Titre: Title:	DevOps4ML : analyse et reconstruction semi-automatique des artéfacts et pipelines d'apprentissage machine
Auteur: Author:	Aquilas Tchanjou Njomou
Date:	2021
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Tchanjou Njomou, A. (2021). DevOps4ML : analyse et reconstruction semi-automatique des artéfacts et pipelines d'apprentissage machine [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/9103/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/9103/
Directeurs de recherche: Advisors:	Marios-Eleftherios Fokaefs
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

DevOps4ML: analyse et reconstruction semi-automatique des artéfacts et pipelines d'apprentissage machine

AQUILAS TCHANJOU NJOMOU

Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maitrise ès sciences appliquées*

Génie informatique

Août 2021

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

DevOps4ML: analyse et reconstruction semi-automatique des artéfacts et pipelines d'apprentissage machine

présenté par **Aquilas TCHANJOU NJOMOU**

en vue de l'obtention du diplôme de *Maitrise ès sciences appliquées*

a été dûment accepté par le jury d'examen constitué de :

Foutse KHOMH, président
Marios-Eleftherios FOKAEFS, membre et directeur de recherche
Giuliano ANTONIOL, membre

DÉDICACE

A mes parents, qui n'ont cessé de m'encourager à suivre mes rêves, et qui ont cru en moi, continuellement.

REMERCIEMENTS

Je tiens à exprimer ma plus profonde gratitude à mon directeur de recherche, le Dr Marios Fokaefs, pour ses précieux conseils, ses encouragements et sa patience tout au long de mon parcours de maîtrise. Ensuite, je tiens également à remercier le Dr Bram Adams qui a activement participé à plusieurs aspects de ce travail. Ses conseils, recommandations et corrections ont été un atout précieux dans l'accomplissement de ce travail. Je tiens aussi à remercier Alexandra et Dimitri, dont l'assistance m'a permis de mener à bien les expérimentations complexes et fastidieuses.

Sur un plan plus personnel, je tiens à remercier ma famille, pour leur soutien émotionnel et psychologique. Aussi, je tiens particulièrement à remercier Nicolas, pour son écoute, ses encouragements et son soutien inestimable pendant cette maitrise.

Enfin, je remercie les membres de mon comité, le Dr Foutse Khomh et le Dr Antoniol Giuliano pour l'évaluation de mon mémoire de maîtrise.

RÉSUMÉ

Les promesses de l'Apprentissage Machine (AM) et les avancées les plus récentes en termes d'apprentissage profond et de réseaux de neurones entraînent un intérêt grandissant de la part des organisations et des développeurs vers ces technologies. Cependant, les applications d'AM sont fondamentalement différentes des logiciels traditionnels du fait de leur processus de développement impliquant des activités et des composants plus complexes à gérer et à maintenir. Étant donné le manque de services ciblés pour les logiciels d'AM par les systèmes de contrôle de version traditionnels (VCS), de nombreuses plates-formes pour la configuration et la gestion du cycle de vie de l'apprentissage automatique et la configuration ont vu le jour. Ces outils tirent parti des pratiques d'ingénierie logicielle et du DevOps pour améliorer la façon dont les développeurs créent, exploitent et maintiennent des applications d'AM. Néanmoins, l'aspect récent de ces outils et l'insuffisance de ressources scientifiques sur le sujet posent des limitations à leur adoption, et une majorité de projets d'AM sont encore construits en utilisant les plateformes traditionnelles de développement logiciel. Au vu de ces éléments, cette étude vise à analyser les pratiques, défis et opportunités relatives aux processus et outils de développement de l'AM d'une part, et d'autre part à proposer des solutions favorisant l'automatisation de ces processus.

Au travers du minage des dépôts logiciels, nous avons (1) analysé les pratiques de développement d'AM et de gestion des artéfacts relatifs à partir des dépôts logiciels, (2) identifié les défis relatifs à l'adoption/migration des plateformes MLOps et avons proposé des solutions et recommandations afin de réduire l'impact de ces défis et optimiser les activités afférentes, (3) proposé des méthodologies systématiques de reconstruction automatique des pipelines et des liens entre les artéfacts d'AM.

ABSTRACT

The promises of Machine Learning (ML) and its recent advancements in deep learning and neural networks cause a growing interest from organizations and developers. However, ML applications are fundamentally different from traditional software because of their development process involving a more complex management of related activities and components. Given the lack of targeted services for ML applications by traditional version control systems (VCS), specialised platforms for the configuration and management of the machine learning lifecycle have been proposed. These tools leverage software engineering and DevOps practices to improve the way developers create, operate, and maintain ML applications. However, they are still in an early stage of adoption, and the lack of scientific resources on the subject pose many limitations to their adoption. Thus, a majority of ML projects are still built using traditional software development platforms and processes. This study aims to analyze and explain the practices, challenges and opportunities related to ML development processes and tools. We also propose targeted solutions to prepare ML projects for automation.

Through Mining of Software Repositories (MSR) techniques, we (1) analyze ML development practices from software repositories, (2) identify challenges related to the adoption/migration of MLOps platforms; propose solutions to reduce the impact of these challenges and optimize related activities, (3) propose systematic methodologies for automatic reconstruction of pipelines and links between ML artefacts.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

AI	Artificial Intelligence
AIMMX	Automated aI Model Metadata eXtractor
AM	Apprentissage Machine
API	Application Programming Interface
AST	Abstract Syntax Tree
CD	Continuous Development
CI	Continuous Integration
CML	Continuous Machine Learning
CPU	Central Processing Unit
CRISP	CRoss Industry Standard Process for Data Mining
DevOps	Development Operations
DVC	Data Version Control
FDD	Feature Driven Development
GPU	Graphics Processing Unit
IA	Intelligence Artificielle
IaC	Infrastructure as Code
IO	Input Output
JSON	JavaScript Object Notation
KDD	Knowledge Discovery in Database
ML	Machine Learning
MLLC	Machine Learning Life Cycle
MLOps	Machine Learning Operations
MSR	Mining Software Repositories
MSR4ML	Mining Software Repositories for Machine Learning
NLP	Natural Language Processing
QA	Quality Assurance
RAM	Random Access Memory
SCM	Software Configuration management
SDLC	Software Development Life Cycle
SGBD	Système de Gestion des Bases de Données

SOLID	Single-responsibility, Open–closed, Liskov substitution, Interface segregation, Dependency inversion
SQA	Software Quality Assurance
TDSP	Team Data Science Process
VCS	Version Control System

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Pour des raisons de simplification et de généralisation, les termes Intelligence Artificielle (IA) et Apprentissage Machine (AM) sont utilisés de façon interchangeable dans ce chapitre. Les précisions relatives à chaque terme seront abordées plus en détail dans la partie 2.4.

1.1 Avant-propos

Les systèmes intelligents constituent une classe particulière de systèmes logiciels dans le sens où la logique métier et sa mise en œuvre sont des artéfacts[1] logiciels. En effet, contrairement aux applications logicielles traditionnelles dont la logique est déterminée par programmation codifiée des règles et instructions, les comportements des algorithmes d'Intelligence artificielle (IA) et modèles d'Apprentissage Machine (AM) sont conditionnés par des éléments externes à l'algorithme d'apprentissage : les données d'entraînement initiales d'une part, et d'autre part le système cible auquel ils seront implémentés en tant que composants logiciels [1].

Dans ce contexte, un défi important est que les deux aspects du système, à savoir « logiciel » et « AM », ont des cycles de vie différents [2]. Pour une application d'AM, les données doivent être collectées et prétraitées, puis divisées en ensembles d'entraînement et de test. Le modèle doit être conçu, entraîné, testé et sa qualité évaluée. Entre autres, sa mise en œuvre logicielle doit être conçue, codée, configurée, déployée et testée, ainsi que le reste des composants logiciels du système cible où il sera intégré.

Pour les systèmes logiciels, il est courant de suivre le cycle de vie et les versions de chaque artéfact dans des référentiels spéciaux, appelés systèmes de contrôle de version (VCS pour Version Control System en anglais). Malheureusement, ces systèmes sont conçus de façon générique afin de prendre en charge les logiciels indépendamment de leur domaine. Dans leur état actuel et dans le cas particulier des systèmes intelligents, il devient difficile de faire la distinction entre les artéfacts

[1] Un artéfact est un sous-produit généré lors du développement d'un logiciel. Il s'agit de tout élément impliqué dans le processus de gestion du cycle de développement logiciel. Cela peut inclure les données, diagrammes, spécifications, scripts de configuration, etc.

logiciels et les artéfacts d'AM, incluant les données, les modèles pré-entraînés, les fichiers de configuration et autres.

L'incapacité de séparer ces deux aspects (aspect logiciel et côté AM) peut entraîner un certain nombre de défis pour les équipes de développement. Plus particulièrement, il serait difficile d'expliquer si les changements dans la qualité ou le comportement du modèle sont dus à des modifications du logiciel, des données ou des artéfacts d'AM dans le référentiel. Cela pourrait entraîner une surcharge de ressources et activités supplémentaires pour que les experts appropriés enquêtent sur ces problèmes. Bien que des outils spécialisés aient été développés pour prendre en charge une partie des activités de développement des systèmes d'AM, y compris la gestion de version des artéfacts et le déploiement automatique de modèles en production, ces outils ne résolvent pas le problème fondamental de la distinction entre les artéfacts logiciels et les artéfacts d'AM. De ce fait, ils nécessitent encore un effort considérable de la part des développeurs afin de mettre en place et préparer le projet d'AM à partir de son référentiel logiciel.

1.2 Motivation et problématique

L'Intelligence Artificielle (IA) a connu un bond fulgurant au cours des dernières années [3]. Les développements récents en termes de traitement du langage naturel, d'apprentissage profond et de robotique démontrent les possibilités innombrables offertes par l'IA, qui devient ainsi un important moteur de changement dans de nombreux secteurs [4]. En effet, l'adoption de l'IA par les organisations a augmenté de 16% entre 2016 et 2019 [5], et ce chiffre devrait doubler au cours des cinq prochaines années [6]. Cependant, cette adoption est encore limitée par l'absence de ressources et technologies pouvant aider les organisations lors de l'implémentation de solutions d'IA. L'étude réalisée en 2019 par Alsheibani et al. portant sur les barrières à l'adoption de l'IA par les entreprises [5] a ainsi montré que le principal frein au développement de l'IA dans les organisations est l'absence de ressources (connaissances et outils) nécessaires pour évaluer, construire et opérer les applications d'IA. De plus, ce constat a été observé tant chez les entreprises n'ayant pas encore adopté l'IA, que celles ayant plusieurs projets d'IA déjà implémentés.

Ainsi, les problématiques méthodologique (meilleures pratiques) et technologiques (outils de développements d'applications d'IA) se posent. D'une part, les pratiques, connaissances et outils traditionnels d'ingénierie logicielle, généralement appliquées aux systèmes d'IA, ne suffisent pas

à aborder et gérer pleinement le cycle de développement des applications d'IA, à cause des différences[2] notables entre les artéfacts logiciels et d'IA [7]. Les outils traditionnels de développement logiciels (tels que les VCS) n'offrent pas les fonctionnalités nécessaires à la gestion du cycle de vie des applications d'IA : pas de gestion de version pour les grandes quantités de données; support quasi-inexistant de la traçabilité des modifications entre les artéfacts d'entrée (données brutes, hyperparamètres, code source) et les artéfacts de sortie (données filtrées, modèles, métriques, etc.) [8]. D'autre part, bien qu'il existe des plateformes spécialisées pour la gestion du cycle de vie des applications d'IA, elles sont encore à un stade précoce d'adoption, et les contraintes de configuration, d'intégration et de maintenance des projets d'IA dans ces plateformes sont non négligeables [9].

Dès lors, ces deux aspects révèlent l'importance d'étudier et adapter les pratiques et outils utilisés lors du développement des applications d'IA.

1.3 Objectifs de recherche

Au vu des limitations des outils traditionnels de développement logiciel et des contraintes d'adoption des plateformes de gestion du cycle de vie des applications d'AM, nous souhaitons effectuer une réflexion sur les pratiques et outils de développement des systèmes d'AM, dans le but d'améliorer la productivité des développeurs et des experts impliqués dans le développement de systèmes intelligents. Notre objectif général est d'établir de façon méthodique et structurée la distinction entre les aspects logiciels et AM. Cette distinction permettra ainsi d'aider à la maintenance et l'évolution des applications d'AM grâce à (1) une meilleure identification des causes ayant entraîné des changements particuliers au modèle final et (2) l'automatisation d'une partie des opérations de configuration et d'exécution des flux opérationnels d'AM.

Plus précisément, notre projet a quatre objectifs :

1. Identifier, analyser et expliquer les pratiques actuelles de développement des applications d'AM du point de vue de l'ingénierie logicielle (gestion de versions, stockage des données

[2] Ces différences seront développées davantage dans la partie 2.4.2.

d’entraînement et de configuration, organisation des artéfacts, etc.) et proposer des recommandations;

2. Étudier les défis relatifs à l’adoption des plateformes de gestion du cycle de vie des application d’AM et proposer des solutions et recommandations;
3. Établir des liens de traçabilité entre les artéfacts dans les projets logiciels d’AM, en les étiquetant de manière appropriée dans les référentiels logiciels pour améliorer l'explicabilité du changement, automatiser les flux de travail et améliorer la communication. Nos liens de traçabilité peuvent aider les développeurs à comprendre si un changement dans une sortie observable d'un artéfact d’AM (par exemple, sa précision) est dû à un changement de configuration, de code ou de données. De cette façon, le changement peut être mieux compris, plus facilement inversé s'il y a un besoin, et son impact peut devenir plus visible;
4. Proposer des mécanismes semi-automatiques de reconstruction des pipelines d’AM à partir du code source, en vue d’optimiser la configuration et l’intégration des dépôts d’AM vers les plateformes de gestion du cycle de vie des applications d’AM.

1.4 Hypothèse et assomptions de recherche

Notre hypothèse se base principalement sur les pratiques de minage des dépôts logiciels (en anglais Mining Software Repositories, MSR). Nous nous basons sur le fait que les dépôts logiciels contiennent des informations importantes relatives aux pratiques de développement, et peuvent aider à dresser un portrait plus ou moins exact d’un projet logiciel donné à chaque étape de son évolution. En nous basant sur la littérature existante [2, 10], nous supposons que les applications d’Apprentissage Machine sont traditionnellement développées en utilisant les pratiques d’ingénierie logicielle. La suite du présent document se base sur ces assomptions, et, sauf mention contraire, nous soutenons l’hypothèse selon laquelle les pratiques de développement identifiées par minage de dépôts logiciels publics peuvent être généralisées et refléter des tendances populaires auprès des développeurs.

1.5 Méthodologie

Afin de garantir une analyse plus détaillée, le sujet principal a été séparé en trois sous-sujets :

Sous-sujet 1: Identification des méthodes de développement des applications d'AM : il s'agit d'étudier les dépôts logiciels publics afin d'identifier et expliquer les pratiques utilisées par les développeurs lors de la construction de systèmes d'AM;

Sous-sujet 2: Analyse des défis relatifs à l'adoption/migration vers les plateformes DevOps de gestion du cycle de vie des applications d'AM (plateformes MLOps) ;

Sous-sujet 3: Reconstruction de la traçabilité des artéfacts et pipelines d'apprentissage machine : à partir des pratiques de développement et des défis identifiés précédemment, nous proposerons des techniques systématiques afin de reconstruire la traçabilité des artéfacts et préparer les dépôts d'AM pour l'automatisation.

Bien que la méthodologie générale consiste en l'analyse de dépôts logiciels existants, chaque sous-sujet utilise au moins l'une des méthodologies d'études parmi les suivantes : minage des dépôts logiciels, analyse statistique, étude expérimentale et prototypage, tel que décrit dans le Tableau 1.1.

1.5.1 Collecte des données

Le corpus de données utilisé est constitué de dépôts publics d'Apprentissage Machine recueillis grâce au projet '*Papers With Code'* [11]. Ce projet compile différents articles traitant de l'AM et leurs dépôts *GitHub*[3] respectifs [12]. Ce corpus a été téléchargé le 12 mai 2019 et comprenait alors 4975 projets d'AM. Un premier filtrage a été réalisé afin d'éliminer les doublons et sélectionner seulement les projets ayant *Python* comme langage de programmation principal. *Github* fournit une interface permettant de recueillir des statistiques sur un dépôt particulier. Cette interface a été utilisée afin d'identifier quel est le langage prioritaire dans chaque dépôt d'AM, et ainsi ne sélectionner que ceux construits en *Python.* Ce langage a été choisi du fait de sa prévalence pour le développement d'applications d'AM. En effet, il s'agit du langage de programmation le plus utilisé et le plus priorisé pour le développement d'applications d'AM [13]. Un second filtrage a été effectué afin de supprimer les projets inaccessibles publiquement. Il s'agit de projets dont les tentatives d'accès renvoyaient des erreurs de type non disponibles parce qu'ils ont été supprimés

[3] Github est une plateforme de contrôle de version basée sur Git permettant aux développeurs de collaborer, stocker et gérer leurs projets de développement logiciel.

ou rendus privés par leur propriétaire. Nous avons alors obtenu un corpus global de 2681 projets qui a servi de base pour tout le travail. Selon l'objectif et la capacité d'analyse requise pour chaque sujet, des filtrages supplémentaires ont été réalisés, et la taille du corpus final par sous-sujet est présentée dans le Tableau 1.1. Les détails des activités de filtrage pour chaque sous-sujet seront présentés dans leurs parties respectives, afin de conserver une certaine cohérence et faciliter la compréhension.

Tableau 1.1 Récapitulatif des méthodes d'étude utilisées

Sous-sujet	**Méthodes utilisées**	**Taille du corpus de données**
1- Identification des méthodes de développement des applications d'AM	• Minage de dépôts logiciels • Analyse statistique	532
2- Challenges d'adoption/migration des plateformes MLOps	• Étude expérimentale • Analyse statistique	13
3- Reconstruction de la traçabilité des artéfacts et pipelines d'apprentissage machine	• Prototypage	Groupe A : 9 Groupe B : 13

Lors de la réalisation du présent travail, la disponibilité des applications d'AM commerciales et open-source possédant les informations nécessaires (c'est-à-dire le code source, les fichiers, les commits, etc.) était assez limitée. En effet, avant d'utiliser le corpus « Papers with code », nous avons initialement cherché sur GitHub des projets commerciaux pertinents, mais nous nous sommes rapidement rendu compte qu'ils étaient difficiles à trouver, et qu'ils n'étaient pas correctement documentés (par exemple, des projets qui avaient le terme 'apprentissage automatique' dans leur description (fichier README), mais qui après analyse détaillée n'implémentait pas un système d'AM). Nous discuterons plus tard des limites de ce choix et de la manière dont ce travail peut être étendu en conséquence.

1.5.2 Minage des dépôts logiciels

Le Minage des dépôts logiciels (En anglais Mining Software Repositories et en abrégé MSR) est une pratique populaire utilisée dans de nombreux projets de recherche [14-17] et consiste à fouiller, extraire et analyser les données historiques des dépôts logiciels dans le but d'identifier de

l'information pertinente relativement à des pratiques de développement particulières. Il s'agit d'extraire, comparer, vérifier et étudier les artéfacts logiciels (communications, historique de version, équipes, bogues, etc.) suivant différents axes d'analyse [18]. Cette technique a été largement utilisée dans le présent travail, et a permis d'établir une base solide pour les solutions, recommandations et outils proposés. Pour plus de clarté, les pratiques spécifiques de minage adoptées pour chaque sous-sujet (analyse des fichiers sources, identification des artéfacts logiciels, inspection manuelle du code, etc.) seront abordées dans leurs parties respectives

1.5.3 Analyse statistique

L'analyse statistique est une composante de l'analyse des données et consiste en la collecte et l'interprétation de données dans le but d'identifier des modèles et des tendances sous-jacentes [19]. Par exemple, des statistiques relatives au nombre de fichiers d'un certain type dans les dépôts logiciels peuvent indiquer la prévalence d'un langage de programmation spécifique par rapport à un autre. Dans cette étude, les données ont été premièrement échantillonnées et filtrées (retrait des projets non pertinents) afin de réduire tout biais de données. L'analyse statistique a permis d'établir les prévalences de différentes pratiques de développement logiciels adoptées par les développeurs et de proposer des pistes d'amélioration.

1.5.4 Étude expérimentale

L'étude expérimentale consiste à réaliser une expérience spécifique et méthodologique autour d'un sujet précis, afin d'observer et analyser les résultats qui en découlent [20]. Dans le cas particulier des systèmes logiciels, deux types de pratiques d'expérimentation ont été proposées par Juristo et al.: l'expérimentation par observation et l'expérimentation par interaction [21]. Dans le premier cas, les chercheurs préparent le cadre et les sujets/objets d'expérimentation, mais les faits sont perçus de l'extérieur par les chercheurs sans aucune interférence de leur part sur les sujets ou objets observés, sauf celle provoquée par l'observation elle-même. Dans le deuxième cas, les chercheurs ne sont pas de simples observateurs/récepteurs : ils entrent en contact avec l'élément étudié. Cette interaction consiste à soumettre l'objet à de nouvelles conditions et à observer les réactions. Dans ce cas, les chercheurs interfèrent avec le cadre d'expérimentation et les observations sont le résultat d'une telle interférence. Les chercheurs ont ainsi le contrôle de la réalité pendant l'expérimentation. Dans le cadre de cette étude, l'expérimentation par interaction a été choisie principalement au cours

du sous-sujet 2. En effet, nous avons sélectionné un ensemble de projets et avons réalisé les processus de migration vers les plateformes de gestion du cycle de vie des applications d'AM, afin de simuler les activités traditionnelles qui s'y rapportent. La procédure suivie est détaillée dans la partie 4.4.2.

1.5.5 Expérimentation par prototypage

Le prototypage est un processus expérimental consistant à mettre en œuvre des idées sous des formes tangibles [22]. Marion et al. Définissent l'expérimentation par prototypage comme tout type de représentation, sur tout support, conçu pour comprendre, explorer ou communiquer ce que cela pourrait être de s'engager avec le produit, l'espace ou le système conçu [23]. Il s'agit plus précisément d'implémenter une preuve de concept relativement à une idée, méthode ou technologie proposée sous forme abstraite. Au cours de cette étude, les résultats et propositions découlant des sous-sujets 1 et 2 ont été utilisés afin d'introduire un prototype pour un système semi-automatique de reconstruction de la traçabilité des artéfacts et pipelines d'AM.

1.6 Plan du mémoire

La suite du mémoire est organisée de la façon suivante :

- Le Chapitre 2 présente une revue de littérature sur les principaux éléments abordés dans le cadre de ce travail à savoir : (1) les méthodologies de développement logiciels et d'AM. Il s'agit principalement des taxonomies, processus et normes de gestion du cycle de vie des logiciels, ainsi qu'aux processus DevOps; (2) les applications d'Apprentissage Machine ainsi que leurs processus de développement; (3) l'état de l'art sur les plateformes DevOps pour la gestion du cycle de vie des applications d'AM;
- Dans le Chapitre 3 , nous présentons les détails de l'étude et des résultats du sous-sujet 1 relativement aux pratiques de développement d'applications d'AM identifiés par minage de dépôts logiciels;
- Le Chapitre 4 quant à lui regroupe les défis et recommandations identifiées lors de l'analyse expérimentale réalisée pour le sous sujet 2 (défis d'adoption/migration vers les plateformes MLOps);

- Le Chapitre 5 présente les différents outils et mécanismes proposés pour l'automatisation d'une partie des activités nécessaires afin de rendre les dépôts d'AM prêt pour l'automatisation ;
- Enfin, le Chapitre 6 présente les principales conclusions du travail ainsi que les recommandations futures.

Une partie de ce travail (la première moitié du chapitre 5) a été publiée à la conférence SANER 2021 : *Aquilas Tchanjou Njomou, Alexandra Johanne Bifona Africa, Bram Adams, Marios Fokaefs: MSR4ML: Reconstructing Artifact Traceability in Machine Learning Repositories. SANER 2021: 536-540*

CHAPITRE 2 GÉNÉRALITÉS

2.1 Introduction

Le développement d'applications logicielles obéit à un ensemble de méthodes et d'activités plus ou moins codifiées. Ces activités déterminent et conditionnent tant les performances du logiciel [24] que son exploitation [25] et sa maintenance [26]. Cependant, les recherches actuelles en matière d'Apprentissage Machine (AM) tendent à démontrer que certaines éléments propres à ces systèmes les différencie des logiciels traditionnels [2].

Ce chapitre a pour objectif principal de présenter les généralités relatives à l'analyse et au développement d'applications logicielles et d'AM. Plus spécifiquement, nous souhaitons introduire les thèmes abordés dans ce chapitre à savoir :

- Les outils et processus de développent logiciels traditionnels ainsi que leurs limitations
- La philosophie DevOps, son origine et ses principes directeurs
- Les systèmes d'Apprentissage Machine et leurs particularités en termes de conception et d'implémentation
- Les notions récentes de MLOps et l'état de l'art sur les pratiques dérivées
- Le minage de dépôts logiciels et son utilisation au sein de la communauté de recherche.

2.2 Cycle de développement des applications logicielles

Les applications logicielles passent généralement par un ensemble d'étapes au cours de leur cycle de vie. Ces étapes et les activités qui s'y rapportent sont regroupées autour d'un terme générique : Le cycle de vie du développement logiciel, ou Software Development Life Cycle (SDLC) [27] . Le SDLC est utilisé pour la planification, la conception, la création, le test et le déploiement d'applications, et a pour but de garantir la qualité des systèmes crées et leur adéquation avec les attentes du client, tout en respectant les ressources allouées [28]. Il comporte différentes phases cycliques présentées dans la Figure 2.1 [29].

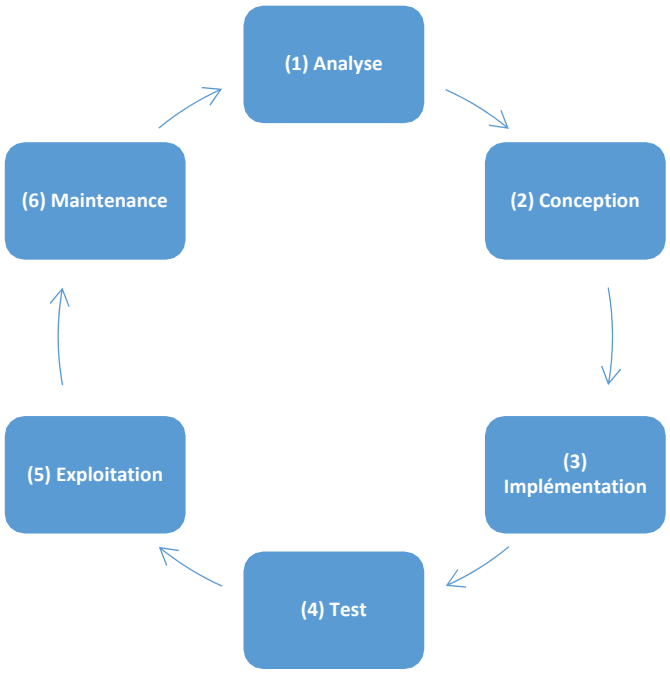


Figure 2.1 Processus général du SDLC

1. **L'analyse des besoins (**ou encore spécifications des besoins) : Il s'agit ici de mettre le logiciel dans son contexte en déterminant son but, ses fonctionnalités générales et son état (s'il s'agit d'un nouveau logiciel ou de l'amélioration d'un logiciel existant).
2. **La conception**, qui consiste au choix des solutions proposées en utilisant certaines spécifications. Elle peut comprendre la description des fonctionnalités de l'application qui sont mentionnées de façon générale lors de l'étape de l'analyse des besoins.
3. **L'implémentation** qui consiste au développement du logiciel proprement dit. Dans cette étape, les éléments du logiciel (code, données, configuration, etc.) sont mis en place progressivement de façon à former un tout.
4. **Les tests** qui permettent de vérifier et étudier la qualité du logiciel et sa conformité par rapport aux spécifications
5. **L'exploitation** qui concerne le déploiement et l'utilisation régulière du logiciel dans un environnement de production.
6. **La maintenance** qui consiste à modifier l'application après le déploiement du produit et lors de son exploitation. Son but est de corriger les erreurs et anomalies détectées lors de l'exploitation du logiciel ou simplement d'y ajouter de nouvelles fonctionnalités.

2.2.1 Méthodes de développement logiciel

En pratique, le SDLC peut être appliqué de différentes manières, et il existe plusieurs méthodes proposant un cadre théorique et pratique décrivant l'exécution de ces activités de façon plus ou moins standardisée [30].

2.2.1.1 Méthodes séquentielles

Il s'agit d'un ensemble de modèles linéaires qui se subdivisent le processus de développement en plusieurs phases. Le principe de ces modèles consiste à terminer chaque phase par la production de certains livrables (documents ou programmes) à une date précise. Les résultats sont obtenus sur la base d'itérations entrecoupées, et sont ensuite soumis à une revue. Le passage à la phase suivante est conditionné par la validation de ces livrables, tel que le montre la Figure 2.2.

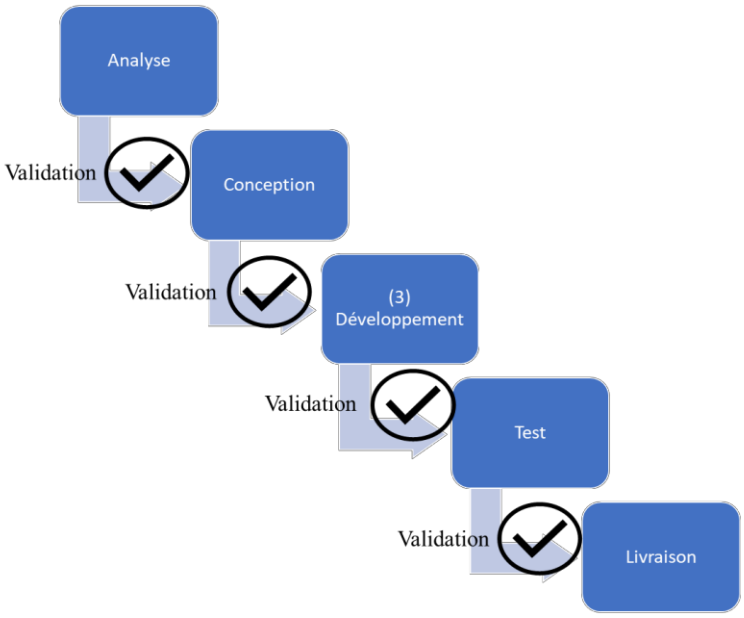


Figure 2.2 Processus de développement séquentiel

Ces modèles incluent notamment le modèle en cascade [31] ou en V [32]. Bien qu'ils présentent un certain nombre d'avantage comme un contrôle plus strict sur les phases et produits de développement, ces modèles ont été vivement critiqués à cause de plusieurs inconvénients et des impacts qu'ils ont sur le cycle de vie du logiciel à savoir :

- **Des délais accrus** : Étant donné que chaque séquence ne peut être entamée que si la séquence précédente est terminée, tout retard au niveau d'une séquence se répercutera sur toutes les séquences suivantes, entraînant ainsi des délais non négligeables [33];

- **Collaboration limitée avec les clients** : Dans les modèles itératifs, le client n'intervient généralement que lors de la phase de définition des besoins. L'interaction avec le client est ainsi limitée au cours des phases subséquentes, ce qui réduit la collaboration [34].

- **Complexité et rigidité des processus** : Étant donné la faible intervention du client au cours du développement, la flexibilité du logiciel s'en trouve réduit dans la mesure où (1) les changements provenant du client ne sont détectés que très tardivement dans le processus, et (2) la coupure entre chaque étape rend difficile le retour en arrière [35].

2.2.1.2 Méthodes agiles

Les méthodes agiles regroupent des méthodes itératives basées sur l'adaptabilité et la flexibilité des processus. Ces méthodes obéissent à un ensemble de principes déterminées par le manifeste Agile [36] dont les 4 principales valeurs sont :

- Les individus et leurs interactions plus que les processus et les outils;
- Des logiciels opérationnels plus qu'une documentation exhaustive;
- La collaboration avec les clients plus que la négociation contractuelle;
- L'adaptation au changement plus que le suivi d'un plan;

Guidées par ces valeurs, la plupart des méthodes agile (SCRUM [37], Kanban [38], FDD [39] et bien d'autres) subdivisent les processus en des étapes itératives et plus petites, et prônent une meilleure communication et collaboration entre les différentes équipes impliquées dans le développement du logiciel. La Figure 2.3 présente la vision générale de la méthodologie Agile, qui est centrée sur la satisfaction client.

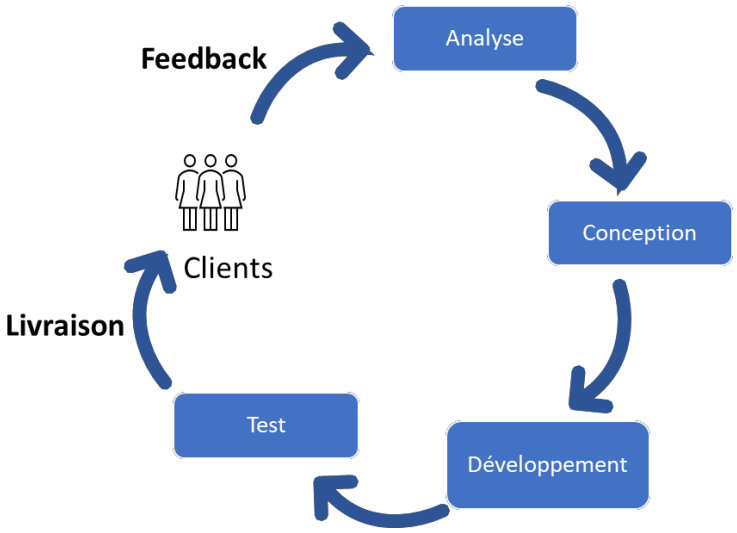


Figure 2.3 Processus général des méthodes Agiles

Les méthodes Agile sont parmi les plus populaires actuellement utilisées pour le développement de logiciels [40]. Cependant, la première valeur Agile met l'accent sur les individus et leurs interactions. Ceci impose généralement une réorganisation des équipes logicielles lors de l'adoption des pratiques Agiles et l'adoption d'outils spécialisés pour faciliter la collaboration [41], tels que les systèmes de contrôle de version.

2.2.2 Systèmes de contrôle de version : Exemple de Git

La pluralité des activités de développement logiciel entraîne une multiplication des équipes et de leurs compétences. De ce fait, de nombreux outils ont vu le jour afin de faciliter le développement collaboratif ainsi que la communication entre les différentes équipes. Parmi ces outils, l'on note l'utilisation accrue des systèmes de contrôle de version (VCS pour Version Control System en anglais) qui ont pour rôle de suivre et gérer les changements apportés aux artéfacts logiciels [42]. En effet, lorsque différentes équipes travaillent ensemble sur un même logiciel, il est difficile de gérer les modifications apportées par chaque développeur. L'utilisation d'un VCS vient dès lors combler ce vide en garantissant un cadre commun et organisé permettant aux équipes de suivre l'évolution du logiciel en tout temps.

Il existe plusieurs systèmes VCS, dont le plus connu est *Git* [43]. Créé en 2005 par Linus Torvald, *Git* a rapidement évolué pour devenir le système de contrôle de version open-source le plus utilisé

parmi les développeurs [44]. Afin de gérer l'évolution et les modifications survenant dans le code source, il utilise un ensemble d'éléments et d'activités. Nous nous concentrerons sur la description des éléments abordés dans le présent travail, soit les notions de dépôt logiciel, de branches et de commits.

- Dépôt logiciel : Il s'agit de l'élément fondamental représentant le code source du logiciel et ses artéfacts (documentation, configuration, et bien d'autres). Le dépôt logiciel est représenté sous forme de dossier où sont stockés toutes les données relatives à l'historique d'évolution du logiciel
- Branche : Il s'agit de la représentation modifiable d'une version du code. Pour un même dépôt logiciel, il est possible d'avoir plusieurs branches parallèles permettant de travailler simultanément sur plusieurs aspects du logiciel sans créer de conflits entre les modifications. Généralement, il existe une branche principale correspondant la version de référence du logiciel, et les modifications sur les autres branches sont alors fusionnées à la branche principale après avoir été vérifiés.
- Commit : Il s'agit d'une entité basique de *Git* permettant d'enregistrer les modifications apportées à un instant donné. Ainsi, chaque modification à un artéfact est enregistrée et sauvegardée afin de permettre un suivi ou un retour en arrière.

L'un des outils populaires de gestion de code basée sur *Git* est *GitHub*. Il s'agit d'une plateforme Web de gestion des dépôts logiciels proposant une interface graphique et une collaboration accrue grâce au stockage en ligne des dépôts et des outils de communications adaptés.

2.3 L'approche DevOps

Cette partie ne présente pas une définition exhaustive de l'approche DevOps, mais se concentre plutôt sur la présentation des éléments essentiels pour la compréhension du présent travail.

DevOps [45] est un acronyme signifiant Development-Operations (Développement-Opérations). Il s'agit de réduire l'écart et les incompréhensions entre l'équipe de développement et l'équipe responsable des opérations de gestion d'environnement et de déploiement logiciel. En effet, le développement logiciel nécessite des compétences particulières à chaque étape de son évolution. Dans les organisations, ces compétences sont généralement regroupées autour de deux équipes, les

développeurs (Dev) et les opérationnels (Ops). Les Dev sont généralement responsables de planifier, construire et compiler l'application, tandis que les Ops sont responsable de la création et la gestion des environnements, de l'exécution du logiciel et du monitoring. Les tests peuvent être effectués par l'une ou l'autre des équipes, ou les deux.

Cependant, ces activités sont souvent interdépendantes, et des problèmes de collaboration peuvent apparaître [46] :

- Grand écart entre Dev et Ops entraînant des problèmes de communication;
- Non standardisation entre les outils utilisés par les différentes équipes;
- Tâches fastidieuses et répétitives de configuration des environnement, d'exécution de tests, etc.

La philosophie DevOps a ainsi vu le jour en 2008 afin de réduire ces limitations et optimiser les activités interdépendantes entre les différentes équipes. DevOps se base principalement sur plusieurs principes parmi lesquels la collaboration, la communication, l'automatisation et les itérations continues. Les deux derniers principes consistent à automatiser le plus possible les processus répétitifs, et d'assurer les activités de façon continue et sans disruption. On parle alors d'intégration continue, déploiement continu, monitoring continu, etc.

Pour assurer cette continuité, de nombreux outils d'automatisation ont vu le jour. Ces outils tirent parti des technologies les plus récents en matière d'architecture logicielle (containeurs[4], micro-services, infonuagique, etc.) afin d'automatiser la configuration des environnements, les tests unitaires, la compilation et le déploiement. Ces activités sont configurées sous forme de flux (ou pipeline), souvent appelé pipeline CI/CD (Continuous Integration/Continuous Deployment) dont un exemple est présenté dans la Figure 2.4.

Il existe différents outils d'automatisation (*GitHub Actions*, *Travis CI*, *Jenkins*, etc.) dont le principe de fonctionnement est généralement le même : ils requièrent des scripts de configuration

[4] Un containeur est un système regroupant un ensemble de fonctions matérielles et logicielles autour d'un environnement virtualisé

(définis manuellement) où sont spécifiés les données d'environnement ainsi que les instructions à exécuter à chaque étape du pipeline.

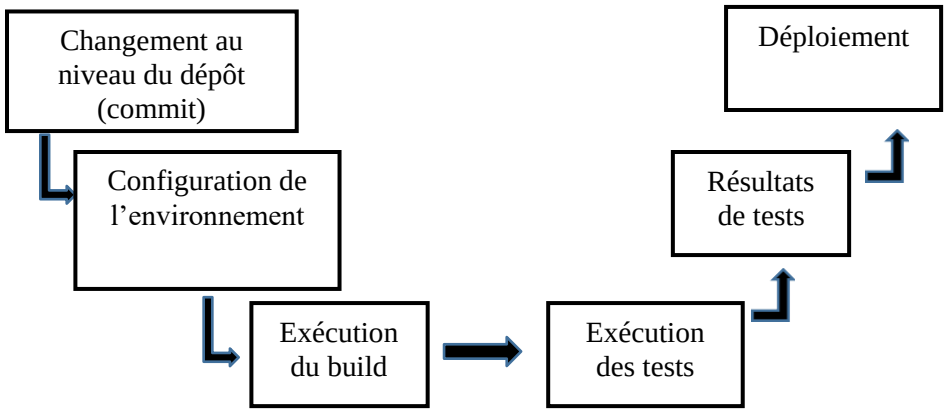


Figure 2.4 Exemple d'un pipeline CI/CD

2.4 Développement des applications d'Apprentissage Machine : Principes et artéfacts

L'Intelligence Artificielle (IA) regroupe l'ensemble des théories des techniques permettant de mettre au point des programmes informatiques complexes capables de simuler certains traits de l'intelligence humaine (raisonnement, apprentissage…). Son objectif est de proposer et construire des systèmes intelligents pouvant réaliser des tâches complexes.

L'Apprentissage Machine (AM) est une branche de l'Intelligence Artificielle qui permet aux systèmes d'apprendre à effectuer une action (classification prédiction, recommandation, etc.) à partir de données historiques.

2.4.1 Construire une application d'Apprentissage Machine : principes et cycle de vie

L'Apprentissage Machine fait référence à l'étude de programmes informatiques (ou algorithmes) qui peuvent apprendre par l'exemple et généraliser à partir d'exemples existants d'une tâche. Mitchell et al. [47] proposent un formalisme court et largement utilisé sur ce sujet qui énonce ce qui suit : Un programme informatique est dit apprendre d'une expérience *E* en relation avec une classe de tâches *T* et une mesure de performance *P*, si ses performances dans l'exécution des tâches de *T*, telles que mesurées par *P*, s'améliorent avec l'expérience *E*. Le domaine de l'AM vise ainsi à

apprendre aux machines à effectuer des tâches en fournissant quelques exemples [48]. Sur la base de ces exemples, des algorithmes spécialisés sont capables de capturer des modèles afin de faire des prédictions sur des données auparavant inconnues et non utilisées lors de la phase d'apprentissage. Le terme prédiction fait donc référence à la sortie d'un algorithme après qu'il a été formé sur un ensemble de données historiques et appliqué à de nouvelles données en essayant de prédire la probabilité d'un résultat particulier.

Malgré son évolution croissants, il n'existe pas de méthodologie prescriptive pour le développement d'applications d'AM, du fait qu'elles sont actuellement implémentées pour des tâches spécifiques [49], ce qui ne permet pas une normalisation du processus de développement. Cependant, une approche globale qui considère les principaux éléments d'AM a tendance à être utilisée. La Figure 2.5 présente le flux général et les artéfacts produits lors du développement des systèmes d'AM qui comprend 4 principales phases, préparation des données, entraînement du modèle, l'évaluation et le déploiement final. Chaque étape produit un ensemble d'artefacts généralement améliorés au cours du cycle de vie du modèle [50].

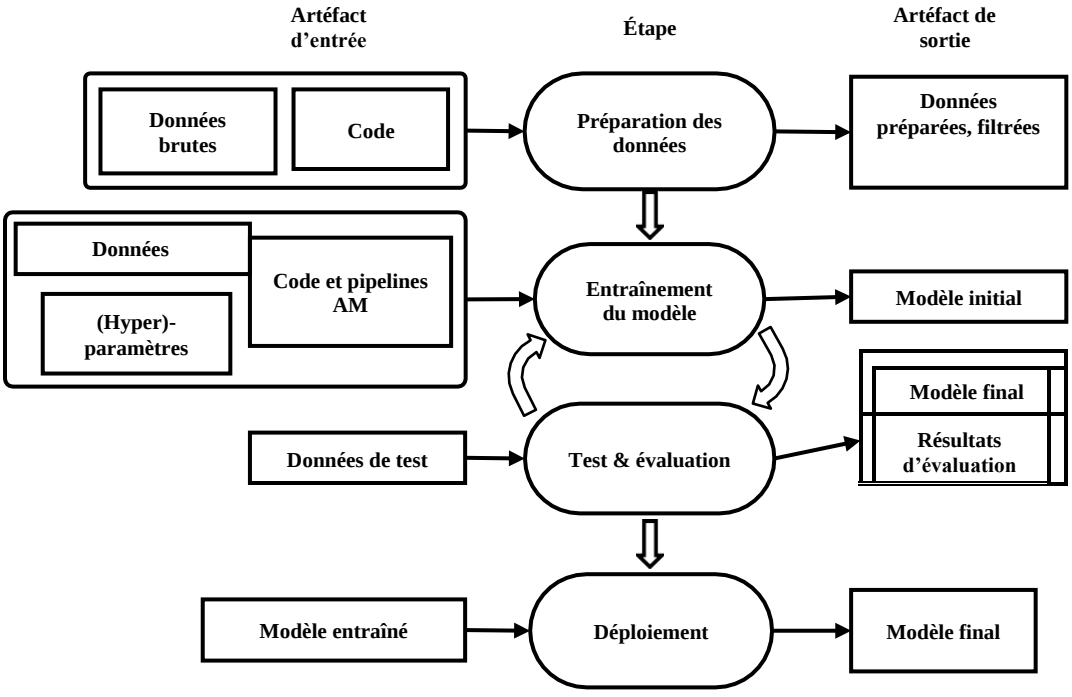


Figure 2.5 Flux général et artéfacts de développement des applications d'AM

Lors de la phase de préparation des données, il est nécessaire de collecter des données qui seront utilisées pour entraîner le modèle. Ces données proviennent généralement de plusieurs sources

(bases de données, fichiers, API, etc.) et peuvent avoir des formats différents. Il est donc nécessaire de les trier, filtrer et/ou agréger afin d'obtenir une source et un format spécifique que le modèle utilisera. L'étape d'entraînement permet à l'algorithme d'AM d'apprendre à partir des données prétraitées. Cela nécessite le choix de la technique la plus appropriée pour répondre au problème initial et le choix des outils et modules de développement. Il existe une grande variété de plates-formes et de librairies pour le développement d'applications d'AM, dont certaines sont plus adaptées à des techniques d'AM spécifiques. Cette phase consiste également à régler les paramètres et les hyperparamètres du modèle pour assurer le meilleur taux d'apprentissage et les meilleures performances. Le résultat est un modèle qui est testé dans la phase d'évaluation du modèle, qui consiste à exécuter le modèle sur des données de test et à enregistrer ses performances à travers ses métriques. Une fois que le modèle passe la phase de test, il peut ensuite être déployé et surveillé dans un environnement de production. Si le modèle ne passe pas la phase d'évaluation, il est généralement recyclé à l'aide de différents ensembles de données, paramètres et/ou hyperparamètres pour améliorer ses métriques. L'évolution des métriques au cours de l'entraînement du modèle permet ainsi d'évaluer les tendances d'entraînement et de prédiction.

2.4.2 Limitation des outils SDLC traditionnels

Les applications d'AM sont un type particulier d'application logiciel, dans la mesure où ceux-ci requièrent un principe de développement spécifique tel que présenté dans la Figure 2.6

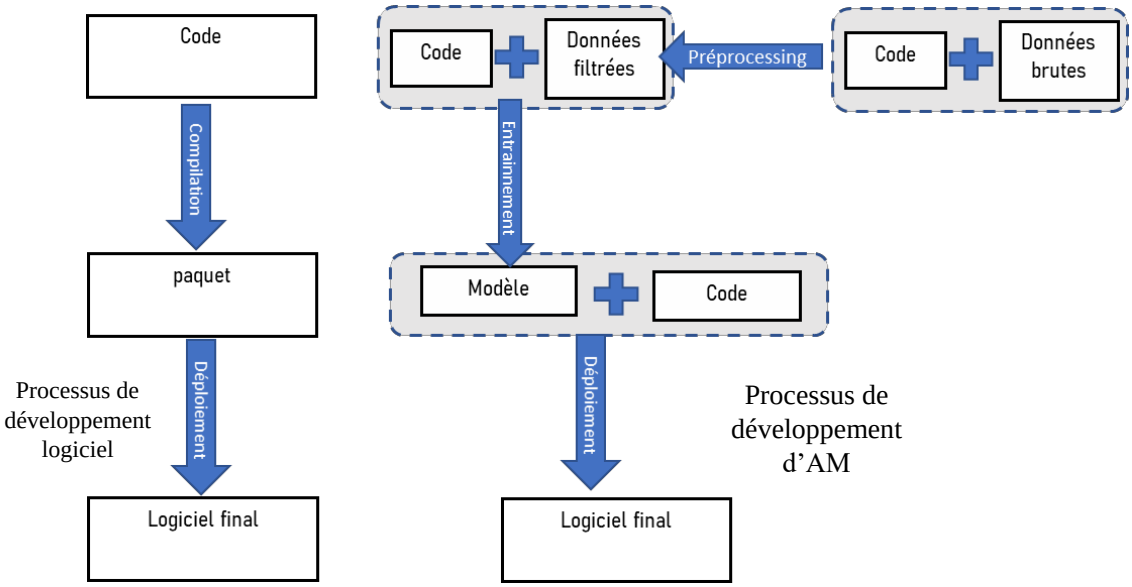


Figure 2.6 Comparaison des processus généraux de développement logiciel et d'AM

Lors du développement d'un système logiciel, du code source est compilé afin de produire une application finale (code + compilation=>application finale) tandis que dans les applications d'AM, le code source utilise des données d'entrée, passe par différentes phases (entraînement, évaluation, validation, etc.) et produit un modèle, qui à son tour sera intégré à un autre code afin de produire l'application finale (code source + données => modèle + code => application finale).

Les artéfacts d'AM, tout comme ceux du logiciel, sont stockés dans des systèmes VCS, qui sont largement utilisés par les développeurs pour gérer les artefacts logiciels [10]. Cependant, ils ont été principalement conçus pour le développement de logiciels traditionnels et manquent de fonctionnalités pertinentes pour les projets d'AM. Par exemple, *Git* ne permet pas de suivre les modifications des données, des hyperparamètres, des modèles finaux et des dépendances entre ces artéfacts [8]. Entre autres, Amershi et al. [2] ont découvert que les artéfacts d'AM sont plus difficiles à manipuler que les artéfacts logiciels traditionnels. Cela ajoute plus de complexité du point de vue de la gestion des modèles, en plus des défis causés par la diversité des équipes et la gestion des processus d'AM. Ainsi, l'objectif du Chapitre 3 est d'analyser et comprendre, au regard de ces difficultés, quelles pratiques les développeurs d'AM adoptent.

2.5 MLOps : Automatisation des étapes de développement des applications d'AM

MLOps est un terme hybride représentant l'application de la philosophie DevOps (automatisation et intégration) au développement d'AM [51].

La méthodologie DevOps prône la collaboration entre le personnel de développement et d'exploitation pendant toutes les phases de développement au déploiement du logiciel [52] au moyen des approches de SCM [53] (Software Configuration Management) et de SQA [54] (Software Quality Assurance). Cependant, au vu de la particularité des applications d'AM qui combinent les logiciels traditionnels aux algorithmes et modèles d'apprentissage automatique, l'application de cette méthodologie devient plus complexe. En effet, les approches SCM et SQA du logiciel et des modèles d'AM sont séparées [4], ce qui ne permet pas l'exploitation complète du potentiel de ces applications. Pour faciliter les activités de développement d'AM, de nombreux outils ont vu le jour tels que *Kubeflow* [55], *DVC* [56], *MLflow* [57], *ModelDB* [58] et bien d'autres. Ces outils combinent le contrôle des données et des versions, la gestion des pipelines d'AM et les techniques DevOps pour offrir un environnement

transparent, intégré et centralisé afin de planifier, créer, construire, tester et déployer les applications d'AM. On parle alors de plateformes MLOps. Elles abordent les activités de développement d'AM d'un point de vue logiciel et proposent différentes pratiques d'intégration continue pour automatiser certaines activités comme le tunage des hyperparamètres, le réentraînement des modèles, le traçage des artéfacts, etc.

Les outils et les pratiques MLOps étant nouveaux, nous proposons d'étudier, dans le Chapitre 4, les défis relatifs à l'adoption de ces plateformes, et présenter des solutions et recommandations aux développeurs d'AM.

2.6 Conclusion

Contrairement aux logiciels traditionnels, le développement d'applications d'AM nécessite des étapes, des artefacts, des équipes et des outils complexes [2]. Les méthodes traditionnelles de SDLC ainsi que les techniques DevOps et les outils afférents montrent des limitations importantes relatives à la gestion des activités, processus et composants des systèmes d'AM.

Malgré ces limitations relativement aux outils et méthodes, la popularité de l'AM auprès des développeurs et des organisations ne cesse de grandir (même si, selon les études les plus récentes sur le sujet, la majorité de ces projets n'atteignent jamais l'étape d'exploitation [59]). Il est donc important de comprendre comment les développeurs open source s'adaptent face à ces limitations, et quelles pratiques ils adoptent afin de concilier les processus d'AM et les outils traditionnels de développement logiciel.

CHAPITRE 3 IDENTIFICATION DES PRATIQUES DE DÉVELOPPEMENT D'APPLICATIONS D'AM PAR MINAGE DE DÉPÔTS LOGICIELS

3.1 Introduction et objectifs

La popularité croissante de l'AM suscite un intérêt particulier pour les développeurs et les organisations. La multitude de langages de programmation, de bibliothèques et de ressources disponibles leur offre la possibilité de créer leurs propres systèmes d'AM, sans besoin de connaissances poussées dans le domaine. Cependant, la particularité de ces systèmes, telle qu'expliquée dans la partie 2.4.1, est qu'ils impliquent un processus de développement différent de celui des logiciels traditionnels. Dès lors, il est important de comprendre comment les développeurs adaptent ces processus au cas particulier des applications d'AM afin d'une part d'identifier les pistes d'amélioration relativement à ces pratiques, et d'autre part proposer des outils pertinents pouvant aider à réduire la charge et les contraintes de développement de ces systèmes. Plus spécifiquement, les objectifs de ce chapitre ont été regroupés en trois questions de recherche :

- **QR1 - Comment sont utilisées les principales librairies dans les projets d'AM? :** nous souhaitons identifier les principales librairies utilisées par les développeurs d'AM, et décrire les corrélations existantes entre elles. Nous pensons que l'utilisation de librairies spécifiques pourraient donner des indications sur les pratiques de développement adoptés, ainsi que sur les méthodes d'intégration des artéfacts d'AM aux dépôts logiciels.;
- **QR2 - Comment les développeurs intègrent-ils les artéfacts dans les projets d'AM?** : considérant le fait qu'il n'existe aucun standard pour l'organisation des artéfacts logiciels ou d'AM, l'objectif ici est de comprendre les techniques adoptées par les développeurs pour l'organisation et la gestion des artéfacts d'AM dans les dépôts;
- **QR3 - Comment les projets d'AM évoluent-ils? :** *Git* utilise principalement les commits pour gérer l'évolution des dépôts logiciels traditionnels. Cependant, au vu des limitations des outils traditionnels relativement à la gestion des artéfacts d'AM, nous souhaitons comprendre comment les développeurs utilisent ces outils pour la gestion des versions des artéfacts d'AM.

3.2 Approche méthodologique et corpus de données

Tel que mentionné dans la partie 1.5, l'approche utilisée combine le minage des dépôts logiciels à une analyse statistique. Nous avons étudié les dépôts d'AM et avons identifié les principales habitudes de développement, regroupées sous forme de 8 pratiques générales. Étant donné que chaque question de recherche touche à un aspect particulier des dépôts d'AM (code, artéfacts et évolution), les détails de la méthodologie adoptée pour chaque question de recherche sont fournis dans leurs parties respectives.

Concernant le corpus de données, nous avons étudié 532 projets d'AM de *GitHub* utilisant *Python* comme langage de programmation principal. A partir du corpus initial (2681 projets d'AM), nous avons supprimé les projets ne possédant pas un fichier de description des librairies nécessaires (généralement nommé 'requirements.txt') et ayant moins de 5 commits. Ce choix a été fait dans le but de filtrer les projets dont l'historique et la description en termes de requis sont très faibles, afin d'avoir une base pour les **QR 1 et QR 2.** De ce filtrage, nous avons obtenu 532 dépôts.

3.3 QR1 : Comment sont utilisées les principales librairies dans les projets d'AM?

3.3.1 Motivation

Généralement, le développement d'applications logicielles implique l'utilisation d'un ensemble de librairies[5] publiques ou privées proposant des fonctionnalités spécifiques. Les algorithmes d'AM impliquent des traitements mathématiques (statistiques, combinatoires, corrélations, et bien d'autres) avancés nécessitant un certain niveau de connaissances. De nombreuses librairies ont été créées afin de faciliter l'implémentation de ces traitements, et ainsi démocratiser l'accès à l'AM. Dans la littérature, il a été montré que l'identification des pratiques d'utilisation des librairies dans

[5] Une librairie logicielle est un ensemble de scripts, codes et données proposant des fonctions réutilisables. Les librairies sont généralement ajoutées aux projets logiciels principalement afin de ne pas « réinventer la roue » (Avoir à développer de nouvelles fonctions déjà développées précédemment par d'autres.

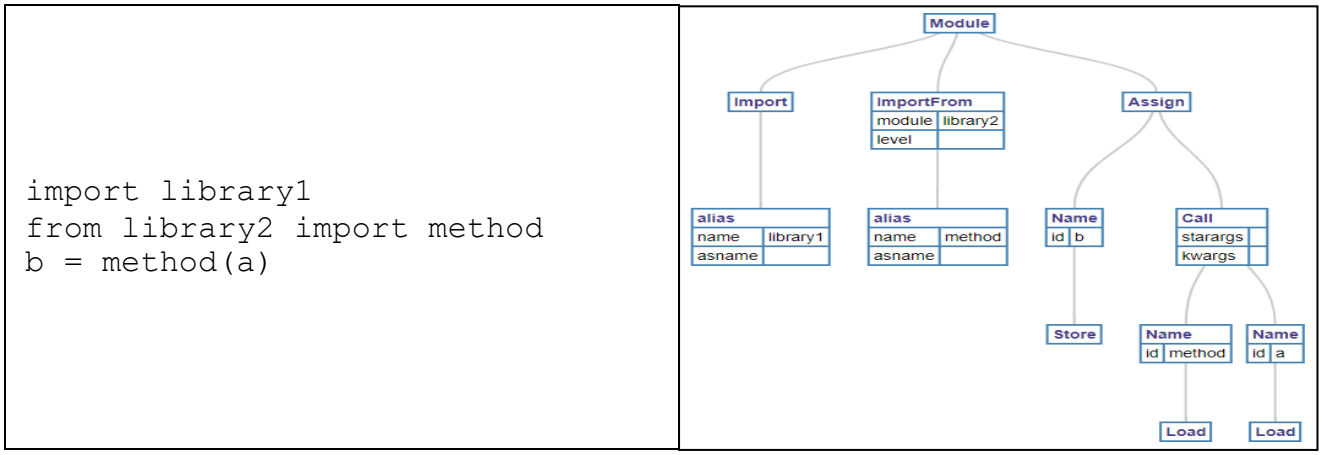
les projets logiciels peut aider les développeurs à optimiser la recherche et le choix de librairies particulières pour leurs projets [60-63].

En analysant la prévalence de ces pratiques dans les dépôts publics d'AM, nous souhaitons ainsi :

1. Identifier les librairies les plus utilisées dans les applications d'AM
2. Extraire les corrélations existantes entre ces différentes librairies
3. Déduire et expliquer la prédominance de pratiques spécifiques en matière d'utilisation de librairies tierces

3.3.2 Approche

L'approche utilisée consiste à effectuer une analyse statistique des librairies importées dans les projets étudiés et de leur utilisation. Avant d'exécuter une portion de code, l'interpréteur[6] *Python* construit un arbre syntaxique abstrait (AST en anglais pour Abstract Syntax Tree) qui est une représentation hiérarchique du code source contenant toutes les informations relatives aux instructions à exécuter. Ces informations sont organisées sous forme de nœuds imbriqués, chaque nœud représentant une instruction spécifique, comme le montre la Figure 3.1. C'est cet arbre que nous avons exploité afin d'identifier les nœuds relatifs aux instructions d'importation de librairies.



Code initial — Arbre syntaxique généré par l'interpréteur Python

Figure 3.1 Exemple de code Python et de son arbre syntaxique abstrait

[6] Un interpréteur est un programme chargé de lire d'exécuter itérativement les instructions qui lui sont fournies.

Dans python, les librairies tierces sont intégrées en utilisant les instructions `‘import [librairie]’` ou `‘from [librairie] import [fonction]’`. Ces instructions génèrent respectivement des nœuds de type ‘*Import’* et *‘ImportFrom’* dans l’arbre syntaxique. Pour ce faire nous avons utilisé la librairie native[7] ‘*Python ast’* [64] afin d’identifier et extraire ces nœuds. Nous avons ainsi pu déterminer la liste des librairies utilisées dans chaque projet et avons compilé leurs proportions d’utilisation. Ensuite, le minage par règle d’association[8] a été utilisé afin de déterminer les relations existantes entre ces librairies [65].

3.3.3 Résultats

> **Pratique 1** : les développeurs open-source ont tendance à utiliser de façon prédominante des librairies spécifiques pour leurs applications d’AM, telle que *Numpy*, *os* et *sys*, utilisées respectivement par 95%, 92% et 70% des projets.

La Figure 3.2 présente la distribution des 20 librairies les plus utilisées. L’on remarque que la librairie tierce[9] ‘*NumPy’* [66] est la plus importée, avec 95% de projets qui l’utilisent. Aussi, les principales librairies les plus importées sont à thématique scientifiques.

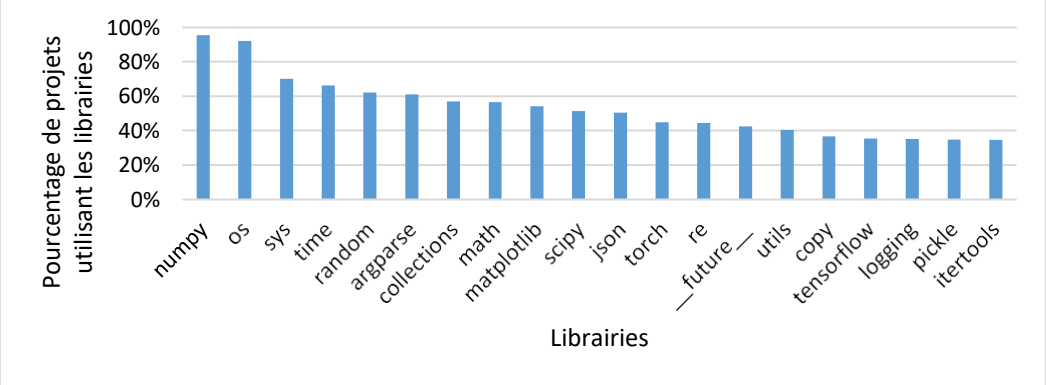


Figure 3.2 Distribution des 20 librairies les plus utilisées à travers les dépôts analysés

---

[7] Les librairies natives sont des librairies généralement fournies par défaut avec le code principal d’un langage de programmation et ont l’avantage de garantir un support complet du fournisseur officiel.

[8] Consiste à regrouper les variables d’une population pour identifier les relations existantes entre elles

[9] Non native, développée et maintenue par une communauté externe au fournisseur officiel de Python

En effet, les librairies *'NumPy', 'collections', 'math', 'scipy', 'torch', 'tensorflow'* et '*itertools'* ont pour principales fonctionnalités la gestion des calculs scientifiques et l'Apprentissage Machine. Ce résultat était attendu, vu la nature des dépôts étudiés (les applications d'AM nécessitent des fonctionnalités de traitement des données par calcul ou par apprentissage automatique).

Entre autres, on observe une prévalence de librairies spécialisées pour la gestion des interactions système. Les librairies natives '*os'* et '*argparse'* arrivent respectivement en deuxième et troisième position. D'après la documentation de *Python* [67], la librairie '*os'* permet de gérer les interactions avec le système d'exploitation, principalement les lectures et écritures sur le disque. La librairie '*argparse'* quant à elle permet de gérer les entrées système en ligne de commande. Bien que nous n'ayons établi aucune hypothèse de départ concernant la prévalence de ces fonctions, nous estimons que ces résultats sont logiques au vu de la nécessité d'interagir avec les données et fichiers externes fournis lors de l'exécution du programme ou stockées dans le système.

> **Pratique 2** : Les développeurs open-source utilisent généralement des groupes de librairies ensemble, avec une forte association entre certains groupes (*numpy, scipy, matplotlib*) d'une part et (*os, argparse, sys*) d'autre part.

Dans le but de déterminer la proportion moyenne de librairies populaires importées par les projets d''AM, nous avons sélectionné les 10 libraires les utilisées, puis avons compilé le nombre de ces librairies importées par chaque projet. La Figure 3.3 présente les résultats obtenus.

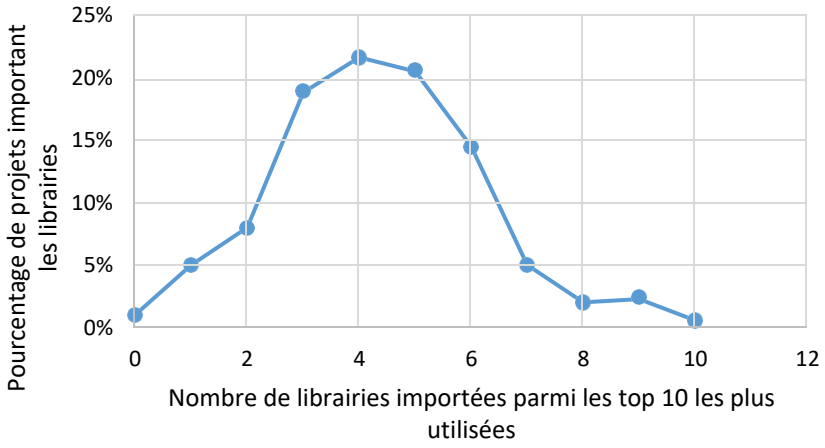


Figure 3.3 Utilisation des librairies parmi les plus populaires

On note que les plus grandes valeurs d'utilisation sont observées lorsqu'il s'agit de 3, 4 et 5 librairies importées. On observe un pic d'utilisation pour la valeur 4 (22%), ce qui signifie que 22% de projets importent 4 librairies parmi les 10 librairies les plus utilisées. A partir de ces résultats, nous supposons que ces librairies sont utilisées ensemble, et qu'il est possible d'estimer la probabilité d'utilisation d'une ou de plusieurs librairies, individuellement ou collectivement, ainsi que leurs relations.

Pour ce faire, nous avons analysé leurs règles d'association, qui peuvent être défini de la façon suivante : Soit $I = \{i_1, i_2, \dots, i_n\}$ l'ensemble de toutes les librairies et $T = \{t_1, t_2, \dots, t_m\}$ l'ensemble des librairies importées dans chaque projet tel que t_i soit un sous ensemble de I ($ie\ t_i \supseteq I$). Une règle d'association est représentée de la forme :

$$X => Y \text{ où } X \in T, Y \in T \text{ et } X \cap Y \neq 0 \text{ [68]}$$

Afin d'évaluer la force d'une règle, ses indices de confiance, support et lift sont déterminés. leurs définitions sont récapitulées dans le Tableau 3.1 (n représente le nombre d'éléments d'un ensemble) [69].

Tableau 3.1 Définition et formules des indices d'appréciation des règles d'association

Indice	**Définition**	**Formule**	**Signification**
Confiance	Proportion de transactions de T contenant X qui contiennent aussi T	$\frac{n(X \cap Y)}{n(X)}$	Indique la fréquence d'apparition de cette règle. Plus il est élevé, et plus la règle est observée fréquemment
Support, noté $Supp(X \cap Y)$	Proportion de transactions de T qui contiennent $X \cap Y$	$\frac{n(X \cap Y)}{n(T)}$	Mesure la fréquence d'apparition des groupes de librairie X et Y
Lift	Amélioration apportée par rapport à un jeu de transactions aléatoires	$\frac{Supp(X \cap Y)}{Supp(X).Supp(Y)}$	Détermine à quel point la règle est significative. Une valeur de $lift > 1$ traduit une corrélation positive entre X et Y

Les principales règle obtenues (top 15 classé par ordre décroissant de confiance) sont présentées dans le Tableau 3.2 sous la forme {Antécédent}=> {Conséquent}, ainsi que leurs indices de

confiance, de support et de lift. Par exemple, une représentation de la forme {*numpy, os*} => {*matplotlib*} signifie qu'il y'a une association entre ces trois librairies, et donc qu'un dépôt utilisant '*numpy'* et '*os'* a une certaine probabilité (estimable par les métriques associées) d'utiliser aussi '*matplotlib'*.

Tableau 3.2 Récapitulatif des principales règles d'association

Numéro	**Règle**	**Confiance**	**Support**	**Lift**
1	*{matplotlib, scipy} =>{numpy}*	0,99	0,23	1,14
2	*{matplotlib, argparse} =>{numpy}*	0,98	0,44	1,13
3	*{os} => {argparse}*	0,98	0,35	1,12
4	*{argparse} => {os}*	0,98	0,24	1,12
5	*{sys, os} => {argparse}*	0,97	0,23	1,12
6	*{matplotlib, sys, scipy} => {numpy, tensorflow}*	0,97	0,24	1,12
7	*{json, argparse} => {os}*	0,97	0,25	1,12
8	*{numpy, tensorflow} => {matplotlib}*	0,97	0,25	1,28
9	*{numpy, torch} => {collections, matplotlib}*	0,97	0,24	1,28
10	*{time, collections} => {numpy, sklearn}*	0,97	0,25	1,11
11	*{pandas} => {numpy}*	0,96	0,26	1,11
12	*{pickle, numpy} => {matplotlib}*	0,96	0,25	1,28
13	*{os, argparse} => {numpy, pickle}*	0,96	0,4	1,11
14	*{pickle} => {json}*	0,96	0,26	1,27
15	*{matplotlib, pandas} => {numpy}*	0,96	0,49	1,1

L'on observe une prédominance de '*NumPy'* en tant que conséquent, ce qui s'explique logiquement pat le fait qu'elle soit utilisée par un nombre important de projets. L'on note aussi une association intéressante entre les librairies de gestion des interactions système (règles 3, 4, 5 et 7) traduisant une utilisation commune de ces librairies. Ce tableau peut être utilisé par les développeurs d'AM pour le choix des librairies. Par exemple, pour un développeur souhaitant utiliser la librairie '*scipy'* et '*matplotlib'*, ce tableau lui permettra de se rendre compte du fait qu'il devra aussi très probablement utiliser la librairie '*numpy'* (règle 1). Cela lui permettra de mieux planifier la gestion des librairies externes en (1) déterminant quelles versions utiliser afin d'éviter les problèmes de compatibilité et (2) choisissant d'utiliser ou non une librairie particulière à cause de sa corrélation trop élevée avec des librairies non voulues (à cause d'un faible support technique disponible pour cette librairie par exemple).

Les résultats sur l'utilisation de librairies supposent une contradiction de l'hypothèse selon laquelle les développeurs utilisent différents modes de stockage des données. En effet, seuls 3% des dépôts analysés utilisent des librairies de connexion aux bases de données (2% utilisent '*sqlalchemy'*, 1% utilisent '*mysql'* ou '*sqlite3'*), ce qui signifierait que les développeurs n'utilisent pas les systèmes de gestion de bases de données (SGBD) pour stocker les données d'entraînement de leurs modèles. Aussi, l'utilisation de librairies relatives aux interactions avec les services les plus populaires de stockage infonuagique ('*boto3'* pour *Amazon*, '*azure'* pour *Microsoft Azure* et '*ibm_boto3'* (ou *'ibm_botocore'*) pour *IBM* est très faible (3% au total), ce qui signifierait que les services infonuagiques ne sont pas utilisés par un grand nombre de développeurs d'AM. Or, étant donné que les applications d'AM nécessitent souvent de très grandes quantités de données [70] dont la taille pose des limitations face aux méthodes de stockage traditionnels (Système de fichier local), cela soulève davantage de questionnements concernant les outils et méthodes utilisées pour stocker, gérer et intégrer ces données dans les projets d'AM.

3.4 QR2 : Comment les développeurs intègrent-ils les artéfacts dans les projets d'AM?

3.4.1 Motivation

La nécessité de comprendre comment les développeurs d'AM intègrent les artéfacts externes dans leurs projets vient de la dépendance quasi-totale de ces systèmes envers leurs données d'entraînement, hyperparamètres, etc. [71]. En effet, au vu de la criticité des données qui déterminent la performance finale d'un modèle d'AM, une gestion inefficace des données (stockage, versioning, intégration, évolution) entraîne des problèmes de drift[10], de biais[11] ou encore une baisse soudaine et inexpliquée des performances du modèle [72]. Entre autres, les résultats de

[10] La notion de drift se réfère à l'inadéquation entre les données d'entrée et de sortie d'un modèle devenu obsolète. Cela se produit lorsque les données réelles de prédiction diffèrent grandement des données initiales d'entraînement du modèle, provoquant des résultats erronés.

[11] Un modèle d'AM est biaisé lorsqu'il a tendance à favoriser (ou défavoriser) certains groupes. Le phénomène de biais survient généralement lorsque les données d'entraînement sont inégalement réparties.

la **QR 1** sèment le doute quant aux pratiques de stockage des données qu'il serait important d'éclaircir.

3.4.2 Approche

Tel que mentionné dans la partie 2.4.1, les artéfacts utilisés dans les applications d'AM peuvent provenir de différentes sources, et avoir plusieurs formats, que nous souhaitons identifier. Pour cela, nous avons premièrement analysé les extensions des fichiers présents dans les dépôts analysés, que nous avons séparé en deux principaux groupes : les fichiers de code et les fichiers sans code. Les fichiers de code sont les fichiers utilisés par les langages de programmation pour stocker les instructions de code. Ils utilisent généralement des extensions particulières, afin d'être reconnus par l'interpréteur. Pour Python, ces extensions sont '*.py'*. Les fichiers sans code quant à eux regroupent tous les autres fichiers du dépôt, à l'exception des fichiers et dossiers cachés (dont le nom commence par '.'). En effet, ceux-ci représentent des fichiers de configuration utilisés par des plateformes spécifiques, et donc ne sont pas directement utilisés par le dépôt en lui-même. Davantage de filtrage a été effectué parmi les fichiers sans code :

- Tous les fichiers possédant le terme '*readme'* sans distinction de la typographie des caractères (majuscules ou minuscules) ainsi que ceux finissant par l'extension '*.md'* ont été ignorés. Ces fichiers sont généralement utilisés par convention pour fournir de la documentation relative au dépôt [73].
- Tous les fichiers possédant le terme '*requirements'*, qu'importe l'extension, n'ont pas été comptabilisés, car ils servent par convention à décrire les librairies requises pour exécuter le projet [74].

Une analyse initiale des différents types de fichiers stockés dans les dépôts d'AM présentée dans la Figure 3.4 montre une grande utilisation des fichiers de type '*.ipynb'* reliés à Jupyter Notebook[12]. Ce résultat est logique et confirme une utilisation de plus en plus grandissante de la programmation

[12] Jupyter Notebook est une solution de programmation lettrée permettant de réunir dans un même fichier compilable du code et du texte dans le but de faciliter le codage collaboratif.

lettrée[13] (ou programmation littéraire) en général et de Jupyter Notebook en particulier pour les applications d'AM [7]. Ces fichiers ont donc été ignorés de l'analyse.

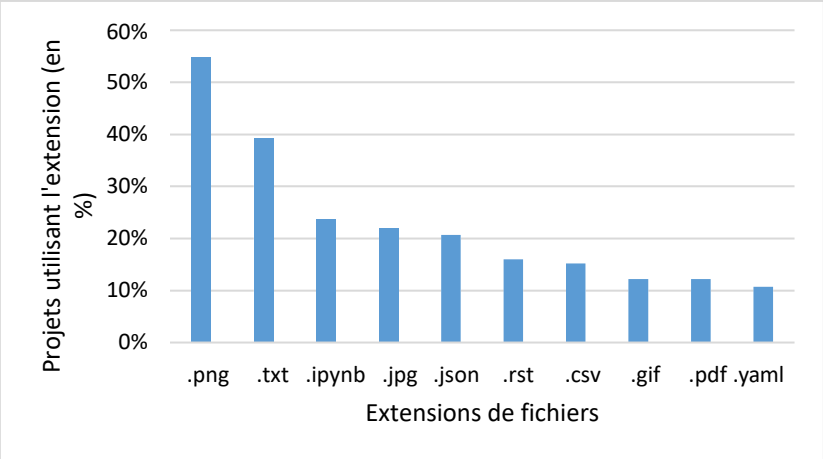


Figure 3.4 Répartition de l'utilisation des extensions de fichiers dans les projets analysés (top 10 des extensions les plus utilisées)

A partir des fichiers sans code identifiés, nous avons développé une méthode heuristique[14] dans le but d'identifier les artéfacts d'AM. Pour chaque heuristique, un script d'inspection a été réalisé afin d'automatiser l'identification. Pour l'heuristique h_1, le script utilise une expression régulière[15] appliquée à chaque fichier et retourne le nom du fichier lorsqu'une correspondance est trouvée. Le nom du fichier représente sa localisation relativement au dossier racine du projet, et inclut ainsi les noms des sous-dossiers dans lequel il se trouve. Nous partons de l'hypothèse selon laquelle les développeurs utiliseraient une convention pour organiser et nommer les artéfacts selon leur catégorie. Ainsi, le nom des artéfacts retournés devrait contenir l'un ou l'autre des termes suivants: '*data'* pour les artéfacts de données, '*model'* pour les modèles, '*(hyper)parameter'* pour les paramètres ou hyperparamètres, et '*metric'* pour les artéfacts stockant les métriques des modèles.

---

[13] La programmation lettrée consiste à combiner le code et l'explication de sa logique au sein d'un même fichier exécutable

[14] Une analyse par heuristique consiste à construire un ensemble de paramètres afin de déduire approximativement un raisonnement.

[15] Une expression régulière en abrégé regex est une suite de caractères plus ou moins codifiés permettant la comparaison et l'identification de patterns dans une autre suite de caractères.

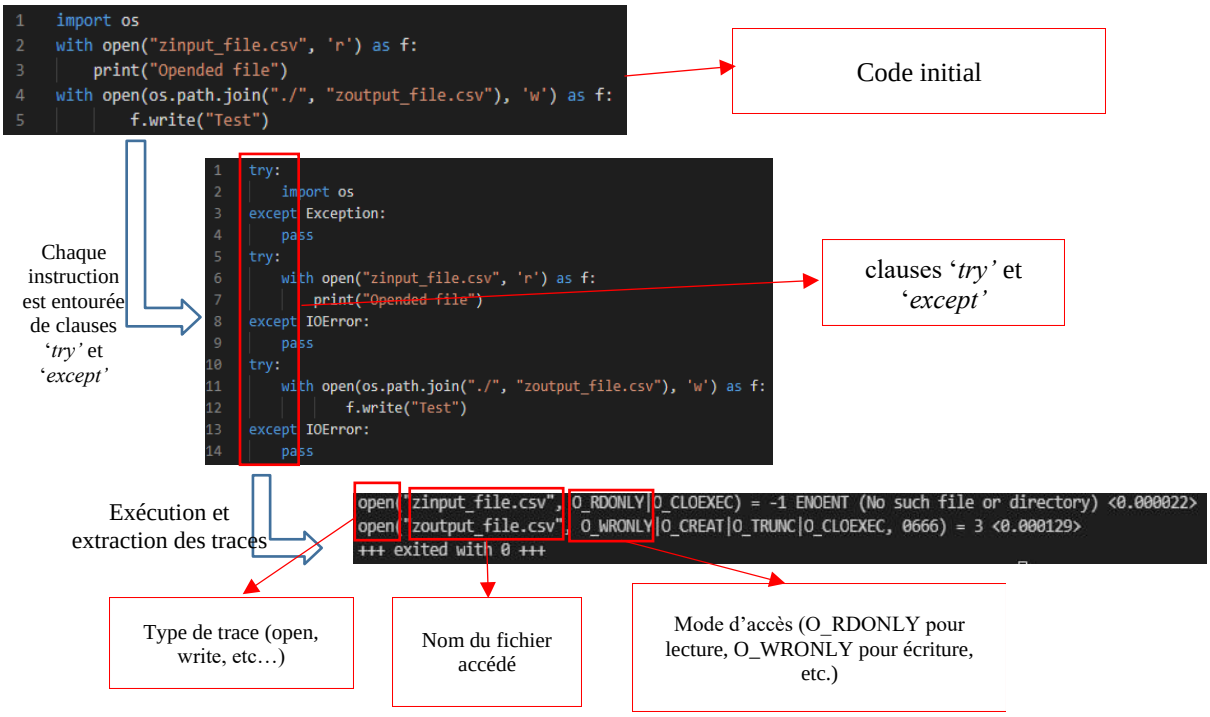


Figure 3.5 Exemple d'exécution du script de l'heuristique h2

Le script de l'heuristique h_2 (dont un exemple d'exécution est présenté à la Figure 3.5) a trois fonctions séquentielles:

1. Pour chaque projet, il crée un environnement *Python3* et installe les dépendances nécessaires à partir du fichier '*requirements.txt*';
2. Pour chaque fichier de code, il entoure les instructions de clauses '*try*' et '*except*[16] afin que le code puisse s'exécuter même si certaines instructions renvoient des erreurs. Chaque fichier de code est alors exécuté, puis le script analyse les traces d'interaction du programme avec le système. Ces traces sont des logs renvoyés par le compilateur python décrivant de façon ordonnée les opérations d'entrée-sortie du programme;

[16] Ces clauses permettent de tester un groupe d'instructions et dans le cas où une erreur survient, la capturer et la traiter de façon spécifique. Sans ces clauses, toute erreur provoquée par une instruction provoquerait l'arrêt complet du programme, tandis que ces clauses permettent de gérer l'erreur, tout en permettant au programme d'exécuter les instructions suivantes.

3. A partir des traces, il extrait les noms de fichier accédés ainsi que le mode d'accès (lecture ou écriture).

Une première analyse a montré l'apparition d'erreurs empêchant l'exécution complète du code, dû notamment au fait que certains fichiers de code nécessitaient des configurations supplémentaires (besoin d'arguments ou variables obligatoires par exemple). Nous avons donc rajouté une troisième heuristique pour plus de fiabilité.

Pour l'heuristique h_3, nous avons premièrement établi manuellement (grâce à une consultation de la documentation) une liste de fonctions d'interaction avec les fichiers systèmes. Ces fonctions sont celles proposées par défaut par Python, ainsi que celles des 10 librairies les plus utilisées identifiées dans la partie 3.3. Un script automatique parcourt alors chaque fichier de code, extrait les arguments de ces fonctions et à partir de là, retourne le nom du fichier accédé ainsi que le mode d'accès. Par exemple, considérons l'instruction '*open(file='filename1', mode='w')*'. Les arguments de cette fonction sont '*file*' et '*mode*' qui représentent respectivement le nom fichier et le mode d'ouverture. La valeur du mode permet de déterminer si le fichier est ouvert soit en lecture ('*r*' pour read) soit en écriture ('*w*' pour write, '*a*' pour append, etc.). La Figure 3.6 présente un exemple de code ainsi que l'organisation des fonctions d'entrée-sortie et leurs arguments.

Figure 3.6 Exemple de fonctions d'entrée-sortie

Les heuristiques ainsi utilisées et les résultats obtenus sont récapitulés dans le Tableau 3.3.

Tableau 3.3 Heuristiques utilisées et résultats préliminaires

Heuristique	Condition	Méthode	Résultat
h_1	Le nom du fichier contient le terme '*data'* ou '*model*' ou '*param'* ou '*metric'*	Trier et extraire tous les noms de fichiers respectant l'heuristique	51288 artéfacts dans 240 dépôts
h_2	Le fichier est chargé lors de l'exécution du code	Exécuter le code et analyser les traces en termes d'entrée-sortie (fichiers avec lesquels le programme interagit)	774 artéfacts dans 141 dépôts
h_3	Le nom du fichier est mentionné dans une fonction d'interaction système	Établir une liste des fichiers d'interaction système, puis examiner leurs arguments pour identifier les fichiers chargés	701 artéfacts dans 181 dépôts

3.4.3 Résultats

> **Pratique 3** : Les développeurs open-source ont tendance à utiliser des fichiers pour sauvegarder les artéfacts d'AM, très peu utilisent des bases de données ou des services cloud.

Les résultats de l'analyse des librairies utilisées ainsi que celle des fichiers présents dans les dépôts montrent une très faible utilisation d'une part des librairies de gestion des bases de données et des fichiers de configuration de base données d'autre part (moins de 3% de projets utilisent des fichiers de configuration de base de données).

- Dans le but de vérifier la concordance entre les heuristiques, nous avons examiné leurs intersections, dont les résultats sont présentés dans le Tableau 3.4, et les autres, qui respectent pourtant la convention de nommage définie, ne sont ouverts par aucun programme;
- Ces fichiers pourraient ne pas être des artéfacts d'AM, mais plutôt des artéfacts logiciels (documentation, configuration, etc.)

Au vu du faible degré de certitude de l'heuristique h_1, nous n'avons considéré que les résultats des heuristiques h_2 et h_3 pour la suite de l'analyse. Ainsi, le nombre total d'artéfacts identifiés se calcule par :

$$N = n(h_2 \cup h_3) = n(h_2) + n(h_3) - n(h_2 \cap h_3)$$

L'on obtient ainsi 834 artéfacts identifiés dans 192 dépôts.Nous supposons que l'intersection entre les heuristiques devrait être élevée, ce qui signifierait que toutes les heuristiques ont pu identifier les mêmes artéfacts.

Tableau 3.4 Intersections entre les heuristiques

Intersection	Résultat
$h_1 \cap h_2$	37 artéfacts dans 21 dépôts
$h_1 \cap h_3$	65 artéfacts dans 28 dépôts
$h_2 \cap h_3$	641 artéfacts dans 129 dépôts
$h_1 \cap h_2 \cap h_3$	12 artéfacts dans 8 dépôts

Les résultats démontrent une grande intersection entre les heuristiques h_2 et h_3, mais cette intersection est très faible lorsqu'il s'agit de l'heuristique h_1, ce qui contredit notre hypothèse. Cela pourrait signifier que:

- Des fichiers identifiés par h_1 pourraient être des anciennes ou différentes versions d'artéfacts qui ne sont plus importants pour la version actuelle du dépôt, mais qui sont quand même conservés pour des raisons de traçabilité: seulement 3 % des artéfacts identifiés par h_1 sont accédés par un programme, et les autres, qui respectent pourtant la convention de nommage définie, ne sont ouverts par aucun programme;
- Ces fichiers pourraient ne pas être des artéfacts d'AM, mais plutôt des artéfacts logiciels (documentation, configuration, etc.)

Au vu du faible degré de certitude de l'heuristique h_1, nous n'avons considéré que les résultats des heuristiques h_2 et h_3 pour la suite de l'analyse. Ainsi, le nombre total d'artéfacts identifiés se calcule par :

$$N = n(h_2 \cup h_3) = n(h_2) + n(h_3) - n(h_2 \cap h_3)$$

L'on obtient ainsi 834 artéfacts identifiés dans 192 dépôts.

> **Pratique 4** : Les développeurs open-source ont tendance à stocker les artéfacts d'entrée/sortie dans un format compréhensible par l'homme, et il existe une prévalence des formats sérialisés (donc non compréhensibles par l'Homme) pour les artéfacts de sortie

En analysant les extensions des artéfacts identifiés suivant leur mode d'interaction, nous obtenons les résultats compilés dans la Figure 3.7. L'utilisation est regroupée par projets. Par exemple, 76% de projets utilisent l'extension .txt pour les artéfacts d'entrée, contre 57% pour les artéfacts de sortie.

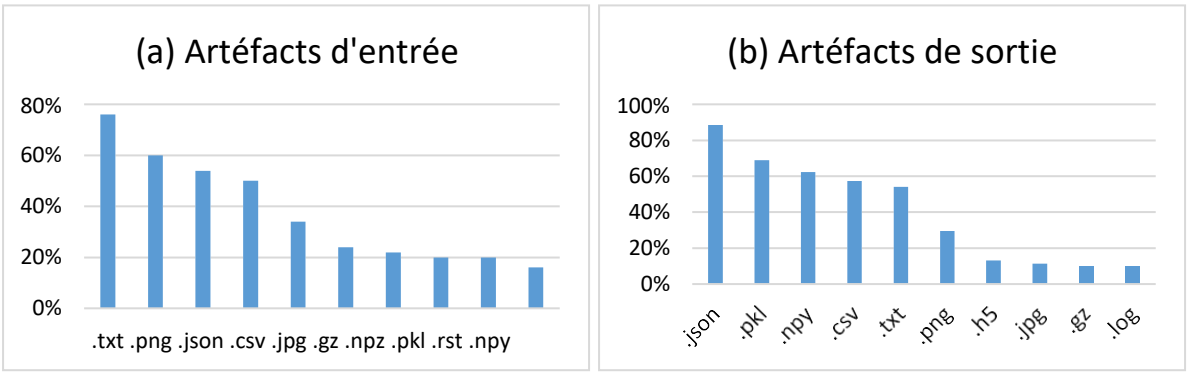


Figure 3.7 Répartition des extensions utilisés par les projets selon le mode d'interaction

Ces résultats démontrent des préférences particulières en termes de méthode de stockage des artéfacts. En effet, Les 10 extensions les plus utilisées peuvent être regroupés en 4 catégories : fichiers textes (*'.txt', '.json', '.csv, 'rst', '.h5'*) qui contiennent du texte brut sans formatage spécial, fichiers image (*'.png', '.jpg'*) contenant des données graphiques, fichiers d'archives (*'.gz'*) pouvant stocker des données compressées (texte, image, voix, vidéo, etc.) et sérialisés (*'npz', 'pkl', 'npy'*),

qui contiennent généralement des données binaires et sont générés par les librairies '*Numpy*' (*'.npz'* et *'.npy'*) et '*Pickle*'[17] (*'.pkl'*). L'utilisation de chaque catégorie est présentée dans la Figure 3.8.

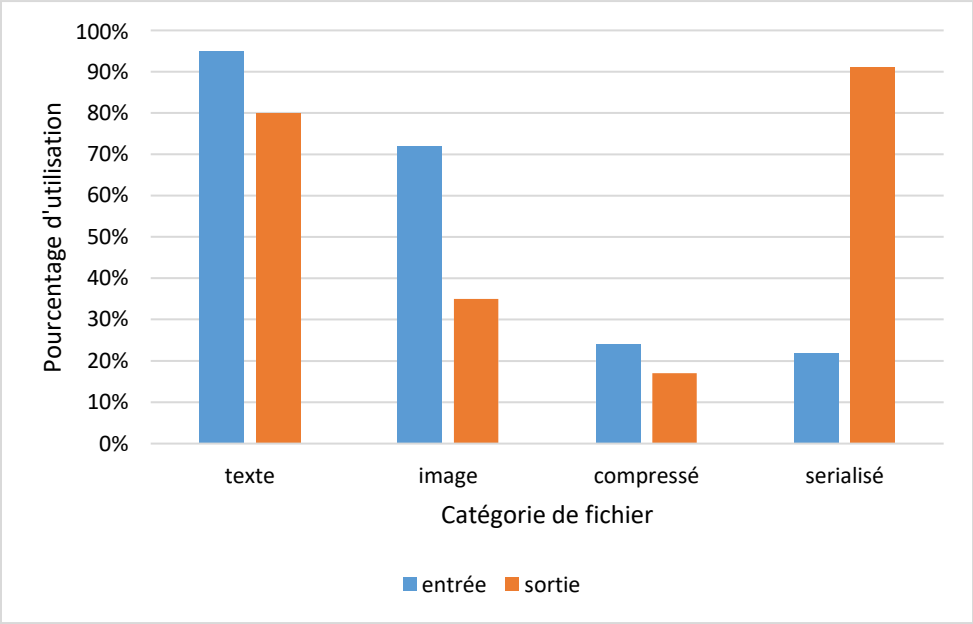


Figure 3.8 Comparaison de l'utilisation de fichiers par catégorie et par mode d'interaction

La dominance de fichiers texte et image en entrée (les fichiers texte et image sont utilisés respectivement par 95% et 72% de projets en entrée, contre 24% et 22% respectivement pour les fichiers compressés et sérialisés) permet de conclure que les artéfacts d'entrée sont généralement stockés sous un format compréhensible par l'Homme. Cela s'explique par le fait que les artéfacts d'entrée sont constitués des données et paramètres d'entraînement, qui sont souvent gérés manuellement par les développeurs dans les phases de nettoyage et d'exploration des données d'une part [71, 75, 76], et les phases de réglage des hyperparamètres d'autre part [77, 78].

De plus, on observe une prédominance de fichiers sérialisés parmi les artéfacts de sortie. Étant donné que les fichiers sérialisés sont généralement stockés sous forme binaire, il n'est pas possible de les consulter directement, et une librairie spéciale devrait être utilisée afin de les désérialiser avant leur utilisation. Aussi, on note une grande utilisation de fichiers texte en sortie. Ces deux résultats s'expliquent assez logiquement par le fait que les artéfacts de sortie sont généralement les

[17] '*Pickle*' est une librairie native sous python permettant la sérialisation de données sous forme de fichiers externes et leur désérialisation pour les réintégrer à un programme en cours d'exécution.

modèles et leurs métriques d'une part, et les données prédites par le modèle d'autre part. Ainsi, les modèles, qui sont des objets dynamiques générés par le programme, sont exportés sous forme sérialisée, tandis que les métriques et les données prédites sont stockées sous forme lisible.

Un point intéressant à noter ici est la faible utilisation de fichiers images en sortie (utilisée par seulement 35% de projets), ce qui soulève un questionnement concernant les méthodes d'évaluation et de validation des modèles. En effet, tel que décrit dans la partie 2.4.1, l'évolution des métriques au cours de l'entraînement du modèle permet d'évaluer les tendances d'entraînement et de prédiction. Une méthode d'analyse de cette évolution consiste à construire automatiquement des graphiques d'évolution lors de l'entraînement et à les sauvegarder pour consultation [79]. Cependant, l'absence d'utilisation de fichiers images en sortie laisse ainsi supposer deux hypothèses :

1. Les graphiques pourraient être générés durant l'exécution du programme, mais sans sauvegarde dans le dépôt;
2. Au lieu de construire les graphiques d'évolution pendant l'entraînement du modèle, les développeurs préféreraient sauvegarder ces données sous forme structurée dans des fichiers textes, et construire les graphiques à partir de ces données historiques.

La deuxième hypothèse pourrait expliquer la prédominance de fichiers '*.json'* pour les artéfacts de sortie. En effet, *JSON* est un format de représentation de données sous forme clé-valeur, qui facilite la compréhension par les développeurs et les programmes. Ceci justifierait aussi le choix de cette méthode de traçage des métriques par DVC, l'une des plateformes de gestion du cycle de vie des programmes d'AM que nous analyserons plus en détail dans la partie 4.2.1.

> **Pratique 5** : Les développeurs open-source ont tendance à ne pas conserver les artéfacts d'AM dans les dépôts logiciels, principalement les artéfacts de sortie

Afin de comprendre l'organisation des artéfacts dans les dépôts logiciels, nous avons analysé les noms de fichiers renvoyés par les heuristiques h_2 et h_3. Chaque dépôt a été réinitialisé et rendu tel qu'il était lorsqu'il a été téléchargé (avant son exécution par le script de h_2). Pour chaque nom de

fichier d'artéfact, nous avons déterminé sa localisation absolue[18] par comparaison et concaténation avec le dossier parent du fichier de code dans lequel cet artéfact a été identifié. Nous avons ainsi pu déterminer la proportion d'artéfacts sauvegardés dans le dépôt, récapitulé dans la Figure 3.9.

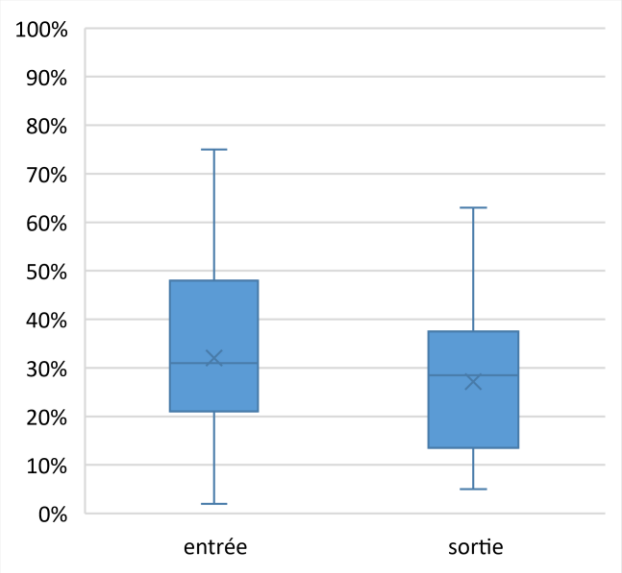


Figure 3.9 Distribution des proportions d'artéfacts présents dans les projets

Pour chaque projet, nous avons calculé, le pourcentage d'artéfacts présents (existant lors du clonage du dépôt) par rapport au total des artéfacts identifiés dans le dépôt. La Figure 3.9 présente la distribution de ces artéfacts dans les projets (artéfacts d'entrée à gauche, et artéfacts de sortie à droite). En mesurant l'intervalle interquartile, on observe que 50% des projets possèdent : (a) entre 21 et 48% de leurs artéfacts d'entrée qui sont présents dans le dépôt, et (b) entre 14 et 37% de leurs artéfacts de sorties qui sont présents dans le dépôt. Cela démontre qu'une majorité de projets ne sauvegardent pas les artéfacts dans leurs dépôts, et que cette pratique est encore plus prononcée quand il s'agit des artéfacts de sortie.

Ce phénomène pourrait s'expliquer par :

[18] Un fichier peut être accédé via sa localisation relative ou absolue. La localisation relative consiste en la position du fichier par rapport à un dossier ou un autre fichier du système, tandis que la localisation absolue consiste au chemin unique d'un fichier en partant du dossier racine du système

- Les limitations des systèmes de contrôle de version traditionnels relatifs à la gestion des artéfacts d'AM. En effet, tel qu'abordé dans la partie 2.4.2, les artéfacts d'AM étant plus difficiles à gérer que les artéfacts logiciels [2], les systèmes de contrôle de version traditionnels construits principalement pour les artéfacts logiciels ne prennent pas en charge la gestion des artéfacts d'AM [8]. *GitHub* limite par exemple la taille des fichiers à 100 Mo, alors que les fichiers de données ou les modèles finaux des applications d'AM sont généralement très lourds (plusieurs Go);
- La sécurité et confidentialité des données pour les projets travaillant avec des données sensibles (données médicales ou privées par exemple) qui empêche leur stockage dans des dépôts publics.

> **Pratique 6** : Il n'existe pas de convention générale pour l'organisation et la dénomination des artéfacts d'AM. Chaque développeur open-source utilise sa propre méthode pour organiser et classifier ces artéfacts.

La convention de nommage est une méthode consistant à définir un ensemble de règles régissant les noms d'objets [80]. Dans le cas des systèmes logiciels, ces règles de nommage peuvent s'appliquer aux artéfacts logiciels de haut niveau (fichiers, modules, paquetages, librairies, etc.) et de bas niveau (variables, fonctions, etc.). Il a été montré que cette technique améliore la compréhensibilité et la maintenabilité des logiciels [81, 82], et peut permettre d'automatiser la reconstruction de l'architecture d'un programme [83].

Ainsi, afin de vérifier si les projets suivaient une quelconque tendance générale et/ou logique pour l'organisation et le nommage des artéfacts d'AM, nous avons procédé à une analyse manuelle des artéfacts existants dans les dépôts. Pour cela, nous avons sélectionné aléatoirement 50 projets. Nous avons ensuite analysé le code source et la documentation de ces projets, puis nous avons étudié les noms des dossiers, sous dossiers et leurs fichiers afin de classifier les artéfacts en 5 catégories :

- Données : regroupe les données d'entraînement, de test (seulement les données de test du modèle et non ceux relatifs au test du logiciel final), de validation et les données prédites;
- Configuration : fichiers où sont spécifiés les paramètres et hyperparamètres du modèle. Les fichiers de configuration du logiciel (déploiement, variables, etc.) sont ignorés;

- Code : Tout fichier de code participant à au moins une étape du processus de création du modèle (entraînement, test, validation ou déploiement) ;
- Modèle : fichiers stockant les modèles finaux sous forme sérialisée;
- Métriques : fichiers (image ou texte) contenant les métriques d'un ou plusieurs modèles.

Nous n'avons pu identifier aucune habitude de nommage pouvant être généralisée à tous les projets. Ceci pourrait expliquer la faible intersection de l'heuristique h_1 avec les autres, dans la mesure où les expressions régulières utilisées supposaient l'existence d'une telle convention de nommage.

En compilant les données obtenues, nous avons néanmoins pu identifier trois modes d'organisation (non liés à une quelconque convention de nommage), dont les proportions sont compilées dans la Figure 3.10.

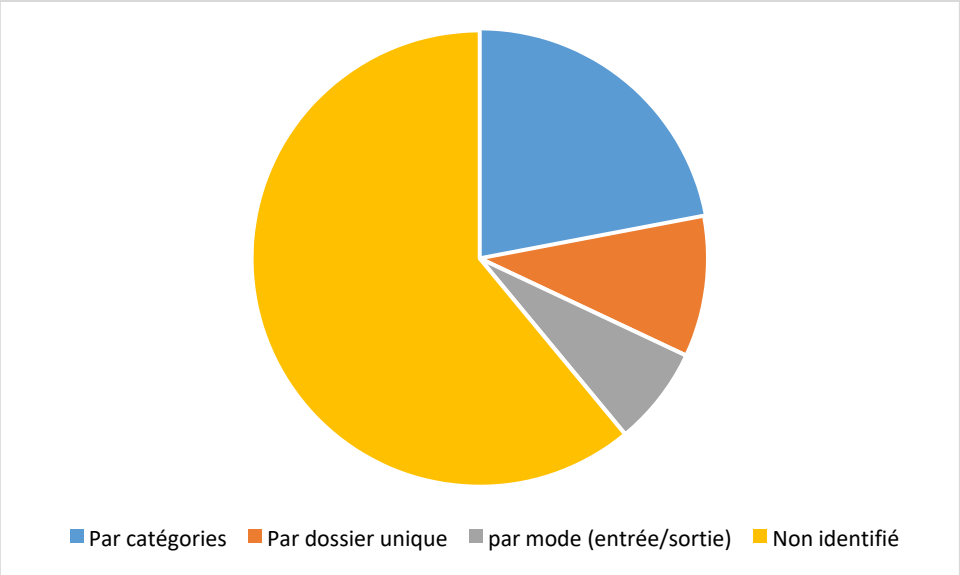


Figure 3.10 Méthodes d'organisation des artéfacts

1. Organisation par catégories : Les artéfacts de chaque catégorie sont regroupés au sein d'un même dossier. 22% des projets étudiés présentaient cette caractéristique, mais sans convention de nommage. L'on observe néanmoins une tendance à utiliser des noms plus descriptifs comme '*ParisHousingClass*' ou '*age_gender'*;
2. Organisation dans un seul dossier : 10% des projets ne possédaient qu'un seul dossier où tous les artéfacts étaient rassemblés;

3. Organisation par mode (entrée/sortie) : Identifié dans 7% des projets, cette méthode regroupe les artéfacts d'entrée et de sortie dans des dossiers dédiés.

Dans 61% des cas, aucune méthode particulière d'organisation n'a pu être identitée, car les artéfacts de même catégorie ou de même mode étaient dispersés entre différents dossiers. Cependant, nous

> **Pratique 7** : Les développeurs open-source utilisent trois principaux moyens d'intégration des artéfacts d'AM au code : argument de lignes de commande, codage en dur et enfin via des fichiers de configuration. Ils préfèrent néanmoins la première méthode.

souhaitons mitiger ce résultat. Le fait que nous n'ayons pas pu identifier une méthode d'organisation ne signifie pas forcément qu'il n'y en a pas, mais dénote de notre incapacité à analyser et comprendre le projet simplement à partir de la documentation (limitée) et du code disponible dans les dépôts logiciels. Cette limitation de la compréhensibilité des projets d'AM est abordée plus en détail dans le Chapitre 4. Des interviews avec les développeurs pourraient aider dans ce cas de figure, mais cela est hors de la portée du présent travail[19].

L'analyse manuelle du code nous a permis d'identifier davantage d'artéfacts qui n'avaient pu être extraits par les heuristiques[20] h_2 et h_3. En effet, ces heuristiques se basent sur l'hypothèse selon laquelle les artéfacts logiciels sont déjà intégrés au code et qu'aucune intervention manuelle n'est requise lors de l'exécution. L'étude des méthodes d'intégration a révélé que dans certains cas, une intervention manuelle est nécessaire au moment de l'exécution du code. En effet, la Figure 3.11 démontre que la méthode prédominante d'intégration des artéfacts est l'utilisation des arguments de ligne de commande. Ce résultat confirme la prévalence de l'utilisation de la librairie '*argparse'* telle qu'identifiée dans la Figure 3.2.

[19] La méthodologie priorisée dans ce travail est le minage des dépôts logiciels, sans aucune interaction avec le développeur afin de réduire le biais dû à la subjectivité.

[20] Les limitations des heuristiques utilisées seront traitées dans la partie 3.7 (obstacles à la validité).

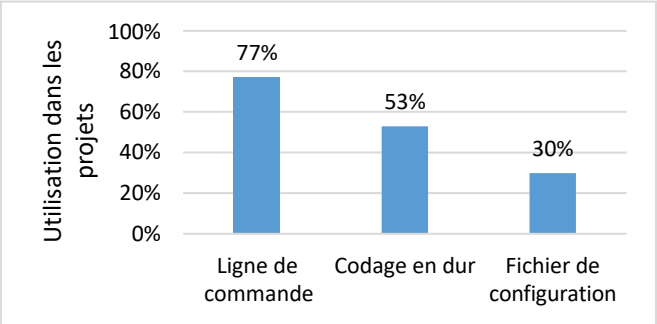


Figure 3.11 Prévalence des méthodes d'intégration d'artéfacts dans les projets

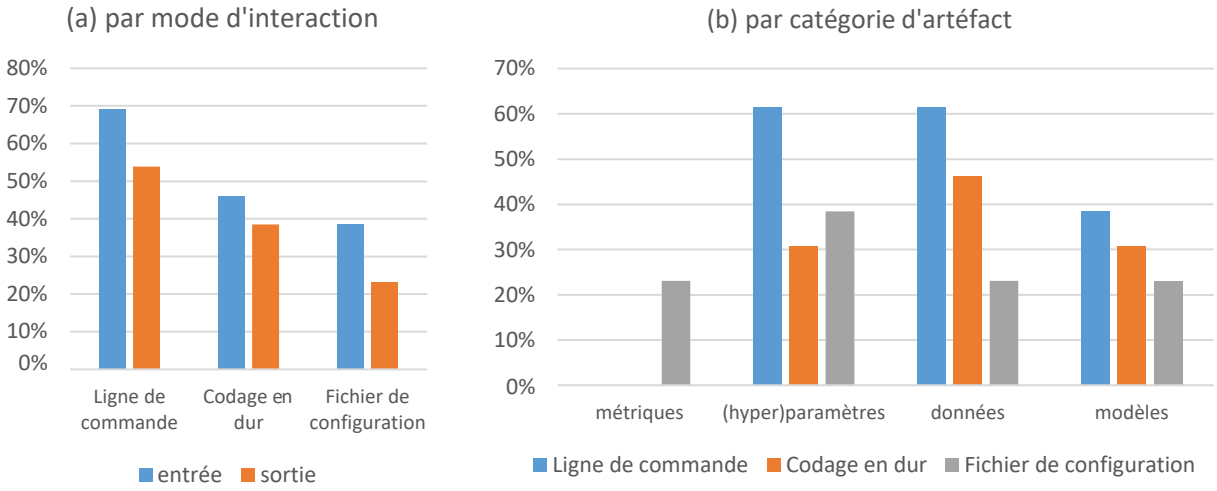


Figure 3.12 Distribution des projets par mode d'intégration et par catégorie d'artéfact

La préférence envers la ligne de commande s'explique par la flexibilité qu'offre cette méthode. Elle permet notamment de fournir des valeurs aux variables du code de façon ad-hoc et générique [84]. Entre autres, l'on remarque aussi une utilisation importante du codage en dur[21] qui est pourtant identifié comme un anti-pattern [85-87].

La Figure 3.12 détaille davantage les méthodes d'intégration en les classifiant selon le mode d'intégration et les catégories d'artéfact. L'on remarque que la ligne de commande est préférée pour les artéfacts d'entrée tout comme ceux de sortie, qu'importe le type d'artéfact (à l'exception des métriques qui sont sauvegardées dans des fichiers de configuration). Cela s'explique par le fait

[21] Le codage en dur consiste à spécifier la valeur de certaines variables de configuration directement dans le code

que les valeurs des métriques sont toujours calculées lors de l'exécution du programme. Il n'est donc pas possible de les spécifier manuellement via la ligne de commande ou par codage en dur.

3.5 QR3 : Comment les projets d'AM évoluent-ils?

3.5.1 Motivation

L'analyse de l'évolution des dépôts est un sujet important dans le domaine de l'ingénierie logicielle. Il permet notamment d'identifier les phases de développement du logiciel [88], de détecter les disruptions [89], et d'étudier le couplage entre différents artéfacts logiciels [90]. Pour le cas particulier des applications d'AM, nous souhaitons identifier d'une part les tendances d'évolutions et de modification des artéfacts, et d'autre part quel est le degré de couplage entre les artéfacts du point de vue évolutif (i.e. comprendre si la modification d'un artéfact ou d'une catégorie d'artéfact entraînerait la modification d'un autre).

3.5.2 Approche et résultats

Nous avons analysé les commits des dépôts. Les commits sont des composants élémentaires utilisés pour mettre à jour les artéfacts des dépôts sur Github. Nous avons d'abord comparé la durée de vie (période entre le premier et le dernier commit) des dépôts et le nombre de commits afin de déterminer s'il existe une corrélation entre ces deux variables. Les Figure 3.13 (a et b) montrent respectivement la proportion du nombre de commit des dépôts et leur durée de vie (en jours). Les fichiers cachés ont été ignorés lors de la détermination de la taille du dépôt. A cause de la grande disparité existant entre les dépôts, les valeurs aberrantes ont été supprimées des figures.

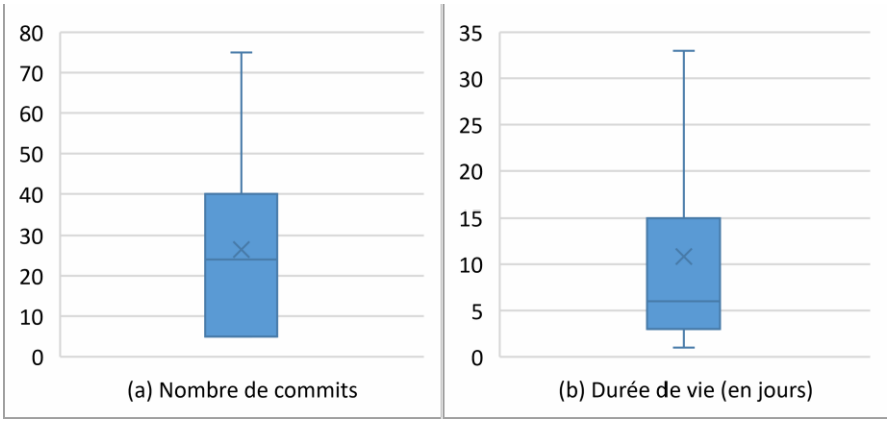


Figure 3.13 Distribution du nombre de commits et de la durée de vie des dépôts

Une première observation permet de conclure que le nombre de commits, et la durée de vie des dépôts sont assez faibles. Avant la suppression des valeurs aberrantes, l'on a pu observer une forte disparité entre les dépôts pour ces deux variables. Ces résultats s'expliquent par le fait que le corpus '*Papers With Code'* contient principalement des projets de recherche, et les développeurs pourraient travailler en interne, publier leur code en une seule fois et n'effectuer que quelques réglages ultérieurs, ce qui expliquerait le faible nombre de commits et la durée de vie.

Nous avons néanmoins analysé les fichiers modifiés par les commits afin de déterminer la proportion de modifications affectant les artéfacts d'AM.

> **Pratique 8** : Les artéfacts de code sont les plus modifiés, et les autres (données, configuration, etc…) ne le sont que très peu. Lorsque c'est le cas, seules des modifications mineures sont apportées.

La Figure 3.14 présente la distribution des artéfacts modifiés par les commits : l'axe des ordonnées représente le pourcentage de commits modifiant une catégorie d'artéfacts.

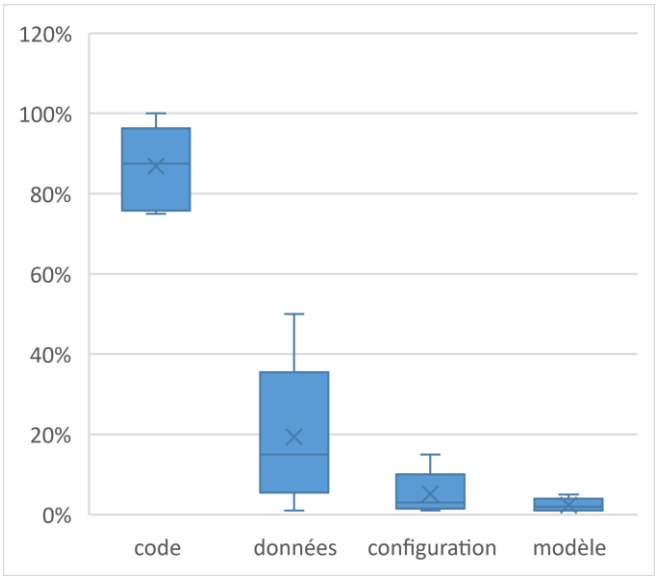


Figure 3.14 Distribution des commits modifiant les artéfacts d'AM, regroupée par catégories

L'on remarque que les artéfacts de code sont le plus modifiés, avec un intervalle interquartile se situant entre 76 et 96%, ce qui signifie que dans la moitié des projets, entre 76 et 96% des commits

modifient un ou plusieurs artéfacts de code. Ceci n'est pas le cas pour les autres artéfacts, dont l'intervalle interquartile se situe en dessous de 40%.

Git permet d'identifier le type de modification (création, suppression, modification) ayant affecté un fichier. Il permet aussi de voir quelles lignes ont été ajoutées, supprimées ou modifiées, et leur nombre. En utilisant ces capacités, nous avons compilé le nombre de lignes de chaque type. De tous les projets étudiés, toutes les modifications aux données n'étaient que des modifications de lignes existantes. Aucune addition ou suppression de ligne n'a été observée. Aussi, le nombre de lignes modifiées est assez faibles, chaque modification n'affectant que quelques lignes (entre 1 et 5 lignes en moyenne par projet et par modification). Ceci, couplé au fait que très peu de commits ont été trouvés dans les dépôts, soutient le fait qu'au niveau des commits, les artéfacts d'AM ne sont que très peu modifiés. Nous souhaitons néanmoins préciser qu'un commit n'est pas forcément équivalent à une modification. Un fichier peut être modifié plusieurs fois, mais Git n'enregistrera ces modifications que lorsque le commit sera exécuté (toutes les modifications effectuées seront groupées en une seule, du point de vue de *Git*).

Ainsi, le faible taux de commit et de modification des artéfacts d'AM ne nous ont pas permis d'analyser l'évolution des dépôts d'AM plus en profondeur.

3.6 Implications et recommandations

- **Pour les développeurs d'applications d'AM :** Les résultats de la présente étude dressent un portrait global des pratiques les plus observées dans les dépôts d'AM, ainsi que leur prévalence. Les développeurs novices (n'ayant pas assez d'expérience sur les pratiques de développement d'AM) pourraient exploiter ces résultats afin de mieux gérer leurs projets principalement à l'étape de planification. Par exemple, les résultats de la **QR2** peuvent aider pour le choix d'une méthode de stockage, d'organisation ou d'intégration des artéfacts d'AM;
- **Pour les chercheurs :** l'observation de certaines pratiques telles que le codage en dur des chemins des artéfacts (identifié comme anti-pattern) ou le fait qu'une grande partie des artéfacts d'AM ne sont pas intégré à l'outil de gestion de version (contrairement à ce qui est recommandé pour les applications logicielles) ouvre la voie à davantage de recherche relativement aux pratiques de développement d'AM. Cette étude s'étant limitée à décrire

ces pratiques, nous n'avons porté aucune critique particulière (positive ou négative) concernant l'impact de ces pratiques sur les projets. Un axe de recherche pertinent pourrait être d'étudier l'impact des pratiques observées dans les projets d'AM relativement aux critères standards de qualité par exemple (maintenabilité, performance, sécurité, etc.);

- **Pour les développeurs des plateformes d'AM**: les méthodes identifiées par la présente étude pourraient servir à implémenter des fonctionnalités spécifiques comme la classification et l'organisation automatique des artéfacts dans le but de faciliter leur gestion et leur intégration au code.

3.7 Obstacles de validité

L'étude abordée dans ce chapitre tente d'obtenir une image des pratiques utilisées dans le développement d'applications d'AM. Nous avons analysé les données intrinsèques qui capturent différents aspects de ces projets et caractérisent leur processus de développement. Ces mesures et données, cependant, présentent certaines limites.

Premièrement, les méthodes heuristiques utilisées pour identifier les artéfacts d'AM peuvent avoir entraîné des faux positifs et des faux négatifs. En effet, ces heuristiques partent du postulat que les fichiers chargés dans le code sont automatiquement des artéfacts d'AM, ce qui pourrait ne pas être cas : un fichier de log par exemple est automatiquement considéré comme artéfact d'AM, mais il pourrait en fait être lié au sous-système utilisant le modèle d'AM final. Il pourrait donc ne pas être considéré comme artéfact d'AM. Ce point est assez mitigé néanmoins, dans la mesure où l'identification d'un artéfact comme étant d'AM ou logiciel serait suggestif. Une recherche supplémentaire sur ce sujet devrait être menée afin d'établir clairement et si possible, les limites entre les artéfacts de logiciel et d'AM, lorsqu'ils sont stockés dans un même dépôt.

Entre autres, l'analyse manuelle a permis d'identifier une partie des artéfacts qui ont été manqués par l'analyse heuristique. Étant donné que les techniques de l'analyse heuristique ont été utilisées pour construire un système d'identification automatique des artéfacts d'AM à partir du code source, cette limitation sera abordée dans la partie afférente (5.4.3) où les performances de cette méthode seront analysées plus en profondeur. Cette méthode offre un taux de détection de 60% des artéfacts dans sa version basique (celle utilisée dans ce chapitre).

Deuxièmement, en échantillonnant les fichiers et en les analysant manuellement pour savoir comment les développeurs traitent les données, nous avons peut-être manqué des informations importantes provenant d'autres dépôts, à cause de la marge d’erreur qui est de 8%. Cependant, nous pensons que les résultats obtenus pourraient être généralisables, en prenant en compte les variations dues au taux d’erreur.

3.8 Conclusion

Ce chapitre présente un aperçu des pratiques adoptées par les développeurs lors de la construction des applications d’AM. En utilisant le minage des dépôts logiciels et l’analyse statistique, nous avons identifié et analysé différentes méthodes utilisées par les développeurs, regroupées sous forme de 8 pratiques générales.

Cependant, nous croyons qu’une partie de ces pratiques est fortement influencée par les limites des systèmes traditionnels de gestion du cycle de vie des logiciels, ce qui expliquerait l’attrait grandissant de la communauté de développeurs pour les plateformes spécialisées de gestion du cycle de vie des modèles d’AM. Les pratiques identifiées se basent sur les VCS traditionnels, et le passage à des plateformes spécialisé présente un certain nombre de défis découlant de ces pratiques que nous souhaitons analyser dans le chapitre suivant.

CHAPITRE 4 ADOPTION/MIGRATION DES PLATEFORMES MLOPS : DÉFIS, SOLUTIONS, RECOMMANDATIONS ET PERSPECTIVES

4.1 Introduction

Au vu du manque de services ciblés pour les logiciels d'AM au sein des systèmes de contrôle de version traditionnels (VCS), de nombreuses plates-formes spécialisées pour la gestion du cycle de vie des applications ont vu le jour, sous le nom de plateformes MLOps (Machine Learning Operations). Ces outils tirent parti des pratiques d'ingénierie logicielle et du DevOps afin d'améliorer la façon dont les développeurs créent, exploitent et maintiennent des applications d'AM. Ce chapitre vise à identifier les principaux défis auxquels les développeurs sont confrontés lors de l'adoption et/ou de la migration de projets d'AM existants vers ces plateformes. Grâce à une analyse expérimentale et statistique, nous explorons une méthodologie de migration générique et enregistrons les différents défis rencontrés à chaque étape de la migration. Sur la base de notre propre expérience, nous recommandons également des solutions potentielles pour surmonter ces défis et proposons quelques étapes que les développeurs peuvent suivre lors de la création de projets d'AM afin de faciliter toute migration future vers les plates-formes MLOps.

4.2 Contextualisation

Le développement d'applications d'AM implique des étapes, activités et artéfacts différents par rapport aux autres types de logiciels [2]. De nombreuses études ont mis en évidence les limites majeures des pratiques et des outils traditionnels de génie logiciel en matière de gestion des activités de développement de systèmes d'AM. Ils fournissent souvent un support limité ou inexistant pour la gestion des versions des données, la gestion des pipelines et la reproductibilité des modèles [9, 91, 92].

Compte tenu de ces limitations, différentes méthodologies telles que TDSP (Team Data Science Process) [93], KDD (Knowledge discovery in databases) [94], et CRISP-DM (CRoss Industry Standard Process for Data Mining) [95] ont été proposées pour gérer les données et les étapes impliquées dans les processus d'AM. Ces processus mettent en évidence des activités communes qui peuvent être regroupées sous une méthodologie plus large et générique appelée Cycle de vie

de l'Apprentissage Machine ou Machine Learning Life Cycle (MLLC). Il s'agit d'un processus systématique composé de différentes étapes et activités qui assurent l'optimisation du développement, l'exécution et la maintenance des applications d'AM. Pour faciliter ces activités, de plus en plus de développeurs ont recours aux plateformes MLOps[22].

Ces plateformes visent à gérer les différentes étapes et artéfacts impliqués dans le développement d'AM, comme le montre la Figure 2.5. Ils fournissent généralement les fonctionnalités principales suivantes : gestion de version du modèle, configuration et exécution du pipeline, suivi des artéfacts, reproduction des expérimentations et réglage des hyperparamètres [96, 97]. Compte tenu du potentiel de ces nouveaux outils, de plus en plus de développeurs les adoptent pour gérer leurs processus de développement d'AM. Cependant, l'utilisation de ces outils n'en est qu'à ses débuts et ils sont encore à l'étape d'exploration par les développeurs [9]. Ainsi, il n'existe pas de connaissances générales ou de méthodologie pouvant faciliter et guider leur adoption. Cela entraîne une augmentation de la charge de travail nécessaire lors de la configuration et de la maintenance de ces outils.

4.2.1 Plateformes étudiées

Nous étudions principalement *DVC* (Data Version Control) [56] et *MLflow* [57], deux plateformes MLOps open source populaires. Nous pensons que les résultats fournis par l'analyse de ces plateformes pourraient être généralisables aux plateformes MLOps.

DVC a été annoncé publiquement par *Iterative*[23] en mai 2017. Initialement développé pour la gestion des versions de données, il a évolué pour inclure la gestion des expérimentations et la gestion des versions. *MLflow* quant à lui a été annoncé pour la première fois en 2019 par *Databricks*[24] et intégré à l'écosystème *Apache Spark*, qui regroupe un ensemble d'applications spécialisées pour la gestion des grandes quantités de données [100]. Ces deux plateformes ont été choisies pour deux raisons :

---

[22] Voir partie 2.5

[23] [98] "Iterative, Developers tools for Machine Learning." https://iterative.ai/ (accessed June 13, 2021).)

[24] [99] "Databricks, the Data and AI company." https://databricks.com/ (accessed June 15, 2021).)

1. Leur popularité. En effet, elles ont respectivement 8,1k et 9,5k étoiles sur *GitHub*. Les étoiles sont un moyen offert à la communauté de développeurs par *Github* afin de noter les dépôts logiciels [101]. Dans la littérature, elles ont été associées à l'appréciation par la communauté [102] et à la popularité des dépôts [103];
2. Leur couverture des fonctionnalités générales liées à la gestion d'AM que sont : (a) le traçage des artéfacts, (b) la configuration et l'exécution du pipeline, et (c) la gestion des versions des artéfacts et des modèles. Chaque plateforme gère ces activités d'une manière spécifique.

4.2.2 Fonctionnalités étudiées

4.2.2.1 Traçage des artéfacts

MLflow assure le suivi des artéfacts via son module *MLflow Tracking*. Il s'agit d'une API qui enregistre les paramètres, les versions de code, les métriques et les artéfacts lors de l'exécution du code. Pour cela, des instructions de code spécifiques (telles que `'mlflow.log_artifact()'`) doivent être ajoutées au code d'AM principal pour que ces artéfacts soient suivis. *DVC* gère le suivi en utilisant des fichiers externes et des paires clé/valeur. Il est nécessaire d'enregistrer les paramètres importants du modèle dans un fichier externe, qui est fourni en entrée pour que *DVC* puisse suivre les modifications apportées à chaque paire clé/valeur.

4.2.2.2 Configuration et exécution du pipeline

Dans les deux plates-formes, les flux de travail d'AM sont gérés via des fichiers de pipeline basés sur *yaml*[25] [104] qui spécifient les nœuds, les lignes de commande et les paramètres du pipeline.

Un pipeline est la représentation et l'implémentation logicielle du flux d'AM. Les nœuds du pipeline représentent alors les étapes du flux d'AM, et spécifient la commande permettant de lancer l'exécution de l'étape correspondante à ce nœud. Par exemple, l'étape d'entrainement du flux

[25] YAML est un langage de sérialisation de données permettant de stocker et gérer des données de configuration de façon organisée tout en étant facilement compréhensible par l'Homme et par les logiciels.

d'AM peut être représenté par un nœud nommé « train » dans le pipeline, et sa commande de lancement peut être `'python run.py -step train'`.

L'implémentation spécifique dans chaque plateforme est légèrement différente, mais l'idée de configuration générale est la même sur les deux plates-formes, comme le montre la Figure 4.1.

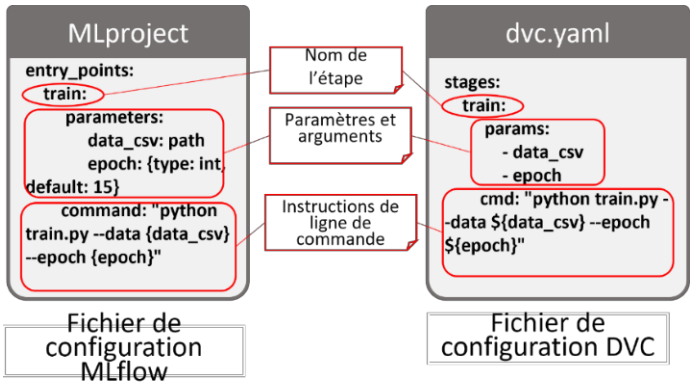


Figure 4.1 Exemple de configuration des nœuds du pipeline d'AM dans *MLflow* et *DVC*

L'exécution du pipeline se fait en restaurant les nœuds du graphe de dépendance spécifié implicitement dans le fichier de configuration (appelés '*entry_points'* dans *MLflow* et *'stages'* dans *DVC*), puis en les exécutant dans le bon ordre.

4.2.2.3 Gestion des versions des artéfacts et des modèles

Les deux plates-formes utilisent *Git* pour contrôler les versions des modèles et des artéfacts. Malgré le fait que *DVC* possède une couche de gestion de version personnalisée par-dessus *Git* (en créant un fichier texte qui stocke le hachage md5 et l'emplacement de l'artéfact ou du modèle), les deux plates-formes s'appuient toujours sur des commits pour contrôler la version des artéfacts et des modèles après chaque exécution du pipeline. De plus, *MLflow* fournit un magasin de modèles (model store en anglais) pour enregistrer, stocker, déployer et exécuter des modèles.

4.3 Objectifs

Ce chapitre vise à identifier les défis potentiels auxquels les développeurs peuvent être confrontés lors de l'intégration de leurs projets d'AM dans les plateformes MLOps. Cette intégration implique deux activités possibles selon le moment où l'outil est utilisé pour gérer le projet d'AM :

a) L’adoption : fait référence à l'utilisation de l'outil dès le début du développement du projet. Au cours de cette phase, un certain nombre de décisions de conception et d'architecture sont prises qui conduiront aux activités suivantes;

b) La migration : il s’agit de l’intégration à une phase tardive du développement, lorsque la plupart du code et des artéfacts sont déjà produits. Par conséquent, l'intégration consiste principalement à modifier le projet d'origine en éditant, ajoutant ou supprimant les bibliothèques, l’architecture et/ou les artéfacts.

À travers une étude expérimentale détaillée, nous identifions les principaux défis liés à ces deux activités. Nous avons réalisé la migration de ces projets vers les plateformes *DVC* et *MLflow*, et avons enregistré les principaux défis auxquels nous avons été confrontés. Ensuite, nous proposons des options et des solutions que nous avons appliquées afin de surmonter ces défis. En utilisant l'analyse statistique, nous fournissons également un aperçu de l'impact de ces défis sur les activités d'adoption précoces et fournissons des recommandations sur les pratiques que les développeurs peuvent adopter lors de la création de projets d’AM pour une meilleure organisation, et ainsi aider à réduire l'impact de ces défis lors de toute future activité migratoire.

4.4 Méthodes et données d’analyse

Afin d’évaluer les défis liés à la migration/adoption des plateformes MLOps, nous avons réalisé une étude expérimentale sur 13 projets du corpus de données ‘*Papers With Code’* [11].

4.4.1 Sélection des projets à analyser

A partir du corpus initial (2681 projets d’AM[26]), nous avons d'abord filtré l'ensemble de données pour extraire uniquement les projets compatibles avec python-3 et possédant un fichier ‘*README’* et un fichier ‘*requirements.txt’*. Nous supposons que ces fichiers servent à décrire le projet en termes d’utilisation [73] d’une part et de dépendances d’autre part [74]. De cette première étape de filtrage, nous avons obtenu 1847 projets. Nous avons ensuite extrait aléatoirement un échantillon de 40 projets possédant au moins 30 commits (afin de s’assurer que les projets ont un certain

[26] Voir partie 1.5.1

historique). Enfin, nous avons exclu les projets incomplets ou non pertinents. Les projets incomplets sont des projets avec une documentation pauvre, signifié par l'absence du nom du jeu de données d'entraînement utilisé et des instructions sur la façon d'exécuter le projet. Les projets non pertinents sont les projets qui n'entraînent aucun modèle d'AM, mais qui introduisent plutôt un nouvel algorithme. Pour exclure ces projets, nous avons vérifié manuellement les fichiers '*README'* des 40 projets et supprimé les projets qui ne répondaient pas aux les critères suivants :

- Le projet n'est pas une API et peut être exécuté à partir de la ligne de commande;
- Le projet comporte au moins une étape d'entraînement du modèle;
- Le dépôt contient la liste des instructions (avec les commandes à exécuter à chaque étape) sur la façon de le reproduire.

Finalement, nous avons obtenu 13 projets, récapitulés dans le Tableau 4.1. La colonne *Nombre de commits* fait référence au nombre de commits dans le projet au moment de l'analyse[27], et la colonne *taille* est la taille totale du dépôt, à l'exclusion des données d'historique (tout le dossier caché '*.git'* a été ignoré).

Tableau 4.1 Récapitulatif des projets analysés

Projet	**nom**	**Nombre de commits**	**Taille (Kb)**
p_1	YerevaNN/DIIN-in-Keras	113	657
p_2	hitvoice/DrQA	35	810
p_3	vuanhtu1993/Keras-SRGANs	36	518609
p_4	somewacko/deconvfaces	65	1842
p_5	inningbytes/deep-MLsa	36	4828
p_6	saikrishna-1996/deep_pepper_chess	429	373857
p_7	allenai/dqa-net	311	247
p_8	manideep2510/eye-in-the-sky	93	167881
p_9	abhi4ssj/few-shot-segmentation	45	507968
p_{10}	ai-med/quickNAT_pytorch	92	358518
p_{11}	brain-research/realistic-ssl-evaluation	30	173
p_{12}	ryandgoldenberg1/svrg_projectt	160	3836
p_{13}	Baidi96/text2sql	35	59939

[27] Les projets ont été téléchargés le 15 janvier 2021.

4.4.2 Analyse expérimentale et statistique

Tout d'abord, les projets ont été clonés dans une machine virtuelle exécutant Centos 7 et utilisant *Anaconda* [105] pour gérer les environnements python. Ensuite, nous avons suivi un processus itératif consistant en un ensemble d'étapes séquentielles inspirées du processus de migration de données [106] comme le montre la Figure 4.2 (chaque étape est discutée plus en détail dans la section 4.5). Après avoir cloné un projet, nous avons configuré son environnement initial en fonction de sa documentation, notamment en installant des dépendances à partir du fichier '*requirements.txt'*, puis en éditant le code source si nécessaire (6 projets possédaient des variables qui devaient être complétées manuellement afin d'exécuter le projet).

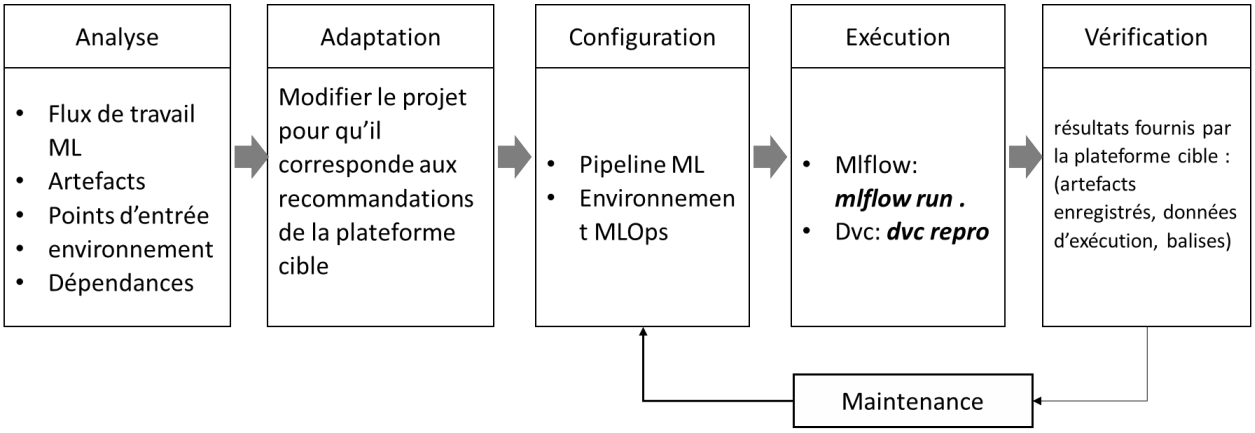


Figure 4.2 Étapes du processus de migration appliqué dans l'étude

Nous avons limité la modification du code source au strict minimum (c'est-à-dire là où cela est explicitement mentionné dans la documentation du projet et lorsque cela est nécessaire pour que le projet s'exécute correctement). Ceci afin d'éviter des modifications inutiles aux métriques du projet (complexité, lignes de code et autres) et de s'assurer que le projet est aussi proche que possible de l'original. Ainsi, les seules modifications autorisées dans cette phase étaient liées à la saisie de données de variables vides telles que des chemins de fichiers statiques ou des données de configuration.

Sur la base de la documentation du projet, nous avons identifié les exigences d'environnement pour chaque projet, à savoir la version python, les exigences matérielles, les dépendances et les requis système. Nous avons utilisé une méthode par tâtonnement (essais et erreurs) pour identifier d'autres exigences qui manquaient dans la documentation. Nous avons ensuite construit l'environnement du projet et exécuté les différentes étapes de migration. Lorsque des erreurs liées à des dépendances

ou à des insuffisances matérielles (par exemple GPU) apparaissaient, nous mettions à jour les exigences avec les dépendances manquantes et redémarrions le processus jusqu'à ce que le projet soit complètement exécuté ou jusqu'à ce que nous obtenions des erreurs ingérables, c'est-à-dire toute autre erreur non liée à des dépendances manquantes.

Au cours de l'analyse expérimentale, nous avons également collecté des statistiques sur la façon dont les projets organisent et exécutent les étapes d'AM. Ces statistiques sont utilisées pour analyser et expliquer la prévalence de certains défis et/ou pratiques dans les projets.

4.5 Rapport des expérimentations : défis identifiés et solutions proposées

Au cours du processus de migration expérimentale décrit précédemment (partie 4.4.2), nous avons rencontré un certain nombre de difficultés, qui ont été organisées et présentées sous la forme de 11 défis répartis en 6 catégories : analyse, configuration des environnements, configuration du pipeline d'AM, exécution et automatisation, vérification et enfin maintenance. Pour chacun de ces défis, nous présentons la cause et l'impact du défi; nous discutons aussi des options permettant d'atténuer ces défis. Ces éléments sont récapitulés dans l'annexe C (en anglais).

4.5.1 Phase 1 : Analyse

C'est la première étape qui consiste à analyser et identifier l'architecture, les artéfacts et les dépendances du projet. Cette étape a été principalement réalisée manuellement avec la participation d'un assistant. Nous étions alors deux auteurs[28] à effectuer les activités de migration. Les deux auteurs possèdent une expérience intermédiaire en matière de développement d'applications d'AM.

Nous avons vérifié la documentation et le code source de chaque projet, puis avons créé un fichier JSON nommé '*experiment.json'* où toutes les informations recueillies ont été stockées. Ces informations concernent principalement les artéfacts d'AM et les données du flux d'exécution du projet (étapes d'AM, jeux de données, hyperparamètres, métriques).

[28] Dans le reste de ce chapitre, le terme auteur lorsqu'utilisé pour parler des activités de migration réalisées, représentera l'une des (ou les deux) personnes impliquées dans ces activités.

Les principaux défis rencontrés au cours de cette phase sont liés à l'analyse et à la compréhension.

4.5.1.1 Défi AN1 : Identification complexe du pipeline d'AM du projet

La première tâche lors de la migration était de comprendre le flux d'exécution du projet. Contrairement aux logiciels traditionnels, le développement d'AM nécessite de nombreuses étapes interdépendantes qui sont souvent regroupées sous forme de pipelines d'AM [107]. Chaque étape du pipeline représente une activité spécifique et reproductible, comme l'entraînement, les tests ou l'évaluation, avec ses propres artéfacts et code. Dans les plates-formes MLOps, les pipelines utilisent des fichiers de configuration spécifiques pour enregistrer l'ensemble du processus d'AM et configurer l'environnement du projet. Pour réussir la migration, une bonne connaissance du processus du projet et de la plateforme est nécessaire. Par conséquent, ce défi est pertinent pour les développeurs de projets qui décident (a) d'adopter ou de migrer vers une plateforme MLOps, (b) de basculer entre des plateformes, ou (c) d'intégrer de nouveaux développeurs peu familiers avec les projets ou la plateforme utilisée.

Pour mieux démontrer et comprendre le défi, dans nos expérimentations, nous avons divisé les projets en deux groupes : le premier groupe a été migré d'abord vers *MLflow* puis vers *DVC* et le second d'abord vers *DVC* puis vers *MLflow*. La deuxième migration dans chaque cas a été effectuée par un auteur différent à l'aide d'un rapport produit lors de la première migration. Outre le manque de connaissances préalables des auteurs sur les projets migrés, le défi s'est manifesté plus clairement en termes d'incohérences. Premièrement, les projets n'ont pas explicitement spécifié les phases d'AM dans leur documentation, ce qui est requis par les plateformes. Deuxièmement, si les phases étaient documentées, leurs noms ou leurs nombres n'étaient pas toujours cohérents avec la plateforme cible. Troisièmement, les incohérences entre les deux plates-formes, *MLflow* et *DVC* en termes de phases ont causé des irrégularités lors de la configuration du processus de migration. Enfin, l'absence d'une référence standard convenue entre les auteurs qui ont effectué la migration a causé des problèmes supplémentaires dans la communication et le traitement des rapports. Bien que ce dernier problème découle de notre processus expérimental, il ne s'agit pas d'un scénario irréaliste pour un processus de migration réel.

4.5.1.2 Option AN1 : Documenter le flux d'AM de manière standardisée

Afin de résoudre le défi AN1, nous avons légèrement modifié notre processus expérimental. Nous considérons d'abord que la migration est mieux organisée en deux équipes : l'équipe 'AM principale, chargée d'acquérir les connaissances nécessaires sur le projet, et l'équipe de migration, chargée d'acquérir les connaissances sur les plateformes. Les deux auteurs ont donc repris le rôle de chacune de ces équipes. L'équipe de migration a créé un modèle générique qui contiendrait les informations nécessaires et requises par les plates-formes cibles sur le flux de travail, les métriques, les artéfacts et les paramètres du projet. Ce modèle a été complété par l'équipe 'AM principale avec des valeurs spécifiques au projet, puis renvoyé à l'équipe de migration pour exécuter la migration. Ce fichier a également permis d'automatiser une partie du processus de configuration, en servant de base pour générer automatiquement les fichiers de pipeline pour chaque plateforme.

4.5.1.3 Défi AN2 : Difficulté de retrouver et comprendre les artéfacts et leurs liens

Les plates-formes MLOps exploitent les artéfacts et les liens entre eux pour suivre l'état et l'évolution du projet, mais il n'existe pas de norme pour stocker, organiser et documenter les artéfacts d'AM. La récupération de ces informations est une tâche fastidieuse, manuelle et exigeante [91]. Au cours de nos expériences, l'une des activités les plus difficiles a été d'identifier les artéfacts et leurs informations, puis de les transmettre à la plateforme cible. Cependant, chaque projet avait sa propre façon d'organiser les artéfacts, avec des descriptions d'artéfacts manquantes, éparpillées dans le projet ou, pire encore, enchevêtrées dans le code.

Par exemple, *MLflow* et *DVC* nécessitent tous deux le lien entre un artéfact de jeu de données et le fichier de code dans lequel il est utilisé. Dans *MLflow*, la fonction `'log_artifact()'` utilisée pour enregistrer un artéfact, doit être ajoutée dans le code qui utilise (en entrée) ou produit (en sortie) cet artéfact. Cependant, étant donné les dépendances entre les fichiers de code, un artéfact spécifique peut être utilisé simultanément par de nombreux fichiers de code pendant l'exécution. Il est donc nécessaire de déterminer avec précision pendant le processus migratoire quel fichier de code doit contenir l'instruction. Dans *DVC*, le fichier principal '*dvc.yaml'* enregistre les nœuds de pipeline et utilise les liens entre les artéfacts du nœud, représentés sous forme de dépendances ('*deps'*) ou de sorties ('*outs'*), pour déterminer automatiquement les étapes à exécuter après toute modification qui a affecté ses artéfacts associés.

Le défi est encore une fois lié au manque de documentation explicite des entités d'AM (phases et artéfacts) et à la séparation claire des autres entités pouvant exister dans le dépôt du projet (code, configuration, données). Même avec une bonne connaissance du projet, le processus de récupération de valeurs et de paramètres spécifiques peut être fastidieux et difficile, variant en fonction de la taille du projet.

4.5.1.4 Option AN2 : Reconstruire les liens de traçabilité entre les artéfacts

Lors de la migration, nous avons enregistré des fichiers d'entrée et de sortie référencés dans le code de chaque phase d'AM. Par exemple, supposons que la phase d'entraînement d'un projet, telle que spécifié dans la documentation, est capturée dans la ligne de commande suivante : `'python -m train.py'`. Le *fichier de base* '*train.py'* utilise deux fichiers d'entrée '*train-dataset1.csv'* et '*train-dataset2.csv'* pour entraîner un modèle et l'enregistre dans le fichier de sortie '*model.pkl'*. Cela établit des liens traçables entre les différents artéfacts, c'est-à-dire le code, les ensembles de données d'entrée et le modèle de sortie. Cependant, la récupération de liens peut ne pas toujours être simple car les liens sont cachés dans les dépendances importées. Un outil capable de récupérer des graphiques de dépendances entre modules python pourrait être utile pour récupérer ces liens.

> **Recommandation 1 (phase d'analyse):** la documentation est d'une importance cruciale. Dans un dépôt logiciel générique, il est vital de documenter explicitement les composants et artéfacts spécifiques à la nature du projet, soit ceux d'AM dans notre cas. Cela peut aider une équipe dédiée à terminer la migration vers une plateforme MLOps, même si elle ne possède pas une connaissance complète du projet.

4.5.2 Phase 2 : configuration des environnements

Cette étape consiste à configurer l'environnement de développement et celui du pipeline d'AM tel que recommandé par chaque plateforme, à savoir *DVC* et *MLflow*. L'environnement spécifie les dépendances du projet et de la plateforme. La configuration et la gestion des environnements de développement et de test dans les applications d'AM peuvent être difficiles [108, 109]. Les applications d'AM ont souvent des exigences matérielles spécifiques [110], ainsi que des

bibliothèques au niveau système qui sont spécifiques à chaque phase de développement d'AM [111].

4.5.2.1 Défi EC1 : Configuration et gestion des exigences matérielles

La configuration et la gestion des exigences matérielles telles que le GPU est une tâche difficile, à la fois dans des environnements physiques et conteneurisés. Sans une définition claire et centrale des exigences matérielles pour chaque étape du processus d'AM ainsi que des outils supplémentaires pour intégrer les ressources matérielles informatiques intensives dans des environnements conteneurisés, toute cette configuration et sa maintenance doivent être effectuées manuellement. Lors de la migration, les premiers tests que nous avons effectués étaient sur une machine qui n'avait pas de GPU. Nous avons rencontré plusieurs erreurs lors de l'exécution, où des projets qui n'avaient pas d'exigences spécifiques en matière de GPU dans leurs descriptions ont généré des erreurs relatives au GPU. Cela nous a obligé à changer l'environnement d'origine pour une machine avec GPU. Même avec le nouvel environnement, le projet *brain-research/realistic-ssl-evaluation* n'a pas pu être exécuté en raison d'exigences de version de GPU personnalisées et de bibliothèques de gestion de GPU obsolètes.

Un défi connexe consiste à configurer ces exigences matérielles dans des environnements conteneurisés. Le développement d'AM est grandement facilité par les conteneurs qui permettent des processus d'entraînement et tests continus. Par exemple, *MLflow* offre une intégration avec les conteneurs *Docker* et *kubernetes*[29]. Cependant, étant donné les exigences matérielles spécifiques de ces applications, la virtualisation du matériel des machines hôtes (en particulier le GPU) se révèle être une tâche complexe [112]. Des outils spécifiques existent pour cette tâche, mais ils doivent être ajoutés à la pile de l'environnement de migration et gérés individuellement, ce qui augmente la charge de travail nécessaire.

[29] Il s'agit de deux systèmes de virtualisation par containers

4.5.2.2 Option EC1: Appliquer la gestion des dépendances logicielles aux exigences matérielles

Nous avons appliqué la même procédure de gestion des dépendances aux exigences matérielles. Pour chaque projet, nous avons dressé une liste détaillée des exigences matérielles et de leur version (par exemple GPU, CPU, taille minimale de RAM). Cela nous a permis d'exploiter et parfois d'automatiser la configuration matérielle à la fois dans des environnements physiques et conteneurisés. Le fichier que nous avons utilisé était similaire à ceux utilisés par les outils d'*Infrastructure as Code*[30] (IaC) tels que *chef* [113] ou *Puppet* [114]. Ces outils pourraient être utilisés pour gérer les dépendances matérielles dans des environnements conteneurisés, mais nécessiteront davantage d'efforts de configuration et de maintenance, ainsi qu'une intégration avec les outils MLOps.

4.5.2.3 Défi EC2 : Gestion complexe des librairies et dépendances système

DVC ne prend pas en charge la gestion de configuration d'environnement, tandis que *MLflow* gère cette configuration au niveau du langage à la fois dans les environnements physiques et conteneurisés (virtuels). Cependant, ce dernier crée un pipeline de configuration d'environnement parallèle à celui projet initial. Ainsi, il double la quantité de travail nécessaire à la configuration de l'environnement (toutes les modifications apportées à l'environnement d'origine doivent être répliquées manuellement dans l'environnement cible). De plus, *MLflow* et *DVC* ne prennent pas en charge la gestion des dépendances au niveau du système, une méthode supplémentaire doit donc être implémentée pour cela.

Nous définissions les dépendances au niveau du langage comme étant l'ensemble des bibliothèques inhérentes au code principal (par exemple, directement importées et utilisées dans le code) et les dépendances au niveau du système, comme celles n'étant pas liées au code principal, mais qui sont nécessaires pour effectuer des tâches supplémentaires pendant l'exécution du flux d'AM. Par exemple, dans de nombreux projets, les données sont compressées (par exemple, '*hitvoice/DrQA'*) et parfois enregistrées dans un environnement cloud (par exemple, '*allenai/dqa-net'*). Ainsi, leur

[30] L'IaC consiste à gérer les ressources matérielles sous forme de code. Il tire parti de la conteneurisation et de la virtualisation afin de fournir un cadre de configuration automatisée des environnements.

flux d'AM a une phase de « téléchargement et extraction de données » qui nécessite un logiciel de compression/décompression (tel que '*tar'* ou '*gunzip'*) et des outils de téléchargement de fichiers (par exemple, '*wget'* ou '*curl'*).

Python gère les dépendances au niveau du langage via les fichiers d'exigences ('*requirements'*) ou les fichiers de configuration ('*setup.py'*). Cependant, nous avons constaté qu'une partie des dépendances requises est manquante dans le fichier '*requirements.txt'* de certains projets, ce qui entraîne une surcharge lors de la reproduction du projet, car il faut suivre une méthode d'essai et d'erreur pour trouver les dépendances manquantes et la version requise. Notre hypothèse de départ était que tous les projets auraient une liste complète des dépendances requises dans leur fichier '*requirements.txt'*. En y regardant de plus près, nous avons constaté que ce n'était pas toujours le cas, avec 7 projets sur 13 (54%) ayant au moins une dépendance non répertoriée, comme le montre la Figure 4.3.

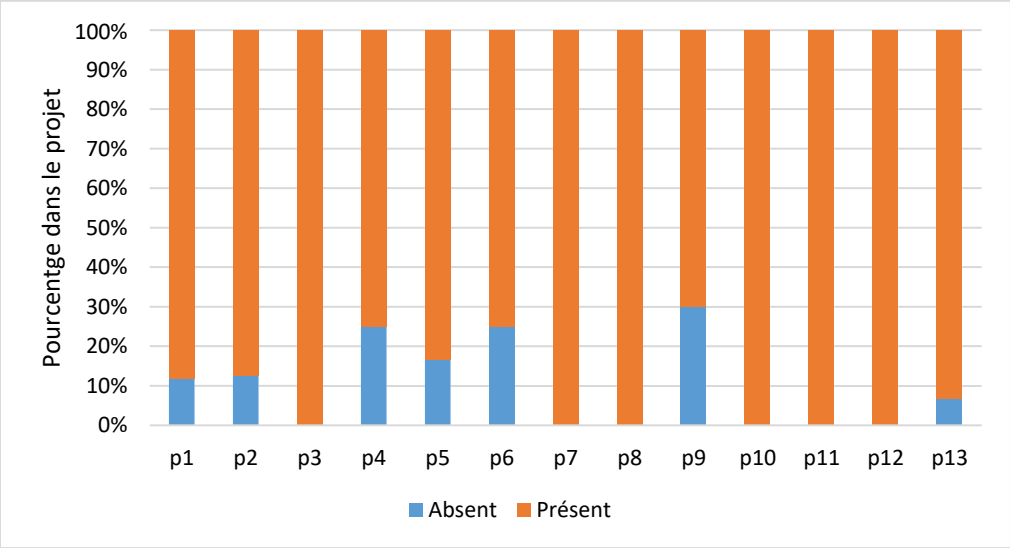


Figure 4.3 Distribution des librairies requises présentes ou absentes dans la description des exigences

Les exigences manquantes peuvent s'expliquer par le fait que les développeurs essaient souvent différentes bibliothèques lors du développement d'applications d'AM [92] afin de trouver celles qui conviennent le mieux à leurs tâches d'AM. Ce faisant, ils pourraient négliger de mettre à jour la liste des dépendances chaque fois qu'ils essaient une nouvelle bibliothèque, car certains essais peuvent être simplement temporaires avant le passage à une nouvelle ou le retour à l'ancienne.

D'un autre côté, *Python* ne gère pas les dépendances au niveau du système, car il s'agit de logiciels pouvant fonctionner de façon indépendante à *Python*. Compte tenu de cette limitation, *MLflow* limite les instructions qui peuvent être spécifiées dans les clés '*command*' des nœuds du pipeline aux instructions '*bash*'[31] et *Python*. Lors de la migration vers *MLflow*, toutes les instructions des phases du pipeline qui ne répondaient pas à ces exigences ont été encapsulées dans un fichier '*bash'*. Par exemple, le projet '*Baidi96/text2sql'* a une phase de préparation impliquant la décompression du jeu de données. La ligne de commande de la phase est '`tar -xjvf data.tar.bz2`'. L'outil '*tar'* est alors une exigence au niveau système dans ce projet, car il ne peut être géré comme une dépendance *Python*. Nous avons créé un fichier '*bash'* nommé '*step01_data-preparation.sh'* et changé le champ '*command*' du nœud correspondant à cette phase en '`bash step01_data-preparation.sh`'. *DVC,* d'autre part, n'a pas ces limitations et peut accepter n'importe quelle instruction de ligne de commande valide dans la clé '*cmd'* des nœuds du pipeline.

4.5.2.4 Option EC2 : Lien automatisé entre l'environnement initial et celui de la migration

Étant donné que l'environnement de migration est un doublon de l'environnement principal avec des dépendances supplémentaires liées à la plateforme MLOps, nous avons créé un script qui relie

> **Recommandation 2 (phase de configuration d'environnement)** : Les applications d'AM ont des dépendances supplémentaires à celles des logiciels traditionnels, principalement au niveau matériel et système. Ces dépendances devraient, tout comme les dépendances logicielles, être gérées via des fichiers et des outils de configuration spéciaux. De plus, afin de tirer parti de l'intégration continue dans les environnements conteneurisés, (1) des outils pouvant optimiser l'intégration de ressources matérielles spécialisées dans des conteneurs (comme le GPU) doivent être utilisés, et (2) les dépendances système doivent être définies et gérées comme celles du langage.

---

[31] La ligne de commande *bash* est un outil permettant d'interagir avec le système (principalement les systèmes *Linux*), en leur fournissant un ensemble d'instructions spécifiques rédigées par l'utilisateur. Les fichiers stockant de telles instructions utilisent généralement l'extension '*.sh'*.

l'environnement initial à l'environnement de migration. Ce script utilise le fichier '*requirements.txt*' et toutes les exigences de dépendance au niveau du matériel et du système ajoutées, puis met automatiquement à jour les fichiers de configuration de l'environnement cible chaque fois qu'il y a un changement dans les originaux. Cela a permis de réduire la quantité de travail manuel nécessaire pour maintenir l'environnement de migration.

4.5.3 Phase 3 : Configuration du pipeline d'AM

Les phases d'AM définies dans des pipelines sont configurés en (1) ajoutant des fichiers de configuration du pipeline et (2) en ajoutant des instructions ou des extraits de code au code d'origine pour permettre le suivi des artéfacts et des paramètres. La deuxième méthode est disponible uniquement pour *MLflow,* mais pas pour *DVC*. Bien qu'il existe des recommandations données par les fournisseurs de plateformes MLOps pour configurer ces pipelines, Barrak et al. ont constaté que ces recommandations sont rarement suivies par les développeurs [9].

MLflow et *DVC* utilisent des fichiers de configuration ('*MLproject'* et '*conda.yaml'*) pour *MLflow*, '*dvc.yaml'* et '*params.yaml'* pour *DVC*) lors de la configuration globale du pipeline. *DVC* peut traquer les artéfacts à l'aide de son fichier de configuration tandis que *MLflow* nécessite l'utilisation d'une API dédiée, ce qui implique de modifier le code principal (par exemple, l'instruction `'mlflow.log_artifact()'` doit être rajoutée au code pour tracer un artéfact particulier). Ce genre de méthode augmente la complexité, dans le sens où le code de développement et le code de traçage, qui sont deux activités distinctes, s'emmêlent dans les mêmes fichiers. Il s'agit d'une violation flagrante du principe de responsabilité unique SOLID[32] qui stipule qu'un module ne devrait avoir qu'une seule responsabilité à travers le projet [115]. Le problème peut également être lié aux notions d'enchevêtrement de code ('*code entanglement'*) comme décrit dans la programmation orientée aspect [116], où les '*crosscutting concerns'*[33] injectés directement dans le

[32] Les principes SOLID sont un ensemble de principes d'architecture logicielles permettant de rendre le code du logiciel plus compréhensible, flexible et maintenable

[33] En programmation orientée Aspect, les '*cross-cutting concerns'* sont des aspects du programme affectant d'autres '*concerns'* (Un '*concern'* étant une entité du programme apportant des fonctionnalités précises)

code principal sont connus pour causer des problèmes de compréhension et de maintenabilité du code source [117].

4.5.3.1 Défi PC1 : Modifier le projet pour qu'il corresponde aux spécifications de la plateforme

Dans nos expériences, nous avons constaté que des modifications importantes au projet étaient nécessaires pendant la migration, par exemple, l'ajout de fichiers de configuration de pipeline et/ou la modification du code source pour le suivi des artéfacts. Cependant, il existe un fort couplage entre les fichiers de pipeline MLOps et les artéfacts d'AM [9] qui augmente la complexité et les efforts requis pour maintenir ces éléments.

4.5.3.2 Option PC1 : Exploiter les données organisées identifiées à partir du projet d'AM

Nous avons utilisé le fichier de description JSON généré lors de la phase d'analyse (parties 4.5.1.2 et 4.5.1.4) pour faciliter et automatiser le processus de configuration du pipeline. Comme le fichier contenait déjà toutes les informations nécessaires (phases, artéfacts et leurs liens), les pipelines ont été complétés plus facilement. De plus, nous avons construit un script qui prend en entrée le fichier de description et crée automatiquement les fichiers de configuration du pipeline MLOps pour *MLflow* et *DVC*. Cependant, cette méthode ne fonctionne que pour les fichiers de pipeline, mais n'a pas pu être appliquée pour les configurations nécessitant un code personnalisé comme le suivi des artéfacts dans *MLflow*.

4.5.3.3 Défi PC2 : Configuration des flux multi-étapes

MLflow ne gère pas nativement les flux multi-étapes. Un code personnalisé doit être ajouté pour exécuter chaque étape déjà défini dans le fichier de pipeline (le fichier '*MLproject'*). D'autre part, *DVC* peut gérer des pipelines à plusieurs étapes sans aucune configuration personnalisée. Par conséquent, pour la migration *MLflow*, une étape importante consiste à configurer les pipelines multi-étapes. Cela consiste à ajouter du code personnalisé qui extraira chaque phase du fichier de pipeline et les exécutera de manière séquentielle pour reproduire le flux. Grâce à son API, *MLflow* fournit des fonctions pour manipuler chaque phase du pipeline avant, pendant et après l'exécution. Cependant, cela ajoute une surcharge de travail de deux manières :

- Premièrement, les développeurs qui n'ont pas beaucoup d'expérience avec *MLflow* auront du mal à identifier les fonctionnalités pertinentes à implémenter. Par exemple, lors de la construction du code multi-étapes personnalisé pour chaque projet, nous nous sommes concentrés uniquement sur la principale fonctionnalité attendue : exécuter chaque phase du workflow de manière séquentielle. Cela a conduit à des erreurs lors de l'exécution car nous avons manqué la gestion des paramètres (c'est-à-dire l'intégration des paramètres définis dans le pipeline à leur étape correspondante), la transmission des résultats précédents aux phases suivantes et la gestion des exécutions existantes ou en double, entre autres;
- Deuxièmement, l'effort nécessaire pour maintenir ce code est non négligeable, car il contient le code principal qui exécutera l'ensemble du pipeline.

4.5.3.4 Option PC2 : Utiliser les informations d'autres plates-formes et produire un code générique pour l'exécution des pipelines multi-étapes

En tirant parti des informations obtenues lors de la migration de projets vers *DVC*, nous avons développé un code générique qui a aidé à automatiser l'exécution des flux multi-étapes dans MLflow. Nous avons appliqué le même raisonnement que *DVC* qui consiste à récupérer l'historique des exécutions, les nœuds et leurs artéfacts, puis à les exécuter de façon séquentielle, tel que présenté dans la Figure 4.4.

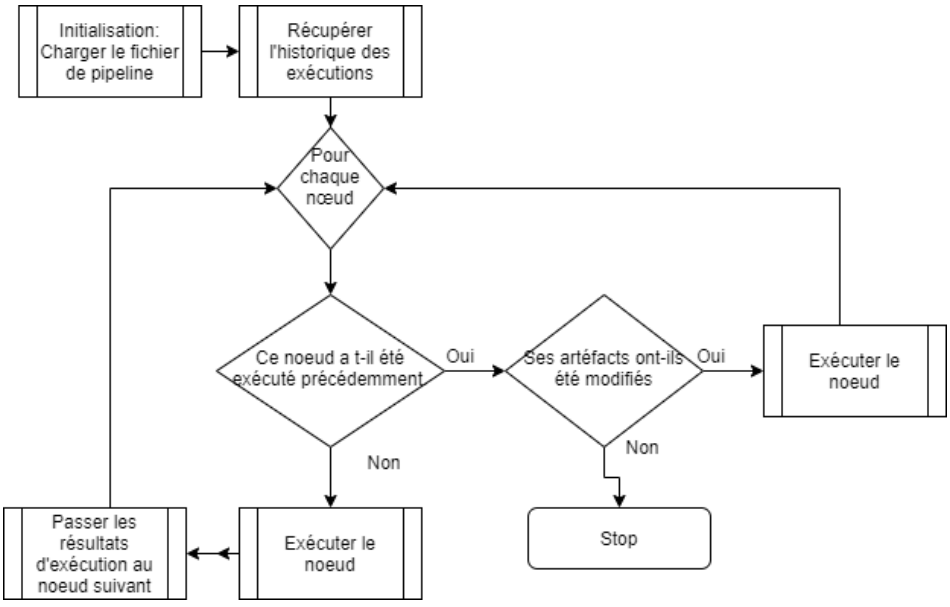


Figure 4.4 Diagramme d'exécution des pipelines multi-étapes

4.5.3.5 Défi PC3 : Impact de l'architecture du projet initial sur la migration

Il pourrait être nécessaire d'apporter des modifications critiques à l'architecture initiale du flux d'AM afin de tirer pleinement parti du potentiel des plates-formes MLOps.

En ce qui concerne l'architecture du projet, chaque plateforme a sa propre recommandation. Par exemple, *DVC* recommande que les métriques et les paramètres soient stockés dans des fichiers de configuration externes, et *MLflow* exige que les métriques soient des nombres identifiables individuellement dans le code. Les plateformes étudiées utilisent les nœuds du pipeline (appelées '*entry-points'* dans *MLflow* et '*stages'* dans *DVC*) pour gérer les étapes du flux d'AM. Ces nœuds sont exécutés séquentiellement par la plateforme selon le fichier de configuration établi. Cela nécessite que le flux d'AM du projet soit divisé en étapes, chaque étape ayant son propre nœud dans le pipeline. Nous avons constaté que ce n'est pas toujours le cas, certains projets mélangeant différentes étapes en un seul nœud. Par exemple, le projet '*spinningbytes/deep-MLsa'* combine toutes les étapes (préparation des données, formation, test et déploiement) en un seul nœud. Pour étudier la prévalence de cette pratique dans chaque projet, nous avons obtenu le nombre de nœuds p et le nombre d'étapes du flux d'AM s, puis avons calculé le nombre de nœuds par étape $n_{p/s} = e/s$ pour chaque projet. Les résultats sont présentés dans la Figure 4.5.

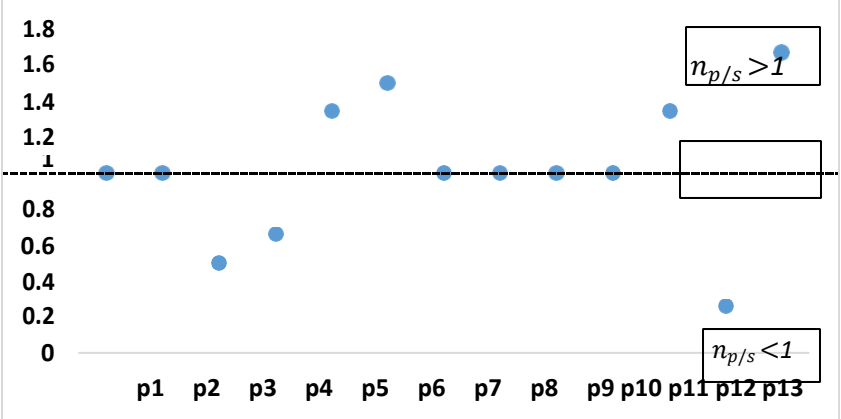


Figure 4.5 Distribution du nombre de nœuds par étapes

Lorsque $n_{p/s} = 1$, cela signifie que chaque étape du flux est associée à un nœud unique. Lorsque $n_{p/s} < 1$, cela signifie qu'il y a plus d'étapes que de nœuds configurés dans le pipeline : certaines étapes sont regroupées en un seul nœud. Par exemple, les étapes d'entraînement et de test sont exécutées en même temps dans un seul nœud du pipeline. Enfin, lorsque $n_{p/s} > 1$, cela signifie

qu'il y a plus de nœuds que d'étapes : certaines étapes sont divisées en plusieurs nœuds, permettant ainsi des sous-étapes. Par exemple, l'étape de prétraitement des données pourrait être divisée en deux nœuds liés au téléchargement et à la transformation des données. Le deuxième cas ($n_{p/s} < 1$) doit être évité car cela signifie que le projet n'est pas correctement divisé en étapes comme recommandés par les recherches récentes sur le sujet [9, 118].

L'architecture de certains projets permet l'isolation de certaines étapes via des arguments de ligne de commande (arguments de ligne de commande spéciales pour déclencher l'entraînement ou le test par exemple). Le projet '*spinningbytes/deep-MLsa'* a deux nœuds (entraînement et test) qui sont déclenchés par la même commande `'python runner.py -c [entraînement ou test]'`. La valeur de l'argument `'-c'` détermine laquelle des étapes d'entraînement ou de test doit être exécutée. Étant donné que *DVC* est capable d'identifier les modifications qui ont affecté les artéfacts d'un nœud spécifique et de réexécuter uniquement ce nœud, cette stratégie n'aide pas à tirer pleinement parti de ses possibilités.

Par exemple, considérons l'entraînement et les tests comme deux étapes différentes. Dans la configuration du pipeline, il devrait y avoir deux nœuds représentant chaque étape. Supposons que ce ne soit pas le cas et que l'entraînement est déclenché avec `'python run.py -step = train'` et le test est déclenché avec `'python run.py -step = test'`. Les deux étapes ont leur propre nœud qui peut être clairement spécifié dans le pipeline, mais ils ont le même fichier de base ('*run.py'*). Toute modification à ce fichier provoquera une réexécution des deux nœuds, même si la modification n'a affecté qu'une petite partie du code lié seulement à l'étape de test (ou d'entraînement).

Nous avons également compilé des statistiques (nombre d'étapes par fichier de base de nœud) sur la prévalence de cette pratique (comment les projets regroupent les étapes d'AM dans un seul fichier). Les résultats sont présentés dans la Figure 4.6 où f est le nombre total de fichiers de base uniques dans le projet, et p est le nombre total de nœuds. $n_{f/p} = f/p$ représente le nombre moyen de fichiers de base par nœud. Les implications sont les mêmes que le nombre d'étapes par nœud. Lorsque $n_{f/p} = 1$, cela signifie que le projet est optimisé pour les exécutions basées sur des modifications, tandis que $n_{f/p} < 1$signifie qu'il ne l'est pas et que les fichiers de base doivent être refactorisés et séparés en fonction du nombre d'étapes qu'ils exécutent. Par exemple, si '*run.py'* est

le fichier de base pour l'entraînement et le test du modèle, il faudrait le diviser en deux fichiers qui géreront séparément la logique d'entraînement et celle de test.

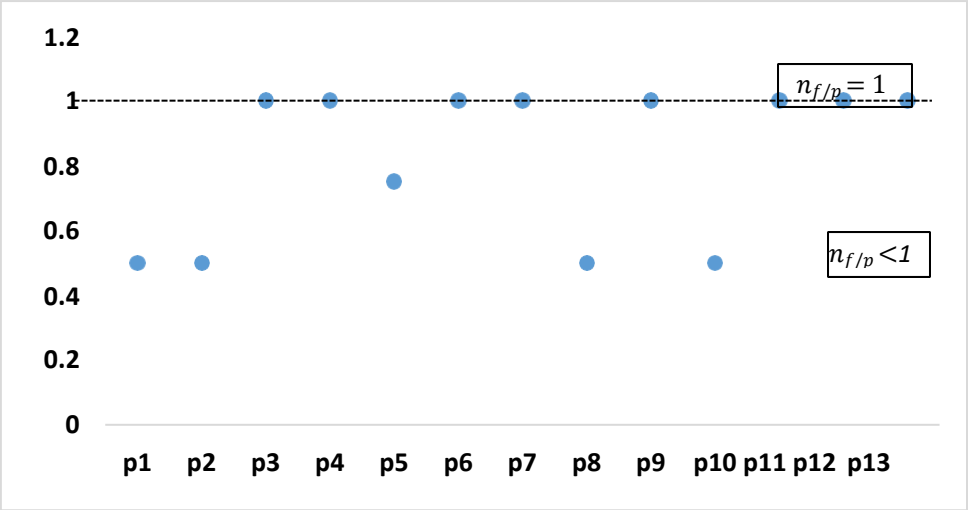


Figure 4.6 Distribution du nombre de nœuds par fichiers de base

4.5.3.6 Option PC3-1: Considérer l'architecture initiale du projet lors du choix d'une plateforme MLOps

Nous avons constaté que certains projets étaient mieux adaptés à des plates-formes spécifiques. *DVC* est plus adapté aux projets qui utilisent des fichiers externes pour stocker les paramètres et les métriques du projet. Les projets qui ne suivent pas cette méthode peuvent être plus adaptés à *MLflow*. Cependant, les deux plates-formes s'appuient sur un modèle paires clé-valeur pour stocker les paramètres et les métriques. Ils doivent donc être clairement identifiables dans le projet (les métriques et paramètres doivent être stockés dans leur propre variable ou avoir leur propre clé associée). Nous avons trouvé des projets (par exemple, '*manideep2510/eye-in-the-sky'*) qui ne stockaient pas de métriques dans leurs propres variables, mais à la place les métriques étaient calculées pendant l'exécution par une fonction spécifique sous forme de matrice de confusion et enregistrées soit vers la sortie standard, soit dans les fichiers de journalisation (*logs*). Nous avons dû modifier cette fonction pour renvoyer les valeurs des métriques individuellement.

4.5.3.7 Option PC3-2: Décomposer le pipeline d'AM en phases

Diviser le projet en phases plus petites est plus efficace que de garder toutes les étapes en une seule grande phase, et ceci pour de nombreuses raisons. Premièrement, il permet d'isoler chaque étape, permettant des modifications ciblées et une meilleure gestion du flux d'AM. Deuxièmement, cela aide à tirer parti des exécutions basées sur des modifications. Comme *DVC* peut suivre les étapes avec de nouvelles modifications et n'exécuter que celles-ci, le regroupement de toutes les étapes dans un seul fichier de code entraînera toujours l'exécution de l'ensemble du pipeline après chaque modification, même si cela ne concernait qu'une petite partie du flux d'AM. En outre, les projets qui visent à migrer vers ces plates-formes doivent éviter d'utiliser la programmation lettrée telle que *Jupyter Notebook* pour stocker leur flux de travail d'AM, car toutes les cellules sont traitées comme un seul fichier.

> **Recommandation 3 (phase de configuration du pipeline)** : L'architecture du projet est liée à la plateforme MLLC et elle impactera la migration vers la plateforme. Les développeurs doivent considérer les caractéristiques de leur architecture avant de choisir la plateforme cible. Ils doivent également suivre une approche modulaire pour leur flux de travail d'AM et sa mise en œuvre doit également refléter cette modularité

4.5.4 Phase 4 : Exécution (manuelle et automatique)

L'exécution se fait en exécutant la commande qui lance le pipeline. En utilisant *MLflow*, la commande est `'mlflow run <uri>'`, où `'<uri>'` est le chemin vers la racine du projet. Pour *DVC*, la commande est `'dvc repro'` qui doit également être exécutée à partir de la racine du projet. La commande d'exécution est lancée manuellement après chaque modification (ou groupe de modifications) du projet affectant tout code ou artéfact.

4.5.4.1 Défi EA1: Absence et chevauchement des fonctionnalités CI/CD

L'automatisation MLOps consiste à exécuter automatiquement chaque étape impliquée dans le pipeline d'AM avec le moins d'intervention humaine nécessaire. Nous avons utilisé des actions GitHub (*GitHub Actions*) pour configurer l'intégration et le déploiement continu (CI/CD) pour les projets. Un pipeline d'automatisation typique à l'AM est illustré à la Figure 4.7. Chaque fois qu'un commit est poussé en amont, un pipeline CI/CD se déclenche et va configurer un nouvel

environnement, exécuter toutes les étapes du workflow d'AM tel que défini dans le pipeline, enregistrer les résultats et éventuellement supprimer l'environnement créé. Une particularité des projets d'AM est que nous pouvons différencier deux pipelines lors de la configuration de l'automatisation. Le premier pipeline est le workflow d'AM et le second est le pipeline CI/CD qui gère la préparation de l'environnement et inclut le flux d'AM.

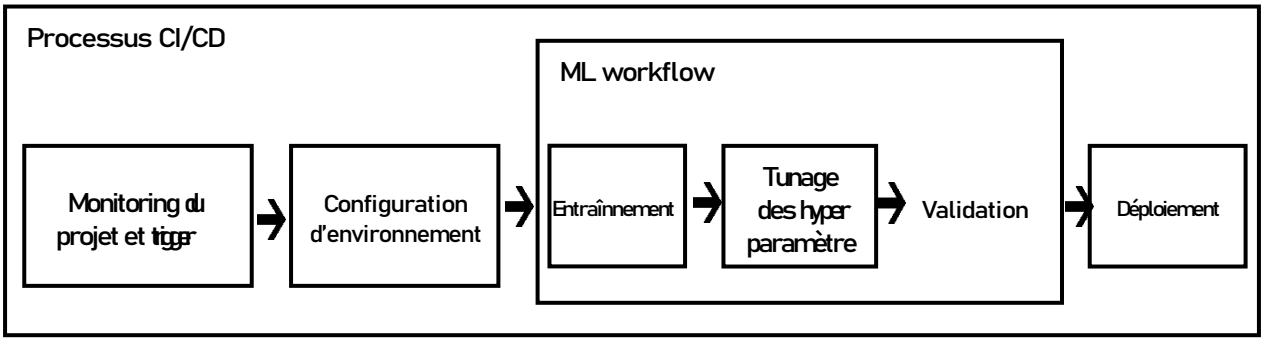


Figure 4.7 Processus CI/CD typique pour les applications d'AM

Dans notre étude, nous avons constaté qu'aucune des plates-formes n'effectue un suivi des modifications de code pour déclencher automatiquement le pipeline CI/CD, et qu'un outil externe doit être utilisé. Les outils CI/CD tels que *Github Actions* ou *Travis CI* créent généralement

> **Recommandation 4 (phase d'exécution/automatisation)** : Le chevauchement des pipelines d'AM et de CI/CD peut provoquer des confusions. Les outils existants peuvent offrir des environnements hybrides CI/AM/CD, spécialement conçus pour les applications d'AM, comme *CML*, qui intègre *DVC* avec *GitHub*.

l'environnement de base comme première étape avant d'exécuter les étapes suivantes du processus CI/CD. Cependant, comme vu dans la section 4.5.2, *MLflow* prend en charge la configuration d'environnement, ce qui provoque un conflit. Le même conflit se produit en déploiement continu. L'utilisation d'environnements hybrides comme *CML*[34] et *Github Actions* peut aider à mitiger ce conflit.

[34] [119] "CML | Data Version Control · DVC." https://dvc.org/doc/cml (accessed July 5, 2021).

4.5.5 Phase 5 : Validation

4.5.5.1 Défi VE1 : Validation complexe du résultat des exécutions.

Les processus CI/CD des logiciels traditionnels renvoient généralement un résultat booléen en fonction de la réussite ou de l'échec d'une exécution (ou '*build'* en anglais). Cependant, les processus d'AM ne consistent pas qu'en exécutions ou en déploiement, mais la qualité du modèle doit également être validée. Les résultats d'exécution d'AM comprennent la construction de plusieurs autres métriques, des matrices de confusion complexes et potentiellement des visualisations. Il est donc plus difficile d'implémenter des systèmes de validation automatiques.

4.5.5.2 Option VE1 : Exploiter les balises et les commentaires manuels et calculés pour suivre et enregistrer les résultats des analyses

MLflow et *DVC* offrent différents moyens d'ajouter des balises, des descriptions et des commentaires aux analyses. Ceux-ci peuvent être utilisés pour valider les résultats en calculant et en associant à chaque exécution des paires clé-valeur ou encore des données textuelles. Par

> **Recommandation 5 (phase de validation)** : Lors de la migration vers une plateforme MLOps, il convient de s'assurer de tirer parti de toutes les fonctionnalités disponibles. La configuration de la vérification automatique dans l'outil peut réduire l'effort manuel et la complexité des activités de vérification et validation du modèle.

exemple, lors de l'exécution du projet '*Baidi96/text2sql'*, nous avons défini des seuils arbitraires pour trois métriques : *'dev_acc_q', 'train_acc_qm*' et *'best_val_acc'*. À l'aide de ces seuils, les balises booléennes suivantes ont été automatiquement attribuées aux modèles après chaque exécution pour aider à valider les résultats : '*overfit'* (indique si le modèle est surajusté aux données) et *'to_register'* (indique si le modèle peut être enregistré dans le magasin de modèles ou non). De cette façon, les résultats sont vérifiés automatiquement sur la base des seuils définis et inclus dans les balises, réduisant ainsi l'effort manuel requis pour la validation.

4.5.6 Phase 6 : Maintenance

4.5.6.1 Défi MA1 : Évolution et versioning des artéfacts liés à la plateforme (artéfacts MLOps)

En ce qui concerne la gestion des fichiers de pipeline *MLOps*, Barrak et al. [9] ont découvert que (1) les fichiers de configuration du pipeline ont une complexité non linéaire qui peut ajouter une surcharge de gestion et (2) il existe un fort couplage entre les fichiers de configuration de pipeline et les artéfacts d'AM. Compte tenu de cela, il est difficile d'identifier quelle modification d'AM pourrait entraîner des changements dans les fichiers de configuration du pipeline. Par exemple, considérons le projet '*yochaiz/darts-UNIQ'* qui a une phase d'entraînement exécutée par le code `'python train_search.py --data cifar100'`. La modification, par exemple du nom de l'argument `'--data'` en `'--dataset'` dans le code nécessitera un changement correspondant dans le fichier de configuration du pipeline. La propagation des changements et leur gestion ultérieure peuvent devenir encore plus difficiles lorsque le nombre de contributeurs augmente.

4.5.6.2 Option MA1 : Établir des liens de traçabilité entre les artéfacts d'AM et les artéfacts MLOps

Construire des liens de traçabilité entre les artéfacts d'AM et les artéfacts MLOps implique de connecter des éléments du pipeline MLOps avec des instructions spécifiques dans le code. Selon la Figure 4.8, des bibliothèques particulières (parseurs) sont généralement utilisées pour gérer les instructions de ligne de commande. Ainsi, il est possible de créer des outils liés à la bibliothèque utilisée pour l'identification automatique des nœuds du pipeline en fonction du code. Par exemple, nous avons constaté que plus de 76,9% des projets étudiés utilisent '*argparse*' pour gérer les instructions et les arguments de la ligne de commande. Nous avons construit un outil capable de détecter ces arguments dans le code et leurs valeurs par défaut afin d'établir automatiquement des liens entre les commandes correspondantes et les fichiers de code. Les liens sont ensuite utilisés pour mettre à jour les configurations de pipeline lorsque le fichier de code correspondant est modifié.

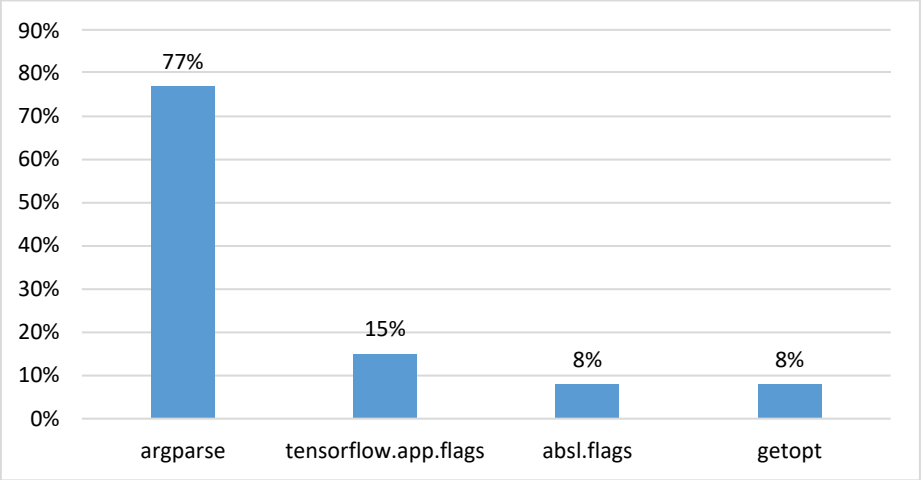


Figure 4.8 Proportion d'utilisation des parseurs dans les projets

4.5.6.3 Défi MA2 : Impossibilité de conserver l'historique avant la migration

MLflow et *DVC* enregistrent les commits associés aux exécutions du pipeline d'AM, et ils peuvent lier les modifications d'artéfacts à une exécution spécifique afin d'améliorer la traçabilité. Cependant, ils ne fournissent pas de fonctionnalités pour copier l'historique du projet avant sa migration. Cela peut impliquer la perte d'une partie importante de l'historique du projet, ce qui peut également avoir des conséquences sur la traçabilité, les fonctions CI/CD, aggraver certains des défis présentés ici et entraver certaines fonctionnalités de la plateforme MLOps.

4.5.6.4 Option MA2 : Adaptation et standardisation des données d'évolution entre les outils de versioning traditionnels et les plateformes MLOps

Après avoir réalisé que lors de la migration, l'historique du projet serait perdu, nous nous sommes assurés de conserver tous les éléments d'historique, c'est-à-dire, commits, qui étaient pertinents pour la plateforme MLOps cible. Pour ce faire, nous nous sommes appuyés sur les liens établis

> **Recommandation 6 (phase de maintenance)**: Adapter et aligner l'historique du projet du point de vue de l'AM et du logiciel est nécessaire pour conserver l'historique du projet dans les plateformes MLOps. Des outils automatisés ont été proposés pour récupérer l'évolution d'AM à partir de son historique en tirant parti des commits et des liens de traçabilité entre les artéfacts.

entre les artéfacts de pipeline, par exemple les modèles, et les artéfacts de code respectifs. Ensuite, tous les commits affectant les artéfacts de code ont été adaptés à la plateforme MLOps par identification de commits reliés à des exécutions du code. Un outil permettant l'automatisation de cette activité dans les plateformes traditionnelles de gestions de version a par ailleurs été proposé dans la partie 5.4 [91].

4.6 Obstacles à la validité

- **Taille des données** : Comme la majeure partie de l'étude impliquait une analyse manuelle, le nombre de projets étudiés est par nécessité relativement limité. Cela pourrait impacter la confiance dans les résultats de l'étude, étant donné que les conclusions pourraient être liées à des projets spécifiques et ne pas représenter exactement la réalité. Bien que 13 soit trop peu pour avoir un échantillon représentatif, la conception expérimentale demande beaucoup d'efforts, et il aurait été plus difficile d'analyser davantage de projets. Cependant, cela a surtout un impact sur l'analyse statistique. Jones et al. [120] ont montré que les logiciels du même domaine ont tendance à être construits de la même manière par les développeurs. Étant donné que les défis identifiés au cours du processus expérimental sont liés aux pratiques de développement, nous pensons que cela peut aider d'autres développeurs d'AM qui adopteront des pratiques similaires à celles identifiées dans cette étude. Dans l'ensemble, l'objectif de l'étude est d'identifier et de discuter de ces défis, plutôt que de viser la signification statistique de leur prévalence.

- **Validité externe** : Le corpus de projets analysé contient principalement des projets de recherche. Ainsi, les conclusions de cette étude pourraient être plus pertinentes pour ce type de projets. D'autres analyses devraient reproduire l'étude actuelle sur des projets d'AM en entreprise. Aussi, ce travail pourrait être étendu à d'autres technologies MLOps comme *Pachyderm* ou *Kubeflow*.

- **Subjectivité : p**uisqu'il n'existe pas de norme pour l'organisation et la gestion des versions des artéfacts d'AM, la phase d'analyse du processus de migration peut être biaisée par l'identification subjective des étapes et/ou des artéfacts d'AM. Tous les projets ne décrivaient pas clairement les phases, et dans ces cas, les auteurs devaient étudier manuellement le code et déduire les phases d'AM. Ainsi, ces éléments pourraient être liés à notre compréhension du code, au lieu de l'implémentation réelle que l'auteur original aurait pu avoir en tête. Au lieu de poser une limite

à l'étude, nous pensons que cela met l'accent sur le premier défi (identification du flux et des artéfacts d'AM). D'autres études devraient reproduire cette analyse avec une implication directe des auteurs originaux des projets d'AM, afin d'identifier plus précisément les étapes et les artéfacts.

4.7 Travaux connexes

Les outils de gestion du cycle de vie de l'Apprentissage Machine sont à un stade précoce d'adoption [9]. Malgré le nombre croissant d'outils open source et privés possédant des fonctionnalités plus ou moins étendues pour gérer les différents éléments impliqués dans le développement d'AM, les activités et les efforts nécessaires pour adopter ces outils n'ont pas été largement examinés. Barrak et al. [9] ont étudié la prévalence des pipelines MLOps dans les projets *DVC* et ont constaté que (1) les développeurs ne suivent pas souvent les meilleures pratiques recommandées par la plateforme, (2) il existe un fort couplage entre les artéfacts *DVC* et les artéfacts d'AM, et (3) une tendance à la hausse (main non constante) de la complexité du pipeline *DVC*. Ces résultats soulignent l'importance de notre propre étude, alors que de plus en plus de développeurs adoptent les plateformes MLOps.

Renggli et al. [121] explorent et proposent des solutions aux défis liés à l'automatisation du développement de l'AM et aux pratiques MLOps. Les défis identifiés sont principalement liés aux activités d'AM, mais certains d'entre eux sont similaires à ceux que nous avons trouvés, tels que la validation/sélection de modèles et le concept de '*drift'* pouvant résulter d'une gestion insuffisante des données. Dans la même lignée de travaux, Grandlund et al. [118] ont trouvé des défis similaires à ceux observés dans notre étude. En étudiant les pipelines MLOps dans différentes organisations, les auteurs ont souligné la nécessité de (a) diviser le pipeline d'AM en plusieurs parties, (b) standardiser le format des données et les descriptions dans toute l'entreprise, et (c) accorder une attention particulière aux artéfacts d'AM et à leur évolution.

4.8 Conclusion

L'adoption d'outils de gestion du cycle de vie d'AM est une pratique précoce mais croissante parmi les développeurs. Cette étude explore les principaux défis pouvant être rencontrés lors de l'adoption de ces outils, en particulier lors de la migration d'un projet existant vers ces plateformes. En spécifiant une méthodologie générique d'adoption/migration, nous avons mené une analyse expérimentale sur 13 projets pour les intégrer dans les plateformes *MLflow* et *DVC*. Nous avons

ensuite identifié les principaux défis auxquels nous avons été confrontés au cours de ces activités et signalé les solutions que nous avons appliquées pour les résoudre, lorsque cela fut possible. Nous avons constaté qu'en raison des caractéristiques spécifiques des applications d'AM, il existe un besoin de normalisation et de spécification plus rigoureuse du flux et des artéfacts d'AM. En fin de compte, nous avons résumé notre expérience et nos conclusions dans des recommandations aux développeurs et aux experts d'AM pour éviter et/ou réduire l'impact de ces défis.

CHAPITRE 5 RECONSTRUCTION DE LA TRAÇABILITÉ DES ARTÉFACTS ET PIPELINES D'APPRENTISSAGE MACHINE

Une partie de ce chapitre a été présentée à la conférence IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) 2021 sous le nom : MSR4ML: Reconstructing Artifact Traceability in Machine Learning Repositories

5.1 Introduction

Tel que démontré dans les chapitres précédents, les modèles d'AM sont étroitement liés à leurs données, ce qui implique un processus de développement différent des autres types de logiciels. Les projets logiciels traditionnels reposent souvent sur des plates-formes de contrôle de version, telles que *GitHub*, mais ces plates-formes n'ont pas encore été étendues pour prendre en charge les projets d'AM. La gestion des versions des données est insuffisante, et ils n'offrent pas de moyens d'observer et suivre les liens entre les artéfacts d'AM. Ainsi, la traçabilité et l'évolution du modèle peuvent devenir difficiles pour les développeurs. Bien que certaines plates-formes d'AM spécifiques existent, elles nécessitent toujours une spécification manuelle considérable du pipeline d'AM, de ses artefacts d'AM et des liens entre eux.

5.2 Objectifs

Sur la base de nos études précédentes dont les résultats sont présentés dans les chapitres 3 et 4, nous souhaitons proposer une méthodologie générique permettant de reconstruire les éléments impliqués dans le flux d'AM, à partir de son référentiel logiciel. Plus précisément, nous prévoyons :

- Proposer un processus modulaire de reconstruction automatique de la traçabilité des artéfacts d'AM à partir des dépôts logiciels, démontrer le processus et investiguer ses performances en termes de taux de détection, de flexibilité et d'adaptabilité;
- Introduire une technique d'extraction et de génération semi-automatique du pipeline d'AM à partir de son code source, et analyser ses performances en termes de capacité de détection et de modularité.

5.3 Approche

Nous souhaitons tirer parti des pratiques et défis relatifs au développement d'AM identifiés précédemment afin de proposer différents outils permettant de réduire la charge de travail relativement à la gestion des pipelines et artéfacts d'AM. Notre approche se base principalement sur l'analyse des dépôts logiciels, ainsi que sur les approximations exprimées dans les chapitres précédents[35]. Ainsi, les deux méthodes proposées exploitent l'analyse statique du code source, l'extraction des commits, la classification des artéfacts, afin de proposer un cadre générique[36] pour la gestion des projets d'AM.

5.4 MSR4ML : reconstruction de la traçabilité des artéfacts des dépôts d'AM

5.4.1 Principe

MSR4ML (Mining Software Repositories fora Machine Learning) est un cadriciel modulaire pour l'identification automatique et le suivi de l'utilisation des artéfacts dans les projets d'AM basés sur des systèmes de contrôle de version (en l'occurrence *Git*). Il explore le code et le dépôt d'un projet d'AM afin d'extraire des informations pertinentes sur l'utilisation des artéfacts et reconstruire les liens entre eux. La Figure 5.1 présente l'architecture générale du cadriciel qui se compose de 4 modules différents : un analyseur de code source, un identificateur de l'utilisation des artéfacts, un classificateur d'artéfacts et un traqueur de commit. Chaque module peut être étendu et personnalisé en tant que plug-in pour le cadriciel, tout en exposant une interface standardisée, de sorte que la communication entre les modules est indépendante de leur implémentation respective.

---

[35] Nous assumons entre autres que les systèmes de contrôle de version sont largement utilisés pour gérer les projets d'AM, d'où l'importance de les adapter à ceux-ci.

[36] La généricité des méthodes proposées est assez importante, du fait de la grande disparité pouvant exister entre les outils et les pratiques d'AM.

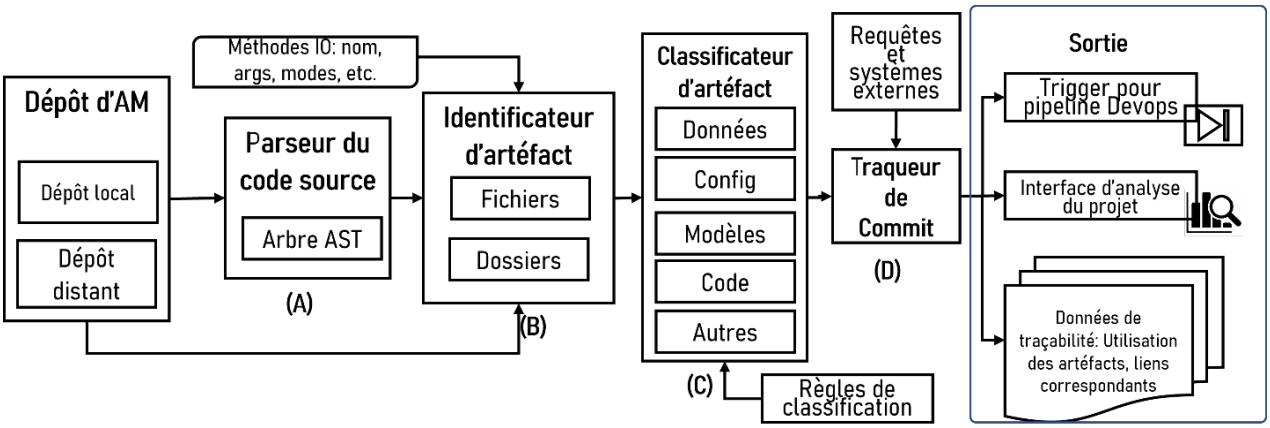


Figure 5.1 Architecture générale du cadriciel MSR4ML

Liens de traçabilité: le processus de traçabilité proposé tire parti des fonctionnalités de Git et des liens modèle-artefact pour suivre l'évolution d'un projet d'AM. Il peut être utilisé à de nombreuses fins telles que l'extraction de métadonnées, l'analyse du modèle, l'intégration continue, etc. Considérons l'exemple suivant de son application pour l'analyse de modèle. Un développeur pourrait se demander : « Quels changements ont entraîné une baisse des performances d'un modèle particulier M ? » La requête correspondante peut être adaptée ainsi : "Récupérer tous les commits affectant le modèle et ses artefacts entre la version du modèle à l'instant t et la précédente." Le cadriciel suivra les étapes suivantes pour obtenir les informations :

- En utilisant le nom de fichier du modèle, extraire l'heure exacte t_1 et t_2 représentant respectivement les deux derniers commits modifiant le modèle (t_2 étant le plus récent);
- Récupérer tous les artéfacts liés au modèle ;
- Extraire tous les commits modifiant ces artéfacts entre t_1 et t_2 ;
- Retourner ces commits, en les classant selon la priorité du lien entre l'artéfact et le modèle.

Le résultat est similaire à celui d'une commande '*git-blame*'[37], et nous pouvons ainsi examiner quelles modifications sont responsables d'un changement dans une métrique de qualité du modèle. Le résultat de cet exemple est présenté dans l'annexe E.

[37] Il s'agit d'une commande *Git* permettant d'analyser chaque modification ayant affecté un artéfact logiciel particulier, offrant ainsi la possibilité de suivre l'évolution de cet artéfact

Plusieurs composants architecturaux sont requis pour mettre en œuvre ce processus, tels que présentés dans la Figure 5.1.

5.4.1.1 Parseur du code source (A)

Le parseur est chargé d'analyser les fichiers de code et d'obtenir leur représentation AST. La sortie de l'analyseur est un nœud AST étendu représentant le contenu d'un fichier de code. L'AST permet de parcourir les nœuds étiquetés du code et de trouver des éléments spécifiques, notamment des invocations de méthodes, des déclarations et des accès aux variables, des chaînes littérales et autres. L'AST résultant doit être complet, de sorte qu'en cas d'ingénierie inverse, il reproduise le code original.

5.4.1.2 Identificateur de l'utilisation d'artéfacts (B)

L'identificateur d'artéfact traverse l'AST produit par le parseur et identifie toutes les méthodes ou fonctions qui interagissent avec les fichiers. L'hypothèse est que les méthodes interagissant avec les fichiers peuvent révéler des liens entre le code et les différents artéfacts. Les composants de ce module sont illustrés dans la Figure 5.2. Il utilise une liste de méthodes existantes (appelées *'méthodes IO'* dans l'architecture) et vérifie leurs invocations dans le code. Les méthodes IO sont des fonctions qui peuvent être utilisées pour interagir avec un fichier. Ils prennent généralement en entrée le nom du fichier et le mode (lecture, écriture, ajout). Par exemple, en python, la méthode par défaut utilisée pour gérer les fichiers est la fonction `'open(file, mode)'.` Si l'argument `'mode'` n'est pas défini, il passe par défaut au mode lecture.

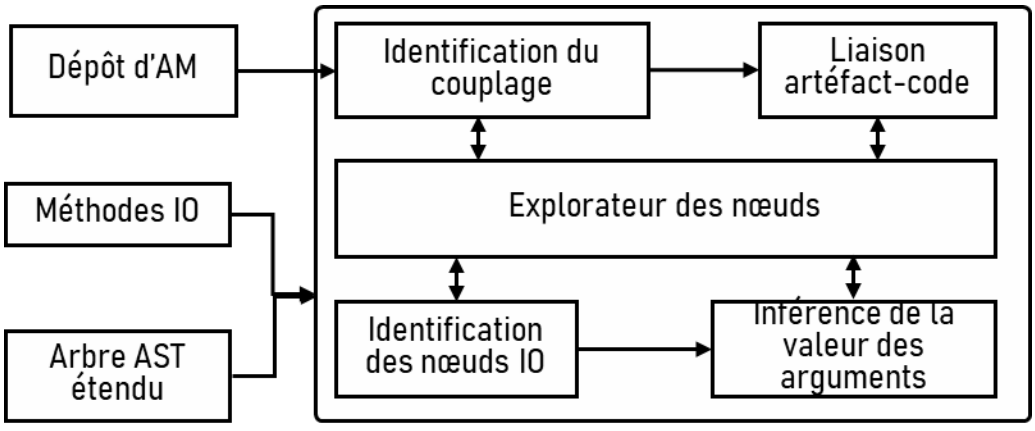


Figure 5.2 Architecture du module d'identification de l'utilisation des artéfacts

La liste des méthodes d'entrée/sortie (méthodes IO) est basée sur ce qui est spécifié dans les API des bibliothèques et des langages de programmation. Chaque langage fournit différentes méthodes pour interagir avec les fichiers. La création de la liste des méthodes IO implique un travail manuel préalable et intensif de lecture de la documentation des différentes bibliothèques. Une façon de résoudre ce problème est d'identifier les bibliothèques les plus utilisées dans les projets d'AM et d'étudier les méthodes IO qu'elles fournissent. Lors de la mise en œuvre du cadriciel, les résultats du Chapitre 3[38] ont été utilisées.

Le nom et le mode d'utilisation de l'artéfact sont extraits par le sous-module '*inférence de la valeur des arguments'*. Ce module prend en entrée le nœud AST d'une variable et tente de déterminer sa valeur en recherchant dans l'ensemble de l'AST. La construction de fonctions d'inférence n'est pas très difficile pour les variables dont les valeurs sont affectées de manière statique dans le code, en particulier pour les types de variables simples comme les chaînes de caractères ou les opérations binaires telles que les concaténations. Cependant, pour les types plus complexes (par exemple le chargement de variables à partir de fichiers de configuration), la création de fonctions d'inférence peut être difficile. Une solution consiste à étendre le module avec une analyse de code dynamique en exécutant le nœud pour obtenir la valeur de la variable [122]. Le Tableau 5.1 présente un exemple d'organisation des informations des méthodes IO.

Tableau 5.1 Exemple de représentation des méthodes IO

Clé	**Description**	**Exemple**
lib	Nom de la librairie	os
name	Nom de la méthode	open
fparam_name	Nom du paramètre stockant le fichier à charger	file
fparam_position	Position du paramètre stockant le fichier à charger	0
mparam_name	Nom du paramètre spécifiant le mode	flags
mparam_position	Position du paramètre spécifiant le mode	1
mparam_type	Type du paramètre spécifiant le mode	Object
mvalues	Valeurs possibles du paramètre mode et leur signification (en lecture ou en écriture)	{'os.O_RDONLY':'r', 'os.O_WRONLY':'w', 'os.O_RDWR':'rw', 'os.O_APPEND':'w'}
mdefault	Mode par défaut	r

[38] Voir partie 3.3, pratiques 1 et 2

5.4.1.3 Classificateur d'artéfacts (C)

Ce module est chargé de classer les artefacts en différentes catégories. Nous pouvons distinguer quatre artefacts principaux dans les projets d'AM: données, configuration, code et modèles [2, 50]. De plus, les paramètres, hyperparamètres et métriques peuvent être considérés comme des artefacts lorsqu'ils sont stockés dans un emplacement explicite, comme un fichier de configuration. La Figure 5.3 décrit les principaux composants de ce module.

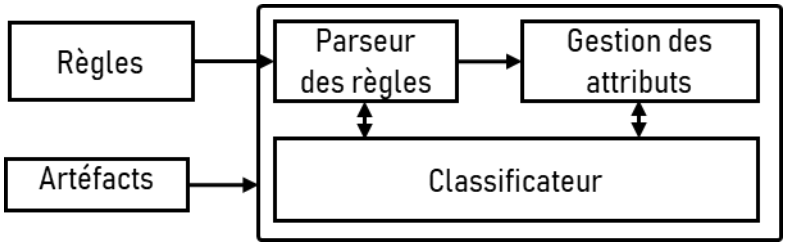


Figure 5.3 Architecture du module de classification

Le classificateur prend en compte des règles pour affecter efficacement un type à un artefact. Ces règles sont hautement personnalisables et hiérarchisées. La priorité peut être catégorique (élevée, moyenne, faible) ou continue (de 0 à 10 par exemple) et est utilisée pour atténuer tout conflit lorsque les artefacts sont affectés à plusieurs types. Par exemple, un artefact identifié comme 'données' et 'modèle' sera classé en tant que 'modèle' si la règle utilisée pour l'identifier comme 'modèle' a une priorité plus élevée que celle utilisée pour le classer comme 'données'.

Il est possible que certains artefacts référencés par le système ne soient pas présents dans le dépôt. Par exemple, les données peuvent ne pas être autorisées à résider dans un dépôt public pour des raisons de confidentialité. Par conséquent, avant l'exécution du processus de classification, le module vérifie si l'artefact récupéré se trouve dans le dépôt. Si ce n'est pas le cas, le module peut utiliser des règles spéciales (si définies) pour gérer les artefacts absents du dépôt. La liste des artefacts classifiés est stockée dans un fichier, dont le format peut être configuré par le développeur, mais le format par défaut est JSON. Le développeur peut en outre ajouter des informations personnalisées (attributs) sur les artefacts, telles que la version de l'artefact. Les attributs peuvent être ajoutés ou mis à jour à la suite de processus manuels ou automatisés, par exemple un pipeline DevOps.

Après avoir identifié le nom et le mode de l'artéfact, l'artéfact est lié au code où il a été chargé et un poids est attribué au lien. Le poids sert à estimer la criticité d'un lien, selon que le code modifie

ou non l'artefact. Cette étape est réalisée par le sous-module '*liaison artéfact-code*', et pourrait être optimisée en ajoutant un sous-module de détection de couplage.

Le sous-module '*identification du couplage*' peut grandement améliorer la précision des liens en les considérant à des niveaux plus profonds. En pratique, il identifie les liens transitifs entre les artéfacts, le fichier de code qui y accède et les autres fichiers de code couplés au fichier accesseur. Il peut également être utilisé pour évaluer le poids des liens en fournissant des informations détaillées sur les classes/méthodes accessibles entre les fichiers couplés, isolant ainsi la partie des fichiers couplés impliquée dans le lien. Par exemple, considérons un fichier de code F_1 qui charge un artefact A . Considérons également que F_1 est couplé avec un autre fichier de code F_2. A est lié à F_1 puisqu'il est chargé dans ce fichier, et il peut également être lié à F_2 avec un poids supérieur ou inférieur, selon différentes règles telles que le niveau de couplage entre F_1 et F_2 ou si F_2 exécute des opérations critiques qui peuvent affecter l'artefact et son utilisation tout au long du projet. Dans notre prototype, le poids est simplement déterminé en fonction de la distance entre l'artefact et les fichiers couplés. Le fichier de code qui accède directement à l'artefact a le poids le plus élevé. Dans l'architecture proposée, il est facile de remplacer cela par des calculs de poids plus complexes ou un système d'analyse de l'impact des changements [123].

5.4.1.4 Traqueur de commits (D)

Le traqueur de commits dont l'architecture est présentée dans la Figure 5.4 est responsable du suivi des commits associés à chaque artefact du projet, en interrogeant *Git* pour les données de commit pertinentes. Alors que la logique de base permet simplement l'interaction avec un dépôt *Git*, cela peut être étendu avec des plug-ins pour interagir avec d'autres plates-formes, comme *GitHub* ou *Bitbucket*, ou inclure d'autres bibliothèques tierces de contrôle de version.

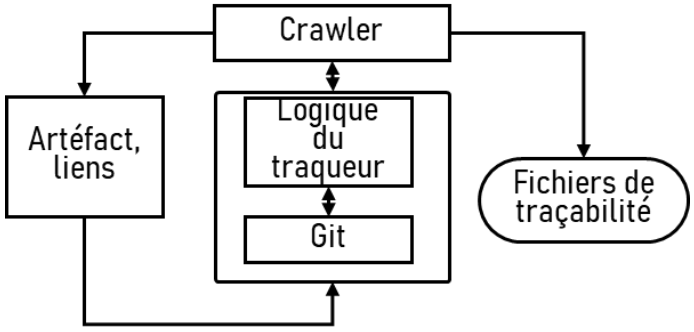


Figure 5.4 Composants du module de traçage des commits

Le traqueur implémente la logique principale de suivi et l'interface avec git. Cette logique peut être constituée de requêtes *Git* brutes, des fonctions personnalisées pour s'interfacer avec *Git*, une API pouvant interagir avec des dépôts distants ou des bibliothèques tierces. Le traqueur peut être utilisé de deux manières : en mode *pull* (*batch*) ou en mode *push* (*stream*), en fonction de la taille du dépôt et des exigences de performances pour l'analyse.

- **Pull**: Le traqueur n'est utilisé que lorsque le développeur (ou tout autre composant externe) a besoin de récupérer des données associées aux artefacts et à leur évolution. Ainsi, les données de validation résident dans le dépôt, sans avoir besoin d'une base de données externe pour l'analyse. Cependant, les arbres *Git* peuvent être volumineux, complexes et difficiles à analyser. Les performances du tracker sont directement liées à la taille du dépôt, ce qui peut créer des problèmes de performances.

- **Push :** Alternativement, un '*crawler*'[39] peut être ajouté pour surveiller les commits, extraire les informations pertinentes et les stocker dans un environnement séparé pour l'analyse. Le principal avantage de cette méthode est que les données de validation pertinentes deviennent facilement disponibles dans une base de données, permettant ainsi une analyse rapide. Ce robot d'exploration peut également activer des parties d'un pipeline DevOps. Par exemple, lorsqu'un commit qui affecte un artefact (données ou modèle) est détecté, l'outil peut déclencher des événements, comme l'entraînement ou le test du modèle, ou mettre à jour les attributs de l'artefact, comme leur version. Bien que cette méthode présente de nombreux avantages, elle nécessite un espace de stockage supplémentaire. Le dépôt pourrait servir à cela, mais la conséquence serait l'augmentation de sa taille et la diminution de sa cohésion. Lors de la mise en œuvre du mode push du tracker, il convient de faire attention aux événements cycliques. Par exemple, si le traqueur détecte un changement dans les hyperparamètres et déclenche un réentraînement du modèle, qui peut en retour affecter les hyperparamètres, cela peut éventuellement conduire à une boucle infinie. L'outil doit être configuré pour éviter ces situations.

---

[39] Un crawler est un programme s'exécutant automatiquement et de façon régulière afin d'explorer un univers (dans notre cas le dépôt *Git*) et identifier des éléments précis (dans notre cas, des commits)

5.4.2 Prototype

Comme preuve de concept, nous avons implémenté un prototype du cadriciel pour reconstruire la traçabilité dans les projets d'AM basés sur *Python*. Le prototype est également construit en *Python* et utilise des bibliothèques externes pour certains de ses modules.

La bibliothèque '*astroid' [124]* a été utilisée comme parseur de code. Il fournit des fonctions d'inférence de valeurs[40] des variables pour les primitives[41] *Python*, ainsi qu'une interface pour rajouter des fonctions d'inférence personnalisées. Pour l'identifiant d'artefact, nous avons utilisé une liste de méthodes IO présentes dans les bibliothèques les plus populaires utilisées dans les projets d'AM[42]. Nous avons étudié la documentation des 50 premières bibliothèques identifiées pour trouver toutes leurs méthodes IO (en plus des méthodes par défaut fournies par *Python*). Les différents modes d'interaction trouvés dans ces méthodes ont été classés en deux groupes : '*entrée'* ou '*sortie'*. Ils ont ensuite été organisés suivant une représentation JSON.

Pour le sous-module d'inférence, nous avons implémenté des fonctions d'inférence pour certaines méthodes de la bibliothèque '*os'* qui étaient souvent utilisées par les développeurs pour gérer les chemins de fichiers multiplateformes. Les méthodes les plus utilisées ont été déterminées en extrayant tous les nœuds AST des méthodes IO trouvées dans le code, ainsi que les types de leurs arguments. Concernant le classificateur d'artéfacts, nous avons utilisé des règles heuristiques statiques qui classent les artefacts en 4 catégories principales : données, modèle, code et configuration. Certaines de ces règles sont présentées dans l'annexe A.

Nous avons construit un script *Python* autour de *Git* en tant que traqueur de commit pour tester des exemples de questions de base que les développeurs pourraient avoir sur leur modèle, comme celle présentée à la section 5.4.1 (***Liens de traçabilité***).

---

[40] Ces fonctions permettent de déterminer de façon statique (sans exécution du code) la valeur d'une variable à partir du code

[41] Les primitives *Python* sont des variables et fonctions de base telles que les entiers, le chaines de caractères ou encore les fonctions d'addition, de soustraction, de concaténation, etc.

[42] Voir partie 3.3

5.4.3 Évaluation et résultats

Nous présentons une évaluation des modules d'identification et de classification des artefacts. En effet, ce sont ces modules qui déterminent l'extraction principale des artéfacts, et les autres modules sont orientés essentiellement vers l'analyse. Pour cela, nous avons sélectionné aléatoirement 20 dépôts à partir du corpus de données initial et avons vérifié manuellement la documentation et le code du projet pour identifier les artéfacts. Au cours de ce processus manuel, 5 des dépôts n'avaient pas de données ou d'artefacts de modèle (ils introduisaient plutôt un algorithme d'AM, mais ne construisaient pas de modèle) et 6 projets n'ont pas pu être analysés par le parseur en raison d'erreurs d'analyse[43]. Ces projets ont donc été retirés de l'ensemble, nous laissant un total de 9 dépôts. Les artefacts identifiés ont ensuite été comparés à ceux trouvés automatiquement par le module d'identification, et leur type a été comparé à celui trouvé par le module de classification. La précision, le rappel et le score F_1 ont été calculés pour cette comparaison comme décrit ci-après.

Considérons $A = \{(a_i, m_i)\}$ la population des artefacts trouvés, avec a_i étant un nom d'artefact et m_i son mode. Considérons A_1 = *artefacts trouvés par le processus manuel* et A_2 = *artefacts trouvés par l'outil*. Nous définissons n comme le nombre d'éléments d'une population. L'intersection des ensembles A_1 et A_2 ($A_1 \cap A_2$) représente les artefacts trouvés par les deux (Vrais positifs) et leur union ($A_1 \cup A_2$) tous les artefacts trouvés. Le nombre d'artefacts trouvés uniquement par le processus manuel et non par l'outil est de $n_1 = n(A_1) - n(A_1 \cap A_2)$ et le nombre d'artefacts trouvés uniquement par l'outil et non par le processus manuel (Faux positifs) est $n_2 = n(A_2) - n(A_1 \cap A_2)$. Ainsi, $précision = n(A_1 \cap A_2)/n(A_2)$, $rappel = n(A_1 \cap A_2)/n(A_1)$ et $F_1 = 2 \times \frac{precision \times recall}{precision + recall}$

La précision quantifie le nombre de prédictions justes par rapport au nombre total de prédictions.

Le rappel (recall) détermine le nombre de prédictions justes par rapport au nombre total d'artéfacts existants (vrais positifs)

Le score F_1 équilibre à la fois les préoccupations de précision et de rappel en une seule métrique.

---

[43] Le parseur utilisé ne peut analyser que des fichiers de code *Python3*, alors que ces projets exécutaient d'anciennes versions de python

Les résultats du module de classification sont présentés dans le Tableau 5.2.

Tableau 5.2 Résultats d'évaluation du module d'identification d'artéfacts

Projet	$n(A_1)$	$n(A_2)$	$n(A_1 \cap A_2)$	n_1	n_2	précision	rappel	F_1
p1	6	7	4	2	3	0.57	0.66	0.61
p2	8	2	1	7	1	0.5	0.125	0.2
p3	6	1	1	5	0	1	0.16	0.28
p4	2	1	1	1	0	1	0.5	0.66
p5	10	2	2	8	0	1	0.2	0.33
p6	2	2	2	0	0	1	1	1
p7	7	6	6	1	0	1	0.85	0.92
p8	4	6	4	0	2	0.66	1	0.8
p9	10	12	10	0	2	0.83	1	0.9
Moyenne						0.84	0.61	0.63

Comme le montre le Tableau 5.2, la précision et le rappel moyens sont respectivement de 0.84 et 0.61, ce qui signifie que le module est capable de trouver plus de 60% des artefacts identifiés manuellement. MSR4ML est un outil de recommandation, ce qui signifie que son objectif est de localiser les artefacts corrects, mais pas nécessairement tous les artéfacts. L'outil essaie d'éviter le « spamming » des développeurs avec des recommandations fausses. Lors du développement du module, nous avons observé que ces résultats sont étroitement liés aux fonctions d'inférence disponibles. Rétrospectivement, nous avons constaté que les fonctions d'inférence supplémentaires ont augmenté le rappel de près de 35%[44]. Un autre point intéressant pour une analyse future est que selon nos résultats, tous les faux positifs étaient des fichiers de sortie, principalement des fichiers temporaires et de journalisation (logs). Lors de l'évaluation manuelle, ils n'ont pas été considérés comme des artéfacts, car nous pensons qu'ils n'affectent pas vraiment le processus de développement d'AM, ni la qualité des modèles finaux. Cependant, nous reconnaissons que cette hypothèse peut être discutable. Un axe pertinent de recherche futur pourrait être d'interroger les développeurs d'AM afin d'étudier leurs points de vue par rapport à cette classe d'artefacts.

Pour le module de classification, nous avons déterminé un ensemble de règles basiques (récapitulées dans l'annexe A) utilisant le nom de l'artéfact, son extension et son mode d'intégration afin de le classifier comme données, modèle, configuration ou métriques.

Nous avons ensuite comparé le total des éléments trouvés automatiquement dans les projets avec ceux déterminés manuellement. Les résultats sont présentés dans le Tableau 5.3 Résultats

[44] Sur un premier test exécuté avec les fonctions d'inférence par défaut fournies par '*astroid'*, le rappel était de 0,45

préliminaires du module de classification. La colonne '*manuel'* regroupe les éléments identifiés manuellement, et la colonne '*Automatique'* contient ceux identifiés automatiquement.

Tableau 5.3 Résultats préliminaires du module de classification

	Manuel	Automatique	Correspondance	Précision	Rappel	F_1
Données	11	3	2	0.66	0.18	0.28
Configuration	5	1	1	1	0.2	0.33
Modèle	3	0	0	0	0	0
Métrique	2	1	1	1	0.5	0.2
Moyenne				0.66	0.2	0.2

L'on a obtenu des résultats mitigés (rappel et score F_1 faibles) qui, avec une analyse plus détaillée, coïncident avec les pratiques 5 et 6 observées dans le Chapitre 3:

- Premièrement, une partie des artéfacts identifiés n'étaient pas présents dans les dépôts. Cette pratique a été étudiée dans la partie 3.4 (Pratique 5) où l'on a observé que les développeurs ont tendance à ne pas stocker une partie des artéfacts (principalement les artéfacts de sortie) dans les dépôts. Ainsi, étant donné que le module de classification se base principalement sur les artéfacts présents dans le dépôt, une partie des artéfacts n'ont pu être classifiés ;

- Deuxièmement selon la pratique 6, il n'existe pas de convention générale adoptée par les développeurs pour l'organisation et le nommage des artéfacts d'AM. Ainsi, les résultats obtenus dénotent plutôt de l'inadéquation des règles utilisées plutôt que des résultats du module proprement dit. Nous pensons que les règles de classification devraient être déterminées par le développeur selon le standard d'organisation qu'il a choisi. Ainsi, l'importance du parseur de règles du module de classification prend tout son sens dans la mesure où il devrait proposer une interface configurable par le développeur qui pourra rajouter et modifier des règles de classification. Aussi, une méthodologie d'organisation et de nommage des artéfacts d'AM pourrait être intéressante comme axe de recherche.

Globalement, nous observons que le cadriciel ajoute de nouvelles informations concernant la traçabilité des projets d'AM. Selon le prototype présenté, le procédé fournit des résultats intéressants mais nécessite encore un certain nombre d'étapes manuelles (l'ajout de règles de classification et la spécification de fonctions d'inférence pour les variables les plus complexes), que nous avons déjà commencé à automatiser. Selon les métriques d'évaluation, les étapes

manuelles sont nécessaires pour récupérer 40% des artefacts restants. Étant donné que le rappel peut être amélioré en ajoutant plus de méthodes IO et d'inférence, nous avons regroupés les fonctions déjà implémentées dans un dépôt qui sera rendu public[45] et pourra être complété par la communauté (chaque développeur pourra partager ses propres règles et fonctions d'inférence qui pourront être réutilisées).

5.5 Méthode semi-automatique d'extraction et d'organisation des pipelines d'AM

5.5.1 Principe et fonctionnement

La méthode proposée combine une partie automatique et une partie manuelle, telle que présentée dans la Figure 5.5.

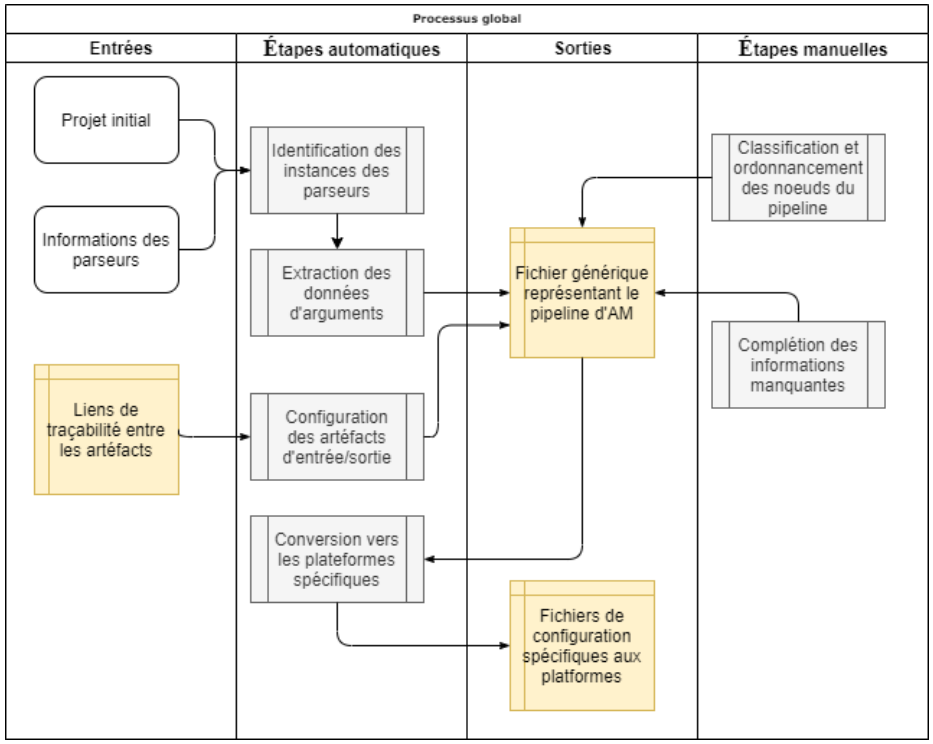


Figure 5.5 Processus général de reconstruction du pipeline d'AM

[45] Pour des questions de confidentialité, le dépôt n'était pas encore rendu public au moment de la rédaction du présent document

Dans l'ensemble, ce processus prend en entrée le projet initial d'AM ainsi que les données sur les parseurs (librairie, nom et paramètres de la fonction d'instanciation, etc.) puis effectue différentes étapes d'extraction des instances des parseurs et de leurs arguments pour produire un fichier générique de représentant le pipeline d'AM du projet. Ce fichier est ensuite complété par deux étapes manuelles (classification des nœuds et complétion des informations manquantes). Il peut ainsi être utilisé pour générer automatiquement les fichiers de configuration lors de la migration vers une plateforme d'AM. Un exemple de génération automatique du fichier de pipeline *MLproject* et de configuration d'environnement (*conda.yaml*) est présenté dans l'annexe D.

5.5.1.1 Données d'entrée

Le Tableau 5.4 récapitule les principales données d'entrée ainsi que leurs définitions.

Tableau 5.4 Données d'entrée requises par le processus

Donnée	**Description**	**Valeurs possibles**	**Exemple**
Projet initial	Chemin du dossier racine où est stocké le projet.	• Chemin local (absolu ou relatif) • Chemin distant (lien web par exemple)	./MyMLProject
Informations des parseurs	Données des parseurs selon l'implémentation du processus (librairie, nom de la fonction d'instanciation, paramètres, données retournées, etc.)	• Fichier ordonné (JSON, yaml, etc.) • Nœud AST • Objet dynamique • méthode	Pour la librairie '*argparse'* par exemple, l'on peut utiliser l'instance du parseur (objet dynamique), soit `argparse.ArgumentParser()`
Liens de traçabilité entre les artéfacts (facultatif)	Liens entre les artéfacts d'entrée, le code d'AM et les artéfacts de sortie (tels que produit par le cadriciel MSR4ML par exemple)	• Fichier ordonné (JSON, yaml, etc.) • Données textuelles classifiées (sous forme de liste, dictionnaire, base de données, etc.)	Voir Annexe B

5.5.1.2 Activités automatiques

Le cœur du processus est constitué d'un ensemble d'activités automatisées et implémentées suivant une logique séquentielle.

1. **Identification des instances du(des) parseurs et extraction des arguments**: A partir des données d'entrée, le processus identifie les instances des parseurs dans chaque fichier de code. Il extrait ainsi leur description, paramètres et valeurs par défaut. Pour chaque instance identifiée, un nœud est créé. Au terme du processus automatique, tous les nœuds sont regroupés afin de reconstituer le pipeline d'AM et le sauvegarder dans un fichier générique suivant le schéma représenté dans la Figure 5.6.

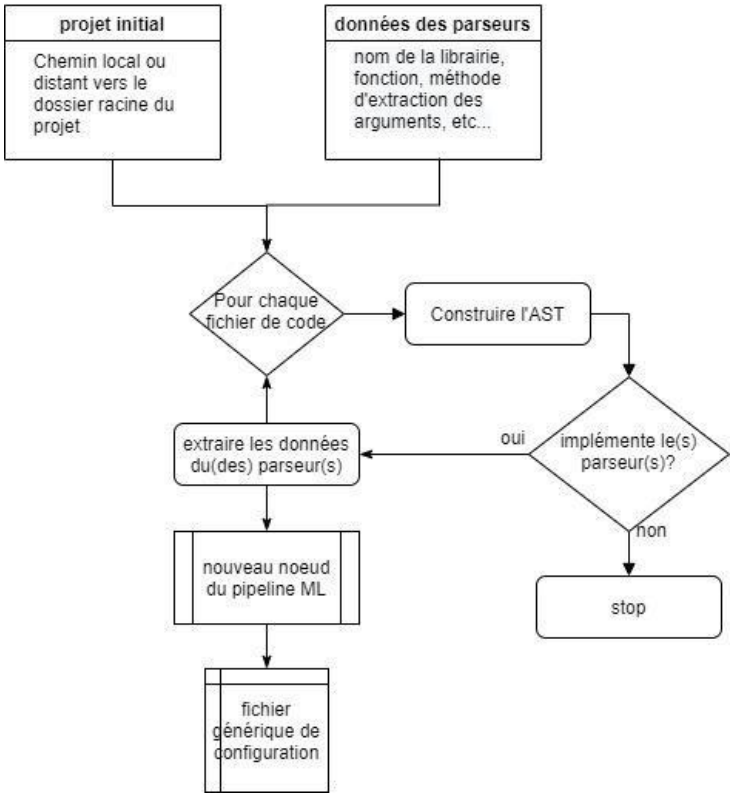


Figure 5.6 Schéma d'extraction du pipeline d'AM

2. **Configuration des artéfacts d'entrée/sortie :** A partir des liens de traçabilité fournis par le cadriciel MSR4ML, il est possible de configurer et organiser les artéfacts pour leur intégration au fichier générique de configuration du pipeline, où ils seront associés à des nœuds spécifiques du pipeline. Par exemple, dans DVC, les champs *'deps'* et *'outs'* de chaque nœud du pipeline doivent contenir les chemins vers les artéfacts liés à ce nœud, de telle sorte que chaque modification à l'un de ces artéfacts entraînera la réexécution de ce nœud. Cependant, les chemins ne doivent pas s'entrecouper, ce qui signifie qu'un dossier et un fichier de ce même dossier ne doivent pas être définis ensemble dans le même champ et pour le même nœud. Ainsi, la phase de

configuration permet de réorganiser les artéfacts sous un format standardisé et adapté au pipeline générique.

3. **Conversion vers les plateformes spécifiques :** La conversion consiste à transformer le pipeline générique en un pipeline spécifique à une plateforme donnée. En effet, comme vu dans la partie 4.2.2.2, chaque plateforme utilise un format dédié (certaines plateformes nécessitant des informations non requises par d'autres) pour la configuration du pipeline, même si la logique de base est la même. Le pipeline générique contient toutes les données nécessaires pour construire un pipeline spécifique. La conversion se fait en sélectionnant les données pertinentes pour la plateforme et en les réorganisant pour qu'elles soient en adéquation avec ses requis. Par exemple, lors de la conversion, les nœuds génériques seront convertis en nœuds '*entry_points'* dans *MLflow*, et '*stages'* dans *DVC.*

5.5.1.3 Activités manuelles

Les activités manuelles surviennent à différents niveaux du processus, dépendamment du degré de détail voulu par le développeur et l'état initial du projet.

4. **Standardisation des phases d'AM de haut niveau** : Il s'agit de déterminer globalement la liste des principales étapes du flux d'AM. D'après la littérature, ces étapes sont principalement: pré-traitement des données, entraînement, test, évaluation, déploiement et exécution. Cependant, il pourrait arriver que des développeurs rajoutent des étapes particulières à leur processus d'AM comme la phase de tunage des hyperparamètres. Les étapes de haut niveau peuvent être complétées avec des sous étapes. Par exemple, l'étape de pré-traitement des données peut comporter différentes sous-étapes de téléchargement des données et/ou de filtrage des données. Il n'est pas obligatoire de spécifier les sous-étapes de façon stricte. Seules les étapes de haut niveau devraient être standardisées afin de tirer parti des métriques proposées pour évaluer les pipelines[46]. Cette activité se fait de préférence avant toute activité automatique, car elle pourrait être utilisée pour automatiser les étapes manuelles d'assignation et d'ordonnancement des nœuds.

[46] Voir partie 4.5.3

5. **Assignation des nœuds identifiés aux étapes** : Dans la version initiale de la méthode, une intervention manuelle est nécessaire afin de relier les nœuds identifiés à des étapes du flux d'AM. Cela vient principalement de la difficulté d'extraire la logique et la fonctionnalité d'un programme simplement à partir du code source [125]. Cependant, il est possible d'utiliser la description des nœuds pour une assignation automatique. En effet, les parseurs offrent généralement la possibilité d'ajouter une description pour chaque instance du parseur. En spécifiant l'étape assignée à l'objet du parseur instancié dans cette description (par exemple `parser = ArgumentParser(description='Lance l'entraînement du modèle')`), il est possible d'utiliser des outils d'analyse textuels (Natural Language Processing NLP) afin de déterminer automatiquement l'étape du nœud qui en découle (pour le cas d'exemple mentionné précédemment, une simple expression régulière permettrait d'identifier le terme 'entraînement' dans la description, et donc lier le nœud à l'étape d'entraînement.).

6. **Ordonnancement des étapes** : Les processus d'AM s'exécutent généralement de façon itérative. Il est donc nécessaire de spécifier leur ordonnancement suivant l'ordre logique de leur exécution. Cette spécification se fait de façon manuelle dans le prototype implémenté car tout comme l'assignation des nœuds, l'ordonnancement des étapes nécessite de connaître la logique fonctionnelle du programme analysé, soit quelle fonctionnalité devrait être exécutée en premier [126]. L'automatisation de l'extraction de ces connaissances à partir du code source est un sujet complexe (surtout lorsqu'il y a des sous-étapes), et à notre connaissance, il n'existe encore aucun outil capable de définir la logique fonctionnelle d'un programme à partir de son code source. Il est néanmoins possible, grâce à la standardisation des étapes et leur description, d'automatiser cette fonction. (En utilisant une nomination des étapes par index par exemple, concaténer le nom de chaque étape à son index : 01_preprocessing, 02_training, etc.)

7. **Configurer les valeurs par défaut des arguments, lorsque ceux-ci n'ont pas été configuré dans le code** : Pour les cas où des valeurs par défaut n'ont pas été assigné aux arguments dans l'instruction d'instanciation du parseur, il est nécessaire de configurer ces valeurs manuellement dans le fichier générique créé. Néanmoins, afin d'une part de profiter des fonctionnalités offertes par les librairies des parseurs (documentation générée automatiquement) et optimiser la flexibilité du code, il est conseillé lorsque possible de définir des valeurs par défaut aux arguments des parseurs dans le code [127, 128].

5.5.2 Prototype

Nous avons construit un prototype pour la librairie '*argparse*'. Cette librairie a été choisie du fait de sa popularité auprès des développeurs (voir Figure 3.2 et Figure 4.8). Notre prototype pourrait être étendu à d'autres types de parseur, dans la mesure où les informations qu'ils offrent sont généralement les mêmes. Le prototype, construit en *Python*, implémente les principales activités automatiques de la méthode afin de reconstruire le pipeline d'AM. Nous avons aussi implémenté un script de conversion vers la plateforme *MLflow*. Ce script génère automatiquement les fichiers de configuration du pipeline '*MLprojet'* et celui de l'environnement '*conda.yaml'* et les enregistre dans le dossier racine du projet (voir Annexe D). Ainsi, aucune activité manuelle de configuration de ces éléments n'est requise lors de la migration vers *MLflow*. Aussi, le script est capable de mettre à jour automatiquement ces fichiers de configuration à chaque changement des données pertinentes dans le projet (noms et valeur par défaut des arguments, nouvelle librairie, etc.)

5.5.3 Évaluation et résultats

L'évaluation concerne principalement la méthode générant le pipeline générique, et non le script de conversion vers *MLflow*. Pour évaluer le prototype, les projets mentionnés au Chapitre 4 et possédant la librairie '*argparse'* comme parseur ont été utilisé (soit 10 projets). Le module a été analysé sous trois axes : Le taux d'extraction des nœuds, la probabilité de détection des valeurs des arguments, et le taux d'erreur introduit dans les instructions d'exécution des nœuds.

Pour des raisons de clarté et de cohérence, des explications et des recommandations en termes d'axe de recherche futurs sont également inclus dans l'analyse des résultats.

5.5.3.1 Extraction des nœuds

La même méthodologie d'évaluation[47] que celle utilisée pour l'outil MSR4ML a été mis en œuvre. Ainsi, pour chaque projet, nous avons calculé le nombre d'éléments identifiés. Le Tableau 5.5 récapitule les résultats obtenus relativement aux taux d'extraction des nœuds.

[47] Voir partie 5.4.3

Tableau 5.5 Résultats d'extraction des nœuds du pipeline

	Manuel	Automatique	Correspondance	Précision	Rappel	F_1
p1	2	2	2	1	1	1
p2	2	1	1	1	0.5	0.66
p3	1	1	1	1	1	1
p4	2	4	2	0.5	1	0.66
p6	1	2	1	0.5	1	0.66
p7	4	10	4	0.4	1	0.57
p9	2	3	2	0.66	1	0.8
p10	2	2	2	1	1	1
p12	2	3	2	0.66	1	0.8
p13	2	2	2	1	1	1
Moyenne				0.77	0.95	0.81

On observe que le rappel est de 0.95 et la précision 0.77, ce qui signifie que l'outil extrait efficacement 77% des nœuds des pipelines d'AM. Cela traduit une sur-extraction des nœuds. En effet, dans plusieurs projets, on observe que le nombre de nœuds extraits est supérieur au nombre de nœuds identifiés en consultant la documentation. Une analyse détaillée des différences observées a permis d'établir les principales raisons pouvant expliquer ce fait :

1. Le parseur est utilisé pour des activités autres que celles reliées au flux d'AM. Ceci a été remarqué dans le projet p_7 par exemple, où des activités de journalisation et de génération de la documentation sont gérées en utilisant la librairie '*argparse'.* Cela dénote notamment du problème de séparation entre les artéfacts logiciels et ceux d'AM, qui résident ensemble dans le même dépôt;

2. Plusieurs fonctions implémentent différentes versions du parseur (p_4, p_6 et p_{12}). Chacune de ces implémentations est extraite comme possible nœud par l'outil, car la fonction à exécuter est déterminée dynamiquement au moment de l'exécution du code. Ce cas de figure est le plus observé. Cependant, nous pensons que cette détection peut néanmoins être utile au développeur pour la traçabilité même si elle n'est pas intégrée à la version finale du pipeline (elle permet de connaitre toutes les implémentations du parseur et ainsi de pouvoir migrer entre différentes versions lors d'essais par exemple);

3. Le parseur implémenté n'est pas utilisé (p_9). Dans ce cas, il s'agissait d'une ancienne implémentation du parseur qui a été abandonnée au profit d'une nouvelle, mais qui a quand même été conservée dans le projet. Cette pratique a tendance à créer des '*code zombie'*, qui sont des

portions de code expirés et non utilisés, mais qui sont quand même conservés dans l'application. Cette pratique est connue pour poser des défis en termes de maintenance et de sécurité [129].

5.5.3.2 Identification des données des nœuds

Tableau 5.6 Résultats d'évaluation de l'identification des données de nœuds

	Manuel	Automatique	Correspondance	Précision	Rappel	F_1
Valeurs par défaut	64	55	55	1	0.85	0.92
Types par défaut	64	61	55	0.9	0.85	0.88
Fichiers de base	18	14	11	0.78	0.61	0.68
Moyenne				0.84	0.9	0.86

L'on remarque que les métriques sont satisfaisantes et traduisent un taux de détection élevé pour la majorité des éléments. Une itération subséquente nous a permis d'analyser en détail les erreurs. Les raisons suivantes expliquent l'origine de ces erreurs.

1. Concernant la détection des valeurs par défaut, nous avons remarqué que les éléments faux étaient constitués d'arguments dont la valeur n'a pas été détectée (Ce champ était null dans le résultat). Cela signifie que le système n'a pas pu associer une valeur à cet argument lors du traitement, notamment à cause de l'utilisation de valeurs dynamiques dans son assignation. En effet, tous les arguments dont les valeurs n'ont pu être déterminés avaient été assigné des valeurs dynamiques et non des primitives *Python*. Par exemple, le projet p_6 possède un argument dont la valeur par défaut, telle qu'implémentée dans le code, est `'datetime.now()'`. De ce fait, cette valeur par défaut change à chaque exécution du code, ce qui ne permet pas de lui assigner une valeur statique;

2. Quant aux types par défaut extraits, les raisons sont semblables à celles relatives aux valeurs par défaut, soit la non-utilisation de primitives *Python*. En effet, le prototype dans sa version initiale ne reconnait que les primitives *Python* comme type (int, bool, str). Par exemple, l'argument `'--wv_cased'` du projet p_2 avait un type par défaut `'str2bool'` qui est importé dans le code à partir d'une librairie tierce. Pour résoudre cela, La fonction d'extraction des types a alors été modifiée afin de retourner les valeurs exactes des types tels que trouvés, sans se limiter aux primitives Python. Les métriques afférentes ont été largement amélioré, avec un score F_1 de 100% pour la détection des types.

3. Concernant les fichiers de base, la raison principale des différences est le fait que l'outil dans sa version initiale considérait que tout fichier où une instance du parseur est trouvée est un fichier de base. Cependant, après analyse minutieuse, nous avons remarqué que ce n'était pas toujours le cas. Dans certains projets (p_{10} par exemple), le parseur était instancié dans un fichier (‘*utils.py*) sous forme de fonction, et cette fonction renvoyait l'objet instancié comme valeur de retour. Ce fichier était ensuite utilisé par un autre (‘*run.py’*) lors de l'exécution, constituant ainsi le fichier de base du nœud associé. Nous pensons que cette limitation pourrait être résolue en utilisant une extraction dynamique, afin de traiter ces cas de figure.

5.5.3.3 Taux d'erreur introduit dans les nœuds

Étant donné que les instructions d'exécution des nœuds sont déduites automatiquement à partir des fichiers de base, des valeurs par défaut des arguments et des types par défauts, nous définissons une métrique permettant de calculer le taux d'erreur e introduit par les nœuds à partir de la précision de chaque élément (V représente les valeurs par défaut, T représente les types par défaut et F les fichiers de base). Ce taux d'erreur peut être utilisé pour estimer la quantité de travail nécessaire relativement à la vérification et la correction des nœuds extraits.

Soit p la précision d'un élément, la précision moyenne à partir de ces trois éléments est : $p_{moy} = \frac{\sum_{i=1}^{3} p(i)}{3}$, p(i) étant la précision de chaque élément. Sachant que le taux d'erreur e est proportionnel à la précision suivant la formule $e = 1 - p$ [130], l'on déduit le taux d'erreur moyen qui est $e_{moy} = 1 - \frac{\sum^{3} p(i)}{\frac{i=1}{3}}$.

$$e_{moy} = 1 - \frac{p(V) + p(T) + p(F)}{3}$$

Plus le taux d'erreur est élevé, et plus le nombre de modifications manuelles à apporter au pipeline extrait sera important. La définition des seuils d'acceptation pourra dépendre de chaque développeur, mais nous pensons qu'un taux d'erreur de plus de 0.5 (50%) serait à éviter dans la mesure où cela traduirait une mauvaise implémentation de la méthodologie proposée. Pour le prototype construit, le taux d'erreur est de 0.13, ce qui est acceptable.

Étant donné les analyses du Chapitre 4 relativement à l'organisation des nœuds par les plateformes, l'on peut observer que le type n'est pas utilisée partout (seul *MLflow* nécessite la spécification du type dans la configuration du pipeline). Nous pensons donc qu'il serait possible d'améliorer cette

formule en introduisant un poids inférieur à 1 pour la précision des types. Pour cela, un axe de recherche recommandé serait d'étudier la prévalence de cette pratique (la nécessité de spécifier des types dans les pipelines) dans d'autres plateformes afin d'en déduire le poids le plus adapté selon la récurrence de cette pratique.

Nous n'avons pas introduit de métrique pour l'extraction des nœuds du pipeline pour deux raisons :

1. Le rappel est assez élevé (95%), ce qui signifie que dans la grande majorité des cas, tous les nœuds seront extraits

2. Il n'existe pas de standard relativement à la séparation entre artéfacts logiciels et d'AM. Ainsi, chaque développeur peut décider de considérer un artéfact comme relatif à l'aspect d'AM, ou du logiciel, ou des deux. Par exemple, le système pourrait identifier un nœud de création d'une API hébergeant le modèle final comme faisant partie du flux d'AM, tandis qu'un développeur pourrait le considérer comme artéfact logiciel. Nous souhaitons ainsi laisser ce choix au développeur. Davantage de recherche devrait être mené dans ce domaine afin de fournir une meilleure clarification de cette séparation.

5.6 Travaux connexes

La traçabilité et la reproductibilité de l'Apprentissage Machine sont un sujet d'actualité dans la communauté des chercheurs [131]. Aravantinos et al. ont proposé un modèle de traçabilité descriptive pour les applications d'Apprentissage en Profondeur [132]. En utilisant des méthodes de génie logiciel, les auteurs ont introduit un ensemble d'activités et d'artéfacts pour la gestion du développement des applications d'Apprentissage en Profondeur. Le modèle proposé intègre également les exigences et l'architecture du modèle. Cela a été une source d'inspiration précieuse lors de la mise en œuvre de notre cadriciel MSR4ML. Notre contribution peut être utilisée comme une extension de leur modèle, qui est principalement descriptif. Par exemple, les artéfacts et leurs attributs tels que proposés dans le modèle peuvent être utilisés pour organiser et gérer les types d'artéfacts extraits par notre module de classification.

La traçabilité d'AM nécessite une gestion complète des métadonnées du projet. Tsay et al. ont proposé AIMMX, un outil d'extraction automatique des métadonnées du modèle à partir des dépôts logiciels [133]. À l'aide des techniques MSR et de l'apprentissage automatique, AIMMX extrait les informations pertinentes des dépôts d'AM, telles que le nom du jeu de données et l'algorithme

d'AM utilisé par le modèle. Cependant, contrairement à notre cadriciel, l'outil ne récupère pas les fichiers de données réels, ni le lien avec les fichiers de code. Il est principalement axé sur l'identification des métadonnées du modèle à des fins de connaissance.

Meurice et al. [134] utilisent une méthode similaire à celle du module d'identification d'artéfacts pour évaluer l'utilisation d'artéfacts de données dans le code. Ils utilisent l'analyse statique des fichiers de code pour identifier l'utilisation des systèmes de gestion de base de données dans les projets Java, comme nous le faisons pour extraire l'utilisation des artéfacts dans les projets d'AM : spécifier les noms de méthode, extraire les arguments du nœud d'invocation des méthodes et déduire leurs valeurs.

5.7 Conclusion

Dans ce chapitre, nous avons présenté deux outils facilitant la configuration des processus et artéfacts d'Apprentissage Machine : un cadriciel modulaire de reconstruction de la traçabilité des artéfacts d'AM et une méthode d'extraction des pipelines d'AM à partir du code source. Nous implémenté des prototypes pour ces deux éléments, et avons obtenus des résultats prometteurs en termes de capacité de reconstruction et de taux d'extraction.

CHAPITRE 6 SYNTHÈSE ET CONCLUSION

6.1 Synthèse des travaux

Cette étude porte sur l'analyse et la reconstruction des processus d'Apprentissage Machine à partir de leurs dépôts logiciels. Les travaux conduits concernent principalement l'identification des pratiques de développement d'AM, les défis relatifs à l'adoption de plateformes spécialisées pour l'optimisation des processus de ces applications, ainsi que des méthodes de reconstruction de ces processus.

Notre hypothèse de départ supposait qu'il était possible de reconstruire les pratiques, artéfacts et pipelines de développement des applications d'AM à partir du minage de dépôts logiciels. En utilisant une approche mixte combinant le minage des dépôts logiciels, l'analyse statistique et le prototypage, nous avons exploré la structure des dépôts d'AM et des plateformes MLOps. Ces plateformes bien que prometteuses, présentent un ensemble de défis que nous avons étudié. Au terme de l'étude, nous avons démontré l'hypothèse de départ, et nos observations ont établi un lien causal entre les pratiques de développement d'AM et la nécessité de proposer un cadre et des outils génériques pouvant aider les développeurs à mieux concevoir, organiser, construire, exécuter et maintenir les applications d'AM.

6.2 Considérations sur la portée des résultats

Le corpus de données étudié dans le présent travail étant principalement constitué de projets de recherche, les résultats obtenus pourraient être baisés envers ce type de projets. Dans la littérature actuelle, la plupart des travaux abordant des projets d'entreprise plus professionnels et plus matures ont tendance à n'étudier qu'un ou quelques systèmes d'AM propres à l'entreprise, sous forme d'études de cas (voir par exemple [1, 2, 118]. Cela dénote de la difficulté d'obtenir un accès en masse à ce type de projets, notamment à cause des considérations de confidentialité.

Le déploiement de systèmes d'AM en entreprise bénéficie d'un encadrement plus spécifique, ainsi que d'une meilleure disponibilité des compétences et des outils technologiques. De ce fait, les pratiques décrites dans le Chapitre 3 ou encore les défis d'adoption des plateformes MLOps devraient être contextualisés au domaine open source. Au vu de la démocratisation des systèmes d'AM, de plus en plus de développeurs se lancent dans la mise en place de systèmes d'AM pour

des besoins individuels ou de recherche. Les résultats actuels seraient donc utiles principalement pour cette communauté, et un axe de recherche pertinent serait de reproduire l'étude actuelle dans des environnements d'entreprise et des systèmes d'AM plus grands et plus matures.

6.3 Contribution au progrès des connaissances

Les résultats obtenus démontrent des avancements intéressant relativement aux processus de développement d'AM ainsi que les outils afférents.

1. **Identification des pratiques de développement pour applications d'AM :** nous avons identifié 8 pratiques générales observées lors du développement des applications d'AM en utilisant les dépôts logiciels. Ces pratiques démontrent notamment la corrélation élevée entre les librairies d'AM d'une part, et les limites des systèmes de contrôle de version traditionnels relativement au développement d'AM d'autre part;
2. **Identification des défis d'adoption des plateformes MLOps, proposition de solutions et recommandations :** par une étude expérimentale des activités d'adoption et de migration des plateformes *MLflow* et *DVC*, nous avons identifié 11 défis et proposé un ensemble de solutions et recommandations afin d'optimiser ces activités et réduire l'impact de ces défis. Nous avons aussi introduit deux métriques permettant d'estimer le degré de modularité des flux d'AM. A partir des seuils proposés, les développeurs peuvent ainsi déterminer si l'architecture de leur projet d'AM est optimisé pour la migration vers les plateformes MLOps, ou si une restructuration (refactoring) est nécessaire;
3. **Processus générique de migration des projets d'AM vers les plateformes**: à partir des expérimentations menées lors de l'identification des défis d'adoption des plateformes MLOps, nous avons introduit un processus générique de migration des dépôts logiciels vers les plateformes MLOps comprenant 6 phases principales : Analyse; Adaptation; configuration des environnements et du pipeline; exécution et validation; maintenance. Ce processus peut être utilisé par la communauté afin de planifier, organiser et exécuter les activités d'adoption et de migration vers les plateformes MLOps;
4. **Cadriciel générique de reconstruction de la traçabilité des artéfacts des projets d'AM :** nous avons présenté un Framework qui combine l'analyse de code statique avec des techniques MSR afin de permettre aux développeurs de récupérer l'utilisation et la

traçabilité des artefacts dans les projets d'AM basés sur *Git*. Notre approche est capable de gérer des projets d'AM à un stade avancé de développement pour reconstruire les liens entre les artefacts et le code. Dans une évaluation préliminaire, nous avons montré que l'outil peut récupérer une majorité d'artefacts d'AM (plus de 60%) dans leurs dépôts logiciels. Les possibilités d'utilisation de ce Framework incluent le suivi et l'explication des modifications ayant affecté des artéfacts particuliers ou encore l'exécution automatique de pipelines DevOps (de réentraînement par exemple) lorsqu'une modification touchant des artéfacts liés est détectée;

5. **Méthode semi-automatique d'extraction et d'organisation des pipelines d'AM** : nous avons introduit, appliqué et testé une méthode d'extraction et d'organisation des pipelines d'AM à partir du code source qui offre une précision de 77%. Cette méthode permet notamment de faciliter et automatiser les tâches de configuration des pipelines d'AM en vue de leur intégration dans les plateformes MLOps.

RÉFÉRENCES

[1] M. S. Rahman, E. Rivera, F. Khomh, Y.-G. Guéhéneuc, and B. Lehnert, "Machine learning software engineering in practice: An industrial case study," *arXiv preprint arXiv:1906.07154*, 2019.

[2] S. Amershi *et al.*, "Software engineering for machine learning: A case study," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019: IEEE, pp. 291-300.

[3] S. Alsheibani, Y. Cheung, and C. Messom, "Artificial Intelligence Adoption: AI-readiness at Firm-Level," in *PACIS*, 2018, p. 37.

[4] A. Agrawal, J. Gans, and A. Goldfarb, *Prediction machines: the simple economics of artificial intelligence*. Harvard Business Press, 2018.

[5] S. A. Alsheibani, D. Cheung, and D. Messom, "Factors inhibiting the adoption of artificial intelligence at organizational-level: A preliminary investigation," 2019.

[6] D. Rettas, S. Lerner, and B. White, "The Evolution of Artificial Intelligence," *Enterprise Digitalization*, pp. 3-8, 2019.

[7] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 25-31 May 2019 2019, pp. 507-517, doi: 10.1109/MSR.2019.00077.

[8] P. S. Janardhanan, "Project repositories for machine learning with TensorFlow," *Procedia Computer Science*, vol. 171, pp. 188-196, 2020/01/01/ 2020, doi: https://doi.org/10.1016/j.procs.2020.04.020.

[9] A. Barrak, E. E. Eghan, and B. Adams, "On the Co-evolution of ML Pipelines and Source Code - Empirical Study of DVC Projects," *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 422-433, 2021.

[10] N. N. Zolkifli, A. Ngah, and A. Deraman, "Version Control System: A Review," *Procedia Computer Science*, vol. 135, pp. 408-415, 2018/01/01/ 2018, doi: https://doi.org/10.1016/j.procs.2018.08.191.

[11] K. Sinha, J. Pineau, J. Forde, J. Dodge, and R. Stojnic, "Papers with Code: the latest in machine learning, 2020." [Online]. Available: https://paperswithcode.com.

[12] "GitHub." www.github.com (accessed July 19, 2021).

[13] C. Voskoglou, "What is the best programming language for Machine Learning," *Towards Data Science*, vol. 5, 2017.

[14] A. E. Hassan, "The road ahead for mining software repositories," in *2008 Frontiers of Software Maintenance*, 2008: IEEE, pp. 48-57.

[15] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of software maintenance and evolution: Research and practice*, vol. 19, no. 2, pp. 77-131, 2007.

[16] H. Kagdi, J. I. Maletic, and B. Sharif, "Mining software repositories for traceability links," in *15th IEEE International Conference on Program Comprehension (ICPC'07)*, 2007: IEEE, pp. 145-154.

[17] W. Poncin, A. Serebrenik, and M. Van Den Brand, "Process mining software repositories," in *2011 15th European Conference on Software Maintenance and Reengineering*, 2011: IEEE, pp. 5-14.

[18] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," (in English), *J Softw Maint Evol-R*, vol. 19, no. 2, pp. 77-131, Mar-Apr 2007, doi: 10.1002/smr.344.

[19] W. J. Dixon and F. J. Massey Jr, "Introduction to statistical analysis," 1951.

[20] S. L. Pfleeger, "Experimental design and analysis in software engineering," *Annals of Software Engineering*, vol. 1, no. 1, pp. 219-253, 1995/12/01 1995, doi: 10.1007/BF02249052.

[21] N. Juristo and A. M. Moreno, *Basics of software engineering experimentation*. Springer Science & Business Media, 2013.

[22] R. Budde, K. Kautz, K. Kuhlenkamp, and H. Züllighoven, "What is prototyping?," *Information Technology & People*, 1992.

[23] M. Buchenau and J. F. Suri, "Experience prototyping," in *Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques*, 2000, pp. 424-433.

[24] P. Jain, A. Sharma, and L. Ahuja, "The Impact of Agile Software Development Process on the Quality of Software Product," in *2018 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, 29-31 Aug. 2018 2018, pp. 812-815, doi: 10.1109/ICRITO.2018.8748529.

[25] M. Agrawal and K. Chari, "Software effort, quality, and cycle time: A study of CMM level 5 projects," *IEEE Transactions on software engineering*, vol. 33, no. 3, pp. 145-156,2007.

[26] D. Horie, T. Kasahara, Y. Goto, and J. Cheng, "A new model of software life cycle processes for consistent design, development, management, and maintenance of secure information systems," in *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*, 2009: IEEE, pp. 897-902.

[27] N. B. Ruparelia, "Software development lifecycle models," *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 3, pp. 8-13, 2010.

[28] M. Tuteja and G. Dubey, "A research study on importance of testing and quality assurance in software development life cycle (SDLC) models," *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 2, no. 3, pp. 251-257, 2012.

[29] S. B. A. S. U. Rani, "A detailed study of Software Development Life Cycle (SDLC) Models," *International Journal of Engineering and Computer Science*, vol. 6, 2017.

[30] A. Alshamrani and A. Bahattab, "A comparison between three SDLC models waterfall model, spiral model, and Incremental/Iterative model," *International Journal of Computer Science Issues (IJCSI)*, vol. 12, no. 1, p. 106, 2015.

[31] K. Petersen, C. Wohlin, and D. Baca, "The waterfall model in large-scale development," in *International Conference on Product-Focused Software Process Improvement*, 2009: Springer, pp. 386-400.

[32] C. Johansson and C. Bucanac, "The v-model," *IDE, University Of Karlskrona, Ronneby,* 1999.

[33] M. Z. Toh, S. Sahibuddin, and M. N. r. Mahrin, "Adoption Issues in DevOps from the Perspective of Continuous Delivery Pipeline," presented at the Proceedings of the 2019 8th International Conference on Software and Computer Applications, Penang, Malaysia, 2019. [Online]. Available: https://doi.org/10.1145/3316615.3316619.

[34] K. Baseer, A. R. M. Reddy, and C. S. Bindu, "A systematic survey on waterfall vs. agile vs. lean process paradigms," *I-Manager's Journal on Software Engineering,* vol. 9, no. 3, p. 34, 2015.

[35] N. M. A. Munassar and A. Govardhan, "A comparison between five models of software engineering," *International Journal of Computer Science Issues (IJCSI),* vol. 7, no. 5, p. 94, 2010.

[36] M. Fowler and J. Highsmith, "The agile manifesto," *Software development,* vol. 9, no. 8, pp. 28-35, 2001.

[37] K. Schwaber, "Scrum development process," in *Business object design and implementation*: Springer, 1997, pp. 117-134.

[38] M. O. Ahmad, J. Markkula, and M. Oivo, "Kanban in software development: A systematic literature review," in *2013 39th Euromicro conference on software engineering and advanced applications*, 2013: IEEE, pp. 9-16.

[39] S. Aftab *et al.*, "Using FDD for small project: An empirical case study," *International Journal of Advanced Computer Science and Applications,* vol. 10, no. 3, pp. 151-158, 2019.

[40] R. Hoda, N. Salleh, and J. Grundy, "The rise and evolution of agile software development," *IEEE software,* vol. 35, no. 5, pp. 58-63, 2018.

[41] G. Azizyan, M. K. Magarian, and M. Kajko-Matsson, "Survey of Agile Tool Usage and Needs," in *2011 Agile Conference*, 7-13 Aug. 2011 2011, pp. 29-38, doi: 10.1109/AGILE.2011.30.

[42] D. Spinellis, "Version control systems," *IEEE Software,* vol. 22, no. 5, pp. 108-109, 2005.

[43] L. Torvalds and J. HAMANO. "Git." www.git-scm.com (accessed July 7, 2021).

[44] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development*. " O'Reilly Media, Inc.", 2012.

[45] M. Hüttermann, *DevOps for developers*. Apress, 2012.

[46] E. Diel, S. Marczak, and D. S. Cruzes, "Communication challenges and strategies in distributed DevOps," in *2016 IEEE 11th International Conference on Global Software Engineering (ICGSE)*, 2016: IEEE, pp. 24-28.

[47] T. M. Mitchell, "Machine learning," 1997.

[48] J. Alzubi, A. Nayyar, and A. Kumar, "Machine learning from theory to algorithms: an overview," in *Journal of physics: conference series*, 2018, vol. 1142, no. 1: IOP Publishing, p. 012012.

[49] S. Raschka, *Python machine learning*. Packt publishing ltd, 2015.

[50] D. Sato, A. Wider, and C. Windheuser. "Continuous delivery for machine learning." Retrieved from martinFowler.com. https://martinfowler.com/articles/cd4ml.html (accessed September 13, 2020).

[51] S. Mäkinen, H. Skogström, E. Laaksonen, and T. Mikkonen, "Who Needs MLOps: What Data Scientists Seek to Accomplish and How Can MLOps Help?," *arXiv preprint arXiv:2103.08942*, 2021.

[52] A. Wahaballa, O. Wahballa, M. Abdellatief, H. Xiong, and Z. Qin, *Toward unified DevOps model*. 2015.

[53] "IEEE Standard for Configuration Management in Systems and Software Engineering," *IEEE Std 828-2012 (Revision of IEEE Std 828-2005)*, pp. 1-71, 2012, doi: 10.1109/IEEESTD.2012.6170935.

[54] "IEEE Standard Adoption of ISO/IEC 15026-1--Systems and Software Engineering--Systems and Software Assurance--Part 1: Concepts and Vocabulary," *IEEE P15026-1, Jul2014*, pp. 1-34, 2014.

[55] E. Bisong, "Kubeflow and kubeflow pipelines," in *Building Machine Learning and Deep Learning Models on Google Cloud Platform*: Springer, 2019, pp. 671-685.

[56] "Data version control • dvc." https://dvc.org/ (accessed July 2, 2021).

[57] "MLflow, a platform for the Machine Learning LifeCycle." https://mlflow.org/ (accessed July 2, 2021).

[58] M. Vartak *et al.*, "ModelDB: a system for machine learning model management," in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, 2016, pp. 1-3.

[59] "Dimensional Research. Artificial Intelligence and Machine Learning Projects Are Obstructed by Data Issues." (accessed July 8, 2021).

[60] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, 2009, pp. 57-62.

[61] V. Bauer, L. Heinemann, and F. Deissenboeck, "A structured approach to assess third-party library usage," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012: IEEE, pp. 483-492.

[62] M. A. Saied, A. Ouni, H. Sahraoui, R. G. Kula, K. Inoue, and D. Lo, "Automated Inference of Software Library Usage Patterns," *arXiv preprint arXiv:1612.01626*, 2016.

[63] M. A. Saied, A. Ouni, H. Sahraoui, R. G. Kula, K. Inoue, and D. Lo, "Improving reusability of software libraries through usage pattern mining," *Journal of Systems and Software*, vol. 145, pp. 164-179, 2018/11/01/ 2018, doi: https://doi.org/10.1016/j.jss.2018.08.032.

[64] "ast — Abstract Syntax Trees." https://docs.python.org/3/library/ast.html (accessed June 21, 2021).

[65] J. Hipp, U. Güntzer, and G. Nakhaeizadeh, "Algorithms for association rule mining—a general survey and comparison," *ACM sigkdd explorations newsletter*, vol. 2, no. 1, pp. 58-64, 2000.

[66] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357-362, 2020.

[67] "Python 3.9.5 documentation." https://docs.python.org (accessed June 21, 2021).

[68] G. Piatetsky-Shapiro, "Discovery, Analysis, and Presentation of Strong Rules," in *Knowledge Discovery in Databases*, 1991.

[69] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases," in *SIGMOD '93*, 1993.

[70] L. Zhou, S. Pan, J. Wang, and A. V. Vasilakos, "Machine learning on big data: Opportunities and challenges," *Neurocomputing*, vol. 237, pp. 350-361, 2017.

[71] J. Huang, Y.-F. Li, and M. Xie, "An empirical analysis of data preprocessing for machine learning-based software cost estimation," *Information and Software Technology*, vol. 67, pp. 108-127, 2015/11/01/ 2015, doi: https://doi.org/10.1016/j.infsof.2015.07.004.

[72] I. Žliobaitė, M. Pechenizkiy, and J. Gama, "An overview of concept drift applications," *Big data analysis: new algorithms for a new society*, pp. 91-114, 2016.

[73] G. A. A. Prana, C. Treude, F. Thung, T. Atapattu, and D. Lo, "Categorizing the content of GitHub README files," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1296-1327, 2019.

[74] S. Nagar, "Introduction to Python basics," in *Introduction to Python for engineers and scientists*: Springer, 2018, pp. 13-30.

[75] S. Grafberger, J. Stoyanovich, and S. Schelter, "Lightweight Inspection of Data Preprocessing in Native Machine Learning Pipelines," in *CIDR*, 2021.

[76] S. B. Kotsiantis, D. Kanellopoulos, and P. E. Pintelas, "Data preprocessing for supervised leaning," *International journal of computer science*, vol. 1, no. 2, pp. 111-117, 2006.

[77] Z. Cai, Y. Long, and L. Shao, "Classification complexity assessment for hyper-parameter optimization," *Pattern Recognition Letters*, vol. 125, pp. 396-403, 2019.

[78] G. Luo, "A review of automatic selection methods for machine learning algorithms and hyper-parameter values," *Network Modeling Analysis in Health Informatics and Bioinformatics*, vol. 5, no. 1, pp. 1-16, 2016.

[79] J. Zhou, W. Huang, and F. Chen, "Facilitating Machine Learning Model Comparison and Explanation Through A Radial Visualisation," *arXiv preprint arXiv:2104.07377*, 2021.

[80] D. B. Terry, "An analysis of naming conventions for distributed computer systems," *ACM SIGCOMM Computer Communication Review*, vol. 14, no. 2, pp. 218-224, 1984.

[81] N. Anquetil and T. Lethbridge, "File clustering using naming conventions for legacy systems," in *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, 1997: Citeseer, p. 2.

[82] K. Antin, "File naming conventions: why you want them and how to create them," ed, 2016.

[83] N. Anquetil and T. C. Lethbridge, "Recovering software architecture from the names of source files," *Journal of Software Maintenance: Research and Practice*, vol. 11, no. 3, pp. 201-221, 1999.

[84] N. Stephenson, *In the beginning... was the command line*. Avon Books New York, 1999.

[85] D. Paul and O. Michael, "Continuous Delivery Patterns and Anti-Patterns," 2018. Accessed: July 1, 2021. [Online]. Available: https://victorsalaun.github.io/assets/pdf/Continuous_Delivery_Patterns_and_Anti_Patterns.pdf

[86] M. Blaha, *Patterns of data modeling*. CRC Press, 2010.

[87] A. Walker, D. Das, and T. Cerny, "Automated code-smell detection in microservices through static analysis: A case study," *Applied Sciences*, vol. 10, no. 21, p. 7800, 2020.

[88] R. Robbes, "Mining a change-based software repository," in *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, 2007: IEEE, pp. 15-15.

[89] J. Vouillon and R. Di Cosmo, "Broken sets in software repository evolution," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013: IEEE, pp. 412-421.

[90] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325-364, 2011.

[91] A. T. Njomou, A. J. B. Africa, B. Adams, and M. Fokaefs, "MSR4ML: Reconstructing Artifact Traceability in Machine Learning Repositories," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021: IEEE, pp. 536-540.

[92] M. Zaharia *et al.*, "Accelerating the machine learning lifecycle with MLflow," *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 39-45, 2018.

[93] W. R. B. Severtson, L. Franks, and G. Ericson, "What is the team data science process?," *Microsoft Azure*, 2017.

[94] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "The KDD process for extracting useful knowledge from volumes of data," *Communications of the ACM*, vol. 39, no. 11, pp. 27-34, 1996.

[95] R. a. H. J. Wirth, "CRISP-DM: Towards a standard process model for data mining," *Proceedings of the 4th International Conference on the Practical Applications of Knowledge Discovery and Data Mining*.

[96] A. Goyal, "Machine Learning Operations," *International Journal of Information Technology Insights & Transformations [ISSN: 2581-5172 (online)]*, vol. 4, no. 2, 2020.

[97] E. Raj, "Edge MLOps framework for AIoT applications," 2020.

[98] "Iterative, Developers tools for Machine Learning." https://iterative.ai/ (accessed June 13, 2021).

[99] "Databricks, the Data and AI company." https://databricks.com/ (accessed June 15, 2021).

[100] M. Zaharia *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56-65, 2016.

[101] "GitHub Stars: inspire, educate & influence developer communities." https://stars.github.com/ (accessed July 1, 2021).

[102] H. Borges and M. T. Valente, "What's in a github star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112-129, 2018.

[103] H. Borges, A. Hora, and M. T. Valente, "Predicting the popularity of GitHub repositories," in *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2016, pp. 1-10.

[104] O. Ben-Kiki, C. Evans, and B. Ingerson, "Yaml ain't markup language (yaml™) version 1.1," *Working Draft 2008-05*, vol. 11, 2009.

[105] "Anaconda, The World's Most Popular Data Science Platform." (accessed May 21, 2021).

[106] F. Luan, M. Nygård, and T. Mestl, "A mathematical framework for modeling and analyzing migration time," in *Proceedings of the 10th annual joint conference on Digital libraries*, 2010, pp. 323-332.

[107] M. Arnold *et al.*, "Towards automating the AI operations lifecycle," *arXiv preprint arXiv:2003.12808*, 2020.

[108] R. Ramler and J. Gmeiner, "Practical Challenges in Test Environment Management," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, 2014: IEEE, pp. 358-359.

[109] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post, "Software development environments for scientific and engineering software: A series of case studies," in *29th International Conference on Software Engineering (ICSE'07)*, 2007: Ieee, pp. 550-559.

[110] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang, "Hardware for machine learning: Challenges and opportunities," in *2017 IEEE Custom Integrated Circuits Conference (CICC)*, 2017: IEEE, pp. 1-8.

[111] I. Stančin and A. Jović, "An overview and comparison of free Python libraries for data mining and big data analysis," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019: IEEE, pp. 977-982.

[112] D. Kang, T. J. Jun, D. Kim, J. Kim, and D. Kim, "ConVGPU: GPU management middleware in container based virtualized environment," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017: IEEE, pp. 301-309.

[113] "Chef Software DevOps and automation tool." https://www.chef.io/ (accessed June 15, 2021).

[114] "Puppet - Powerful infrastructure automation and delivery." https://puppet.com/ (accessed June 15, 2021).

[115] R. C. Martin, "Design principles and design patterns," *Object Mentor*, vol. 1, no. 34, p. 597, 2000.

[116] G. Kiczales and E. Hilsdale, "Aspect-oriented programming," in *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, 2001, p. 313.

[117] L. Bergmans and M. Aksit, "Composing multiple concerns using composition filters," *Communications of the ACM*, vol. 44, no. 10, 2001.

[118] T. Granlund, A. Kopponen, V. Stirbu, L. Myllyaho, and T. Mikkonen, "MLOps Challenges in Multi-Organization Setup: Experiences from Two Real-World Cases," *arXiv preprint arXiv:2103.08937*, 2021.

[119] "CML | Data Version Control · DVC." https://dvc.org/doc/cml (accessed July 5, 2021).

[120] C. Jones, "Variations in software development practices," *IEEE software*, vol. 20, no. 6, pp. 22-27, 2003.

[121] C. Renggli, L. Rimanic, N. M. Gürel, B. Karlaš, W. Wu, and C. Zhang, "A Data Quality-Driven View of MLOps," *arXiv preprint arXiv:2102.07750*, 2021.

[122] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst, "Dynamic inference of abstract types," in *Proceedings of the 2006 international symposium on Software testing and analysis*, 2006, pp. 255-265.

[123] R. S. Arnold and S. A. Bohner, "Impact analysis-towards a framework for comparison," in *1993 Conference on Software Maintenance*, 1993: IEEE, pp. 292-301.

[124] "Astroid's documentation." http://pylint.pycqa.org/projects/astroid/en/latest/ (accessed June 2, 2021).

[125] I. Schröter, J. Krüger, J. Siegmund, and T. Leich, "Comprehending Studies on Program Comprehension," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 22-23 May 2017 2017, pp. 308-311, doi: 10.1109/ICPC.2017.9.

[126] A. von Mayrhauser and A. M. Vans, "From code understanding needs to reverse engineering tool capabilities," in *Proceedings of 6th International Workshop on Computer-Aided Software Engineering*, 1993: IEEE, pp. 230-239.

[127] L. Buitinck *et al.*, "API design for machine learning software: experiences from the scikit-learn project," *arXiv preprint arXiv:1309.0238*, 2013.

[128] J. Savolainen, J. Bosch, J. Kuusela, and T. Männistö, "Default values for improved product line management," in *Proceedings of the 13th International Software Product Line Conference*, 2009, pp. 51-60.

[129] S. Eder, M. Junker, E. Jürgens, B. Hauptmann, R. Vaas, and K.-H. Prommer, "How much does unused code matter for maintenance?," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012: IEEE, pp. 1102-1111.

[130] P. Branco, L. Torgo, and R. Ribeiro, "A survey of predictive modelling under imbalanced distributions," *arXiv preprint arXiv:1505.01658*, 2015.

[131] Z. C. Lipton and J. Steinhardt, "Research for practice: troubling trends in machine-learning scholarship," *Communications of the ACM*, vol. 62, no. 6, pp. 45-53, 2019.

[132] V. Aravantinos and F. Diehl, "Traceability of deep neural networks," *arXiv preprint arXiv:1812.06744*, 2018.

[133] J. Tsay, A. Braz, M. Hirzel, A. Shinnar, and T. Mummert, "Aimmx: Artificial intelligence model metadata extractor," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 81-92.

[134] L. Meurice, C. Nagy, and A. Cleve, "Static analysis of dynamic database usage in java systems," in *International Conference on Advanced Information Systems Engineering*, 2016: Springer, pp. 491-506.

ANNEXE A EXEMPLE DE RÈGLES DE CLASSIFICATION

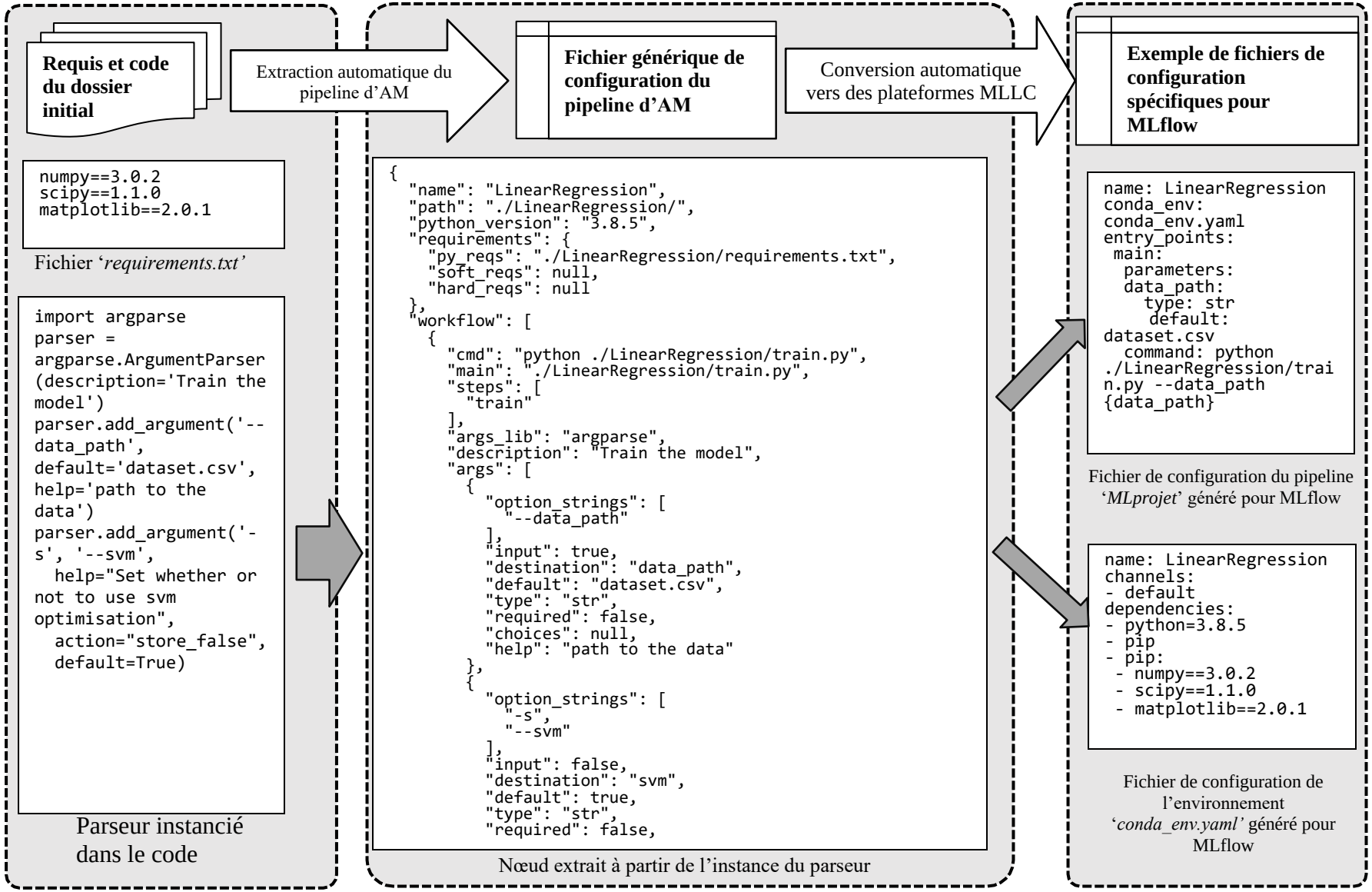
Règle	Méthode	Règle	Priorité	Exemple	Résultat
1	Selon le nom de l'artéfact	Si 'data' est présent dans le nom de l'artéfact	1	data/glove.txt	{'possible_categories' : {'data' : 1}, 'categorie' : 'data'}
		Si 'model' est présent dans le nom de l'artéfact		results/final_model.npy	{'possible_categories' : {'model : 1}, 'categorie' : 'model}
		Si 'conf est présent dans le nom de l'artéfact		Conf/params.yml	{'possible_categories' : {'conf : 1}, 'categorie' : 'conf}
2	Selon l'extension de l'artéfact	Data si l'extension est .csv, .txt, .h5	2	data/glove.txt	{'possible_categories' : {'data' : 2}, 'categorie' : 'data'}
		Modèle si l'extension est .npy, .pkl, .npz		results/final_model.npy	{'possible_categories' : {'model : 2}, 'categorie' : 'model}
		Conf si l'extension est .json, .ini, .yml		Conf/params.yml	{'possible_categories' : {'conf : 2}, 'categorie' : 'conf}

ANNEXE B ORGANISATION DES LIENS DE TRAÇABILITÉ

ANNEXE C RÉCAPITULATIF DES DÉFIS, CAUSES, SOLUTIONS ET RECOMMANDATIONS

	AN1	AN2	EC1	EC2	PC1	PC2	PC3	EA1	VE1	MA1	MA2
Challenge	Identifying and understanding project's workflow	Retrieving artifacts and their links	Configure hardware requirements	Complex language-level and system-level dependency management	Editing the project to match the platform's specifications	Configuring a multi-step workflow	Impact of the initial project's architecture on the migration	Lacking and overlapping of CI/CD features	Validating the result of runs	Evolution and versioning of platform-related artifacts	Inability to maintain history before the migration
Cause	ML and software artifacts are mixed up in the same project	No standard for artifact naming and organization	Specialised hardware requirements for ML apps	Incomplete language-level dependency management, no standard methodology for system-level dependency management	Strong coupling between pipeline files and ML artifacts	Platforms require specific organisation for workflow's steps	Pipelines need to be accurately broken into steps	Conflict between migration and CI/CD pipelines	complex and multiple output, results and validation criteria	Nonlinear complexity of configuration pipelines	no feature to copy the history from before the migration
Option	Record the project's ML workflow in a standardized and rigorous way	Reconstruct traceability links between artifacts	Apply dependency management to hardware requirements	Automated link between core and migration environment	Leverage organised data identified from the ML project	Use insights from other platforms and produce generic code for multi-step workflow execution	Consider the initial architecture of the project when choosing a target MLLC platform; break down the pipeline into phases		Leverage manual and computed tags and comments to track and record run results	Establish traceability links between ML and platform-related artifacts	Adaptation and standardization of evolution data between traditional versioning tools and MLLC platforms
Recommandation	Improve documentation with ML-related artifact information		Integrate specialised tools for hardware management in the project (like nvidia for Docker)	Break down the workflow into stages; avoid using literate programming for stage management	Break down the workflow into stages; avoid using literate programming for stage management			Hybrid platform combining ci/cd and ML management features	Configure validation to be performed by the tool	Automated tool to retrieve ML evolution	

ANNEXE D PROCESSUS DE GÉNÉRATION DU PIPELINE D'AM



ANNEXE E EXEMPLE DE RÉSULTAT DU FRAMEWORK MSR4ML


```
{
  "./data-preprocess.py": [
    {
      "io_method": "open",
      "lineno": 7,
      "artéfact_location": "./data/raw_stock.txt",
      "artéfact_type": "input",
      "weight": 1,
      "version": "1.0",
      "possible_categories": {"data": 1},
      "categorie": "data"},
    {"io_method": "to_csv",
      "lineno": 58,
      "artéfact_location": "./data/stock.csv",
      "artéfact_type": "output",
      "weight": 2,
      "version": "1.1",
      "possible_categories": {"data": 1},
      "categorie": "data"}],
  "./train.py": [
        {
      "io_method": "open",
      "lineno": 12,
      "artéfact_location": "./data/stock.csv",
      "artéfact_type": "input",
      "weight": 1,
      "version": "1.0",
      "possible_categories": {"data": 1},
      "categorie": "data"},
    {"io_method": "save",
      "lineno": 79,
      "artéfact_location": "./model/SVM.pkl",
      "artéfact_type": "output",
      "weight": 1,
      "version": "1.0",
      "possible_categories": {"model": 1},
      "categorie": "model"},
    {"io_method": "open",
```


Liens de traçabilité

Cette commande permet d'obtenir la liste des modifications ayant potentiellement affecté les deux dernières versions du modèle SVM.pkl (entre la version 1.0 (la dernière) et la version précédente). Il est possible de spécifier deux versions manuellement afin 'obtenir toutes les modifications pertinentes survenus entre ces deux versions.

```
msr4ml blame ./SVM.pkl -v latest
```


```
{'atefact': 'SVM.pkl',
 'versions': {
     'latest': '1.0',
     'previous': '0.9'
  }
 'blames': [
    {'artefact': './SVP.pkl',
     'categorie': 'model',
     'steps': ['train', 'test', 'deploy'],
     'commits': ['5ce6b9e'],
    'priority' : 'high'},
    {'artefact': 'train.py',
     'categorie': 'code',
     'steps': ['train'],
     'commits': ['f4daf1b', '9cg58z7', h5edo'],
     'priority': 'high'},
    {'artefact': '/data/stock.csv',
     'categorie': 'data',
     'steps': ['data-preprocess', 'train'],
     'commits': ['f4daf1b', '9cg58z7', h5edo'],
     'priority': 'high}]}
```


A partir des identifiants des commits, il est possible de consulter le message, l'auteur, la date et les modifications effectués sur chaque artéfact