UQAC
Université du Québec
à Chicoutimi

VERS L'ÉVALUATION DE LA STABILITÉ DES SYSTÈMES À L'AIDE DE LA DENSITÉ DES LOGS

PAR MARCELO MEDEIROS DE VASCONCELLOS

MÉMOIRE PRÉSENTÉ À L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI EN VUE DE L'OBTENTION DU GRADE DE MAÎTRE ÈS SCIENCES (M.SC.) EN INFORMATIQUE

QUÉBEC, CANADA

RÉSUMÉ

Les systèmes informatiques sont de plus en plus présents dans notre vie quotidienne. Pour augmenter leur disponibilité et leur fiabilité, les développeurs insèrent des instructions de journalisation dans le code source des systèmes, lesquelles génèrent des fichiers de journaux qui enregistrent des informations telles que l'état interne, les variables modifiées durant l'exécution, etc. Les administrateurs surveillent les systèmes en examinant ces fichiers de journaux et les développeurs utilisent cette base pour résoudre certains problèmes. Des recherches récentes ont montré que le nombre d'instructions de journaux (LPS) par ligne de code dans le fichier, également appelé densité des journaux (LOGD), est en relation directe avec la maintenabilité du système ; cela signifie que plus on trouve des LPS dans un fichier, plus sa densité de défauts après la publication est élevée. Cela s'explique par le fait que les développeurs ont tendance à inclure plus d'instructions de journalisation, et ont plus de questions et de préoccupations, à propos des portions du système qui sont davantage sujettes à des défauts. De nos jours, il existe des systèmes logiciels qui sont utilisés depuis longtemps et qui, par conséquent, ont été maintenus de façon constante pendant plusieurs versions. Cependant, le processus de décision pour la mise en œuvre de nouvelles versions manque d'indicateurs à propos de la stabilité de ces versions. À cet égard, la LOGD peut constituer une métrique qui peut indiquer la stabilité du code du système. Des recherches antérieures ont étudié la Densité des journaux (LOGD) sur plusieurs systèmes. Cependant, la densité n'a pas été étudiée au cours du développement des systèmes, entre leurs différentes versions, et on ne sait pas quelles sont les implications liées à l'évolution de la densité entre eux. L'objectif de cette étude est d'analyser l'évolution de LOGD au cours du cycle de développement d'un système open source, en particulier Hadoop. Comme première contribution, nous proposons d'utiliser la différence de LOGD entre les versions consécutives d'un logiciel comme indicateur de la stabilité du code. Nous avons évalué le code de 131 versions du projet Hadoop, développé par l'Apache Software Foundation (Apache Software Foundation (ASF)) de la version 0.1.0 à la version 3.3.5, en évaluant la LOGD au cours de l'évolution du système. La deuxième contribution est l'outil que nous avons développé pour effectuer le calcul et qui est disponible sous licence *open source* dans le but d'être utilisé pour d'autres recherches. Nous avons constaté que la LOGD subit de fortes variations au début du projet et qu'elle tend à diminuer au cours du développement du système. Nous concluons également que la différence entre les LOGD de versions consécutives d'un logiciel est un bon indicateur pour éviter la dépréciation du code de journalisation dans les systèmes.

ABSTRACT

Computer systems are increasingly present in our daily lives. To increase availability and reliability, developers insert log instructions into the source code of systems to generate log files that record information such as status and runtime variables, etc. System administrators monitor systems via log files and developers use the base to troubleshoot problems. Recent research has shown that the number of log instructions (LPS) per line of code in the file, also known as log density (LOGD), has a direct relationship with system maintainability, i.e., the more LPS in a file, the higher its defect density after release, because developers tend to include more files that have more issues and concerns and are therefore more prone to defects. Nowadays, there are software systems that have been in use for a long time and, therefore, are constantly maintained for several releases. However, the decision-making process for the implementation of new versions lacks indicators that reveal the stability of these versions. In this regard, one metric that can indicate the stability of the system code is LOGD. Previous research has studied LOGD between systems. However, density has not been studied during the development of systems, between their different versions, and what are the implications related to the evolution of density between them. The objective of this study is to analyze the evolution of LOGD during the development cycle of an open source system, in particular Hadoop. We propose to use the difference in LOGD between consecutive versions of a software as an indicator of code stability. We evaluated the code of 131 versions of the project Hadoop, developed by ASF from version 0.1.0 to version 3.3.5, evaluating the LOGD during the evolution of the system. The second contribution is the tool that we developed to perform the calculation and that is available under opensource license with the purpose of being used for further research. We found that the LOGD undergoes a strong variation at the beginning of the project and that it tends to reduce during the development of the system. This shows that with the maturity of the project. We also conclude that the difference between the LOGD between consecutive versions of a software is a good indicator to avoid the depreciation of the daily code in the systems.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ABRÉVIATIONS

ACL	Apache Commons Logging
ASF	Apache Software Foundation
AST	Abstract Syntax Tree
JUL	Java Utility Logging
LCP	Évolution du code de journal
LOGD	Densité des journaux
LOLC	Nombre de lignes de code des journaux
LP	Pratiques de journalisation
LPS	Instructions de journalisation
REGEX	Expressions régulières
SLOC	Total des lignes de code sans les commentaires
SQL	Structured Query Language
NLPSA	Nombre de LPS ajoutés
NLPSC	Nombre de LPS corrélés entre les versions
NLPSI	Nombre de LPS inchangés
NLPSM	Nombre de LPS mis à jour
NLPSS	Nombre de LPS supprimés
PLPSA	Pourcentage de LPS ajoutés
PLPSI	Pourcentage de LPS inchangés
PLPSM	Pourcentage de LPS mis à jour
PLPSS	Pourcentage de LPS supprimés

DÉDICACE

Je dédie ce travail à ma famille,
mes parents Francisco Américo de Vasconcellos et Ignês Medeiros de Vasconcellos,
mes filles Carolina Conrado de Vasconcellos et Cecília Conrado de Vasconcellos et
mon épouse, Liana Geisa Conrado Maia.
Merci pour le soutien et l'affection que vous m'avez offerts dans ce parcours.
Sachez que j'en suis très reconnaissant.

REMERCIEMENTS

Je tiens à remercier tous ceux qui m'ont soutenu pendant la réalisation de ma maîtrise et pendant la rédaction de ce mémoire. Je tiens à remercier également toute l'équipe du Département d'informatique et de mathématique (DIM) de l'UQAC pour tout l'attention dans les moments de doutes au long du projet. Plus particulièrement, j'aimerais remercier mes directeurs de recherche Sylvain Halle et Fabio Petrillo pour leur accompagnement et pour leur expertise précieuse offerte tout au long de ma maîtrise.

Je remercie également mes collègues du groupe de recherche *Software is My Art Software Enginnering* (SmArtSE) pour le partage de leurs connaissances, pour leur aide et leur amitié durant ces deux dernières années. Finalement, j'aimerais remercier ma famille et mes amis pour leurs encouragements tout au long de mes études.

CHAPITRE I
INTRODUCTION

1.1 CONTEXTE

Les systèmes informatiques sont de plus en plus présents dans notre vie quotidienne. En conséquence, leur surveillance est une tâche importante et non triviale. Pour accomplir cette tâche, les développeurs insèrent des instructions de journalisation (en anglais *log printing statement* ou LPS) dans le code source des systèmes ; ces déclarations génèrent ce que l'on appelle des messages de journal. Ces messages contiennent de nombreuses informations sur le comportement des systèmes logiciels pendant leur exécution. Les messages de journal sont couramment utilisés pour la surveillance des systèmes, le diagnostic des problèmes et la conformité aux réglementations [1]. L'importance de la journalisation est indéniable, car les informations d'exécution stockées dans les enregistrements sont utilisées par les développeurs à diverses fins, telles que : la détection des anomalies et des défaillances [2, 3, 4, 5], le diagnostic en ligne ou post-mortem des performances et des défaillances [6, 7, 8, 9, 10], la détection de modèles [11, 12, 13], le profilage [13], la prise de décision commerciale [14], la sécurité [15, 16] et l'observation du comportement de l'utilisateur [17].

Shang *et al.* [18] s'est penchée sur la relation entre les journaux de code source et les défauts des logiciels. Ils ont constaté que les fichiers de code contenant des lignes de journal ajoutées par les développeurs (LPS) présentaient davantage de défauts après la publication. Ils ont également identifié une corrélation positive entre les fichiers contenant des LPS et les défauts après la publication. L'étude n'a pas suggéré de supprimer les journaux, mais a recommandé d'allouer des ressources supplémentaires à la maintenance préventive pour les fichiers contenant plus de journaux et de LPS, car ces fichiers ont tendance à présenter plus de problèmes et donc une plus grande probabilité de défauts.

1.2 PROBLÉMATIQUE ET MOTIVATION

Les systèmes actuels ont une longue durée de vie, passant par plusieurs versions, y compris des versions préliminaires (alpha et beta). Ils font l'objet d'une maintenance constante et souvent réalisée par une grande équipe de développeurs. Le déploiement de nouvelles versions du système peut entraîner des défauts après la sortie. Par exemple, mentionnons les problèmes cités par Jennifer Mertens via le site "Belgium Iphone" dans son rapport intitulé *iOS 14 : un nouveau bug remet les applications d'Apple par défaut* [19].

Les décisions relatives au déploiement ou à la publication de nouvelles versions nécessitent des indicateurs qui révèlent la stabilité des systèmes, et nous pensons que les LPS capturent les préoccupations et les doutes des développeurs concernant le code. Les développeurs ont tendance à intégrer davantage de LPS pour suivre le comportement en cours d'exécution des points complexes et critiques du code. Dans la plupart des cas, les journaux sont utilisés pour résoudre les problèmes, de sorte que l'inclusion d'un plus grand nombre de lignes de journalisations dans un fichier de code source par un développeur peut être un indicateur que cette partie particulière du code est plus critique. En ce sens, la densité de journaux est une mesure qui peut indiquer la stabilité du code du système. Comme on l'a vu, des recherches antérieures ont d'ailleurs étudié la densité des logs (LOGD) à travers les systèmes [18].

1.3 OBJECTIF

Notre objectif est d'analyser la densité des journaux (LOGD), l'influence de l'ajout et de la suppression du code de journalisation et la manière dont les éléments d'un LPS influencent la valeur de cette métrique au cours de l'évolution d'un logiciel. Pour nous mener à notre objectif, nous avons développé l'hypothèse suivante :

Hypothèse : *La différence entre la densité des journaux (LOGD) de deux versions consécutives du même système est un indicateur de la stabilité du code des systèmes.*

Dans notre recherche, nous utilisons le concept de stabilité proposé par Wu *et al.* [20] qui est la capacité du système à faire face à de nouvelles exigences ou à des changements sans s'effondrer, ou sans changements majeurs. Ainsi, moins le code source est modifié, plus le code du système est stable.

Pour ce faire, nous avons divisé notre objectif en trois questions de recherche (QR) :

QR1 : Quelle est la tendance suivie par la densité des logs LOGD à mesure que le système évolue ? En répondant à cette question de recherche, notre objectif est de comprendre comment la LOGD varie au cours de l'évolution d'un logiciel. Cette question de recherche est importante pour comprendre la fiabilité et la stabilité de cette mesure.

QR2 : Comment les actions d'ajout, de suppression et de mise à jour des LPSs influencent-elles le changement de la LOGD ? En répondant à cette question de recherche, notre objectif est comprendre les principaux facteurs influençant les changements dans la LOGD du système pendant son évolution. Cette question de recherche est importante pour comprendre comment le développement du système et l'inclusion ou la suppression d'une instruction influence la stabilité de cette mesure.

QR3 : Quelles relations existent entre les éléments d'un LPS et la métrique LOGD d'un système ? En répondant à cette question de recherche, notre objectif est d'identifier les relations entre les éléments d'un LPS et la LOGD d'un système. Cette question de recherche est importante car elle nous permet de comprendre quels sont les éléments qui influencent le plus un changement dans la LOGD et comment ils l'influencent, s'il existe une relation entre les changements dans les éléments d'un LPS, ainsi que la manière dont ces éléments peuvent influencer la stabilité de cette mesure.

1.4 SOLUTION PROPOSÉE

Notre étude propose de surveiller la différence de LOGD entre les versions consécutives d'un système, au cours de l'évolution de son code fonctionnel, dans l'optique où l'utilisation de cette métrique peut être vue comme un support à la prise de décision avant le lancement ou le déploiement de nouvelles versions.

Pour atteindre notre objectif, nous avons développé un outil interactif pour analyser et surveiller l'évolution du code des journaux à travers la métrique de la densité des journaux (LOGD). Nous validons l'outil par son application dans une étude de cas basée sur le système Hadoop, une plateforme open source écrite en Java, en extrayant les LPS et en comparant les versions stables du système, et en présentant ces informations à travers une interface graphique intuitive.

Notre approche consiste à extraire les informations à travers un *pipeline* qui repère les LPS et leurs caractéristiques sémantiques et syntaxiques, extrait les méthodes et leurs caractéristiques à l'aide de la bibliothèque *Lizard* [21] et compare les versions consécutives du même projet à l'aide de la bibliothèque *Difflib* [22].

Les informations sont ensuite stockées dans une base de données et présentées à l'utilisateur via une interface graphique, permettant une analyse interactive, la recherche de LPS à travers des filtres et la visualisation d'extraits de code source.

1.5 RÉSULTATS ET CONTRIBUTIONS

Nous constatons que la différence de la LOGD entre une version d'un système et celle qui précède tend à diminuer au cours de l'évolution du code de ce système. Cette mesure peut être utilisée pour évaluer l'exhaustivité et la stabilité du code, car il s'agit non seulement d'une

relation entre le code des journaux et le code de fonctionnalité, mais aussi d'une relation entre deux versions consécutives du même système.

Notre outil est disponible pour le développement et l'utilisation et peut être téléchargé en ligne [1].

1.6 ORGANISATION

Le reste du mémoire est structuré comme suit. Dans le chapitre 2 (Cadre théorique), nous présenterons la base théorique nécessaire à une bonne compréhension de notre proposition. Nous présenterons les concepts liés aux bibliothèques de journalisation, les défis liés à aux pratiques de journalisation. Nous détaillerons la mesure de LOGD, et nous examinerons quelques études liées à l'extraction des LPS. Nous terminerons en présentant quelques études liées cette fois-ci à la pratique de journalisation.

Dans le chapitre 3 (Implémentation de l'approche), nous expliquerons en détail la mise en œuvre de la solution proposée, présenterons des informations sur l'outil et les étapes réalisées par notre *pipeline* pour l'extraction d'informations à partir du code source d'un projet quelconque.

Dans le chapitre 4 (Validation) nous décrirons l'étude de cas, présenterons la préparation et la collecte d'informations étape par étape et détaillserons les résultats de nos questions de recherche. Nous répondrons ensuite aux questions de recherche énoncées en début de mémoire et présenterons quelques observations sur les difficultés rencontrées au cours du développement de la recherche.

Enfin, nos contributions, les limites de notre approche et les perspectives de travaux futurs sont discutées dans le chapitre 5 (Conclusion).

1. https://github.com/Log-Severity-Level/sloganalyzer

CHAPITRE II

CADRE THÉORIQUE

Dans ce chapitre, nous décrivons les concepts de base pour comprendre cette étude. Dans la section 2.1 nous présentons les concepts liés aux bibliothèques de journaux. Dans la section 2.5 nous mettons en évidence les principales utilisations de journalisation. Dans la section 2.3 nous mettons en évidence les défis liés à la *LPS*. Enfin, dans la section 2.6 nous présentons les travaux académiques antérieurs visant à comprendre les pratiques de journalisation.

2.1 BIBLIOTHÈQUES DE JOURNALISATION

L'enregistrement d'informations pendant l'exécution du système est une information précieuse qui peut être utilisée à diverses fins. Un LPS comporte trois parties : un message statique qui décrit le contexte, les variables qui fournissent des informations sur le contexte d'exécution et le niveau de sévérité [23]. La figure 2.1 montre un exemple de LPS dans le code et l'enregistrement correspondant qui est généré au moment de l'exécution.

2.2 ÉVOLUTION DES BIBLIOTHÈQUES DE JOURNALISATION

Selon Kabinna *et al.* [23], l'histoire des bibliothèques de journaux peut être divisée en quatre époques, elles sont :

- **Journalisation *ad hoc* :** Au début, les développeurs s'appuyaient principalement sur des méthodes d'impression telles que *System.out.println* ou *System.err.println* dans Java ou *syslog* dans Linux pour générer des journaux et surveiller les projets. Cependant, ces fonctions souffraient de l'absence de niveaux de sévérité, de difficultés de configuration et de maintenance et d'un manque de normalisation.

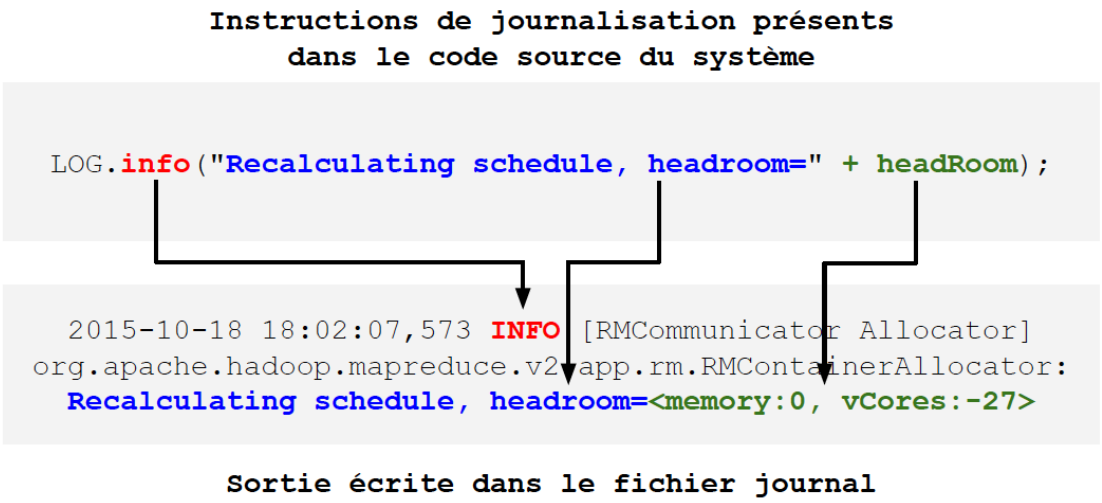

FIGURE 2.1 : Exemple de LPS et de la sortie écrite dans le fichier du journal
©Marcelo Medeiros de Vasconcellos, 2023

- **Bibliothèques de base :** À cette époque, des bibliothèques de base sont apparues, introduisant par le fait même des niveaux de sévérité pour les instructions de journalisation (exemple : *Error*, *Fatal*, *Warn*, *Info*, *Debug* et *Trace*). La configuration du processus a également a été facilitée, car ces bibliothèques pouvaient être configurées pour l'ensemble du projet via un seul fichier. D'autres éléments de configuration ont pris de l'importance, tels que : date, heure et emplacement de sortie. Parmi les documents de bibliothèque de cette époque que nous pouvons mentionner figurent *Log4j*, sorti en 2001, et Java Utility Logging (JUL), en 2002. Au fur et à mesure que le nombre de bibliothèques augmentait, les développeurs ont rencontré des difficultés à utiliser différentes bibliothèques dans le même projet.
- **Bibliothèques d'abstraction :** Afin de faciliter l'utilisation de plusieurs bibliothèques dans un grand projet, des abstractions de bibliothèque ont été développées. La première bibliothèque qui prenait en charge l'abstraction de journal était Apache Commons Logging (ACL) en 2003, où les développeurs écrivaient des déclarations au format ACL avec la possibilité d'appeler n'importe quelle autre bibliothèque dans le *back-end*. Une

autre bibliothèque d'abstraction de journal similaire à ACL est *Slf4j*, qui imposait une surcharge de performances inférieures car elle avait une forme paramétrisée de LPS qui aidait le développeur à écrire du code plus propre et ne souffrait pas de problèmes de chargement de classe. Cependant, le principal inconvénient de ces solutions était qu'elles avaient besoin d'autres bibliothèques pour générer les journaux, et que la configuration de cette interaction était difficile.

- **Bibliothèques d'unification d'enregistrement :** Ces bibliothèques contenaient le meilleur des deux mondes car elles fournissaient à la fois la bibliothèque de base et la bibliothèque d'abstraction. La première bibliothèque de cette période, pour Java, était *Logback*, en 2011, qui contenait une bibliothèque de base et une bibliothèque d'abstraction intégrées. De son côté, *Log4j 2* intègre toutes les fonctionnalités de *Logback* avec les avantages de mieux performer et d'être agréé par l'ASF.

2.3 DÉFIS LIÉS À LA PRATIQUE DE JOURNALISATION

Selon Gholamian *et al.* [24], en raison de l'absence de normes et de directives bien acceptées [25, 26, 27], les développeurs s'appuient sur leur expérience et leur intuition pour écrire des LPS. Ce processus manuel d'insertion d'instructions dans le code source des systèmes, nous révèlent quatre défis majeurs : où enregistrer, que faut-il enregistrer, comment enregistrer et quand enregistrer.

2.3.1 OÙ ENREGISTRER?

Les LPS peuvent être insérées dans plusieurs emplacements du code, et les développeurs doivent donc décider où pendant le développement. L'enregistrements d'une ligne de log dans un fichier est une tâche intensive d'E/S et un enregistrement excessif peut entraîner une augmentation des coûts de maintenance et une diminution de la performance [28, 29, 30].

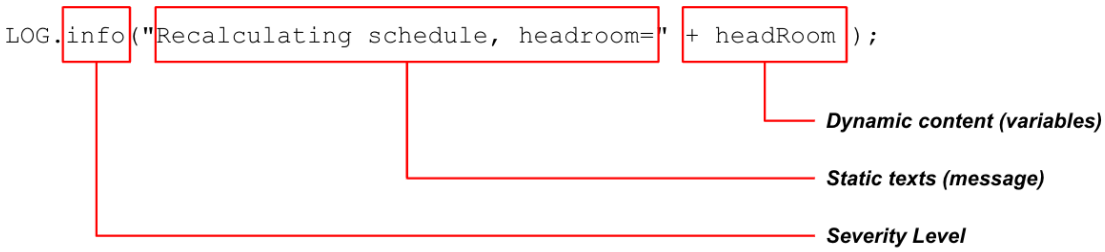

FIGURE 2.2 : Exemple de *Log printing statement* et de ses éléments
©Marcelo Medeiros de Vasconcellos, 2023

Selon Gholamian *et al.* [24], la recherche dans ce domaine s'est intéressée à identifier les points d'enregistrement appropriés. L'une des approches utilisées est l'analyse du code source à la recherche des caractéristiques des blocs de code. Yuan *et al.* [31] développent une recherche en identifiant les blocs de code d'exception non enregistrés et en y insérant automatiquement des LPS. D'autres recherches dans ce domaine sont axées sur le journalisation, telles que les chemins d'exécution sans ambiguïté [28], la minimisation des E/S et les coûts généraux de performance [30], et les approches d'extraction et d'apprentissage des ressources [32, 26].

2.3.2 QUE FAUT-IL ENREGISTRER?

Ce défi réside dans ce qui devrait être inclus dans le journal. Le message inséré et ses variables doivent être clairs et informatifs sur l'état actuel du système. La solution consiste à fournir suffisamment d'informations sur les trois éléments d'un LPS :

- **Niveau de sévérité :** Le niveau de sévérité indique si l'enregistrement doit, ou non, être émis pendant l'exécution du système. Il est important de choisir un niveau approprié, car ceux qui sont choisis par erreur peuvent ne pas être émis, ou s'ils le sont, ils peuvent être négligés ; ils peuvent aussi provoquer une surcharge. Cette négligence ou cette surcharge peut avoir un impact négatif sur l'expérience du client et affecter la qualité du produit.

Mendes *et al.* [33] ont étudié les principales bibliothèques de journaux utilisées dans 15 langages de programmation et ont proposé des règles sur le niveau à utiliser dans tel ou tel cas. Leur enquête a permis d'identifier 19 niveaux de sévérité parmi 40 bibliothèques existantes, à savoir : *Finest*, *Verbose*, *Finer*, *Trace*, *Debug*, *Basic*, *Fine*, *Config*, *Info*, *Success*, *Notice*, *Warn*, *Error*, *Fault*, *Severe*, *Critical*, *Alert*, *Fatal* et *Emerg*.

Dans le travail développé par Mendes *et al.* [33] ils ont fait une synthèse et ont abouti à six niveaux de sévérité principaux, leurs concepts suivent ci-dessous :

(a) **Debug** : responsable de l'enregistrement des états variables et des détails sur les événements intéressants et les points de décision dans le flux d'exécution d'un système logiciel.

(b) **Trace** : responsable de l'enregistrement du suivi des états variables et des événements dans un système logiciel.

(c) **Info** : responsable du registre des événements normaux, informant de la progression attendue et de l'état d'un système logiciel.

(d) **Warning** : responsable de l'enregistrement des situations potentiellement dangereuses causées par des événements et des états inattendus.

(e) **Error** : responsable du registre de l'apparition d'un comportement inattendu d'un système logiciel.

(f) **Fatal** : responsable du registre des événements critiques qui conduisent un système logiciel à la défaillance.

- **Textes statiques (message) :** Les textes statiques doivent décrire le contexte du document d'une manière lisible par l'humain. Actuellement, les développeurs sont chargés de composer manuellement les textes statiques. Des textes statiques mal rédigés ou obsolètes peuvent être une source de confusion et avoir un impact sur les différentes tâches d'analyse des journaux [34]. Shang [35] a identifié cinq catégories d'informa-

tions que les praticiens cherchent souvent à obtenir dans les messages du journal : la signification, la cause, le contexte, l'impact du message du journal et la solution au message du journal. De plus, selon lui, les lignes de journal peuvent être améliorées en ajoutant un contexte ou une solution.

- **Contenu dynamique (variables) :** Le contenu dynamique reflète l'état des systèmes pendant l'exécution et est le résultat de l'exécution des variables et des invocations de méthodes. Il est important d'enregistrer les informations d'exécution nécessaires afin de satisfaire les différents besoins de journaux des développeurs et des opérateurs. La recherche dans ce domaine cherche à s'améliorer en incluant automatiquement les valeurs des variables accessibles [36].

2.3.3 COMMENT ENREGISTRER?

Ce défi concerne la manière dont le code du journal, en tant que sous-système, se combine avec le reste du système logiciel. Les bibliothèques de journalisation et les utilitaires permettent d'organiser les enregistrements et améliorer leur formatage et leur qualité. Ceci étant dit, l'amélioration des bibliothèques de journalisation peut avoir un impact positif sur la manière dont on enregistre [37].

Dans une grande partie des systèmes, le code de journalisation est imbriqué dans différents modules du code source. Certains chercheurs suggèrent de modulariser le code des journaux en tant que sous-système indépendant, mais de nombreux logiciels ont encore tendance à mélanger le code de fonctionnalité et le code des journaux, ce qui constitue un défi de taille pour maintenir un code des journaux de haute qualité à mesure que le code de fonctionnalité évolue. Ce défi est lié à l'interaction entre le code des journaux et le code de fonctionnalité [24].

2.3.4 QUAND ENREGISTRER?

Ce défi, dont la plus récent étude est celle de Mizouchi [38], discute de la question de savoir si une déclaration doit être stockée, ou non, en effectuant un ajustement du niveau de sévérité nominal en réponse aux caractéristiques d'exécution. De mauvaises pratiques en matière d'enregistrement causent des problèmes tels que des absences ou des surcharges du fichier de journal. Une surcharge excessive de journaux ou d'informations peut générer beaucoup de bruit et nuire à la détection des erreurs, ainsi qu'engendrer des dépenses de stockage, de développement et de maintenance supplémentaires. Au contraire, l'absence de LPS peut également rendre difficile la détection et la correction des défaillances.

Ce défi est directement lié à la capacité d'adaptation de l'ajustement du degré de verbosité du journal. Par exemple, si une anomalie est détectée, le système devrait permettre un journal plus détaillé, et si le système fonctionne normalement, il peut minimiser le volume de journalisation [24].

En résumé, ces défis associés aux codes de journaux ont suscité des recherches visant à analyser et à exploiter les LPS, leurs Évolution du code de journal (LCP) et leurs problèmes, aboutissant au développement d'outils et d'approches pour prédire et suggérer automatiquement les LPS à insérer et leurs détails associés, tels que : les niveaux de sévérité, les messages statiques et les variables. Dans notre travail, nous nous concentrerons sur les deux défis suivants : « Que faut-il enregistrer ? » et « Où enregistrer ? ».

2.4 DENSITÉ DES JOURNAUX (LOGD)

On peut calculer diverses mesures à propos des LPS et leur relation par rapport au code du programme lui-même. Parmi les mesures, on peut mentionner le nombre de variables déclarées dans LPS, le contexte de LPS, le nombre de lignes de un LPS, le nombre de LPS

lui-même, le nombre de lignes de code dans le fichier (LOLC), le nombre de LPS modifiées, le nombre de mots dans le message statique, et le niveau de sévérité, entre autres. Dans ce qui suit, nous allons travailler avec la densité des logs (LOGD).

La LOGD mesure la présence du code de journalisation [1, 39, 40]. Elle est calculée en divisant le SLOC par le LOLC. Cette mesure est importante car elle indique dans quelle proportion un système est couvert par des LPS. Lors du calcul du SLOC et du LOLC, seules les lignes de code sont comptées, et les commentaires et les lignes vides sont exclus.

$$LOGD = \frac{SLOC}{LOLC} \tag{2.1}$$

Shang *et al.* [18] ont étudié la relation entre les caractéristiques du registre et les défauts des logiciels. Ils ont constaté que les fichiers de code source avec des LPS ont une densité de défauts plus élevée après la publication que ceux qui n'ont pas de LPS. Ils ont également constaté qu'il existe une corrélation positive entre les fichiers de code source contenant des lignes de journal ajoutées par les développeurs et les fichiers de code source présentant des défauts après la publication, et que les mesures liées au journal complètent les mesures traditionnelles du produit et du processus pour expliquer les défauts après la publication. Dans leur article, il ne suggèrent pas de supprimer les journaux, assurant que la journalisation est essentielle pour comprendre et surveiller les défauts. Il suggèrent plutôt aux développeurs d'allouer davantage de ressources à la maintenance préventive des fichiers contenant plus de journaux et de modifications des LPS, car les développeurs ont tendance à inclure plus de LPS dans les fichiers contenant des questions et des préoccupations et qui sont donc plus susceptibles d'être défectueux.

2.5 EXTRACTION DES INSTRUCTIONS DES JOURNAUX (LPS)

Beaucoup de recherches sont apparues dans le but d'exploiter, de comprendre et de caractériser les différentes pratiques de journalisation (*logging practices* ou Pratiques de journalisation) [25, 26, 41, 42]. La compréhension des pratiques employées constitue un point d'entrée pour aider les développeurs à améliorer leurs habitudes d'extraction actuelles. Pour ce faire, la première étape consiste donc à extraire les LPS du code source.

Il existe deux classes de Pratiques de journalisation (LP), qui sont : l'exploration et le forage des LPS directement dans le code source, ainsi que l'exploration et le forage des fichiers de journaux générés par le système lors de son exécution. Dans notre travail, nous nous concentrerons sur la premiere d'entre elles.

Selon Gholamian *et al.* [24], l'exploration et le forage des LPS peuvent être divisés en trois principaux domaines d'utilisation. Un premier domaine se concentre sur les LP. Ce domaine vise à obtenir un aperçu des habitudes de journalisation des développeurs. Les travaux dans ce domaine comprennent des études sur la caractérisation des pratiques actuelles [41, 42, 43], la recherche d'erreurs récurrentes dans le code de journaux et sa relation avec la qualité du code [40, 42] et la vérification des bibliothèques, des paramètres et de leurs utilitaires [37, 44].

Un deuxième domaine s'intéresse à la LCP. Ce domaine étudie l'évolution des LPS dans le code source, à mesure que ces déclarations changent au fil du temps. Certaines études concluent que le code des journaux évolue de manière significative au fil du temps, et présente un taux de rotation élevé, parfois supérieur à la modification du code de fonctionnalité, tout au long du développement logiciel [25, 28, 39, 45]. À ce stade, nous pouvons également observer des travaux qui évaluent les migrations de bibliothèques au cours de la durée de vie du projet [23] et d'autres qui proposent des outils pour prédire les révisions probables du code

de journaux, par exemple, *LogTracker* [46]. Li *et al.* [47] proposent de guider les révisions des instructions de journalisation, en tirant les leçons de leur évolution passée, et en supposant que le code de journaux avec un contexte similaire mérite des modifications similaires.

Un troisième domaine étudie la détection des problèmes liés aux (LP). L'ajout intensif de LPSs dans un fichier peut révéler des erreurs et des LPs inappropriées qui entraînent des problèmes d'enregistrement et résulter en des journaux de mauvaise qualité. Certaines recherches dans ce domaine englobent à la fois les LPs et les LCP, au fur et à mesure que des questions se dégagent lors de l'examen des pratiques et de leur évolution. Yuan *et al.* [48] a présenté une étude sur les défaillances du monde réel dans les systèmes distribués, et a noté que la plupart des défaillances produisent des messages de journal liés à des défauts explicites qui peuvent être utilisés pour les reproduire. Cependant, les journaux sont bruyants, ce qui rend leur analyse fastidieuse. Des efforts ont été menés pour identifier et réduire ces problèmes, par exemple en proposant des méthodes pour trouver des erreurs récurrentes d'enregistrements (anti-modèles) [25], pour proposer des modifications et pour ajuster le niveau de sévérité, ajouter ou soustraire des variables ou modifier des textes statiques [41], identifier des doublons d'enregistrements [49, 50], des messages inappropriés [51] ou même l'absence de LPs [51, 28]. La détection de ces problèmes est utile car les développeurs peuvent appliquer des révisions aux LPs et donc améliorer les LPs. Des outils ont été développés pour la détection automatique des problèmes liés aux journaux, tels que *DLFinder* [50] et *LCAnalyzer* [25].

2.6 ÉTUDES SUR LA PRATIQUE DE JOURNALISATION (LP)

Yuan *et al.* [52] ont effectué une étude sur la caractérisation des LPs dans les systèmes open-source en C/C++. Dans leur étude, ils ont présenté un aperçu des modifications dans les journaux, comprenant les modifications des niveaux de sévérité, les modifications dans les variables, les modifications dans le contenu statique et les modifications concernant

l'emplacement des déclarations. Ils ont conclu que les développeurs consacrent des efforts importants à la modification du niveau de sévérité, du texte statique et des valeurs des variables des messages de journal, mais qu'ils modifient rarement l'emplacement des messages. L'une des principales conclusions de leur étude est que parmi les logiciels analysés, en moyenne, on trouve une ligne de code de journalisation toutes les 30 lignes de code.

Shang *et al.* [45] ont réalisé une étude de cas afin d'explorer le LCP des LPS dans deux projets open source et un projet industriel. Ils observent que le code du journal change à un taux élevé entre les versions, ce qui peut entraîner une rupture de la fonctionnalité des applications de traitement du journal. Ils suggèrent également que la plupart des changements de code de journalisation pourraient être évités.

Shang *et al.* [40] ont présenté une étude sur la caractérisation des pratiques actuelles de journalisation et leur relation avec la qualité du code. Cette étude a été menée sur quatre versions de Hadoop et JBoss. Ils ont défini des métriques de qualité de code liées à la journalisation, telles que : le nombre de lignes de journal dans un fichier, le niveau moyen de journalisation, le nombre moyen de lignes de journaux ajoutées ou supprimées dans une action de type *commit*, et la fréquence des changements de code pour corriger les défauts en lien avec la rotation des journaux. Leurs résultats ont montré que les caractéristiques des journaux fournissent des indicateurs forts des fichiers de code source sujets aux défauts. Leurs études étaient basées sur l'historique des versions *gits*. Ils ont choisi d'utiliser l'extraction des LPS au moyen l'arbre syntaxique du code source (Abstract Syntax Tree (AST)) et, en raison de ce choix, ils ne pouvaient obtenir des informations que sur le nombre de LPS et ne pouvaient pas obtenir d'informations sur le nombre de lignes de code. Selon les auteurs, la qualité des données contenues dans les dépôts peut avoir un impact sur la validité interne de leur étude.

Chen *et al.* [1] ont appliqué aux projets de l'Apache Software Foundation (ASF) la même étude que celle réalisée par Yuan *et al.* [52] sur les projets C/C++ et a constaté que

les LPS sont activement maintenus et qu'une grande partie des mises à jour des journaux sert à améliorer la qualité des LPS (par exemple, le formatage, les changements de style et la correction de l'orthographe et de la grammaire) plutôt que de co-modifier les implémentations de fonctionnalités (par exemple, la mise à jour des noms de variables). Cependant son étude était comparative entre des catégories de systèmes différents, et n'étudiait pas l'évolution du code d'un même système. Ainsi, dans le cas de Hadoop, il ont étudié la LOGD uniquement dans la version 2.6.0. Les auteurs ont utilisé *J-REX* [53] et *ChangeDistiller* [54] pour recueillir ces données et ont développé un script basé sur des expressions régulières pour compter les LOLC.

Finalement, Kabinna *et al.* [39, 23] ont eux aussi étudié les LCP dans les projets de l'ASF et constaté qu'une grande quantité de LPS change pendant la durée de vie du projet. Ils ont discuté des facteurs affectant la stabilité d'un LPS, et ont constaté que la propreté des fichiers, l'expérience des développeurs, la LOGD et le SLOC sont des paramètres importants pour déterminer la stabilité du LPS.

CHAPITRE III
IMPLÉMENTATION DE L'APPROCHE

Dans ce chapitre, nous présenterons notre approche en détail. Dans la section 3.1 (Approche et fonctionnalités), nous décrivons notre proposition et les fonctionnalités de l'outil. Dans la section 3.2 (Architecture du système), nous détaillons l'architecture de notre système. Dans la section 3.3 (Architecture des données), nous détaillons le modèle de données utilisé dans notre système, en décrivant les principales classes que l'on y retrouve. Dans la section 3.4 (Interface graphique), nous présenterons l'interface graphique, ainsi que certaines fonctionnalités présentes dans cette interface et certaines options de filtrage et de recherche. Dans la section 3.5 (Extraction des données), nous présenterons en détail notre pipeline, incluant les étapes d'extraction des LPS allant du clonage des repos à l'extraction du LPS, l'extraction des métadonnées des méthodes, l'intégration des LPS avec les méthodes et la comparaison entre les versions consécutives du système afin d'acquérir des connaissances concernant le LCP.

3.1 APPROCHE ET FONCTIONNALITÉS

Le *sLogAnalyser* est un outil développé en langage Python, doté une interface graphique, et qui permet l'extraction de données liées aux LPSs à partir du code source d'un projet quelconque.

Notre outil permet d'abord d'extraire les LPS à partir du code source du système. L'outil permet de choisir des dépôts Git, importer des *releases* de ces dépôts, ainsi que reconnaître automatiquement la séquence de création du dépôt en fonction du nom de la version ; il peut également exécuter les différents *pipelines* de traitement de données sur le code source récupéré. Enfin, il permet de rechercher et filtrer les données enregistrées dans sa base de

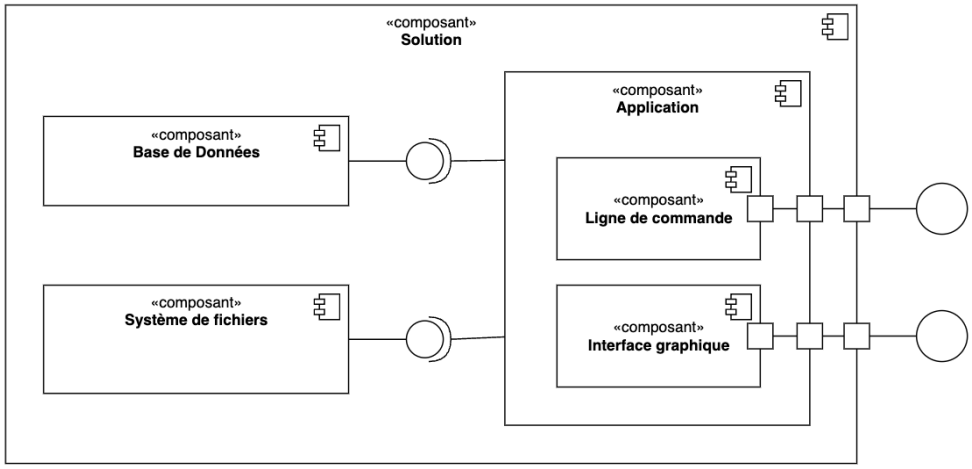

FIGURE 3.1 : Architecture - Diagramme des composants
©Marcelo Medeiros de Vasconcellos, 2023

données interne. L'accès à l'interface graphique permet d'effectuer des recherches et de consulter des données à travers une interface intuitive.

3.2 ARCHITECTURE DU SYSTÈME

Notre solution est structurée sur la base d'une architecture de composants avec une caractéristique de couplage faible, comme nous pouvons le voir dans le diagramme des composants de la figure 3.1. Dans ce qui suit, nous énumérons les principaux composants et décrivons leur fonction.

- **Composant de service de domaine :** Ce composant est responsable de l'extraction des données des fichiers. C'est par l'intermédiaire de ce composant que le système effectue la collecte des données et l'exploration des fichiers dans les dépôts Git.
- **Application :** Ce composant représente la partie web logique du système, à travers laquelle l'utilisateur effectue les fonctions d'extraction et de visualisation des résultats.

- **Composant ligne de commande :** Ce composant regroupe les fonctions qui sont exécutées sur les lignes de commande et les pipelines qui effectuent l'exploration des données. Pour ce faire, il interroge le service de domaine et stocke les données récupérées dans la base de données interne de l'outil.
- **Interface utilisateur :** C'est à travers ce composant que l'utilisateur enregistre les projets à analyser et leurs différentes versions. Il permet aussi la visualisation des données et le filtrage des résultats de l'opération.
- **Composant base de données :** Ce composant gère la base de données relationnelle qui prend en charge le stockage des données des journaux ainsi que les données extraites des projets après extraction.
- **Composant système de fichiers :** Ce composant représente le système de fichiers qui stocke les fichiers des dépôts Git.

3.3 ARCHITECTURE DES DONNÉES

L'outil proposé doit permettre la gestion de l'extraction des données à différents niveaux de granularité et selon différentes dimensions : système entier, branches, fichiers, messages, niveaux de sévérité, LPS et leurs paramètres associés. Notre solution contient donc une base de données relationnelle qui est chargée de stocker les données extraites des fichiers source analysés. Dans la figure 3.2, nous présentons le diagramme relationnel de la base de données. Nous soulignons que ce diagramme représente seulement les principales tables métier du système, et que les tables responsables du contrôle d'accès et de l'authentification des utilisateurs sont exclues de celui-ci.

Les principales tables de la base de données sont les suivantes.

- **System :** Responsable de la maintenance des données du système et de son adresse dans la plateforme GitHub. Elle stocke également l'expression régulière (« regex ») qui sera

utilisée pour filtrer les *tags* des dépôts qu'il souhaite cloner. Nous verrons plus en détail l'utilisation de cette regex dans la section 3.4.

- **Branch :** Responsable de la maintenance des données de *releases* du système dans les dépôts. Cette table contient les données relatives à la quantité de LPS, à la quantité de fichiers, à la quantité de méthodes et à leur complexité cyclomatique, toutes liées à un *release* spécifique.
- **SeverityLevel :** Responsable du stockage des données relatives au niveau de sévérité des enregistrements et de l'ordre de ces niveaux en fonction de leur sévérité. Les enregistrements du niveau de sévérité sont effectués automatiquement en fonction de la récupération des données.
- **File :** Responsable du stockage des données d'un projet particulier. Les noms de fichiers sont toujours stockés avec leur chemin relatif vers le répertoire de la *release*, de sorte qu'il est possible d'identifier le même fichier dans différentes versions du système. Cette table contient également le nombre de systèmes et de versions, ainsi que le nombre de LPS auxquelles ce fichier est lié.
- **Method :** Responsable du stockage des informations sur les méthodes du système, cette table permet d'identifier dans quelle méthode se trouve un LPS, ainsi que d'identifier la complexité de la méthode. Ce tableau contient également des données telles que le nombre de niveaux de sévérité différents présents dans la méthode, le nombre de LPS, le nombre d'instructions conditionnelles, le nombre d'instructions de capture/-blocage (`try/catch`), le nombre de boucles, le nombre d'instructions de continuation (`continue`) et le nombre d'instructions de rupture (`break`).
- **Message :** Responsable du stockage des messages uniques observés dans les LPS. Seuls les messages texte sont stockés, à l'exclusion des variables, qui sont remplacées par "{}". Parmi les principales informations stockées dans cette table, on peut citer le nombre

de variables contenues dans un message, les fichiers où ils apparaissent, les niveaux de sévérité, les versions et les systèmes auxquels chaque message est associé.

- **LogStatement :** Responsable du stockage des données relatives aux LPS. Cette table stocke des données sur les LPS, telles que : messages, fichier, ligne, niveau de sévérité et nombre de variables. Cette table contient également des données relatives à l'emplacement des LPS ; par exemple, s'il s'agit d'une instruction conditionnelle, d'une instruction en boucle, d'une instruction de capture/blocage et/ou si elle est associée à une instruction `break` ou `continue`.
- **BranchComparison :** Responsable du stockage des données des versions séquentielles d'un système, c'est-à-dire la séquence de création des versions de ce système. Grâce à cette table, notre outil peut comparer les données de deux versions successives.
- **BranchFile :** Responsable du stockage des données relatives à chaque fichier de chaque version. Si un fichier existe dans plus d'une version d'un même système, il apparaître donc dans plus d'une ligne de cette table ; chaque enregistrement aura son propre SLOC et son LOLC.
- **BranchFileComparison :** Responsable du stockage des données comparatives entre deux versions séquentielles, cette table stocke les données relatives entre autres aux changements dans les niveaux de sévérité, les messages et les déclarations.
- **MethodStatement :** Il est important de mentionner que chaque déclaration peut être liée à plus d'une méthode, car les méthodes peuvent être insérées dans d'autres méthodes. Cette table est chargée de stocker la relation entre chaque LPS et toutes les méthodes dans lesquelles cette déclaration est insérée. De cette façon, nous pouvons identifier la méthode parente la plus proche par la plus petite différence entre le numéro de ligne de départ de la méthode et le numéro de ligne de départ de la déclaration, et la méthode super parente, comme la méthode dont la différence entre le numéro de ligne de la méthode et le numéro de ligne de la déclaration est la plus grande.

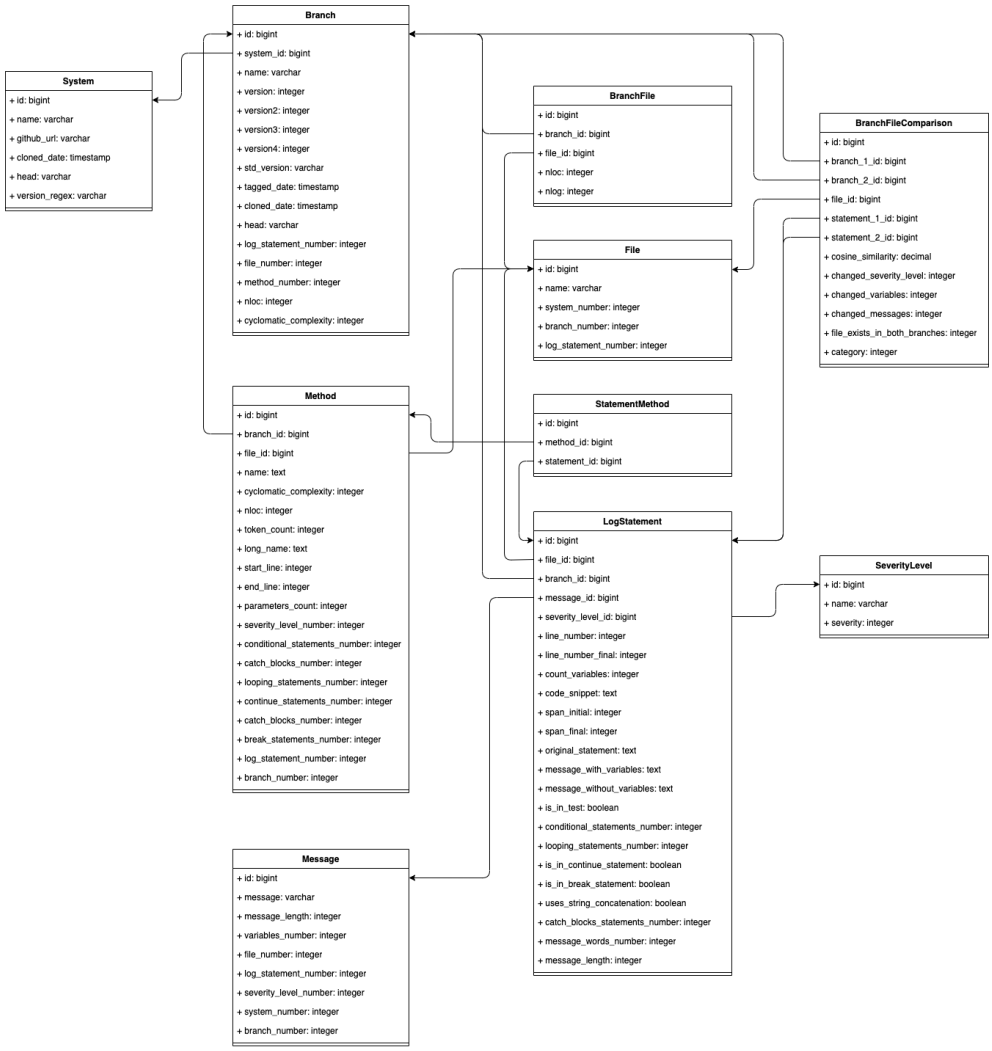

FIGURE 3.2 : Diagramme du schéma relationnel de la base de données
©Marcelo Medeiros de Vasconcellos, 2023

3.4 INTERFACE GRAPHIQUE

L'interface graphique du système permet à l'utilisateur d'effectuer plusieurs recherches liées aux données extraites. La page initiale, représentée à la figure 3.3, présente un portrait d'ensemble qui informe la quantité de systèmes, les versions, les fichiers uniques, les niveaux de sévérité, les LPS et le nombre de messages statiques, ainsi qu'un tableau dynamique qui permet de filtrer par version et par système.

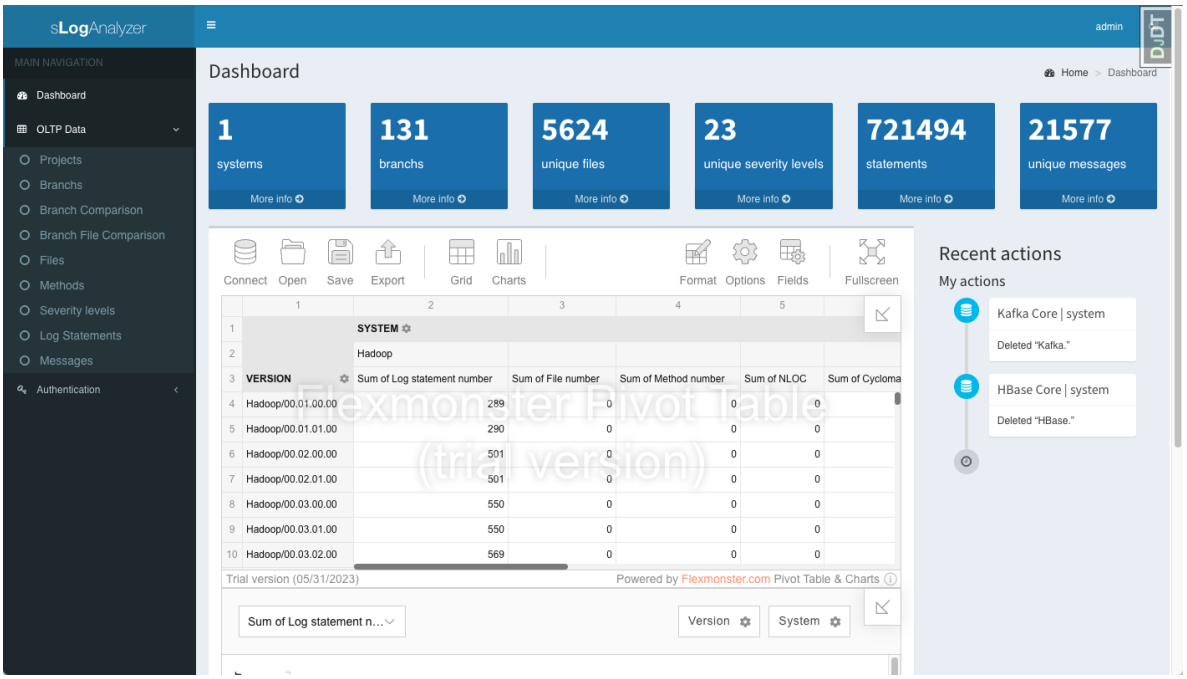

FIGURE 3.3 : Interface utilisateur - Écran de la liste des systèmes
©Marcelo Medeiros de Vasconcellos, 2023

En accédant à la page système, représentée à la figure 3.4, l'utilisateur sera en mesure d'enregistrer de nouveaux projets et même d'importer automatiquement les tags du projet en question à partir de la plateforme GitHub. La seule page où un enregistrement manuel est autorisé est la page système, dans toutes les autres pages l'enregistrement est effectué automatiquement pendant l'exécution des *pipelines*.

Dans l'écran de la liste des sytèmes, il existe un champ appelé « Version detection regex », illustré dans la figure 3.6. Dans ce champ, lors de l'enregistrement du système, l'utilisateur doit insérer une regex pour filtrer les versions à analyser. Par exemple, dans notre cas, nous utilisons la regex représentée à la figure 3.5 pour filtrer les versions dont le nom obéit aux patrons suivants : « release-X.X.X » et « rel/release-X.X.X », où les X sont des valeurs numériques. Notez que les variables v1, v2, v3 et v4 doivent être insérées dans la regex, v3 et

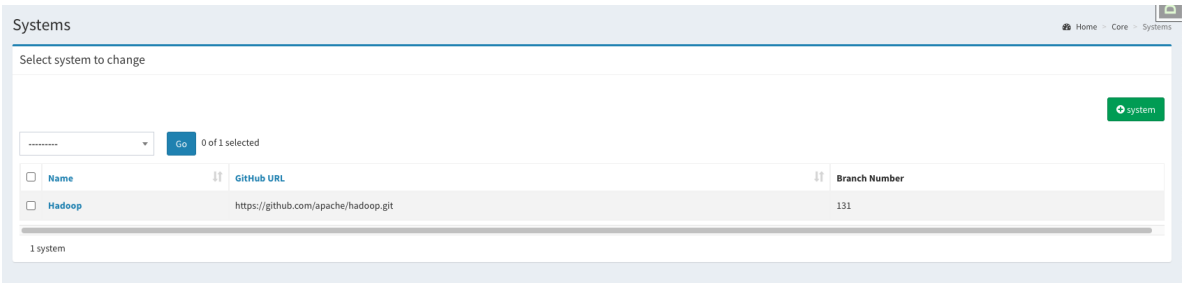

FIGURE 3.4 : Interface utilisateur - Écran de la liste des systèmes
©Marcelo Medeiros de Vasconcellos, 2023

```
^(rel/release-|release-)(?P<v1>\d+)\.(?P<v2>\d+)\.(?P<v3>\d+)$
```

FIGURE 3.5 : Exemple de regex utilisée pour filtrer les versions à cloner
©Marcelo Medeiros de Vasconcellos, 2023

v4 n'étant pas obligatoires. Ces variables seront utilisées par le pipeline de comparaison des versions pour identifier l'ordre des versions du système.

Pour faciliter l'explication de la Expressions régulières (REGEX) de la figure 3.5, nous la diviserons en deux parties, à savoir le début, représenté par "`(rel/release-|release-)`" et l'extraction des variables, représentée par les elements "`(?P<v1>\d)+`", "`(?P<v2>\d)+`" ou "`(?P<v3>\d)+`".

Dans la première partie de REGEX, on peut observer le signe '`^`' représentant le début de la phrase et le contenu délimité par les parenthèses. Chaque ensemble entre parenthèses est appelé groupe. Le groupe délimite le contenu attendu, dans ce cas comme il y a l'élément conditionnel "|", alors la regex cherchera un passage commençant par "`rel/release-`" ou "`release-`".

L'extraction des variables est représentée par les éléments "`(?P<v1>\d)+`", "`(?P<v2>\d)+`" ou "`(?P<v3>\d)+`". Le "`\d+`" indique que le groupe doit être un élément

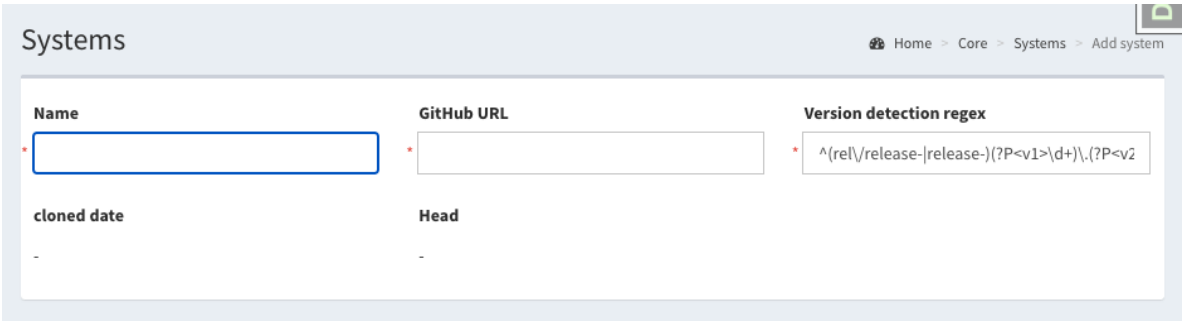

FIGURE 3.6 : Interface utilisateur - Écran de la liste des systèmes
©Marcelo Medeiros de Vasconcellos, 2023

numérique avec au moins un algorithme et l'élément "`?P<v1>`" indique que le contenu pris par ce groupe sera nommé "v1" dans ce cas. En langage python, lorsque la REGEX est extraite par la méthode `groupdict`, le résultat est un dictionnaire dont la clé est le nom du groupe. Dans notre système, si cette clé n'existe pas, la valeur "0" est attribuée au groupe.

Les éléments "`\.`" représentent que les groupes ci-dessus doivent être séparés par un ".".

Les données des LPS sont accessibles via la page « Log Statements », représentée par la figure 3.7. Il sera possible d'y rechercher les caractéristiques d'une instruction de journalisation, telles que la version à laquelle elle appartient, le niveau de sévérité, si elle se trouve dans un test, la taille du message sans les variables, le nombre de mots, le nombre de variables, si le message se trouve à l'intérieur d'une instruction conditionnelle, d'une boucle, d'un `break`, d'un `continue` ou d'un `try/catch`, et enfin si elle utilise la concaténation de chaînes avec les variables.

Si l'utilisateur est intéressé par la recherche des messages statiques, il peut accéder à la page « Messages », illustrée dans la figure 3.8. Dans cette page, il peut filtrer les messages par système, en fonction du nombre de fichiers où le message apparaît, de la longueur du message

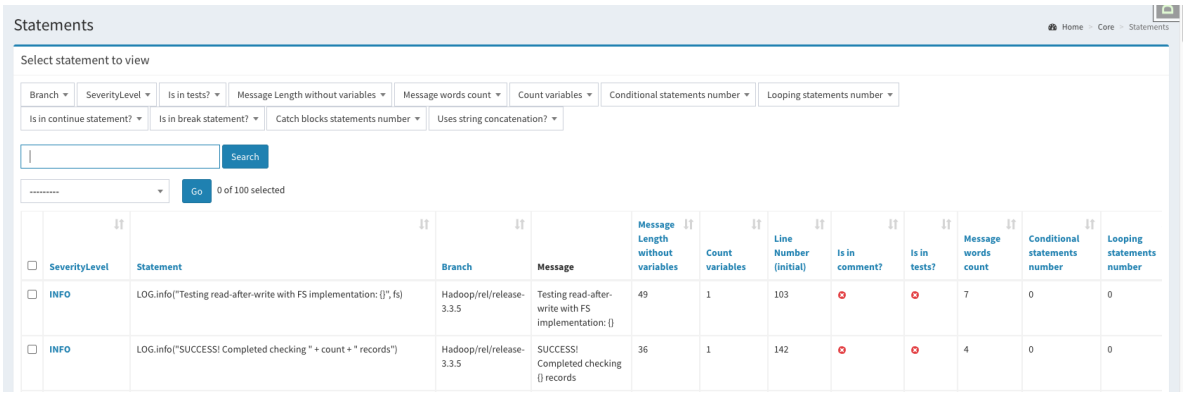

FIGURE 3.7 : Interface graphique - Écran Liste des LPS
©Marcelo Medeiros de Vasconcellos, 2023

(à l'exclusion de la taille des variables), du nombre de variables, du nombre de LPS qui sont liées à ce message, et finalement en fonction du nombre de niveaux de sévérité qui sont liés à ce message. Grâce à cette page, l'utilisateur peut par exemple visualiser les messages les plus répétés dans le système, ou encore identifier un message qui apparaît dans plus d'un niveau de sévérité parmi d'autres recherches qui peuvent le guider dans l'amélioration de ses LPS.

La page « Branch File Comparison » permet à l'utilisateur de rechercher les LPS qui ont subi des changements dans leurs niveaux de sévérité, leurs messages ou leurs variables, y compris le filtrage par rapport aux fichiers et aux versions.

3.5 EXTRACTION DES DONNÉES

Dans cette section, nous expliquons comment fonctionne l'extraction des données du code source des dépôts. La première étape du traitement sera le clonage des dépôts dans un dossier local ; ensuite, le pipeline effectuera l'extraction des LPS et des attributs des méthodes, où ils apparaissent. L'extraction des LPS est importante pour pouvoir les caractériser, et les attributs des méthodes sont importants pour pouvoir compter les SLOC et identifier

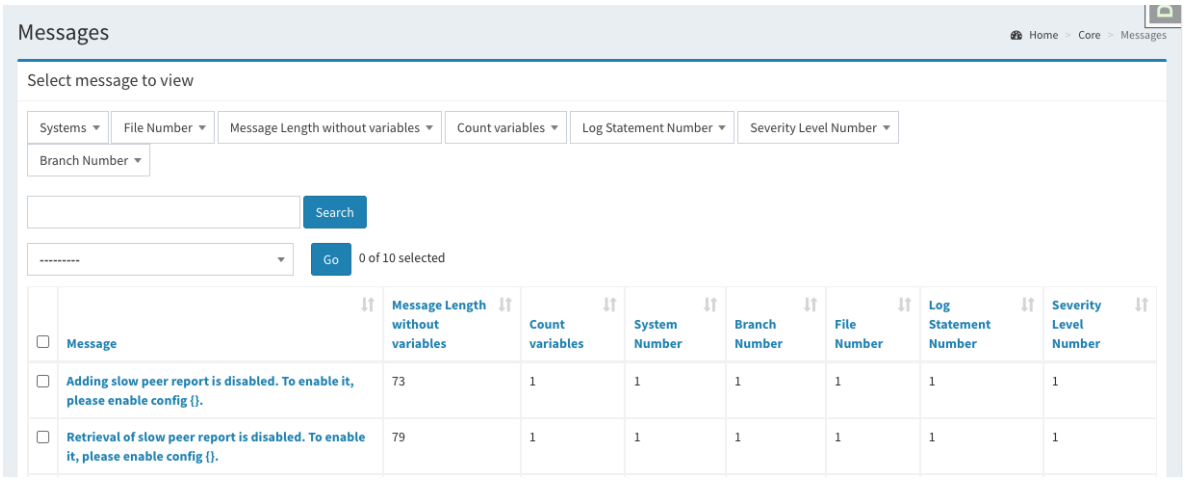

FIGURE 3.8 : Interface graphique - Écran Liste des messages
©Marcelo Medeiros de Vasconcellos, 2023

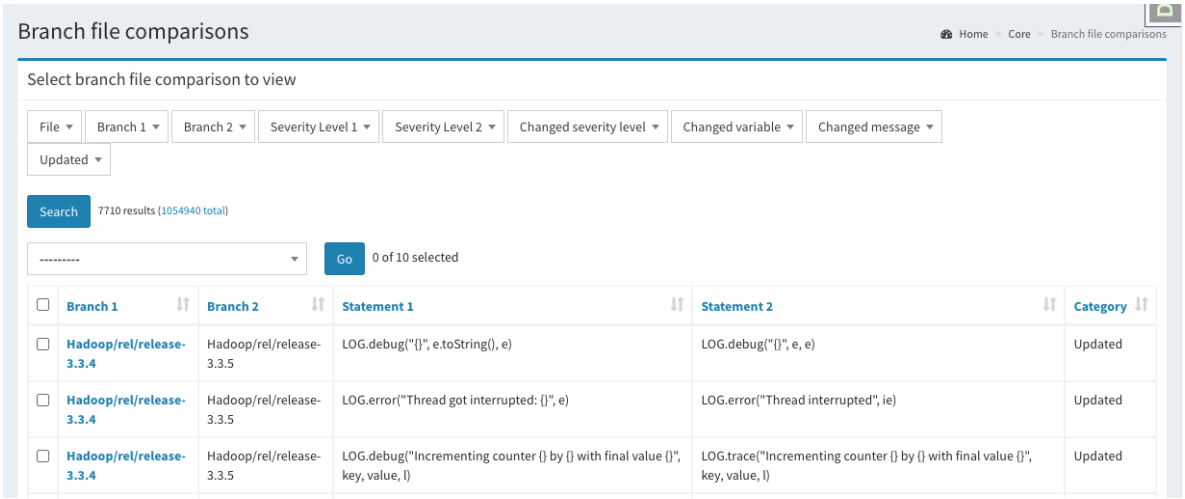

FIGURE 3.9 : Interface graphique - Écran Liste des LPS
©Marcelo Medeiros de Vasconcellos, 2023

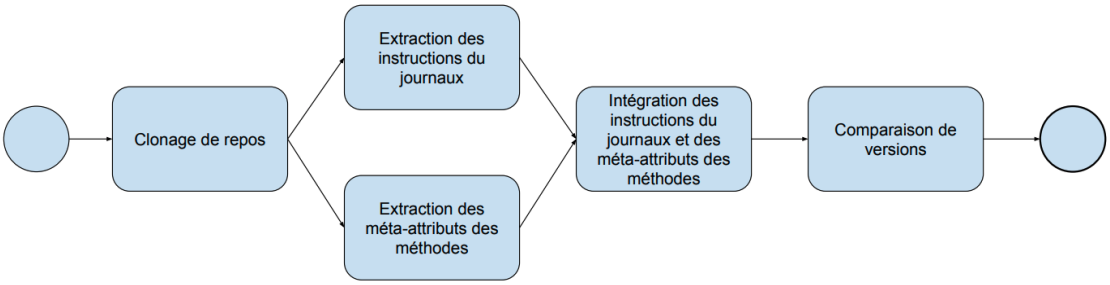

FIGURE 3.10 : Schéma simplifié du *pipeline* de traitement des projets
©Marcelo Medeiros de Vasconcellos, 2023

l'emplacement des LPS dans la méthode. La dernière étape du pipeline consistera à comparer les LPS des versions successives. C'est à ce moment que l'outil peut identifier les changements dans les LPS. Le processus complet est représenté dans la figure 3.10.

3.5.1 CLONAGE DE DÉPÔTS

Dans cette étape, le clonage de toutes les branches des projets que nous souhaitons analyser est effectué. Ce *pipeline* lit les méta-données contenues dans le dépôt et recherche la liste de toutes les branches enregistrées, après quoi il crée un clone de chaque branche et l'enregistre dans un répertoire local défini par une variable globale du système. Par la suite, l'outil enregistre dans la base de données des informations comme la date et d'heure à laquelle le dépôt a été cloné, ainsi que la référence à la tête (*head*) du dépôt.

3.5.2 EXTRACTION DES INSTRUCTIONS DE JOURNALISATION (LPS)

Cette étape est chargée d'extraire les données concernant les fichiers, les messages et les LPSs. La première étape est la lecture des fichiers, où le système passe par chaque *release* et lit les fichiers avec l'extension `.java` (puisque l'outil se concentre sur ce langage). Chaque

fichier est traité séparément ; le contenu du fichier source en cours d'examen est inséré dans une variable appelée `content`.

```
   Données : content, regex
 1 content ← content.removeStringsAndComments() ;
 2 match ← Regex.match(regex, content) ;
 3 listOfPositionPairs ← [] ;
 4 tant que match faire
 5 |   pour Every match faire
 6 |   |   initial, end ← match.span() ;
 7 |   |   listOfPositionPairs.append([initial, end]) ;
 8 |   |   lengthMatch ← end − initial ;
 9 |   |   newString ← "∗" ∗ lengthMatch ;
10 |   |   content ← content[: initial] + newString + content[end :] ;
11 |   fin
12 |   match ← regex.match(regex, content) ;
13 fin
   Résultat : listOfPositionPairs[]
```

Algorithme 3.1 : Capture des ouvertures et fermetures de méthodes

Certaines étapes de traitement sont appliquées au contenu de cette variable. Tout d'abord, nous remplaçons tout le contenu des variables de texte, des commentaires sur une seule ligne et des commentaires sur plusieurs lignes contenus dans cette variable par des astérisques. Ce processus est important car il supprimera les LPS qui sont commentées (c'est-à-dire, ne font pas partie de l'exécution du système). On évitera également les erreurs qui peuvent survenir lors de la phase d'extraction de la structure du fichier, car les parenthèses ou les crochets qui ne correspondent pas entraîneront une erreur d'analyse ; ce travail est effectué par la fonction `removeStringsAndComments`. Le remplacement des variables de texte, des commentaires sur une seule ligne et sur plusieurs lignes est effectué à l'aide des regex données dans les figures 3.11 à 3.13.

```
"[^"]*"
```

FIGURE 3.11 : Regex utilisée pour trouver les strings dans le contenu du fichier.

```
\/\/.*
```

FIGURE 3.12 : Regex utilisée pour trouver les commentaires d'une seule ligne dans le contenu du fichier.

Après avoir appliqué ces transformations, la phase d'extraction de la structure du fichier commence. Pour cela, le système lit toutes les clés (de la forme "{}", en utilisant la regex de la figure 3.14) et les parenthèses "()" (en utilisant la regex de la figure 3.15) contenues dans le code source et stocke dans une variable la liste des tuples contenant le point d'ouverture et de fermeture de la balise. Il effectue ce travail selon l'algorithme 3.1. Cet algorithme est exécuté une fois avec chacune des regex et son résultat est stocké en mémoire.

Pour illustrer, nous utiliserons les lignes 41 à 60 du fichier `ExecutorHelper.java`, présent dans la version 3.3.4 du code source de Apache Hadoop ; cet extrait est illustré dans la figure 3.16.

```
^[ \t]*\/\*[\S\s]*?\*\/[ \t]*$
```

FIGURE 3.13 : Regex utilisée pour trouver les commentaires sur plusieurs lignes dans le contenu du fichier.

```
\{[^{}]*\}
```

FIGURE 3.14 : Regex utilisée pour trouver les ouvertures et les fermetures des méthodes dans le contenu du fichier.

```
\([^()]*\)
```

FIGURE 3.15 : Regex utilisée pour trouver les ouvertures et les fermetures des méthodes dans le contenu du fichier.

```
41      //For additional information, see: https://docs.oracle
42      // .com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor
43      // .html#afterExecute(java.lang.Runnable,%20java.lang.Throwable)
44
45      // Handle JDK-8071638
46      if (t == null && r instanceof Future<?> && ((Future<?>) r).isDone()) {
47        try {
48          ((Future<?>) r).get();
49        } catch (ExecutionException ee) {
50          LOG.debug(
51              "Execution exception when running task in {}", Thread.currentThread()
52                  .getName());
53          t = ee.getCause();
54        } catch (InterruptedException ie) {
55          LOG.debug("Thread ( {} ) interrupted: ", Thread.currentThread(), ie);
56          Thread.currentThread().interrupt();
57        } catch (Throwable throwable) {
58          t = throwable;
59        }
60      }
```

FIGURE 3.16 : Illustration de l'état initial de la variable "content".
©Marcelo Medeiros de Vasconcellos, 2023

Au début, l'algorithme lit le fichier complet et le garde en mémoire. Ensuite, le système exécute la fonction `removeStringsAndComments` qui remplace toutes les variables de texte statique et les commentaires dans le code source par des séquences d'astérisques. Le résultat de la substitution est illustré dans la figure 3.17.

Ensuite, l'algorithme, à l'aide de la regex 3.14, recherche les méthodes d'ouverture et de fermeture ; à noter que la regex n'inclut que les parties qui ne contiennent pas de parenthèses à l'intérieur. Dans la figure 3.17, les rectangles représentent le résultat des recherches par regex.

FIGURE 3.17 : Illustration du contenu de la variable `content` après le remplacement des variables de texte statique et des commentaires par des astérisques
©Marcelo Medeiros de Vasconcellos, 2023

Le résultat des paires ordonnées est stocké en mémoire. A titre d'illustration, voici le résultat de la première exécution en boucle de l'algorithme :

```
[[1159, 1181], [1226, 1251], [1280, 1282], [1351, 1353],
[1372, 1374], [1409, 1411], [1419, 1421], [1704, 1715],
[1725, 1727], [1752, 1763], [1770, 1772], [1788, 1811],
[1912, 1914], [1939, 1941], [1967, 1969], [1985, 2010],
[2082, 2084], [2119, 2121], [2131, 2133], [2149, 2170],
[2218, 2229], [2321, 2323], [2331, 2333], [2374, 2376]]
```

Ensuite, tous les caractères sont remplacés par des listes d'astérisques ; le résultat de ce remplacement est illustré à la figure 3.18. L'algorithme effectue cette opération de manière répétée jusqu'à ce qu'il n'y ait plus de parenthèses dans le contenu de la variable. La figure 3.18 présente le deuxième tour de l'algorithme et la figure 3.19 présente le troisième et dernier tour de l'algorithme.

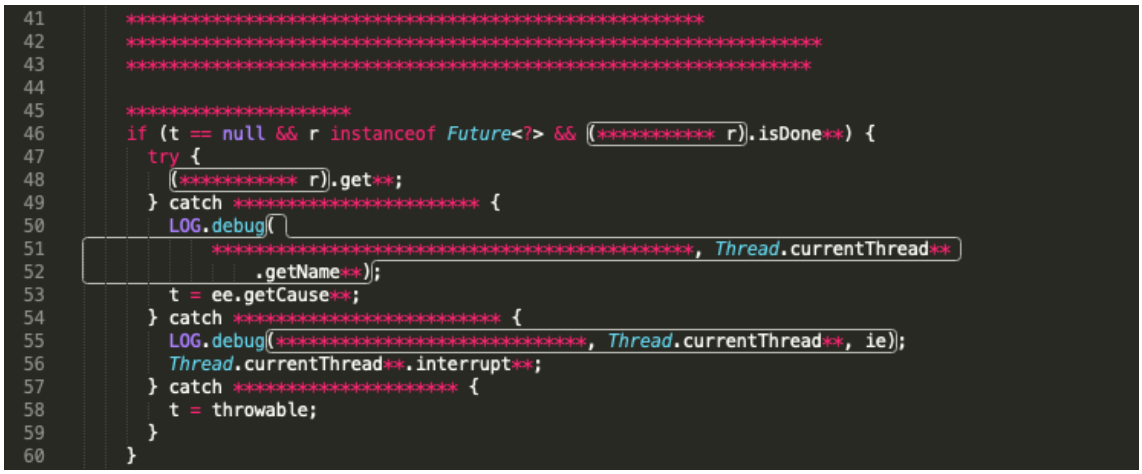

FIGURE 3.18 : Illustration du contenu de la variable `content` après le premier bouclage de l'algorithme
©Marcelo Medeiros de Vasconcellos, 2023

Le résultat des paires ordonnées dans la deuxième exécution de la regex est :

```
[[1261, 1283], [1301, 1422], [1703, 1718], [1751, 1766],
[1831, 1942], [2030, 2089], [2246, 2337]]
```

Le résultat des paires ordonnées dans la troisième exécution de la regex est :

```
[[1663, 1728]]
```

Le bouclage se termine lorsque la regex n'identifie plus de morceaux correspondants. L'état final de la variable est illustré dans la figure 3.20.

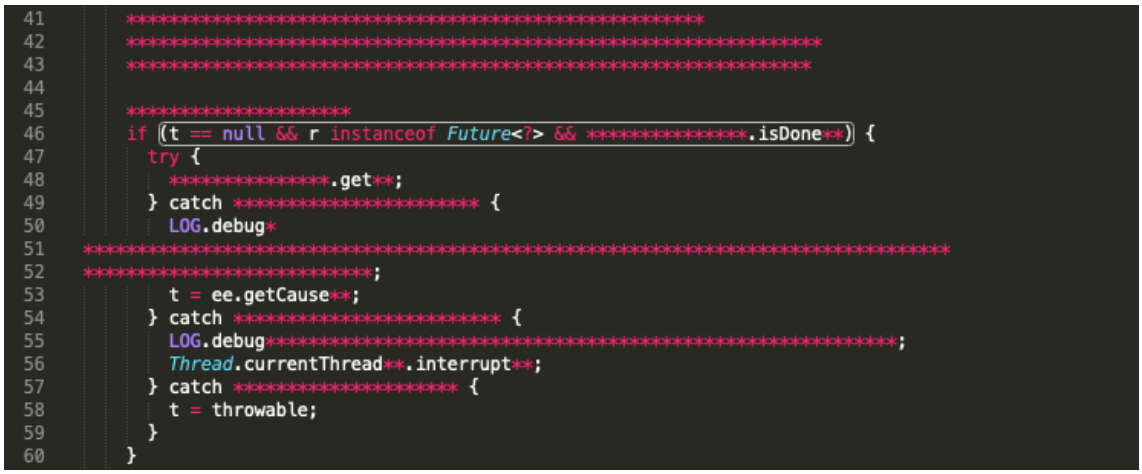

FIGURE 3.19 : Illustration du contenu de la variable `content` après le second bouclage de l'algorithme

©Marcelo Medeiros de Vasconcellos, 2023

FIGURE 3.20 : Illustration du contenu de la variable `content` à la fin de l'exécution de l'algorithme

©Marcelo Medeiros de Vasconcellos, 2023

À la fin de l'exécution de l'algorithme 3.1, le système stocke en mémoire la liste des paires de parenthèses représentant les points d'ouverture et de fermeture des méthodes. La sortie de l'algorithme est une variable comprenant toutes les paires, par exemple :

```
[[1159, 1181], [1226, 1251], [1280, 1282], [1351, 1353],
```

```
[1372, 1374], [1409, 1411], [1419, 1421], [1704, 1715],
[1725, 1727], [1752, 1763], [1770, 1772], [1788, 1811],
[1912, 1914], [1939, 1941], [1967, 1969], [1985, 2010],
[2082, 2084], [2119, 2121], [2131, 2133], [2149, 2170],
[2218, 2229], [2321, 2323], [2331, 2333], [2374, 2376],
[1261, 1283], [1301, 1422], [1703, 1718], [1751, 1766],
[1831, 1942], [2030, 2089], [2246, 2337], [1663, 1728]]
```

Une fois ce traitement effectué, on effectue une recherche de LPS en utilisant la regex ci-dessous, qui parcourt le fichier à la recherche de LPS d'une ou plusieurs lignes et renvoie la portion de chaîne dans laquelle le LPS est contenue dans le code source. Dans ce cas, la fonction regex utilise l'indicateur `DOTALL`, où le symbole « . » remplace n'importe quel caractère y compris les sauts de ligne.

```
(?i)[a-zA-Z0-9]*log[a-zA-Z0-9]*[\s]*\.[\s]*(|log\([a-zA-Z0-9\.]*)
(?P<severity_level>fatal|error|warn|info|debug|trace|severe|warn
|config|fine|finer|finest)[\s]*(\\(|\,)
```

FIGURE 3.21 : Regex utilisée pour capturer les LPS
©Marcelo Medeiros de Vasconcellos, 2023

Pour faciliter la compréhension de l'action de l'expression régulière, nous avons utilisé le site web Regex101 [55], qui permet de construire, tester et déboguer des regex en ligne d'une manière interactive et visuelle. Les exemples d'extraits de code identifiés par l'algorithme sont illustrés dans la figure 3.22, et les exemples qui ne sont pas identifiés par l'algorithme (donc les contre-exemples) sont illustrés dans la figure 3.23.

FIGURE 3.22 : Exemples d'extraits de code identifiés par la regex 3.21
©Marcelo Medeiros de Vasconcellos, 2023

FIGURE 3.23 : Exemples d'extraits de code qui ne sont pas identifiés par la regex 3.21
©Marcelo Medeiros de Vasconcellos, 2023

FIGURE 3.24 : Extraits de code identifiés par la regex 3.21
©Marcelo Medeiros de Vasconcellos, 2023

Pour identifier la fermeture de chaque LPS, il identifie le point de fermeture dont l'ouverture se trouve dans l'extrait de code identifié par la regex du LPS, en utilisant la sortie de l'algorithme 3.1.

Pour illustrer cette action, nous utiliserons le même extrait du fichier utilisé ci-dessus entre les lignes 41 et 60 du fichier "ExecutorHelper.java" après avoir supprimé les variables de texte statique et les commentaires, et nous appliquerons la regex 3.21.

La regex a identifié deux LPS dans l'extrait de fichier, l'une comprise dans l'intervalle [1822, 1832] et l'autre dans l'intervalle [2021, 2031]. Notre pipeline recherchera parmi l'ensemble des paires identifiées par l'algorithme 3.1 l'ensemble de paires qui commence entre la même étendue où la regex a identifié le LPS ; dans ce cas il identifiera les paires [1831, 1942], où le nombre 1831 est compris dans l'intervalle [1822, 1832], et [2030, 2089],

FIGURE 3.25 : Identification des LPS
©Marcelo Medeiros de Vasconcellos, 2023

où le nombre 2030 est compris dans l'intervalle [2021, 2031]. De cette manière, le système identifie les LPS comprises entre les intervalles [1822, 1942] et [2021, 2089], respectivement, en utilisant la ligne de départ du LPS et le numéro de ligne de clôture de la méthode. La figure 3.25 en illustre le fonctionnement, où les rectangles rouges représentent les sections liées aux LPS et les rectangles jaunes représentent les sections liées aux méthodes.

Dans notre étude, nous extrayons également les LPS qui sont produites au moyen d'instructions qui écrivent dans la console (`System.out.println` et `System.err.println`). Dans ce cas, nous considérons le titre des déclarations comme des niveaux de sévérité afin de pouvoir identifier les changements survenus entre ces déclarations et leur transformation en déclarations provenant des journaux. Les figures 3.27 et 3.28 illustrent des extraits de code identifiés et non identifiés par l'expression régulière 3.26, respectivement.

```
(?P<severity_level>System\.out\.println|System\.err\.println)\(
```

FIGURE 3.26 : Regex utilisée pour capturer les instructions d'impression
©Marcelo Medeiros de Vasconcellos, 2023

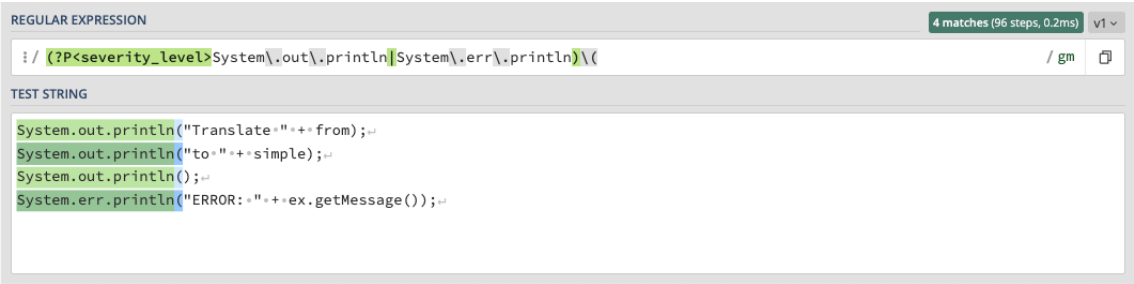

FIGURE 3.27 : Exemples d'extraits de code identifiés par l'regex 3.26
©Marcelo Medeiros de Vasconcellos, 2023

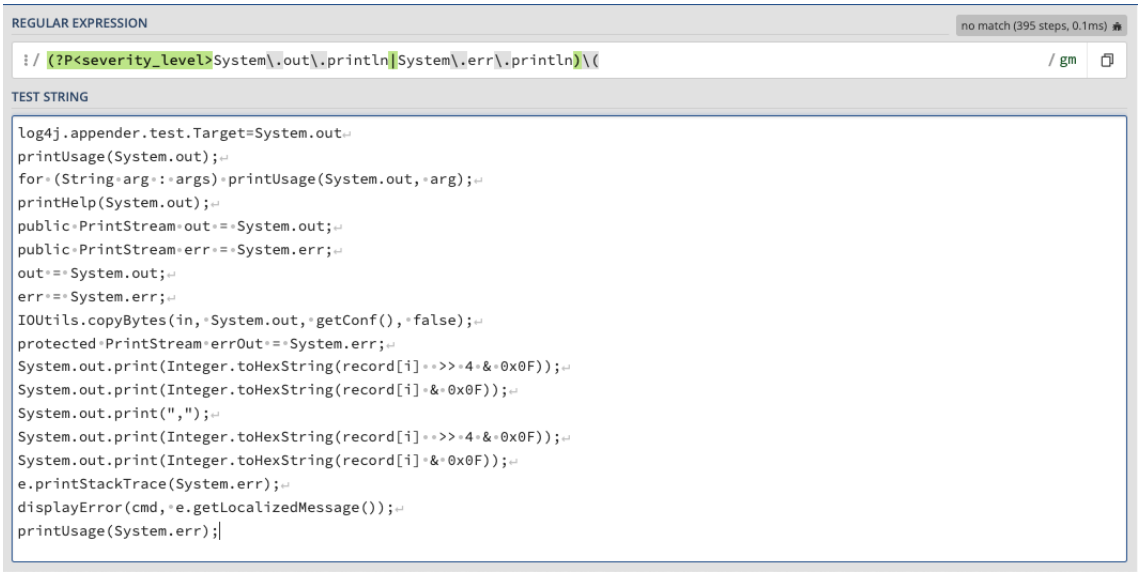

FIGURE 3.28 : Exemples d'extraits de code qui ne sont pas identifiés par l'regex 3.26
©Marcelo Medeiros de Vasconcellos, 2023

Le tableau 3.1 présente certaines des variables extraites du système pendant l'exécution de ce pipeline. Ces variables sont extraites pour aider à comprendre les défis suivants :

TABLEAU 3.1 : Caractéristiques extraites au cours de l'opération LPS
©Marcelo Medeiros de Vasconcellos, 2023

Variable	Description	Défi associé
line_number_initial	Numéro de ligne où la LPS commence	
line_number_final	Numéro de ligne où se termine la LPS	
span_initial	Numéro du caractère qui commence la LPS	
span_final	Numéro du caractère qui termine la LPS	
severity_level	Niveau de sévérité du journal	2.3.4
filename	Nom complet du fichier	2.3.2
original_statement	Contenu de la LPS originale	2.3.2
message	Message statique avec informations de localisation variables	2.3.2
message_without_variables	Message statique sans variables	2.3.2
message_with_variables	Message statique original	2.3.2
message_words_count	Nombre de mots du message statique	2.3.2
message_length	Nombre de caractères du message statique	2.3.2
variables_number	Nombre de variables du message statique	2.3.2
code_snippet	Extrait de code, comprenant 5 lignes avant et 5 lignes après	2.3.1
is_in_test	Indication de la présence dans les dossiers de test	2.3.1
uses_string_concatenation	Indique l'utilisation de la concaténation de chaînes de caractères.	2.3.1
conditional_statements_number	Nombre de déclarations conditionnelles qu'il est inséré	2.3.1
looping_statements_number	Nombre d'instructions de bouclage qu'il insère	2.3.1
catch_blocks_statements_number	Nombre d'instructions catch/block qu'il insère	2.3.1
is_with_continue_statement	Indicateur d'insertion dans une déclaration de continuation	2.3.1
is_with_break_statement	Indicateur d'insertion dans une déclaration d'arrêt	2.3.1

« Comment enregistrer ? », « Quand enregistrer ? » et « Où enregistrer ? ». Les données extraites dans ce *pipeline* sont stockées dans plusieurs tables, parmi lesquelles on peut citer : `core_file`, `core_message` et `core_statement`.

Les expressions régulières sont également utilisées à plusieurs fins dans ce *pipeline*, parmi lesquelles nous pouvons mentionner l'identification des commentaires avec plusieurs lignes et le comptage des variables.

3.5.3 EXTRACTION DES MÉTA-ATTRIBUTS DES MÉTHODES

Ce *pipeline* exploite le code source pour extraire des informations concernant les méthodes du logiciel analysé. Il utilise la bibliothèque *Lizard* [21], un analyseur de complexité cyclomatique extensible pour de nombreux langages de programmation. Il est capable d'extraire des informations relatives à la complexité cyclomatique, au nombre de lignes de code et à d'autres variables de chaque fichier ou méthode. Au moyen de cet outil, il a été possible

d'extraire des informations sur les méthodes du système analysé, en capturant les métriques suivantes : complexité cyclomatique, SLOC, nombre de *tokens*, nom complet de la méthode (y compris ses arguments), numéro de ligne de la méthode de départ et numéro de ligne de la méthode de fin. Le numéro de ligne de la méthode de début et le numéro de ligne de la méthode de fin seront utilisés par le pipeline d'intégration pour identifier dans quelle(s) méthode(s) se trouve un LPS. En outre, nous utilisons la regex de la figure 3.29 pour extraire le nombre total de lignes représentant des instructions de journalisation de journaux (excluant les importations et l'initialisation des méthodes).

```
(.*LOG.is.*|.*getLogger.*|.*LoggerFactory\.*|import.*\.(logging|slf4j)\..*;)
```

FIGURE 3.29 : regex utilisée pour identifier les importations et l'initialisation des méthodes des journaux.

La figure 3.30 illustre des exemples de lignes identifiées à l'aide de la regex 3.29. Ces informations sont stockées dans la base de données et seront utilisées plus tard par le pipeline d'intégration pour compter le nombre de lignes du journal. À noter qu'à ce stade, la déclaration `LOG.debug(` n'est pas identifiée par cette regex ; le nombre de lignes se référant aux LPS sera fourni par le pipeline d'extraction des LPS, car de cette manière il sera possible d'effectuer le décompte précis des LPS multi-lignes.

Grâce à ce *pipeline*, nous extrayons des informations liées aux méthodes du système. Avec cela il est possible d'extraire des informations pour une meilleure compréhension du défi « Où enregistrer ? ». Ce *pipeline* extrait des informations telles que : les méthodes, la localisation, la complexité cyclomatique, le nombre de lignes de journalisation (excluant les importations et l'initialisation des méthodes) et les SLOC, entre autres.

```
21  package org.apache.hadoop.util.concurrent;
22
23  import org.slf4j.Logger;
24  import org.slf4j.LoggerFactory;
25
26  import java.util.concurrent.ExecutionException;
27  import java.util.concurrent.Future;
28
29  /** Helper functions for Executors. */
30  public final class ExecutorHelper {
31
32    private static final Logger LOG = LoggerFactory
33        .getLogger(ExecutorHelper.class);
34
35    static void logThrowableFromAfterExecute(Runnable r, Throwable t) {
36      if (LOG.isDebugEnabled()) {
37        LOG.debug("afterExecute in thread: " + Thread.currentThread()
38            .getName() + ", runnable type: " + r.getClass().getName());
39      }
40
```

FIGURE 3.30 : Identification de l'importation et initialisation des méthodes de journalisation ©Marcelo Medeiros de Vasconcellos, 2023

3.5.4 INTÉGRATION DES LPS ET DES MÉTA-ATTRIBUTS DES MÉTHODES

Ce pipeline est chargé d'établir une connexion entre les données extraites des pipelines précédents. Il établit la relation entre les données des méthodes et les LPS et effectue également certains calculs pour faciliter la découverte des connaissances à l'aide de l'interface du système. Il exécute ce pipeline par le biais d'une séquence de requêtes Structured Query Language (SQL).

Le pipeline effectue cette tâche en recherchant l'identification de la ou des méthodes dans lesquelles le LPS est insérée —c'est-à-dire que le numéro de ligne de départ du LPS doit être inférieur au numéro de ligne initial de la méthode et que le numéro de ligne final du LPS doit être supérieur au numéro de ligne final de la méthode. Cette opération est effectuée directement dans la base de données au moyen de la requête SQL illustrée à la figure 3.31. Cette liaison est stockée dans la table `public.core_methodstatement`.

```
INSERT INTO public.core_methodstatement (statement_id, method_id)
SELECT DISTINCT s.id AS statement_id, m.id AS method_id
  FROM public.core_statement s
 INNER JOIN public.core_method m
       ON m.branch_id=s.branch_id AND m.file_id=s.file_id
          AND m.start_line <= s.line_number
          AND m.end_line >= s.line_number_final
          AND m.id IS NOT NULL AND s.id IS NOT NULL;
```

FIGURE 3.31 : Instruction SQL utilisée pour créer, identifier et stocker la relation entre les méthodes et les relevés dans la base de données.

3.5.5 COMPARAISON DE VERSIONS

Ce *pipeline* est chargé de faire la comparaison entre les fichiers des différentes versions (*releases*) d'un même projet. Sa première fonction est d'ordonner les versions de manière chronologique en fonction du numéro. Chaque *release* ne peut avoir qu'au plus un prédécesseur et un successeur.

Après avoir identifié chaque paire séquentielle de *releases*, notre *pipeline* identifie chaque fichier présent dans l'une ou l'autre, et applique un traitement différent selon sa présence :

1. **Fichiers existant uniquement dans la *release* précédente :** Toutes les LPS existant uniquement dans les fichiers présents dans le *release* précédente sont enregistrées dans notre base de données en tant que déclarations supprimées.
2. **Fichier existant uniquement dans la *release* ultérieure :** Toutes les LPS n'existant que dans le *release* ultérieure sont enregistrées dans notre base de données en tant que déclarations ajoutées.

3. **Existant à la fois dans précédente et ultérieure :** La différence entre les fichiers est calculée, les extraits qui constituent la différence sont identifiés et chaque LPS qui est présent dans ces extraits est comparé un par un. La *similarité cosinus* entre chaque paire est calculé ; si cette valeur est supérieure à 50%, la paire est sauvegardée dans une liste de candidats potentiels. On trie ensuite ces paires en fonction de leur cosinus de similarité, du plus élevé au plus bas. L'algorithme tente ensuite d'associer les LPS du premier fichier à celles du second au moyen d'un algorithme glouton détaillé dans l'algorithme 3.2.

Nous allons expliquer cette dernière étape en l'illustrant à l'aide d'extraits des versions du fichier `Journal.java` présent dans les versions 3.0.2 et 3.0.3 de Hadoop, et tenter d'apparier les LPS de ces deux fichiers. Les différences entre les fichiers sont illustrées dans la figure 3.32. Sur le côté gauche, les suppressions associées à la version 3.0.2 sont surlignées en orange et sur le côté droit, les ajouts associés à la version 3.0.3 sont surlignés en vert.

```
205     LOG.info("Scanning storage " + fjm);
206     List<EditLogFile> files = fjm.getLogFiles(0);
207
208     while (!files.isEmpty()) {
209       EditLogFile latestLog = files.remove(files.size() - 1);
210       latestLog.scanLog(Long.MAX_VALUE, false);
211       LOG.info("Latest log is " + latestLog);
212       if (latestLog.getLastTxId() == HdfsServerConstants.INVALID_TXID) {
213         // the log contains no transactions
214         LOG.warn("Latest log " + latestLog + " has no transactions. " +
215             "moving it aside and looking for previous log");
216         latestLog.moveAsideEmptyFile();
217       } else {
218         return latestLog;
219       }
220     }
221
222     LOG.info("No files in " + fjm);
223     return null;
224   }
225
226   /**
227    * Format the local storage with the given namespace.
228    */
229   void format(NamespaceInfo nsInfo) throws IOException {
230     Preconditions.checkState(nsInfo.getNamespaceID() != 0,
231         "can't format with uninitialized namespace info: %s",
232         nsInfo);
233     LOG.info("Formatting " + this + " with namespace info: " +
234         nsInfo);
235     storage.format(nsInfo);
236     refreshCachedData();
237   }
```

```
205     LOG.info("Scanning storage " + fjm);
206     List<EditLogFile> files = fjm.getLogFiles(0);
207
208     while (!files.isEmpty()) {
209       EditLogFile latestLog = files.remove(files.size() - 1);
210       latestLog.scanLog(Long.MAX_VALUE, false);
211       LOG.info("Latest log is " + latestLog + " ; journal id: " + journalId);
212       if (latestLog.getLastTxId() == HdfsServerConstants.INVALID_TXID) {
213         // the log contains no transactions
214         LOG.warn("Latest log " + latestLog + " has no transactions. " +
215             "moving it aside and looking for previous log"
216             + " ; journal id: " + journalId);
217         latestLog.moveAsideEmptyFile();
218       } else {
219         return latestLog;
220       }
221     }
222
223     LOG.info("No files in " + fjm);
224     return null;
225   }
226
227   /**
228    * Format the local storage with the given namespace.
229    */
230   void format(NamespaceInfo nsInfo) throws IOException {
231     Preconditions.checkState(nsInfo.getNamespaceID() != 0,
232         "can't format with uninitialized namespace info: %s",
233         nsInfo);
234     LOG.info("Formatting journal id : " + journalId + " with namespace info: "
+
235         nsInfo);
236     storage.format(nsInfo);
237     refreshCachedData();
238   }
```

FIGURE 3.32 : Extrait du fichier `Journal.java` du projet Hadoop
©Marcelo Medeiros de Vasconcellos, 2023

```
--- Hadoop/rel/release-3.0.2/.../Journal.java
+++ Hadoop/rel/release-3.0.3/.../Journal.java
@@ -211 +211 @@
-LOG.info("Latest log is " + latestLog);
+LOG.info("Latest log is " + latestLog + " ; journal id: " + journalId);
@@ -215 +215,2 @@
-"moving it aside and looking for previous log");
+"moving it aside and looking for previous log"
++ " ; journal id: " + journalId);
@@ -233 +234 @@
-LOG.info("Formatting " + this + " with namespace info: " +
+LOG.info("Formatting journal id : " + journalId + " with namespace info: " +
(...)
```

FIGURE 3.33 : Résultat de l'exécution de la bibliothèque DiffLib dans l'extrait de code du fichier `Journal.java`

Pour chaque fichier existant à la fois dans la source et la destination, nous effectuons la comparaison entre leurs contenus, en utilisant la bibliothèque *Difflib* [22]. Pour chaque extrait de changement identifié est capturé la ligne de début et la ligne de fin des changements, comme le montre la figure 3.33. À remarquer que si nous appliquions directement notre regex 3.21 pour identifier les LPS à cette sortie, nous ne pourrions pas identifier que le changement de la ligne 215 est associé à un LPS, puisque le changement ne s'est pas produit sur la même ligne que l'appel à la méthode de journal (`LOG.warn`). En conséquence, la regex ne pourrait pas identifier ce changement comme se produisant dans un LPS. C'est ce qui justifie une approche différente lors de l'identification des LPS lorsque l'on compare deux versions d'un même fichier.

Le résultat de la figure 3.33 montre que les lignes `+211`, `+215` et `+233`, représentées par les descriptions `-211`, `-215`, `-233`, respectivement, de la version 3.0.2 ont été supprimées et que les lignes `+211`, `+215`, `+216` et `+234`, représentées par les désignations `+211`, `+215.2` et `+234`, respectivement, de la version 3.0.3 ont été ajoutées.

Le système interroge ensuite la base de données pour trouver les LPSs qui commencent ou se terminent sur ces lignes de code dans les versions respectives. Les LPS sont stockées dans deux listes distinctes et sont utilisées comme entrées de l'algorithme 3.2 ; cet algorithme nous renvoie une liste de paires (*pairsCosSim*) ordonnées de manière décroissante en fonction de leur similarité cosinus. La valeur de coupure que nous définissons est arbitrairement définie à 0.5, de cette façon pour les changements effectués dans les déclarations qui dépassent 50 % de son contenu ne sont pas considérés comme similaires.

```
   Données : lps_1d[], lps_2d[]
 1 var pairsCosSim ← [] ;
 2 var matched_1 ← [] ;
 3 var matched_2 ← [] ;
 4 pour lps1 dans lps_1d faire
 5 |   pour lps2 dans lps_2d faire
 6 |   |   cossim ← CosSim(lps1, lps2);
 7 |   |   si cossim > 0,5 alors
 8 |   |   |   pairsCosSim.append([cossim, lps1, lps2]);
 9 |   |   fin
10 |   fin
11 fin
12 Trier les pairsCosSim par ordre décroissant de cossim ;
   Résultat : pairsCosSim[]
```

Algorithme 3.2 : Algorithme de formation de paires

Le cosinus de similarité est une mesure de la similarité entre deux vecteurs dans un espace vectoriel qui évalue la valeur du cosinus de l'angle entre eux. Cette fonction trigonométrique donne une valeur de 1 si l'angle entre eux est nul, c'est-à-dire si les deux vecteurs pointent au même endroit. Si les vecteurs étaient orthogonaux, le cosinus s'annulerait, et s'ils étaient orientés dans la direction opposée, sa valeur serait de -1. Ainsi, la valeur de cette métrique se situe entre -1 et 1, c'est-à-dire dans l'intervalle fermé [-1, 1].

$$\cos(\theta) = \frac{A \cdot B}{\|A\| \, \|B\|} = \frac{\sum_{i=1}^{n} \mathbf{A}_i \mathbf{B}_i}{\sqrt{\sum_{i=1}^{n} (\mathbf{A}_i)^2} \sqrt{\sum_{i=1}^{n} (\mathbf{B}_i)^2}} \tag{3.1}$$

Cette distance est souvent utilisée dans la recherche et la récupération d'informations représentant des mots (ou des documents) dans un espace vectoriel [56]. Dans l'exploration de textes, la similarité en cosinus est appliquée pour établir une métrique de similarité entre les textes [57]. Dans l'exploration de données, elle est généralement utilisée comme indicateur de la cohésion des groupes de textes. La similarité par cosinus ne doit pas être considérée comme une métrique de la distance car elle ne satisfait pas à l'inégalité triangulaire.

Pour illustrer notre propos, nous allons calculer le cosinus de la similarité entre les deux LPS suivants :

```
LOG.warn("Exception closing reader", e)
LOG.warn("Exception closing reader for " + this.storeName, e)
```

La première étape du calcul consiste à transformer le texte en un dictionnaire de mots, où la clé est le mot lui-même et la valeur le nombre de fois qu'il apparaît dans le LPS. Le résultat des deux LPS sera :

```
vector1 = {
  "LOG": 1, "warn": 1, "Exception": 1,
  "closing": 1, "reader": 1, "e": 1
}
vector2 = {
  "LOG": 1, "warn": 1, "Exception": 1, "closing": 1,
  "reader": 1, "for": 1, "this": 1, "storeName": 1, "e": 1
```

```
}
```

Après cette étape, les mots qui sont à l'intersection des deux dictionnaires sont identifiés :

```
{"Exception", "LOG", "closing", "e", "reader", "warn"}
```

Le numérateur de la fonction est calculé en multipliant la valeur correspondant à chaque mot dans cette intersection ; ici, comme nous avons un vecteur de 6 éléments et que la valeur de chaque élément est un, le numérateur sera donc 6. Ensuite, on calcule la somme du carré des valeurs de chaque dictionnaire ; pour le premier dictionnaire, qui a 6 éléments et dont chacun a la valeur 1, le résultat de la somme des carrés sera 6. Pour le second vecteur qui a 6 éléments et dont chacun a la valeur 1, le résultat de la somme des carrés sera 9. Le dénominateur est alors calculé en prenant la racine carrée de 6 multipliée par la racine carrée de 9, soit $7,3485$. Le résultat de la similarité cosinus de la comparaison entre ces deux LPS sera égal à 6 divisé par 7.3485, soit 0.9129.

Comme on peut le voir, ce calcul prend en compte l'ensemble du contenu du LPS, y compris la classe, la méthode, le niveau de sévérité, le message statique et les variables existantes. L'utilisation de la similarité cosinus est importante pour identifier la similarité entre deux LPS dans les cas suivants :

- identifier les LPS qui ont été déplacées vers d'autres positions dans le fichier ;
- si plusieurs instructions ont été incluses, modifiées ou supprimées dans le même bloc de contenu modifié, le cosinus de similarité permet d'identifier celles dont le contenu est le plus proche ;
- réduire les faux positifs dans le cas où les LPS ont été complètement modifiées.

Données : $files$ (Liste des fichiers existants dans *branch*1 et *branch*2)

pour $file$ *dans* $files$ **faire**
    $matched\_1, matched\_2 \leftarrow [], []$;
    $diff_1, diff_2 \leftarrow getDifference(contentFile1, contentFile2)$;
    pour $diff_1$, $diff_2$ *dans* $zip(diff_1, diff_2)$ **faire**
        $lps\_1d \leftarrow lps\_1.filter(diff_1)$;
        $lps\_2d \leftarrow lps\_2.filter(diff_2)$;
        $pairsCosSim \leftarrow makeLPSPairs(lps\_1d[], lps\_2d[])$;
        pour *cossim,* lps_1, lps_2 *dans* $pairsCosSim$ **faire**
            si $lps\_1 \notin matched\_1$ *et* $lps\_2 \notin matched\_2$ **alors**
                $pairs.append([lps\_1, lps\_2, \text{“Updated”}])$;
                $matched\_1.append(lps\_1)$ et $matched\_2.append(lps\_2)$;
            fin
        fin
        pour $lps\_1d \notin matched\_1$ **faire**
            $pairs.append([lps\_1d, None, \text{“Removed”}])$
        fin
        pour $lps\_2d \notin matched\_2$ **faire**
            $pairs.append([None, lps\_2d, \text{“Added”}])$
        fin
    fin
fin

Résultat : *pairs* (Liste des paires de LPS)

Algorithme 3.3 : Algorithme de comparaison de fichiers

L'algorithme 3.3 est chargé d'effectuer ce tri pour nous, en considérant en priorité les paires dont les similarités cosinus sont les plus élevées. L'algorithme détecte les changements existants dans les fichiers ; si le cosinus de similarité est supérieur à 0,5, l'énoncé est alors étiqueté comme *Updated*. Pour les autres messages présents dans la différence de fichier, s'ils appartiennent au *release* source, ils sont étiquetés comme *Removed* et, s'ils appartiennent au *release* destination, ils sont étiquetés comme *Added*.

Après l'exécution de cet algorithme, l'outil recherche les LPS qui sont externes à la différence du contenu des fichiers détectés par l'algorithme de la bibliothèque *Difflib* [22]. Pour toutes ces LPS, l'outil donne l'étiquette comme *Unchanged*.

L'objectif de ce *pipeline* est de comparer les données extraites des lignes de code entre des versions consécutives d'un projet. Pour les fichiers présents dans les deux versions consécutives, le système effectue une comparaison entre modification, suppression ou inclusion, pour les fichiers présents uniquement dans la version précédente, il enregistre les LPS comme suppression et pour les fichiers présents uniquement dans la version suivante, il enregistre les LPS trouvées comme une inclusion.

A la fin de ce *pipeline*, toutes les données nécessaires à l'analyse sont stockées dans la base de données.

CHAPITRE IV
VALIDATION

Dans ce chapitre, nous présentons la validation par une étude de cas de l'approche proposée dans le chapitre 3, laquelle consistait à développer et à mettre en œuvre un outil capable d'extraire des informations du code source de dépôts Java open source.

Pour ce faire, nous analysons l'efficacité des fonctionnalités des trois modules proposés par l'approche : le module d'extraction des données, le module d'entraînement du modèle et le module d'inférence. Le projet sélectionné pour cette étude est le système Hadoop de l'Apache Software Foundation (Apache Software Foundation), un projet à code source ouvert qui compte actuellement plus de 490 contributeurs. Nous avons choisi ce projet car de nombreux chercheurs effectuent des travaux académiques en l'utilisant comme base, et nous pensons qu'il s'agit d'un projet mature et optimisé, notamment en ce qui concerne les pratiques de journalisation.

Dans la section 4.1 (Étude de cas), nous décrivons l'étude de cas à travers les étapes de la conception, des résultats et de l'analyse. Dans la section 4.2 (Discussion), nous analysons les données et discutons des résultats par rapport aux hypothèses du problème et aux questions de recherche.

4.1 ÉTUDE DE CAS

Dans la sous-section 4.1.1 (Conception), nous décrivons les conditions et les étapes de notre approche. Dans la sous-section 4.1.2 (Préparation et collecte), nous décrivons comment les critères de sélection des *releases* du projet. Enfin, dans la sous-section 4.1.3 (Résultats), nous décrivons les résultats de notre recherche.

4.1.1 CONCEPTION

Notre outil consiste en une solution logicielle qui vise à fournir des informations sur les LP. L'objectif de notre solution est d'extraire le LPS du code source, d'effectuer des comparaisons entre les versions stables du projet. Dans cette évaluation, notre solution permettra d'examiner les changements apportés au cours du développement du système. Nous avons commencé par définir le *pipeline* d'extraction, comme décrit dans la sous-section 4.1.2. Une fois les conditions initiales établies, l'outil a été déployé sur un serveur local. Enfin, les opérations de collecte et d'analyse des données ont été lancées.

Les données ont ensuite été évaluées et des séances de travail ont été organisées avec les données de production. Dans la sous-section 4.1.3, nous présenterons les résultats de certaines données extraites du système.

4.1.2 PRÉPARATION ET COLLECTE

Dans cette seconde étape, nous avons fait la sélection du système à analyser. Tel que mentionné en début de chapitre, nous avons choisi Hadoop, comme proposé dans le directive de Runeson *et al.* [58] pour la sélection des cas. Selon lui, le cas doit être sélectionné intentionnellement, le but de cette sélection étant d'étudier un cas qui devrait être "typique", "critique", "révélateur" ou "unique" à certains égards. Parmi les arguments justifiant ce choix, on note tout d'abord que Hadoop est un système qui a déjà été étudié par de nombreuses recherches, y compris des recherches liées aux journaux, et qu'il est donc considéré comme un candidat très approprié. De plus, le projet Hadoop compte actuellement 326 branches, parmi lesquelles nous pouvons mentionner les *releases* candidates, ainsi les *releases* et les branches secondaire utilisées dans le processus de développement. Comme critères d'exclusion, nous avons retiré de notre liste toutes les branches dont les noms n'obéissaient pas aux patrons suivants : "release-X.X.X" et "rel/release-X.X.X". De cette façon, 131 branches des versions

TABLEAU 4.1 : Vue d'ensemble de la quantité de *releases* analysée par version
©Marcelo Medeiros de Vasconcellos, 2023

Project	Version	Quantité
Hadoop	0	68
Hadoop	1	10
Hadoop	2	33
Hadoop	3	20

TABLEAU 4.2 : Aperçu du contenu extrait
©Marcelo Medeiros de Vasconcellos, 2023

Description	Quantité
Nombre des Releases	131
Nombre de fichiers (.java)	524 460
Nombre des LPS	1 015 836
Nombre des relations entre les releases	1 054 014

0, 1, 2 et 3 du projet ont été analysées, comme le montre le tableau 4.1. Au total, ces versions comptent 524 460 fichiers Java distincts.

4.1.3 RÉSULTATS

Après avoir exécuté notre *pipeline*, l'outil a enregistré les données extraites dans le système correspondant à chaque LPS, produisant 1 015 836 enregistrements ; il a également consigné chaque changement détecté entre des versions consécutives, produisant 1 054 014 enregistrements. Ces tables ont ensuite été analysées et certaines validations ont été effectuées sur l'approche proposée.

Le temps d'exécution par pipeline a été mesuré, les résultats sont présentés dans le tableau 4.3, et le temps d'exécution total pour les versions sélectionnées de Hadoop était de 7 heures 40 minutes 36 secondes (27636.3201 secondes).

TABLEAU 4.3 : Temps d'exécution du pipeline
©Marcelo Medeiros de Vasconcellos, 2023

Pipeline	Temps d'exécution
Clonage de repos	14min 16sec (856.7184 sec)
Extractions des instructions du journaux	3h 13min 48sec (11628.3890 sec)
Extractions des méta-attributs des méthodes	3h 00min 49sec (10849.7378 sec)
Intégrations des instructions du journaux et des méta-attributs des méthodes	5min 24sec (324.4656 sec)
Comparaison de versions	1h 06min 17sec (3977.0093 sec)

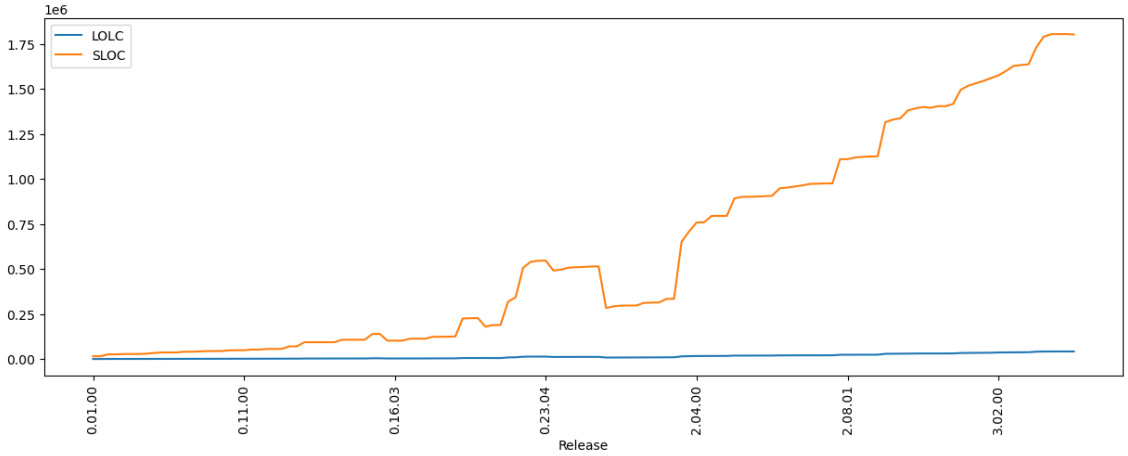

FIGURE 4.1 : Évolution de la LOLC et de la SLOC par version.
©Marcelo Medeiros de Vasconcellos, 2023

Une fois ces données colligées et traitées, il est maintenant possible de tenter de répondre aux questions de recherche mentionnées en début de mémoire.

QR1 : QUELLE EST LA TENDANCE SUIVIE PAR LA DENSITÉ DES LOGS À MESURE QUE LE SYSTÈME ÉVOLUE?

Pour répondre à cette question, nous utilisons les mesures de SLOC et de LOLC. D'après la figure 4.1, nous pouvons observer que numériquement, le code de fonctionnalité évolue à une vitesse plus élevée que le code de journalisation.

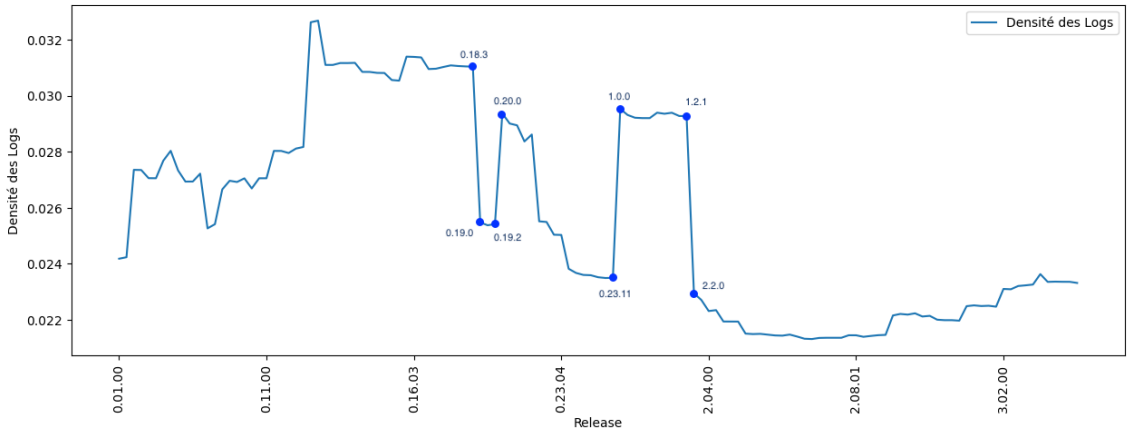

FIGURE 4.2 : Évolution de la LOGD par release.
©Marcelo Medeiros de Vasconcellos, 2023

Pour mieux comprendre la relation entre ces deux mesures, nous avons calculé la LOGD pour chaque version du système. Nous avons observé que la version 0.13.1 avait la LOGD la plus élevée, soit 0,0327, et que la version 2.7.3 avait la LOGD la plus faible, soit 0,0213. Nous notons également que, pour le cas de Hadoop, entre la version 2.2.0 et 3.3.5, la LOGD s'est stabilisée avec une valeur moyenne de 0.0221 et un faible écart-type de 0.0007. Dans cet intervalle de version la valeur minimale de LOGD de 0.0213 (dans la version 2.7.3) et la valeur maximale est de 0.0236 (dans la version 3.3.0). Cela signifie que, en moyenne, on retrouve une ligne de code de journalisation à toutes les 45.249 lignes de code de fonctionnalité.

Nous pouvons commenter la diminution présentée entre les versions 0.19.0 et 0.19.2 et l'augmentation présentée entre les versions 1.0.0 et 1.2.1. La diminution représentée sur la figure 4.2 entre les versions 0.19.0 et 0.19.2 se réfère à l'augmentation du code de fonctionnalité sans l'inclusion dans la même proportion du code de journal, ceci peut également être observé par la quantité de fichiers inclus, dans la version 0.18.3, le système avait 927 fichiers .java et dans la version 0.19.0, il en avait 1 707, soit une augmentation de 780 fichiers, puisque dans la version 0.19.2, le système avait 1728 fichiers, tandis que dans la version 0.20.0, il n'avait que 1 267 fichiers .java, soit une réduction de 461 fichiers.

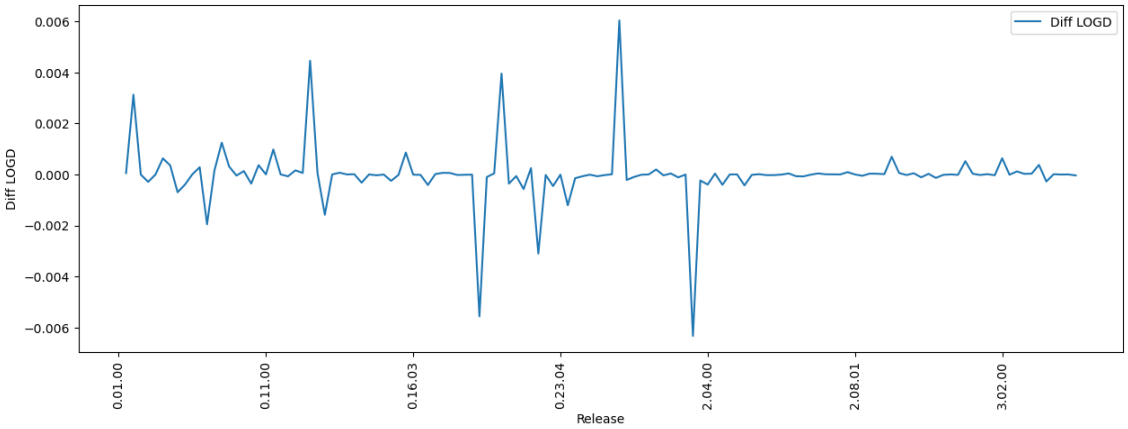

FIGURE 4.3 : Valeur absolue de la différence de LOGD par version.
©Marcelo Medeiros de Vasconcellos, 2023

La forte densité du journal observée entre les versions 1.0.0 et 1.2.1 est liée aux changements structurels du système qui se reflètent également dans la quantité de lignes de code de fonctionnalité. Jusqu'à la version 0.22.0, le système présentait une structure à projet unique. Entre les versions 0.23.0 et 0.23.11, le système a été divisé en plusieurs projets, ce qui a augmenté le nombre de lignes de code de fonctionnalité. La version structurée en un seul projet s'est poursuivie entre les versions 1.0.0 et 1.2.1, et ce n'est que dans la version 2.2.0 que le système a reçu définitivement la version dans laquelle le système est structuré en plusieurs projets.

Les versions comprises entre 2.0.0 et 2.1.1 n'ont pas été évaluées car elles n'ont pas été présentées comme des versions officielles, n'ayant été publiées qu'en tant que versions alpha et beta.

Nous pouvons observer globalement qu'à mesure que le code évolue, la différence entre la LOGD de la version actuelle et de sa version précédente tend à diminuer, montrant une fois de plus la tendance à la stabilisation de la métrique.

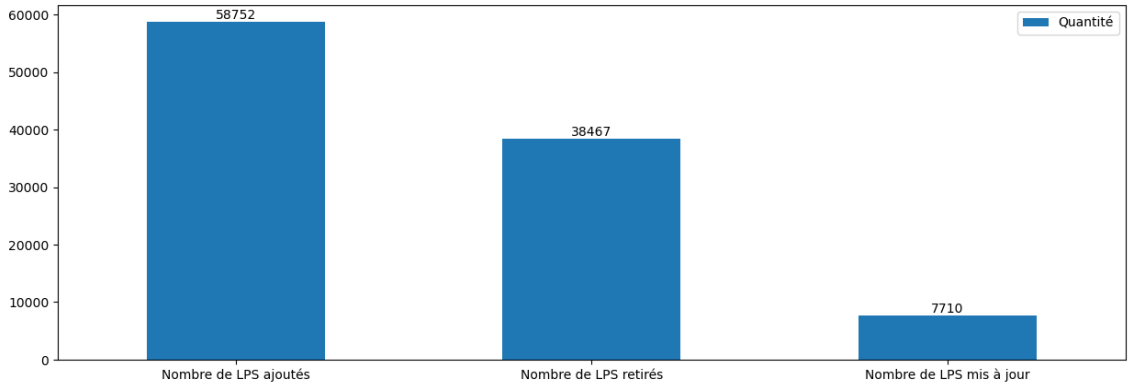

FIGURE 4.4 : Modifications du code source concernant les LPS.
©Marcelo Medeiros de Vasconcellos, 2023

QR2: COMMENT LES ACTIONS D'AJOUT, DE SUPPRESSION ET DE MISE À JOUR INFLUENCENT-ELLES LE CHANGEMENT DE LA LOGD?

Pour répondre à cette question, nous avons mené une enquête basée sur la comparaison entre des versions consécutives, tel que présenté dans la section 3.5.5. Nous avons mesuré les paramètres suivants : Nombre de LPS ajoutés (NLPSA), Nombre de LPS supprimés (NLPSS), Nombre de LPS mis à jour (NLPSM), Nombre de LPS inchangés (NLPSI), Nombre de LPS corrélés entre les versions (NLPSC), Pourcentage de LPS ajoutés (PLPSA), Pourcentage de LPS supprimés (PLPSS), Pourcentage de LPS mis à jour (PLPSM) et Pourcentage de LPS inchangés (PLPSI). Comme nous pouvons l'observer dans le tableau 4.4, la plus grande quantité de changements dans le code concernant les LPS est liée à l'ajout de nouvelles LPS, avec 58 752 enregistrements, suivie par la suppression de LPS avec 38 467 enregistrements. La mise à jour des LPS a une faible incidence si on la compare à l'ajout et à la suppression de déclarations avec seulement 7 710 enregistrements trouvés.

Pour comprendre un peu mieux ces changements, nous pouvons les visualiser par version, comme dans la figure 4.5. Nous observons qu'il existe une relation importante entre

TABLEAU 4.4 : Versions principales avec ajouts et retirés de LPS
©Marcelo Medeiros de Vasconcellos, 2023

Releases	Nombre de LPS ajoutés	Nombre de LPS retirés	Valeur absolue de la différence
1.02.01→2.02.00	8744	5919	2825
0.23.11→1.00.00	5067	7184	2117
0.20.02→0.21.00	5437	3357	2080
2.08.05→2.09.00	2364	420	1944

l'ajout et la suppression de LPS. Deux des quatre plus grands pics de changements sont liés aux changements de version, comme dans le passage de la version 0 à la version 1 (0.23.11→1.00.00) et le passage de la version 1 à la version 2 (1.02.01→2.02.00), comme observé dans le tableau 4.4. Les résultats complets peuvent être consultés dans l'annexe B.

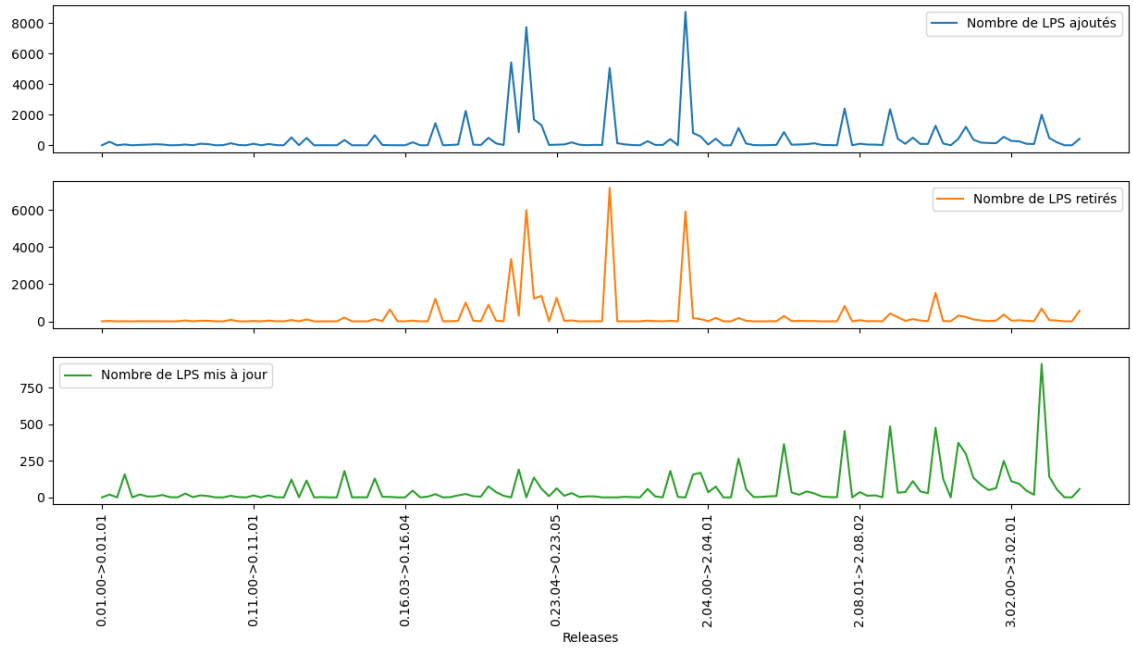

FIGURE 4.5 : Quantité de changements par version.
©Marcelo Medeiros de Vasconcellos, 2023

QR3 : QUELLES RELATIONS EXISTENT ENTRE LES ÉLÉMENTS D'UN LPS ET LA MÉTRIQUE LOGD D'UN SYSTÈME ?

Pour répondre à cette question, nous étudions les principaux changements concernant les éléments d'un LPS. Les métriques considérées sont les suivantes :

- **Nombre de LPS avec changement de niveaux de sévérité :** Lorsqu'il y a un changement dans le titre du niveau de sévérité, par exemple : les changements de WARN à WARNING sont identifiés comme des changements dans le niveau de sévérité. Pour illustrer ceci, la figure 4.6 montre un extrait de code contenant un exemple de modification du niveau de sévérité. Dans le premier exemple, il s'agit d'un changement entre les niveaux de sévérité *debug* et *trace*, dans le deuxième exemple, il s'agit d'un changement entre *warning* et *warn* et dans le troisième exemple, il s'agit d'un changement entre les instructions d'impression à la console. On rappelle que dans ce travail, nous considérons les instructions d'impression comme des LPS.

```
LOG.debug(" got #" + id)
LOG.trace(" got #" + id)

LOG.warning("Unknown task tracker polling; ignoring: " + taskTracker)
LOG.warn("Unknown task tracker polling; ignoring: " + taskTracker)

System.out.println("  out=" + output)
System.err.println("  out1=" + output)
```

FIGURE 4.6 : Exemples de changements dans les niveaux de sévérité

- **Nombre de LPS avec changement de variables :** Lorsqu'il y a un changement dans le contenu de la variable, soit un changement dans le nom de la variable ou dans le nombre de variables. La figure 4.7 montre un extrait de code avec un exemple de modification

de variables ; dans le premier exemple, on observe un changement dans le nom de la variable et dans le deuxième exemple, un changement dans le nombre de variables.

```
LOG.info("TaskTracker up at: " + this. taskReportPort)
LOG.info("TaskTracker up at: " + this. taskReportAddress)

LOG.error("META scanner", e)
LOG.error("Scan one META region: " + region.toString(), e)
```

FIGURE 4.7 : Exemple de changement de variables
©Marcelo Medeiros de Vasconcellos, 2023

- **Nombre de LPS avec changement de message statique :** Ceci inclut toute modification du message, y compris les changements de texte, la ponctuation, etc. La figure 4.8 présente un extrait de code avec un exemple de modification du message statique. Dans le premier exemple, le mot « file » et un espace ont été supprimés ; dans le deuxième exemple, un espace et un signe de ponctuation ont été supprimés.

```
LOG.info("Could not complete  file " + src + " retrying...")
LOG.info("Could not complete " + src + " retrying...")

LOG.info("Starting Periodic block scanner .")
LOG.info("Starting Periodic block scanner")
```

FIGURE 4.8 : Exemple de changement du message statique
©Marcelo Medeiros de Vasconcellos, 2023

- **Nombre de LPS avec changement de nombre de lignes :** Cette métrique inclut toute modification du nombre de lignes dans le LPS. La figure 4.9 montre un exemple de modification du nombre de lignes : le LPS comportait auparavant trois lignes et une ligne a été ajoutée à cette même déclaration.

TABLEAU 4.5 : Types de changements dans le nombre de lignes
©Marcelo Medeiros de Vasconcellos, 2023

Type de changement	Quantité de lignes	Quantité de LPS	Pourcentage
Diminution du nombre de lignes	-712	594	37.5712
Augmentation du nombre de lignes	1298	987	62.4288

```
LOG.warn("Client is requesting a new log segment " + txid +
    " though we are already writing " + curSegment + ". " +
    "Aborting the current segment in order to begin the new one.");

LOG.warn("Client is requesting a new log segment " + txid +
    " though we are already writing " + curSegment + ". " +
    "Aborting the current segment in order to begin the new one." +
    " ; journal id: " + journalId);
```

FIGURE 4.9 : Exemple de modification du nombre de lignes, du message statique et du nombre de variables
©Marcelo Medeiros de Vasconcellos, 2023

Notez que ces changements sont distincts et que le même LPS peut subir des changements dans le message statique, dans les variables, dans le niveau de sévérité et dans le nombre de lignes. Dans la figure 4.9, il est également possible d'observer que le LPS mentionné a subi des changements dans son message statique, dans le nombre de variables et dans le nombre de lignes.

Comme nous pouvons l'observer dans la figure 4.10, sur un total de 7 710 LPS mis à jour, nous avons pu identifier que 1 788 se sont répercutés sur la quantité de lignes, soit en l'augmentant ou en la diminuant. Sur ces 1 788 LPS, 594 étaient responsables de la suppression de 712 lignes et 987 étaient responsables de l'ajout de 1 298 lignes, ce qui est indiqué dans le tableau 4.5.

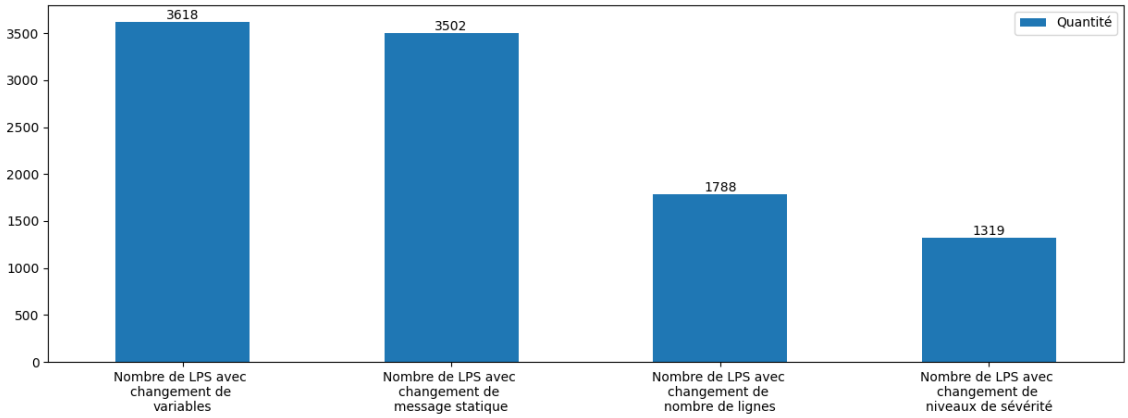

FIGURE 4.10 : Types de changements.
©Marcelo Medeiros de Vasconcellos, 2023

TABLEAU 4.6 : Relation entre la variation du nombre de lignes d'un LPS et la variation du nombre de variables
©Marcelo Medeiros de Vasconcellos, 2023

Changement du nombre de lignes	Changement du nombre de variables	Quantité de lignes	Quantité de LPS	Pourcentage
Diminution	Diminution	-624	256	30.5125
Augmentation	Diminution	20	19	2.2646
Diminution	Augmentation	-40	31	3.6949
Augmentation	Augmentation	801	533	63.5280

Nous procédons ensuite à l'étude de chaque élément d'un LPS modifié, en commençant par les variables. Dans le tableau 4.6, nous avons pu identifier que l'altération des variables est en forte relation avec la quantité de lignes : dans 94,04 % des LPSs, ces deux métriques évoluaient de manière corrélée, c'est-à-dire que si l'on augmente ou diminue le nombre de variables, on augmente ou diminue le nombre de lignes, et vice versa. La colonne *Pourcentage* du tableau 4.6 donne le pourcentage de LPS par rapport au total de LPS présenté dans le tableau.

Nous avons ensuite procédé à l'évaluation de la relation entre la taille du message et l'augmentation du nombre de lignes du journal. Encore une fois, nous avons observé un lien fort entre ces deux mesures : dans 90.94 % des cas, elles évoluaient de manière corrélée (c'est-à-dire qu'une augmentation de la taille du message était accompagnée d'une augmentation du

TABLEAU 4.7 : Relation entre l'évolution du nombre de lignes d'un LPS et l'évolution des messages.
©Marcelo Medeiros de Vasconcellos, 2023

Changement des nombre des lignes	Changement de la taille du message	Quantité de lignes	Quantité de LPS	Pourcentage
Diminution	Diminution	-775	379	32.3655
Augmentation	Diminution	52	44	3.7575
Diminution	Augmentation	-74	62	5.2946
Augmentation	Augmentation	1028	686	58.5824

TABLEAU 4.8 : Relation entre l'évolution du nombre de lignes d'un LPS et l'évolution du niveau de sévérité.
©Marcelo Medeiros de Vasconcellos, 2023

Type de changement des lignes	Changement de niveau des severité	Quantité des lignes	Quantité des LPS	Pourcentage
Diminution	Diminution de severité	-139	58	26.9767
Augmentation	Diminution de severité	88	73	33.9535
Diminution	Augmentation de severité	-52	31	14.4186
Augmentation	Augmentation de severité	81	53	24.6512

nombre de lignes, et inversement pour une réduction). Ceci peut être visualisé dans le tableau 4.7, où la colonne *Pourcentage* indique le pourcentage de LPS par rapport au total de LPS présenté dans le tableau.

Nous avons ensuite extrait les données relatives aux niveaux de sévérité. Selon le tableau 4.8, nous pouvons conclure que les deux métriques présentent cette fois-ci une faible corrélation ; la colonne *Pourcentage* indique le pourcentage de LPS par rapport au total de LPS présenté dans le tableau.

4.2 DISCUSSION

Dans cette section, nous discutons de l'évaluation obtenue à partir de la mise en œuvre de l'étude de cas dans la section 4.1. En complément, pour atteindre l'objectif défini pour ce mémoire, nous listons les réponses aux questions de recherche QR1, QR2 et QR3.

TABLEAU 4.9 : Validation de l'approche
©Marcelo Medeiros de Vasconcellos, 2023

	SLOC	LOLC	SLOC / LOLC
Chen *et al.* [1]	891,627	19,057	47
Notre approche	891,648	19,171	46,510

4.2.1 RÉPONSES AUX QUESTIONS DE RECHERCHE

QR1 : Quelle est la tendance suivie par la LOGD à mesure que le système évolue ?

On a vu que Yuan [52] avait conclu dans son travail qu'il y avait en moyenne 30 lignes de code de fonctionnalité pour chaque ligne de code de journalisation. De leur côté, Chen *et al.* [1] ont calculé cette même métrique dans plusieurs projets différents et ont identifié que la densité de code varie en fonction de la catégorie du système considéré. Notre étude fournit une information complémentaire en montrant que cette valeur varie au cours de la vie d'un projet ; au début, elle a tendance à fluctuer davantage, puis à se stabiliser au fil du temps. Cette tendance peut être mieux mesurée en utilisant la différence entre la densité de journaux de la version actuelle et la densité de journaux de la version précédente, nous permettant ainsi de suivre cette valeur. Il est même envisageable de proposer cette métrique comme mesure de la stabilité d'un système.

QR1 : *La LOGD tend à se stabiliser au cours du développement du logiciel.*

Nous avons également effectué une comparaison avec les données analysées dans le cadre de la recherche menée par Chen *et al.* [1] concernant le projet Hadoop dans sa version 2.6.0. Le résultat est présenté dans le tableau 4.9 et montre que les nombres extraits par notre approche ont une grande similarité avec les valeurs calculées par ces chercheurs.

QR2 : Comment les actions d'ajout, de suppression et de mise à jour influencent-elles le changement de la LOGD ?

Au cours de l'évolution du système, avec l'augmentation du code de fonctionnalité, il est nécessaire d'augmenter le code de journalisation. Cela se produit principalement dans les premières versions, où la densité de journalisation est affectée, principalement par l'ajout de LPS. D'ailleurs, au cours du développement, on observe globalement davantage d'ajouts que de suppressions de LPS.

QR2 : *Dans les premières versions du logiciel et dans les changements entre les versions avec la restructuration du logiciel, ces changements ont tendance à déstabiliser cette mesure. Cependant, au cours de l'évolution du système, dans les changements apportés à la même version majeure, cette mesure reste stable dans la plupart des cas.*

QR3 : Quelles relations existent entre les éléments d'un LPS et la métrique LOGD d'un système ?

L'augmentation de la quantité de variables et l'augmentation de la taille du message a une corrélation directe avec l'augmentation du nombre de lignes de LPS, mais le niveau de sévérité a une corrélation inverse avec le nombre de lignes. Autrement dit, plus le niveau de sévérité est élevé, plus le nombre de lignes de LPS de ce niveau est faible. Cependant, avec l'évolution du code du système, le niveau de sévérité moyen a tendance à diminuer, ce qui peut entraîner une augmentation de la densité des logs. Toutefois, cette évolution est insignifiante si on la compare au taux de croissance du code des fonctionnalités du système.

QR3 : *Le nombre de variables et la taille du message ont une relation directe avec le LOGD ; s'ils augmentent ou diminuent, le LOGD aura également tendance à augmenter ou à diminuer.*

4.2.2 AUTRES OBSERVATIONS

Au cours de l'expérimentation, nous avons éprouvé certaines difficultés avec l'extraction des LPS. Cela se produit parce que, malgré la normalisation offerte par les bibliothèques de

journalisation actuelles, dans plusieurs cas les développeurs utilisent toujours des instructions d'impression à la console, lesquelles sont par définition moins structurées et plus difficiles à analyser. Dans le cas de notre expérience, nous avons observé 202 486 instructions d'impression et 813 350 LPS, c'est-à-dire qu'environ 19,93 % des points d'enregistrement dans nos expériences ont été exécutés par des instructions d'impression au lieu de LPS. De plus, les développeurs incluent parfois des fonctions prenant des arguments en tant que variables dynamiques dans le message d'un LPS, et la construction de ces messages peut être effectuée par concaténation de chaînes ou par inclusion de paramètres. Dans ces cas, il peut être difficile de connaître exactement le message produit par une instruction, étant donné qu'une partie de son contenu est généré au moment de l'exécution.

Certaines autres difficultés ont été rencontrées au cours du développement du logiciel, que nous mentionnons ci-dessous.

- **Identification des changements dans les versions consécutives :** Au début de notre recherche, nous pensions considérer uniquement les lignes de code ayant subi des modifications entre deux versions successives. Cependant, après quelques expériences, nous avons reconnu qu'en visualisant uniquement les lignes qui changent, nous pouvions parfois perdre une partie d'une instruction, et donc interpréter incorrectement une modification (comme le montre la ligne 215 de la figure 3.32). Cela nous a obligés à revoir la solution de manière à lire toutes les LPS existantes dans chaque fichier et à les comparer. Cependant, il existe des cas où les messages et/ou les méthodes sont déplacés et dont les messages sont très similaires. Dans ce cas, la similarité cosinus, utilisant uniquement le LPS comme point de comparaison peut ne pas être en mesure de regrouper correctement les paires entre deux fichiers. Pour résoudre ces cas, nous incluons également dans le texte utilisé pour le calcul de la similarité cosinus le nom de la fonction dans laquelle le LPS est inséré.

- **Modification de la position d'un LPS :** Lorsqu'on trouve plus d'un LPS dans le jeu de modifications identifié par *DiffLib*, on doit faire la jointure de toutes les modifications existantes dans un fichier et comparer toutes les modifications entre elles, ce qui complexifie l'opération.
- **Changement de nom de fichier :** L'outil ne fait pas le suivi des fichiers qui changent de nom en cours de projet. Autrement dit, lorsque le nom du fichier est modifié, notre approche stocke les déclarations comme supprimées et les déclarations présentes dans le nouveau fichier sont enregistrées comme nouvelles. Ce résultat peut partiellement affecter les recherches axées sur les changements de LPS, car dans ce cas, l'historique des LPS est perdu. Nous avons travaillé en tenant compte de cette limitation dans notre recherche, cette limitation peut partiellement affecter QR2 et QR3, mais elle n'affecte pas le résultat global de la recherche, parce que dans le cas du calcul de la densité du journal, le système effectue le calcul total, quels que soient les noms de fichiers. Nous avons observé que, dans notre étude, 1 054 014 rapprochements entre versions consécutives ont été extraits et que, parmi ceux-ci, seules 104 929 opérations d'inclusion, de modification ou d'exclusion ont été identifiées, ce qui équivaut à 9,96 % du montant total. En d'autres termes, 90,04 % des enregistrements sont restés inchangés et ont été identifiés directement.

CHAPITRE V
CONCLUSION

5.1 REVUE DES CONTRIBUTIONS

Dans ce travail, nous avons proposé d'étudier l'évolution des instructions de journalisation dans le code source d'un projet Java. Pour ce faire, nous avons présenté le design et l'implémentation d'un outil, appelé *sLogAnalyzer*, qui permet de repérer automatiquement les instructions de journalisation d'un corpus de code au moyen d'expressions régulières, en extraire diverses caractéristiques et les sauvegarder dans une base de données. Il est alors possible d'interroger cette base afin de calculer diverses métriques sur ces instructions de journalisation ; l'outil permet également la comparaison des instructions entre plusieurs versions d'un même projet.

Ce dernier élément s'avère particulièrement original. En effet, les travaux antérieurs se sont concentrés sur l'étude de la densité des LPS dans des versions uniques d'un système, sans étudier son comportement tout au long du développement. Notre analyse du projet Apache Hadoop nous a permis d'observer que le code des fonctionnalités évolue plus rapidement que le code des journaux ; cependant, dans ce projet précis, ces deux mesures ont commencé à évoluer dans une proportion presque constante à partir de la version 2.2.0.

En utilisant la différence entre la LOGD d'une version et de celle qui précède, nous avons pu définir une mesure du processus logiciel qui suit l'évolution du code qui tient compte du code de journalisation. Nous pouvons donc conclure que la différence entre la LOGD de deux versions successives peut être utilisée comme mesure de contrôle de la stabilité du code. Une valeur négative de cette métrique signifie une augmentation du code des journaux, alors qu'une valeur positive signifie une réduction du code des journaux. Chaque projet peut

définir sa limite de contrôle supérieure et sa limite de contrôle inférieure en fonction de ses besoins, de manière à pouvoir surveiller et contrôler correctement la non-dépréciation du code de journalisation.

Les journaux capturent les préoccupations et les questions des développeurs concernant le code [18] ; en conséquence, ces derniers ont tendance à incorporer plus de LPS aux points critiques et complexes du code, entraînant par le fait même une augmentation de la LOGD du système. La réduction de l'LOGD peut également constituer un problème, car elle peut résulter de l'ajout de nouveau code de fonctionnalité qui n'est pas accompagné d'une journalisation appropriée. Le suivi de ces indicateurs pourrait donc aider les développeurs à réduire les différences de code entre les versions et, par conséquent, faire en sorte que les systèmes soient déployés plus rapidement et avec une plus grande stabilité du code —réduisant ainsi les défauts après la publication. C'est en ce sens que l'on peut affirmer que la différence de LOGD entre des versions consécutives est une mesure qui indique la stabilité du code du système.

Pour illustrer ce à quoi pourrait ressembler l'utilisation de l'indicateur de différence LOGD, nous utiliserons la figure 5.1. Supposons que l'équipe de déploiement d'un projet impose que la variation de LOGD soit toujours comprise entre -0,002 et 0,002. Les versions du logiciel dont les différences LOGD sont supérieures à 0,002 ou inférieures à -0,002 ne seraient alors pas acceptées ; dans le cas précis de Hadoop, cela signifierait que les versions 0.2.0, 0.13.0, 0.19.0, 0.20.0, 0.23.0, 1.0.0 et 2.2.0 n'auraient pas été publiées, ou alors l'auraient été en tenant compte d'un certain « risque ». L'équipe de développement pourrait même surveiller cet indicateur pour définir le moment où une pré-version est prête à devenir une version officielle.

Bien que nous ne puissions pas garantir que les versions 0.2.0, 0.13.0, 0.19.0, 0.20.0, 0.23.0, 1.0.0 et 2.2.0 comportent plus de bogues que les autres versions, notre approche a permis d'identifier les versions dans lesquelles des changements structurels se produiront au

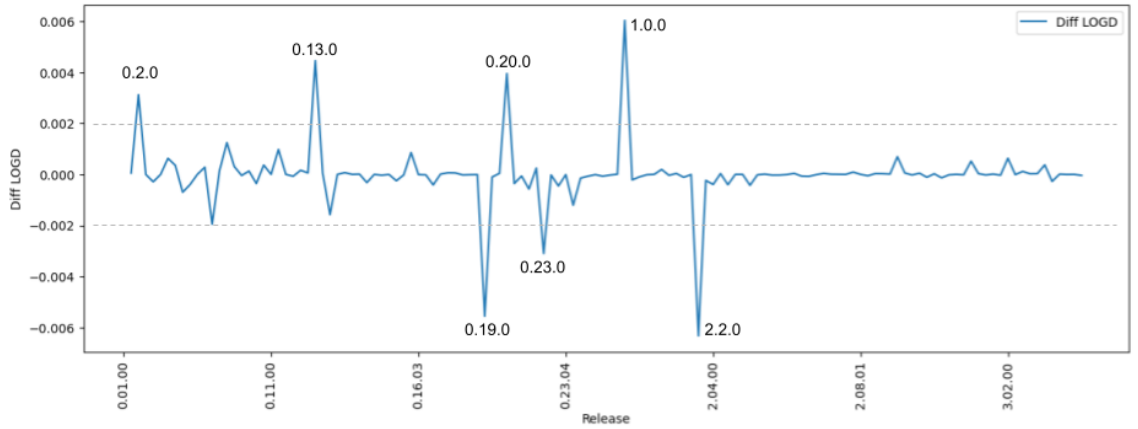

FIGURE 5.1 : Exemple de contrôle par la différence de LOGD
©Marcelo Medeiros de Vasconcellos, 2023

cours du développement du projet Hadoop et, par conséquent, plus la quantité de changements, en particulier structurels, est importante, plus la possibilité de bogues est grande.

Dans cette optique, l'outil *sLogAnalyzer* développé dans le cadre de ce projet pourrait aider les chercheurs et les ingénieurs logiciel à examiner les LPS dans le but d'améliorer la qualité des systèmes. De plus, bien que notre travail se concentre sur la LOGD, notre outil peut être utilisé à d'autres fins, telles que l'analyse des messages, l'analyse du niveau de sévérité, l'analyse des variables, l'analyse de la localisation des LPS et l'extraction de données pour une application dans des modèles d'apprentissage automatique, entre autres.

5.2 LIMITES DE L'APPROCHE PROPOSÉE ET TRAVAUX FUTURS

Notre approche se limite à l'étude de la LOGD dans des fichiers Java dans un projet open source. Bien que nous surveillons les performances de notre pipeline, notre étude n'est pas axée sur la vitesse d'extraction, mais sur la qualité de cette extraction. De la même façon, même si l'outil fournit quelques pages et filtres pour la recherche, nous ne nous concentrons pas sur le développement d'une solution qui présente toutes les possibilités de recherche au

moyen d'une interface graphique, nous nous concentrons plutôt sur la qualité de l'extraction, dans ce cas le chercheur devrait utiliser un client SQL pour accéder directement à la base de données et pour effectuer la recherche et tirer le meilleur parti des données.

La généralisation des conclusions tirées des données expérimentales quant à la LOGD et sa relation avec d'autres éléments du code est également limitée. En effet, notre preuve de concept s'est cantonnée à l'étude des versions d'un seul système (Apache Hadoop). Les mêmes opérations devraient être répétées sur d'autres systèmes afin de confirmer ou d'infirmer les tendances observées lors de l'étude de Hadoop. Il est cependant rassurant de constater que, sur les métriques qui ont été mesurées par d'autres travaux indépendants (principalement la densité des logs elle-même), les résultats que nous obtenons vont dans le même sens que ceux déjà publiés.

À la lumière de ces observations, nous suggérons le développement de futurs travaux connexes axés sur la surveillance métrique et le développement d'outils couvrant les points suivants :

- Reproduction de cette étude dans d'autres systèmes (autant open source qu'à code source fermé) ;
- Développement de l'outil pour qu'il puisse analyser des fichiers dans d'autres langages de programmation ;
- Développement de l'outil visant à améliorer les performances et le temps d'exécution du pipeline ;
- Application de LOGD en tant que méthodologie de suivi de la qualité au cours du développement d'un système.

BIBLIOGRAPHIE

[1] B. Chen *et al.*, “Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation,” *Empirical Software Engineering*, vol. 22, n° 1, pp. 330–374, 2017.

[2] W. Xu, L. Huang, A. Fox, D. Patterson, et M. I. Jordan, “Detecting large-scale system problems by mining console logs,” dans *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 117–132.

[3] Z. M. Jiang, A. E. Hassan, G. Hamann, et P. Flora, “Automatic identification of load testing problems,” dans *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 307–316.

[4] Q. Fu, J.-G. Lou, Y. Wang, et J. Li, “Execution anomaly detection in distributed systems through unstructured log analysis,” dans *2009 ninth IEEE international conference on data mining*. IEEE, 2009, pp. 149–158.

[5] M. Du, F. Li, G. Zheng, et V. Srikumar, “Deeplog : Anomaly detection and diagnosis from system logs through deep learning,” dans *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 1285–1298.

[6] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, et S. Pasupathy, “Sherlog : error diagnosis by connecting clues from run-time logs,” dans *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, 2010, pp. 143–154.

[7] K. Nagaraj, C. E. Killian, et J. Neville, “Structured comparative analysis of systems logs to diagnose performance problems.” dans *NSDI*, n° 1, 2012, p. 353.

[8] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, et P. Flora, “Leveraging performance counters and execution logs to diagnose memory-related performance issues,” dans *2013 IEEE international conference on software maintenance*. IEEE, 2013, pp. 110–119.

[9] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, et M. Stumm, “Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle.” dans *Osdi*, 2016, pp. 603–618.

[10] Y. Zhang, S. Makarov, X. Ren, D. Lion, et D. Yuan, “Pensieve : Non-intrusive failure reproduction for distributed systems using the event chaining approach,” dans *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 19–33.

[11] M. Aharon, G. Barash, I. Cohen, et E. Mordechai, “One graph is worth a thousand logs : Uncovering hidden structures in massive system event logs,” dans *Machine Learning and Knowledge Discovery in Databases : European Conference, ECML PKDD 2009, Bled, Slovenia, September 7-11, 2009, Proceedings, Part I 20*. Springer, 2009, pp. 227–243.

[12] A. A. Makanju, A. N. Zincir-Heywood, et E. E. Milios, “Clustering event logs using iterative partitioning,” dans *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’09. New York, NY, USA : Association for Computing Machinery, 2009, p. 1255–1264. [En ligne]. Repéré à : https://doi.org/10.1145/1557019.1557154

[13] R. Vaarandi et M. Pihelgas, “Logcluster - a data clustering and pattern mining algorithm for event logs,” dans *2015 11th International Conference on Network and Service Management (CNSM)*, 2015, pp. 1–7.

[14] T. Barik, R. DeLine, S. Drucker, et D. Fisher, “The bones of the system : A case study of logging and telemetry at microsoft,” dans *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016, pp. 92–101.

[15] I. Y. M. Al-Mahbashi, M. Potdar, et P. Chauhan, “Network security enhancement through effective log analysis using elk,” dans *2017 International Conference on Computing Methodologies and Communication (ICCMC)*. IEEE, 2017, pp. 566–570.

[16] M. J. Turcotte, N. A. Heard, et A. D. Kent, “Modelling user behaviour in a network using computer event logs,” dans *Dynamic Networks and Cyber-Security*. World Scientific, 2016, pp. 67–87.

[17] G. Lee, J. Lin, C. Liu, A. Lorek, et D. V. Ryaboy, “The unified logging infrastructure for data analytics at twitter,” *CoRR*, vol. abs/1208.4171, 2012. [En ligne]. Repéré à : http://arxiv.org/abs/1208.4171

[18] W. Shang, M. Nagappan, et A. E. Hassan, “Studying the relationship between logging characteristics and the code quality of platform software,” *Empirical Software Engineering*, vol. 20, n° 1, pp. 1–27, 2015.

[19] J. Mertens, “Ios 14 : Un nouveau bug remet les applications d’apple par défaut,” Oct 2020. [En ligne]. Repéré à : https://belgium-iphone.lesoir.be/2020/10/22/ios-14-un-nouveau-bug-remet-les-applications-dapple-par-defaut/

[20] S. Wu, H. Hamza, et M. E. Fayad, “Implementing pattern languages using stability concepts,” dans *meeting of ChiliPLoP*, vol. 3. Citeseer, 2003.

[21] T. Yin, “Lizard : A simple code complexity analyser,” Sep 2013. [En ligne]. Repéré à : https://pypi.org/project/lizard/

[22] P. S. Foundation, “difflib — helpers for computing deltas,” 2018. [En ligne]. Repéré à : https://docs.python.org/3/library/difflib.html

[23] S. Kabinna, C.-P. Bezemer, W. Shang, et A. E. Hassan, “Logging library migrations : A case study for the apache software foundation projects,” dans *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 154–164.

[24] S. Gholamian et P. A. Ward, “A comprehensive survey of logging in software : From logging statements automation to log mining and analysis,” *arXiv preprint arXiv :2110.12489*, 2021.

[25] B. Chen et Z. M. Jiang, “Characterizing and detecting anti-patterns in the logging code,” dans *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 71–81.

[26] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, et T. Xie, “Where do developers log ? an empirical study on logging practices in industry,” dans *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 24–33.

[27] A. Pecchia, M. Cinque, G. Carrozza, et D. Cotroneo, “Industry practices and event logging : Assessment of a critical software development process,” dans *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 169–178.

[28] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, et Y. Zhou, “Log20 : Fully automated optimal placement of log printing statements under specified overhead threshold,” dans

Proceedings of the 26th Symposium on Operating Systems Principles, 2017, pp. 565–581.

[29] S. Gholamian et P. A. Ward, "What distributed systems say : A study of seven spark application logs," dans *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2021, pp. 222–232.

[30] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, et T. Xie, "Log2 : A cost-aware logging mechanism for performance diagnosis," dans *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, 2015, pp. 139–150.

[31] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, et S. Savage, "Be conservative : Enhancing failure diagnosis with proactive logging," dans *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 293–306.

[32] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, et D. Zhang, "Learning to log : Helping developers make informed logging decisions," dans *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 415–425.

[33] E. Mendes et F. Petrillo, "Log severity levels matter : A multivocal mapping," dans *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021, pp. 1002–1013.

[34] B. Chen et Z. M. Jiang, "A survey of software log instrumentation," *ACM Computing Surveys (CSUR)*, vol. 54, n° 4, pp. 1–34, 2021.

[35] W. Shang, M. Nagappan, A. E. Hassan, et Z. M. Jiang, "Understanding log lines using development knowledge," dans *2014 IEEE international conference on software maintenance and evolution*. IEEE, 2014, pp. 21–30.

[36] D. Yuan, J. Zheng, S. Park, Y. Zhou, et S. Savage, "Improving software diagnosability via log enhancement," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, n° 1, pp. 1–28, 2012.

[37] B. Chen et Z. M. Jiang, "Studying the use of java logging utilities in the wild," dans *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 397–408.

[38] T. Mizouchi, K. Shimari, T. Ishio, et K. Inoue, “Padla : a dynamic log level adapter using online phase detection,” dans *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 135–138.

[39] S. Kabinna, C.-P. Bezemer, W. Shang, M. D. Syer, et A. E. Hassan, “Examining the stability of logging statements,” *Empirical Software Engineering*, vol. 23, n° 1, pp. 290–333, 2018.

[40] W. Shang, M. Nagappan, et A. E. Hassan, “Studying the relationship between logging characteristics and the code quality of platform software,” *Empirical Software Engineering*, vol. 20, pp. 1–27, 2015.

[41] D. Yuan, S. Park, et Y. Zhou, “Characterizing logging practices in open-source software,” dans *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 102–112.

[42] B. Chen et Z. M. J. Jiang, “Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation,” *Empirical Software Engineering*, vol. 22, n° 1, pp. 330–374, 2017.

[43] “The world’s largest open source foundation.” [En ligne]. Repéré à : http://www.apache.org/

[44] C. Zhi, J. Yin, S. Deng, M. Ye, M. Fu, et T. Xie, “An exploratory study of logging configuration practice in java,” dans *2019 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2019, pp. 459–469.

[45] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, et P. Flora, “An exploratory study of the evolution of communicated information about the execution of large software systems,” *Journal of Software : Evolution and Process*, vol. 26, n° 1, pp. 3–26, 2014.

[46] S. Li, X. Niu, Z. Jia, J. Wang, H. He, et T. Wang, “Logtracker : Learning log revision behaviors proactively from software evolution history,” dans *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 178–188.

[47] S. Li, X. Niu, Z. Jia, X. Liao, J. Wang, et T. Li, “Guiding log revisions by learning

from software evolution history," *Empirical Software Engineering*, vol. 25, nº 3, pp. 2302–2340, 2020.

[48] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, et M. Stumm, "Simple testing can prevent most critical failures : An analysis of production failures in distributed data-intensive systems," dans *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 249–265.

[49] Z. Li, T.-H. Chen, J. Yang, et W. Shang, "Studying duplicate logging statements and their relationships with code clones," *IEEE Transactions on Software Engineering*, vol. 48, nº 7, pp. 2476–2494, 2021.

[50] ——, "Dlfinder : characterizing and detecting duplicate logging code smells," dans *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 152–163.

[51] M. Hassani, W. Shang, E. Shihab, et N. Tsantalis, "Studying and detecting log-related issues," *Empirical Software Engineering*, vol. 23, nº 6, pp. 3248–3280, 2018.

[52] D. Yuan, S. Park, et Y. Zhou, "Characterizing logging practices in open-source software," dans *2012 34th international conference on software engineering (ICSE)*. IEEE, 2012, pp. 102–112.

[53] W. Shang, Z. M. Jiang, B. Adams, et A. E. Hassan, "Mapreduce as a general framework to support research in mining software repositories (msr)," dans *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 21–30.

[54] B. Fluri, M. Wursch, M. PInzger, et H. Gall, "Change distilling : Tree differencing for fine-grained source code change extraction," *IEEE Transactions on software engineering*, vol. 33, nº 11, pp. 725–743, 2007.

[55] F. Dib, "Build, test, and debug regex." [En ligne]. Repéré à : http://www.regex101.com/

[56] A. Singhal *et al.*, "Modern information retrieval : A brief overview," *IEEE Data Eng. Bull.*, vol. 24, nº 4, pp. 35–43, 2001.

[57] P.-N. Tan, M. Steinbach, et V. Kumar, “Introduction to data mining. ed,” *Addison-Wesley Longman Publishing Co., Inc.*, 2005.

[58] P. Runeson, M. Host, A. Rainer, et B. Regnell, *Case study research in software engineering : Guidelines and examples*. John Wiley & Sons, 2012.

APPENDICE A

ANALYSE DES LPS PAR VERSION

TABLEAU A.1 : Analyse des LPS par version
©Marcelo Medeiros de Vasconcellos, 2023

	Release	Nombre de LPS	LOLC	SLOC	Densité des Logs	Diff LOGD
1	0.01.00	289	375	15511	0.0242	NaN
2	0.01.01	290	376	15519	0.0242	0.0001
3	0.02.00	501	712	26033	0.0273	0.0031
4	0.02.01	501	712	26039	0.0273	-0.0000
5	0.03.00	550	754	27872	0.0271	-0.0003
6	0.03.01	550	754	27875	0.0270	-0.0000
7	0.03.02	569	782	28254	0.0277	0.0006
8	0.04.00	602	828	29541	0.0280	0.0004
9	0.05.00	662	912	33371	0.0273	-0.0007
10	0.06.00	707	982	36465	0.0269	-0.0004
11	0.06.01	707	982	36462	0.0269	0.0000
12	0.06.02	717	993	36490	0.0272	0.0003
13	0.07.00	717	1025	40579	0.0253	-0.0020
14	0.07.01	721	1032	40617	0.0254	0.0001
15	0.08.00	789	1120	42022	0.0267	0.0012
16	0.09.00	839	1189	44103	0.0270	0.0003
17	0.09.01	837	1187	44101	0.0269	-0.0000
18	0.09.02	847	1202	44443	0.0270	0.0001
19	0.10.00	902	1287	48229	0.0267	-0.0004
20	0.10.01	924	1326	49023	0.0270	0.0004
21	0.11.00	924	1326	49023	0.0270	0.0000
22	0.11.01	1004	1472	52524	0.0280	0.0010
23	0.11.02	1004	1472	52525	0.0280	-0.0000
24	0.12.00	1049	1564	55957	0.0280	-0.0001
25	0.12.01	1059	1584	56351	0.0281	0.0002
26	0.12.03	1062	1589	56415	0.0282	0.0001
27	0.13.00	1507	2288	70140	0.0326	0.0045
28	0.13.01	1510	2294	70201	0.0327	0.0001
29	0.14.00	1887	2905	93419	0.0311	-0.0016
30	0.14.01	1887	2905	93420	0.0311	-0.0000
31	0.14.02	1892	2917	93603	0.0312	0.0001
32	0.14.03	1892	2917	93605	0.0312	-0.0000
33	0.14.04	1893	2918	93614	0.0312	0.0000
34	0.15.00	2035	3304	107108	0.0308	-0.0003
35	0.15.01	2036	3306	107176	0.0308	-0.0000
36	0.15.02	2037	3307	107327	0.0308	-0.0000
37	0.15.03	2037	3307	107341	0.0308	-0.0000
38	0.16.00	2580	4230	138429	0.0306	-0.0003
39	0.16.01	2597	4253	139282	0.0305	-0.0000
40	0.16.02	1963	3207	102160	0.0314	0.0009

	Release	Nombre de LPS	LOLC	SLOC	Densité des Logs	Diff LOGD
41	0.16.03	1964	3209	102252	0.0314	-0.0000
42	0.16.04	1963	3208	102283	0.0314	-0.0000
43	0.17.00	2123	3490	112766	0.0309	-0.0004
44	0.17.01	2124	3491	112754	0.0310	0.0000
45	0.17.02	2127	3501	112851	0.0310	0.0001
46	0.18.00	2354	3833	123328	0.0311	0.0001
47	0.18.01	2354	3833	123417	0.0311	-0.0000
48	0.18.02	2369	3850	124019	0.0310	-0.0000
49	0.18.03	2393	3888	125282	0.0310	-0.0000
50	0.19.00	3631	5731	224974	0.0255	-0.0056
51	0.19.01	3642	5753	226764	0.0254	-0.0001
52	0.19.02	3656	5778	227373	0.0254	0.0000
53	0.20.00	3249	5283	179917	0.0294	0.0040
54	0.20.01	3344	5464	188394	0.0290	-0.0004
55	0.20.02	3357	5479	189328	0.0289	-0.0001
56	0.21.00	5437	9022	318080	0.0284	-0.0006
57	0.22.00	5990	9811	342912	0.0286	0.0002
58	0.23.00	7745	12930	506851	0.0255	-0.0031
59	0.23.01	8206	13763	539979	0.0255	-0.0000
60	0.23.03	8153	13652	545370	0.0250	-0.0005
61	0.23.04	8170	13676	546508	0.0250	-0.0000
62	0.23.05	6939	11712	491769	0.0238	-0.0012
63	0.23.06	6965	11743	496162	0.0237	-0.0001
64	0.23.07	7109	11967	507158	0.0236	-0.0001
65	0.23.08	7143	12022	509711	0.0236	-0.0000
66	0.23.09	7146	12023	511319	0.0235	-0.0001
67	0.23.10	7169	12066	513739	0.0235	-0.0000
68	0.23.11	7184	12089	514585	0.0235	0.0000
69	1.00.00	5067	8363	283250	0.0295	0.0060
70	1.00.01	5203	8585	292916	0.0293	-0.0002
71	1.00.02	5254	8669	296775	0.0292	-0.0001
72	1.00.03	5268	8681	297322	0.0292	-0.0000
73	1.00.04	5268	8681	297325	0.0292	-0.0000
74	1.01.00	5504	9166	311869	0.0294	0.0002
75	1.01.01	5520	9197	313328	0.0294	-0.0000
76	1.01.02	5540	9227	313965	0.0294	0.0000
77	1.02.00	5913	9775	333929	0.0293	-0.0001
78	1.02.01	5919	9784	334299	0.0293	-0.0000
79	2.02.00	8744	14925	650608	0.0229	-0.0063
80	2.03.00	9380	16093	708840	0.0227	-0.0002
81	2.04.00	9835	16911	758110	0.0223	-0.0004
82	2.04.01	9879	16973	759803	0.0223	0.0000
83	2.05.00	10128	17427	794566	0.0219	-0.0004
84	2.05.01	10128	17427	794620	0.0219	-0.0000
85	2.05.02	10128	17427	794659	0.0219	-0.0000
86	2.06.00	11082	19171	891648	0.0215	-0.0004
87	2.06.01	11168	19342	900300	0.0215	-0.0000
88	2.06.02	11181	19362	900893	0.0215	0.0000
89	2.06.03	11181	19360	902012	0.0215	-0.0000
90	2.06.04	11190	19381	904151	0.0214	-0.0000
91	2.06.05	11209	19418	906215	0.0214	-0.0000
92	2.07.00	11786	20342	947644	0.0215	0.0000
93	2.07.01	11809	20361	951629	0.0214	-0.0001
94	2.07.02	11835	20413	957552	0.0213	-0.0001
95	2.07.03	11891	20536	963813	0.0213	-0.0000
96	2.07.04	12005	20754	972249	0.0213	0.0000
97	2.07.05	12028	20790	973689	0.0214	0.0000
98	2.07.06	12040	20817	974894	0.0214	0.0000
99	2.07.07	12041	20819	975063	0.0214	-0.0000
100	2.08.00	13621	23787	1109385	0.0214	0.0001

	Release	Nombre de LPS	LOLC	SLOC	Densité des Logs	Diff LOGD
101	2.08.01	13621	23787	1109424	0.0214	-0.0000
102	2.08.02	13665	23933	1119055	0.0214	-0.0001
103	2.08.03	13710	24041	1122503	0.0214	0.0000
104	2.08.04	13743	24115	1124553	0.0214	0.0000
105	2.08.05	13758	24152	1125702	0.0215	0.0000
106	2.09.00	15702	29119	1314730	0.0221	0.0007
107	2.09.01	15914	29513	1329216	0.0222	0.0001
108	2.09.02	15984	29648	1336745	0.0222	-0.0000
109	2.10.00	16362	30670	1379970	0.0222	0.0000
110	2.10.01	16404	30777	1391888	0.0221	-0.0001
111	2.10.02	16483	30967	1399000	0.0221	0.0000
112	3.00.00	16237	30687	1394989	0.0220	-0.0001
113	3.00.01	16332	30855	1403736	0.0220	-0.0000
114	3.00.02	16332	30855	1403736	0.0220	0.0000
115	3.00.03	16443	31116	1416742	0.0220	-0.0000
116	3.01.00	17420	33605	1494792	0.0225	0.0005
117	3.01.01	17684	34161	1517653	0.0225	0.0000
118	3.01.02	17814	34423	1530893	0.0225	-0.0000
119	3.01.03	17946	34728	1543783	0.0225	0.0000
120	3.01.04	18042	35036	1559748	0.0225	-0.0000
121	3.02.00	18231	36365	1574508	0.0231	0.0006
122	3.02.01	18481	36913	1599074	0.0231	-0.0000
123	3.02.02	18685	37764	1627684	0.0232	0.0001
124	3.02.03	18750	37907	1632200	0.0232	0.0000
125	3.02.04	18827	38049	1636324	0.0233	0.0000
126	3.03.00	20148	40867	1729789	0.0236	0.0004
127	3.03.01	20560	41781	1789474	0.0233	-0.0003
128	3.03.02	20717	42122	1803456	0.0234	0.0000
129	3.03.03	20720	42125	1803953	0.0234	-0.0000
130	3.03.04	20720	42125	1804040	0.0234	-0.0000
131	3.03.05	20574	42002	1801892	0.0233	-0.0000

APPENDICE B

ANALYSE DES CHANGEMENTS ENTRE LES VERSIONS

TABLEAU B.1 : Analyse des changements entre les versions
©Marcelo Medeiros de Vasconcellos, 2023

	Releases	NLPSA	NLPSS	NLPSM	NLPSI	NLPSC	PLPSA (%)	PLPSS (%)	PLPSM (%)	PLPSI (%)
1	0.01.00->0.01.01	3	2	0	287	292	1.0274	0.6849	0.0000	98.2877
2	0.01.01->0.02.00	231	20	19	251	521	44.3378	3.8388	3.6468	48.1766
3	0.02.00->0.02.01	0	0	0	501	501	0.0000	0.0000	0.0000	100.0000
4	0.02.01->0.03.00	56	7	158	336	557	10.0539	1.2567	28.3662	60.3232
5	0.03.00->0.03.01	0	0	0	550	550	0.0000	0.0000	0.0000	100.0000
6	0.03.01->0.03.02	28	9	20	521	578	4.8443	1.5571	3.4602	90.1384
7	0.03.02->0.04.00	40	7	6	556	609	6.5681	1.1494	0.9852	91.2972
8	0.04.00->0.05.00	65	5	7	590	667	9.7451	0.7496	1.0495	88.4558
9	0.05.00->0.06.00	50	5	16	641	712	7.0225	0.7022	2.2472	90.0281
10	0.06.00->0.06.01	0	0	1	706	707	0.0000	0.0000	0.1414	99.8586
11	0.06.01->0.06.02	10	0	1	706	717	1.3947	0.0000	0.1395	98.4658
12	0.06.02->0.07.00	48	48	26	643	765	6.2745	6.2745	3.3987	84.0523
13	0.07.00->0.07.01	4	0	2	715	721	0.5548	0.0000	0.2774	99.1678
14	0.07.01->0.08.00	100	32	14	675	821	12.1803	3.8977	1.7052	82.2168
15	0.08.00->0.09.00	76	26	9	754	865	8.7861	3.0058	1.0405	87.1676
16	0.09.00->0.09.01	0	2	0	837	839	0.0000	0.2384	0.0000	99.7616
17	0.09.01->0.09.02	11	1	0	836	848	1.2972	0.1179	0.0000	98.5849
18	0.09.02->0.10.00	133	78	11	758	980	13.5714	7.9592	1.1224	77.3469
19	0.10.00->0.10.01	22	0	2	900	924	2.3810	0.0000	0.2165	97.4026
20	0.10.01->0.11.00	0	0	0	924	924	0.0000	0.0000	0.0000	100.0000
21	0.11.00->0.11.01	100	20	13	891	1024	9.7656	1.9531	1.2695	87.0117
22	0.11.01->0.11.02	0	0	0	1004	1004	0.0000	0.0000	0.0000	100.0000
23	0.11.02->0.12.00	83	38	14	952	1087	7.6357	3.4959	1.2879	87.5805
24	0.12.00->0.12.01	11	1	1	1047	1060	1.0377	0.0943	0.0943	98.7736
25	0.12.01->0.12.03	3	0	0	1059	1062	0.2825	0.0000	0.0000	99.7175
26	0.12.03->0.13.00	514	69	122	871	1576	32.6142	4.3782	7.7411	55.2665
27	0.13.00->0.13.01	3	0	1	1506	1510	0.1987	0.0000	0.0662	99.7351
28	0.13.01->0.14.00	482	105	115	1290	1992	24.1968	5.2711	5.7731	64.7590
29	0.14.00->0.14.01	0	0	0	1887	1887	0.0000	0.0000	0.0000	100.0000
30	0.14.01->0.14.02	5	0	2	1885	1892	0.2643	0.0000	0.1057	99.6300
31	0.14.02->0.14.03	0	0	0	1892	1892	0.0000	0.0000	0.0000	100.0000
32	0.14.03->0.14.04	1	0	0	1892	1893	0.0528	0.0000	0.0000	99.9472
33	0.14.04->0.15.00	345	203	181	1509	2238	15.4155	9.0706	8.0876	67.4263
34	0.15.00->0.15.01	1	0	0	2035	2036	0.0491	0.0000	0.0000	99.9509
35	0.15.01->0.15.02	1	0	1	2035	2037	0.0491	0.0000	0.0491	99.9018
36	0.15.02->0.15.03	0	0	0	2037	2037	0.0000	0.0000	0.0000	100.0000
37	0.15.03->0.16.00	660	117	129	1791	2697	24.4716	4.3382	4.7831	66.4071
38	0.16.00->0.16.01	21	4	4	2572	2601	0.8074	0.1538	0.1538	98.8850
39	0.16.01->0.16.02	6	640	3	1954	2603	0.2305	24.5870	0.1153	75.0672
40	0.16.02->0.16.03	3	2	0	1961	1966	0.1526	0.1017	0.0000	99.7457

	Releases	NLPSA	NLPSS	NLPSM	NLPSI	NLPSC	PLPSA (%)	PLPSS (%)	PLPSM (%)	PLPSI (%)
41	0.16.03->0.16.04	0	1	0	1963	1964	0.0000	0.0509	0.0000	99.9491
42	0.16.04->0.17.00	197	37	46	1880	2160	9.1204	1.7130	2.1296	87.0370
43	0.17.00->0.17.01	1	0	0	2123	2124	0.0471	0.0000	0.0000	99.9529
44	0.17.01->0.17.02	3	0	5	2119	2127	0.1410	0.0000	0.2351	99.6239
45	0.17.02->0.18.00	1446	1219	22	886	3573	40.4702	34.1170	0.6157	24.7971
46	0.18.00->0.18.01	0	0	0	2354	2354	0.0000	0.0000	0.0000	100.0000
47	0.18.01->0.18.02	19	4	2	2348	2373	0.8007	0.1686	0.0843	98.9465
48	0.18.02->0.18.03	49	25	14	2330	2418	2.0265	1.0339	0.5790	96.3606
49	0.18.03->0.19.00	2255	1017	23	1353	4648	48.5155	21.8804	0.4948	29.1093
50	0.19.00->0.19.01	42	31	8	3592	3673	1.1435	0.8440	0.2178	97.7947
51	0.19.01->0.19.02	19	5	5	3632	3661	0.5190	0.1366	0.1366	99.2079
52	0.19.02->0.20.00	489	896	76	2684	4145	11.7973	21.6164	1.8335	64.7527
53	0.20.00->0.20.01	126	31	36	3182	3375	3.7333	0.9185	1.0667	94.2815
54	0.20.01->0.20.02	18	5	11	3328	3362	0.5354	0.1487	0.3272	98.9887
55	0.20.02->0.21.00	5437	3357	0	0	8794	61.8262	38.1738	0.0000	0.0000
56	0.21.00->0.22.00	853	300	191	4946	6290	13.5612	4.7695	3.0366	78.6328
57	0.22.00->0.23.00	7745	5990	0	0	13735	56.3888	43.6112	0.0000	0.0000
58	0.23.00->0.23.01	1687	1226	136	6383	9432	17.8859	12.9983	1.4419	67.6739
59	0.23.01->0.23.03	1315	1368	59	6779	9521	13.8116	14.3682	0.6197	71.2005
60	0.23.03->0.23.04	20	3	8	8142	8173	0.2447	0.0367	0.0979	99.6207
61	0.23.04->0.23.05	37	1268	62	6840	8207	0.4508	15.4502	0.7555	83.3435
62	0.23.05->0.23.06	55	29	11	6899	6994	0.7864	0.4146	0.1573	98.6417
63	0.23.06->0.23.07	190	46	29	6890	7155	2.6555	0.6429	0.4053	96.2963
64	0.23.07->0.23.08	36	2	3	7104	7145	0.5038	0.0280	0.0420	99.4262
65	0.23.08->0.23.09	7	4	7	7132	7150	0.0979	0.0559	0.0979	99.7483
66	0.23.09->0.23.10	26	3	7	7136	7172	0.3625	0.0418	0.0976	99.4980
67	0.23.10->0.23.11	18	3	0	7166	7187	0.2505	0.0417	0.0000	99.7078
68	0.23.11->1.00.00	5067	7184	0	0	12251	41.3599	58.6401	0.0000	0.0000
69	1.00.00->1.00.01	136	0	0	5067	5203	2.6139	0.0000	0.0000	97.3861
70	1.00.01->1.00.02	58	7	4	5192	5261	1.1025	0.1331	0.0760	98.6885
71	1.00.02->1.00.03	16	2	2	5250	5270	0.3036	0.0380	0.0380	99.6205
72	1.00.03->1.00.04	0	0	0	5268	5268	0.0000	0.0000	0.0000	100.0000
73	1.00.04->1.01.00	273	37	58	5173	5541	4.9269	0.6677	1.0467	93.3586
74	1.01.00->1.01.01	25	9	7	5488	5529	0.4522	0.1628	0.1266	99.2585
75	1.01.01->1.01.02	20	0	0	5520	5540	0.3610	0.0000	0.0000	99.6390
76	1.01.02->1.02.00	402	29	181	5330	5942	6.7654	0.4881	3.0461	89.7004
77	1.02.00->1.02.01	6	0	3	5910	5919	0.1014	0.0000	0.0507	99.8479
78	1.02.01->2.02.00	8744	5919	0	0	14663	59.6331	40.3669	0.0000	0.0000
79	2.02.00->2.03.00	804	168	157	8419	9548	8.4206	1.7595	1.6443	88.1755
80	2.03.00->2.04.00	578	123	168	9089	9958	5.8044	1.2352	1.6871	91.2733
81	2.04.00->2.04.01	48	4	35	9796	9883	0.4857	0.0405	0.3541	99.1197
82	2.04.01->2.05.00	436	187	74	9618	10315	4.2269	1.8129	0.7174	93.2429
83	2.05.00->2.05.01	0	0	0	10128	10128	0.0000	0.0000	0.0000	100.0000
84	2.05.01->2.05.02	0	0	0	10128	10128	0.0000	0.0000	0.0000	100.0000
85	2.05.02->2.06.00	1134	180	265	9683	11262	10.0693	1.5983	2.3530	85.9794
86	2.06.00->2.06.01	115	29	56	10997	11197	1.0271	0.2590	0.5001	98.2138
87	2.06.01->2.06.02	13	0	2	11166	11181	0.1163	0.0000	0.0179	99.8658
88	2.06.02->2.06.03	3	3	3	11175	11184	0.0268	0.0268	0.0268	99.9195
89	2.06.03->2.06.04	15	6	7	11168	11196	0.1340	0.0536	0.0625	99.7499
90	2.06.04->2.06.05	23	4	9	11177	11213	0.2051	0.0357	0.0803	99.6789

	Releases	NLPSA	NLPSS	NLPSM	NLPSI	NLPSC	PLPSA (%)	PLPSS (%)	PLPSM (%)	PLPSI (%)
91	2.06.05->2.07.00	863	286	364	10559	12072	7.1488	2.3691	3.0152	87.4669
92	2.07.00->2.07.01	37	14	34	11738	11823	0.3129	0.1184	0.2876	99.2811
93	2.07.01->2.07.02	51	25	18	11766	11860	0.4300	0.2108	0.1518	99.2074
94	2.07.02->2.07.03	72	16	41	11778	11907	0.6047	0.1344	0.3443	98.9166
95	2.07.03->2.07.04	130	16	27	11848	12021	1.0814	0.1331	0.2246	98.5609
96	2.07.04->2.07.05	23	0	6	11999	12028	0.1912	0.0000	0.0499	99.7589
97	2.07.05->2.07.06	15	3	2	12023	12043	0.1246	0.0249	0.0166	99.8339
98	2.07.06->2.07.07	1	0	1	12039	12041	0.0083	0.0000	0.0083	99.9834
99	2.07.07->2.08.00	2405	825	454	10762	14446	16.6482	5.7109	3.1427	74.4981
100	2.08.00->2.08.01	0	0	0	13621	13621	0.0000	0.0000	0.0000	100.0000
101	2.08.01->2.08.02	105	61	36	13524	13726	0.7650	0.4444	0.2623	98.5283
102	2.08.02->2.08.03	48	3	11	13651	13713	0.3500	0.0219	0.0802	99.5479
103	2.08.03->2.08.04	44	11	14	13685	13754	0.3199	0.0800	0.1018	99.4983
104	2.08.04->2.08.05	16	1	1	13741	13759	0.1163	0.0073	0.0073	99.8692
105	2.08.05->2.09.00	2364	420	486	12852	16122	14.6632	2.6051	3.0145	79.7172
106	2.09.00->2.09.01	440	228	32	15442	16142	2.7258	1.4125	0.1982	95.6635
107	2.09.01->2.09.02	95	25	36	15853	16009	0.5934	0.1562	0.2249	99.0255
108	2.09.02->2.10.00	497	119	111	15754	16481	3.0156	0.7220	0.6735	95.5889
109	2.10.00->2.10.01	86	44	41	16277	16448	0.5229	0.2675	0.2493	98.9604
110	2.10.01->2.10.02	90	11	28	16365	16494	0.5457	0.0667	0.1698	99.2179
111	2.10.02->3.00.00	1280	1526	477	14480	17763	7.2060	8.5909	2.6854	81.5178
112	3.00.00->3.00.01	117	22	127	16088	16354	0.7154	0.1345	0.7766	98.3735
113	3.00.01->3.00.02	0	0	0	16332	16332	0.0000	0.0000	0.0000	100.0000
114	3.00.02->3.00.03	416	305	374	15653	16748	2.4839	1.8211	2.2331	93.4619
115	3.00.03->3.01.00	1213	236	298	15909	17656	6.8702	1.3367	1.6878	90.1053
116	3.01.00->3.01.01	366	102	134	17184	17786	2.0578	0.5735	0.7534	96.6153
117	3.01.01->3.01.02	175	45	86	17553	17859	0.9799	0.2520	0.4815	98.2866
118	3.01.02->3.01.03	149	17	50	17747	17963	0.8295	0.0946	0.2783	98.7975
119	3.01.03->3.01.04	139	43	64	17839	18085	0.7686	0.2378	0.3539	98.6398
120	3.01.04->3.02.00	554	365	250	17427	18596	2.9791	1.9628	1.3444	93.7137
121	3.02.00->3.02.01	287	37	110	18084	18518	1.5498	0.1998	0.5940	97.6563
122	3.02.01->3.02.02	260	56	94	18331	18741	1.3873	0.2988	0.5016	97.8123
123	3.02.02->3.02.03	97	32	46	18607	18782	0.5165	0.1704	0.2449	99.0683
124	3.02.03->3.02.04	83	6	18	18726	18833	0.4407	0.0319	0.0956	99.4318
125	3.02.04->3.03.00	2009	688	913	17226	20836	9.6420	3.3020	4.3818	82.6742
126	3.03.00->3.03.01	477	65	143	19940	20625	2.3127	0.3152	0.6933	96.6788
127	3.03.01->3.03.02	198	41	55	20464	20758	0.9538	0.1975	0.2650	98.5837
128	3.03.02->3.03.03	3	0	1	20716	20720	0.0145	0.0000	0.0048	99.9807
129	3.03.03->3.03.04	0	0	0	20720	20720	0.0000	0.0000	0.0000	100.0000
130	3.03.04->3.03.05	425	571	57	20092	21145	2.0099	2.7004	0.2696	95.0201