permettre une intégration continue des modèles au sein d'écosystèmes logiciels toujours plus complexes (Kalske, Mäkitalo & Mikkonen, 2018).

L'architecture monolithique a constitué la référence en matière de développement logiciel, offrant une approche unifiée et relativement simple pour concevoir, déployer et maintenir des applications (Abgaz *et al.*, 2023). Toutefois, avec la montée en complexité des systèmes modernes, cette approche montre rapidement ses limites : difficulté de mise à l'échelle, contraintes de maintenance, rigidité technologique et risques accrus liés à un point de défaillance unique. Face à ces défis, l'ingénierie logicielle a progressivement évolué vers des architectures basées sur les microservices (Media, 2020). Celles-ci apportent des avantages particulièrement adaptés aux systèmes intégrant des composants IA, tels qu'un faible couplage, une forte cohésion, la possibilité de déploiement indépendant, une meilleure tolérance aux pannes et la capacité de mettre à l'échelle sélectivement certains modules selon les charges de travail. Ces propriétés sont particulièrement importantes pour les composants d'IA, qui nécessitent souvent des ressources

de calcul intensives, des mises à jour régulières des modèle afin d'éviter qu'une erreur locale n'affecte l'ensemble du système.

Bien que les architectures en microservices offrent des avantages indéniables, la migration d'un système IA monolithique vers ce paradigme soulève des difficultés majeures. Les outils classiques de décomposition, conçus pour des systèmes traditionnels, reposent principalement sur des critères fonctionnels ou analytiques pour délimiter les services (Fowler & Lewis, 2014). Or, ces approches se révèlent inadaptées aux systèmes IA, caractérisés par des composants spécifiques et fortement interdépendants (e.g., collecte et préparation des données, entraînement et réentraînement des modèles, etc.) qui impliquent chacun des contraintes particulières en termes de ressources, d'orchestration et d'infrastructure. À cela s'ajoute la nature évolutive des modèles IA, nécessitant des mises à jour fréquentes, ce qui complique davantage leur découpage par des méthodes standard, initialement pensées pour des systèmes plus statiques. Le rôle des données constitue un autre facteur déterminant. Dans un système basé sur l'apprentissage automatique, elles ne sont pas périphériques ni purement transactionnelles, mais structurent et alimentent l'ensemble du pipeline, l'utilisation des données destinés à l'entraînement. Elles influencent directement la qualité des modèles et les performances globales du système. Sans une stratégie de décomposition explicitement adaptée aux contraintes propres aux systèmes d'IA et à la logique de bout en bout de leurs flux de données, les bénéfices attendus de l'architecture en microservices demeurent largement compromis.

L'objectif principal de ce projet de maîtrise est de proposer une approche intégrée qui accompagne les ingénieurs logiciels et les architectes systèmes dans la migration des systèmes IA monolithiques vers des architectures en microservices robustes, modulaires et évolutives, en mobilisant les styles architecturaux et les patrons de conception appropriés. En articulant les pratiques traditionnelles d'ingénierie logicielle avec les spécificités propres aux systèmes d'IA, nous visons à assister la migration de tels systèmes vers une architecture en microservices.

Dans ce contexte, deux questions de recherche principales guident notre travail :

- **QR1 :** Comment les outils existants pour l'identification de microservices performent-ils lorsqu'ils sont appliqués à des systèmes monolithiques ?
- **QR2 :** Quelle est la performance de notre approche proposée dans la décomposition des systèmes IA en microservices ?

Pour répondre à ces questions de recherche nous proposons une méthodologie structurée, visant à guider la décomposition et la transformation des systèmes IA monolithiques vers des architectures en microservices. Notre contribution s'articule autour de deux axes complémentaires :

- **Une recherche sur les approches de décomposition existantes :** Nous menons une analyse de l'état de l'art afin d'identifier les méthodes et outils existants pour la décomposition des systèmes monolithiques en microservices, en évaluant leur pertinence dans le contexte particulier des systèmes d'IA. Cette recherche permet de dégager les forces et limites de chaque approche et sert de base à une évaluation empirique comparative de leurs performances.
- **Une approche de décomposition adaptée aux systèmes IA monolithiques :** Nous proposons une stratégie spécifiquement conçue pour les systèmes IA monolithiques, englobant les étapes clés allant de l'ingestion et du prétraitement des données jusqu'au déploiement et à la mise à jour des modèles. Cette approche s'appuie sur une analyse sémantique du code monolithique afin d'identifier les concepts centraux et de regrouper les composants en microservices candidats. Nous validons notre approche sur trois systèmes IA monolithiques et comparons nos résultats de décomposition à deux approches de référence issues de la littérature. Les résultats démontrent l'efficacité de notre méthode dans la production de décompositions modulaires et sensibles aux spécificités des systèmes d'IA, avec une précision de 84% et un rappel de 65%, surpassant ainsi les approches de référence. Notons que les résultats de cette contribution ont été publiés à la 23ème "*International Conference on Service-Oriented Computing*" (**ICSOC 2025**), la meilleure conférence en ingénierie des services.

Ce document est structuré en quatre chapitres principaux. Le chapitre 1 présente le contexte général et les fondements conceptuels de la recherche. Il introduit les notions essentielles relatives aux architectures logicielles monolithiques et en microservices, en mettant en évidence leurs différences structurelles et leurs impacts sur les systèmes d'apprentissage automatique. Ce chapitre aborde également les concepts clés liés aux systèmes d'IA, tels que les pipelines de l'apprentissage automatique, les modèles préentraînés et l'émergence des LLMs, afin de situer le travail dans le contexte de l'ingénierie logicielle moderne et de l'intelligence artificielle.

Le chapitre 2 est consacré à la recherche et évaluation empirique des approches de décomposition existantes. Il décrit d'abord la méthodologie de recherche des travaux antérieurs portant sur la décomposition de systèmes monolithiques en microservices, en s'intéressant aussi bien aux approches traditionnelles qu'aux outils récents intégrant des techniques d'IA. Le chapitre présente ensuite une analyse comparative de ces approches à travers une expérimentation empirique, évaluant leurs performances, leurs métriques (telles que la précision, le rappel, la modularité et la cohésion) et leurs limites lorsqu'elles sont appliquées à des systèmes monolithiques d'apprentissage automatique.

Le chapitre 3 constitue la contribution principale de cette recherche. Il introduit notre outil de décomposition de systèmes IA monolithiques, fondé sur l'utilisation de patrons de conception et assisté par des LLMs. Ce chapitre détaille les composantes de l'approche proposée, notamment les étapes d'identification des pipelines IA, d'encodage sémantique du code et de regroupement des classes en microservices candidats.

Le chapitre 4 présente ensuite la validation expérimentale de l'outil sur plusieurs systèmes monolithiques, et compare ses performances à deux outils de décomposition issus de la littérature, afin de démontrer sa pertinence et son efficacité.

Enfin, le chapitre 5 conclut le travail en proposant une synthèse générale des contributions de la recherche, une discussion des limites rencontrées, ainsi que des pistes et des recommandations pour des travaux futurs, notamment en matière d'automatisation et de généralisation des approches de décomposition des systèmes IA complexes.

# CHAPITRE 1

# CONCEPTS FONDAMENTAUX

## 1.1 Introduction

L'évolution des architectures logicielles a profondément marqué la manière dont les systèmes sont conçus et déployés. Historiquement, l'approche la plus intuitive et la plus largement adoptée consistait à développer des applications monolithiques, où l'ensemble du code, de la logique métier et de la base de données était regroupé dans un seul bloc applicatif (Abgaz *et al.*, 2023). Cette organisation offrait une relative simplicité de conception et facilitait le déploiement initial en tant qu'unité unique. Cependant, à mesure que les systèmes ont gagné en ampleur et que les besoins des utilisateurs se sont intensifiés, cette approche a montré ses limites, notamment en termes de flexibilité, de maintenabilité et de capacité à répondre efficacement aux variations de charge (Stephanie Susnjara, 2020). Face à ces défis, un large corpus de travaux de recherche s'est intéressé à des alternatives plus adaptées, conduisant à l'émergence de l'architecture à base de microservices (Saucedo, Rodríguez, Rocha & dos Santos, 2025). Cette dernière se distingue par sa modularité, son évolutivité et sa capacité à faciliter le développement, le déploiement et la maintenance de composants indépendants, répondant ainsi aux exigences croissantes des systèmes modernes (Taibi, Lenarduzzi & Pahl, 2017). Néanmoins, la majorité de ces recherches se sont concentrées sur des systèmes non basés sur l'IA, que nous désignons par systèmes traditionnels, laissant en arrière-plan les spécificités propres aux systèmes intégrant des modèles d'apprentissage automatique. Afin de mieux situer notre contribution, ce chapitre introduit et définit les principaux concepts sur lesquels s'appuie notre travail de recherche.

## 1.2 Architectures monolithiques

Une architecture monolithique, comme illustré dans la figure 1.1, désigne un système logiciel conçu, implémenté, empaqueté et déployé comme une entité unique (Fowler & Lewis, 2014). Sa principale force réside dans sa simplicité, offrant une approche directe pour la conception, le développement et le déploiement d'applications. Dans ce genre d'architecture, l'ensemble de la

base de code est encapsulé dans une seule unité, où tous les composants, tels que la logique métier, le traitement des données et l'interface utilisateur, sont intégrés dans un même exécutable ou artefact de déploiement (Dragoni *et al.*, 2017). Un système de base de données commun soutient l'ensemble, centralisant la gestion des données et facilitant à la fois le débogage et les processus de déploiement. Historiquement, ce schéma architectural a été largement adopté en raison de sa cohérence interne, de sa facilité de mise en œuvre et du faible coût organisationnel qu'il implique dans les phases initiales du développement (Gravanis, Kakarontzas & Gerogiannis, 2022).

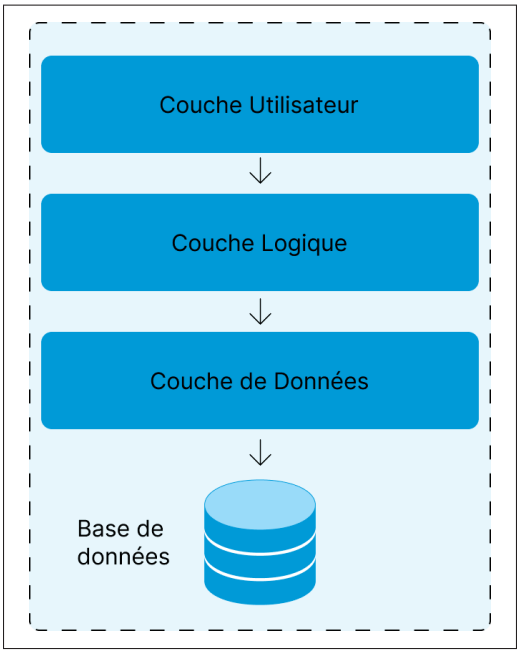


Figure 1.1 Architecture monolithique

Toutefois, les avantages des systèmes monolithiques tendent à s'estomper à mesure que les applications gagnent en taille et en complexité. Bien que cette architecture permette de localiser une part importante des fonctionnalités dans un espace unifié et initialement plus facile à gérer, elle devient rigide et difficile à faire évoluer au fil du temps. Les applications monolithiques matures, développées et enrichies sur plusieurs années, finissent souvent par se fossiliser : l'accumulation de dette technique engendre des structures opaques qui rendent le produit pratiquement impossible à maintenir de manière raisonnable (Fritzsch, Bogner, Zimmermann & Wagner, 2019). Même à

des stades plus précoces, aucun développeur ou architecte ne peut conserver une vision détaillée de tous les composants et de leurs interfaces, ce qui complique la compréhension globale du système et alourdit la maintenance.

En outre, le modèle monolithique impose des contraintes fortes en matière de mise à l'échelle et de flexibilité. Modifier des choix de conception initiaux demande des efforts considérables, et l'adoption de nouvelles technologies s'avère fastidieuse, puisque toute mise à jour requiert bien souvent le redéploiement complet de l'application. Cette structure étroitement couplée accroît également les risques de points uniques de défaillance, compromettant la résilience opérationnelle. Par conséquent, bien que l'architecture monolithique accélère le développement et le déploiement dans les phases initiales, elle constitue un frein à l'évolution, à l'adaptabilité et à la pérennité des systèmes. Pour répondre à ces défis, de nombreuses organisations se sont tournées vers des architectures plus modulaires, et en particulier vers l'architecture en microservices, que nous détaillerons dans la suite (Fritzsch *et al.*, 2019).

## 1.3 Architectures en microservices

Les microservices, représentés dans la figure 1.2, représentent un paradigme architectural dans lequel un système est décomposé en un ensemble de petits services autonomes, chacun encapsulant une capacité métier spécifique et conçu pour être développé, déployé et mis à l'échelle de manière indépendante (Fowler & Lewis, 2014; Newman, 2015, 2019). Contrairement aux architectures monolithiques, qui centralisent la logique métier et la gestion des données dans une unité fortement couplée, l'architecture en microservices privilégie la modularité en attribuant à chaque service la responsabilité de son propre code et de sa propre base de données, garantissant ainsi une forte encapsulation et réduisant les interdépendances (Dragoni *et al.*, 2017). Les services interagissent entre eux via des protocoles de communication légers tels que REST/HTTP ou gRPC, ou encore au moyen de mécanismes asynchrones basés sur des files de messages ou des flux d'événements, ce qui favorise le découplage et permet d'adopter des modèles de cohérence éventuelle (Dragoni *et al.*, 2017).

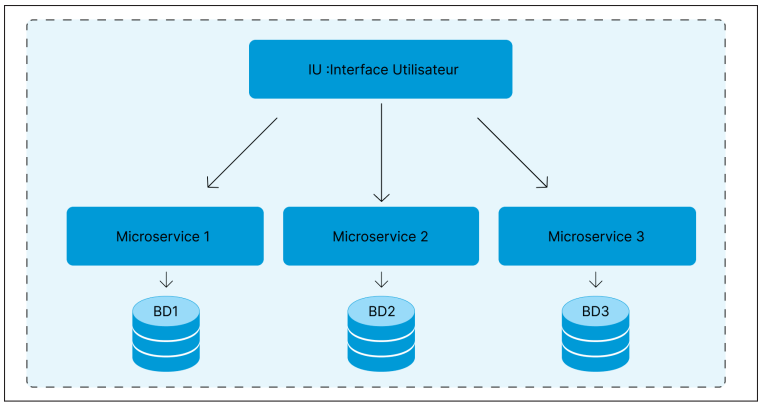


Figure 1.2 Architecture en microservices

L'un des principaux atouts de cette approche réside dans sa capacité à accueillir l'hétérogénéité technologique : chaque service peut être développé dans un langage ou un système de stockage différent, en fonction des besoins spécifiques de son domaine (Fowler & Lewis, 2014). Cette flexibilité, combinée à la possibilité de déployer et de mettre à l'échelle les services de manière indépendante, répond efficacement aux limites des systèmes monolithiques traditionnels, telles que la faible capacité de montée en charge, le manque de résilience et les coûts élevés liés à la coordination des mises à jour. De plus, les microservices s'alignent étroitement avec les pratiques modernes de développement logiciel (Fowler & Lewis, 2014), notamment DevOps et l'intégration/déploiement continus, en permettant des itérations rapides, une meilleure isolation des défaillances et une responsabilisation accrue des équipes autour de leurs services. Cette isolation contribue également à la robustesse globale du système : la panne d'un service ne se propage pas nécessairement aux autres composants, à condition d'appliquer des mécanismes adaptés comme les coupe-circuits, les stratégies de reprise ou les systèmes de cloisonnement. Cependant, ces avantages s'accompagnent d'une complexité accrue liée à la nature distribuée du système : les appels locaux d'un monolithe sont remplacés par des communications réseau, introduisant de la latence, des défaillances partielles et un besoin renforcé d'observabilité à travers la journalisation, la supervision et la traçabilité distribuée. De même, la gestion des données devient plus délicate, les garanties transactionnelles fortes cédant souvent la place à des modèles de cohérence éventuelle et à des mécanismes tels que les sagas ou l'orchestration événementielle.

Pour cette raison, l'adoption des microservices doit être envisagée avec discernement, car si elle apporte modularité, agilité et scalabilité ciblée, elle engendre également un surcoût opérationnel significatif (Kalske *et al.*, 2018). Dans certains contextes, un monolithe modulaire bien conçu peut constituer une alternative plus pragmatique. Malgré ces défis, les microservices connaissent une adoption croissante en raison de leur capacité à répondre aux exigences des systèmes à grande échelle en matière de scalabilité, de maintenabilité et de résilience. Leur pertinence est particulièrement manifeste dans le domaine de l'intelligence artificielle et de l'apprentissage automatique, où ils permettent de séparer clairement les différentes étapes du cycle de vie d'un modèle, de l'ingestion et du prétraitement des données à l'entraînement, la mise en production, l'inférence et la supervision, en offrant la possibilité de faire évoluer chaque composant indépendamment, tout en assurant la cohérence du système grâce à des contrats d'interface bien définis (Fowler & Lewis, 2014). Ainsi, l'architecture microservices constitue une réponse pertinente aux limites des architectures monolithiques, offrant modularité, agilité et robustesse, tout en imposant de relever les défis propres aux systèmes distribués.

## 1.4 Migration des systèmes monolithiques vers des microservices

La décomposition des systèmes monolithiques en microservices constitue une stratégie proactive visant à (1) prolonger la durée de vie des systèmes monolithiques existants et (2) favoriser la réutilisation du logiciel. Elle repose généralement sur des transformations du code afin d'améliorer le code source des systèmes ou de traiter la dette technique accumulée, tout en préservant la fonctionnalité initiale du système (Fritzsch *et al.*, 2019). Dans la littérature existante, plusieurs méthodologies de décomposition ont été proposées et qu'on peut diviser en trois catégories : les approches (1) *descendantes (top-down)*, (2) *ascendantes (bottom-up)* et (3) *hybrides* (Abdellatif *et al.*, 2021).

Dans l'approche descendante, un artefact monolithique de haut niveau, tel qu'un diagramme de cas d'utilisation ou un diagramme d'activités, est décomposé afin de modéliser et d'implémenter des microservices spécifiques. À l'inverse, la stratégie ascendante consiste à analyser le code source d'un système monolithique existant, à extraire les composants réutilisables, et à réécrire

certaines parties du système sous forme de nouveaux microservices. Les approches hybrides combinent les deux méthodes afin d'assurer une transition plus complète et efficace (Filippone, Mehmood, Autili, Rossi & Tivoli, 2023). Dans le cadre de notre recherche, nous nous concentrons exclusivement sur les approches ascendantes, en nous intéressant particulièrement à l'extraction de microservices candidats à partir de l'analyse du code source d'un système monolithique existant.

Le processus de décomposition repose sur trois artefacts principaux (Sellami, Ouni, Saied, Bouktif & Mkaouer, 2022a) :

1. **La phase de collecte des entrées** : elle peut inclure des artefacts basés sur des modèles (cas d'utilisation, modèles UML), des entrées basées sur le code (analyse du code source), des données dynamiques issues de traces d'exécution capturant des scénarios réels, ainsi que des données de versionnage provenant de l'historique des versions et des contributions dans les dépôts (Abgaz *et al.*, 2023).
2. **La phase de prétraitement** : dépendante du type d'entrée, elle est généralement classée en *analyse statique*, *dynamique* ou *sémantique* (Sellami *et al.*, 2022a).
3. **La phase algorithmique** : elle exploite les résultats du prétraitement pour identifier les microservices potentiels. Ces algorithmes peuvent s'appuyer sur des approches d'*apprentissage automatique*, des *algorithmes génétiques* ou encore des *méthodes heuristiques*.

## 1.5 Concepts fondamentaux pour la décomposition assistée par l'IA

Après avoir présenté les architectures monolithiques et microservices ainsi que leurs limites, il est nécessaire d'introduire certains concepts fondamentaux qui serviront de base à notre approche de décomposition assistée par l'intelligence artificielle. Ces notions ne relèvent pas uniquement du génie logiciel traditionnel, mais s'appuient également sur des avancées récentes en intelligence artificielle et en traitement du langage. Dans ce cadre, nous définirons tout d'abord les Arbres Syntaxiques Abstraits (ASTs), utilisés pour représenter la structure hiérarchique du code source. Nous aborderons ensuite la notion de modèles préentraînés, qui offrent une base de

représentations déjà acquises et peuvent être adaptées à différents contextes d'apprentissage. Enfin, nous présenterons les LLMs, dont la capacité à comprendre et générer du code ou du texte naturel joue un rôle central dans notre démarche de décomposition. L'examen de ces concepts permet de mieux situer les fondements théoriques et techniques qui soutiennent notre approche proposée.

### 1.5.1 Arbres syntaxiques abstraits

Un arbre syntaxique abstrait est une représentation arborescente de la structure syntaxique d'un programme, construite selon la grammaire du langage. Contrairement au code source brut, l'AST met en évidence la hiérarchie des éléments (méthodes, déclarations, expressions, instructions conditionnelles, etc.), en distinguant les nœuds non terminaux (par exemple, 'Method Declaration', 'If Statement') et les nœuds terminaux associés à des valeurs concrètes (par exemple, 'public', 'int', 'compare'). Son objectif principal est de capturer la structure syntaxique du code, ce qui permet de dépasser une simple lecture textuelle du code source et d'exploiter la richesse de l'information structurée contenue dans le programme. Dans la littérature en ingénierie logicielle, les ASTs sont largement utilisés pour des tâches telles que la détection de clones de code, la recherche de fragments, la classification ou encore la génération de résumés de code. Leur valeur réside dans leur capacité à préserver des informations structurelles essentielles, même lorsque la similarité lexicale entre deux morceaux de code est faible (Purdue University, School of Electrical and Computer Engineering, 2017).

Dans le cadre de notre travail, l'AST est utilisé comme point d'entrée pour l'extraction des classes depuis la base de code. Concrètement, en analysant la structure hiérarchique produite par l'AST, nous pouvons identifier de manière fiable les déclarations de classes et leurs éléments associés, indépendamment des variations de style ou de format du code source. Cette extraction structurée constitue une étape clé de notre processus de décomposition, car elle garantit que les unités logicielles identifiées correspondent bien à des entités conceptuelles du système, facilitant ainsi leur regroupement ultérieur dans des microservices candidats.

### 1.5.2 Modèles préentraînés

Les modèles préentraînés se sont imposés dans la science du langage et de l'ingénierie logicielle grâce à une architecture devenue incontournable : le Transformateur. Proposé par (Vaswani *et al.*, 2017), ce dernier rompt avec les approches séquentielles classiques (comme les RNNs, LSTMs, CNNs) en reposant exclusivement sur le mécanisme d'auto-attention, qui permet de modéliser les dépendances globales entre tokens sans contrainte de distance. Par définition, un token correspond ici à une unité élémentaire de texte qu'il s'agisse d'un mot, d'un sous-mot ou même d'un caractère servant de base au traitement et à la représentation contextuelle des séquences. Le Transformateur encode chaque séquence comme un ensemble de vecteurs contextualisés (Sellami & Saied, 2025). Cette capacité à capturer en parallèle des relations locales et longues portées a rapidement fait du Transformater une architecture générique et réutilisable dans des contextes très variés, de la traduction automatique aux tâches de compréhension de texte. À partir de ce socle architectural, la communauté a développé une nouvelle génération de modèles préentraînés qui exploitent l'apprentissage auto-supervisé sur d'immenses corpus textuels. BERT (Devlin, Chang, Lee & Toutanova, 2019) a montré l'efficacité de la modélisation du langage masqué pour produire des représentations contextuelles bidirectionnelles riches., tandis que RoBERTa (Liu *et al.*, 2019) a affiné cette approche par une optimisation plus poussée du processus d'entraînement et en exploitant des jeux de données massifs. Ces travaux ont mis en évidence le rôle fondamental de l'encodage : transformer des unités symboliques (mots, sous-mots, tokens) en vecteurs denses exploitables par l'auto-attention, un mécanisme qui permet à chaque élément d'une séquence de pondérer l'importance des autres éléments afin de capter les dépendances contextuelles à longue portée (Devlin *et al.*, 2019). Cette logique a été transposée au domaine des langages de programmation, qui partagent avec les langues naturelles une structure séquentielle tout en ajoutant une dimension syntaxique et sémantique rigide. Dans ce contexte, les Transformateurs présentent un avantage majeur : ils peuvent représenter à la fois les relations locales, comme les liens entre variables, et les relations à longue distance, telles que les interactions entre méthodes ou les liens hiérarchiques entre classes. Là où les approches lexicales ou statistiques échouaient à généraliser, les modèles préentraînés sur le code

apprennent des représentations universelles qui capturent la logique interne et sémantique des programmes, indépendamment des variations superficielles de style ou de nommage.

C'est dans cette perspective qu'a été introduit CodeBERT (Feng *et al.*, 2020), un modèle Transformateur préentraîné conjointement sur du langage naturel et du code source. Basé sur l'architecture RoBERTa-base, CodeBERT compte environ 125 millions de paramètres et est entraîné sur un corpus de 2,1 millions de paires fonction-documentation et 6,4 millions d'extraits de code non annotés couvrant six langages de programmation majeurs. Son originalité réside dans la combinaison de deux objectifs : la modélisation du langage masqué, héritée de BERT, et la détection de remplacement de jetons, qui apprend à distinguer un jeton original d'un jeton substitué par un générateur contextuel. Ce schéma hybride permet d'exploiter pleinement la richesse des données bimodales tout en bénéficiant du volume considérable de données.

Dans le cadre de notre travail, CodeBERT sera mobilisé comme un encodeur robuste, pour offrir des embeddings, qui sont des représentations vectorielles qui traduisent le sens et la structure du texte ou du code (Feng *et al.*, 2020), riches et alignés entre langage naturel et code source. En nous appuyant sur ce modèle, nous disposons d'un modèle préentraîné qui dépasse la simple similarité lexicale et capture les dépendances profondes nécessaires à la décomposition des systèmes monolithiques en microservices.

### 1.5.3 Grand modèle de langage

Les LLMs désignent des modèles d'apprentissage profond de grande taille, généralement fondés sur l'architecture des Transformateurs (Vaswani *et al.*, 2017), et comportant typiquement des dizaines voire des centaines de milliards de paramètres. Entraînés sur des volumes massifs de textes, ces modèles acquièrent une aptitude générale à comprendre et générer du langage naturel bien supérieure à celle de modèles plus petits, avec l'émergence de capacités inédites absentes des anciens modèles (par exemple, apprentissage contextuel à partir de quelques exemples). Grâce à cet entraînement à grande échelle, les LLMs ont démontré des performances remarquables sur un large éventail de tâches de traitement automatique du langage (Minaee *et al.*, 2025).

Du point de vue de leur capacité de résolution de tâches, les modèles de langage ont suivi une évolution marquée par quatre grandes générations comme indique figure 1.3. Les premiers modèles statistiques, reposant sur des approches probabilistes comme les n-grammes, étaient essentiellement utilisés pour soutenir des tâches spécifiques telles que la recherche ou la reconnaissance vocale. Ils ont été supplantés par les modèles neuronaux, capables d'apprendre des représentations plus générales et de réduire la dépendance de l'ingénierie manuelle des caractéristiques. L'émergence des modèles préentraînés a ensuite constitué un tournant majeur en introduisant des représentations sensibles au contexte, ajustables à une large variété de tâches grâce au fine-tuning (Zhao *et al.*, 2025). Enfin, les LLMs incarnent la génération actuelle : en tirant parti de l'effet d'échelle sur les données et le nombre de paramètres, ils se positionnent comme des solveurs génériques, aptes à traiter une diversité de problèmes réels.

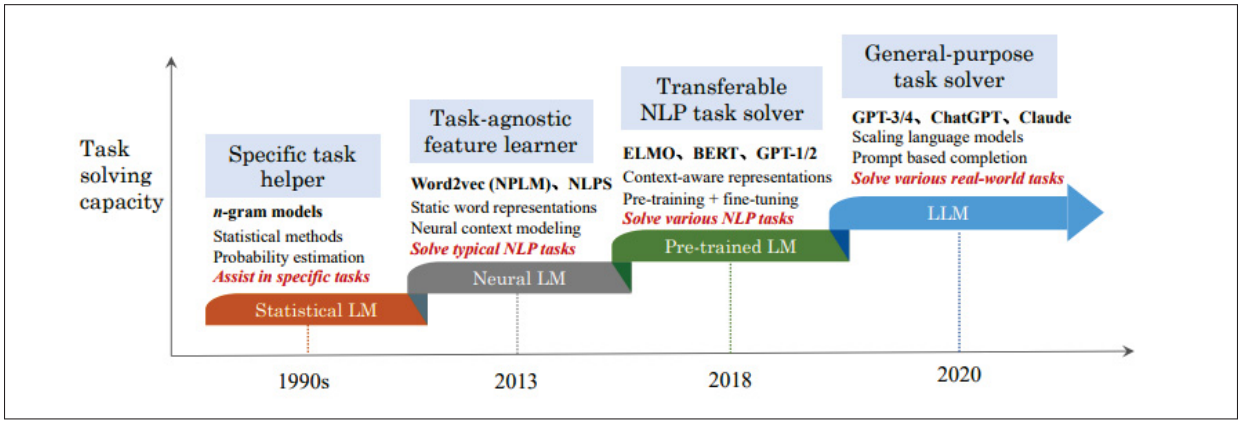


Figure 1.3 Évolution des modèles de langage selon leur capacité de résolution de tâches
Tirée de (Minaee *et al.*, 2025)

Aujourd'hui, ces LLMs constituent l'infrastructure fondamentale de nombreux outils en traitement automatique du langage naturel, se distinguant notamment en génération de texte, traduction, réponse à des questions et résumé automatique, tout en démontrant une remarquable capacité d'adaptation à de nouvelles tâches via l'apprentissage par contexte. Cette polyvalence trouve également un écho dans le domaine du code source, où leurs capacités d'analyse sémantique et de compréhension contextuelle permettent d'automatiser, au moins en partie, l'exploration et l'organisation de bases de code complexes (Zhang *et al.*, 2025). Ils rendent

possible l'identification d'entités logicielles clés, comme les classes centrales et leurs méthodes, la cartographie de leurs dépendances, et l'extraction de composants susceptibles de constituer des microservices, ouvrant ainsi la voie à des applications prometteuses pour la décomposition de systèmes monolithiques vers des architectures orientées microservices.

## 1.6 Conclusion

En somme, ce chapitre a présenté le cadre conceptuel et technologique de notre étude, en abordant les principes fondamentaux relatifs aux architectures monolithiques et en microservices, ainsi que les principales étapes de décomposition. Nous avons également exposé les notions techniques clés mobilisées dans notre approche, telles que l'utilisation des ASTs pour l'extraction de classes, les modèles préentraînés et les LLMs. Cette base théorique et technique constitue le socle nécessaire à la compréhension des approches existantes dans le domaine. Le chapitre suivant sera ainsi consacré à une recherche de la littérature, visant à identifier, analyser et comparer les différentes méthodes et outils de décomposition en microservices proposés dans les travaux antérieurs, afin de situer notre contribution dans un contexte scientifique plus large.

# CHAPITRE 2

# REVUE DE LITTÉRATURE

## 2.1 Introduction

Dans ce chapitre, nous présenterons une recherche consacrée aux différentes approches de migration des systèmes monolithiques vers des architectures à microservices. Nous y exposerons et comparerons les principales méthodes proposées dans les travaux existants, en mettant en évidence leurs principes de fonctionnement, leurs stratégies de décomposition, ainsi que les types d'entrées et d'algorithmes qu'elles mobilisent. À la suite de cette analyse, nous dresserons une liste de six approches open source de décomposition que nous avons sélectionnées pour une évaluation empirique. Cette évaluation vise à mesurer leurs performances sur un ensemble de deux systèmes open source, en nous appuyant sur des métriques d'évaluation issues de la littérature. L'objectif global de ce chapitre est donc double : d'une part, identifier et caractériser les approches de décomposition existantes, et d'autre part, évaluer empiriquement la performance de certaines d'entre elles afin de mieux comprendre leurs capacités, leurs limites et leur applicabilité dans le cadre de la décomposition de systèmes IA plus complexes.

## 2.2 Processus de recherche des approches existantes

Le processus de recherche des approches existantes vise à identifier et analyser les approches pertinentes de décomposition des systèmes monolithiques en microservices. Cette section présente d'abord la stratégie de recherche et les critères retenus, ainsi que la formulation des requêtes utilisées, avant de proposer une liste des outils sélectionnés selon leurs caractéristiques et méthodes.

### 2.2.1 Formulation des requêtes de recherche

La formulation de la requête de recherche constitue une étape essentielle de notre stratégie. Elle a été guidée par les questions de recherche définies dans la phase initiale de notre étude. Le

processus global de recherche des approches est illustré à la figure 2.1. Afin de construire une requête à la fois exhaustive et précise, nous avons procédé à l'extraction des termes principaux associés à nos problématiques, parmi lesquels : *« microservices identification approaches »*, *« refactoring tools »*, *« monolith decomposition »*, *« microservice migration »*, *« system refactoring »* et *« legacy app decomposition »*.

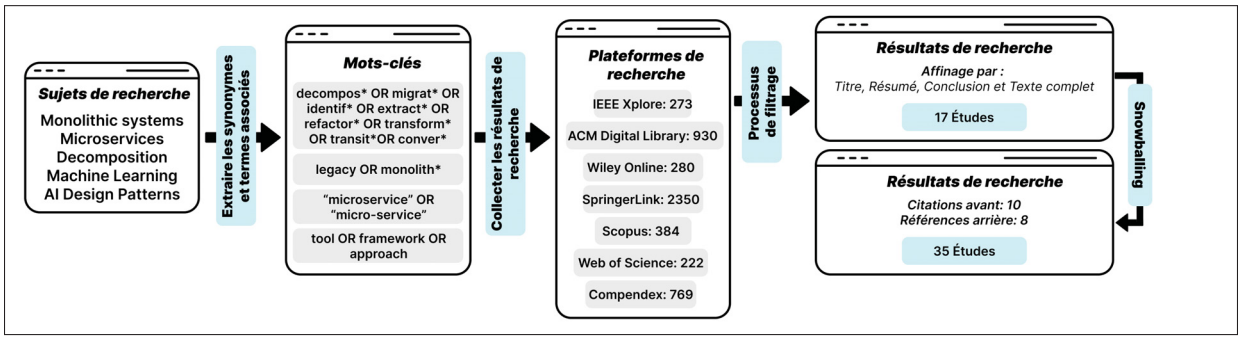


Figure 2.1 Processus de recherche des approches de décomposition

Dans un souci d'élargissement du champ de recherche, nous avons enrichi cette première sélection par l'identification de synonymes et variantes terminologiques, ce qui nous a permis de couvrir un spectre plus large de publications susceptibles d'utiliser des formulations différentes pour décrire des concepts similaires. Cette étape visait à réduire le risque de biais lié à la diversité terminologique existant dans la littérature scientifique.

Sur cette base, nous avons construit une requête combinant les principaux mots-clés et leurs variantes à travers l'utilisation d'opérateurs booléens (`AND`, `OR`). Cela nous a permis de formuler une équation de recherche équilibrée, capable d'inclure des contributions significatives tout en limitant le bruit documentaire. La requête finale utilisée se présente comme suit :

(decompos* OR migrat* OR identif* OR extract* OR refactor* OR transform* OR transit* OR conver*) AND (legacy OR monolith*) AND ("microservice" OR "micro-service") AND (tool OR framework OR approach)

Cette formulation traduit notre volonté de capter l'ensemble des travaux portant sur la décomposition, la refactorisation ou la migration de systèmes monolithiques vers des architectures microservices, tout en tenant compte des différentes dénominations susceptibles d'être utilisées dans les publications académiques. Le tableau 2.1 présente la répartition des résultats initiaux obtenus sur les principales bases de données scientifiques. Les plateformes *SpringerLink* et *ACM Digital Library* ont généré le plus grand nombre de résultats, suivies par *Compendex* et *IEEE Xplore*, tandis que *Web of Science*, *Scopus* et *Wiley Online Library* ont fourni un volume plus restreint. Cette diversité de sources a permis d'assurer une couverture équilibrée des travaux issus à la fois de la recherche académique et des publications industrielles.

Tableau 2.1 Résumé des résultats de recherche

| **Plateformes** | **Nombre total de résultats** |
|---|---|
| IEEE Xplore | 273 |
| ACM Digital Library | 930 |
| SpringerLink | 2350 |
| Compendex | 769 |
| Web of Science | 222 |
| Scopus | 384 |
| Wiley Online Library | 280 |

### 2.2.2 Stratégie de recherche

Notre démarche de recherche s'est appuyée sur une approche systématique et rigoureuse visant à recenser les travaux les plus pertinents dans le domaine de la décomposition des systèmes monolithiques en microservices. En nous inspirant de la méthodologie proposée par Kitchenham *et al.* (Kitchenham, 2004), nous avons défini un ensemble de critères de sélection et d'exclusion permettant de garantir la pertinence scientifique et la reproductibilité de notre étude.

Dans un premier temps comme détaillé dans le tableau 2.2, nous avons restreint notre corpus aux études publiées à partir de 2016, en considérant uniquement des publications en langue anglaise. Ce choix est motivé par deux considérations principales : d'une part, le champ des

microservices est relativement récent et connaît un essor marqué à partir de cette période ; d'autre part, les contributions scientifiques les plus influentes dans le domaine de l'ingénierie logicielle sont majoritairement publiées en anglais. Nous avons également retenu uniquement les travaux publiés dans des revues, conférences ou ateliers scientifiques reconnus, à condition que le texte intégral soit accessible en ligne et que l'étude fournisse des détails suffisants sur les méthodologies de décomposition proposées.

Tableau 2.2 Résumé des critères de la stratégie de recherche

| **Critère** | **Description** |
|---|---|
| Période temporelle | Publications parues à partir de 2016 |
| Langue | Études publiées uniquement en anglais |
| Type de publication | Articles publiés dans des revues, conférences ou ateliers scientifiques |
| Disponibilité | Texte intégral disponible en ligne |
| Pertinence | L'étude propose explicitement une méthodologie ou un outil de décomposition d'un système monolithique vers des microservices |

En somme, les critères que nous avons définis : période temporelle, langue, qualité des sources, disponibilité du texte intégral, et pertinence de la méthodologie proposée, constituent la base de notre stratégie de recherche. Celle-ci nous a permis d'identifier et de retenir un ensemble d'études et d'outils représentatifs des efforts actuels dans le domaine de la décomposition des systèmes monolithiques en microservices.

Après avoir obtenu les premiers résultats issus de notre requête de recherche, nous avons élargi le périmètre d'exploration en appliquant une stratégie de boule de neige (snowballing), à la fois en aval (backward snowballing) et en amont (forward snowballing) (Wohlin, 2014). Cette méthode consiste, d'une part, à analyser systématiquement les références citées dans les articles sélectionnés, et d'autre part, à identifier les travaux qui citent ces mêmes articles. Au total, cette phase de raffinement a permis d'identifier 10 études supplémentaires par snowballing amont et 8 par snowballing aval, conduisant à un corpus final de 35 articles sélectionnés après application des critères d'inclusion et d'exclusion.

L'utilisation de la méthode de boule de neige s'est révélée particulièrement précieuse pour mettre en évidence des études pertinentes qui n'étaient pas apparues dans les résultats initiaux. Par exemple, certains outils ont été découverts uniquement par l'intermédiaire de références croisées, tandis que d'autres études mentionnaient des outils déjà repérés, renforçant ainsi la cohérence et la robustesse de notre corpus. Ce processus itératif a permis de construire une base documentaire plus complète et représentative, en dépassant les limites inhérentes à une recherche strictement fondée sur les requêtes initiales.

### 2.2.3 Approches de décomposition présentées dans la littérature

Dans cette section, nous présentons les différentes approches de décomposition en microservices identifiées. Ces outils sont résumés dans le tableau I-1 dans l'annexe I.

L'analyse de la littérature met en évidence une production croissante de travaux de recherche autour de la migration des systèmes monolithiques vers les microservices. Le tableau I-1 illustre cette diversité en répertoriant les principaux outils et approches proposés au fil des années. Ces contributions se distinguent non seulement par la variété de leurs méthodologies, mais également par les types d'entrées exploitées et les techniques algorithmiques mobilisées. Certaines approches, telles que (Gysel, Kölbener, Giersche & Zimmermann, 2016) ou (Jin *et al.*, 2021), s'appuient sur des analyses statiques ou dynamiques du code et sur les traces d'exécution pour identifier les frontières entre services. D'autres privilégient des méthodes issues de l'intelligence artificielle, comme les réseaux de neurones sur graphes (Desai, Bandyopadhyay & Tamilselvam, 2021; Trabelsi, Moha, Guéhéneuc & Geffard, 2024a), l'apprentissage profond (Sooksatra, Chy, Arju, Cerny & Rivas, 2024) ou encore l'apprentissage par renforcement (Sellami & Saied, 2024). Parallèlement, des outils tels que (Sellami *et al.*, 2022a) adoptent des stratégies d'optimisation évolutionnaire, tandis que d'autres approches (Filippone *et al.*, 2023) s'oriente vers des algorithmes de clustering combinatoire. On observe également des approches hybrides combinant plusieurs sources d'information, comme le code source, l'historique des versions, ou encore des artefacts liés à l'architecture (diagrammes UML, modèles entité-relation).

## 2.3 Choix méthodologique des outils de décomposition en vue de l'évaluation

Suite à cette recherche, nous avons identifié plusieurs approches proposant des mécanismes de décomposition des systèmes monolithiques vers des architectures à base de microservices. Certaines de ces approches s'appuient sur des outils ou prototypes logiciels permettant d'automatiser, au moins partiellement, le processus de décomposition. Dans la section suivante, nous nous concentrons sur ces approches outillées afin d'étudier plus en détail les outils qu'elles proposent. Nous présentons les critères méthodologiques retenus pour leur sélection, ainsi qu'une analyse comparative et empirique de leurs performances respectives.

### 2.3.1 Critères de sélection des outils de décomposition

Dans cette section, après avoir recensé et analysé les différentes approches de décomposition proposées dans la littérature, notre objectif est désormais d'en comparer empiriquement la performance. Pour ce faire, nous entamons d'abord un processus de sélection visant à retenir uniquement les outils pour lesquels une comparaison empirique est effectivement réalisable. Nous présentons ensuite les caractéristiques essentielles de ces outils, en mettant en évidence leurs spécificités respectives en termes d'entrées attendues, d'algorithmes employés, de métriques d'évaluation mobilisées, ainsi que des systèmes de référence utilisés. Cette démarche permet de cadrer de manière rigoureuse l'évaluation empirique et de situer clairement la portée des résultats obtenus.

### 2.3.2 Analyse des approches des outils de décomposition

Dans cette sous-section, nous proposons une analyse des approches adoptées par les outils de décomposition retenus. Dans le cadre de notre étude, nous avons choisi de concentrer notre attention exclusivement sur les approches ascendantes (Section 1.4) et ayant un code source ouvert, en particulier celles qui s'intéressent à l'extraction de microservices candidats à partir d'une application existante. L'objectif est de mettre en lumière les éléments structurants qui différencient ou rapprochent ces outils, notamment la nature des entrées qu'ils exploitent (code

source, graphes de dépendances, traces d'exécution, etc.) et les algorithmes de décomposition mobilisés ainsi que les systèmes de référence utilisés par les outils de décomposition, afin d'apprécier la pertinence et la transférabilité des conclusions tirées. Cette analyse permet ainsi de dégager une vision comparative et critique des approches, préparant le terrain à l'évaluation empirique que nous conduisons par la suite.

À partir des approches identifiées et présentées précédemment, nous avons retenu six approches qui répondent à nos critères de sélection, illustrées dans la table 2.3, que nous analyserons en détail dans la section suivante. Ces critères incluent la période de publication, la langue, le type et la disponibilité des publications, ainsi que leur pertinence vis-à-vis de la problématique de décomposition des systèmes monolithiques. Nous avons également ajouté un critère de disponibilité du code source, essentiel pour pouvoir exécuter les outils sélectionnés, reproduire leurs résultats et assurer la validité empirique des comparaisons.

L'objectif est d'en réaliser une évaluation empirique à l'aide de métriques issues de la littérature, afin de comparer leurs performances et d'apprécier leur efficacité sur un ensemble de systèmes open source représentatifs.

#### 2.3.2.1 Analyse des types d'entrées utilisées par les outils

La littérature analysée (Fritzsch *et al.*, 2019; Abgaz *et al.*, 2023; Saucedo *et al.*, 2025) met en évidence la variété des types d'entrées mobilisés par les approches de décomposition. Malgré cette diversité, il se dégage une convergence claire autour de quatre grandes catégories d'entrées principales : (1) celles fondées sur le code, (2) celles reposant sur des modèles, (3) celles issues des versions ou de l'historique d'évolution du logiciel, et (4) celles dérivées des traces d'exécution. Comme le montre le tableau 2.3, la majorité des outils identifiés reposent sur des entrées basées sur le code (67%), confirmant ainsi la prédominance de cette approche dans la littérature. Les approches exploitant les traces d'exécution demeurent moins fréquentes (33%), tandis que celles fondées sur des modèles restent plus marginales (17%). Ces catégories et leur répartition mettent en lumière une tendance nette en faveur des approches centrées sur

Tableau 2.3 Vue d'ensemble des outils de décomposition en microservices à évaluer

| **Outils** | **Types d'entrées** | **Granularité atomique de sortie** |
|---|---|---|
| Service Cutter : A Systematic Approach to Service Decomposition (Gysel *et al.*, 2016) | Entrées basées sur les modèles | Nano-entité (donnée, opération ou artefact) |
| Fo-SCI : Service Candidate Identification from Monolithic Systems based on Execution Traces (Jin *et al.*, 2021) | Entrées basées sur les traces d'exécution | Niveau classe |
| Mono2Micro : A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices (Kalia *et al.*, 2021) | Entrées basées sur les traces d'exécution et le code | Niveau classe |
| CO-GCN : Graph Neural Network to Dilute Outliers for Refactoring Monolith Application (Desai *et al.*, 2021) | Entrées basées sur le code | Niveau classe |
| MSExtractor : Improving Microservices Extraction Using Evolutionary Search (Sellami *et al.*, 2022a) | Entrées basées sur le code | Niveau classe |
| MAGNET : Method-based Approach using Graph Neural Networks for Microservices Identification (Trabelsi *et al.*, 2024a) | Entrées basées sur le code | Niveau méthode |

le code, tout en soulignant l'existence de stratégies alternatives qui enrichissent la diversité méthodologique. Dans les sections qui suivent, nous examinerons plus en détail les usages spécifiques de chacun de ces types d'entrées, en évaluant leur pertinence ainsi que leur efficacité pour soutenir le processus de décomposition en microservices.

**Entrées basées sur le code**

En se concentrant sur les approches reposant sur l'analyse du code, on observe que les outils de décomposition exploitent fréquemment le code source du système afin d'en extraire les informations critiques nécessaires à leurs algorithmes. Cette démarche offre une compréhension globale et détaillée de la structure interne du système, révélant les relations et dépendances complexes entre classes et modules. L'analyse du code permet ainsi de mettre en évidence

des interactions essentielles à l'identification précise des microservices candidats. De ce fait, les entrées basées sur le code constituent l'approche la plus couramment employée dans de nombreuses méthodologies et outils de décomposition, comme le soulignent (Abgaz *et al.*, 2023) et (Saucedo *et al.*, 2025).

Au-delà de l'analyse traditionnelle du code source, certains travaux élargissent leur périmètre en intégrant des artefacts liés à l'infrastructure, tels que les manifestes Kubernetes ou les fichiers Docker Compose (Maggi, Verdecchia, Scommegna & Vicario, 2024; Rademacher, Sachweh & Zündorf, 2020; Granchelli *et al.*, 2017). L'examen de ces artefacts fournit des informations précieuses sur les modalités de déploiement et d'orchestration du système. Les manifestes Kubernetes décrivent notamment la configuration des ressources et leurs interactions, offrant ainsi des indices pertinents pour identifier des composants faiblement couplés susceptibles d'être transformés en microservices (Maggi *et al.*, 2024). De manière analogue, les fichiers Docker Compose renseignent sur la définition des services, leurs dépendances et leurs schémas de communication, contribuant à une meilleure compréhension du comportement dynamique et de l'organisation structurelle du système. Nous avons choisi de mentionner ces approches afin d'illustrer jusqu'où l'analyse du code source peut être étendue, bien qu'elles ne fassent pas partie de notre évaluation empirique.

L'analyse du code source peut être abordée selon deux perspectives distinctes (Sellami *et al.*, 2022a). Certains outils s'appuient sur l'analyse statique afin d'extraire les dépendances et les relations entre composants, constituant ainsi une première base de compréhension de l'architecture du système. D'autres la complètent par une analyse sémantique, qui dépasse la simple extraction de dépendances pour fournir une vision plus fine et globale, renforçant in fine l'efficacité du processus de décomposition (Saucedo *et al.*, 2025).

L'analyse statique permet notamment d'identifier les composants de la base de code, de révéler les dépendances, de détecter des duplications ainsi que des segments de code inutilisés ou redondants. Elle ne se limite donc pas à la seule extraction des dépendances : elle constitue également un outil précieux pour améliorer la qualité du code en mettant en évidence la structure

des éléments qui pourront être optimisés dans les phases ultérieures. Certains outils recourent par exemple à la bibliothèque Soot [1] comme dans l'outil proposé par (Desai *et al.*, 2021), qui permet d'analyser des systèmes Java, de construire des graphes d'appels et de réaliser d'autres opérations avancées. D'autres intègrent Spoon (Pawlak, Monperrus, Petitprez, Noguera & Seinturier, 2015) par exemple l'outil proposé par (Lourenço & Silva, 2023), une bibliothèque open source destinée à l'analyse et la transformation du code Java. Toutefois, l'analyse statique pose également des défis : le traitement de grandes bases de code peut s'avérer coûteux en ressources et générer des imprécisions, notamment en raison de son incapacité à capturer les aspects sémantiques des éléments identifiés.

Pour pallier ces limites, plusieurs approches recourent à l'analyse sémantique (Sellami *et al.*, 2022a; Brito, Cunha & Saraiva, 2021), seule ou en complément de l'analyse statique (Sellami *et al.*, 2022a). Cette intégration permet d'extraire la logique métier implicite contenue dans la base de code, facilitant ainsi une décomposition orientée domaine fondée sur les règles métier implémentées. Elle contribue également à l'identification d'unités cohésives et d'éléments étroitement liés grâce à l'interprétation sémantique du code, en établissant des correspondances entre éléments sur la base de leurs fonctionnalités, ce qui améliore la précision et la pertinence du processus de décomposition. Néanmoins, cette approche présente certaines contraintes. Elle est particulièrement efficace lorsque le code est accessible et bien maintenu, mais peut perdre en fiabilité en présence de code mal structuré, obsolète ou peu documenté, compromettant ainsi la justesse de l'extraction des éléments. De plus, l'analyse du code repose sur une représentation statique ou sémantique du système, ce qui peut conduire à négliger les comportements dynamiques et les interactions à l'exécution susceptibles d'influencer l'identification des microservices. Enfin, le traitement de larges bases de code exige des ressources considérables, tant en puissance de calcul qu'en temps, ce qui limite son application dans des contextes nécessitant une identification rapide ou en temps réel. À cela s'ajoute la dépendance forte au langage de programmation, qui impose souvent des adaptations ou le développement de solutions spécifiques à chaque

[1] Soot GitHub Repository : https://github.com/soot-oss/soot

langage, restreignant la généricité et la transférabilité de cette approche dans des environnements technologiques hétérogènes.

**Entrées basées sur les traces d'exécution :**

Une forme récurrente d'entrée mobilisée dans les approches de décomposition en microservices repose sur l'exploitation des traces d'exécution en temps réel. Cette méthode est largement utilisée pour capturer les relations et dépendances entre les différents éléments d'un système. L'examen des appels et des dépendances générés lors de scénarios d'exécution spécifiques permet de regrouper les microservices candidats en fonction des interactions simultanées observées, ce qui accroît la précision de la classification (Abgaz *et al.*, 2023). En outre, cette approche a l'avantage d'intégrer les interactions des utilisateurs, en se rapprochant de leur expérience réelle. Elle reflète ainsi les schémas d'utilisation actuels tout en mettant en évidence les différentes fonctionnalités du système du point de vue de l'utilisateur final (Jin *et al.*, 2021).

Une caractéristique distinctive de cette approche réside dans son indépendance vis-à-vis du code source. En traitant le système comme une boîte noire, seules les invocations et les appels produits lors de l'exécution sont pris en compte. Cette perspective permet de dégager une vue d'ensemble du système, sans avoir besoin d'accéder directement au code source, et de tracer des frontières entre microservices candidats de manière plus efficace sur la base des usages réels observés.

Cependant, cette méthode présente également certaines limites. La principale difficulté réside dans la collecte de données : l'obtention de traces d'exécution représentatives peut être complexe, ce qui limite parfois la capacité à généraliser les résultats (Abgaz *et al.*, 2023). De plus, les traces collectées ne couvrent pas nécessairement l'intégralité des scénarios d'exécution possibles, entraînant une représentation partielle du comportement global du système. Enfin, la gestion de cas particuliers reste problématique : certaines situations non triviales, comme celles liées aux fonctionnalités de test ou aux comportements exceptionnels, requièrent un traitement spécifique qui dépasse le cadre des exécutions régulières.

**Entrées basées sur les modèles :**

Les entrées basées sur les modèles adoptent une approche plus abstraite, en se concentrant sur des artefacts de conception tels que les descriptions architecturales, les diagrammes UML, les cas d'utilisation ou encore les modèles entité-association utilisés pour structurer le système (Abgaz *et al.*, 2023; Akkaya & Ovatman, 2022) comme c'est notamment le cas dans l'outil *ServiceCutter* qui exploite ce type de modèles pour guider la décomposition en microservices. Ce niveau d'abstraction constitue un atout majeur pour les organisations en phase de transition, car il offre une vue d'ensemble au niveau architectural et facilite la conception raffinée des microservices. Cette approche se révèle particulièrement efficace dans les scénarios dits de nouvelle conception (greenfield), où le développement d'un système commence à partir de zéro, sans héritage d'un code existant. Dans ce contexte, l'architecture peut être pensée dès le départ selon une logique de microservices, ce qui permet d'anticiper les dépendances et de structurer le système dès les premières étapes de conception. Cependant, son application dans les scénarios dits de restructuration de systèmes existants (brownfield) où les développeurs partent d'une base logicielle déjà en place pour la décomposer progressivement en microservices peut s'avérer plus complexe.

En effet, bien que cette approche favorise une conception claire et cohérente des architectures microservices, elle présente certaines limites. La principale réside dans la précision, souvent restreinte par le haut niveau d'abstraction des modèles. De plus, à mesure que les applications évoluent et s'élargissent, maintenir ces modèles à jour peut devenir particulièrement exigeant, ce qui compromet à terme la fiabilité et la pertinence des informations qu'ils véhiculent.

**Entrées basées sur les versions :**

Certaines approches exploitent les entrées basées sur les versions, souvent en complément d'autres types d'entrées (Kalia *et al.*, 2021). Cette méthode s'appuie sur l'historique des versions d'un projet, incluant les journaux de commits, l'historique des contributeurs et autres artefacts associés, tels que ceux disponibles sur GitHub (Lourenço & Silva, 2023). L'intérêt de cette approche réside dans sa capacité à restituer la dynamique du processus de développement du

code, en retraçant sa construction incrémentale et en identifiant les modules ayant bénéficié de contributions collaboratives.

Parmi ses avantages, on relève notamment la possibilité de générer des microservices candidats adaptés au travail en équipe, grâce à l'analyse des segments de code modifiés conjointement. Cette méthode présente également l'avantage d'être peu coûteuse et relativement simple à mettre en œuvre pour collecter des données d'entrée (Lourenço & Silva, 2023). Toutefois, elle souffre de plusieurs limites inhérentes. L'historique de versions ne permet pas une compréhension sémantique du code et doit donc être combiné avec des entrées basées sur le code afin de pallier cette lacune (Lourenço & Silva, 2023). En outre, l'efficacité de cette approche dépend fortement des pratiques de gestion de versions adoptées par l'équipe de développement. Dans le contexte des projets open source, la diversité des contributeurs peut entraîner une variabilité accrue et parfois des résultats divergents. Enfin, cette dépendance aux habitudes de validation de version et à une perspective centrée sur l'équipe constitue un défi lorsque les contributions au code proviennent de multiples sources, au-delà d'un seul noyau de développement.

#### 2.3.2.2 Analyse des algorithmes de décomposition

La littérature met en évidence une diversité de méthodes visant à identifier les microservices candidats, chacune adoptant une approche spécifique du processus de décomposition. De manière générale, ces approches peuvent être regroupées en trois grandes catégories. La première, avec environ 33% des outils, s'appuie sur des techniques d'apprentissage automatique, notamment l'utilisation de réseaux de neurones sur graphes (Desai *et al.*, 2021; Trabelsi *et al.*, 2024a). La deuxième mobilise des algorithmes génétiques (33%), où le processus d'identification est guidé par des fonctions d'évaluation centrées principalement sur l'optimisation du couplage et de la cohésion entre les candidats proposés (Sellami *et al.*, 2022a; Jin *et al.*, 2021). Enfin, une troisième catégorie, représentant également environ 33% des approches (Sellami *et al.*, 2022a; Jin *et al.*, 2021), repose sur des techniques de regroupement (clustering), cherchant à révéler des ensembles cohésifs de composants à partir de critères structurels ou comportementaux. Dans ce qui suit, nous allons détailler chaque type d'algorithme de décomposition.

**Approches fondées sur l'apprentissage automatique**

La littérature propose un ensemble varié d'approches pour l'identification des microservices candidats. Parmi celles-ci, un sous-groupe se distingue par l'utilisation d'algorithmes d'apprentissage automatique.

- **CoGCN :** Desai *et al.* ont introduit CoGCN (Desai *et al.*, 2021), un outil conçu pour décomposer des systèmes monolithiques en microservices en intégrant une méthodologie fondée sur la détection d'« outliers ». L'approche distingue deux types d'outliers : les outliers structurels et les outliers d'attributs. Les premiers correspondent à des classes présentant un couplage élevé avec différents clusters, tandis que les seconds concernent des classes dont leurs occurrences dans les traces d'utilisation sont plus proches d'autres regroupements. La méthodologie repose sur une transformation du système en un graphe, où les nœuds représentent les classes et les arêtes leurs relations. Un réseau de neurones sur graphes est appliqué à cette représentation afin d'assigner les classes à des microservices tout en minimisant l'impact des outliers, ce qui améliore la qualité de la décomposition.
- **MAGNET :** Trabelsi *et al.* (Trabelsi *et al.*, 2024a) ont proposé MAGNET, une approche automatisée pour la décomposition des systèmes monolithiques en microservices candidats en considérant les méthodes comme niveau de décomposition. La méthodologie combine une analyse statique et sémantique du code source. Le processus débute par la construction d'un graphe de dépendances enrichi à l'aide de MoDisco[2], capturant les dépendances inter-méthodes à travers un métamodèle détaillé du code. Parallèlement, des représentations vectorielles des classes et méthodes sont extraites afin de saisir leurs dépendances fonctionnelles et contextuelles. Ces représentations sont ensuite traitées par un réseau de neurones sur graphes, qui assigne les clusters et regroupe les méthodes en unités cohésives susceptibles de constituer des microservices.

Enfin, au-delà des approches non supervisées, la littérature met en avant le potentiel des approches supervisées et par apprentissage par renforcement (Abgaz *et al.*, 2023). Le premier type s'appuie sur des données annotées afin d'entraîner des modèles capables de prédire directement les

[2] https ://projects.eclipse.org/projects/modeling.modisco

frontières optimales de microservices, souvent guidés par l'expertise humaine. L'apprentissage par renforcement, quant à lui, ajuste progressivement les frontières des microservices par interaction avec l'environnement, permettant une adaptation dynamique aux besoins évolutifs du système.

**Approches fondées sur les algorithmes génétiques**

Les algorithmes génétiques constituent un second axe de recherche majeur. Leur intérêt réside dans leur capacité à optimiser simultanément plusieurs fonctions objectives et à explorer en parallèle de multiples solutions candidates, accélérant ainsi la recherche dans des systèmes de grande taille et fortement interconnectés (Jin *et al.*, 2021).

- **FoSCI :** Jin *et al.* (Jin *et al.*, 2021) ont proposé FoSCI, une approche exploitant les traces d'exécution collectées par Kieker, un outil spécialisé dans le suivi de l'exécution. L'identification des microservices repose sur l'algorithme NSGA-II. L'algorithme génétique regroupe des atomes fonctionnels, définis comme des ensembles de classes apparaissant fréquemment dans les mêmes traces d'exécution. Deux fonctions objectives sont optimisées : la connectivité structurelle (mesurant l'intensité des interactions entre classes dans un même groupe) et la connectivité conceptuelle (évaluant la cohésion sémantique des classes d'un groupe à partir de la similarité lexicale de leurs identifiants). Les atomes fonctionnels sont initialement générés par un algorithme de regroupement hiérarchique appliqué aux traces d'exécution. L'approche se conclut par une étape d'identification des interfaces de microservices, définissant ainsi la décomposition finale.
- **MSExtractor :** Sellami *et al.* (Sellami *et al.*, 2022a) ont introduit MSExtractor, un outil de décomposition fondé sur une approche de recherche évolutionnaire multi-objectifs. Le code source est analysé statiquement et sémantiquement, puis injecté dans l'algorithme IBEA (Indicator-Based Evolutionary Algorithm). Ce dernier a démontré de meilleures performances que d'autres algorithmes évolutionnaires de référence, tels que NSGA-II (Non-dominated Sorting Genetic Algorithm II) et SPEA2 (Strength Pareto Evolutionary Algorithm 2). IBEA (Indicator-Based Evolutionary Algorithm) optimise simultanément trois fonctions objectives :

le couplage, la cohésion et la granularité. L'optimisation conjointe de ces critères permet de produire des microservices candidats équilibrés en termes de taille, de fonctionnalités et d'interdépendances.

**Approches fondées sur le clustering**

Enfin, une troisième famille d'approches s'appuie sur des algorithmes de clustering pour l'extraction et la migration vers des microservices.

- **Mono2Micro :** Kalia *et al.* (Kalia *et al.*, 2021; Kalia *et al.*, 2020) ont développé Mono2Micro, un outil conçu pour les systèmes Java monolithiques. L'approche utilise un algorithme de regroupement hiérarchique et s'appuie sur deux types d'entrées : l'analyse statique du code source et les traces d'exécution collectées dans différents scénarios. Cette combinaison permet d'exploiter deux stratégies complémentaires : la décomposition spatiale, basée sur les dépendances et les cas d'utilisation extraits du code, et la décomposition temporelle, construite à partir des interactions dynamiques observées dans les traces d'exécution, représentées sous forme d'arbres de contexte d'appels.
- **Service Cutter :** Gysel *et al.* (Gysel *et al.*, 2016) ont proposé Service Cutter, l'un des premiers outils développés spécifiquement pour l'identification des microservices. Il met en œuvre deux algorithmes de regroupement : l'algorithme déterministe de Girvan-Newman et l'algorithme non déterministe ELP. Alors que les méthodes déterministes garantissent des résultats stables pour les mêmes entrées (notamment en termes de nombre de microservices et de critères de couplage), les approches non déterministes peuvent produire des résultats variables en raison de l'aléa inhérent à leur fonctionnement. Service Cutter exploite comme entrée des modèles entité-association, représentant la structure et les relations du système. Il s'appuie sur 16 critères de couplage, intégrant des aspects de compatibilité, de contraintes et de communication, afin d'identifier des microservices candidats cohésifs.

#### 2.3.2.3 Analyse des métriques utilisées pour l'évaluation

Cette section s'attache à examiner les métriques mobilisées dans la littérature pour évaluer les outils d'identification de microservices. La majorité des travaux tendent à regrouper ces métriques en trois grandes catégories : (1) celles liées à l'indépendance fonctionnelle, (2) celles portant sur la modularité, et (3) celles relatives à l'indépendance vis-à-vis de l'évolutivité. L'objectif de cette analyse est de mettre en évidence la pertinence de ces métriques dans l'évaluation des performances des outils, en explicitant leur rôle et leur finalité.

- **Structural Modularity Quality (SM/SMQ)**

  La métrique *Structural Modularity* (SM) évalue la qualité des partitions produites en examinant à la fois leur cohésion interne et leurs connexions externes (Jin *et al.*, 2021). La cohésion interne correspond au degré de connectivité entre les éléments d'un même microservice candidat : une forte intra-connectivité reflète une décomposition efficace où les classes partagent bibliothèques, fonctions ou structures de données communes (Mancoridis, Mitchell, Rorres, Chen & Gansner, 1998). La cohésion est quantifiée via la variable $scoh_i$, tandis que le couplage entre partitions est mesuré par *scop*, qui représente le degré de dépendance entre services.

  Soit $\mathcal{M} = \{S_1, S_2, ..., S_N\}$ la partition du système en $N$ microservices candidats, où chaque $S_i$ représente un ensemble d'entités logicielles (par exemple des classes ou modules). La formule générale pour le calcul de l'indice de modularité structurelle (SM), proposée par Kalia et al. (Kalia *et al.*, 2021), est présentée à l'équation (2.1) :

$$\mathrm{SM}_{\mathrm{scoh}},\ \mathrm{SM}_{\mathrm{scop}} = \mathrm{ComputeSM}(\mathcal{M}), \tag{2.1}$$

  L'équation (2.2) définit ensuite le calcul du score global SM :

$$\mathrm{SM} = \frac{1}{N}\sum_{i=1}^{N} \mathrm{scoh}_i \ - \ \frac{1}{\frac{N(N-1)}{2}}\sum_{i\neq j} \mathrm{scop}_{i,j}, \tag{2.2}$$

  Les quantités élémentaires utilisées dans ce calcul sont données par l'équation (2.3) :

$$\text{scoh}_i = \frac{u_i}{N^2}, \qquad \text{scop}_{i,j} = \frac{\epsilon_{i,j}}{2\,|N_i - N_j|}. \tag{2.3}$$

Ici, $u_i$ représente le nombre de connexions internes au microservice $i$, tandis que $\epsilon_{i,j}$ correspond aux arêtes reliant les microservices $i$ et $j$. De plus, $N_i$ et $N_j$ indiquent respectivement le nombre d'entités dans les microservices candidats $i$ et $j$. Comme indiqué dans (Kalia *et al.*, 2021), des valeurs plus élevées de SM reflètent une meilleure décomposition.

- **Conceptual Modularity Quality (CMQ)**

  La métrique *Conceptual Modularity Quality* (CMQ) mesure la modularité du système à partir de ses relations conceptuelles (Razzaq & Ghayyur, 2023). Contrairement à SM, qui s'appuie sur des relations structurelles, CMQ évalue la cohésion sémantique ($ccoh_i$) et le couplage conceptuel ($ccop_{i,j}$) des microservices, en considérant qu'un lien existe entre deux entités dès lors qu'un chevauchement lexical apparaît entre leurs termes textuels (Abgaz *et al.*, 2023). La définition générale de CMQ est présentée dans l'équation (2.4) :

$$CMQ = \frac{1}{N}\sum_{i=1}^{N} ccoh_i \; - \; \frac{1}{\frac{N(N-1)}{2}} \sum_{i \neq j} ccop_{i,j}, \tag{2.4}$$

  Les composantes élémentaires employées dans ce calcul sont précisées à l'équation (2.5) :

$$ccoh_i = \frac{u_i}{N^2}, \qquad ccop_{i,j} = \frac{\epsilon_{i,j}}{2(N_i - 1)(N_j - 1)}. \tag{2.5}$$

  Des valeurs élevées de CMQ indiquent une architecture conceptuellement cohésive et mieux structurée, comme souligné dans les travaux de (Razzaq & Ghayyur, 2023; Abgaz *et al.*, 2023).

- **Number of Interfaces (IFN)**

  La métrique *Interface Number* (IFN) quantifie le nombre d'interfaces exposées par chaque microservice (Jin *et al.*, 2021; Kalia *et al.*, 2021). Un IFN faible reflète un périmètre fonctionnel clair, conforme au principe de responsabilité unique, réduisant ainsi la complexité, les dépendances inutiles et améliorant la maintenabilité. La définition générale de IFN est présentée dans l'équation (2.6) :

$$IFN = \frac{1}{N} \sum_{j=1}^{N} ifn_j, \tag{2.6}$$

tandis que le calcul individuel de chaque microservice $j$ est donné par l'équation (2.7) :

$$ifn_j = \frac{|I_j|}{j}, \tag{2.7}$$

où $I_j$ correspond à l'ensemble des interfaces publiées par le microservice $j$.

- **Non-Extreme Distribution (NED)**

  La métrique *Non-Extreme Distribution* (NED) vise à vérifier que la distribution des entités entre microservices reste équilibrée, en évitant des clusters trop petits ou trop volumineux (Kalia *et al.*, 2021; Desai *et al.*, 2021). La définition générale de NED est présentée dans l'équation (2.8) :

$$NED = 1 - \frac{\sum_{k=1,\ k \text{ not extreme}}^{K} n_k}{|V|}, \tag{2.8}$$

  où $n_k$ désigne le nombre d'entités dans le cluster $k$ et $V$ l'ensemble des entités du système. Les clusters dits « extrêmes » correspondent à des tailles hors de l'intervalle $[5,\ 100]$. Comme indiqué dans (Kalia *et al.*, 2021; Desai *et al.*, 2021), des valeurs faibles de $1 - NED$ sont préférables, car elles traduisent une meilleure homogénéité dans la taille des microservices candidats.

- **Inter Call Percentage (ICP)**

  L'*Inter-Partition Call Probability* (ICP) mesure l'intensité des appels entre partitions, fournissant un indicateur du couplage dynamique (Kalia *et al.*, 2020, 2021). La définition de ICP pour deux partitions $i$ et $j$ est donnée par l'équation (2.9) :

$$\text{ICP}_{i,j} = \frac{c_{i,j}}{\sum_{\substack{i=1,\ j=1 \\ i \neq j}} c_{i,j}}, \tag{2.9}$$

  où $c_{i,j}$ représente le volume d'appels effectués entre les partitions $i$ et $j$. Comme discuté dans (Kalia *et al.*, 2020, 2021), un ICP faible indique une architecture faiblement couplée, donc plus modulaire et plus scalable.

- **Business Context Purity (BCP)**

  La métrique *Business Context Purity* (BCP) évalue la pureté fonctionnelle d'un microservice en mesurant l'entropie des contextes métier auxquels appartiennent ses classes (Kalia *et al.*, 2020). La définition générale de BCP est présentée dans l'équation (2.10) :

$$BCP = \frac{1}{M} \sum_{i=1}^{M} \mathrm{bcp}_i, \tag{2.10}$$

  tandis que le calcul de la pureté du microservice $i$ est détaillé dans l'équation (2.11) :

$$\mathrm{bcp}_i = -\sum_{j=1}^{m_i} \frac{1}{m_i} \log\left(\frac{1}{m_i}\right), \tag{2.11}$$

  où $M$ désigne le nombre total de partitions candidates et $m_i$ le nombre de contextes métiers auxquels est associé le microservice $i$.

  Comme souligné dans (Kalia *et al.*, 2020), un BCP faible reflète une meilleure cohésion fonctionnelle et une spécialisation plus claire au sein des microservices.

**Interprétation des métriques :**

- **Structural Modularity Quality (SM/SMQ)** : Mesure la cohésion interne et le couplage externe des microservices. Des valeurs élevées indiquent une meilleure décomposition et un couplage réduit.
- **Conceptual Modularity Quality (CMQ)** : Évalue l'alignement conceptuel des microservices. Des valeurs élevées reflètent une forte cohésion interne et un couplage réduit.
- **Number of Interfaces (IFN)** : Mesure le degré d'exposition des interfaces publiées par les microservices. Des valeurs faibles suggèrent des services bien définis et centrés sur un objectif précis.
- **Non-Extreme Distribution (NED)** : Évalue l'équilibre dans la taille des partitions. Des valeurs faibles indiquent une répartition plus homogène, conduisant à des clusters plus maintenables et scalables.
- **Inter Call Percentage (ICP)** : Quantifie les flux de communication entre partitions. Des valeurs faibles traduisent un couplage lâche et une architecture plus efficace.
- **Business Context Purity (BCP)** : Mesure la cohésion fonctionnelle des partitions en fonction de leur contexte métier. Des valeurs faibles indiquent une meilleure spécialisation et un alignement plus net avec les objectifs métier.

#### 2.3.2.4 Analyse des systèmes monolithiques utilisés pour l'évaluation des approches

Dans la littérature existante, divers systèmes ont été utilisés comme références afin d'évaluer la qualité des partitions générées par les différents outils d'identification de microservices. Nous avons compilé la liste de ces systèmes de référence, associés aux outils séléctionnés, dans le tableau 2.5. Il est à noter que la majorité de ces systèmes de référence sont basés sur le langage Java. Pour extraire les informations pertinentes relatives à la taille et aux caractéristiques de ces applications, nous avons eu recours à l'outil *Understand* développé par Scitools[3], tel que

[3] Understand by Scitools : https://scitools.com

présenté dans le tableau 2.4. Parmi les systèmes les plus fréquemment employés, on retrouve notamment :

Tableau 2.4 Systèmes utilisés dans les décompositions

| **Système** | **#Lignes de code** | **#Classes** | **#Méthodes** |
|---|---|---|---|
| JPetStore | 8627 | 41 | 394 |
| DayTrader | 24781 | 118 | 1134 |
| AcmeAir | 5816 | 34 | 300 |
| JForum | 15781 | 340 | 491 |
| PetClinic | 13719 | 44 | 164 |
| SpringBlog | 4217 | 48 | 195 |

Tableau 2.5 Systèmes de référence utilisés par les approches à évaluer

| **Outil** | **Systèmes de référence** |
|---|---|
| Service Cutter | Trading System, Cargo Tracking System |
| Fo-SCI | Solo, SpringBlog, JForum, Apache Roller, Agilefant, XWiki |
| Mono2Micro | DayTrader, Acme Air, JPetStore, Plants |
| MSExtractor | JPetStore, SpringBlog, JForum, Roller |
| CoGCN | DayTrader, PBW, DietApp, Acme Air |
| MAGNET | Compiere, FXML-POS, PetClinic, JForum |

**DayTrader**[4] est un système open-source fréquemment mobilisé comme référence dans l'évaluation de la performance des outils de décomposition. Cette plateforme Java simule un système de courtage en ligne, offrant des fonctionnalités telles que l'authentification des utilisateurs, la consultation de portefeuilles, la recherche de cotations boursières et la gestion des transactions.

**Acme Air**[5] est une autre application open-source largement utilisée. Conçu pour représenter une compagnie aérienne fictive, ce système Java est capable de monter en charge pour supporter des

[4] DayTrader : https://github.com/WASdev/sample.daytrader7

[5] Acme Air : https://github.com/blueperf/acmeair-monolithic-java

milliards d'appels API par jour. Disponible en version monolithique et en version microservices [6], Acme Air constitue un terrain d'évaluation idéal pour comparer les résultats des outils de décomposition à une implémentation microservices existante.

**JPetStore** [7] est un système open-source léger permettant la gestion d'une animalerie. De par sa simplicité et sa taille maîtrisable, il sert de banc d'essai pertinent pour évaluer la qualité des partitions générées par différents outils.

**JForum** [8] est une application web Java dédiée aux forums de discussion. Grâce à ses fonctionnalités robustes, ce système est souvent utilisé comme base d'évaluation dans les travaux académiques.

**Apache Roller** [9] est un serveur de blogs collaboratif basé sur Java, offrant une solution riche en fonctionnalités pour l'hébergement de blogs de tailles variées.

**Petclinic Spring** [10] est une application Spring Boot souvent utilisée dans les comparaisons expérimentales. Disponible en version monolithique et en version microservices [11], elle constitue, à l'instar d'Acme Air, une référence pour confronter les partitions proposées aux implémentations microservices existantes.

**SpringBlog** [12] est une application simple de microblogging permettant de publier des notes et articles accompagnés de pièces jointes.

---

[6] Acme Air microservices-based : https://github.com/blueperf/acmeair-mainservice-java

[7] JPetStore : https://github.com/mybatis/jpetstore-6

[8] JForum : https://sourceforge.net/projects/jforum2

[9] Apache Roller : https://roller.apache.org

[10] Petclinic Spring : https://github.com/spring-projects/spring-petclinic

[11] Petclinic Spring microservices-based : https://github.com/spring-petclinic/spring-petclinic-microservices

[12] SpringBlog : https://github.com/Raysmond/SpringBlog

## 2.4 QR1 : Comment les outils sélectionnés pour l'identification de microservices performent-ils lorsqu'ils sont appliqués à des systèmes monolithiques ?

Dans cette section, nous cherchons à répondre à la question de recherche *« Comment les outils sélectionnés pour l'identification de microservices performent-ils lorsqu'ils sont appliqués à des systèmes monolithiques ? »*. Pour ce faire, nous présentons d'abord les études de cas retenues ainsi que la préparation du *système référentiel* utilisé. Nous discutons ensuite les résultats issus de la décomposition, en nous appuyant sur deux axes d'évaluation complémentaires : d'une part, des métriques de décomposition du système, et d'autre part, une comparaison par rapport au *vérité au sol*. Cette double approche permet d'analyser empiriquement les performances des outils choisis et d'apprécier la pertinence des partitions qu'ils génèrent.

### 2.4.1 Études de cas

Afin d'évaluer de manière rigoureuse les outils de décomposition en microservices sélectionnés, nous avons mis en œuvre un processus méthodologique complet comprenant la sélection des systèmes de référence, l'exécution des outils et la préparation des systèmes références. Nous avons commencé par examiner la littérature afin d'identifier les applications de référence les plus fréquemment utilisées dans les études de décomposition. Parmi celles-ci, *JPetStore* et *DayTrader* se sont imposées comme les candidates les plus représentatives, comme le souligne également (Saucedo *et al.*, 2025). Ces deux systèmes, développés en Java et représentatifs respectivement d'architectures de petite et moyenne taille, garantissent une cohérence entre les évaluations et facilitent la comparaison avec les travaux antérieurs. Pour établir des données de référence fiables, nous avons étudié et analysé une version en microservices existante du système JPetStore disponible dans le dépôt public [13]. Après un examen approfondi de l'architecture et la vérification de sa conformité aux principes de conception des microservices, nous l'avons adoptés comme système de référence pour notre évaluation. En revanche, nous n'avons pas identifié de version en microservices suffisamment documentée du système DayTrader pouvant être utilisée comme système de référence dans notre étude.

[13] https://github.com/pzaragoza93/MSAJPetStore/tree/main

Pour chaque outil étudié, nous avons obtenu la version open source la plus récente, puis procédé à son exécution sur nos environnements en utilisant les systèmes de référence sélectionnés. Nous avons reproduit le processus d'exécution, extrait les résultats de décomposition, et les avons évalués à l'aide des sept métriques définies dans la section précédente. Cette méthodologie nous a permis d'évaluer la performance de chaque outil dans un cadre contrôlé et comparable, en nous appuyant sur des entrées homogènes et un système de référence validé.

### 2.4.2 Évaluation basée sur la décomposition des systèmes

Les résultats complets de ces évaluations sont synthétisés dans les tableaux 2.6 et 2.7, correspondant respectivement aux systèmes de référence JPetStore et DayTrader.

Tableau 2.6 Résultats d'évaluation des outils avec l'application *JPetStore*

| **Outils** | **SMQ** (↑) | **IFN** (↓) | **ICP** (↓) | **CMQ** (↑) | **NED** (↓) | **BCP** (↓) |
|---|---|---|---|---|---|---|
| FoSCI | **0.561** | 7.25 | 0.512 | 0.898 | 0.5 | 2.871 |
| MSExtractor | 0.180 | 6.0 | 0.614 | **0.938** | 0.417 | 1.599 |
| CoGCN | 0.545 | 7.4 | 0.663 | 0.875 | 0.706 | 2.700 |
| Mono2Micro | 0.367 | 7.8 | 0.651 | 0.7 | 0.706 | 2.185 |
| Magnet | 0.449 | 16.429 | 0.747 | 0.697 | 0.600 | 2.258 |
| ServiceCutter | 0.036 | **4.143** | **0.465** | 0.191 | **0.389** | **1.364** |

Tableau 2.7 Résultats d'évaluation des outils avec l'application *DayTrader*

| **Outils** | **SMQ** (↑) | **IFN** (↓) | **ICP** (↓) | **CMQ** (↑) | **NED** (↓) | **BCP** (↓) |
|---|---|---|---|---|---|---|
| FoSCI | 0.138 | 20.5 | 0.58 | 0.985 | 0.754 | 3.189 |
| MSExtractor | 0.074 | 6.833 | 0.392 | **0.996** | 1.000 | 2.345 |
| CoGCN | 0.214 | 9.000 | **0.295** | 0.799 | 0.689 | 2.046 |
| Mono2Micro | **0.293** | 11.400 | 0.411 | 0.995 | 0.581 | 2.959 |
| Magnet | 0.264 | 29.143 | 0.814 | 0.886 | **0.361** | 3.616 |
| ServiceCutter | 0.056 | **2.4** | 0.781 | 0.145 | 0.716 | **0.85** |

**Résultats selon la métrique SMQ**

**FoSCI** atteint systématiquement certains des meilleurs scores SMQ. Cela traduit une forte cohésion interne et des clusters bien connectés. Toutefois, l'augmentation du nombre de microservices entraîne une baisse des scores SMQ, suggérant que des décompositions plus fines affaiblissent la cohésion structurelle. Cette tendance illustre le compromis inhérent entre modularité et granularité dans la stratégie de clustering de FoSCI. **CoGCN** présente des performances en SMQ comparables à celles de FoSCI, maintenant une modularité élevée dans les décompositions à faible nombre de microservices. **MAGNET** se distingue par des valeurs SMQ stables et relativement élevées, quel que soit le niveau de granularité. Sa granularité au niveau des méthodes, combinée à l'encodage des dépendances inter-méthodes, lui permet de générer des clusters cohésifs. Cette constance fait de MAGNET un outil particulièrement adapté aux décompositions fines. **Mono2Micro** obtient des valeurs SMQ modérées, qui, sans être les meilleures, restent proches de celles d'outils davantage orientés sur la structure. Sa stratégie de clustering hiérarchique favorise un équilibre entre partitions larges et partitions trop fines, produisant ainsi une cohésion moyenne mais relativement constante. **MSExtractor** se situe parmi les moins performants en matière de SMQ. Ses valeurs sont nettement inférieures à celles de la plupart des autres outils, ce qui s'explique par son orientation vers une optimisation multi-objectifs. En cherchant à équilibrer couplage et granularité plutôt qu'à maximiser la cohésion structurelle, **MSExtractor** tend à générer des unités de plus petite taille, fonctionnellement isolées mais peu modulaires. Enfin, **Service Cutter** présente les performances les plus faibles pour la métrique SMQ. Ses décompositions, quel que soit l'algorithme utilisé, produisent systématiquement des microservices trop fins, conduisant à une fragmentation excessive et à une faible cohésion intra-service. Cette granularité extrême compromet la formation de services véritablement modulaires.

**Résultats selon la métrique IFN**

En ce qui concerne la métrique IFN, rappelons qu'une valeur plus faible traduit un nombre réduit de dépendances inter-services, ce qui simplifie l'intégration et réduit le couplage. **FoSCI**

a présenté des scores IFN stables et relativement faibles pour tous les niveaux de décomposition. Sa stratégie de regroupement des classes fortement liées sur le plan structurel et sémantique permet de limiter la surface d'exposition des interfaces entre microservices. **CoGCN** a également obtenu de très bons résultats pour la métrique IFN. L'utilisation des réseaux de neurones de graphes lui permet d'optimiser les relations entre nœuds et de produire des décompositions caractérisées par un minimum d'interactions inter-services. **MSExtractor** a maintenu des valeurs IFN modérément basses, confirmant son objectif de conception visant à minimiser le couplage par une optimisation multi-objectifs. Ses résultats traduisent une isolation efficace des services. L'équilibre recherché entre cohésion et couplage se traduit par des décompositions présentant une inflation limitée du nombre d'interfaces. **Mono2Micro** a affiché des performances moyennes en matière d'IFN. Comme il repose largement sur l'exploitation des traces d'exécution, les partitions générées reflètent fidèlement les interactions dynamiques réelles du système, mais elles ne garantissent pas toujours une minimisation des interfaces. Si l'outil capture correctement les dépendances observées en phase d'exécution, il peut néanmoins conserver des chemins de communication inter-services inutiles, en particulier lorsque les comportements temporels dominent les données de traces. **MAGNET**, en revanche, a enregistré des scores IFN relativement élevés, surtout lorsque le nombre de microservices augmentait. Cette tendance s'explique par sa granularité au niveau des méthodes, qui engendre une plus grande surface d'interactions inter-services lorsque des méthodes couvrent plusieurs préoccupations fonctionnelles. Ainsi, bien que MAGNET assure une forte cohésion structurelle (comme le montrent les résultats SMQ), cela se fait au prix d'une complexité accrue au niveau des interfaces, en particulier dans les décompositions fines. Enfin, **Service Cutter** a obtenu les valeurs IFN les plus faibles parmi les outils évalués, ce qui constitue un résultat positif puisqu'il reflète une exposition limitée des interfaces par microservice. Cela suggère que les services générés sont bien ciblés et conformes au principe de responsabilité unique, favorisant leur maintenabilité et leur réutilisation. Toutefois, ces résultats doivent être nuancés : ils sont fortement influencés par la tendance de l'outil à produire des décompositions très fines. Dans le cas de *DayTrader*, par exemple, le système a été fragmenté en un grand nombre de microservices. Si cette granularité contribue à réduire les

valeurs IFN, elle peut également engendrer une surcharge de gestion due à la multiplication des services de petite taille.

**Résultats selon la métrique ICP**

Dans la suite de notre évaluation, nous considérons la métrique ICP, qui mesure la proportion d'appels inter-services. Un faible ICP indique que les interactions à l'exécution sont bien contenues au sein des frontières de chaque microservice, favorisant ainsi la cohésion et l'efficacité des performances en réduisant la surcharge de communication entre services. **CoGCN** obtient de très bons résultats pour l'ICP. Ce comportement s'explique par sa stratégie de décomposition sensible aux *outliers*, qui regroupe les classes similaires sur le plan structurel et sémantique, réduisant ainsi le nombre d'appels entre partitions. Sa capacité à optimiser les chemins de communication à l'exécution tout en traitant les classes atypiques lui permet de maintenir des valeurs ICP basses. **FoSCI** présente également de solides performances pour l'ICP, grâce à son regroupement basé sur les « atomes fonctionnels ». En rassemblant les classes fortement couplées selon leur proximité structurelle et comportementale à l'exécution, FoSCI veille à ce que les chemins d'appel fréquents demeurent contenus dans le même microservice. Le résultat est une cohésion élevée et une fuite minimale des appels à l'exécution, garantissant une bonne performance ICP quel que soit le niveau de granularité. **MAGNET** atteint lui aussi des scores ICP élevés. Malgré sa granularité au niveau des méthodes et une complexité d'interface plus importante (IFN), l'outil parvient à contenir efficacement les dépendances à l'exécution. Cette performance est due à sa stratégie de regroupement fondée sur les dépendances fonctionnelles, qui maintient ensemble les méthodes fréquemment invoquées, assurant ainsi la préservation efficace des chemins d'appel. **MSExtractor** obtient des scores ICP honorables, principalement grâce à son objectif de réduction du couplage par optimisation multi-objectifs. Bien que l'outil ne se spécialise pas explicitement dans la modélisation des comportements dynamiques, sa stratégie d'équilibrage des partitions contribue indirectement à réduire les interactions inter-services en alignant les frontières fonctionnelles. **Mono2Micro** enregistre des valeurs ICP supérieures à la moyenne, grâce à son extraction des arbres de contexte d'appel à partir des traces d'exécution.

Cette approche permet de capturer les dépendances temporelles et de regrouper les classes qui interagissent fréquemment dans des scénarios d'utilisation réels. Toutefois, les performances fluctuent légèrement selon la qualité des traces et la variabilité des exécutions du système. À l'inverse, **ServiceCutter** affiche de faibles performances pour l'ICP. Sa tendance à générer un nombre excessif de microservices, quelle que soit la variante algorithmique utilisée, entraîne une multiplication des appels inter-services, d'autant plus que la dynamique d'exécution n'est pas directement prise en compte. Cette fragmentation accroît la surcharge de communication à l'exécution, ce qui nuit à l'efficacité de l'ICP.

**Résultats selon la métrique CMQ**

Un CMQ élevé implique que la décomposition conserve une intégrité conceptuelle, rendant les microservices plus faciles à comprendre, à maintenir et à faire évoluer. **Mono2Micro** atteint des valeurs comparables, grâce à son analyse du code source et à sa capacité à segmenter l'application selon des cas d'utilisation métier distincts. **MSExtractor** se distingue positivement en CMQ. Son approche orientée vers la décomposition fonctionnelle et sa stratégie multi-objectifs lui permettent de trouver un compromis efficace entre granularité, couplage et regroupement conceptuel. **MAGNET**, en revanche, tend à accorder moins d'importance à l'abstraction conceptuelle au profit d'un regroupement fondé sur les dépendances. Si cette stratégie garantit une forte cohésion fonctionnelle, elle peut négliger les relations conceptuelles de plus haut niveau. Enfin, **Service Cutter** obtient de faibles scores CMQ. Sa tendance à générer un nombre excessif de microservices conduit souvent à la dispersion de classes conceptuellement liées dans des partitions distinctes, ce qui nuit à la cohésion logique et complique la compréhension des services.

**Résultats selon la métrique NED**

En ce qui concerne la métrique NED, un score élevé reflète un meilleur équilibre dans la taille des partitions. **Mono2Micro** enregistre les valeurs NED les plus élevées parmi tous les outils. Son approche hiérarchique favorise naturellement des partitions équilibrées, avec une distribution

proportionnelle des classes. Cela évite à la fois une concentration excessive de logique et une fragmentation inutile, faisant de Mono2Micro une solution adaptée aux architectures maintenables et évolutives. **FoSCI** obtient des scores NED comparables,. L'outil tend à maintenir des partitions équilibrées en regroupant des classes structurellement et sémantiquement proches. **MSExtractor** affiche de bons résultats NED pour les systèmes de grande taille, confirmant la capacité de son optimisation multi-objectifs à éviter les partitionnements extrêmes. L'outil parvient à équilibrer granularité et distribution des tailles, garantissant qu'aucun microservice ne devienne disproportionné. Cela favorise la maintenabilité des applications de taille moyenne à grande. **CoGCN** maintient généralement des scores NED stables. Cette sensibilité est probablement due au traitement des outliers, qui peut déséquilibrer les tailles des clusters. **MAGNET** présente des résultats NED faibles. Sa granularité au niveau des méthodes le rend sensible à la création de partitions déséquilibrées lorsque peu de microservices sont générés. **Service Cutter** se classe parmi les moins performants pour la métrique NED. Ses stratégies de décomposition conduisent fréquemment à une forte fragmentation, produisant un grand nombre de microservices de petite taille avec des déséquilibres marqués.

**Résultats selon la métrique BCP**

Concernant la métrique BCP, **FoSCI** obtient de très bons résultats. Son recours au clustering sémantique, basé sur les identifiants et la sémantique des classes, permet de regrouper des éléments représentant une fonctionnalité métier cohérente. **CoGCN** se démarque également. Sa combinaison d'analyses fondées sur les attributs et d'embeddings de graphes lui permet de capturer les similarités sémantiques et de produire des services alignés sur des contextes métier clairs. **MAGNET** affiche des scores BCP stables et équilibrés. Son approche de clustering fondée sur les graphes exploite à la fois les dépendances fonctionnelles et les relations sémantiques entre méthodes, produisant des services respectant la structure du code et les frontières du domaine. **Mono2Micro** obtient également de bons résultats BCP. Sa capacité à préserver les frontières correspondant aux cas d'usage métier, en combinant signaux statiques et dynamiques, favorise la production de services cohérents et alignés sur le domaine. Bien que moins précis

sémantiquement, l'outil respecte la logique temporelle d'exécution, qui reflète souvent les processus métier. **MSExtractor** obtient des scores BCP relativement bas. Sa décomposition est davantage orientée vers l'indépendance fonctionnelle et la minimisation des interfaces que vers la cohérence sémantique. Par conséquent, les services produits regroupent parfois des classes fonctionnellement indépendantes mais non liées par un contexte métier commun. Enfin, **Service Cutter** présente les résultats les plus faibles en BCP. Sa sur-fragmentation disperse les classes conceptuellement liées entre de multiples microservices, entraînant une dilution sémantique marquée.

Dans l'ensemble, les résultats montrent que **FoSCI** et **CoGCN** se distinguent comme les outils les plus performants et les plus constants à travers l'ensemble des métriques. **FoSCI** présente une excellente cohésion structurelle (SMQ) et une forte pureté contextuelle métier (BCP), tout en maintenant un faible couplage (IFN, ICP) et un bon équilibre de taille (NED). Ces performances homogènes traduisent une approche robuste, capable de produire des décompositions cohérentes tant sur le plan structurel que fonctionnel. **CoGCN** confirme également son efficacité : sa capacité à capturer les dépendances structurelles et sémantiques via les réseaux de neurones de graphes lui permet d'obtenir des résultats très compétitifs sur la plupart des métriques, en particulier pour IFN, ICP et BCP.

**MAGNET** se positionne juste derrière, performant en termes de cohésion (SMQ) et de confinement des appels à l'exécution (ICP), mais pénalisé par une granularité parfois trop fine qui augmente le couplage (IFN) et déséquilibre les partitions (NED). **Mono2Micro** se distingue par son équilibre global : bien qu'il n'excelle dans aucune métrique spécifique, il maintient des scores stables et harmonieux (CMQ, NED, BCP), confirmant la pertinence de sa combinaison d'analyses statiques et dynamiques. À l'inverse, **MSExtractor** et **Service Cutter** affichent des performances plus faibles. Le premier privilégie l'optimisation multi-objectifs au détriment de la cohésion interne, tandis que le second souffre d'une sur-fragmentation entraînant une baisse significative de la cohésion et de la modularité (SMQ, CMQ, NED, BCP).

Enfin, les tendances observées se révèlent globalement consistantes sur les deux systèmes évalués, malgré quelques variations liées à la taille et à la complexité des applications. FoSCI et CoGCN conservent leur supériorité relative, tandis que Service Cutter et MSExtractor demeurent en retrait, confirmant la stabilité des résultats et la fiabilité comparative des outils sur des contextes variés de décomposition.

### 2.4.3 Évaluation basée sur un système de référence

Afin d'évaluer la performance des outils de décomposition, nous avons adopté des métriques de **précision** et de **rappel** (Sellami *et al.*, 2022a), ainsi que **F1 score**. Notre processus d'évaluation repose sur un dépôt open source sur Github [14] contenant une version du système monolithique de référence *JPetStore*, implémentée selon une architecture en microservices.

Après une analyse approfondie, nous avons déterminé que ce dépôt constitue une représentation fidèle d'une architecture microservices bien structurée, pouvant servir de vérité au sol pour notre système de validation. Les microservices qui la composent ont été validés sur la base de critères de conception solides : ils présentent une isolation fonctionnelle claire, une séparation cohérente des responsabilités et une granularité adaptée aux domaines métier qu'ils implémentent. De plus, la communication entre services repose sur des interfaces bien définies et une faible interdépendance, garantissant une modularité et une maintenabilité élevées. Ces caractéristiques font de ce système un candidat de référence pour l'évaluation de la qualité des décompositions produites par les outils étudiés. Nous avons ensuite mesuré dans quelle mesure les outils de décomposition étudiés s'alignent avec cette décomposition de référence. Pour quantifier la justesse des décompositions extraites, nous nous appuyons sur les définitions suivantes de la précision et du rappel, adaptées de (Sellami *et al.*, 2022a).

La **précision** ($P$) mesure la proportion de classes correctement assignées au sein d'un microservice extrait par rapport au nombre total de classes qui lui sont attribuées. Elle est définie comme indiqué dans l'équation (2.14) :

[14] https ://github.com/pzaragoza93/MSAJPetStore/tree/main

$$P = \frac{1}{|M|} \sum_{\forall m_i \in M} \frac{|m_i \cap Corr(m_i, M_t)|}{|m_i|}, \tag{2.14}$$

où :

- $M$ est l'ensemble des microservices extraits,
- $M_t$ est l'ensemble des microservices de référence (*vérité au sol*),
- $Corr(m_i, M_t)$ est le microservice de référence présentant le plus grand nombre de classes en commun avec le microservice extrait $m_i$,
- $|m_i \cap Corr(m_i, M_t)|$ représente le nombre de classes correctement assignées dans $m_i$.

De manière analogue, le **rappel** ($R$) mesure la proportion de classes correctement assignées par rapport au nombre total de classes présentes dans le microservice de référence correspondant. Il est défini comme indiqué dans l'équation (2.15) :

$$R = \frac{1}{|M|} \sum_{\forall m_i \in M} \frac{|m_i \cap Corr(m_i, M_t)|}{|Corr(m_i, M_t)|}, \tag{2.15}$$

Le F1-score est une mesure harmonique qui combine la précision et le rappel afin de fournir une évaluation globale de la performance de la décomposition. Sa définition est donnée dans l'équation (2.16) :

$$F1 = 2 \times \frac{\text{Précision} \times \text{Rappel}}{\text{Précision} + \text{Rappel}}, \tag{2.16}$$

Le F1-score a été calculé à partir des valeurs de précision et de rappel obtenues pour chaque outil de décomposition. Il permet d'équilibrer l'importance des faux positifs et des faux négatifs, offrant ainsi une mesure plus représentative de la qualité globale des résultats.

Ces métriques permettent d'évaluer dans quelle mesure les outils de décomposition préservent les frontières logiques des microservices en comparaison avec l'architecture de référence.

Tableau 2.8 Précision, rappel et F1-score pour *JPetStore*

| **Outil de décomposition** | **Précision** | **Rappel** | **F1-score** |
|---|---|---|---|
| MSExtractor | 69.05% | 37.96% | 49.2% |
| Mono2Micro | 84.33% | 52.22% | 64.5% |
| FoSCI | 72.92% | 50.00% | 59.3% |
| CoGCN | 72.50% | 39.58% | 51.0% |
| MAGNET | 63.61% | 38.19% | 47.8% |

Les résultats d'évaluation présentés dans le tableau 2.8 offrent une comparaison des outils de décomposition en termes de précision, de rappel et de F1-score lorsqu'ils sont appliqués à *JPetStore*. Les résultats indiquent que **Mono2Micro** atteint la meilleure précision (84.33%), traduisant une excellente capacité à identifier des microservices dont les classes sont majoritairement correctement assignées. Son rappel (52.22%) demeure toutefois plus faible, ce qui suggère que certaines classes pertinentes du système de référence n'ont pas été correctement incluses. En revanche, **FoSCI** une précision de 72.92% et un rappel de 50.00%, ce qui se reflète dans un F1-score de 59.3%, indiquant une performance globale moins solide que Mono2Micro. **CoGCN** obtient une précision de 72.50% et un rappel de 39.58%, avec un F1-score de 51.0%, démontrant une tendance à bien identifier les classes correctes mais à manquer une partie notable des classes attendues. **MAGNET**, avec une précision de 63.61% et un rappel de 38.19% (F1-score de 47.8%), présente des performances plus modestes comparé aux outils existants, indiquant une couverture plus limitée des microservices de référence. Enfin, **MSExtractor** atteint un F1-score de 49.2%, avec une précision (69.05%) supérieure à celle de MAGNET mais un rappel plus faible (37.96%), ce qui traduit une sensibilité réduite au regroupement de classes correctes en microservices.

Dans l'ensemble, **Mono2Micro** se distingue par la précision la plus élevée et offre le meilleur compromis entre exhaustivité en terme de microservices identifiés et exactitude, illustrant sa capacité à générer une décomposition plus équilibrée du système.

## 2.5 Discussion des résultats obtenus

Les résultats de l'évaluation mettent en lumière à la fois les performances et les limites des outils de décomposition. Cette discussion aborde, d'une part, les principaux défis et contraintes rencontrés par ces approches, et d'autre part, leur applicabilité dans le contexte particulier des systèmes d'intelligence artificielle, où la complexité des pipelines de données et l'intégration des modèles posent des exigences spécifiques.

### 2.5.1 Défis et limites des approches de décomposition

Lors de l'analyse des différentes stratégies de collecte de données, notre étude met en évidence une dépendance marquée à l'utilisation du code source comme entrée principale des outils de décomposition. Si cette approche s'avère efficace pour des applications de petite taille, dont le code est accessible et bien structuré, elle présente toutefois plusieurs limites. La principale réside dans l'absence de compréhension contextuelle : s'appuyer exclusivement sur le code source ne permet pas de saisir pleinement le comportement global du système, ses dépendances et ses interactions à l'exécution, ce qui conduit à négliger des relations essentielles, particulièrement dans les systèmes de grande envergure caractérisés par des dépendances complexes et des comportements dynamiques. De plus, cette focalisation sur le code fait abstraction d'artefacts non codés tels que les diagrammes UML, les schémas de bases de données ou encore les fichiers de configuration. Ces éléments jouent pourtant un rôle central dans la délimitation des frontières des microservices, car ils offrent une vue architecturale précieuse, illustrant notamment les parcours utilisateurs, les flux d'entrée/sortie et l'organisation générale du système. Un autre écueil fréquent des approches basées sur le code réside dans la sous-estimation des interactions avec la base de données. Dans le cadre des architectures en couches, et plus particulièrement des applications orientées services web, la couche de persistance est essentielle. Ne pas la prendre en compte peut conduire à générer des « monolithes distribués » plutôt que de véritables microservices indépendants (Filippone *et al.*, 2023).

L'analyse statique du code, en particulier, reste limitée par sa nature instantanée : elle offre une représentation du système à un moment donné, sans tenir compte de son évolution continue. Pour pallier cette limite, l'intégration d'une analyse sémantique apparaît comme une voie prometteuse. En allant au-delà des dépendances structurelles, l'analyse sémantique apporte une compréhension contextuelle des composants logiciels et enrichit ainsi l'efficacité des outils de décomposition, en rapprochant l'analyse centrée sur le code d'une vision plus globale du système.

L'utilisation des traces d'exécution constitue une autre approche, notamment lorsque le code source n'est pas accessible et que le système est traité comme une « boîte noire ». Ces traces offrent une vision fidèle du comportement réel du système et des dépendances effectives entre ses composants. Toutefois, leur exploitation soulève plusieurs défis. La collecte de traces complètes requiert une instrumentation approfondie couvrant une large variété de scénarios d'utilisation et de parcours utilisateurs, afin d'assurer une représentativité adéquate des interactions. Par ailleurs, leur intégration avec l'analyse du code doit être soigneusement réalisée pour éviter les divergences et favoriser une vision complémentaire. Des outils tels que Mono2Micro (Kalia *et al.*, 2021) illustrent cette synergie en combinant analyse statique et traces d'exécution pour capter à la fois les dépendances structurelles et les usages réels du système.

Une troisième approche consiste à exploiter l'historique des dépôts (commits, auteurs, etc.) en complément de l'analyse du code. Cette intégration permet une compréhension plus holistique de la base de code avant l'analyse historique. Cependant, elle est elle aussi confrontée à des limites notables. L'une des principales difficultés provient du bruit présent dans l'historique des commits : modifications liées au formatage, insertion de code temporaire de débogage, etc. Identifier et filtrer les changements réellement significatifs nécessite ainsi des mécanismes additionnels. De plus, l'historique ne fournit pas de compréhension sémantique du code. À cela s'ajoute le fait que certaines fonctionnalités critiques, comme les tests ou les interactions avec les bases de données, sont souvent omises par ces approches. En effet, de nombreux outils se concentrent sur la capture des scénarios opérationnels normaux, négligeant les phases de test. Cela pose une question méthodologique majeure : faut-il intégrer les fonctionnalités de test dans

la décomposition, en les associant aux classes métier correspondantes, ou bien les isoler dans un microservice dédié ?

La première option renforce la cohésion entre test et implémentation, en facilitant leur maintenance et leur co-évolution. La seconde, en revanche, permet de séparer les préoccupations, favorisant des environnements de test indépendants et plus spécialisés, tout en offrant des avantages en termes de scalabilité et de déploiement. Néanmoins, ce découplage peut fragiliser le lien entre les tests et le code métier qu'ils valident, entraînant un risque d'obsolescence. Ces constats mettent en lumière la complexité de l'exploitation des données historiques dans la décomposition, et soulignent la nécessité d'une intégration fine avec les analyses du code pour parvenir à une évaluation plus exhaustive et fiable de la base logicielle.

**Points clés :**

- La plupart des outils de décomposition reposent principalement sur le code source, efficace pour des applications de petite taille mais insuffisant pour les systèmes complexes.
- L'analyse du code néglige souvent les artefacts non codés (diagrammes UML, schémas de bases de données, fichiers de configuration), pourtant essentiels pour définir les frontières des microservices.
- Les interactions avec les bases de données sont fréquemment sous-estimées, ce qui peut mener à la création de monolithes distribués plutôt que de véritables microservices.
- L'analyse statique fournit une vision figée du système, sans prise en compte de sa dynamique ni de son évolution future.
- Les traces d'exécution offrent une vision précieuse du comportement réel du système, mais nécessitent une collecte représentative et une intégration étroite avec l'analyse statique.
- L'exploitation de l'historique des commits complète l'analyse du code mais introduit du bruit, manque de compréhension sémantique et omet souvent les fonctionnalités liées aux tests et aux bases de données.

L'analyse croisée des résultats, à partir des deux évaluations, met en évidence que les outils de décomposition n'excellent pas tous selon les mêmes dimensions. Les métriques structurelles (SMQ, IFN, ICP, BCP, NED) révèlent la capacité de FoSCI et CoGCN à générer des architectures cohérentes, bien modularisées et faiblement couplées, traduisant une compréhension fine des relations internes au système. Cependant, lorsque l'on considère les mesures empiriques (précision, rappel, F1-score), la performance perçue change légèrement : Mono2Micro surpasse les autres en termes de précision et conserve un équilibre solide entre exhaustivité et exactitude, confirmant la robustesse observée dans les indicateurs structurels. Cette convergence des résultats indique que Mono2Micro reste l'outil le plus régulier et le plus polyvalent dans des scénarios de migration réels. CoGCN, bien qu'excellent sur le plan de la cohésion et du couplage, affiche un rappel plus faible, suggérant qu'il identifie des microservices très cohérents mais parfois incomplets un comportement adapté aux systèmes complexes où la précision des frontières fonctionnelles prime sur la couverture totale. À l'inverse, Mono2Micro privilégie la stabilité et la rigueur dans la répartition des classes, ce qui le rend efficace pour des systèmes industriels bien structurés cherchant une migration encadrée. Enfin, MAGNET, MSExtractor et ServiceCutter, moins performants globalement, produisent néanmoins des partitions exploitables pour une première exploration ou un affinement progressif de la décomposition. Ensemble, ces résultats soulignent que chaque outil présente une orientation propre : certains optimisent la qualité structurelle globale, d'autres la fidélité à la référence, illustrant la nécessité d'adapter le choix de l'outil au profil du système et aux objectifs de la migration.

### 2.5.2 Applicabilité sur les systèmes IA

L'application des techniques de décomposition et de migration existantes, initialement conçues pour les systèmes logiciels traditionnels, met en évidence plusieurs limites fondamentales lorsqu'elles sont transposées aux systèmes basés sur l'apprentissage automatique. En effet, la nature architecturale et structurelle des systèmes IA diffère profondément de celle des systèmes classiques. Dans les logiciels traditionnels, la logique métier est généralement distribuée au sein

de classes bien définies, de modules organisés en couches et de dépendances explicites, offrant ainsi une base structurelle riche pour la décomposition.

De plus, les systèmes IA, développées majoritairement en Python grâce à la maturité de son écosystème de bibliothèques telles que TensorFlow, PyTorch ou scikit-learn, présentent une granularité structurelle réduite. Les fonctionnalités complexes sont souvent encapsulées dans des appels d'API concis, ce qui se traduit par des structures de dépendances et des hiérarchies de classes limitées. Les étapes de traitement de données, d'ingénierie des caractéristiques, d'entraînement et d'évaluation des modèles sont fréquemment entremêlées dans des scripts monolithiques ou des cellules de notebooks. Dans ce contexte, les artefacts structurels sur lesquels reposent les méthodes traditionnelles de décomposition sont moins disponibles, voire inexistants, rendant difficile l'identification de frontières de microservices pertinentes.

Bien que de nombreux outils de décomposition aient été proposés dans la littérature pour faciliter la migration des systèmes monolithiques vers des architectures en microservices (Saucedo *et al.*, 2025; Oumoussa & Saidi, 2024; Abgaz *et al.*, 2023), leur applicabilité aux systèmes IA demeure limitée et inexplorée. Ces outils sont majoritairement orientés vers des environnements comme Java et ignorent les spécificités des flux de travail propres aux systèmes IA.

## 2.6 Conclusion

En conclusion, ce chapitre a permis d'identifier et d'analyser les différentes approches de migration des systèmes monolithiques vers des architectures à microservices présentées dans la littérature. Nous avons ensuite procédé à une évaluation empirique de plusieurs de ces approches open source, afin de comparer leurs performances respectives à l'aide de métriques issues de travaux antérieurs. L'analyse des résultats obtenus a mis en évidence les forces et limites de ces approches. Nous soulignons la nécessité de disposer de méthodes plus adaptées aux particularités des systèmes IA et nous validerons empiriquement cet aspect dans les chapitres suivants. Nous présenterons dans ce qui suit notre approche proposée, qui vise à automatiser la décomposition

des systèmes monolithiques à base d'IA vers une architecture en microservices, en s'appuyant sur des patrons de conception et des outils de décomposition intelligents.

# CHAPITRE 3

# APPROCHE DE DÉCOMPOSITION DES SYSTÈMES IA MONOLITHIQUES EN MICROSERVICES

## 3.1 Introduction

Au cours de la dernière décennie, on observe une intégration croissante de l'apprentissage automatique au sein des systèmes logiciels, couvrant un large échelle d'applications telles que les moteurs de recommandation, la reconnaissance d'images ou encore l'analytique prédictive (Chinimilli & Sadasivuni, 2024). Cette intégration a considérablement accru la complexité et les exigences architecturales de ces systèmes. Contrairement aux applications logicielles traditionnelles, les systèmes basés sur l'IA se distinguent par une composition architecturale et un cycle de vie spécifiques, articulés autour de phases interdépendantes : collecte et prétraitement des données, ingénierie des caractéristiques, entraînement des modèles, évaluation, déploiement et supervision (Amershi *et al.*, 2019). Chacune de ces étapes introduit des dépendances particulières, des flux de données complexes et des contraintes computationnelles spécifiques.

Dans ce contexte, et comme évoqué précédemment, il apparaît essentiel d'explorer comment le paradigme des microservices peut également bénéficier aux systèmes modernes fondés sur l'apprentissage automatique. Ainsi, dans ce chapitre, qui constitue la partie centrale de ce mémoire, nous présentons notre approche de décomposition automatisée des systèmes IA monolithiques en microservices, comblant ainsi une lacune des méthodes de migration existantes qui négligent souvent les structures spécifiques aux systèmes IA. En s'appuyant sur des patrons architecturaux pour distinguer les couches logicielles et les composants d'apprentissage, notre approche exploite aussi des LLMs afin de classifier les rôles des classes au sein du flux de traitement de données pour l'apprentissage automatique. Ces classifications, enrichies par des représentations vectorielles, sont ensuite regroupées par un algorithme de clustering pour produire des microservices candidats cohérents.

## 3.2 Conception de l'approche proposée

Afin de surmonter les limites des techniques traditionnelles de migration lorsqu'elles sont appliquées aux systèmes basés sur l'apprentissage automatique, nous proposons une approche de décomposition automatisée visant à isoler et modulariser les composants des systèmes IA monolithiques. Comme l'illustre la figure 3.1, cette approche s'appuie sur l'analyse sémantique du code, l'exploitation de modèles de langage préentraînés et l'apprentissage non supervisé, afin de guider la transformation de fonctionnalités fortement couplées en microservices faiblement couplés. Elle se déroule en trois phases principales, précédées d'une phase préliminaire visant à préparer les artefacts nécessaires à l'analyse.

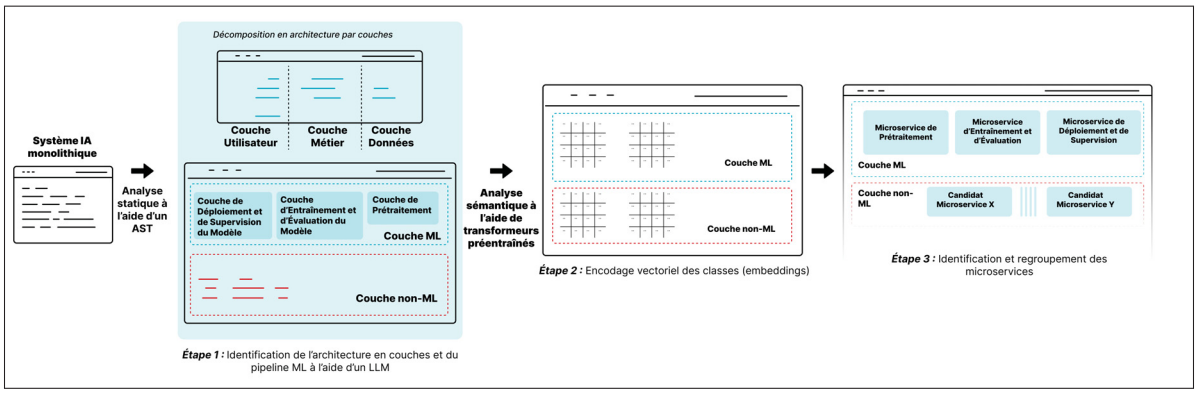


Figure 3.1 Vue d'ensemble de l'approche proposée pour l'identification de microservices candidats

- **Extraction des classes à analyser à travers un AST :**
  Pour cette étape de prétraitement préliminaire, nous avons utilisé un AST afin de parcourir les différents projets et d'en extraire automatiquement les classes constituant la base du système. L'utilisation de l'AST permet de représenter la structure interne du code sous forme d'arbre hiérarchique, facilitant ainsi l'identification des classes, de leurs attributs et de leurs dépendances. Cette extraction systématique a été appliquée à l'ensemble des projets étudiés, garantissant une représentation homogène et exploitable des composants logiciels. Ces classes extraites servent ensuite d'entrée aux étapes suivantes, où elles sont

analysées sémantiquement à l'aide de modèles d'apprentissage pour identifier les similarités structurelles et fonctionnelles en vue de la décomposition.

- **Identification de l'architecture en couches et des étapes du pipeline de l'apprentissage automatique via un LLM :** La première étape consiste en une analyse bidimensionnelle de la base de code à l'aide d'un grand modèle de langage. En s'appuyant sur des patrons architecturaux issus de la littérature (Yokoyama, 2019; Amershi *et al.*, 2019), l'analyse combine une perspective horizontale, distinguant les couches interface utilisateur, logique métier et accès aux données, et une perspective verticale, permettant de séparer les composants liés à l'apprentissage automatique des fonctionnalités classiques, puis de catégoriser les classes selon les grandes étapes du pipeline IA (prétraitement des données, entraînement, déploiement, etc.).
- **Représentation vectorielle des classes :** Dans un second temps, les classes identifiées sont transformées en représentations vectorielles de grande dimension grâce au modèle CodeBERT (Feng *et al.*, 2020). Ce choix repose sur sa capacité reconnue à capturer à la fois la structure syntaxique et les relations contextuelles du code. Le résultat est une matrice d'embeddings enrichie, intégrant les informations issues du code et de la classification réalisée à l'étape précédente. Cette matrice constitue la base de l'analyse non supervisée de la phase suivante.
- **Clustering et identification des microservices candidats :** Enfin, un algorithme de regroupement non déterministe, en l'occurrence HDBSCAN (Campello, Moulavi & Sander, 2013), est appliqué afin de regrouper les classes présentant des similarités sémantiques. Cet algorithme, basé sur la densité, est particulièrement adapté à des ensembles hétérogènes tels qu'une base de code. Les clusters obtenus correspondent à des frontières potentielles de microservices, qui sont ensuite interprétées et validées à la lumière des patrons de conception propres aux systèmes IA, garantissant ainsi cohérence architecturale et modularité fonctionnelle.

Cette chaîne de traitement automatisée permet ainsi une décomposition guidée par les patrons de conception, spécifiquement adaptée aux systèmes IA. Les sections suivantes en détailleront le fonctionnement au travers d'un exemple illustratif.

### 3.2.1 Phase 1 : Identification de l'architecture en couches et du pipeline de l'apprentissage automatique

**Entrée :** Code source de chaque classe ainsi qu'un prompt définissant les différentes couches verticales (*Utilisateur*, *Logique*, *Données*) et les catégories horizontales (*Non-IA*, *Prétraitement IA*, *Entraînement & Évaluation*, *Déploiement & Surveillance*).
**Sortie :** Correspondance des classes avec les étiquettes verticales et horizontales.
**Description de la phase :** La première phase de notre approche consiste à mettre en évidence l'organisation structurelle du système à travers une analyse sémantique du code source réalisée à l'aide des LLMs. Cette étape opère une double décomposition de l'application : d'une part une décomposition verticale, qui associe les classes aux différents niveaux architecturaux, et d'autre part une décomposition horizontale, qui permet de distinguer les composants liés aux composants d'apprentissage automatique de ceux relevant des fonctionnalités traditionnelles du système.

Pour la décomposition verticale, notre approche s'appuie sur le modèle architectural bien établi à trois couches, qui distingue la couche Utilisateur, la couche Logique et la couche Données (Yokoyama, 2019; Vámossy & Romhányi, 2021) ; comme l'illustre la figure 3.2 et expliqué dans le tableau 3.1. Ce schéma, largement adopté dans les systèmes traditionnels, favorise une séparation claire des responsabilités, améliorant ainsi la modularité, la maintenabilité et les performances globales. En l'absence d'une telle base architecturale orientée métier, les efforts de décomposition risquent de conduire à l'accumulation de dette technique et à l'apparition d'anti-patrons, se traduisant par des frontières de microservices mal définies, une dépendance accrue entre services, ainsi qu'une communication inefficace et coûteuse à corriger (Zaragoza, Seriai, Seriai, Shatnawi & Derras, 2022).

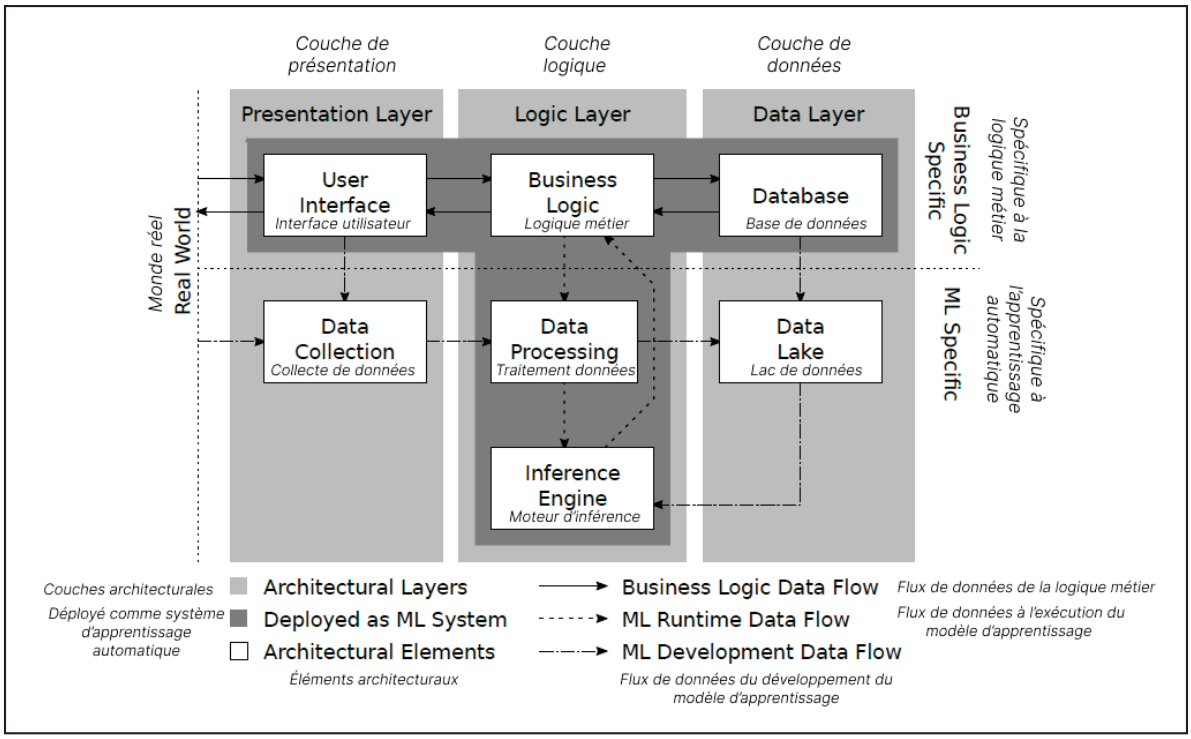


Figure 3.2 Architecture en couches pour l'apprentissage automatique
Adaptée de (Yokoyama, 2019)

Pour éviter ces limites, nous organisons explicitement le système en trois couches complémentaires. La couche Utilisateur est dédiée aux interactions avec l'utilisateur, en assurant le routage des requêtes et la mise en forme des réponses de manière adaptée et cohérente. La couche Logique regroupe les règles et opérations métier essentielles, tout en intégrant des fonctionnalités spécifiques à l'IA, telles que le prétraitement des données, l'entraînement des modèles et leur évaluation (Amershi *et al.*, 2019). Enfin, la couche Données prend en charge la gestion persistante et sécurisée des données de l'application ainsi que des modèles d'apprentissage, en appliquant des mécanismes de contrôle d'accès et de gouvernance des données.

**Récapitulatif des couches identifiées :**

Dans la décomposition horizontale, nous adoptons le patron architectural en couches appliqué à l'apprentissage automatique (Yokoyama, 2019), qui permet d'isoler systématiquement les

Tableau 3.1 Résumé des couches et classifications identifiées lors de la phase de décomposition verticale

| Couche | Description |
|---|---|
| **Utilisateur** | Interactions avec l'utilisateur, routage des requêtes, mise en forme des réponses. |
| **Logique** | Règles métier, logique applicative et pipeline de machine learning (prétraitement, entraînement, évaluation). |
| **Données** | Stockage persistant, gestion des données applicatives et des modèles, contrôle d'accès sécurisé. |

fonctionnalités propres à ce type d'apprentissage au sein de couches architecturales distinctes, comme identifiées dans la figure 3.3 et détaillées dans le tableau 3.2.

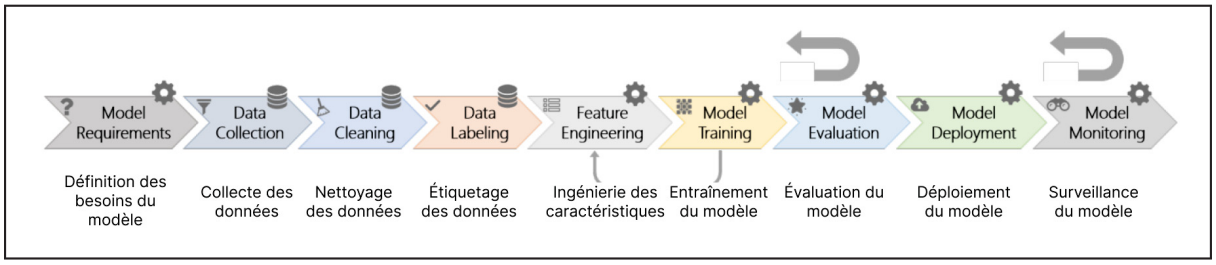


Figure 3.3 Le patron architectural en couches appliqué au machine learning
Adaptée de (Amershi *et al.*, 2019)

La séparation entre les composants IA et non-IA découle de leurs exigences spécifiques, de leurs dépendances et de leurs modes de gestion fonctionnelle. Cette distinction favorise une vision plus claire et organisée du système, en définissant précisément le rôle et les frontières de chaque composant, tout en améliorant la modularité et la capacité d'adaptation de l'architecture. Un tel découpage facilite la scalabilité : par exemple, lors de charges élevées d'inférence, la couche IA peut évoluer indépendamment sans impacter les fonctionnalités non-IA, optimisant ainsi l'allocation des ressources et garantissant performance et rentabilité (Zaragoza *et al.*, 2022). À l'intérieur de la couche IA, nous identifions les classes correspondant aux principales étapes du pipeline : prétraitement des données, entraînement, évaluation, déploiement et supervision. Chaque étape est traduite en microservice candidat afin de réduire le couplage et de renforcer la séparation des responsabilités. Ainsi, le *microservice de Prétraitement* regroupe

des tâches telles que le nettoyage des données, l'ingénierie des caractéristiques, l'annotation et la collecte. Le *microservice d'Entraînement* prend en charge l'initialisation des modèles, l'optimisation et l'ajustement des hyperparamètres pour générer des modèles robustes. Le *microservice d'Évaluation* s'appuie sur le calcul de métriques et l'analyse d'erreurs pour valider les performances. Le *microservice de Déploiement* assure la mise en production des modèles avec gestion des points d'accès et équilibrage de charge. Enfin, le *microservice de Supervision* surveille la performance en continu, active des mécanismes d'alerte et conserve les journaux afin de garantir l'intégrité du modèle (Amershi *et al.*, 2019). En parallèle, une décomposition horizontale distingue les classes générales de celles liées au ML, selon le schéma proposé par (Amershi *et al.*, 2019), en classant les composants dans quatre couches principales : les fonctionnalités générales non-ML, le prétraitement, l'entraînement et l'évaluation, puis le déploiement et la supervision en production.

**Récapitulatif des couches identifiées :**

Tableau 3.2 Résumé des couches et de leurs responsabilités identifiées lors de la phase de décomposition horizontale

| **Couche / Catégorie** | **Description et responsabilités principales** |
|---|---|
| **Couche non-IA** | Regroupe les fonctionnalités générales du système : gestion des interfaces de programmation (API), interface utilisateur, journalisation et opérations sur la base de données. |
| **Couche de prétraitement des données** | Regroupe les activités de préparation des données : nettoyage, extraction de caractéristiques, transformation, annotation et collecte. |
| **Couche d'entraînement et d'évaluation du modèle** | Concerne l'apprentissage et la validation : initialisation des modèles, optimisation, ajustement des hyperparamètres, calcul des métriques et analyse des erreurs. |
| **Couche de déploiement et de supervision du modèle** | Englobe la mise en production et le suivi : déploiement des modèles, gestion des points d'accès, équilibrage de charge, suivi des performances, alertes et journalisation. |

Afin de garantir une classification précise des éléments de code par le LLM, nous avons conçu une stratégie structurée de prompt engineering, inspirée de travaux antérieurs sur la conception

efficace de requêtes (White *et al.*, 2023; Liu *et al.*, 2021). Ces études démontrent que la fiabilité des résultats produits par les modèles de langage dépend fortement de la clarté, de la structure et de l'exhaustivité des invites formulées. En nous appuyant sur ces constats, nous avons élaboré un gabarit de prompt spécifiquement adapté à la classification en couches architecturales, composé de six éléments complémentaires : (1) le nom explicite et la catégorie du concept analysé, (2) l'intention de la tâche de classification, (3) une motivation détaillée justifiant la démarche, (4) une description de la structure et des rôles fonctionnels des entités concernées, (5) des extraits de code représentatifs servant d'ancrage concret, et (6) un résumé des conséquences, incluant les bénéfices attendus et les compromis associés (White *et al.*, 2023).

Cette structuration fournit au modèle des repères contextuels riches, tout en réduisant l'ambiguïté fréquemment observée avec des prompts non contraints. Pour renforcer encore la reproductibilité et limiter l'impact du caractère stochastique des modèles de grande taille, nous avons fixé le paramètre de température à une valeur faible, réduisant ainsi la variabilité des sorties (voir le paquet de réplication pour les détails de configuration[1]). Par ailleurs, chaque tâche de classification a été exécutée cinq fois, et la catégorie finale a été retenue selon un principe de vote majoritaire. Cette approche, proche d'un mécanisme d'agrégation, permet d'atténuer l'effet des prédictions aberrantes et d'assurer que l'étiquette attribuée reflète le résultat le plus stable à travers plusieurs itérations. En combinant un schéma de prompt structuré avec des exécutions multiples et une consolidation des résultats, notre méthode maximise les informations contextuelles disponibles pour le modèle, réduit le bruit de classification et produit des résultats plus robustes et cohérents.

[1] https ://github.com/HakimGhlissi/AIDecompose-ML-Monolith-Decomposition-Approach

Gabarit de prompt pour la classification en couches

Vous êtes un `{role/personne}`. Votre tâche consiste à analyser la définition de la classe donnée et à l'assigner à l'une des couches architecturales suivantes du système :

**Couches :**

**- 1. Couche Utilisateur :** {description de la couche}

**- 2. Couche Logique :** {description de la couche}

**- 3. Couche Données :** {description de la couche}

**Informations sur la classe :**

**- Nom de la classe :** {class_name}

**- Fichier de la classe :** {class_file}

**Code de la classe :**

{class_code}

**Instruction :** Analysez le code de la classe et répondez avec la couche appropriée.

**Exemple de la première phase sur une application de test**

Pour illustrer notre processus de décomposition, nous avons développé un système monolithique de classification d'images basé sur le jeu de données CIFAR-10 (Krizhevsky, Nair & Hinton, 2009)[2]. Ce système intègre à la fois des composants spécifiques à l'apprentissage automatique et des éléments applicatifs généraux. Comme illustré à la figure 3.4, nous avons appliqué une décomposition horizontale et verticale : la première vise à séparer les classes liées à l'apprentissage automatique de celles qui ne le sont pas, tandis que la seconde consiste à classifier chaque classe selon les couches architecturales *Utilisateur*, *Logique* et *Données*. Cette structuration en couches clarifie les rôles fonctionnels de chaque composant et constitue une base essentielle pour les étapes d'encodage sémantique et de regroupement (clustering) qui

[2] https ://drive.google.com/drive/folders/13KAGgobgNukSnjVTpnPJl5-nCKvI7D44

suivent. Cette représentation hiérarchique favorise non seulement l'isolation des responsabilités fonctionnelles, mais établit également le socle conceptuel des phases ultérieures du pipeline de décomposition.

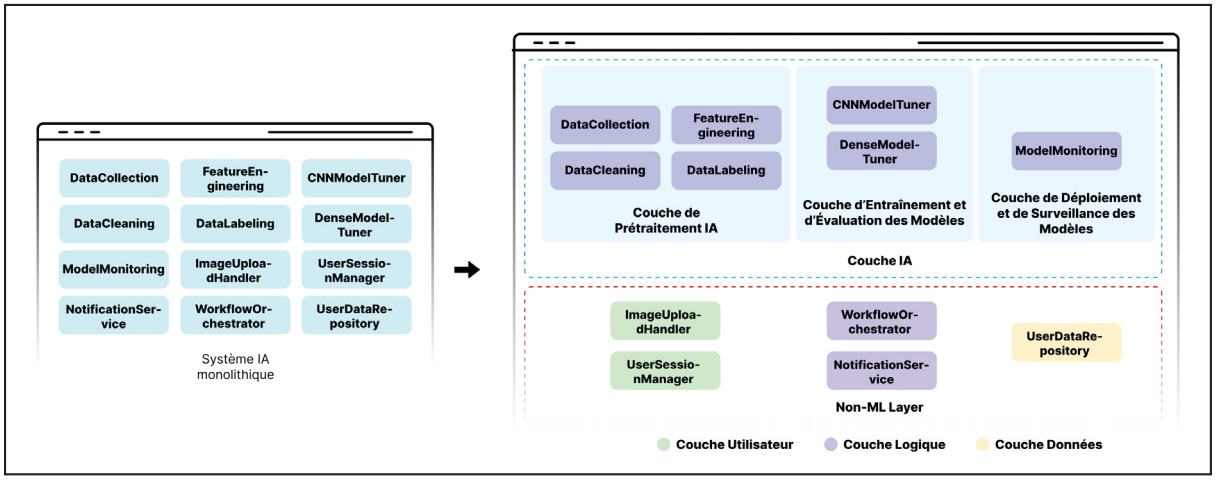


Figure 3.4 Exemple d'identification des couches sur une application de test

### 3.2.2 Phase 2 : Encodage des classes

**Entrée :** Code source de chaque classe.

**Sortie :** Matrice de caractéristiques à dimensions réduites contenant les embeddings CodeBERT de l'ensemble des classes.

**Description de la phase :** Dans cette deuxième phase du processus de décomposition, l'objectif est de transformer le code source des classes en représentations vectorielles exploitables pour les étapes analytiques ultérieures. Comme l'illustre le pipeline de la figure 3.5, cette transformation s'opère en plusieurs étapes successives et complémentaires. Tout d'abord, chaque classe est soumise à une tokenisation et un encodage à l'aide de CodeBERT (Feng *et al.*, 2020), un modèle de type transformer préentraîné spécifiquement pour l'analyse de code. Comme mentionné dans le Chapitre 1, ce modèle, entraîné sur un vaste corpus de programmes couvrant plusieurs langages, est capable de capturer non seulement la structure syntaxique mais également la dimension sémantique des fragments de code. Ainsi, l'encodage de chaque classe génère un vecteur qui reflète ses caractéristiques logiques et structurelles. Ces vecteurs

sont ensuite regroupés dans une matrice d'embeddings, où chaque ligne correspond à une classe et constitue une représentation riche en caractéristiques, apte à traduire les similarités et divergences entre entités logicielles. Toutefois, la dimensionnalité élevée de ces représentations peut introduire des problèmes de redondance et alourdir les calculs. Pour pallier à cette difficulté, nous appliquons une réduction de dimensionnalité via l'analyse en composantes principales (Khaledian, Ghadiridehkordi & Khaledian, 2025). Cette technique statistique permet d'identifier les axes les plus informatifs et de projeter les données dans un espace réduit, tout en conservant les relations sémantiques les plus pertinentes. Le résultat de cette transformation est une matrice compacte de caractéristiques, exportée sous forme de fichier CSV (valeurs séparées par des virgules), qui résume l'essentiel de l'information encodée par CodeBERT.

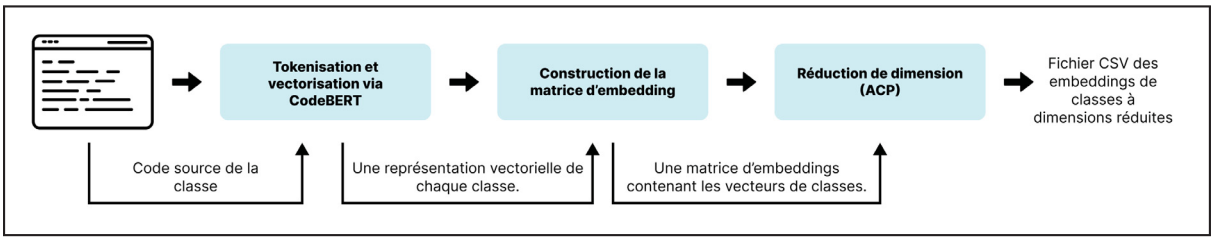


Figure 3.5 Phase 2 : Encodage des classes

Cette matrice, comme illustrée dans la figure 3.5, à dimension réduite constitue une base robuste et optimisée pour la suite du processus, notamment l'étape de clustering, où les classes seront regroupées en fonction de leurs similarités afin de guider la décomposition du système en microservices.

Le choix de CodeBERT pour cette tâche repose sur des considérations méthodologiques précises et mérite une justification approfondie. Contrairement aux approches classiques reposant sur des représentations lexicales (comme les n-grammes ou les TF-IDF), CodeBERT exploite la puissance des modèles de type transformer et a été entraîné conjointement sur du code source et de la documentation associée. Cette double modalité lui confère une capacité unique à capturer à la fois les structures syntaxiques explicites et les relations sémantiques implicites présentes dans le code. En pratique, cela signifie que CodeBERT est capable de générer des représentations vectorielles qui ne se limitent pas à des similitudes superficielles de tokens, mais qui traduisent

véritablement le rôle fonctionnel et logique des classes dans un système logiciel. De plus, son entraînement sur un large ensemble de langages de programmation lui permet de généraliser efficacement, même dans des contextes où le code est hétérogène ou complexe. En ce sens, CodeBERT constitue un outil de choix pour la création d'une matrice de caractéristiques riche et contextuelle, capable d'alimenter de manière robuste l'étape de clustering. Cette pertinence s'explique par sa faculté à fournir des embeddings sensibles au contexte, ce qui augmente considérablement la précision dans la détection des proximités sémantiques entre classes et, par extension, améliore la qualité de la décomposition en microservices.

**Exemple de l'encodage sémantique des classes** Afin d'illustrer la phase d'encodage sémantique de notre pipeline de décomposition, nous avons appliqué notre implémentation à la même application monolithique à petite échelle utilisée lors de la phase précédente. Dans cette étape, chaque classe a été traitée individuellement afin d'en extraire une représentation sémantique. À l'aide du modèle transformeur préentraîné, CodeBERT, nous avons encodé le code source de chaque classe sous la forme d'un vecteur de longueur fixe capturant à la fois ses caractéristiques structurelles et sémantiques. Chaque fichier de classe, chargé à partir d'un fichier CSV structuré, a été transmis au tokenizer et au modèle d'embedding de CodeBERT. L'état caché final du modèle a ensuite été moyenné pour produire un embedding unique par classe, résumant efficacement sa sémantique. Ces embeddings de haute dimension ont été agrégés dans une matrice de caractéristiques, sur laquelle nous avons appliqué une ACP afin de réduire la complexité computationnelle et d'éliminer les redondances. Cette projection dans un espace de plus faible dimension conserve les distances sémantiques significatives entre les classes et prépare les données pour la phase de regroupement (clustering) qui suit.

### 3.2.3 Phase 3 : Décomposition en microservices

**Entrée :** Matrice enrichie de caractéristiques combinant les étiquettes de classification des couches (provenant de la Phase 1) et les embeddings sémantiques des classes (issus de la Phase 2).

**Sortie :** Correspondance *Classe–Cluster*, où chaque classe est associée à un cluster spécifique

représentant un microservice candidat potentiel.

**Description de la phase :** Dans cette troisième et dernière phase du processus de décomposition, l'objectif est d'identifier des frontières cohérentes de microservices à partir des représentations vectorielles des classes obtenues lors des phases précédentes. Le point de départ est une matrice de caractéristiques enrichie, qui combine d'une part les étiquettes issues de la classification des couches (Phase 1, distinguant entre composantes liées à l'IA et composantes non-IA), et d'autre part les embeddings sémantiques produits par CodeBERT (Phase 2). Cette matrice fournit une représentation à la fois fonctionnelle et sémantique des classes du système, constituant ainsi une base solide pour la phase de regroupement.

Pour effectuer ce regroupement, nous avons recours à HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise), un algorithme de clustering hiérarchique fondé sur la densité. Cet algorithme présente plusieurs avantages qui le rendent particulièrement adapté à notre cas d'usage. Contrairement aux méthodes de clustering traditionnelles telles que k-means, HDBSCAN ne nécessite pas de définir au préalable le nombre de clusters, ce qui est essentiel dans un contexte où la granularité optimale des microservices n'est pas connue à l'avance. De plus, il est capable de détecter des clusters de formes arbitraires et de tailles hétérogènes, ce qui reflète mieux la réalité des systèmes logiciels complexes, où certaines parties du code peuvent se regrouper de manière dense (par exemple, des classes fortement couplées autour d'un module spécifique) tandis que d'autres sont plus diffuses.

Un autre atout majeur de HDBSCAN est sa capacité à gérer le bruit en identifiant explicitement des points (ici des classes) qui ne s'intègrent dans aucun cluster cohérent. Ces points sont alors considérés comme des éléments isolés, souvent symptomatiques de classes utilitaires ou de composants transversaux qui ne doivent pas nécessairement être intégrés à un microservice particulier. En outre, en construisant une hiérarchie de densité et en extrayant une partition optimale de cette hiérarchie, HDBSCAN offre une approche robuste qui équilibre à la fois la stabilité et la significativité des clusters produits.

Ainsi, l'application de HDBSCAN à notre matrice de caractéristiques permet de générer une carte de correspondance classe-cluster, où chaque cluster est interprété comme un candidat microservice. Cette approche garantit que les regroupements reposent non seulement sur des similarités sémantiques entre classes, mais aussi sur des proximités fonctionnelles identifiées par la classification des couches, assurant des frontières de microservices à la fois pertinentes et cohérentes avec la structure logique du système logiciel.

**Exemple de la décomposition finale** Dans cette dernière phase de notre approche de décomposition, nous avons appliqué l'algorithme de regroupement HDBSCAN aux vecteurs de caractéristiques réduits générés à la phase précédente à l'aide de CodeBERT et de l'ACP. L'entrée de cette phase correspond à la matrice enrichie de caractéristiques, qui encode les représentations sémantiques de chaque classe, enrichies par les annotations des couches architecturales. À partir de cette matrice d'embedding, HDBSCAN identifie des regroupements de classes sémantiquement similaires, que nous interprétons comme des candidats microservices.

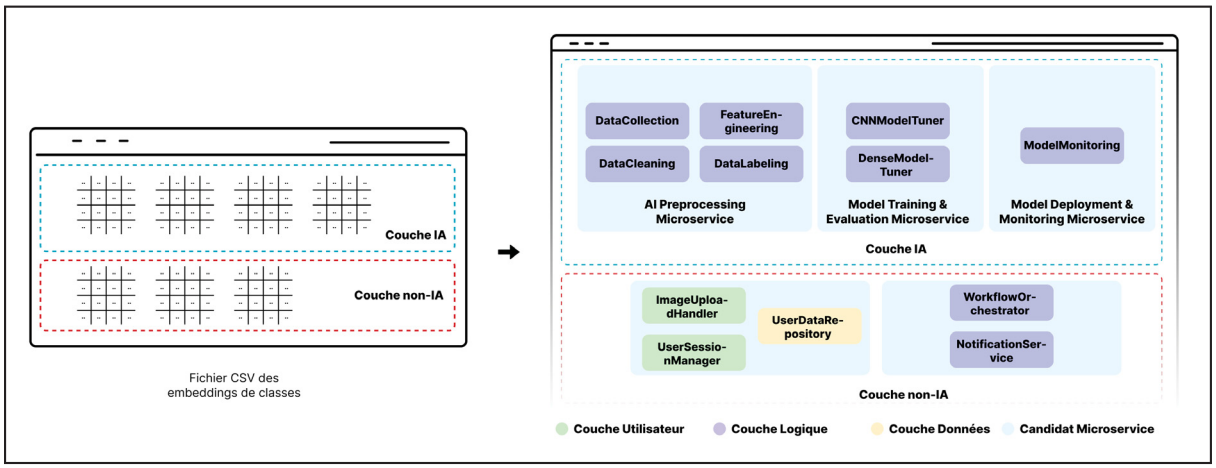


Figure 3.6 Exemple des microservices identifiés à travers une application test

Comme illustré à la figure 3.6, les clusters obtenus traduisent une partition du système fondée sur la sémantique et la cohérence fonctionnelle, en accord avec les principes des microservices tels que le faible couplage et la forte cohésion. Par exemple, les classes *DataCleaning*, *DataCollection*, *FeatureEngineering* et *DataLabeling* ont été regroupées au sein d'un microservice de prétraitement des données, tandis que *CNNModelTrainer* et *DenseModelTrainer*

ont formé un microservice d'entraînement de modèles. Le résultat final est un ensemble modulaire de candidats qui oriente la migration évolutive du monolithe CIFAR-10 vers une architecture microservices plus flexible et scalable.

## 3.3 Conclusion

Ce chapitre a présenté notre approche proposée, en s'appuyant sur une démarche systématique combinant les principes d'ingénierie logicielle, les patrons de conception et les spécificités propres aux systèmes d'apprentissage automatique. Après avoir posé les fondations conceptuelles de notre méthode, nous avons détaillé les trois phases principales qui la composent. La première phase a consisté à identifier l'architecture en couches et le pipeline d'apprentissage automatique du système monolithique, afin d'en comprendre la structure fonctionnelle et de repérer les dépendances essentielles. La deuxième phase a introduit un encodage des classes à l'aide de représentations vectorielles produites par CodeBERT, permettant de capturer la sémantique du code source et de préparer les données pour l'analyse de similarité. Enfin, la troisième phase a porté sur la décomposition en microservices, réalisée grâce à des techniques de réduction de dimension et de regroupement, aboutissant à l'identification d'ensembles de classes cohérents, faiblement couplés et fortement cohésifs. Tout au long du chapitre, nous avons intégré un exemple d'application illustrant concrètement chaque étape de l'approche proposée. Cet exemple a permis de simplifier la compréhension du processus global, d'en démontrer la faisabilité technique et de mettre en évidence la progression logique entre les différentes phases de décomposition. L'approche développée offre ainsi un cadre méthodologique robuste pour transformer des applications IA complexes en architectures microservices modulaires, favorisant la réutilisabilité, la maintenabilité et l'évolutivité du système.

Dans le chapitre suivant, nous procéderons à une évaluation de notre approche afin de répondre à la question de recherche QR2, portant sur l'efficacité de l'outil de décomposition proposé. Cette évaluation sera conduite sur plusieurs systèmes de référence issus de la littérature et comparée à des approches existantes de l'état de l'art, afin de positionner les performances de notre méthode et d'en démontrer la valeur ajoutée dans un contexte applicatif réel.

# CHAPITRE 4

# ÉVALUATION DE L'APPROCHE DE DÉCOMPOSITION PROPOSÉE

## 4.1 Introduction

Ce chapitre présente en détail le protocole d'évaluation élaboré pour mesurer l'efficacité et la pertinence de l'outil de décomposition proposé dans le cadre de ce travail. L'objectif est de fournir une réponse à la question de recherche *QR2 : Quelle est la performance de l'outil proposé pour la décomposition des systèmes IA en microservices ?* Dans un premier temps, nous introduisons les métriques d'évaluation retenues, issues de la littérature sur la migration des systèmes monolithiques vers les microservices. Ces métriques visent à mesurer la qualité structurelle des partitions générées, en prenant en compte aussi bien la cohésion interne des services que le couplage entre eux. Nous décrivons ensuite les études de cas et les systèmes de référence sélectionnés pour valider empiriquement notre approche. Ces cas d'étude couvrent des applications représentatives des systèmes IA monolithiques, permettant ainsi d'évaluer la généralisabilité de la méthode dans différents contextes. Enfin, nous confrontons les résultats obtenus à ceux produits par plusieurs outils de décomposition existants issus de l'état de l'art, afin de positionner notre outil par rapport aux solutions de référence et de mettre en évidence ses forces, ses limites et ses perspectives d'amélioration.

## 4.2 Métriques d'évaluation

Afin d'évaluer la qualité des décompositions produites par notre approche, nous avons retenu un ensemble de métriques issues de la littérature sur la décomposition en microservices. Ces métriques permettent d'analyser la qualité des partitions sous différents angles complémentaires, allant de la modularité conceptuelle à l'équilibre des tailles et à l'alignement avec un système de référence. Conceptual Modularity Quality (CMQ) (Jin *et al.*, 2021) est mobilisée pour estimer la cohésion intra-service et le couplage inter-service sur la base des similarités conceptuelles. Bien que formellement définie dans la section précédente, nous l'utilisons ici comme indicateur central de la qualité de structuration, un CMQ élevé reflétant des services à la fois cohésifs

et faiblement couplés. Non-Extreme Distribution (NED) (Desai *et al.*, 2021) complète cette analyse en évaluant l'équilibre de la répartition des classes entre les différents microservices. Cette métrique permet de vérifier que la décomposition évite des partitions trop petites ou trop volumineuses, favorisant ainsi une architecture plus stable et maintenable. Enfin, nous intégrons les métriques classiques de précision et de rappel, afin de mesurer l'alignement des partitions extraites avec une décomposition de référence. Ces mesures, largement utilisées dans la littérature, garantissent une comparaison objective avec les approches existantes et permettent de quantifier la justesse des frontières microservices proposées par notre outil.

## 4.3 Systèmes de validation et outils de décomposition

Nous avons évalué notre approche de décomposition basée sur les LLMs sur trois systèmes IA monolithiques, décrits dans le tableau 4.1 :

1. Un système d'analyse de texte développé en Java, comprenant 101 classes [1] ;
2. *Asparagus* [2], un système IA monolithique en Python dédié à l'échantillonnage, l'entraînement et l'application de modèles, contenant 73 classes ;
3. Une plateforme en Python de prédiction boursière et d'actualités, composée de 82 classes [3].

Ces systèmes ont été sélectionnées de manière à offrir un cadre d'évaluation diversifié, en variant la taille, le domaine d'application et le langage de programmation des systèmes.

Afin d'établir des décompositions en microservices de référence, deux étudiants en maîtrise avec une expérience significative en développement web ont conduit de manière indépendante une analyse manuelle itérative de chaque système. Leur tâche consistait (1) à associer chaque classe aux couches du système définies lors de la première phase de notre approche, et (2) à décomposer chaque système en microservices. Les résultats ont ensuite été confrontés et harmonisés au fil de

[1] https://github.com/sea-boat/TextAnalyzer

[2] https://github.com/MMunibas/Asparagus

[3] https ://github.com/LOG795-PFE031

cycles de révision jusqu'à atteindre un consensus. Cette décomposition de référence a servi de base pour mesurer à la fois la précision de classification des LLMs et la qualité structurelle des microservices candidats générés. L'ensemble des décompositions de référence est fourni dans notre paquet de réplication.

Tableau 4.1 Caractéristiques des applications IA utilisées

| **Application** | **Langage(s)** | **Nombre de classes** |
|---|---|---|
| *Asparagus* | Python | 73 |
| Stock & News Prediction | Python | 82 |
| Text Analyzer | Java | 101 |

## 4.4 Outils de décomposition de référence

Pour évaluer la performance de notre approche de décomposition en microservices, nous l'avons comparée à deux outils existants de décomposition de systèmes monolithiques. Un défi majeur de cette évaluation réside dans la limitation du nombre d'outils open source qui sont applicables à des systèmes basés sur l'apprentissage automatique. La plupart des solutions disponibles sont orientées vers des applications Java traditionnelles, ce qui limite leur applicabilité aux besoins particuliers des systèmes IA (Saucedo *et al.*, 2025).

Dans ce contexte, *ServiceCutter* (Gysel *et al.*, 2016) et *CoGCN* (Desai *et al.*, 2021) ont été retenus comme outils de comparaison, car ils répondent à nos critères de sélection : (1) ils sont indépendants du langage, (2) disponibles en open source, et (3) largement reconnus dans la littérature comme des références pour l'identification de microservices.

## 4.5 QR2 : Quelle est la performance de l'outil proposé dans la décomposition des systèmes IA en microservices ?

Cette section vise à répondre à la QR2 et évaluer l'efficacité de l'outil proposé dans le processus de décomposition des systèmes IA monolithiques en microservices. Cette analyse permettra de

dégager les points forts et les limites de l'approche, et de situer sa contribution par rapport aux outils de décomposition existants.

### 4.5.1 Identification et classification de la couche d'apprentissage automatique à l'aide des LLMs

Dans la première phase de notre approche, nous avons utilisé plusieurs LLMs pour classifier les composants du système selon les étapes du pipeline d'apprentissage automatique. Nous avons pris en considération quatre modèles open source dans le cadre de notre évaluation : *LLaMA-3.1-8B-Instant*, *LLaMA-3.3-70B-Versatile* développés par Meta, *Qwen3-32B* d'Alibaba Cloud, ainsi que *GPT OSS 20B* d'OpenAI. Ces modèles ont été choisis pour leur accessibilité, leurs bonnes performances en matière de raisonnement et d'analyse de code, ainsi que leur compatibilité avec des plateformes d'inférence à faible latence telles que Groq[4]. Ces caractéristiques les rendent particulièrement adaptés à l'analyse de larges bases de code et à la capture de dépendances sémantiques à longue portée. Tous les modèles ont suivi un format d'invite standardisé et des contraintes de tokens uniformes, garantissant ainsi une équité et une reproductibilité des résultats. La qualité des prédictions a été évaluée à l'aide des métriques de précision, de rappel et d'exactitude, en les comparant à la décomposition de référence de chaque système qui a été étiquetée par des experts pour les trois études de cas. Le modèle LLaMA-3.3-70B a été retenu pour la phase de décomposition, car il a produit les résultats les plus précis et les plus cohérents sur le plan sémantique.

Tableau 4.2 Performance de classification des LLMs sur trois systèmes de référence

| **Modèle** | **Asparagus** | | | **Stock & News Prediction** | | | **Text Analyzer** | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Préc.** | **Rappel** | **Exact.** | **Préc.** | **Rappel** | **Exact.** | **Préc.** | **Rappel** | **Exact.** |
| LLaMA 3.1-8B | 62% | 69% | 81% | 60% | 60% | 77% | 47% | 47% | 70% |
| Qwen3-32B | 42% | 46% | 69% | 52% | 52% | 72% | 43% | 43% | 67% |
| GPT-OSS-20B | 73% | 80% | 87% | 77% | 77% | 87% | 64% | 64% | 79% |
| LLaMA 3.3-70B | **74%** | **82%** | **88%** | **80%** | **80%** | **89%** | **65%** | **65%** | **80%** |

[4] https://console.groq.com

Les résultats présentés dans la table 4.2 montrent que le modèle LLaMA-3.3-70B surpasse les autres modèles sur l'ensemble des trois systèmes de référence. Cette supériorité peut être attribuée à plusieurs facteurs liés à la taille et à la capacité d'apprentissage du modèle. En effet, les modèles de grande taille disposent d'un nombre plus élevé de paramètres, leur permettant de mieux capturer les dépendances contextuelles et les relations sémantiques complexes présentes dans le code source. Cette expressivité accrue favorise une compréhension plus fine des structures architecturales implicites, facilitant la distinction entre les couches fonctionnelles (données, logique, utilisateur) et les composants propres à l'apprentissage automatique. De plus, LLaMA-3.3-70B bénéficie d'un pré-entraînement sur un corpus de code et de texte diversifié, ce qui améliore sa capacité à raisonner sur des schémas de conception et des conventions de nommage fréquemment utilisés dans les systèmes logiciels. Cette richesse contextuelle réduit les erreurs de classification sémantique et renforce la cohérence de la décomposition en microservices, en alignant les classes selon leurs rôles fonctionnels et leurs responsabilités logiques. Ainsi, la performance accrue observée chez les modèles de grande taille ne se limite pas à une amélioration quantitative des métriques (précision, rappel, exactitude), mais traduit une meilleure compréhension conceptuelle du code, essentielle pour garantir la justesse et la stabilité de la décomposition proposée.

### 4.5.2 Évaluation de l'approche avec les outils de décomposition existants

Dans cette section, nous comparons la performance de notre approche avec celles des principales méthodes de décomposition dans l'état de l'art. Nous avons appliqué notre approche ainsi que les deux outils de référence CoGCN et ServiceCutter aux trois études de cas. La qualité des décompositions obtenues a été évaluée à l'aide de deux catégories de métriques : (1) la précision et le rappel, calculés par rapport au système de réference, et (2) les métriques CMQ et NED, sélectionnées comme indicateurs principaux de la qualité architecturale de la décomposition. Le choix de ces deux dernières métriques se justifie par leur complémentarité et leur pertinence directe vis-à-vis des objectifs de notre approche. La métrique CMQ permet d'évaluer la cohérence conceptuelle des microservices générés, en mesurant le degré d'alignement

sémantique et fonctionnel entre les classes appartenant à un même cluster. Cette mesure traduit la capacité de notre approche à produire des services fortement cohésifs et faiblement couplés, conformément aux principes fondamentaux des architectures microservices.

Comme le montre le tableau 4.3, notre approche a obtenu une précision moyenne de 84% et un rappel de 65%, surpassant nettement les outils de référence. En comparaison, ServiceCutter n'a atteint qu'une précision de 32% et un rappel de 38%, tandis que CoGCN a obtenu une précision de 65% mais seulement 20% de rappel, révélant une capacité limitée à identifier correctement la totalité des frontières de microservices. Ces écarts confirment la capacité de notre approche à produire des décompositions plus précises, cohérentes et complètes, traduisant une meilleure compréhension structurelle et sémantique du code source.

Sur le plan qualitatif, notre méthode a également produit des valeurs de CMQ plus élevées et des scores de NED mieux équilibrés, traduisant des microservices fortement cohésifs et faiblement couplés. Plus précisément, notre approche atteint un CMQ moyen de 87% et un NED de 54%, alors que CoGCN obtient un CMQ de 79% et un NED de 61%, et ServiceCutter un CMQ de 42% pour un NED de 76%.

Ces résultats montrent que les microservices générés par notre pipeline sont à la fois sémantiquement alignés et structurellement stables, en accord avec les principes de modularité et de séparation des responsabilités.

Bien que CoGCN obtienne un CMQ relativement compétitif (79%), ses valeurs de précision et de rappel restent faibles, notamment sur les systèmes Asparagus (précision : 73%, rappel : 80%) et TextAnalyzer (précision : 64%, rappel : 64%). Ces performances suggèrent des frontières de microservices parfois mal alignées et une cohérence interne inégale. De plus, la génération de ses matrices d'entrée demeure complexe pour les systèmes développés en Python, contrairement à Java, où plusieurs outils facilitent cette extraction et automatisent la construction des graphes de dépendances.

Quant à ServiceCutter, il présente les résultats les plus faibles : son CMQ de 42% et son NED de 76% reflètent une sur-décomposition, qui réduit la cohésion interne des services et augmente le couplage entre composants. Ce comportement, déjà documenté dans la littérature, illustre la difficulté de ServiceCutter à produire un niveau de granularité adéquat lorsque l'outil est appliqué à des systèmes complexes intégrant des composants d'apprentissage automatique.

Ces résultats mettent en évidence la force de notre approche en matière de compréhension sémantique et d'identification des couches architecturales, permettant de distinguer efficacement les composants liés à l'apprentissage automatique de ceux qui ne le sont pas. Cette capacité favorise la production de décompositions plus cohérentes, guidées par les LLMs et les patrons de conception architecturaux. Bien que les valeurs de rappel suggèrent encore certaines difficultés à traiter les classes ambiguës ou chevauchantes, la précision constamment élevée confirme la capacité de notre méthode à produire des décompositions de systèmes IA monolithiques fiables et pertinentes sur le plan architectural.

Tableau 4.3 Comparaison des performances de décomposition sur les systèmes de référence

| Outil | Asparagus | | | | Stock & News Prediction | | | | Text Analyzer | | | | Valeurs moyennes | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Préc.↑** | **Rappel↑** | **CMQ↑** | **NED↓** | **Préc.↑** | **Rappel↑** | **CMQ↑** | **NED↓** | **Préc.↑** | **Rappel↑** | **CMQ↑** | **NED↓** | **Préc.↑** | **Rappel↑** | **CMQ↑** | **NED↓** |
| Notre approche | **0.89** | 0.62 | 0.985 | 0.5 | **0.80** | **0.50** | **0.99** | **0.39** | **0.85** | **0.84** | **0.995** | 0.554 | **0.846** | **0.653** | **0.99** | **0.478** |
| CoGCN | 0.69 | 0.23 | **0.99** | **0.472** | 0.72 | 0.16 | 0.869 | 0.741 | 0.56 | 0.21 | 0.99 | 0.812 | 0.656 | 0.20 | 0.95 | 0.675 |
| ServiceCutter | 0.45 | **0.70** | 0.272 | 1.000 | 0.31 | 0.37 | 0.614 | 0.852 | 0.20 | 0.07 | 0.347 | **0.505** | 0.320 | 0.380 | 0.411 | 0.786 |

## 4.6 Discussions et menaces à la validité

Cette section présente une discussion critique des résultats obtenus ainsi qu'une analyse des limites de validité de notre étude. Elle vise à interpréter les performances observées de l'approche proposée, à en dégager les principaux enseignements, et à identifier les facteurs susceptibles d'influencer la validité interne, externe et de construction de nos conclusions.

### 4.6.1 Discussions

À travers nos expérimentations, nous avons constaté que les LLMs se révèlent efficaces pour analyser sémantiquement le code source et identifier les différentes couches architecturales au

sein des systèmes monolithiques à base d'apprentissage automatique. Les modèles, tels que LLama 3.3-70B et GPT-OSS-20B, ont montré une forte capacité à comprendre le rôle des classes et à les associer avec précision aux étapes spécifiques du pipeline IA. Notre première étape, guidée par des patrons architecturaux, a notamment permis d'isoler la couche IA du reste du système, constituant ainsi une base solide pour la classification et la décomposition ultérieures.

Toutefois, l'obtention de résultats consistants et de haute qualité a nécessité un affinage itératif de la stratégie de prompting. Les premières formulations, trop vagues ou insuffisamment spécifiées, entraînaient des erreurs de classification notables, soulignant la sensibilité des réponses des LLMs à la qualité des prompts. Pour y remédier, nous avons adapté et enrichi le gabarit structuré proposé par (White *et al.*, 2023), en l'ajustant au vocabulaire et au contexte propres aux composants des pipelines ML, tout en modulant les paramètres (comme la température) et en multipliant les itérations. Néanmoins, nous reconnaissons que l'ingénierie des prompts demeure un facteur déterminant, et que des approches futures pourraient explorer l'optimisation automatique ou l'adaptation dynamique des prompts pour améliorer encore l'efficacité de l'approche.

Malgré la précision élevée atteinte par notre méthode, nous observons toutefois un rappel plus faible dans certaines configurations, indiquant que certaines classes pertinentes n'ont pas été correctement associées à leur couche architecturale ou à leur rôle fonctionnel. Ce phénomène s'explique principalement par deux facteurs : (1) la variabilité du style et du nommage des classes dans les projets open source, qui limite la généralisation des LLMs à des contextes hétérogènes, et (2) la dépendance du modèle au contenu explicite du code, les classes peu documentées ou faiblement typées étant plus difficiles à interpréter sémantiquement. Pour atténuer cette limitation, plusieurs pistes peuvent être envisagées : enrichir le contexte d'entrée du LLM avec des commentaires, dépendances ou graphes d'appels, combiner l'analyse textuelle avec des représentations structurelles (AST, embeddings hybrides), ou encore recourir à des LLMs spécialisés dans le code dotés d'un contexte d'inférence plus large.

Enfin, nous soulignons que la production de décompositions plus précises, en adéquation avec les vérités terrain d'experts et générant des microservices fortement cohésifs et faiblement couplés,

constitue un socle essentiel pour améliorer la maintenabilité et l'évolutivité des systèmes. Notre approche a démontré sa capacité à capturer et séparer les principales composantes fonctionnelles du pipeline IA (prétraitement des données, entraînement, évaluation) tout en regroupant efficacement les classes non liées à la couche IA, renforçant ainsi des distinctions structurelles nettes dans la décomposition finale. Ces résultats confirment que la combinaison d'une analyse assistée par LLM et d'une décomposition guidée par des patrons architecturaux permet d'obtenir des frontières microservices cohérentes, alignées à la fois sur les rôles sémantiques et sur la logique fonctionnelle des pipelines IA.

### 4.6.2 Menaces à la validité

Bien que nos résultats mettent en évidence le potentiel des LLMs et des stratégies guidées par des patrons architecturaux pour la décomposition de systèmes monolithiques à base de IA, plusieurs menaces à la validité doivent être reconnues.

La première concerne la généralisabilité. Afin de la limiter, nous avons sélectionné un ensemble diversifié d'études de cas, variant à la fois par leur domaine d'application, leur langage de programmation (Java et Python) et leur taille. Cette diversité contribue à renforcer la portée de nos conclusions. Néanmoins, des travaux futurs devront étendre l'évaluation à un éventail plus large de systèmes afin d'accroître encore la généralisabilité des résultats.

La deuxième menace découle du fait que notre évaluation repose principalement sur l'analyse sémantique du code source. Or, de nombreuses approches de la littérature s'appuient sur des artefacts spécifiques au domaine, tels que des diagrammes entité-relation, des modèles UML ou encore des spécifications de services, pour guider l'identification des frontières microservices (Saucedo *et al.*, 2025). Dans la mesure où nos systèmes de référence n'incluent pas ce type d'artefacts, il n'a pas été possible de réaliser une comparaison directe avec ces approches. Nous explorons cependant activement des pistes visant à évaluer notre outil vis-à-vis de techniques de décomposition exploitant ces formes d'entrées.

Enfin, une troisième menace est liée au non-déterminisme induit par l'utilisation des LLMs. Pour y remédier, nous avons mis en œuvre des techniques de raffinement des prompts et réduit le paramètre de température afin de limiter la variabilité des sorties. Chaque LLM a été exécuté cinq fois, et pour chaque classe des systèmes analysés, nous avons retenu la catégorie prédite le plus fréquemment par un vote majoritaire. Bien que nous ayons observé une variabilité faible dans les résultats, des travaux ultérieurs pourraient explorer des techniques de stabilisation supplémentaires afin de mieux contrôler la stochasticité des comportements des LLMs.

## 4.7 Conclusion

Nos expérimentations mettent en évidence l'efficacité de l'approche proposée pour la décomposition des systèmes monolithiques à base d'intelligence artificielle. En s'appuyant sur des patrons architecturaux pour séparer les couches logicielles et les composants IA, puis sur l'utilisation des LLMs afin de classifier les rôles des classes dans le pipeline d'apprentissage, nous avons pu générer des microservices candidats cohérents grâce au regroupement par HDBSCAN.

Appliquée à trois systèmes IA monolithiques distincts et comparée à deux outils de référence de l'état de l'art, notre approche a atteint une précision de 84% et un rappel de 65%. Ces résultats montrent qu'elle surpasse les approches existantes, tout en confirmant sa capacité à identifier correctement les frontières fonctionnelles des microservices.

Enfin, ces résultats ouvrent des perspectives intéressantes : l'intégration des LLMs supplémentaires dans la phase de classification, l'évaluation qualitative des décompositions avec des développeurs, et l'étude de techniques de paquetage automatisé pour faciliter l'intégration des microservices identifiés dans une plateforme cible basée sur les microservices.

# CONCLUSION ET RECOMMANDATIONS

Cette recherche s'inscrit dans le cadre de la migration des systèmes IA monolithiques vers des architectures en microservices, un enjeu majeur de l'ingénierie logicielle moderne. Face à la complexité croissante des systèmes IA et à la nécessité d'assurer leur scalabilité, maintenabilité et réutilisabilité, notre travail vise à proposer une approche systématique et automatisée de décomposition, tout en évaluant empiriquement les outils existants et en identifiant leurs limites.

Dans un premier temps, une recherche de la littérature a permis d'examiner les approches de décomposition existantes appliquées aux systèmes monolithiques, en mettant en lumière leurs fondements, algorithmes et métriques d'évaluation. Cette étude a révélé une absence de méthodes spécifiquement adaptées aux architectures d'apprentissage automatique, où la présence d'étapes telles que la prétraitement des données, l'entraînement, et la prédiction introduit des dépendances et des structures propres aux pipelines IA.

Ensuite, une évaluation empirique des principaux outils de décomposition (tels que MAGNET, FoSCI, CoGCN, MSExtractor, ServiceCutter et Mono2Micro) a été menée afin de comparer leurs performances sur différents systèmes monolithiques. Les résultats ont montré que chaque outil présente des forces et limites distinctes. MAGNET et FoSCI se distinguent par leur capacité à capturer les dépendances fines et à gérer les systèmes de grande taille, CoGCN, MSExtractor et Mono2Micro offrent un meilleur équilibre entre précision, rappel et découplage fonctionnel, ce qui les rend potentiellement plus adaptés aux contextes de migration à grande échelle. Cette étude comparative a ainsi permis de dégager des renseignements importants sur les compromis entre granularité, précision et scalabilité, offrant un cadre de référence pour orienter le choix des outils selon les besoins et la nature du système à migrer.

Sur la base de ces constats, nous avons proposé une approche automatisée de décomposition des systèmes monolithiques basés sur l'apprentissage automatique, combinant les patrons architecturaux et l'analyse sémantique assistée par les LLMs. Notre méthode exploite des

patrons de conception pour séparer les couches architecturales et les composants d'apprentissage automatique, tandis que les LLMs sont utilisés pour classifier les rôles des classes et identifier les microservices candidats via des regroupements sémantiques fondés sur des embeddings préentraînés. Les expériences menées sur trois systèmes monolithiques d'apprentissage automatique ont démontré l'efficacité de cette approche, avec une précision moyenne de 84% et un rappel de 65%, surpassant les deux outils de référence utilisés comme baselines. Ces résultats confirment la pertinence de l'intégration des LLMs et des patrons architecturaux dans les processus de décomposition, permettant de générer des microservices cohérents, faiblement couplés et sémantiques alignés.

Nos observations ont également montré que les LLMs peuvent analyser sémantiquement le code source et identifier les couches architecturales dans les systèmes IA avec une grande efficacité, à condition d'un raffinement itératif des prompts. L'adaptation de gabarits structurés, la calibration des paramètres de génération (température, cohérence lexicale) et la répétition des itérations se sont révélées cruciales pour améliorer la précision et réduire les erreurs de classification. Ces ajustements soulignent l'importance du prompt engineering et ouvrent la voie à des travaux futurs sur l'automatisation du processus de génération de prompts.

Bien que les résultats obtenus soient prometteurs, certaines limites demeurent. D'une part, la dépendance aux modèles de langage soulève des questions de robustesse et de reproductibilité, notamment en raison de la variabilité des réponses selon le contexte et les paramètres. D'autre part, l'évaluation s'est concentrée sur un ensemble restreint de systèmes IA monolithiques, ce qui invite à élargir le corpus expérimental pour confirmer la généralisabilité de l'approche.

Pour les travaux futurs, plusieurs axes de recherche sont envisagés. Tout d'abord, l'approche pourrait être étendue à d'autres types d'artefacts logiciels, tels que les tests, les fichiers de configuration ou encore les scripts d'orchestration, afin d'obtenir une décomposition plus complète et représentative du système global. Ensuite, une évaluation qualitative des

décompositions sera menée en collaboration avec des développeurs et des architectes logiciels, dans le but de valider la pertinence et la cohérence des regroupements identifiés par l'outil. Enfin, un axe de recherche prometteur consistera à automatiser le conditionnement et le déploiement des microservices générés, dans une optique d'intégration continue et de migration assistée de bout en bout.

Finalement, cette recherche propose la première approche de décomposition dédiée aux systèmes IA monolithiques en microservices. En combinant les principes d'ingénierie logicielle, les patrons d'architecture, et la puissance des modèles de langage, elle propose une approche innovante et reproductible capable d'identifier et décomposer les systèmes IA monolithiques. Ce travail ouvre des perspectives prometteuses pour l'automatisation de la décomposition, la gouvernance des architectures IA distribuées, et la mise en place d'un écosystème unifié reliant ingénierie logicielle traditionnelle et intelligence artificielle, un pas significatif vers la prochaine génération d'architectures logicielles intelligentes.

# ANNEXE I

# VUE D'ENSEMBLE DES OUTILS DE DÉCOMPOSITION EN MICROSERVICES

Tableau-A I-1 Vue d'ensemble des outils de décomposition en microservices

| Titre | Année |
|---|---|
| Service Cutter : A Systematic Approach to Service Decomposition (Gysel *et al.*, 2016) | 2016 |
| Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems (Levcovitz, Terra & Valente, 2016) | 2016 |
| Workload-based Clustering of Coherent Feature Sets in Microservice Architectures (Klock, van der Werf, Guelen & Jansen, 2017) | 2017 |
| Extraction of microservices from monolithic software architectures (Mazlami, Cito & Leitner, 2017) | 2017 |
| Microservices Identification Through Interface Analysis (Baresi, Garriga & Renzis, 2017) | 2017 |
| MicroART : A Software Architecture Recovery Tool for Maintaining Microservice-based Systems (Granchelli *et al.*, 2017) | 2017 |
| FoME : Functionality-oriented Microservice Extraction Based on Execution Trace Clustering (Jin, Liu, Zheng, Cui & Cai, 2018) | 2018 |
| Extracting Candidates of Microservices from Monolithic Application Code (Kamimura, Yano, Hatano & Matsuo, 2018) | 2018 |
| Fo-SCI : Service Candidate Identification from Monolithic Systems based on Execution Traces (Jin *et al.*, 2021) | 2021 |
| System Decomposition to Optimize Functionality Distribution in Microservices with Rule Based Approach (Eyitemi & Reiff-Marganiec, 2020) | 2020 |
| Mono2Micro : A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices (Kalia *et al.*, 2021) | 2021 |

Suite à la page suivante

Tableau-A I-1 – suite de la page précédente

| Titre | Année |
|---|---|
| CO-GCN : Graph Neural Network to Dilute Outliers for Refactoring Monolith Applications (Desai *et al.*, 2021) | 2021 |
| Monolith to Microservice Candidates using Business Functionality Inference (Agarwal *et al.*, 2021) | 2021 |
| A Multi-Criteria Strategy for Redesigning Legacy Features as Microservices : An Industrial Case Study (Assunção *et al.*, 2021) | 2021 |
| Identification of microservices from monolithic applications through topic modeling (Brito *et al.*, 2021) | 2021 |
| Monolith to Microservices : Representing Application Software through Heterogeneous Graph Neural Network (Mathai, Bandyopadhyay, Desai & Tamilselvam, 2022) | 2021 |
| Improving Microservices Extraction using Evolutionary Search (Sellami *et al.*, 2022a) | 2022 |
| A Hierarchical-DBSCAN Method for Extracting Microservices from Monolithic Applications (Sellami, Saied & Ouni, 2022b) | 2022 |
| AI-Guided Dependency Analysis for Migrating Monolithic Applications to Microservices Architecture (Nitin, Asthana, Ray & Krishna, 2023) | 2022 |
| From Legacy to Microservices : A Type-Based Approach for Microservices Identification using Machine Learning and Semantic Analysis (Trabelsi *et al.*, 2023) | 2022 |
| Combining Static and Dynamic Analysis to Decompose Monolithic Applications into Microservices (Sellami, Saied, Ouni & Abdalkareem, 2022c) | 2022 |
| An Expert System for Redesigning Software for Cloud Applications (Yedida *et al.*, 2023) | 2022 |

Suite à la page suivante

Tableau-A I-1 – suite de la page précédente

| Titre | Année |
|---|---|
| Monolith Development History for Microservices Identification : A Comparative Analysis (Lourenço & Silva, 2023) | 2023 |
| Microservice extraction based on knowledge graph from monolithic applications (Li, Shang, Wu & Li, 2022) | 2022 |
| Monolith to Microservices : Representing Application Software through Heterogeneous Graph Neural Network (Mathai *et al.*, 2022) | 2022 |
| From Monolithic to Microservice Architecture—An Automated Approach based on Graph Clustering and Combinatorial Optimization (Filippone *et al.*, 2023) | 2023 |
| Automatically Refactoring Application Transactions for Microservice-oriented Architecture (Ishida, Katsuno, Tozawa & Saito, 2023) | 2023 |
| Magnet : Method-Based Approach Using Graph Neural Network for Microservices Identification (Trabelsi *et al.*, 2024a) | 2024 |
| GTMicro : Microservice Identification Approach based on Deep NLP Transformer Model for Greenfield Developments (Bajaj, Bharti, Gupta, Gupta & Yadav, 2024) | 2024 |
| Fully Automated Microservices Identification Approach from Monolithic Systems (Trabelsi, Popa, Pereyrol, Beaulieu & Moha, 2024b) | 2024 |
| Refactoring Microservices to Microservices in Support of Evolutionary Design (Zhong *et al.*, 2025) | 2024 |
| Using Static Analysis to Aid Monolith to Microservice System Transformation : Tuning Fuzzy c-Means in a VAE-Based GNN Approach (Sooksatra *et al.*, 2024) | 2024 |
| An Innovated Microservices Identification Approach Based on Database and Source Code Analysis (Ashraf, Yousef & Mahdi, 2024) | 2024 |
| Extracting Microservices from Monolithic Systems using Deep Reinforcement Learning (Sellami & Saied, 2024) | 2025 |

## LISTE DE RÉFÉRENCES


Abdellatif, M., Shatnawi, A., Mili, H., Moha, N., Boussaidi, G. E., Hecht, G., Privat, J. & Guéhéneuc, Y.-G. (2021). A Taxonomy of Service Identification Approaches for Legacy Software Systems Modernization. *Journal of Systems and Software*, 173, 110868.

Abgaz, Y., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., Jackson, G., Yilmaz, M., Buckley, J. & Clarke, P. (2023). Decomposition of Monolith Applications Into Microservices Architectures : A Systematic Review. *IEEE Transactions on Software Engineering*, 49(8), 4213–4242.

Agarwal, S., Sinha, R., Sridhara, G., Das, P., Desai, U., Tamilselvam, S., Singhee, A. & Nakamuro, H. (2021). Monolith to Microservice Candidates Using Business Functionality Inference. *2021 IEEE International Conference on Web Services (ICWS)*, 758–763.

Akkaya, K. & Ovatman, T. (2022). A Comparative Study of Meta-Data-Based Microservice Extraction Tools. *International Journal of Service Science, Management, Engineering, and Technology (IJSSMET)*, 13(1), 26.

Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B. & Zimmermann, T. (2019). Software Engineering for Machine Learning : A Case Study. *2019 IEEE/ACM 41st International Conference on Software Engineering : Software Engineering in Practice (ICSE-SEIP)*, 291-300.

Ashraf, A., Yousef, A. H. & Mahdi, H. (2024). An Innovated Microservices Identification Approach Based on Database and Source Code Analysis. *2024 6th Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, 463–468.

Assunção, W. K. G., Colanzi, T. E., Carvalho, L., Pereira, J. A., Garcia, A., de Lima, M. J. & Lucena, C. (2021). A Multi-Criteria Strategy for Redesigning Legacy Features as Microservices : An Industrial Case Study. *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 377–387.

Bajaj, D., Bharti, U., Gupta, I., Gupta, P. & Yadav, A. (2024). GTMicro : Microservice Identification Approach Based on Deep NLP Transformer Model for Greenfield Developments. *International Journal of Information Technology*, 16(5), 2751–2761.

Baresi, L., Garriga, M. & Renzis, A. D. (2017). Microservices Identification Through Interface Analysis. *Service-Oriented and Cloud Computing*, 19–33.

Brito, M., Cunha, J. & Saraiva, J. (2021). Identification of Microservices from Monolithic Applications Through Topic Modelling. *Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC '21)*, 1409–1418.

Campello, R. J. G. B., Moulavi, D. & Sander, J. (2013). Density-Based Clustering Based on Hierarchical Density Estimates. *Advances in Knowledge Discovery and Data Mining*, 160–172.

Chinimilli, V. R. P. & Sadasivuni, L. (2024). The Rise of AI : A Comprehensive Research Review. *IAES International Journal of Artificial Intelligence (IJ-AI)*, 13, 2226–2235. doi : 10.11591/ijai.v13.i2.pp2226-2235.

Desai, U., Bandyopadhyay, S. & Tamilselvam, S. (2021). Graph Neural Network to Dilute Outliers for Refactoring Monolith Application. *arXiv preprint arXiv :2102.03827*.

Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. (2019). BERT : Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv :1810.04805*.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. & Safina, L. (2017). Microservices : Yesterday, Today, and Tomorrow. *arXiv preprint arXiv :1606.04036*.

Eyitemi, F.-D. & Reiff-Marganiec, S. (2020). System Decomposition to Optimize Functionality Distribution in Microservices with Rule Based Approach. *2020 IEEE International Conference on Service Oriented Systems Engineering (SOSE)*, 65–71.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D. & Zhou, M. (2020). CodeBERT : A Pre-Trained Model for Programming and Natural Languages. *arXiv preprint arXiv :2002.08155*.

Filippone, G., Mehmood, N. Q., Autili, M., Rossi, F. & Tivoli, M. (2023). From Monolithic to Microservice Architecture : An Automated Approach Based on Graph Clustering and Combinatorial Optimization. *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, 47–57.

Fowler, M. & Lewis, J. (2014). Microservices. *martinfowler.com*. Available at : https://martinfowler.com/articles/microservices.html.

Fritzsch, J., Bogner, J., Zimmermann, A. & Wagner, S. (2019). From Monolith to Microservices : A Classification of Refactoring Approaches. *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, 128–141.

Granchelli, G., Cardarelli, M., Francesco, P. D., Malavolta, I., Iovino, L. & Salle, A. D. (2017). MicroART : A Software Architecture Recovery Tool for Maintaining Microservice-Based Systems. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 298–302.

Gravanis, D., Kakarontzas, G. & Gerogiannis, V. (2022). You Don't Need a Microservices Architecture (Yet) : Monoliths May Do the Trick. *Proceedings of the 2021 European Symposium on Software Engineering (ESSE '21)*, 39–44.

Gysel, M., Kölbener, L., Giersche, W. & Zimmermann, O. (2016). Service Cutter : A Systematic Approach to Service Decomposition. *Service-Oriented and Cloud Computing*, 185–200.

Ishida, A., Katsuno, Y., Tozawa, A. & Saito, S. (2023). Automatically Refactoring Application Transactions for Microservice-Oriented Architecture. *2023 IEEE International Conference on Software Services Engineering (SSE)*, 210–219.

Jin, W., Liu, T., Zheng, Q., Cui, D. & Cai, Y. (2018). Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering. *2018 IEEE International Conference on Web Services (ICWS)*, 211–218.

Jin, W., Liu, T., Cai, Y., Kazman, R., Mo, R. & Zheng, Q. (2021). Service Candidate Identification from Monolithic Systems Based on Execution Traces. *IEEE Transactions on Software Engineering*, 47(5), 987–1007.

Kalia, A. K., Xiao, J., Lin, C., Sinha, S., Rofrano, J., Vukovic, M. & Banerjee, D. (2020). Mono2Micro : An AI-Based Toolchain for Evolving Monolithic Enterprise Applications to a Microservice Architecture. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 1606–1610.

Kalia, A. K., Xiao, J., Krishna, R., Sinha, S., Vukovic, M. & Banerjee, D. (2021). Mono2Micro : A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*, 1214–1224.

Kalske, M., Mäkitalo, N. & Mikkonen, T. (2018). Challenges When Moving from Monolith to Microservice Architecture. *Current Trends in Web Engineering*, 32–47.

Kamimura, M., Yano, K., Hatano, T. & Matsuo, A. (2018). Extracting Candidates of Microservices from Monolithic Application Code. *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, 571–580.

Khaledian, A., Ghadiridehkordi, A. & Khaledian, N. (2025). PCA-RAG : Principal Component Analysis for Efficient Retrieval-Augmented Generation. *arXiv preprint arXiv :2504.08386*. Available at : https://arxiv.org/abs/2504.08386.

Kitchenham, B. (2004). Procedures for Performing Systematic Reviews. *Keele University Technical Report*, 33, 1–26.

Klock, S., van der Werf, J. M. E. M., Guelen, J. P. & Jansen, S. (2017). Workload-Based Clustering of Coherent Feature Sets in Microservice Architectures. *2017 IEEE International Conference on Software Architecture (ICSA)*, 11–20.

Krizhevsky, A., Nair, V. & Hinton, G. (2009). The CIFAR-10 Dataset. *Technical Report, University of Toronto*. Available at : https://www.cs.toronto.edu/~kriz/cifar.html.

Levcovitz, A., Terra, R. & Valente, M. T. (2016). Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. *arXiv preprint arXiv :1605.03175*.

Li, Z., Shang, C., Wu, J. & Li, Y. (2022). Microservice Extraction Based on Knowledge Graph from Monolithic Applications. *Information and Software Technology*, 150, 106992.

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H. & Neubig, G. (2021). Pre-train, Prompt, and Predict : A Systematic Survey of Prompting Methods in Natural Language Processing. *arXiv preprint arXiv :2107.13586*.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L. & Stoyanov, V. (2019). RoBERTa : A Robustly Optimized BERT Pretraining Approach. *arXiv preprint arXiv :1907.11692*.

Lourenço, J. & Silva, A. R. (2023). Monolith Development History for Microservices Identification : A Comparative Analysis. *2023 IEEE International Conference on Web Services (ICWS)*, 50–56.

Maggi, K., Verdecchia, R., Scommegna, L. & Vicario, E. (2024). CLAIM : A Lightweight Approach to Identify Microservices in Dockerized Environments. *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE '24)*, 357–362.

Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y. & Gansner, E. R. (1998). Using Automatic Clustering to Produce High-Level System Organizations of Source Code. *Proceedings of the 6th International Workshop on Program Comprehension (IWPC '98)*, 45–52.

Mathai, A., Bandyopadhyay, S., Desai, U. & Tamilselvam, S. (2022). Monolith to Microservices : Representing Application Software Through Heterogeneous Graph Neural Network. *arXiv preprint arXiv :2112.01317*.

Mazlami, G., Cito, J. & Leitner, P. (2017). Extraction of Microservices from Monolithic Software Architectures. *2017 IEEE International Conference on Web Services (ICWS)*, 524–531.

Media, O. (2020). Microservices Adoption in 2020. *O'Reilly Radar*. Available at : https://www.oreilly.com/radar/microservices-adoption-in-2020/.

Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X. & Gao, J. (2025). Large Language Models : A Survey. *arXiv preprint arXiv :2402.06196*.

Newman, S. (2015). Building Microservices. *O'Reilly Media*, 278.

Newman, S. (2019). *Monolith to Microservices : Evolutionary Patterns to Transform Your Monolith* (éd. 1). Sebastopol, CA : O'Reilly Media.

Nitin, V., Asthana, S., Ray, B. & Krishna, R. (2023). CARGO : AI-Guided Dependency Analysis for Migrating Monolithic Applications to Microservices Architecture. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*, 20.

Oumoussa, I. & Saidi, R. (2024). Evolution of Microservices Identification in Monolith Decomposition : A Systematic Review. *IEEE Access*, 12, 23389–23405.

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C. & Seinturier, L. (2015). Spoon : A Library for Implementing Analyses and Transformations of Java Source Code. *Software : Practice and Experience*, 46, 1155–1179.

Purdue University, School of Electrical and Computer Engineering. (2017). ECE 468 - Fall 2017. *Purdue University Course Material*. Available at : https://engineering.purdue.edu/~milind/ece468/2017fall/assignments/step4/ (accessed 2025-09-04).

Rademacher, F., Sachweh, S. & Zündorf, A. (2020). A Modeling Method for Systematic Architecture Reconstruction of Microservice-Based Software Systems. *Enterprise, Business-Process and Information Systems Modeling*, 311–326.

Razzaq, A. & Ghayyur, S. A. K. (2023). A Systematic Mapping Study : The New Age of Software Architecture from Monolithic to Microservice Architecture—Awareness and Challenges. *Computer Applications in Engineering Education*, 31(2), 421–451.

Saucedo, A. M., Rodríguez, G., Rocha, F. G. & dos Santos, R. P. (2025). Migration of Monolithic Systems to Microservices : A Systematic Mapping Study. *Information and Software Technology*, 177, 107590.

Sellami, K. & Saied, M. A. (2024). Extracting Microservices from Monolithic Systems Using Deep Reinforcement Learning. *Empirical Software Engineering*, 30(1), 1.

Sellami, K. & Saied, M. A. (2025). Contrastive Learning-Enhanced Large Language Models for Monolith-to-Microservice Decomposition. *arXiv preprint arXiv :2502.04604*.

Sellami, K., Ouni, A., Saied, M. A., Bouktif, S. & Mkaouer, M. W. (2022a). Improving Microservices Extraction Using Evolutionary Search. *Information and Software Technology*, 151, 106996.

Sellami, K., Saied, M. A. & Ouni, A. (2022b). A Hierarchical DBSCAN Method for Extracting Microservices from Monolithic Applications. *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE 2022)*, 201–210.

Sellami, K., Saied, M. A., Ouni, A. & Abdalkareem, R. (2022c). Combining Static and Dynamic Analysis to Decompose Monolithic Application into Microservices. *Service-Oriented Computing*, 203–218.

Sooksatra, K., Chy, M. S. H., Arju, M. A. R., Cerny, T. & Rivas, P. (2024). Using Static Analysis to Aid Monolith to Microservice System Transformation : Tuning Fuzzy c-Means in a VAE-Based GNN Approach. *2024 39th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, 43–53.

Stephanie Susnjara, I. S. (2020). The Advantages and Disadvantages of Microservices. *IBM Think*. Available at : https://www.ibm.com/think/insights/microservices-advantages-disadvantages.

Taibi, D., Lenarduzzi, V. & Pahl, C. (2017). Processes, Motivations, and Issues for Migrating to Microservices Architectures : An Empirical Investigation. *IEEE Cloud Computing*, 4(5), 22–32. doi : 10.1109/MCC.2017.4250931.

Trabelsi, I., Abdellatif, M., Abubaker, A., Moha, N., Mosser, S., Ebrahimi-Kahou, S. & Guéhéneuc, Y.-G. (2023). From Legacy to Microservices : A Type-Based Approach for Microservices Identification Using Machine Learning and Semantic Analysis. *Journal of Software : Evolution and Process*, 35(10), e2503.

Trabelsi, I., Moha, N., Guéhéneuc, Y.-G. & Geffard, L. (2024a). Magnet : Method-Based Approach Using Graph Neural Network for Microservices Identification. *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, 1–11.

Trabelsi, I., Popa, B., Pereyrol, J., Beaulieu, P.-O. & Moha, N. (2024b). MicroMatic : Fully Automated Microservices Identification Approach from Monolithic Systems. *Proceedings of the ACM/IEEE 6th International Workshop on Software Engineering Research and Practices for the Internet of Things (SERP4IoT '24)*, 7–13.

Vámossy, Z. & Romhányi, Á. (2021). Benefits of Layered Software Architecture in Machine Learning Applications. *Proceedings of the International Conference on Image Processing and Vision Engineering (IMPROVE 2021)*, 66–72.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Łukasz Kaiser & Polosukhin, I. (2017). Attention Is All You Need. *Advances in Neural Information Processing Systems (NeurIPS)*, 30.

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J. & Schmidt, D. C. (2023). A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. *arXiv preprint arXiv :2302.11382*.

Wohlin, C. (2014). Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE '14)*, 38.

Yedida, R., Krishna, R., Kalia, A., Menzies, T., Xiao, J. & Vukovic, M. (2023). An Expert System for Redesigning Software for Cloud Applications. *Expert Systems with Applications*, 219, 119673.

Yokoyama, H. (2019). Machine Learning System Architectural Pattern for Improving Operational Stability. *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 267-274.

Zaragoza, P., Seriai, A.-D., Seriai, A., Shatnawi, A. & Derras, M. (2022). Leveraging the Layered Architecture for Microservice Recovery. *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, 135–145.

Zhang, L., Jia, T., Jia, M., Wu, Y., Liu, A., Yang, Y., Wu, Z., Hu, X., Yu, P. S. & Li, Y. (2025). A Survey of AIOps in the Era of Large Language Models. *arXiv preprint arXiv :2507.12472*.

Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J.-Y. & Wen, J.-R. (2025). A Survey of Large Language Models. *arXiv preprint arXiv :2303.18223*.

Zhong, C., Li, S., Zhang, H., Huang, H., Yang, L. & Cai, Y. (2025). Refactoring Microservices to Microservices in Support of Evolutionary Design. *IEEE Transactions on Software Engineering*, 51(2), 484–502.