UQAC
Université du Québec
à Chicoutimi

CADRE CONCEPTUEL POUR LES NIVEAUX DE SÉVÉRITÉ DES LOGS : DIRECTIVES ET HEURISTIQUES POUR LES PRATIQUES DE JOURNALISATION DANS LES SYSTÈMES LOGICIELS

PAR EDUARDO MENDES DE OLIVEIRA

THÈSE PRÉSENTÉE À L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI DANS LE CADRE D'UN PROGRAMME EN EXTENSION DE L'UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS EN VUE DE L'OBTENTION DU GRADE DE PHILOSOPHIÆ DOCTOR (PH.D.) EN SCIENCES ET TECHNOLOGIES DE L'INFORMATION

QUÉBEC, CANADA

RÉSUMÉ

Contexte : Les journaux (logs) jouent un rôle crucial dans la gestion des systèmes logiciels modernes, en fournissant des retours en temps réel sur le comportement du système et en aidant les développeurs et les ingénieurs d'exploitation à diagnostiquer les pannes et à surveiller la santé des systèmes. Le choix correct des niveaux de sévérité des journaux est essentiel pour s'assurer que les journaux fournissent des informations pertinentes tant en développement qu'en production. **Problème :** Malgré l'importance de la journalisation, la sélection des niveaux de sévérité des journaux n'est pas une tâche triviale et peut entraîner des difficultés dans la production de données de journalisation fiables. L'un des principaux problèmes est l'absence de directives standardisées et de conseils pratiques pour les développeurs, ce qui peut entraîner soit une journalisation excessive, soit insuffisante. Ces deux extrêmes présentent des risques : une journalisation excessive peut submerger les opérateurs avec des informations non pertinentes, tandis qu'une journalisation insuffisante peut dissimuler des détails critiques nécessaires au diagnostic des problèmes système. L'absence de distinctions claires entre les niveaux de sévérité, ainsi que l'ambiguïté quant à la criticité des événements, compliquent encore davantage la sélection des niveaux de sévérité des journaux. **Objectif :** Cette thèse vise à relever ces défis en développant un cadre conceptuel pour les niveaux de sévérité des journaux, réduisant l'ambiguïté et facilitant les processus de prise de décision. De plus, l'étude explore comment les ajustements des niveaux de sévérité des journaux se produisent au cours du développement et de l'évolution des logiciels, fournissant des informations sur les raisons de ces ajustements. **Méthode :** Tout d'abord, nous avons réalisé une cartographie systématique multivocale des niveaux de sévérité des journaux à partir de la littérature évaluée par des pairs, des bibliothèques de journalisation et des expériences pratiques des développeurs. Notre analyse a porté sur 19 niveaux de sévérité, 42 études et 40 bibliothèques de journalisation. Ensuite, nous avons étudié les ajustements de sévérité sur 376 versions de trois grands systèmes *open-source*. **Résultats :** Les résultats de la cartographie ont révélé des redondances et des chevauchements sémantiques entre les définitions des niveaux de sévérité des journaux, mais aussi une convergence vers six niveaux principaux. Notre analyse des ajustements a identifié plusieurs tendances, y compris la fréquence des changements à l'intersection des environnements de développement et de production. Sur la base de ces résultats, nous avons dérivé un ensemble de 24 heuristiques que les développeurs peuvent utiliser pour guider la sélection, la révision et l'ajustement des niveaux de sévérité des journaux. **Conclusion :** Les contributions de cette thèse sont triples : (i) une cartographie complète des pratiques en matière de niveaux de sévérité des journaux dans la littérature et les bibliothèques de journalisation, (ii) un cadre conceptuel pour les niveaux de sévérité qui fournit une nomenclature standardisée et des finalités claires pour chaque niveau, et (iii) un ensemble d'heuristiques pratiques qui offrent aux développeurs et opérateurs des conseils concrets sur la manière de choisir et d'ajuster les niveaux de sévérité des journaux.

ABSTRACT

Context : Logs play a crucial role in the management of modern software systems by providing real-time feedback on system behavior. They help developers and operations engineers diagnose failures and monitor the health of infrastructures. Correctly choosing log severity levels is essential to ensure that the collected information is relevant, both in development and production environments. **Problem :** Despite the importance of logging, the selection of log severity levels is not a trivial task and can lead to challenges in producing reliable logging data. One major issue is the absence of standardized guidelines and practical advice for developers, which can lead to either excessive or insufficient logging. Both extremes present risks : excessive logging can overwhelm operators with irrelevant information, while insufficient logging may hide critical details that are necessary for diagnosing system issues. The absence of clear distinctions between severity levels, as well as the ambiguity around event criticality, further complicates log severity level selection. **Objective :** This thesis aims to address these challenges by developing a conceptual framework for log severity levels that reduces ambiguity and helps streamline decision-making processes. Additionally, the study explores how log severity level adjustments occur during software development and evolution, providing insights into the reasons behind these adjustments. **Method :** First, we conducted a multivocal systematic mapping of log severity levels based on peer-reviewed literature, logging libraries, and developers' practical experiences. Our analysis covered 19 severity levels, 42 studies, and 40 logging libraries. Next, we investigated severity adjustments across 376 releases of three major open-source systems. **Results :** The mapping results revealed redundancies and semantic overlaps between log severity level definitions, but also a convergence toward six primary levels. Our adjustment analysis identified several trends, including the frequent occurrence of changes at the intersection of development and production environments. Based on these findings, we derived a set of 24 heuristics that developers can use to guide log severity level selection, review, and adjustment. **Conclusion :** The contributions of this thesis are threefold : (i) a comprehensive mapping of the state of log severity level practices in both the literature and logging libraries, (ii) a conceptual framework for log severity levels that provides a standardized nomenclature and clear purposes for each level, and (iii) a set of practical heuristics that offer developers and operators actionable guidance on how to choose and adjust log severity level.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ABRÉVIATIONS

ASF Apache Software Foundation
ASO Answer Stack Overflow
AST Abstract Syntax Tree
AUC Area under the curve
CDJ Catégorisation et décisions de journalisation
CI Critères d'inclusion
CE Critères d'exclusion
CPJ Caractérisation des pratiques de journalisation
CSJ Contextes spécifiques de journalisation
FAST Flow of abstract syntax tree
HBGN Hierarchical block graph network
HP Heuristique potentielle
IMS Incident management system
ITS Issue tracking system
JCL Jakarta Commons Logging
JQL Jira query language
JUL Java Utils Logging
LACC Log-aware code-clone detector
ML Machine learning
MPJ Mauvaises pratiques de journalisation
NSL Niveau de sévérite de log
OP Observation préliminaire
OR Objectifs de recherche
PYLP Popularity of programming language
QSO Questions Stack Overflow
RFC Request for comments
SLOC Source lines of code

DÉDICACE

Je dédie ce travail tout d'abord à ma famille,
en la personne de ma mère, Maria de Jesus, et de mon père, Felipe (in memoriam),
qui ont toujours souhaité que leurs enfants poursuivent des études :
ce sont eux qui m'ont mis sur le chemin de la science.
À chacune de mes sœurs et à chacun de mes frères — ils font partie de mon parcours
d'apprentissage en tant qu'être humain et être social, je les porte avec moi.
À mes nièces et à mes neveux, qui m'ont appris à accorder de la valeur à l'amour.

Je dédie aussi ce travail à chaque immigré
qui quitte sa terre natale à la recherche d'un rêve d'études
et s'aventure sur des terres étrangères sans imaginer ce qui l'attend.
J'en ai rencontré plusieurs sur ce chemin d'outre-mer
— seul(e)s nous comprenons les efforts que cela demande.

REMERCIEMENTS

En concluant ce parcours de recherche pour l'obtention de ce titre, j'ai beaucoup de personnes à remercier.

Tout d'abord Dieu, en qui j'ai placé mon espérance dans tant de moments difficiles.

Je tiens à remercier sincèrement le professeur Sylvain Hallé, qui m'a accueilli en tant qu'étudiant à un moment très délicat de mon parcours doctoral et m'a offert son soutien tant académique que financier. Merci infiniment, professeur — vous m'avez aidé à retrouver confiance en moi.

Je souhaite exprimer ma gratitude envers tout le corps professoral et administratif de l'Université du Québec à Chicoutimi, non seulement pour l'aide académique et le soutien financier, mais également pour avoir mis à ma disposition des professionnels véritablement humains qui m'ont accompagné tout au long de ce processus.

Je dois également remercier mes amis qui m'ont consolé d'innombrables fois, les amis du Brésil, les amis que j'ai faits ici au Québec, qui forment ma famille canadienne, et tant d'autres dispersés dans ce monde.

Je remercie chaleureusement mon collègue de recherche Marcelo Vasconcellos, qui a travaillé à mes côtés durant ce parcours, dans le cadre de son propre cheminement de maîtrise. Ton soutien a été indispensable pour que je puisse relever la tête.

Ma profonde reconnaissance va à ma psychologue, Mariana, qui m'a aidé à traverser les moments les plus sombres que j'ai vécus, et m'a aidé à respirer à nouveau.

Enfin, je tiens également à remercier mon chum, Paulo Eduardo, qui a apporté tant de lumière dans ma vie et qui m'a donné de nouvelles forces pour continuer d'avancer.

INTRODUCTION

Les logiciels modernes jouent un rôle essentiel dans les activités quotidiennes, qu'il s'agisse de la fourniture d'expériences utilisateur personnalisées via des applications mobiles ou de la gestion d'infrastructures critiques. Avec l'évolution des techniques et technologies, la complexité de leur structure et de leur code a également augmenté pour répondre aux exigences toujours croissantes des utilisateurs et du marché. Cependant, malgré ces avancées, la plupart des systèmes finissent par rencontrer des pannes (Sommerville, 2015), ce qui met en évidence la nécessité de développer des pratiques robustes capables de garantir la performance, la fiabilité et la sécurité de ces systèmes.

Dans ce contexte, il est essentiel que les développeurs s'assurent non seulement du bon fonctionnement du code, mais aussi qu'ils soient prêts à identifier et à résoudre rapidement les problèmes lorsqu'ils surviennent. C'est ici que la pratique de la *journalisation* (ou *logging*, en anglais) devient indispensable, en fournissant une vue détaillée et en temps réel du comportement du système.

Lorsque les développeurs insèrent des instructions de log dans le code, ils créent un canal de communication entre le système et ses opérateurs, fournissant ainsi un flux constant de retour d'information. Pour les opérateurs, les logs sont souvent la principale source d'informations pour comprendre et diagnostiquer le comportement d'un système logiciel (El-Masri *et al.*, 2020), pour surveiller le comportement d'un système et enquêter sur ses défaillances (Yuan *et al.*, 2012c; Yao *et al.*, 2020), identifier les problèmes et effectuer le dépannage (Lin *et al.*, 2016).

Pour cette raison, il serait idéal de conserver des enregistrements de toutes les preuves qui peuvent être analysées en temps réel ou ultérieurement afin de capturer des informations précieuses pendant l'exécution des systèmes logiciels (Hassani *et al.*, 2018). Cependant,

lorsque le volume de logs devient important, leur gestion et leur analyse peuvent entraîner des coûts significatifs et devenir difficiles. Toutefois, la capacité à analyser un système complexe ne repose pas uniquement sur l'accès à des journaux adéquats et à des sources d'information pour le débogage, mais aussi sur une expertise appropriée pour interpréter ces données et résoudre efficacement les problèmes (Adkins *et al.*, 2020).

PROBLÈME ET MOTIVATION

Si les pratiques de journalisation sont indispensables, leur mauvaise gestion peut causer des problèmes significatifs. L'exemple de la panne mondiale de YouTube en octobre 2018, causée par un changement apparemment mineur dans une bibliothèque de journalisation, illustre ces risques (Adkins *et al.*, 2020). Cette modification, destinée à améliorer la granularité des logs, a entraîné une saturation de la mémoire des serveurs sous la charge de production, provoquant une cascade de défaillances et paralysant le service.

Cet incident souligne les défis persistants liés au choix des niveaux de sévérité appropriés pour les logs, qui peuvent générer soit trop, soit trop peu d'entrées. Un manque de journalisation peut dissimuler des informations critiques à des niveaux de sévérité inférieurs (Hassani *et al.*, 2018; Fu *et al.*, 2014). À l'inverse, un excès de journalisation peut entraîner la production d'informations inutiles :

- lorsque les instructions de log sont classées avec un niveau de sévérité supérieur à la sémantique de leurs messages (Zeng *et al.*, 2019; Li *et al.*, 2017a), ou d'informations redondantes,
- lorsque des informations répétées sont classées à différents niveaux de sévérité (Chen & Jiang, 2017a).

Ce déséquilibre dans les données de log affecte la performance du système (Chen & Jiang, 2017a; Li *et al.*, 2017a; Yuan *et al.*, 2012b), sa maintenance (Li *et al.*, 2017a; He *et al.*,

2018), ainsi que la surveillance et le diagnostic basés sur les logs (Hassani *et al.*, 2018; Li *et al.*, 2017a; Rong *et al.*, 2018).

Dans le but d'équilibrer les bénéfices et les coûts de la production des entrées de log, les développeurs considèrent l'attribution du niveau de sévérité approprié comme leur principale approche (Li *et al.*, 2020a). Ils consacrent un temps considérable à ajuster ces niveaux (Kabinna *et al.*, 2018), modifiant souvent la sévérité en réévaluant la criticité des événements (Yuan *et al.*, 2012b; Zhao *et al.*, 2017). Par exemple, ils peuvent reconsidérer si une instruction initialement classée comme *Info* ne devrait pas plutôt être de niveau *Error*, ou s'il serait plus approprié de l'ajuster à un niveau intermédiaire, comme *Warn* (Zhao *et al.*, 2017).

Parmi les défis (D) qui rendent le choix du niveau de sévérité difficile, on trouve :

D1. le manque de connaissances sur la façon dont les logs seront utilisés et de compréhension de la criticité d'un événement (Oliner *et al.*, 2012; Zeng *et al.*, 2019), qui peut entraîner des instructions de log qui ne reflètent pas adéquatement la criticité des événements enregistrés ;

D2. l'ambiguïté de certains événements qui semblent être liés à plusieurs niveaux de sévérité (Lin *et al.*, 2016; Zhao *et al.*, 2017). L'absence d'une distinction claire entre les différents niveaux de sévérité peut entraîner des enregistrements incohérents, ce qui rend difficile l'analyse et la surveillance efficace des systèmes ;

D3. un manque de spécifications et de directives pratiques pour effectuer les tâches de journalisation dans les projets et l'industrie (He *et al.*, 2018; Rong *et al.*, 2018; Anu *et al.*, 2019), ce qui peut entraîner une dépendance excessive aux expériences personnelles et aux préférences subjectives.

En l'absence de directives standardisées et de pratiques claires, les choix de niveaux de sévérité des logs peuvent dépendre excessivement de l'expérience personnelle et des préférences individuelles des développeurs (Rong *et al.*, 2018). Cela peut introduire une variabilité indésirable dans la manière dont les événements critiques sont enregistrés, rendant difficile une analyse cohérente des systèmes.

Plusieurs études proposent des solutions pour l'utilisation correcte du niveau de sévérité des logs. Kim *et al.* (2020) proposent une approche pour vérifier l'adéquation des niveaux de sévérité des logs. Li *et al.* (2017a) propose une approche d'apprentissage profond pour la prédiction du niveau de sévérité des logs en utilisant les emplacements de journalisation. Li *et al.* (2020b) discutent de l'endroit où appliquer les emplacements de journalisation et proposent une approche d'apprentissage pour fournir des suggestions de journalisation de blocs de code. D'autres études dans la littérature se concentrent sur *« où journaliser »* comme Zhao *et al.* (2017); Fu *et al.* (2014) et Li *et al.* (2020a). Des études antérieures ont également exploré les dépôts de code et les systèmes de gestion des problèmes pour examiner les pratiques de journalisation (Yuan *et al.*, 2012b; Li *et al.*, 2020a; Zhang *et al.*, 2022; Patel *et al.*, 2022).

Cependant, ces travaux proposent des approches qui n'explorent pas en profondeur les niveaux de sévérité des logs ou ne fournissent pas de descriptions détaillées. On constate également un manque de correspondance des bibliothèques de journalisation et de leurs niveaux de sévérité respectifs. Ce manque de cartographie systématique empêche les développeurs de comprendre pleinement comment les niveaux de sévérité des logs sont définis et utilisés à travers différentes bibliothèques, ce qui peut entraîner des choix incohérents ou inefficaces dans la pratique. En considérant les dépôts de code et les systèmes de gestion d'incidents (*issue trackers*), il manque également une analyse des motivations sous-jacentes des ajustements des niveaux qui pourrait contribuer à une compréhension plus approfondie des décisions prises.

HYPOTHÈSE DE RECHERCHE

Hypothèse de recherche : *Un cadre conceptuel pour les niveaux de sévérité des logs réduit l'ambiguïté et minimise la génération excessive de logs, contribuant ainsi à un processus de prise de décision plus efficace dans la gestion des systèmes logiciels.*

La proposition de ce cadre conceptuel pour les niveaux de sévérité des logs répond aux défis identifiés (D1 - D3) dans la pratique de la journalisation dans les projets logiciels, ce qui rend essentiel la création d'un cadre conceptuel offrant une nomenclature standardisée et des objectifs bien définis pour les niveaux de sévérité des logs. Ce cadre permettra non seulement de réduire l'ambiguïté et l'incohérence dans la classification des événements, mais aussi de fournir une base solide pour la prise de décision, en alignant les pratiques de journalisation avec les standards de l'ingénierie logicielle.

OBJECTIFS DE RECHERCHE

Plus précisément, notre hypothèse générale de recherche se traduit en trois objectifs de recherche (OR) spécifiques :

- **OR$_1$** : **Définir un cadre conceptuel pour les niveaux de sévérité des logs.**
    - **Méthode :** Nous avons examiné l'état de l'art (la littérature) et la pratique (les bibliothèques de journalisation) pour comprendre les objectifs et les applications des niveaux de sévérité.
    - **Contributions :**

- Une mise en correspondance des niveaux de sévérité identifiés dans la littérature, permettant de souligner les similarités sémantiques et les divergences terminologiques.
- Une mise en correspondance des niveaux de sévérité présents dans les bibliothèques de journalisation, révélant la diversité des implémentations et la nécessité de convergence vers une nomenclature standardisée.
- Un cadre conceptuel des niveaux de sévérité identifiant les niveaux essentiels, avec une standardisation de la nomenclature et une définition des finalités des niveaux de sévérité des logs.

- **OR$_2$** : **Comprendre les pratiques d'ajustement des niveaux de sévérité des logs**
  - **Méthode :** Nous adoptons une méthodologie en deux phases pour atteindre cet objectif :
    - *Phase descriptive :* examiner les dépôts de systèmes Java *open-source* afin d'identifier les ajustements de niveaux de sévérité entre leurs différentes versions.
    - *Phase explicative :* analyser les descriptions des problèmes ainsi que les commentaires sur les *commits* relatifs à ces ajustements.
  - **Contributions :**
    - Une description des tendances et des modèles dans les ajustements de sévérité.
    - Une analyse des motivations sous-jacentes aux ajustements des niveaux de sévérité, permettant d'identifier les raisons pour lesquelles les développeurs modifient ces niveaux au fil du temps.
- **OR$_3$** : **Définir des heuristiques pour le choix des niveaux de sévérité des logs.**
  - **Méthode :** Nous dérivons un ensemble d'heuristiques pour la sélection des niveaux de sévérité des logs, basé sur les résultats obtenus.

- **Contribution :**
  - Un ensemble d'heuristiques pour assister les développeurs et les opérateurs dans le choix et l'ajustement des niveaux de sévérité des logs.

ORGANISATION DE LA THÈSE

Cette thèse est structurée comme suit. Dans le **chapitre 1**, nous présentons un aperçu des principaux concepts nécessaires à la bonne compréhension de cette thèse. Nous introduisons le concept de log et de journalisation, en expliquant ce que sont les instructions de log et les entrées de log à travers des exemples concrets. Par la suite, nous abordons de manière préliminaire les niveaux de sévérité des logs, qui sont au cœur de notre recherche. Nous définissons également le concept d'ajustement des niveaux de sévérité, en distinguant les atténuations des aggravations.

Ensuite, nous retraçons l'évolution des pratiques de journalisation, depuis les simples impressions (*prints*) jusqu'aux bibliothèques de logging modernes. Les problématiques liées à la journalisation sont ensuite exposées, notamment les questions de où, quoi, comment et s'il faut journaliser. Nous poursuivons avec une présentation des systèmes de suivi des incidents, en mettant particulièrement l'accent sur Jira, un outil essentiel dans la gestion des ajustements de niveaux de sévérité. Enfin, nous introduisons l'outil SLogAnalyser, qui a été utilisé pour extraire et analyser les informations issues des systèmes étudiés dans le cadre du chapitre 4

Au **chapitre 2**, nous présentons la revue de la littérature, en nous concentrant sur les pratiques de journalisation et les approches automatisées pour le choix des niveaux de sévérité de log. Nous détaillerons l'évolution des approches des études, qui se concentraient initialement exclusivement sur les analyses de code source et de *commits*, pour ensuite inclure les rapports d'incidents et les sondages auprès des développeurs. Nous verrons que, bien que

de nombreuses études soient approfondies sur les pratiques de journalisation, elles laissent des lacunes en ce qui concerne les niveaux de sévérité, souvent traités comme un élément secondaire du sujet principal.

Nous présenterons également les approches utilisant l'apprentissage automatique pour ajuster ou choisir les niveaux de sévérité, allant des techniques supervisées aux méthodes de deep learning. Bien que la diversité des études soit significative, nous observerons que, malgré les constats sur le manque de directives claires pour la journalisation, ces travaux ne fournissent pas toujours des lignes directrices concrètes et facilement applicables pour guider les décisions des développeurs. De plus, même les approches automatisées, bien qu'efficaces, ne possèdent pas un caractère formateur ou explicatif suffisant pour éclairer le processus de décision des praticiens.

Dans le **chapitre 3**, nous présentons les résultats d'une étude préliminaire réalisée pour comprendre comment les niveaux de sévérité des logs sont perçus, dans le but d'obtenir une vue d'ensemble de l'état de l'art à travers une cartographie multivocale de trois sources principales : la littérature, les bibliothèques de logs et les perspectives des praticiens du domaine, tirées d'un site de questions et réponses. Pour chacune de ces sources, nous avons identifié les niveaux de sévérité cités, aboutissant à un ensemble de 19 nomenclatures. De plus, nous avons examiné comment chaque source définit ces niveaux de sévérité, mettant en lumière les convergences et les divergences entre eux.

À partir de ces résultats, nous proposons un ensemble de définitions pour les niveaux de sévérité, ainsi que le concept de « *finalité du niveau de sévérité* », qui pourrait servir de ligne directrice pour clarifier la distinction entre ces niveaux. Ce chapitre vise à fournir une base empirique solide pour mieux orienter le choix et l'ajustement des niveaux de sévérité dans

les systèmes logiciels, en tenant compte à la fois des contextes théoriques et des expériences pratiques.

Au **chapitre 4**, nous présentons une étude en deux phases que nous avons menée sur les ajustements des niveaux de sévérité des logs entre différentes versions de systèmes *open-source*. Ce chapitre vise à répondre à deux questions : quels ajustements des niveaux de sévérité des logs surviennent entre les différentes versions d'un système, et pourquoi ces ajustements sont-ils effectués.

Pour répondre à la première question, nous abordons dans la première phase, la phase descriptive de l'étude, l'identification et la quantification des ajustements de sévérité entre les versions stables des systèmes étudiés, à l'aide des données extraites grâce à l'outil SLogAnalyser. Cette enquête révèle que la majorité des ajustements se produisent entre les niveaux *Debug* et *Info*. Dans la phase explicative, en explorant les raisons de ces ajustements, nous observons que les niveaux de sévérité présentent une mutabilité, pouvant être ajustés en fonction de la maturité du système et des exigences du milieu de production.

Dans le **chapitre 5**, nous abordons la phase prescriptive de l'étude, avec pour objectif de répondre à la question suivante : *Comment améliorer la classification des niveaux de sévérité des logs ?* En s'appuyant sur les résultats des chapitres précédents, nous avons développé 24 heuristiques pour orienter les développeurs dans la sélection et l'ajustement des niveaux de sévérité. Ces heuristiques, fondées sur une analyse empirique, visent à structurer et formaliser la prise de décision, afin de produire des logs à la fois pertinents pour le dépannage et optimisés pour ne pas surcharger les systèmes. Le chapitre présente la méthodologie suivie, discute des résultats obtenus et examine leurs implications pratiques.

La **conclusion** conclut cette thèse en mettant en évidence les contributions de la recherche et en résumant les limites et les orientations pour les travaux futurs.

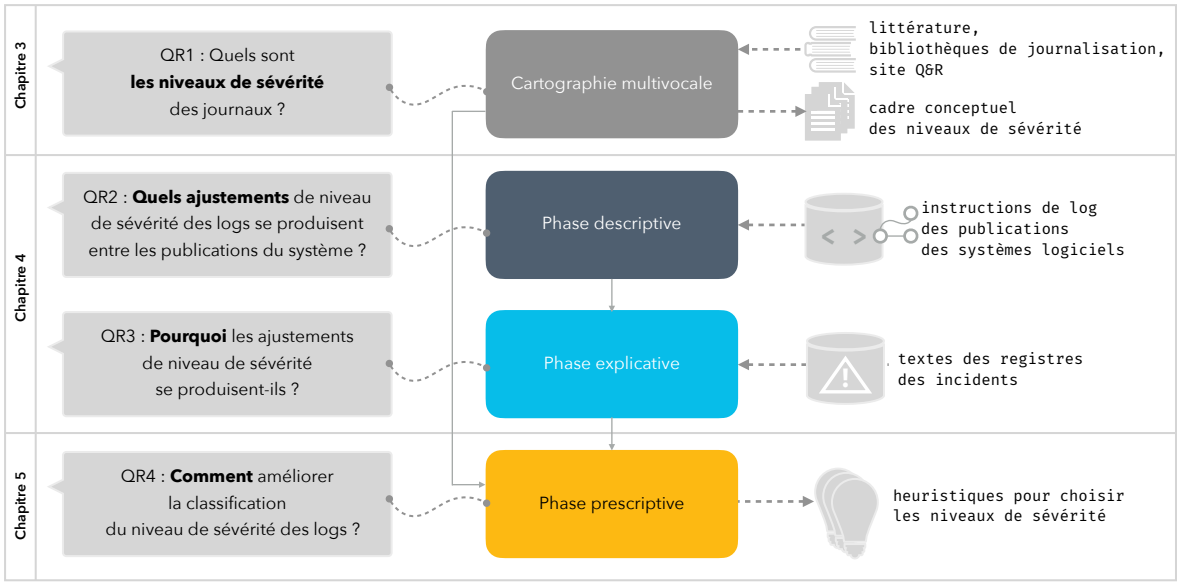

FIGURE 1 : Aperçu de la thèse.

Pour mieux visualiser la méthodologie et les contributions de cette thèse, la figure 1 présente un aperçu global des différentes phases et étapes. Elle illustre les phases méthodologiques ainsi que les contributions spécifiques, telles que la cartographie des niveaux de sévérité dans la littérature et les bibliothèques de journalisation, et la définition d'un cadre conceptuel pour les niveaux de sévérité des logs

Dans le cadre de cette thèse et pour atteindre les objectifs visés, deux articles scientifiques ont été produits, contribuant également à la construction d'une thèse de master. Vous trouverez ci-après les références complètes :

1. **Article 1 :** Mendes, E. & Petrillo, F. (2021). Log severity levels matter : A multivocal mapping. *Dans 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pp. 1002–1013. IEEE.

2. **Article 2 :** Mendes, E., Vasconcelos, M. M., Petrillo, F., Hallé, S. *From description to prescription : unraveling log severity adjustments in open-source software*. Soumis à *Journal of Systems and Software (JSS)* en février 2024, sous révision.

CHAPITRE I

NOTIONS DE BASE

Ce chapitre présente les concepts de base qui sous-tendent cette thèse de recherche. Une description des logs et de leur rôle dans les systèmes logiciels est d'abord proposée à la section 1.1. Ensuite, la section 1.2 introduit les niveaux de sévérité des logs et discute des ajustements de ces niveaux. La section 1.3 présente l'évolution des bibliothèques de journalisation. Ensuite, la section 1.4 aborde les défis liés à la journalisation en répondant aux questions clés telles que où, quoi, et comment journaliser, ainsi que la pertinence de la journalisation. La section 1.5 offre un aperçu des systèmes de suivi des incidents, en se concentrant sur Jira [1] tandis que la section 1.6 présente le SLogAnalyzer (Vasconcellos, 2023), un outil pour faciliter l'analyse et la comparaison des niveaux de sévérité entre les différentes versions des logiciels. Enfin, la section 1.7 présente les remarques finales du chapitre [2].

1.1 LOGS

Bien que souvent sous-estimés, les logs sont une riche source d'informations pour les développeurs et les opérateurs de systèmes logiciels (Gholamian & Ward, 2021). Ils contribuent à la gestion des ressources des systèmes, à la gestion des utilisateurs et des applications, à la sécurité, à la surveillance du comportement du système et au suivi du processus de développement logiciel (Schmidt *et al.*, 2012). Les logs logiciels sont aussi largement utilisés dans les tâches d'assurance de la fiabilité, étant souvent la seule source de données qui enregistre des informations d'exécution (He *et al.*, 2021). De plus, ils jouent un

1. https ://www.atlassian.com/software/jira

2. Certaines figures de ce chapitre ont été adaptées à partir de Cândido *et al.* (2021) (licence CC BY 4.0) et El-Masri *et al.* (2020) (utilisées avec autorisation d'Elsevier).

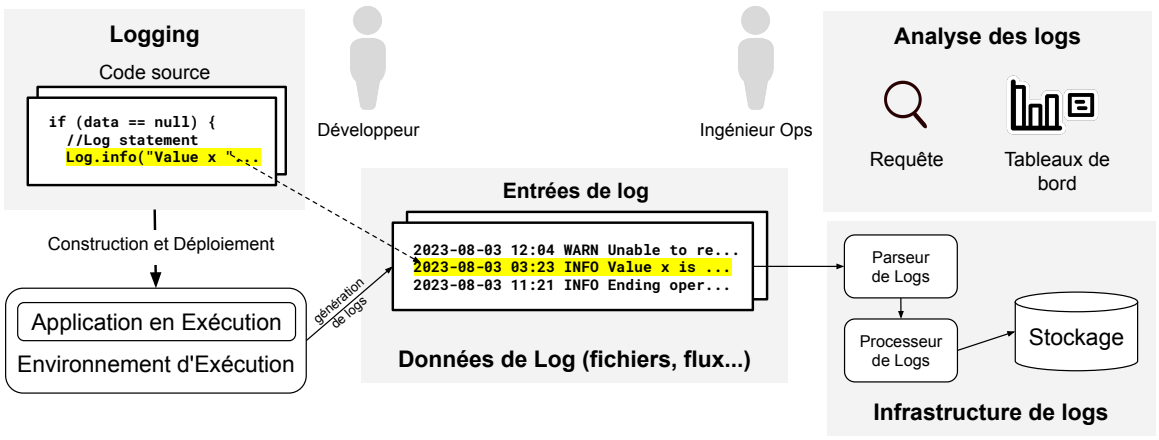

FIGURE 1.1 : Aperçu du cycle de vie des données de log.
Adapté de Cândido *et al.* (2021). Cette œuvre est sous licence CC BY 4.0.

rôle essentiel dans la prise de décisions basées sur les données dans l'industrie (Pecchia *et al.*, 2015).

Le terme « logs » désigne une collection de messages qui décrivent collectivement le contexte d'une occurrence spécifique, telles que des messages de connexion et de déconnexion d'utilisateur, des alertes sur l'imminence de pannes, ou des états critiques pour le développement (Schmidt *et al.*, 2012). Comme présenté dans la figure 1.1, le processus de génération des logs est amorcé dès la phase de développement, lorsque les développeurs déterminent les points où les *instructions de log* seront insérées dans le code source (Li *et al.*, 2018). Une *entrée de log* est ensuite générée à chaque exécution de ces instructions, qui sont ensuite enregistrées dans les *fichiers de log*, également connus sous le nom de *logs d'exécution* ou *logs d'événements* (Cinque *et al.*, 2012), permettant aux développeurs et aux opérateurs d'analyser ces données pour diverses tâches (Gholamian & Ward, 2021), telles que la surveillance du système, la détection d'anomalies ou la compréhension du comportement du système (Cândido *et al.*, 2021).

Pour bien comprendre ces processus, il est important de se familiariser avec quelques termes clés relatifs aux logs :

1. ***Journalisation* (*Logging*) :** Le terme *journalisation* regroupe l'ensemble des pratiques liées à la création et à la maintenance des logs (Gholamian & Ward, 2021), telles que la création et la mise à jour des instructions de log.
2. **Instruction de log (« *Log statement* ») :** Nous utilisons le terme *instruction de log* pour identifier les commandes de code qui génèrent les données de log.

   Les systèmes peuvent avoir leur propre format d'instruction de log propriétaire, ce qui complique la compréhension pour ceux qui ne sont pas familiers avec le domaine spécifique du système. Certains systèmes ne disposent même pas d'une définition claire du format des instructions de log ou ne respectent pas rigoureusement celle-ci (Salfner *et al.*, 2004). Dans sa forme la plus simple, les instructions de journalisation sont des instructions d'affichage (*e.g.,* « `printf()` » en C, « `System.out.print()` » en Java) utilisées dans différents langages de programmation (Gholamian & Ward, 2021).

   Typiquement, les développeurs utilisent des *bibliothèques de journalisation* pour ajouter des instructions de log. Dans l'extrait de code liste 1.1, les lignes présentent des instructions de log réelles provenant du dépôt de code source de Hadoop [3] en utilisant est SLF4J [4]. Dans ces instructions, `LOG` est l'objet de journalisation fourni par une bibliothèque de journalisation, `LOG.info` represente le *niveau de sévérité Info*, et les arguments passés à la méthode forment le *message de log*, composé d'une partie textuelle, *e.g.,* « `maxTaskFailuresPerNode is` », concaténée à une variable (`maxTaskFailuresPerNode`).

3. https ://github.com/apache/hadoop/
4. http ://www.slf4j.org/

```
protected void serviceInit(Configuration conf) throws Exception {
  ...
  LOG.info("nodeBlacklistingEnabled:" + nodeBlacklistingEnabled);
  ...
  LOG.info("maxTaskFailuresPerNode is " + maxTaskFailuresPerNode);
  ...
  LOG.info("blacklistDisablePercent is " + blacklistDisablePercent);
}
```

Liste 1.1 : Un exemple d'instruction de log du dépôt Apache Hadoop.

```
2015-10-18 18:01:53 INFO [main] org.apache.hadoop...RMContainerRequestor: nodeBlacklistingEnabled:true
2015-10-18 18:01:53 INFO [main] org.apache.hadoop...RMContainerRequestor: maxTaskFailuresPerNode is 3
2015-10-18 18:01:53 INFO [main] org.apache.hadoop...RMContainerRequestor: blacklistDisablePercent is 33
```

Liste 1.2 : Trois exemples d'entrées de log de Hadoop. Adapté de (He *et al.*, 2020).

3. **Entrée de log :** Lorsque les instructions de log sont exécutées, elles génèrent des *entrées de log*, comme l'exemple dans la liste 1.2.
4. **Données de log :** Nous utilisons *données de log* pour faire référence à une collection d'entrées de log générées par un système logiciel, que ce soit dans des fichiers de log, des flux de données ou d'autres types de stockage.

La configuration de la bibliothèque de journalisation détermine les champs d'un entrée de log générée, qui comporte deux parties : *en-tête de log* et *message de log*, comme montré

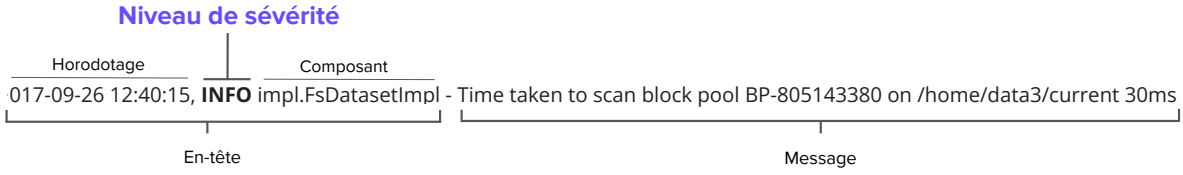

FIGURE 1.2 : Les champs d'une entrée de log.
Adapté de El-Masri *et al.* (2020). Reproduit avec autorisation d'Elsevier.

dans la figure 1.2. Dans l'extrait liste 1.2, troisième ligne, l'entrée de log présente les champs comme suit :

- En-tête de log :
  - *Horodatage :* `2015-10-18 18:01:53`;
  - *Niveau de sévérité :* `INFO`;
  - *Thread :* `[main]`;
  - *Composant :* `org.apache.hadoop...RMContainerRequestor`;
- Message de log :
  - *Champ statique :* `nodeBlacklistingEnabled:`
  - *Champ dynamique :* `true`.

En bref, les développeurs ajoutent des *instructions de log* à des points d'intérêt dans le code source d'un système logiciel pour produire les *données de log* lors de son exécution. Les *entrées de log* sont le résultat de ce processus et constituent les *données de log*. Une fois générées, les données de log peuvent être consommées via des fichiers de log, des bases de données ou des systèmes de traitement de flux de données.

1.1.1 TRACES ET LOGS

Le terme « log » se réfère généralement à l'enregistrement continu de l'exécution d'un système logiciel, documentant des événements tels que les accès des utilisateurs, les opérations effectuées et les défaillances détectées (Roudjane, 2023). Contrairement au « *tracing* », qui capture la séquence temporelle détaillée d'événements lors d'une exécution spécifique du programme via des outils externes (Miranskyy *et al.*, 2016), les logs sont générés directement par les instructions de log insérées dans le code du logiciel.

Alors que le *tracing* fournit des données structurées et détaillées, telles que les chemins d'exécution et les valeurs de variables en temps réel, les logs d'exécution consolident des informations au fil du temps, offrant une vue séquentielle des opérations du système (Roudjane *et al.*, 2021). Les logs d'exécution, par exemple, peuvent inclure des enregistrements de connexions et de déconnexions d'utilisateurs, des alertes d'erreurs ou des avertissements sur l'état du système. Selon Laplante *et al.* (2017), une trace d'exécution est une séquence ordonnée d'événements, potentiellement infinie, mais ce qui distingue les logs d'exécution d'autres sources de données est leur nature continue et séquentielle, documentant les actions et les états du système à des moments successifs dans le temps.

1.2 NIVEAUX DE SÉVÉRITÉ DE LOG

Les logs d'un système logiciel se composent généralement d'un grand nombre d'entrées enregistrées séquentiellement au fil du temps par différents modules et composants. Ces entrées de log documentent divers événements, tels que des actions utilisateur, des états système, des erreurs ou des alertes critiques. *Par états système*, on entend des informations représentant les valeurs courantes de variables ou de paramètres internes du système, comme les compteurs, les indicateurs d'état, les tailles de files d'attente ou l'utilisation des ressources (Sage, 2019). Ces valeurs permettent de suivre le comportement du système au fil du temps et sont souvent enregistrées pour le débogage ou le diagnostic. Toutefois, en raison de la diversité des informations enregistrées, il devient nécessaire de catégoriser ces entrées afin d'en faciliter l'interprétation.

Pour cela, on utilise **les niveaux de sévérité des logs**, qui permettent de prioriser les événements selon leur degré d'urgence. Ce mécanisme permet aux développeurs et aux utilisateurs de spécifier la quantité appropriée de journaux à imprimer pendant l'exécution du logiciel (Li *et al.*, 2017a). En activant l'impression des journaux pour les événements critiques

(comme les erreurs) et en supprimant les journaux moins critiques (comme les événements de suivi) (Gülcü, 2003), les niveaux de sévérité aident à gérer efficacement la surveillance du système.

Ainsi, un niveau moins sévère signifie que le système fonctionne comme prévu, tandis qu'un niveau plus sévère signale un problème nécessitant une intervention (Chen & Jiang, 2017a). Ce classement est essentiel pour garantir que les informations importantes sont traitées avec la priorité appropriée.

Dans le cadre de notre recherche, nous avons identifié 19 nomenclatures différentes pour les niveaux de sévérité des logs en analysant 40 bibliothèques de journalisation. Ce constat révèle une diversité considérable dans la classification des événements à travers ces bibliothèques, ce qui peut rendre difficile l'harmonisation des pratiques de journalisation, notamment en ce qui concerne l'évaluation de la criticité des événements dans des environnements variés. Par exemple, Apache Log4j supporte six niveaux de log : *Trace*, *Debug*, *Info*, *Warn*, *Error*, et *Fatal*. En revanche, Log4Net supporte 15 niveaux. Ce point sera abordé plus en détail au chapitre 3.

Indépendamment de la bibliothèque de log utilisée, ces niveaux de sévérité des logs doivent former un ordre total au sens mathématique, où chaque niveau est strictement plus sévère que le précédent — autrement dit, il ne doit pas y avoir deux niveaux avec la même sévérité.

Cependant, un même niveau de sévérité peut être interprété ou utilisé différemment selon la bibliothèque de journalisation. Par exemple, un message marqué comme *Warn* dans une bibliothèque peut correspondre à un niveau *Error* dans une autre, en raison des conventions propres à chaque outil.

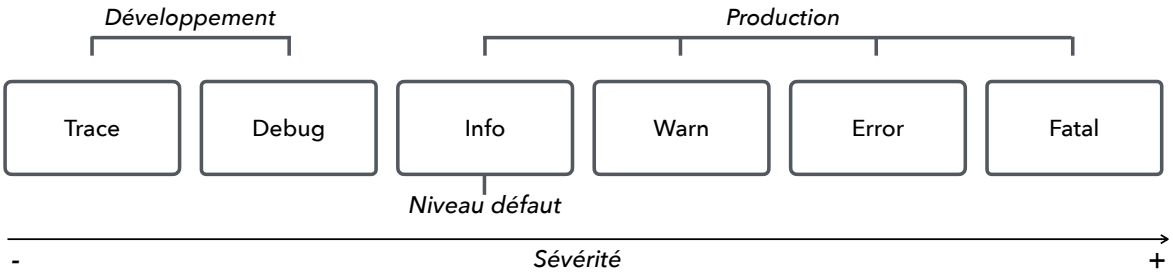

FIGURE 1.3 : Ordre total de niveaux de sévérité de log et environnements cibles.
© Eduardo Mendes de Oliveira, 2024.

Par ailleurs, les niveaux de sévérité des logs ciblent principalement deux environnements distincts : le développement et la production (Liu *et al.*, 2019). Alors que les tâches de développement bénéficient d'entrées de log générées à tous les niveaux, l'environnement de production n'a besoin que d'un sous-ensemble de ces entrées. Notamment, le niveau *Info* marque le seuil minimal des niveaux de sévérité généralement activés en production (figure 1.3). Certaines études désignent également ce seuil comme le « **niveau par défaut** ». Cette terminologie permet de distinguer les événements devant être journalisés en production, de ceux destinés à un usage exclusivement dans les environnements de développement, tels que les niveaux *Debug* ou *Trace*. Dans ce travail, nous adoptons la nomenclature *niveaux de développement* pour les niveaux en dessous du niveau par défaut et *niveaux de production* pour les autres.

Dans le chapitre suivant, nous présenterons et discuterons en détail ces niveaux de sévérité et la façon dont ils se comparent dans les différentes bibliothèques étudiées.

```
// Version Hbase/2.0.0RC0, TableAuthManager.java (ligne 188)
LOG.debug("Skipping permission cache refresh because writable data is empty")
```

```
// Version Hbase/2.5.0RC0, TableAuthManager.java (ligne 136)
LOG.info("Skipping permission cache refresh because writable data is empty")
```

Liste 1.3 : Exemple d'aggravation du niveau de sévérité dans HBase : le niveau passe de *Debug* à *Info* entre les versions 2.0.0RC0 et 2.5.0R0.

1.2.1 AJUSTEMENTS DES NIVEAUX DE SÉVÉRITÉ DES LOGS

Dans ce travail, nous faisons référence à des *ajustements du niveau de sévérité des logs* comme une modification qui se produit dans une instruction de log entre deux versions du code source d'un système analysé.

Cet ajustement peut être qualifié en fonction de la « direction » du changement de niveau. Si le changement se produit d'un niveau moins grave à un niveau plus grave, nous appellerons cet ajustement une *aggravation*, *e.g., Info* vers *Fatal* ; si le changement diminue le niveau d'un niveau plus grave à un niveau moins grave, nous appellerons cet ajustement une *atténuation*, *e.g., Warn* vers *Debug*. Par exemple, la liste 1.3 montre la même instruction de log dans deux versions différentes de HBase : la première de la version Hbase/2.0.0RC0, et la seconde de la version Hbase/2.5.0R0. Dans ce cas, il y a eu un ajustement du niveau de sévérité, une aggravation, car *Info* est un niveau de sévérité plus élevé que *Debug*.

Pour un exemple d'atténuation du niveau de sévérité (liste 1.4) où une instruction de log, initialement de niveau *Warn*, a été modifiée en un niveau *Debug* dans une version ultérieure.

En plus des catégories d'atténuation et d'aggravation, nous faisons référence aux *« équivalences »* comme des ajustements de sévérité caractérisés par des niveaux portant des noms différents mais ayant des sévérités compatibles, *e.g., Debug* vers *Fine* ; ces épisodes se pro-

```
// Version Hadoop/release-0.15.3, PedingReplicationBlocks (ligne 186)
LOG.warn("PendingReplicationMonitor thread received exception. " + ie)
```

```
// Version Hadoop/release-0.16.0, PedingReplicationBlocks (ligne 187)
LOG.debug("PendingReplicationMonitor thread received exception. " + ie)
```

Liste 1.4 : Exemple d'atténuation du niveau de sévérité dans Hadoop : le niveau passe de *Warn* à *Debug* entre les versions 0.15.3 et 0.16.0.

duisent, par exemple, lorsque la bibliothèque de journalisation change d'une version à une autre.

1.3 BIBLIOTHÈQUES DE JOURNALISATION

Comme mentionné précédemment, les premières pratiques de journalisation reposaient sur l'utilisation de fonctions d'affichage natives des langages de programmation, telles que des commandes pour afficher des informations sur la sortie standard. À mesure que les techniques de développement logiciel évoluaient, des protocoles comme Syslog (Leffler *et al.*, 1984) ont introduit des mécanismes plus sophistiqués pour générer des logs, notamment en utilisant des niveaux de sévérité pour catégoriser les événements enregistrés. Avec l'augmentation de la complexité des systèmes et le besoin croissant de frameworks plus performants et flexibles, des bibliothèques dédiées à la journalisation ont vu le jour.

Kabinna *et al.* (2016) décrivent cette évolution en quatre grandes phases : *Journalisation Ad Hoc, Bibliothèques de base, Bibliothèques d'abstraction de logs et Bibliothèques d'unification de logs*, chacune marquant un progrès significatif dans la gestion des logs.

La première phase, celle de la *Journalisation Ad Hoc*, précédait la création des bibliothèques de journalisation. Les développeurs utilisaient alors des fonctions d'affichage simples, telles que « `System.out.println` » en Java ou *syslog* sous Linux, pour capturer les événe-

ments du système. Cela posait plusieurs problèmes, notamment l'absence de contrôle sur la verbosité et les niveaux de sévérité des messages, ainsi que des formats de logs incohérents. Un autre inconvénient majeur était l'impact sur les performances, notamment dans des langages comme Java, où la concaténation de chaînes ralentissait considérablement les systèmes (Lupu, 2010). De plus, les solutions comme *syslog*, bien que permettant de classer les événements selon différents niveaux de sévérité, restaient limitées à certains environnements Unix et manquaient de flexibilité pour s'adapter à des environnements variés (Lonvick, 2001).

La deuxième phase a vu l'émergence des *Bibliothèques de base*, qui ont apporté une approche plus organisée à la journalisation. Ces bibliothèques, comme Log4j [5], ont introduit des fonctionnalités permettant de configurer facilement les niveaux de sévérité et de mieux contrôler la gestion des logs dans les systèmes Java.

Ensuite, avec les *Bibliothèques d'abstraction de logs*, comme Apache Commons Logging [6] (auparavant connu sous le nom de Jakarta Commons Logging ou JCL) en 2003, les développeurs ont pu écrire des instructions de log dans un format standardisé tout en utilisant différentes bibliothèques en arrière-plan, telles que Log4j ou Java Util Logging (JUL) [7]. SLF4J [8] a ensuite amélioré JCL en réduisant le surcoût des performances et en utilisant une journalisation paramétrée. Cependant, ces bibliothèques nécessitaient toujours l'intégration d'autres bibliothèques pour générer les logs, rendant la configuration plus complexe.

Enfin, la quatrième phase, celle des *Bibliothèques d'unification de logs*, a combiné les avantages des bibliothèques de base et d'abstraction. Des bibliothèques comme Logback

5. https://logging.apache.org/log4j/1.x/
6. https://commons.apache.org/proper/commons-logging/guide.html
7. https://docs.oracle.com/javase/7/docs/api/java/util/logging/package-summary.html
8. https://www.slf4j.org/

(2011) [9] et Log4j 2 (2014) [10] sont devenues des solutions complètes, offrant à la fois des fonctionnalités de base et d'abstraction, tout en améliorant les performances et la flexibilité de la journalisation.

1.4 PROBLÉMATIQUES ET ENJEUX DE LA JOURNALISATION

La journalisation dans le développement de logiciels pose plusieurs défis importants en raison de l'absence de normes largement établies et de directives bien définies. Les développeurs s'appuient souvent sur leur expérience personnelle pour déterminer comment et où implémenter les logs, ce qui peut mener à des pratiques variées et parfois inefficaces. Selon Gholamian & Ward (2021), ces défis peuvent être regroupés sous quatre questions majeures : *où journaliser*, *quoi journaliser*, *comment journaliser*, et *faut-il journaliser ?* Nous allons examiner chacun de ces aspects plus en détail.

1.4.1 OÙ JOURNALISER ?

La localisation des instructions de log dans le code est cruciale pour assurer une surveillance efficace. Identifier les points stratégiques où insérer les logs, comme les blocs de gestion d'erreurs ou les retours de fonctions, peut grandement influencer la qualité des logs. Cependant, un excès de logs peut entraîner des surcharges de performances et compliquer la maintenance. Le défi consiste donc à trouver un équilibre.

Dans ce scenario, Yuan *et al.* (2012a) analysent le code source pour identifier les blocs de code sans journalisation, afin d'y insérer des instructions de log. De plus, Zhao *et al.* (2017) présentent un outil qui détermine automatiquement l'emplacement des instructions de log en

9. https://logback.qos.ch/
10. https://logging.apache.org/log4j/2.x/

se basant sur la théorie de l'information pour maximiser leur utilité. Enfin, Fu *et al.* (2014) ont étudié où les développeurs de Microsoft placent leurs logs dans deux systèmes logiciels industriels.

1.4.2 QUOI JOURNALISER ?

Décider du contenu d'un log est tout aussi important que son emplacement. Les messages de log doivent fournir suffisamment de contexte pour aider à diagnostiquer des problèmes, sans toutefois noyer l'utilisateur sous une avalanche d'informations inutiles. Ainsi, il est essentiel de définir avec soin ce qui doit être journalisé, tout en évitant les redondances et les informations superflues.

Plusieurs travaux ont étudié cette problématique, en analysant les types d'événements enregistrés (Fu *et al.* (2014)), la nature des messages (He *et al.* (2018)). Des travaux comme ceux de Li *et al.* (2017a) ont proposé des modèles prédictifs pour suggérer automatiquement le bon niveau de sévérité des nouvelles instructions de log. En parallèle, les études de Yuan *et al.* (2012c) et Liu *et al.* (2019) présentent des approches pour ajouter automatiquement des variables aux instructions de log afin de clarifier les chemins d'exécution. Ces travaux montrent une grande diversité de contenus possibles : suivi d'étapes d'exécution, capture d'exceptions, valeurs de variables critiques, et d'autres informations liées à l'état ou au comportement du système.

Dans cette thèse, nous avons choisi de ne pas proposer une typologie normative du *quoi journaliser*, car notre objectif est de nous concentrer sur *la cohérence des niveaux de sévérité associés aux logs*. Le contenu exact du message est donc volontairement laissé en dehors du champ de cette recherche, bien qu'il soit évidemment étroitement lié à la classification des événements selon leur gravité.

1.4.3 COMMENT JOURNALISER ?

L'intégration des instructions de log dans le système logiciel pose la question de leur structuration et leur gestion. Certains préconisent une approche modulaire, où le code de log serait géré comme un sous-système indépendant. Néanmoins, dans la pratique, le code de log est souvent mêlé au code fonctionnel, rendant sa gestion plus complexe. Ce défi est d'autant plus important à long terme, lorsque le logiciel évolue et que le code de log doit être maintenu. Le travail de Chen & Jiang (2020) souligne que les bibliothèques et utilitaires de journalisation contribuent également à l'organisation des logs et à l'amélioration de leur format et qualité. L'amélioration de ces bibliothèques peut donc jouer un rôle important pour garantir des pratiques de journalisation plus standardisées et efficaces, facilitant ainsi la gestion des logs à mesure que les projets se développent.

1.4.4 FAUT-IL JOURNALISER ?

Enfin, il est parfois nécessaire de se demander s'il est pertinent de journaliser un événement donné. Les besoins en journalisation varient selon l'état du système : en période de fonctionnement normal, il est recommandé de réduire la verbosité pour minimiser les coûts en ressources, tandis qu'en cas de suspicion d'anomalie, il peut être nécessaire d'augmenter le niveau de détail des logs pour faciliter l'investigation.

Le travail de Ding *et al.* (2015) souligne l'importance de limiter la surcharge causée par la journalisation, en particulier dans les systèmes de grande envergure, sans compromettre l'efficacité des logs. Log2, un mécanisme de journalisation basé sur les coûts, prend la décision de journaliser ou non via un mécanisme de filtrage en deux phases, garantissant que les logs moins pertinents sont éliminés, tandis que les plus utiles sont conservés et affichés dans le cadre d'un budget prédéterminé.

1.5 SYSTÈMES DE SUIVI DES INCIDENTS

Dans un environnement de développement logiciel, les équipes sont amenées à discuter régulièrement des aspects positifs et négatifs des outils utilisés, des pratiques techniques adoptées, ainsi que des solutions logicielles et bibliothèques réutilisables découvertes (Sommerville, 2020). Ces échanges permettent d'identifier les problèmes rencontrés et de proposer des solutions pour les résoudre dans les futurs cycles de développement. Par exemple, une équipe pourrait envisager d'ajuster un niveau de sévérité pour améliorer la journalisation des événements dans un projet. Si cette pratique s'avère réussie, il est important de partager ces bonnes pratiques, afin que chacun puisse en bénéficier et les intégrer dans son propre travail.

C'est dans ce contexte que des outils de gestion de projet et de suivi des bugs, tels que Jira, entre autres, prennent toute leur importance. Ces outils simplifient non seulement le processus de communication entre les développeurs, mais garantissent également la scalabilité des projets en offrant une vue centralisée des problèmes et des objectifs. En améliorant l'accès à l'information, ces outils permettent d'accélérer l'intégration des nouveaux membres et facilitent une collaboration efficace (Ortu *et al.*, 2015).

Dans le cadre de cette thèse, nous nous concentrons plus particulièrement sur Jira, car cet outil a été utilisé comme source d'information pour l'analyse des décisions d'ajustement des niveaux de sévérité des logs, comme nous le verrons dans les chapitres suivants.

Selon Atlassian (2024), l'éditeur de Jira, cet outil propose des fonctionnalités de suivi des bogues, de gestion des incidents et de gestion de projet. Il intègre également des outils permettant la migration depuis des solutions concurrentes. Un *système de suivi des incidents* (en anglais, Issue Tracking System - ITS) sert de référentiel utilisé par les développeurs comme support pour le processus de développement logiciel. Il facilite la maintenance corrective,

notamment par le suivi des bogues, mais permet également de gérer d'autres demandes de maintenance, telles que des améliorations ou des modifications de fonctionnalité.

Pour illustrer l'utilisation concrète de Jira dans le suivi des incidents, nous allons examiner une capture d'écran issue du projet Hadoop (Figure 1.4), avec l'exemple de l'incident HADOOP-8075. Dans cette capture, plusieurs éléments essentiels seront mis en avant pour expliquer comment les informations sur un incident sont organisées. Ces éléments, mis en évidence à l'écran, fournissent les principales informations sur chaque incident. Ce sont ces données que nous extrayons pour examiner les motivations des ajustements des niveaux de sévérité des logs, comme nous le verrons au chapitre 3 :

- A : Le nom du projet auquel l'incident est associé.
- B : L'identifiant unique de l'incident (ID).
- C : Le titre descriptif de l'incident.
- D : La description fournie par le reporteur de l'incident.
- E : La discussion entre les développeurs, via une liste de commentaires.
- F : Le statut actuel de l'incident, reflétant son état d'avancement.

Lorsqu'un développeur rencontre un problème ou une anomalie, il peut créer un incident en renseignant un titre et, si possible, une description détaillée. L'identifiant (ID) de l'incident est automatiquement généré par JIRA selon les configurations du projet. Une fois l'incident enregistré, d'autres développeurs ayant les autorisations nécessaires peuvent contribuer à sa résolution en ajoutant des commentaires et en participant à la discussion.

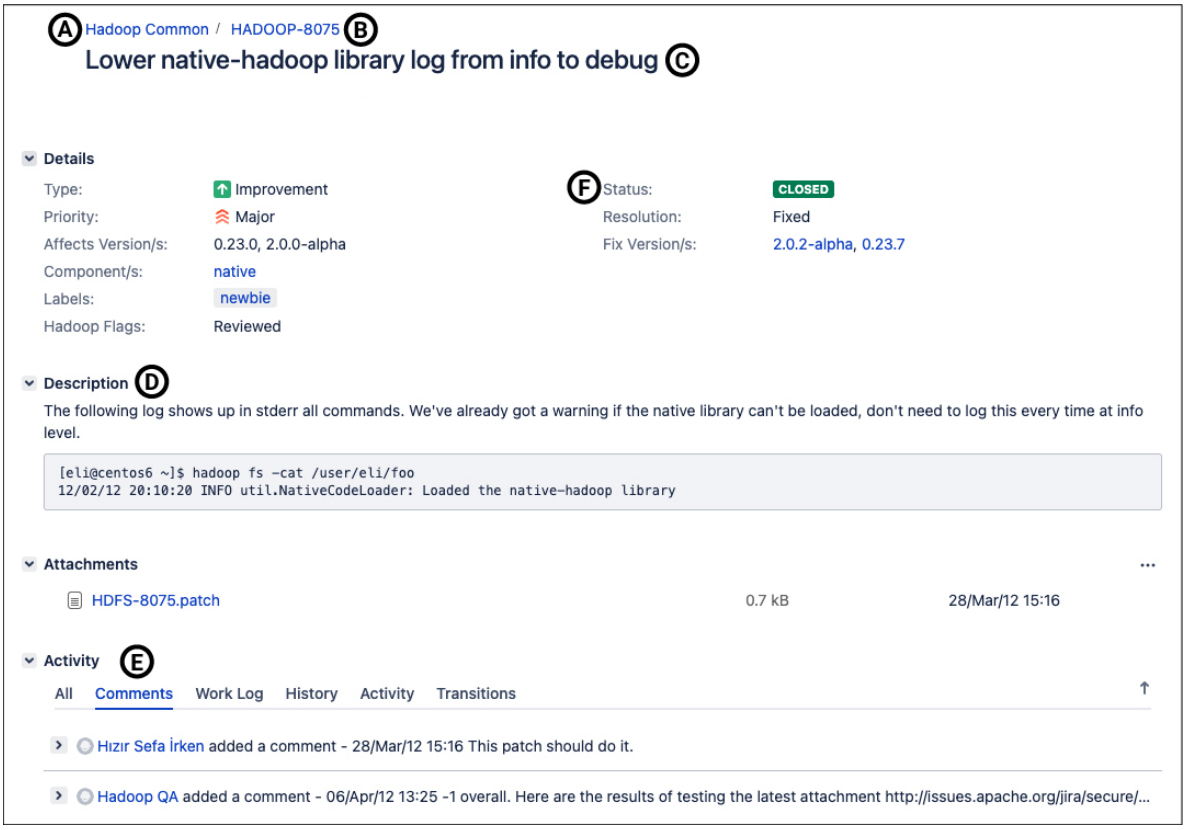

FIGURE 1.4 : Écran Jira pour l'incident HADOOP-8075.
© Eduardo Mendes de Oliveira, 2024.

1.6 SLOGANALYSER

Pour analyser les publications de systèmes logiciels dans le chapitre 4, nous avons utilisé le SLogAnalyser (Vasconcellos, 2023), un outil développé dans un projet auquel l'auteur de cette thèse a participé, puis poursuivi par M. Vasconcellos dans le cadre de son mémoire. Cet outil a joué un rôle central dans l'extraction et la structuration des données empiriques analysées.

Conçu pour faciliter l'analyse et la comparaison des niveaux de sévérité entre les différentes versions des logiciels étudiés, SLogAnalyzer permet l'extraction et la comparaison des instructions de log ainsi que de leurs caractéristiques sémantiques et syntaxiques dans les versions de systèmes Java *open-source* hébergés sur GitHub [11].

L'outil effectue le clonage des dépôts, copiant toutes les branches des projets à analyser. Une fois les dépôts clonés, le processus d'extraction des instructions de log est lancé par SLogAnalyzer. Pour chaque instruction extraite, il identifie un ensemble de 21 informations, incluant notamment l'emplacement (fichier, ligne, extrait de code), le niveau de sévérité, et le message. De plus, il explore le code source pour extraire des informations sur les méthodes de la version, telles que l'emplacement, le nombre de lignes de log, la complexité cyclomatique et le SLOC (Source Lines of Code).

Le pipeline de comparaison des versions de SLogAnalyzer est chargé de comparer les fichiers de différentes versions d'un même projet. Il classe les versions chronologiquement et compare, pour chaque paire séquentielle de versions, les fichiers communs aux deux versions. Pour les fichiers présents dans les deux versions, le pipeline utilise la bibliothèque Difflib [12] afin de comparer les contenus et identifier les modifications. La similarité entre les

11. https://github.com/

12. https ://docs.python.org/3/library/difflib.html

instructions de log est mesurée par la similarité cosinus (Singhal *et al.*, 2001; Tan *et al.*, 2005), une mesure classique utilisée pour comparer des chaînes de texte. Elle évalue à quel point deux instructions de log sont proches en termes de contenu, en calculant l'angle entre leurs représentations vectorielles. Les changements présentant une similarité cosinus supérieure à 0,5 sont considérés comme des ajustements. Ce seuil a été choisi pour s'assurer que seules les modifications substantielles, représentant plus de 50% du contenu des lignes de code, soient classées comme similaires, ce qui permet d'éviter de fausses correspondances entre instructions de log ayant des modifications mineures.

L'objectif de ce pipeline est de comparer les données extraites des lignes de code entre les versions consécutives d'un projet, en catégorisant les changements en modifications, ajouts et suppressions. À la fin du processus, toutes les informations sont stockées dans la base de données pour une analyse ultérieure.

Il est important de souligner que SLogAnalyzer n'est pas présenté ici comme une contribution scientifique centrale, mais bien comme un outil de soutien méthodologique permettant l'extraction structurée des données nécessaires à l'analyse menée. La documentation complète de l'outil – incluant sa structure logicielle, les cas d'utilisation et les scripts de traitement – est disponible dans le mémoire original (Vasconcellos, 2023), ainsi que sur son dépôt GitHub associé.

1.7 REMARQUES FINALES

Ce chapitre a posé les bases nécessaires à la compréhension des concepts centraux qui sous-tendent cette thèse. Nous avons exploré le rôle des logs dans les systèmes logiciels, en soulignant leur importance pour le débogage, le suivi et la maintenance des applications. Nous avons présenté une introduction aux niveaux de sévérité des logs afin de comprendre leur

utilité dans la catégorisation et la priorisation des événements dans les systèmes. De plus, l'évolution des bibliothèques de journalisation a mis en évidence les avancées réalisées pour gérer efficacement les journaux d'exécution.

Les défis liés à la génération des logs, que ce soit en termes de localisation, de contenu ou de pertinence des enregistrements, illustrent les difficultés rencontrées par les développeurs dans la gestion de cette pratique essentielle. Le rôle des systèmes de suivi des incidents, comme Jira, a également été mis en avant pour montrer comment ces outils capturent les décisions des développeurs, y compris celles liées aux ajustements des niveaux de sévérité des logs.

Enfin, la présentation du SLogAnalyzer, un outil conçu pour faciliter l'analyse des modifications des instructions de log entre différentes versions de logiciels, offre un support précieux pour mieux comprendre et gérer les ajustements des niveaux de log.

Dans le prochain chapitre, nous présenterons la revue de la littérature sur les niveaux de sévérité des logs.

CHAPITRE II
REVUE DE LA LITTÉRATURE

Les systèmes logiciels génèrent d'énormes volumes de logs, qui sont essentiels pour le débogage, la surveillance, et la maintenance des systèmes. Cependant, l'efficacité de la journalisation dépend fortement de la manière dont les messages de logs sont générés, interprétés et ajustés. Comme le soulignent Oliner *et al.* (2012), les développeurs n'ont pas toujours une vision complète de la manière dont leur code sera utilisé, ce qui peut entraîner des incohérences dans la journalisation et des difficultés dans l'interprétation des messages de logs. Par exemple, l'utilisation incorrecte des niveaux de sévérité peut mener à des journaux trompeurs, comme le montre l'exemple de BlueGene/L (Liste 2.5), où une situation normale est décrite comme une « FAILURE », illustrant un faux positif.

Ce genre de faux positifs illustre la difficulté qu'ont les développeurs à choisir correctement les niveaux de sévérité des logs, un problème récurrent dans le génie logiciel. Alors que l'étude d'Oliner *et al.* (2012) met en évidence ces défis à un niveau général, d'autres recherches ont exploré plus en détail les pratiques de journalisation dans divers projets logiciels, cherchant à mieux comprendre et à améliorer les processus de gestion de logs.

Dans ce chapitre, nous passons en revue les études qui ont exploré les pratiques de journalisation logicielle et proposé des solutions automatisées pour optimiser la gestion des niveaux de sévérité des logs. Ces travaux se concentrent sur les habitudes des développeurs en matière de logs et sur les défis associés à la gestion des niveaux en différents scenarios.

```
YY-MM-DD-HH:MM NULL RAS BGLMASTER FAILURE ciodb exited normally with exit code 0
```

Liste 2.5 : Entrée de log du BlueGene/L. Adapté de (Oliner *et al.*, 2012).

Ce chapitre est organisé comme suit : dans la section 2.1, nous abordons les travaux liées aux pratiques de journalisation. Ensuite, les travaux qui visent à automatiser le choix des niveaux de sévérité de log sont abordés dans la section 2.2. Enfin, la section 2.3 conclut le chapitre avec des remarques finales.

2.1 PRATIQUES DE JOURNALISATION

Dans cette section, nous examinons les recherches sur les pratiques de journalisation, en mettant l'accent sur les ajustements des niveaux de sévérité. La plupart des travaux se concentrent sur la manière dont les développeurs modifient et ajustent les instructions de log pour améliorer la qualité de la journalisation et assurer une meilleure visibilité des événements critiques. Ces études couvrent différents langages et types de systèmes, allant des applications d'entreprise aux systèmes mobiles et aux environnements d'apprentissage automatique.

La recherche aborde aussi des questions telles que la cohérence des niveaux de sévérité et les décisions entourant leur ajustement lors des révisions de code. Les travaux sont organisés en quatre sous-sections : la *caractérisation des pratiques*, en explorant des systèmes en C/C++ et Java ; *la catégorisation et les décisions de journalisation*, portant sur l'emplacement des instructions de log, la qualité des messages et le choix des niveaux ; *une analyse des mauvaises pratiques*, incluant les incohérences de niveaux de sévérité et les redondances inutiles ; enfin, une sous-section dédiée aux *contextes spécifiques de journalisation*, avec des études sur les tests, les environnements de production et de développement, ainsi que sur la journalisation dans les systèmes d'apprentissage automatique.

2.1.1 CARACTÉRISATION DES PRATIQUES DE JOURNALISATION (CPJ)

L'une des premières études sur les pratiques de journalisation, Yuan *et al.* (2012b), analyse les *commits* sur une période de cinq ans de quatre systèmes en C/C++ – Apache HTTPD [13], OpenSSH [14], PostgreSQL [15] et Squid [16] – en se concentrant sur les modifications qui altèrent le comportement de la journalisation, telles que les niveaux de sévérité, le contenu statique et l'emplacement des messages de log.

Les modifications sont classées en deux types : *celles qui visent à promouvoir la cohérence* avec d'autres parties du code dans la même révision, et *celles qui représentent des ajustements effectués après une analyse postérieure plus approfondie*. L'étude constate que le taux moyen de modifications des instructions de log était environ deux fois plus élevé que celui du reste du code. Bien que la densité du code de journalisation soit relativement faible, elle est modifiée dans environ 18% des *commits*, et 33% de ces modifications sont effectuées après des réflexions postérieures sur des instructions de log incohérentes.

Par ailleurs, l'étude observe que 26% des changements d'instructions de logs implique des ajustements de niveau de sévérité, et que dans 28% de ces modifications, les développeurs reconsidère les compromis entre différents niveaux de sévérité, ce qui indique une compréhension diffuse quant à l'évaluation des coûts (par exemple, des messages excessifs, des surcharges) et des avantages de chaque niveau de sévérité.

Pour clarifier ces ajustements, l'étude classe les mises à jour de niveaux de sévérité en trois catégories :

13. https://httpd.apache.org/
14. https://www.openssh.com/
15. https://www.postgresql.org/
16. https://www.squid-cache.org/

- *des niveaux d'erreur,* qui impliquent des changements vers ou depuis les niveaux de sévérité élevés, tels que *Error* et *Fatal*.
- *des niveaux non-erronés avec niveau par défaut*, où le niveau de log précédent ou actuel est un niveau par défaut (comme *Info*).
- *des niveaux non-erronés sans niveau par défaut*, où les changements impliquent des niveaux de développement (par exemple, de *Debug* à *Trace*) sans utiliser le niveau par défaut du projet.

Selon l'étude, dans 72% des cas, les modifications des niveaux de sévérité concernent des événements d'erreur, ce qui suggère que le caractère critique de l'événement a été initialement sous-estimé ou surestimé. En plus, 28% des modifications concernent des événements non liés à des erreurs, où les développeurs réévaluent l'importance de l'événement enregistré. Cette réévaluation peut s'expliquer par le fait que l'enregistrement d'une véritable erreur avec un niveau de sévérité réservé au développement, tel que *Debug*, pourrait entraîner la perte de messages d'erreur importants pour le diagnostic des défaillances. De même, l'enregistrement d'un événement non-erroné avec un niveau d'erreur pourrait soit induire en erreur les utilisateurs et les développeurs, soit entraîner une surcharge inutile lors de l'exécution en production.

L'étude observe également que plus de 50% des modifications entre les niveaux d'erreur impliquent la transition entre des erreurs « non fatales » et « fatales ». Elle suggère qu'élever une erreur du niveau *Error* au niveau *Fatal* vise à éviter la propagation d'erreurs critiques dans les systèmes, tandis qu'atténuer une erreur de *Fatal* à *Error* vise à améliorer la disponibilité en évitant une interruption inutile des systèmes. Enfin, pour les événements non liés à des erreurs, les résultats indiquent que le choix du niveau de sévérité génère souvent de la confusion. L'utilisation de niveaux trop élevés peut entraîner une surcharge de messages, tandis que

des niveaux plus bas peuvent entraîner la perte d'informations importantes pour l'analyse ultérieure.

L'étude présent aussi un vérificateur de niveau de sévérité pour identifier les incohérences à travers des extraits de code similaires, détectant des clones de code et s'assurant que le code de journalisation maintient une verbosité cohérente.

Une étude de réplication obtient des résultats contradictoires en considérant les systèmes Java. Chen & Jiang (2017b) reproduisent l'étude antérieure en l'appliquant à 21 systèmes Java, avec quelques adaptations spécifiques aux particularités de ces projets. Les données utilisées dans cette étude incluent :

- la dernière version stable de chacun des systèmes à l'époque, qui a été spécifiquement utilisée pour calculer l'une des métriques de l'étude, à savoir la densité de logs.
- les rapports, ouverts entre 2000 et 2015, provenant de deux systèmes de suivi des incidents (BugZilla [17] et Jira).
- l'historique des révisions de code issu des *dumps* SVN, un système de contrôle de version *open-source* logiciels [18], couvrant les années 1999 à 2014.

À partir de l'historique des révisions, ils identifient les modifications des instructions de log en utilisant des expressions régulières telles que : « `.?(pointcut|aspect|log|info|debug|error|fatal|warn|trace|(system.out)|(system.err)).?(.*?);` ».

Pour analyser les ajustements postérieurs, les auteurs ont développé un programme pour détecter les changements dans les instructions de log, y compris le niveau de sévérité, entre deux révisions adjacentes. Ils utilisent les mêmes catégories que celles définies dans l'étude originale, mais contrairement à celle-ci, 76% des ajustements de sévérité dans les

17. https://www.bugzilla.org/

18. https://subversion.apache.org/

projets côté serveur étaient liés à des événements non relatifs à des erreurs. Les ajustements à partir du niveau par défaut étaient les plus fréquents, environ 65%, tandis que seulement 15% concernaient des ajustements de niveaux non par défaut (niveaux de développement), contre 57% dans l'étude initiale. Les auteurs suggèrent que cela pourrait être dû à une meilleure compréhension des niveaux de sévérité par les développeurs Java, par rapport aux développeurs C/C++ de l'étude de Yuan *et al.* (2012b). Ils expliquent que, dans les projets Java, les niveaux de sévérité, souvent issus de bibliothèques de journalisation populaires, sont peut-être mieux définis que dans les projets en C/C++. Malgré cette hypothèse, ils reconnaissent la nécessité de mener davantage d'études pour mieux comprendre ce phénomène.

2.1.2 CATÉGORISATION ET DÉCISIONS DE JOURNALISATION (CDJ)

Les études suivantes font progresser la catégorisation des pratiques de journalisation, chacune apportant une perspective spécifique. Fu *et al.* (2014) proposent une catégorisation des blocs de code qui produisent des logs ; Li *et al.* (2017b) présentent une typologie des raisons des changements dans la journalisation ; He *et al.* (2018) se concentrent sur les catégories de descriptions des messages de journalisation ; Zeng *et al.* (2019), en reproduisant l'étude de Yuan *et al.* (2012b), élargissent la catégorisation initiale des modifications des niveaux de sévérité, tandis que Li *et al.* (2020a) enrichissent cette catégorisation en intégrant l'équilibre entre les coûts et les bénéfices de la journalisation. Les approches et principaux résultats de chaque étude sont détaillés ci-après.

Une étude, menée par Fu *et al.* (2014), propose une analyse systématique des pratiques de journalisation dans deux grands systèmes de Microsoft, avec pour objectif de comprendre où les développeurs insèrent des instructions de log dans le code. En utilisant une analyse statique et des questionnaires auprès de 54 développeurs, l'étude identifie cinq catégories principales de fragments de code qui produisent généralement des journaux : *vérification d'assertions,*

vérification des valeurs de retour, blocs de capture d'exceptions, branches logiques et *points d'observation.*

L'analyse révèle qu'environ la moitié des logs est utilisée pour enregistrer des situations inattendues (comme des exceptions), tandis que l'autre moitié enregistre des informations normales d'exécution. Seule une petite fraction des blocs de capture d'exceptions est journalisée (30% à 42%), ce qui suggère que tous les fragments de code critiques ne nécessitent pas de journalisation. En plus de l'analyse de code, les auteurs présentent un modèle prédictif utilisant un algorithme d'arbre de décision. Ce modèle a été entraîné sur la base de caractéristiques contextuelles extraites des fragments de code pour prédire, avec une précision allant jusqu'à 90 %, mesurée à l'aide du F-score. Le F-score est une métrique qui permet de résumer à la fois la précision (le nombre de prédictions correctes parmi les prédictions positives) et le rappel (le nombre de cas positifs effectivement identifiés) (Goutte & Gaussier, 2005). Il est couramment utilisé pour évaluer l'efficacité globale d'un modèle de classification, en particulier lorsque les classes sont déséquilibrées.

D'un autre côté, l'analyse de Li *et al.* (2017b) étend ce cadre en se concentrant sur les raison des changements dans la journalisation à travers quatre projets Apache : Hadoop, Directory Server [19], Commons HttpClient [20] et Qpid [21]. Pour chacun de ces projets, les auteurs récupèrent l'historique des *commits* de la branche principale (*trunk*), en se concentrant uniquement sur le code source Java. Plus de 5 000 *commits* sont examinés pour chaque projet, avec entre 22,7% et 30% des *commits* contenant des modifications des logs (ajout, modification ou suppression d'instructions de log).

19. https://directory.apache.org/apacheds/
20. https://hc.apache.org/httpclient-legacy/
21. https://qpid.apache.org/

Le processus d'extraction des données débute par une analyse des dépôts de contrôle de version, avec pour objectif d'identifier les *commits* contenant des modifications des instructions de log. Une base de données spécifique est constituée pour centraliser ces modifications et faciliter l'analyse manuelle. En outre, il est constaté que seulement 1,2% à 4,2% des *commits* modifient exclusivement les logs, sans affecter d'autres parties du code. Les auteurs identifient quatre catégories de modifications de logs, subdivisées en 20 raisons distinctes. Ces catégories incluent : *modification des blocs de code*, *amélioration de la journalisation*, *changements induits par des dépendances* et *correction des problèmes de journalisation*.

Parmi les 380 modifications de code liées à la journalisation étudiées, la catégorie *modification des blocs de code* est la plus courante, représentant 260 modifications. En revanche, en ce qui concerne les niveaux de sévérité, une seule raison est identifiée dans la catégorie *problèmes de journalisation*. Parmi les 18 modifications recensées dans cette catégorie, la sous-catégorie *niveau de log inapproprié* se distingue avec 13 occurrences.

L'étude souligne l'importance d'optimiser les niveaux de sévérité pour éviter le bruit excessif ou la perte d'informations essentielles, assurant ainsi une journalisation plus efficace et mieux adaptée aux systèmes en production et en développement. Elle met également en lumière la nécessité d'approfondir la compréhension des ajustements de sévérité des logs dans les recherches actuelles.

Poursuivant la problématique de la catégorisation des pratiques de journalisation, mais en se concentrant cette fois sur les messages de log, He *et al.* (2018) conduisent une étude empirique sur l'utilisation des messages en langage naturel dans les instructions de log, en analysant 10 projets Java et 7 projets C#. Les messages de journalisation sont classés en trois catégories principales : *les messages sur les opérations du programme* (37%), *les messages d'erreurs* (40%) et *les messages sémantiques* (23,5%).

Les messages sur les opérations du programme détaillent les actions effectuées par le système, pouvant décrire des opérations terminées, en cours ou des opérations à venir. Quant aux messages d'erreurs, ils rapportent l'occurrence d'exceptions ou de défaillances du système, en mettant l'accent sur la vérification des valeurs et les exceptions capturées par des blocs `try-catch`. Enfin, les messages sémantiques sont utilisés pour enregistrer la valeur des variables critiques, décrire le fonctionnement des fonctions ou capturer la signification sémantique des branches dans le code. Cette catégorisation est utilisée dans un travail ultérieur (Li *et al.*, 2021c) portant sur l'automatisation de la suggestion des niveaux de sévérité.

Dans cette continuité, Zeng *et al.* (2019) reproduisent l'étude de Yuan *et al.* (2012b) pour examiner les pratiques de journalisation dans les applications Android [22], en cherchant à comprendre les raisons pour journaliser et les impacts des logs sur la performance des applications. L'étude est appliquée sur 1 444 applications Android et révèle que les ajustements de niveaux de sévérité sont moins fréquents par rapport aux études de Yuan *et al.* (2012b) et Chen & Jiang (2017b). De plus, les ajustements dans la catégorie des *niveaux d'erreur* sont moins courants, confirmant les résultats de Chen & Jiang (2017b).

En termes de répartition des niveaux de gravité, ils constatent que plus de la moitié des instructions de log dans les applications se trouvent aux niveaux *Debug* et *Error*, ce qui peut indiquer que les développeurs exploitent souvent les logs pour déboguer et enregistrer les erreurs d'exécution des applications mobiles. De plus, que les instructions de log dans les applications mobiles sont moins fréquemment maintenues. Environ 10% des *commits* contiennent des ajouts, suppressions ou modifications d'instructions de log, un pourcentage nettement plus bas que celui observé dans les applications étudiées par Yuan *et al.* (2012b).

22. https://www.android.com/

Concernant les raisons de journaliser, Zeng *et al.* (2019) utilisent une technique connue sous le nom de *« firehouse interview »* (Murphy-Hill *et al.*, 2014), envoyant des courriels d'entretien aux développeurs juste après l'identification de la correction d'un bogue, afin de tirer parti de la mémoire des décisions récentes. Dans ce contexte, ils identifient huit raisons pour lesquelles la journalisation est utilisée, *principalement pour déboguer*, en imprimant des variables, des traces de pile ou des chaînes de caractères pour enregistrer les étapes d'exécution, et *pour la détection d'anomalies*, en utilisant les instructions de log pour enregistrer des événements inattendus à l'exécution. Les autres raisons rapportées incluent :

- *journalisation comptable* : des instructions de log pour améliorer la compréhension du comportement des applications à l'exécution ;
- *assistance au développement* : des instructions de log pour assister le développement des applications ;
- *performance* : des instructions de log pour mesurer la performance ;
- *cohérence* : des instructions de log ajoutées pour suivre un modèle de code existant ;
- *personnalisation de la bibliothèque de journalisation* : certains développeurs créent leur propre bibliothèque de journalisation personnalisée en adaptant celle par défaut d'Android, plutôt que d'en utiliser une tierce ;
- *bibliothèque tierce* : certains logs proviennent de bibliothèques tierces dont le code source est copié directement, sans référence à une bibliothèque compilée, et peut inclure des instructions de journalisation.

En ce qui concerne le choix des niveaux de sévérité, les auteurs ont observé une incohérence entre le niveau choisi et le raisonnement des développeurs. Par exemple, 11,7% des instructions de log utilisées pour la détection d'anomalies emploient des niveaux comme *Debug*, tandis que 23,7% des instructions utilisent des niveaux comme *Info*, *Warn* et *Error* à des

fins de débogage. De tels niveaux de journalisation incohérents peuvent entraîner la divulgation d'informations non essentielles, ce qui pourrait provoquer une surcharge de performance.

Pour enquêter sur cette hypothèse, l'étude mène des tests sur huit applications entre les applications préalablement sélectionnées en fonction de trois critères : elles sont bien maintenues, contiennent plus de 500 instructions de log et possèdent des fichiers de test liés aux logs. Ces tests sont utilisés pour mesurer l'impact des logs inutiles sur la performance des applications. Les tests portent sur trois scénarios : tous les logs activés, tous désactivés et seulement les logs nécessaires activés. Les métriques analysées comprenaient la consommation de CPU, de batterie et le temps de réponse. Il est observé que, si la désactivation des instructions de log améliore de manière significative les performances, la surcharge causée par les journaux inutiles peut être statistiquement significative.

Enfin, en abordant une vision plus large sur les compromis associés à journalisation, Li *et al.* (2020a) étudient les coûts et les avantages mentionnés dans l'étude de Yuan *et al.* (2012b) en analysant 223 rapports d'incidents ainsi que les résultats d'une enquête menée auprès de 66 développeurs. L'étude aborde trois questions : la première sur les coûts, la deuxième sur les avantages, et la troisième sur la manière dont les développeurs les équilibrent. Pour sélectionner les participants, les auteurs prennent en compte l'expérience des développeurs en matière de journalisation dans les projets les plus étoilés de la Apache Software Foundation (ASF) [23]. Ils déterminent cette expérience en analysant l'historique des modifications de code de chaque développeur. Quant à l'investigation des rapports d'incidents, trois projets Apache sont examinés : Hadoop Common [24] (un sous-projet de Hadoop), Hive [25] et Kafka [26], avec

23. https://www.apache.org/
24. https://github.com/apache/hadoop/tree/trunk/hadoop-common-project.
25. https://hive.apache.org/
26. https://kafka.apache.org/

une analyse des rapports générés entre juin 2012 et juin 2017. Ils sélectionnent 902 rapports à partir de Jira, et après filtrage, ils en ont conservé 533.

Pour identifier les avantages de la journalisation, ils s'appuient sur la classification des raisons pour lesquelles les développeurs journalisent, proposée par Zeng *et al.* (2019). Li *et al.* (2020a) élargissent cette perspective en regroupant les avantages de la journalisation en quatre dimensions principales, offrant ainsi une structure plus systématique. Les quatre dimensions proposées par Li *et al.* (2020a) sont :

- *assistance au dépannage*, qui inclut le diagnostic des échecs à l'exécution et le soutien aux utilisateurs ;
- *suivi de l'état d'exécution*, avec des fonctionnalités de suivi de la progression et de surveillance ;
- *assistance à la compréhension,* visant à améliorer la compréhension des systèmes et à soutenir le développement logiciel ;
- *journalisation comptable*, correspondant à la même catégorie identifiée par Zeng *et al.* (2019).

En complément de l'analyse des avantages, Li *et al.* (2020a) proposent également une classification des coûts associés à la journalisation, regroupés en quatre catégories principales :

- *gestion et traitement de grandes quantités de données de log* : les coûts incluent le stockage des logs, la production de bruit informationnel ainsi que les efforts nécessaires pour collecter, traiter et gérer ces données ;
- *impact sur le comportement du système* : les performances des systèmes peuvent être affectées par la surcharge induite par les logs, en plus des perturbations possibles dans le comportement des systèmes ;

- *impact direct sur les utilisateurs* : les utilisateurs peuvent être déroutés par des informations superflues générées par les logs, et il existe également un risque de divulgation d'informations sensibles ;
- *augmentation des efforts de développement* : les développeurs doivent non seulement prendre en compte le coût de développement et de maintenance du code de journalisation, mais également gérer la baisse de lisibilité du code causée par l'ajout d'instructions de log.

Selon l'étude, un excès de journalisation peut non seulement augmenter les coûts de gestion et de traitement des données de log et affecter les comportements d'exécution du système, mais il a également un impact plus direct sur les développeurs et les utilisateurs. Ceci inclut des efforts accrus pour le développement et la maintenance, ainsi que le risque de confusion pour les utilisateurs ou de divulgation d'informations sensibles. Les développeurs doivent bien comprendre ces coûts pour minimiser les impacts négatifs inutiles, comme le souligne Oliner *et al.* (2012).

Pour gérer ces impacts, l'attribution appropriée des niveaux de sévérité des logs apparaît comme l'une des approches les plus courantes pour équilibrer les coûts et les avantages de la journalisation. Cette approche inclut non seulement l'utilisation correcte et dynamique des niveaux de sévérité, mais aussi l'évaluation continue et la refactorisation de ces niveaux, en plus de privilégier la production de logs pour le développement (comme le débogage) plutôt que pour la production. Une autre stratégie consiste à minimiser les journaux répétitifs en regroupant les messages récurrents dans des informations synthétisées à un niveau de sévérité plus élevé, tout en enregistrant les détails au niveau inférieur.

De plus, ils observent que les développeurs utilisent souvent des stratégies improvisées pour ajuster la journalisation, que ce soit de manière proactive, en déterminant où et quand

enregistrer des logs, ou de manière réactive, en répondant aux besoins à la demande. Par exemple, il est courant que des instructions de log soient laissées dans le code après son écriture initiale, avec des ajustements de niveau effectués lors de l'intégration finale du code (Ding *et al.*, 2015). Ces stratégies visent à différencier les objectifs de journalisation, à minimiser son impact sur les performances et à améliorer la qualité des entrées de logs.

2.1.3 MAUVAISES PRATIQUES DE JOURNALISATION (MPJ)

En ce qui concerne la résolution des problèmes liés à la journalisation, Chen & Jiang (2017a), Hassani *et al.* (2018) et Li *et al.* (2021b) identifient plusieurs problèmes dans des projets *open-source*, y compris des niveaux de log inappropriés, et proposent des solutions automatisées pour détecter et corriger les incohérences entre les messages de log et leurs niveaux de sévérité.

L'étude de Chen & Jiang (2017a) identifie et caractérise les anti-patrons qui apparaissent lors de l'évolution du code de journalisation. Pour ce faire, 352 paires de fragments de code de journalisation modifiés indépendamment sont examinées manuellement dans trois systèmes *open-source* : ActiveMQ [27], Hadoop [28] et Maven [29].

Le processus a suivi une approche basée sur la Théorie Ancrée (Charmaz, 2006), isolant et analysant les modifications du code de journalisation qui se sont produites indépendamment des modifications du code fonctionnel. Des modifications de code détaillées sont extraites à partir des dépôts historiques, en utilisant des techniques d'analyse de programmes pour identifier les changements dans le code de journalisation. Ces changements sont catégorisés en deux types : *les changements de journalisation associés aux changements de code fonctionnel*

27. https://activemq.apache.org/
28. https://hadoop.apache.org/
29. https://maven.apache.org/

et *les changements de journalisation indépendants*. Les premiers correspondent à des cas où les modifications de journalisation accompagnent des évolutions du code métier, telles que le renommage d'une variable ou la restructuration d'une méthode, et visent à conserver la cohérence du log avec le comportement du système. Les seconds, en revanche, sont réalisés sans changement concomitant dans le code fonctionnel, et sont souvent motivés par le besoin de corriger des erreurs, de clarifier le message enregistré ou d'améliorer la qualité du log. L'analyse manuelle se concentre sur les changements indépendants, considérés comme révélateurs de mauvaises pratiques passées, permettant ainsi de caractériser six anti-patrons de journalisation dans les trois systèmes étudiés.

L'étude décrit cinq catégories d'anti-patrons dans le code de journalisation, à savoir : *la vérification des variables nullables, la suppression du casting d'objets, la correction des niveaux de sévérité, le refactoring du code de journalisation* et *la modification du format de sortie*. En ce qui concerne la correction des niveaux de sévérité, bien qu'il existe des directives sur l'association d'informations à chaque niveau de sévérité (dans ce cas, un guide d'une des bibliothèques de journalisation [30]), celles-ci ne sont souvent pas suivies, ce qui indique la nécessité de directives plus spécifiques pour traiter les cas complexes et variés rencontrés dans la pratique. Ils soulignent, comme mentionné dans les travaux ci-dessus, que cet anti-patron peut entraîner une surcharge de journalisation et des volumes importants de données redondantes lors de l'analyse des logs.

Chen & Jiang (2017a) ont aussi introduit LCAnalyzer, un outil d'analyse de code statique conçu pour détecter les anti-patrons, y compris les niveaux de sévérité incorrects dans les pratiques de journalisation.

30. https://commons.apache.org/proper/commons-logging/guide.html#JCL_Best_Practices

L'étude de Hassani *et al.* (2018) se concentre sur l'analyse des problèmes liés à la journalisation dans deux systèmes Apache : Hadoop et Camel. La sélection de ces systèmes a eu lieu après l'analyse des 1 000 projets Java les plus populaires sur GitHub, basée sur le nombre d'étoiles, suivie d'un décompte des instructions de log dans les projets. Le choix de Hadoop et Camel est dû à leur grand nombre d'instructions de log et à leur large adoption dans l'industrie, selon les auteurs.

Pour collecter les incidents liés à la journalisation, Hassani *et al.* (2018) utilisent une méthode basée sur des mots-clés tels que « log », « logging » et « logger », appliquée aux rapports d'incidents disponibles dans les systèmes étudiés via Jira. Seuls les incidents étiquetés comme « bogue » ou « amélioration », résolus et corrigés, sont inclus dans l'analyse. Les incidents liés à de nouvelles fonctionnalités ou ceux qui n'étaient pas correctement résolus ou fermés sont exclus. Au total, 563 incidents liés à la journalisation sont identifiés et analysés manuellement afin de comprendre leurs causes principales et leurs caractéristiques.

Les chercheurs catégorisent les problèmes liés à la journalisation en sept catégories en fonction de leurs causes principales, à savoir : *messages inappropriés*, *instructions manquantes*, *niveau inapproprié*, *problèmes de configuration de la bibliothèque de journalisation*, *problèmes d'exécution, surcharge de journaux* et *modifications de la bibliothèque de journalisation.* En ce qui concerne le niveau de journalisation inapproprié, ils soulignent que la sélection d'un niveau de sévérité trop bas peut entraîner l'omission d'informations importantes, tandis qu'un niveau trop élevé peut générer des informations redondantes, compliquant ainsi l'analyse des logs.

À l'exemple de Chen & Jiang (2017a), Li *et al.* (2021b) identifient des anti-patrons de journalisation. Ils examinent la présence d'instructions de log dupliquées et leurs relations avec

les clones de code dans cinq systèmes *open-source* : Hadoop, CloudStack, ElasticSearch [31], Cassandra [32] et Flink [33]. Ils définissent les instructions dupliquées comme celles qui partagent le même message de log statique et ont mené une analyse manuelle pour identifier des motifs de duplication pouvant être considérés comme des « *code smells* », c'est-à-dire des indicateurs en surface pouvant révéler des problèmes plus profonds dans le système (Fowler *et al.*, 1999). Les auteurs identifient cinq patrons de *code smells* parmi plus de 4 000 instructions de log dupliquées, à savoir :

1. *informations inadéquates dans les blocs de capture* : des logs dupliqués dans des blocs d'exception qui ne précisent pas correctement l'exception capturée.
2. *incohérence dans les informations de diagnostic d'erreurs* : des logs qui documentent les exceptions de manière incohérente, par exemple lorsqu'une instruction de log inclut la pile d'erreurs et l'autre non.
3. *incompatibilité des messages de log* : erreurs causées par le copier-coller de code sans ajuster les messages de log.
4. *niveau de sévérité incohérent* : des messages de log identiques avec des niveaux de sévérité différents.
5. *logs dupliqués dans les blocs de polymorphisme* : des logs dupliqués dans des méthodes qui partagent des superclasses ou des interfaces.

L'étude explore également la relation entre les instructions de log dupliquées problématiques et les clones de code. Les auteurs observent que 83% des cas problématiques se trouvent dans des extraits de code clonés, ce qui suggère que la pratique du clonage de code puisse entraîner des pratiques inadéquates de journalisation. Cependant, 17% des instructions

31. https://www.elastic.co/
32. https://cassandra.apache.org/_/index.html
33. https://flink.apache.org/

dupliquées problématiques se situent dans des blocs de code courts, difficiles à détecter avec des outils de détection de clones.

Li *et al.* (2021b) analysent les incohérences dans les niveaux de sévérité des instructions de log dupliquées et présentent un outil automatisé pour les détecter et les corriger. Pour automatiser la détection de ces problèmes, ils implémentent l'outil d'analyse statique DL-Finder, qui s'appuie sur les patrons identifiés et les catégorisations établies lors de l'étude manuelle. Évalué sur huit systèmes (cinq issus de l'étude manuelle et trois supplémentaires), l'outil identifie avec succès des patrons problématiques (c.-à-d. nécessitant une correction) et justifiables (c.-à-d. ne nécessitant pas de correction), tout en montrant que des problèmes liés à la duplication des logs peuvent apparaître au fur et à mesure de l'évolution des systèmes. DLFinder signale 91 cas de *code smells* de logs dupliqués, qui ont tous été corrigés par les développeurs.

2.1.4 CONTEXTES SPÉCIFIQUES DE JOURNALISATION (CSJ)

La journalisation s'adapte aux particularités de différents contextes et environnements logiciels, exigeant des pratiques spécifiques pour répondre aux besoins de chaque domaine. Dans cette section, nous explorons des recherches sur la journalisation dans quatre contextes : le traitement des exceptions Li *et al.* (2021a), les différences entre les environnements de test et de production Zhang *et al.* (2022), le noyau Linux Patel *et al.* (2022) et les systèmes d'apprentissage automatique Foalem *et al.* (2024). Ces études incluent l'analyse des pratiques et des nuances liées aux niveaux de sévérité.

Li *et al.* (2021a) analysent le code source, l'historique des modifications et les rapports d'incidents de dix projets *open-source* pour comprendre les pratiques des développeurs concernant la journalisation des traces d'exceptions. Les projets étudiés incluent Hadoop,

Directory Server, Hive, Zookeeper, Kafka, Qpid, ActiveMQ, Camel [34], CloudStack [35] et JMeter [36]. L'objectif de cette étude est d'identifier les raisons motivant l'enregistrement ou non des traces d'exceptions, ainsi que les facteurs influençant ces pratiques.

Les chercheurs récupèrent le code source et l'historique des modifications à partir des systèmes de contrôle de version et utilisent des outils d'analyse statique pour extraire les instructions de log des exceptions ainsi que leurs informations contextuelles, telles que le type d'exception.

Pour automatiser l'extraction des instructions de log d'exceptions, un plugin IntelliJ [37] a été développé, permettant d'identifier les blocs de capture d'exceptions dans le code Java et les niveaux de sévérité associés. Les chercheurs analysent l'historique des modifications à l'aide de la commande « `git diff` » une commande permettant de comparer deux versions d'un fichier pour identifier les différences, y compris les modifications dans les instructions de log (Git, 2024). Cette approche vise à détecter les ajouts, suppressions et modifications portant spécifiquement sur les logs d'exceptions. Ils utilisent des expressions régulières pour repérer les changements spécifiques, tels que l'ajout ou la suppression de traces d'exceptions. Afin de mieux comprendre les raisons derrière l'enregistrement ou non des traces d'exceptions, ils examinent les rapports d'incidents manuellement.

L'étude aborde une discussion plus large sur les niveaux de sévérité de log, en les comparant aux autres études. Elle suggère que les traces de pile d'exceptions doivent être évitées ou enregistrées à des niveaux inférieurs pour les erreurs liées aux utilisateurs, les exécutions normales et les exceptions prévues, soulignant ainsi l'importance d'une bonne

34. https://camel.apache.org/
35. https://cloudstack.apache.org/
36. https://jmeter.apache.org/
37. https://www.jetbrains.com/idea/

gestion des niveaux de sévérité. Ce travail met en lumière la nécessité de directives claires pour la journalisation.

En explorant les spécificités de chaque contexte, Zhang *et al.* (2022) investiguent les différences dans les caractéristiques de journalisation entre les environnements de production et de test. Ils conduisent une étude sur 21 projets Apache, parmi lesquels Hadoop, Hbase [38], Hive, Zookeeper [39], Tomcat [40], Lucene [41] et ActiveMQ. Pour identifier ces différences, ils analysent séparément la densité, les distributions et les données historiques des instructions de log de test et de production. Comme Zeng *et al.* (2019), ils réalisent une enquête de type « *firehouse email* » (Murphy-Hill *et al.*, 2014) afin de comprendre pourquoi les développeurs utilisent des instructions de log dans les fichiers de test. Ensuite, ils examinent les logs de test selon leur relation avec les logs de production, explorant ainsi les liens entre ces deux types de journalisation.

Concernant les raisons de l'utilisation des logs en test, ils rapportent que les développeurs les utilisent principalement pour quatre motifs : *la détection de bogues, le suivi des informations opérationnelles, la refactorisation du code* et *la duplication du code*, les deux premiers étant largement prédominants. En ce qui concerne les niveaux de sévérité, leurs résultats montrent que :

- dans les instructions de log en test et en production, le niveau *Info* est le plus couramment utilisé et domine dans les fichiers de test, tandis que *Trace* est le moins utilisé ;
- la distribution des niveaux de sévérité est plus équilibrée dans les instructions de log de production que dans celles de test.

38. https://hbase.apache.org/
39. https://zookeeper.apache.org/
40. https://tomcat.apache.org/
41. https://lucene.apache.org/

Selon l'étude, une raison possible pour laquelle le niveau *Info* est si prédominant dans la pratique de journalisation est qu'il constitue souvent le niveau de sévérité par défaut dans les systèmes logiciels, c'est-à-dire le plus bas niveau de journalisation qui rend les logs visibles aux développeurs à l'exécution. Quant au niveau *Debug*, qui est le deuxième niveau le plus courant à la fois dans les fichiers de test et de production, son utilisation fréquente pourrait être liée à l'intention des développeurs de se concentrer sur la détection et la résolution de bogues pendant le cycle de développement.

Pour le noyau Linux, l'étude de Patel *et al.* (2022) explore l'évolution des pratiques de journalisation en analysant 22 versions du noyau publiées entre 2015 et 2019. Plus de 14 000 *commits* sont examinés pour évaluer la prévalence et l'évolution du code de journalisation au fil du temps. Pour cette analyse, les chercheurs récupèrent le code source et l'historique des modifications en utilisant des outils automatiques et des méthodes basées sur des patrons sémantiques pour identifier les fonctions et macros de journalisation, telles que « `printk()` », une fonction utilisée dans le noyau Linux pour enregistrer des messages avec un niveau de journalisation spécifique (Linux, 2024), ainsi que d'autres fonctions spécifiques au noyau. Cette approche permet de collecter des données sur l'insertion, la modification et la suppression des instructions de log dans les différentes versions du code. versions.

De manière similaire à l'étude de Li *et al.* (2021a), les modifications du code de journalisation sont analysées à l'aide de la commande « `git diff` », tandis que des expressions régulières sont employées pour identifier les ajouts, suppressions et modifications dans les niveaux de log. De plus, les chercheurs examinent manuellement 900 *commits*, sélectionnés de manière aléatoire, afin de comprendre les motivations des développeurs lorsqu'ils modifient le code de journalisation. Des métriques telles que la densité de logs et le ratio de lignes de code consacrées à la journalisation sont calculées, comparant l'utilisation du logging dans différents sous-systèmes du noyau.

Les changements dans les niveaux de sévérité sont particulièrement marquants : sur les 4 127 ajustements identifiés, 92,85% se sont produits entre les niveaux *Error*, *Warning*, *Info* et *Debug*, ce qui peut indiquer l'absence de normes claires pour le choix des niveaux de log. Une analyse plus détaillée a montré qu'environ un tiers de ces changements impliquent des échanges entre les niveaux *Error* et *Debug*, avec 17,45% des instructions attenuées de *Error* à *Debug* pour éviter une surcharge de logs, et 14,93% aggravées de *Debug* à *Error* afin d'accroître la visibilité des messages.

Enfin, dans un domaine distinct, Foalem *et al.* (2024) conduisent une étude empirique visant à comprendre les pratiques de journalisation dans les applications basées sur l'apprentissage automatique (*Machine Learning* ou ML). L'étude examine 502 projets *open-source* de ML, révélant que la densité des journaux dans ces systèmes est significativement inférieure à celle des applications traditionnelles, telles que celles développées en C/C++, Java et Android. En plus, une prédominance des niveaux de sévérité *Info* et *Warn* est observée dans les applications de ML, avec 80% des instructions de log utilisant ces niveaux. Les auteurs attribuent cette tendance à l'utilisation fréquente de ces catégories pour la gestion des données et des modèles, en particulier lors du processus d'entraînement des modèles de ML. Les auteurs soulignent également que les pratiques de journalisation dans les systèmes ML diffèrent de celles des logiciels traditionnels, tant en termes de fréquence que de finalité des instructions de log. Alors que le niveau *Info* est majoritairement utilisé pour consigner les paramètres des modèles et les informations de traitement des données, le niveau *Warn* est employé pour signaler des incohérences dans les opérations de prétraitement des données.

2.1.5 DISCUSSION ET MOTIVATIONS DU PRÉSENT MANUSCRIT

Dans cette section, nous avons présenté les études décrivant les pratiques de journalisation liées aux niveaux de sévérité, organisées en quatre sous-catégories : la caractérisation des

pratiques, la catégorisation des décisions, les mauvaises pratiques et les contextes spécifiques d'utilisation (Tableau 2.1). La plupart de ces travaux repose sur des analyses statiques de code, combinant la comparaison entre différentes versions des systèmes étudiés, l'analyse des rapports d'incidents, ainsi que l'examen des commentaires associés aux modifications.

En mettant ces travaux en perspective, nous pouvons observer une maturation des résultats. Par exemple, l'une des premières propositions classifiait les changements de niveaux de sévérité en utilisant trois catégories à partir du niveau par défaut, reflétant principalement des ajustements entre les niveaux d'erreur et de non-erreur (Yuan *et al.*, 2012b; Chen & Jiang, 2017a). Des études plus récentes ont progressé en indiquant de nouveaux ensembles de catégories, que ce soit en étudiant les raisons pour lesquelles les logs sont enregistrés (Zeng *et al.*, 2019; Li *et al.*, 2020a, 2021b; Zhang *et al.*, 2022), en soulignant le débogage et l'amélioration des logs, ou en explorant où les logs doivent être enregistrés (Fu *et al.*, 2014), en identifiant des catégories liées à la structure, telles que des blocs de code spécifiques, ou encore en fonction de caractéristiques contextuelles (He *et al.*, 2018), décrivant des niveaux utilisés dans les messages relatifs aux opérations ou aux erreurs.

Malgré ces progrès, certaines limitations persistent, ce qui motive notre travail. La divergence des résultats entre certaines recherches (Yuan *et al.*, 2012b; Chen & Jiang, 2017b; Zeng *et al.*, 2019), associée aux incohérences dans le choix des niveaux de sévérité, souvent identifiées comme des anti-patrons (Chen & Jiang, 2017a; Hassani *et al.*, 2018; Li *et al.*, 2021b), suggère que le choix des niveaux de sévérité va au-delà des spécificités de chaque plateforme, renforçant ainsi la nécessité de lignes directrices globales et indépendantes de la plateforme. Une hypothèse avancée est que les niveaux de sévérité sont mieux définis dans les bibliothèques de journalisation Java (Chen & Jiang, 2017b), mais à ce jour, aucune étude approfondie sur ces niveaux dans ces bibliothèques n'a été réalisée.

TABLEAU 2.1 : Pratiques de journalisation.
© Eduardo Mendes de Oliveira, 2024

Catégorie	Étude	Méthode	Résultats Principaux
CPJ	Yuan *et al.* (2012b)	Analyse des *commits* dans 4 systèmes C/C++ (5 ans).	Les ajustements de sévérité sont fréquents, 33 % des modifications de logs corrigent des incohérences.
	Chen & Jiang (2017b)	Réplication d'Yuan *et al.* (2012b) avec 21 systèmes Java.	76 % des ajustements de sévérité dans les systèmes Java concernent des événements non liés à des erreurs.
CDJ	Fu *et al.* (2014)	Analyse statique et questionnaires auprès de 54 développeurs, modèle prédictif basé sur un algorithme d'arbre de décision.	5 catégories de code reçoivent la journalisation (*assertions, vérifications des valeurs de retour, blocs d'exceptions, branches logiques, points d'observation*). Environ 50 % des logs enregistrent des situations inattendues. Précision du modèle prédictif jusqu'à 90 % (F-Score).
	Li *et al.* (2017b)	Analyse manuelle de 4 systèmes et regroupement des raisons de modification des instructions de log en quatre catégories	Identification automatique des incohèrences de sévérité
	He *et al.* (2018)	Analyse empirique de messages de journalisation dans 10 projets Java et 7 projets C#.	Classification des messages en trois catégories : *opérations du programme* (37 %), *erreurs* (40 %), et *sémantiques* (23,5 %).
	Zeng *et al.* (2019)	Analyse des logs dans 1 444 apps Android.	Les ajustements de sévérité sont moins fréquents dans les applications mobiles ; 10 % des commits impliquent des logs.
	Li *et al.* (2020a)	Analyse de 223 rapports d'incidents.	Catégorise les avantages et les coûts de la journalisation en quatre dimensions principales.
MPJ	Chen & Jiang (2017a)	Analyse manuelle de 352 paires de fragments de code de journalisation dans trois systèmes. *open-source* pour identifier des anti-patrons dans l'évolution des logs.	Identification de cinq anti-patrons de journalisation, y compris la correction des niveaux de sévérité et la suppression des objets non nécessaires. L'introduction de l'outil LCAnalyzer pour détecter les niveaux de sévérité incorrects et d'autres anti-patrons.
	Hassani *et al.* (2018)	Analyse de 563 incidents liés à la journalisation dans les systèmes Hadoop et Camel, collectés via Jira et catégorisés manuellement en fonction des causes principales.	Identification de sept catégories de problèmes de journalisation, incluant les niveaux de journalisation inappropriés, avec des cas où des niveaux de log trop bas ou trop élevés impactent la pertinence des journaux.
	Li *et al.* (2021b)	Analyse de cinq systèmes open-source pour détecter des instructions de journalisation dupliquées et leurs relations avec des clones de code. Développement de l'outil DLFinder pour automatiser la détection de motifs problématiques.	Identification de cinq patrons de *code smells* liés aux logs dupliqués, et découverte que 83 % des duplications problématiques se trouvent dans des clones de code. DLFinder a détecté 91 cas de *code smells* de logs dupliqués, tous corrigés.
CSJ	Li *et al.* (2021a)	Analyse du code source, historique des modifications et rapports d'incidents pour 10 projets open-source, avec extraction automatique des logs d'exceptions.	Identification des pratiques de journalisation des exceptions, recommandations pour éviter ou ajuster les traces d'exceptions à des niveaux inférieurs dans certains contextes (erreurs liées aux utilisateurs, exceptions prévues).
	Zhang *et al.* (2022)	Analyse des différences de journalisation entre les environnements de test et de production sur 21 projets Apache. Enquête par courriel pour comprendre les motivations des développeurs à utiliser des logs dans les fichiers de test.	Le niveau *Info* est le plus utilisé dans les journaux de test et de production, avec une distribution plus équilibrée en production. Les logs en test sont principalement utilisés pour la détection de bogues et le suivi des informations opérationnelles.
	Patel *et al.* (2022)	Analyse de 22 versions du noyau Linux. Utilisation d'outils automatiques et d'expressions régulières pour identifier les fonctions de journalisation et analyse manuelle de 900 commits pour comprendre les motivations des développeurs.	Sur les 4 127 ajustements de niveaux de sévérité identifiés, 92,85 % se produisent entre *Error*, *Warning*, *Info* et *Debug*, avec une tendance à atténuer les logs de *Error* à *Debug* (17,45 %) ou à les aggraver de *Debug* à *Error* (14,93 %).
	Foalem *et al.* (2024)	Analyse empirique de 502 projets *open-source* de ML pour examiner les pratiques de journalisation, en mettant l'accent sur la densité des journaux et l'utilisation des niveaux de sévérité.	La densité des journaux dans les applications ML est significativement inférieure à celle des applications traditionnelles. 80 % des instructions de journalisation utilisent les niveaux *Info* et *Warn*, principalement pour la gestion des données et des modèles, en particulier pendant l'entraînement des modèles de ML.

Une autre question importante est la prévalence des ajustements entre niveaux spécifiques, observée dans des travaux comme ceux de Yuan *et al.* (2012b); Li *et al.* (2021a); Foalem *et al.* (2024). Nous souhaitons examiner si ces ajustements de sévérité se produisent en raison d'un manque de connaissance sur l'utilisation correcte des niveaux ou si d'autres facteurs sont en jeu.

En ce qui concerne les catégories présentées par les études (Yuan *et al.*, 2012b; Chen & Jiang, 2017a; Fu *et al.*, 2014; Zeng *et al.*, 2019; Li *et al.*, 2020a), elles sont davantage axées sur l'usage global de la journalisation (*e.g.,* journalisation d'événements, débogage, assistance au dépannage, journalisation comptable, etc.) et moins spécifiques à l'utilisation de chaque niveau de sévérité. Par exemple, dans l'étude de Li *et al.* (2020a), les auteurs décrivent des motivations générales pour journaliser, telles que « suivre l'état d'exécution » ou « enregistrer des erreurs pour l'analyse », sans préciser comment ces objectifs se traduisent en niveaux concrets comme *Info*, *Warn* ou *Error*. Lorsqu'il est explicitement mentionné, le niveau de sévérité apparaît souvent comme une sous-catégorie ou une propriété secondaire des pratiques de journalisation, sans que les raisons précises de son assignation soient explorées. Cette absence de liens directs entre les catégories proposées et les décisions de sévérité crée une lacune dans la compréhension fine des niveaux, qui pourrait être à l'origine des incohérences ou des ajustements a posteriori observés dans la pratique.

Notre travail s'attaquera à cette lacune en réalisant une correspondance systématique de ce que la littérature et les bibliothèques de journalisation proposent en matière de niveaux de sévérité. Nous proposerons des définitions claires pour chaque niveau, ainsi qu'une catégorisation de leurs finalités, afin d'aligner les intentions d'utilisation des niveaux avec les pratiques dans différents contextes.

De plus, les catégories fournies par les études n'offrent pas toujours une applicabilité pratique suffisante pour une utilisation quotidienne, ce qui laisse des lacunes dans les lignes directrices sur la journalisation. Notre étude se concentrera également sur les ajustements des pratiques de journalisation, en cherchant à comprendre pourquoi le niveau de sévérité devient plus ou moins sévère. Bien que les études offrent une vue d'ensemble de ces ajustements (Li *et al.*, 2021a), elles n'approfondissent pas les raisons sous-jacentes, attribuant ces changements principalement à la difficulté de choisir le niveau de sévérité approprié.

Enfin, nous irons plus loin en examinant les facteurs contextuels, tels que l'expérience des développeurs et les pratiques de gestion de version, qui influencent la décision d'ajuster les niveaux de sévérité. Notre objectif est de fournir des orientations plus complètes et plus globales, applicables à différents scénarios de développement et d'exploitation des systèmes, en offrant des heuristiques pratiques qui améliorent la cohérence et l'efficacité des pratiques de journalisation.

2.2 APPROCHES AUTOMATISÉES

Dans la section précédente, certains travaux proposent des solutions logicielles pour aider soit à l'identification des problèmes liés au niveau de sévérité des logs, soit à la suggestion de niveaux corrects. Par exemple, Yuan *et al.* (2012b) présentent un vérificateur simple pour identifier les incohérences, tandis que Chen & Jiang (2017a) introduisent LCAnalyzer, un outil conçu pour détecter les anti-patrons liés à la journalisation.

Dans cette section, nous présentons des travaux qui ont proposé des approches automatisées pour les niveaux de sévérité des logs. Certains de ces travaux sont directement axés sur la sévérité des logs, tandis que d'autres sont plus larges, proposant des solutions allant de la recommandation de l'emplacement de journalisation à la génération complète des instructions

de log avec tous leurs composants. Les techniques proposées incluent des approches basées sur la théorie de l'information, l'apprentissage automatique, jusqu'aux modèles les plus récents de grands modèles de langage (LLM).

2.2.1 REVUE DES OUTILS ET MÉTHODES

Zhao *et al.* (2017) présentent Log20, un outil automatisé qui optimise le placement des instructions de log en se basant sur la théorie de l'information et en minimisant l'impact sur les performances du système. Évalué sur des systèmes tels que HDFS [42], HBase, Cassandra et Zookeeper, l'outil équilibre la quantité d'informations fournies par les logs et la surcharge de performance, permettant aux développeurs de spécifier des limites, comme un impact de 2% sur les performances. Comparé à l'approche manuelle, l'outil génèré seulement 5% du volume de logs tout en maintenant la même informativité, et aide à diagnostiquer 68% des défaillances analysées. En reproduisant la méthodologie de Yuan *et al.* (2012b), Zhao *et al.* (2017) observent que 42% des instructions de log sont modifiées après leur introduction, avec un accent particulier sur les changements de niveau de sévérité pour ajuster l'informativité par rapport à l'impact sur les performances.

En s'appuyant sur des solutions automatisées, Li *et al.* (2017a) proposent une méthode pour recommander automatiquement des niveaux de sévérité appropriés pour de nouvelles instructions de log. Leur approche analyse des caractéristiques contextuelles telles que le *churn* des logs, c'est-à-dire le nombre d'instructions de log modifiées dans une révision. La raison derrière cette caractéristique est que lorsque de nombreuses instructions de log sont ajoutées dans une révision, celles-ci ont tendance à enregistrer des informations détaillées sur l'exécution et à être associées à des niveaux de sévérité plus verbeux (par exemple, les

42. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

développeurs sont plus susceptibles d'ajouter un grand nombre des instructions *Debug* ou *Trace* plutôt que des erreurs graves).

Les auteurs analysent quatre projets *open-source* (Hadoop, Directory Server, Hama [43] et Qpid), en utilisant des modèles de régression ordinale pour prédire les niveaux de sévérité des nouvelles instructions de log. En se basant sur l'analyse de l'historique des révisions de code, les auteurs ont extrait des métriques logicielles et des caractéristiques des blocs de code, construisant ainsi des modèles capables de suggérer le niveau de sevérité.

La performance de ces modèles est évaluée à l'aide de deux métriques principales. La première est l'AUC (*Area Under the Curve*, ou aire sous la courbe), une mesure utilisée pour évaluer la capacité d'un modèle à discriminer entre les classes : plus l'AUC est élevée (proche de 1), meilleure est la performance du modèle (Myerson *et al.*, 2001). La deuxième est le score de Brier, qui mesure la précision des prédictions probabilistes : il correspond à la moyenne des carrés des écarts entre les probabilités prédites et les valeurs observées ; plus ce score est faible, meilleure est la calibration (Wilks, 2011). Dans cette étude, les modèles obtiennent une AUC comprise entre 0,75 et 0,81, et un score de Brier variant de 0,44 à 0,66.

Les résultats indiquent que les développeurs ajustent souvent les niveaux de sévérité entre niveaux adjacents (par exemple, d'*Info* à *Debug*), soulignant que ces ajustements visent à améliorer la granularité des logs sans entraîner de surcharge. L'outil proposé pourrait permetre ainsi d'éviter les erreurs communes, telles que la surestimation ou la sous-estimation de la criticité des événements, tout en fournissant des recommandations pertinentes pour chaque situation.

Anu *et al.* (2019) conduisent une étude quantitative sur quatre projets Apache (Hadoop, Tomcat, Qpid et ApacheDS) pour proposer une approche automatique d'aide à la décision sur

43. https://hama.apache.org/

les niveaux de sévérité des logs. Ils constatent que la plupart des instructions de log (entre 66,36% et 92,47%) se concentrent dans les blocs de gestion des exceptions et de vérification des conditions, en raison de l'imprévisibilité de ces blocs.

En constatant des incohérences dans l'attribution des niveaux *Info*, *Warn* et *Error*, les auteurs proposent l'outil VerbosityLevelDirector pour aider les développeurs à sélectionner les niveaux appropriés. L'approche identifie les blocs de code pertinents et extrait des caractéristiques contextuelles telles que les méthodes déclenchées, le type d'exception capturé et le contenu des messages de log.

Pour prédire le niveau de sévérité, l'outil utilise un modèle de type *random forest* (Breiman, 2001), un ensemble d'arbres de décision utilisé pour la classification, reconnu pour sa robustesse face au surapprentissage. Cependant, les auteurs observent également un problème de déséquilibre de classes dans les données recueillies, par exemple, avec des instances journalisées aux niveaux *Error* et *Warn* beaucoup plus fréquentes que celles au niveau *Info*. Afin d'évaluer plus justement les performances du modèle dans ce contexte déséquilibré, ils utilisent l'AUC. Enfin, pour corriger automatiquement les incohérences observées dans les pratiques de journalisation, l'outil prédit de nouveaux niveaux de sévérité.

Kim *et al.* (2020) proposent une approche automatique pour valider l'adéquation des niveaux de sévérité des logs et recommander des alternatives appropriées lorsque les niveaux d'origine sont jugés inadéquats. Cette approche s'appuie sur l'analyse des instructions de log en utilisant des vecteurs de caractéristiques sémantiques et syntaxiques.

Les vecteurs sémantiques mesurent la relation entre les mots des messages de journalisation et ceux associés à chaque niveau de sévérité, à l'aide du modèle *Word2Vec* (Ma & Zhang, 2015), un modèle de plongement lexical qui encode les mots en vecteurs selon leur contexte sémantique. Les vecteurs syntaxiques capturent des informations structurelles du

code, telles que la classe, la méthode et l'instruction de log elle-même. Ces deux dimensions sont combinées pour former des vecteurs représentant à la fois le contenu sémantique et la structure environnante des logs.

L'approche se déroule en quatre étapes principales : (1) la création d'un modèle de domaine basé sur les messages de logs d'un ensemble de projets *open-source* ; (2) la génération de vecteurs sémantiques ; (3) la construction de vecteurs syntaxiques ; et (4) l'entraînement de modèles d'apprentissage automatique, comme *K-nearest neighbor* (KNN) (Kramer, 2013), qui prédit une valeur en se basant sur les éléments les plus proches dans l'espace des caractéristiques, et *random forest*, pour valider et recommander des niveaux de sévérité appropriés. L'étude constate une amélioration de 6% de la précision par rapport à des développeurs expérimentés, avec 72% des recommandations de l'approche acceptées par les développeurs.

Gholamian & Ward (2020) proposent LACC (*Log-Aware Code-Clone Detector*), un outil de détection de clones de code visant à automatiser la prédiction de l'emplacement des instructions de log en se basant sur des paires de clones. L'étude utilise l'apprentissage profond (*deep learning* ou DL), une technique d'apprentissage automatique utilisant des réseaux de neurones profonds pour apprendre des représentations à partir de grandes quantités de données (Mathew *et al.*, 2021), pour détecter des clones de code dans des projets *open-source* Java, dans le but de prévoir l'ajout d'instructions de log dans des extraits de code similaires. L'approche atteint une précision supérieure à 90% pour prédire l'emplacement des instructions de log dans des projets Java Apache. De plus, le LACC est capable de prédire les détails des instructions de log ainsi que le niveau de sévérité associé.

Poursuivant cette exploitation de l'apprentissage profond dans le domaine de la journalisation, Li *et al.* (2021c) présentent une approche appelée DeepLV pour suggérer des niveaux de log appropriés, en exploitant la nature ordinale des niveaux de sévérité des logs. L'étude

est motivée par l'observation que le contexte syntaxique des instructions de log ainsi que les messages à enregistrer peuvent influencer le choix des niveaux. Les auteurs mènent une étude manuelle préliminaire sur neuf projets *open-source* - Cassandra, ElasticSearch, Flink, HBase, JMeter, Kafka, Karaf [44], Wicket [45] et Zookeeper - et analysent des rapports d'incidents récents, constatant qu'une part significative de ces rapports (42,5%) implique des discussions sur des ajustements ou des ajouts de niveaux de log, ce qui renforce l'importance de prendre des décisions éclairées sur la sévérité des logs.

Pour catégoriser les messages et la localisation des instructions de log, les auteurs utilisent les catégories dérivées des travaux de Fu *et al.* (2014) et He *et al.* (2018) comme base de leurs analyses. Ces catégories incluent les types d'emplacements des instructions de log (*e.g., blocs de capture d'exceptions, branches logiques, début ou fin de méthodes*) et les types de messages de log (*e.g., descriptions d'opérations, messages d'erreurs, et descriptions de variables ou de branches de code*). Sur cette base, ils proposent une approche d'apprentissage profond qui utilise des caractéristiques syntaxiques contextuelles et des messages extraits du code source pour suggérer le niveau de log le plus approprié.

L'approche est évaluée sur les neuf projets sélectionnés, et le modèle, en combinant des caractéristiques syntaxiques et de message, aboutit à un rendement avec une AUC moyenne de 83,7%. Ils observent également que les niveaux de log les plus éloignés en termes de sévérité ont tendance à présenter plus de différences dans les caractéristiques de localisation et de message, ce qui aide à distinguer les niveaux appropriés.

De plus, l'étude met en avant la capacité du modèle à obtenir de bons résultats même dans les suggestions entre différents systèmes, ce qui peut bénéficier aux systèmes sans longue

44. https://karaf.apache.org/

45. https://wicket.apache.org/

histoire de développement. L'utilisation de données d'autres systèmes lors de l'entraînement améliore également les performances de l'approche.

Ouatiti *et al.* (2022) conduisent une étude empirique sur la prédiction des niveaux de log dans les systèmes multi-composants, en investiguant l'efficacité des modèles d'apprentissage automatique globaux et locaux dans le contexte de projets tels que OpenStack [46], Hadoop, Jupyter [47] et ElasticSearch. Les auteurs constatent que les modèles globaux, entraînés avec les données de tous les composants d'un système, présentent des performances inférieures lorsqu'ils sont appliqués à des composants individuels. En revanche, les modèles locaux, spécifiquement entraînés sur des composants individuels, surpassent les modèles globaux dans jusqu'à 77% des composants analysés. De plus, ils observent que l'interprétation des modèles globaux peut être trompeuse, car les caractéristiques les plus importantes pour décider du niveau de journalisation varient de manière significative entre les composants. Pour les composants avec peu de données, les auteurs suggèrent l'utilisation de modèles locaux de composants similaires (*peer-local models*), qui ont démontré de meilleures performances que les modèles globaux.

Tang *et al.* (2022a) présentent une approche automatisée pour aider les développeurs à ajuster les niveaux de sévérité des instructions de log associées aux fonctionnalités du logiciel, en utilisant l'historique Git et un modèle de degré d'intérêt (*Degree of Interest* - DOI). Cette approche identifie les zones du code qui nécessitent plus d'attention en fonction de la « pertinence » des fonctionnalités en évolution, ajustant automatiquement les niveaux de sévérité pour refléter l'intérêt des développeurs pour ces zones. Ils développent cette solution

46. https://www.openstack.org/

47. https://jupyter.org/

sous la forme d'un plug-in pour l'IDE Eclipse [48], intégrant Mylyn [49] et JGit [50], et évaluent son efficacité sur 18 projets Java.

Les résultats montrent que l'outil réussit à analyser 99,22% des instructions de log, augmentant la distribution des niveaux de log d'environ 20% et améliorant le ciblage des logs pertinents dans les contextes de correction de bugs dans 83% des cas. De plus, l'approche permet d'intégrer des requêtes de pull (patches) dans des projets *open-source* populaires, démontrant la faisabilité et l'applicabilité de la solution.

Cette approche se concentre principalement sur ce qu'ils appellent les *logs de fonctionnalités*, qui, à la différence des logs plus critiques (comme ceux associés aux erreurs ou à la sécurité), enregistrent des informations détaillées sur le comportement des fonctionnalités spécifiques du système. Ces logs sont utilisés pour surveiller le comportement des fonctionnalités en cours de développement ou déjà implémentées, fournissant des informations sur leur exécution et permettant des ajustements à mesure que l'utilisation et l'importance des fonctionnalités évoluent. L'outil applique une série de règles pour distinguer les différents types de logs et ajuste automatiquement leurs niveaux de sévérité, réduisant ainsi la surcharge d'informations et mettant en avant les événements les plus pertinents pour le débogage et la maintenance.

Dans une étude connexe, Liu *et al.* (2022) proposent une approche pour suggérer des niveaux de sévérité des logs en utilisant à la fois des informations intra-bloc et inter-bloc des segments de code. Cette approche se distingue par l'intégration de ces deux niveaux d'informations dans une structure de graphe conjointe appelée *Flow of Abstract Syntax Tree* (FAST), qui cartographie l'arbre de syntaxe abstraite (AST) et le flux de contrôle inter-

48. https://eclipseide.org/
49. https://eclipse.dev/mylyn/
50. https://www.eclipse.org/jgit/

procédural (ICFG) des blocs de code. L'AST est une structure hiérarchique représentant la grammaire du code source (Baxter *et al.*, 1998), tandis que l'ICFG décrit les flux de contrôle entre les différentes fonctions d'un programme. Pour traiter cette structure, les auteurs présentent le *Hierarchical Block Graph Network* (HBGN), une architecture neuronale qui utilise des réseaux de neurones à base de graphes pour apprendre les représentations des blocs de code et ainsi suggérer les niveaux de sévérité appropriés.

L'approche est mise en œuvre dans l'outil TeLL (Liu *et al.*, 2022), et évalué sur neuf projets logiciels. TeLL obtient des résultats avec une précision de suggestion des niveaux de sévérité variant de 61,1% à 71,0%. L'analyse conjointe des informations intra et inter-bloc se révèle essentielle pour l'identification correcte des niveaux de sévérité.

Mastropaolo *et al.* (2022) introduisent LANCE (*Log Statement Recommender*), une approche automatisée conçue pour générer et injecter automatiquement des instructions de log complètes dans des méthodes Java, en s'appuyant sur le modèle d'apprentissage profond T5 — un transformeur entraîné pour des tâches de génération de texte conditionnelle, couramment utilisé dans les systèmes de recommandation (Vaswani *et al.*, 2017). LANCE prend en entrée une méthode Java et y insère une instruction de log comprenant le message, le niveau de sévérité approprié, ainsi que l'emplacement correct de l'instruction au sein de la méthode. Évalué sur un ensemble de 12 020 méthodes, LANCE prédit correctement l'emplacement de l'instruction de log dans 65,9% des cas et sélectionne le niveau de sévérité adéquat dans 66,2% des cas. Cependant, l'approche rencontre des difficultés à générer des messages de journalisation significatifs, atteignant un succès dans seulement 15,2% des cas, selon l'évaluation manuelle réalisée par les auteurs. Ce taux reflète la proportion de messages générés jugés adéquats et informatifs par des évaluateurs humains, en se basant sur des critères qualitatifs définis dans l'étude. La limitation la plus importante de LANCE réside dans son incapacité à déterminer si une méthode nécessite ou non des instructions de log, ainsi que

dans le fait qu'il insère toujours une seule instruction par méthode, limitant ainsi la flexibilité dans des scénarios réels.

Pour surmonter ces limitations, Mastropaolo *et al.* (2024) développent LEONID, une extension de LANCE. En plus de conserver les fonctionnalités d'insertion automatique de logs, LEONID est capable de distinguer les méthodes qui nécessitent ou non des instructions de log et peut insérer plusieurs instructions lorsque nécessaire. Le modèle amélioré est entraîné sur un ensemble de données beaucoup plus grand, comprenant 230 000 instances, ce qui permet un gain de 100% dans la génération de messages significatifs par rapport à LANCE. Cependant, la combinaison des techniques d'apprentissage profond avec la récupération d'informations pour améliorer la qualité des messages de log n'entraîne qu'un gain marginal (+5%). LEONID atteint une précision supérieure à 90% dans la prédiction des méthodes nécessitant des instructions de log et réussit à insérer plusieurs instructions correctement dans 17% des cas.

Xu *et al.* (2024) proposent UniLog, un cadre de journalisation automatique basé sur le paradigme d'apprentissage contextuel des grands modèles de langage (Large Language Models - LLM). UniLog vise à offrir une solution de journalisation capable de déterminer les positions de journalisation, de prédire les niveaux de sévérité et de générer des messages de journalisation sans nécessiter d'ajustements importants du modèle. Évalué sur 12 012 extraits de code provenant de 1 465 dépôts GitHub, UniLog atteint une précision de 76,9% dans la sélection des positions de journalisation, 72,3% dans la prédiction des niveaux de sévérité, et un score BLEU-4 de 27,1 pour la génération de messages de journalisation, ce qui reflète une similarité lexicale modérée entre les textes générés et les références humaines. Le score BLEU-4 (*Bilingual Evaluation Understudy*) est une métrique automatique utilisée en traitement du langage naturel pour évaluer la similarité textuelle entre une séquence générée et des références humaines. Il s'appuie sur la présence de séquences de mots similaires (jusqu'à

TABLEAU 2.2 : Approches automatisées.
© Eduardo Mendes de Oliveira, 2024

Catégorie	Description	Étude
Automatisation du niveau de sévérité	Li *et al.* (2017a)	Recommandation des niveaux de sévérité pour de nouvelles instructions de log.
	Anu *et al.* (2019)	VerbosityLevelChecker, une approche qui utilise le *Random Forest* pour predit le niveau de sévérité.
	Kim *et al.* (2020)	Une approche qui applique des caractéristiques sémantiques et syntaxiques et recommande un nouveau niveau de sévérité si le niveau actuel est inapproprié.
	Li *et al.* (2021c)	DeepLV, une approche d'apprentissage profond pour la prédiction des niveaux de journalisation avec une couche de sortie ordinale.
	Tang *et al.* (2022b)	Une approche pour ajuster les niveaux de sévérité des logs en utilisant l'historique Git et un modèle de degré d'intérêt (DOI).
	Liu *et al.* (2022)	TeLL suggère des niveaux de sévérité en utilisant à la fois des informations intra-bloc et inter-bloc des segments de code.
Automatisation de l'instruction de log	Zhao *et al.* (2017)	Log20, un outil automatisé qui optimise la placement des instructions de journalisation en se basant sur la théorie de l'information.
	Gholamian & Ward (2020)	LACC, un outil pour prédire l'emplacement des instructions de log en se basant sur des paires de clones de code.
	Mastropaolo *et al.* (2022)	LANCE, une approche d'apprentissage profond conçue pour générer et injecter des instructions de log complètes dans des méthodes Java.
	Mastropaolo *et al.* (2024)	LEONID, une extension de LANCE qui distingue les méthodes qui nécessitent ou non des instructions de log.
	Xu *et al.* (2024)	UniLog, un cadre de journalisation automatique basé sur un LLM.
Étude empirique	Ouatiti *et al.* (2022)	Une étude empirique qui examine l'efficacité des modèles dans la prédiction des niveaux de journalisation dans les systèmes multi-composants.

quatre mots consécutifs — 4-grammes), et bien qu'il n'évalue pas la cohérence sémantique, il permet une comparaison lexicale standardisée des messages générés (Papineni *et al.*, 2002).

2.2.2 DISCUSSION ET MOTIVATIONS DU PRÉSENT MANUSCRIT

Dans cette section, nous avons présenté six travaux spécifiques axés sur la suggestion ou l'ajustement automatique des niveaux de sévérité des logs, cinq travaux qui présentent des outils pour la génération d'instructions de log, ainsi qu'un travail qui examine les approches existantes pour la prédiction des niveaux de sévérité (Tableau 2.2). Ces études reflètent la difficulté persistante à trouver un équilibre approprié entre les niveaux de sévérité.

Malgré que ces travaux présentent d'excellents résultats en matière d'automatisation, nous pouvons souligner certaines limitations. Par exemple, Zhao *et al.* (2017) ne prennent pas en compte les préoccupations et les pratiques des développeurs, ni n'expliquent le contenu

statique des instructions de log. De plus, des changements dans les conditions d'exécution ou le comportement du système peuvent entraîner une surcharge supplémentaire de logs. Dans le cas de Gholamian & Ward (2020), l'approche dépend de l'existence de paires de clones dans le code, limitant ainsi l'applicabilité uniquement aux extraits de code ayant des clones correspondants dans la base de code. De plus, Kim *et al.* (2020) souligne que, dans de nombreux cas, l'adéquation des niveaux de log peut dépendre des opinions des développeurs, ce qui en fait un point discutable et subjectif.

En ce qui concerne les approches qui automatisent la suggestion de niveaux de sévérité, telles que celle proposée par Li *et al.* (2017a), elles facilitent la prédiction des niveaux, mais laissent une lacune importante dans la compréhension des raisons pour lesquelles certains niveaux sont plus appropriés dans des contextes spécifiques. En d'autres termes, bien que ces outils aident à prédire les niveaux de log, ils ne fournissent pas toujours de directives claires pour faire des choix éclairés en fonction des besoins du système et des contextes de développement.

Par ailleurs, Anu *et al.* (2019) proposent une approche basée sur les intentions de journalisation, mais il y a un manque d'explication détaillée sur la manière dont les intentions de journalisation ont été étudiées et intégrées dans l'analyse. Cette absence de clarté sur les pratiques et les décisions des développeurs peut limiter la compréhension des raisons sous-jacentes aux choix des niveaux de sévérité. Cette lacune souligne la nécessité d'une approche plus détaillée pour saisir les intentions derrière les décisions de journalisation, un aspect que notre recherche vise à approfondir.

Le travail de Li *et al.* (2021c) représente une avancée dans les catégories des niveaux de sévérité, en décrivant la présence et le pourcentage de chaque niveau par catégorie, ce qui est essentiel pour utiliser cette caractéristique dans l'alimentation de son modèle d'apprentissage

profond. Cependant, nous avons également observé qu'il manque une explication concernant la proximité fonctionnelle des niveaux de sévérité dans cette catégorie, par exemple, dans la catégorie « descriptions d'opérations ». En particulier, bien que la majorité des messages de log de niveau *Info* (70,0%) appartiennent à cette catégorie, des niveaux tels que *Trace* (61,5%), *Debug* (47,7%) et même *Warn* (30,3%) sont également utilisés pour décrire des opérations. L'étude ne clarifie pas pourquoi ces niveaux de sévérité, habituellement associés à différentes granularités ou degrés d'urgences, sont tous utilisés pour décrire des opérations. Cette absence d'explication sur la proportion similaire de certains niveaux (*Trace*, *Debug* et *Warn*) pour les descriptions d'opérations peut soulever des questions. La proposition d'une meilleure correspondance entre les niveaux de sévérité et le type de message enregistré, que nous présenterons, pourrait indiquer une corrélation plus cohérente.

En outre, des travaux récents visent à surmonter certaines de ces limitations. Mastropaolo *et al.* (2024) avec LEONID permet d'insérer plusieurs instructions de log et de prédire des messages de journalisation plus précis, tandis que Xu *et al.* (2024) utilise des modèles LLM et l'apprentissage en contexte pour proposer des solutions *end-to-end* dans la génération des logs, avec des résultats prometteurs.

Bien que ces études se concentrent sur le développement d'outils automatisés pour recommander, valider ou corriger les niveaux de sévérité des logs, elles ne fournissent pas toujours aux développeurs des directives claires sur la manière de décider du niveau de log à utiliser. Par ailleurs, comme l'a montré le travail de Ouatiti *et al.* (2022), l'interprétation des modèles globaux peut être trompeuse, car les caractéristiques influençant la décision du niveau varient significativement entre les composants, ce qui limite l'application des solutions globales dans certains contextes.

Notre étude adoptera une approche différente, centrée sur les intentions des développeurs. Nous analyserons des rapports de problèmes pour comprendre les motivations derrière les ajustements de sévérité et en distillerons 24 heuristiques pratiques. Ces heuristiques combleront la lacune laissée par les approches purement automatisées, en fournissant aux développeurs des lignes directrices claires pour prendre des décisions éclairées sur les niveaux de sévérité dans des scénarios réels.

En élargissant le champ d'application au-delà des outils automatisés et en intégrant une variété de scénarios, notre étude visera à améliorer la cohérence et l'efficacité des pratiques de journalisation. Cette approche centrée sur l'humain complète les solutions automatisées existantes en offrant un cadre structuré qui aide les développeurs à prendre des décisions informées sur les niveaux de sévérité des logs dans des scénarios réels. De cette manière, nous espérons fournir aux développeurs non seulement des outils, mais également des connaissances pour améliorer la cohérence et l'efficacité de leurs pratiques de journalisation.

2.3 REMARQUES FINALES

En résumé, la revue de la littérature a mis en lumière les principales approches et défis liés à la gestion des niveaux de sévérité des logs dans les systèmes logiciels. D'une part, les pratiques de journalisation étudiées ont permis de mieux comprendre les ajustements que les développeurs effectuent au niveau des logs, mais elles n'ont pas toujours fourni de cadres pratiques pour guider ces décisions. D'autre part, les approches automatisées, bien qu'efficaces dans la prédiction ou la correction des niveaux de sévérité, ne répondent pas aux besoins des développeurs en termes de compréhension des motivations derrière ces ajustements. Notre recherche vise à combler ces lacunes en proposant une approche qui tient compte non seulement des aspects techniques de la journalisation, mais aussi des intentions humaines qui influencent ces choix.

CHAPITRE III

CARTOGRAPHIE MULTIVOCALE DES NIVEAUX DE SÉVÉRITÉ DES LOGS

Ce chapitre présente une version mise à jour d'une cartographie multivocale préliminaire sur les niveaux de sévérité des logs (Mendes & Petrillo, 2021), intégrant de nouvelles études et synthèses. Par « cartographie multivocale », nous entendons une cartographie qui combine des résultats issus de plusieurs types de sources — scientifiques, industrielles et issues de la pratique — afin d'obtenir une vision plus complète, nuancée et applicable des niveaux de sévérité dans les systèmes logiciels. Selon Wohlin *et al.* (2020), une cartographie multivocale (ou revue multivocale) vise à tirer profit de la complémentarité entre la littérature académique évaluée par les pairs et la littérature dite grise (*blogs*, forums, documentations, etc.), permettant ainsi une meilleure représentativité des problématiques réelles rencontrées dans l'industrie.

Cette cartographie trouve sa motivation dans le constat d'un manque de directives et de spécifications concernant les pratiques de journalisation (He *et al.*, 2018; Rong *et al.*, 2018; Anu *et al.*, 2019). Bien que certaines études se concentrent sur *« où journaliser »* (Zhao *et al.*, 2017; Fu *et al.*, 2014; Li *et al.*, 2020a), *« comment journaliser »* Chen & Jiang (2020), et *« quoi journaliser »* (Li *et al.*, 2017a), il existe une lacune notable dans l'analyse des niveaux de sévérité des journaux. Cela nous a conduits à poser la question suivante : *Quels sont les niveaux de sévérité des journaux ?*

Pour répondre à cette question, nous avons étudié l'état de l'art et la pratique des niveaux de sévérité des journaux, en examinant leurs nomenclatures, définitions et descriptions, et ce en utilisant trois sources différentes : (1) *la littérature évaluée par les pairs*, (2) *les bibliothèques de journalisation*, et (3) *le point de vue des praticiens*. Toutes les données sont disponibles dans notre paquet de reproductibilité (Mendes de Oliveira, 2024a).

Nos résultats montrent un alignement entre les définitions académiques et industrielles des niveaux de sévérité, signalant une concordance des pratiques à travers divers contextes. Nous avons identifié des redondances et des similarités sémantiques entre plusieurs niveaux, et, en analysant les différentes nomenclatures et définitions, nous avons observé une convergence vers quatre finalités principales : *débogage*, *informationnelle*, *avertissement* et *défaillance*. De plus, nous proposons des définitions synthétisées pour ces finalités ainsi que pour les six niveaux de sévérité qui reflètent l'état actuel des pratiques de journalisation.

L'objectif principal de ce chapitre est donc de fournir un cadre théorique et empirique solide pour améliorer la fiabilité des entrées de log. Ces résultats ont pour but d'aider les développeurs à adopter une nomenclature standardisée et encourager les créateurs de bibliothèques de journalisation à fournir des définitions précises et non ambiguës de leurs niveaux de sévérité.

Les principales contributions de ce chapitre sont :

- une cartographie de la littérature sur les niveaux de sévérité des journaux ;
- une cartographie des niveaux de sévérité dans les bibliothèques de journalisation ;
- un ensemble de définitions synthétisées pour six niveaux de sévérité des journaux et quatre finalités généraux pour les niveaux de sévérité.

Ce chapitre est organisé comme suit. La section 3.1 expose la cartographie multivocale des niveaux de sévérité des journaux, couvrant la littérature évaluée par les pairs. Ensuite, la section 3.2 et la section 3.3 présentent respectivement les bibliothèques de journalisation et le point de vue des praticiens. La section 3.4 synthétise les niveaux de sévérité des journaux. Enfin, la section 3.5 discute des principales conclusions, recommandations, et des menaces pour la validité. La section 3.7 conclut le chapitre.

3.1 CARTOGRAPHIE DE LA LITTÉRATURE

Dans le chapitre précédent, nous avons exploré les différentes facettes des pratiques de journalisation à travers la littérature actuelle, mettant en évidence les défis et les limites rencontrés par les développeurs en ce qui concerne l'utilisation des niveaux de sévérité. Cette revue a révélé que, bien que des progrès aient été réalisés, il subsiste un manque de clarté et de cohérence dans la manière dont ces niveaux sont définis et appliqués dans divers contextes.

Dans cette section, nous visons à cartographier les niveaux de sévérité identifiés dans la littérature. Nous cherchons également à comparer leurs définitions respectives et à analyser les divergences ou convergences entre elles.

3.1.1 MÉTHODOLOGIE

Nous avons effectué une recherche systématique en deux étapes afin d'identifier la littérature actuelle sur les niveaux de sévérité des logs, comme illustré dans la figure 3.1. À l'étape 1, nous avons adopté la recherche automatique comme stratégie de recherche, méthode la plus couramment utilisée pour identifier les études pertinentes pour une cartographie systématique, selon Keele (2007). Notre requête de recherche était la suivante :

```
(‘‘log level’’ OR ‘‘log severity’’ OR ‘‘logging level’’
OR ‘‘logging severity’’ OR (‘‘severity level’’ AND (logging OR log)).
```

En 2021, nous avons exécuté cette requête sur Scopus [51], une des bases de données majeure en informatique (Maplesden *et al.*, 2015) avec plus de 60 millions de dossiers (Zhang

51. https ://www.scopus.com

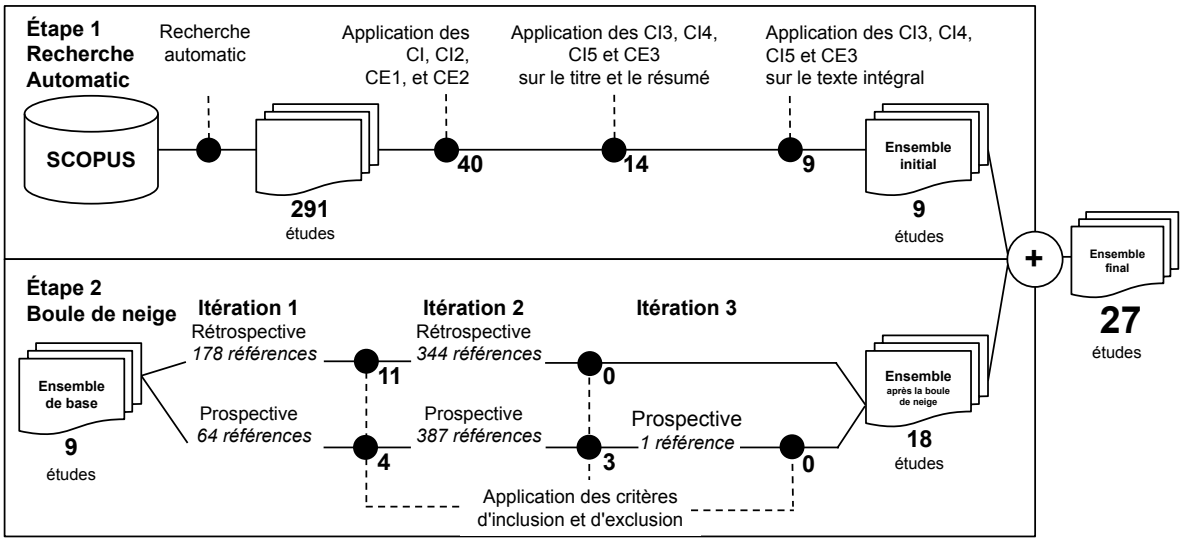

FIGURE 3.1 : Processus original de sélection des études réalisé en 2021. Adapté de Mendes & Petrillo (2021).

et al., 2011), en utilisant trois champs de métadonnées : titre, résumé et mots-clés. Nous avons trouvé **291 études**.

Ensuite, nous avons appliqué les critères d'inclusion (CI) et d'exclusion (CE), spécifiquement :

- **CI1 :** l'étude doit être un article de conférence ou un article de journal ;
- **CI2 :** l'étude doit appartenir au domaine de l'informatique ;
- **CI3 :** l'étude doit être une étude primaire ;
- **CI4 :** l'étude doit aborder les pratiques de journalisation ;
- **CI5 :** l'étude doit décrire l'utilisation des niveaux de sévérité des logs ou les définir ;
- **CE1 :** l'étude n'est pas rédigée en anglais ;
- **CE2 :** l'étude est un doublon ;
- **CE3 :** l'étude ne présente pas de lien entre les pratiques de journalisation et l'utilisation des niveaux de sévérité des logs.

Après avoir appliqué les CI1, CI2, CE1 et CE2, nous avons obtenu **40 études**. Nous avons lu le titre et le résumé de chacune d'elles et, après filtrage par CI3, CI4, CI5 et CE3, nous avons conservé **14 études**. Les 14 articles ont été lus en entier, résultant en un ensemble de base de **neuf études**, après application de CI3, CI4, CI5 et CE3.

À l'étape 2, nous avons utilisé cet ensemble initial comme base pour réaliser trois itérations de boule de neige (*snowballing*), en arrière et en avant. La boule de neige est une méthode de recherche qui consiste à partir d'un ensemble initial d'études pertinentes pour analyser soit leurs références (boule de neige en arrière), soit les articles qui les citent (boule de neige en avant), dans le but d'identifier d'autres études potentiellement pertinentes (Wohlin, 2014). Les itérations de boule de neige ont permis de récupérer 18 études supplémentaires, résultant en un ensemble initial de **27 études**.

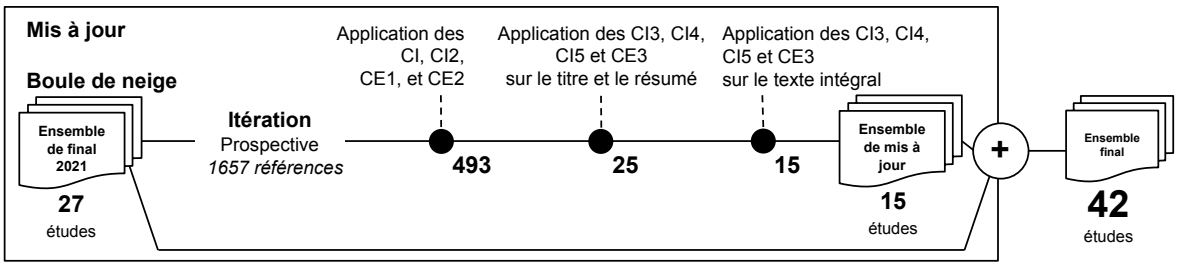

FIGURE 3.2 : Mise à jour du processus de sélection des études réalisée en 2024.
© Eduardo Mendes de Oliveira, 2024

MISE À JOUR DES ÉTUDES SÉLECTIONNÉES

Pour mettre à jour les résultats de la recherche, nous avons appliqué la méthode boule de neige en avant sur les 27 études qui composaient le travail initial. Nous avons opté pour cette stratégie en raison de son adéquation et de son efficacité prouvées (Wohlin *et al.*, 2020).

TABLEAU 3.1 : Liste des études de la littérature.
© Eduardo Mendes de Oliveira, 2024

#	Référence	Titre	Année	Origine
[P01]	Oliner *et al.* (2012)	Advances and challenges in log analysis	2012	Original
[P02]	Yuan *et al.* (2012b)	Characterizing logging practices in open-source software		
[P03]	Gomathy *et al.* (2014)	Developing an error logging framework for ruby on rails application using AOP	2014	
[P04]	Fu *et al.* (2014)	Where do developers log ? An empirical study on logging practices in industry		
[P05]	Shang *et al.* (2015)	Studying the relationship between logging characteristics and the code quality of platform software	2015	
[P06]	Lin *et al.* (2016)	Log clustering based problem identification for online service systems	2016	
[P07]	Li *et al.* (2017a)	Which log level should developers choose for a new logging statement ?	2017	
[P08]	Chen & Jiang (2017a)	Characterizing and detecting anti-patterns in the logging code		
[P09]	Chen & Jiang (2017b)	Characterizing logging practices in Java-based open source software projects – a replication study in Apache Software Foundation		
[P10]	Zhao *et al.* (2017)	Log20 : Fully automated optimal placement of log printing statements under specified overhead threshold		
[P11]	Li *et al.* (2017b)	Towards just-in-time suggestions for log changes		
[P12]	Rong *et al.* (2018)	How is logging practice implemented in open source software projects ? A preliminary exploration	2018	
[P13]	He *et al.* (2018)	Characterizing the natural language descriptions in software logging statements		
[P14]	Hassani *et al.* (2018)	Studying and detecting log-related issues		
[P15]	Chowdhury *et al.* (2018)	An exploratory study on assessing the energy impact of logging on android applications		
[P16]	Kabinna *et al.* (2018)	Examining the stability of logging statements		
[P17]	Yuan *et al.* (2019)	An approach to cloud execution failure diagnosis based on exception logs in Openstack	2019	
[P18]	Anu *et al.* (2019)	An approach to recommendation of verbosity log levels based on logging intention		
[P19]	Zeng *et al.* (2019)	Studying the characteristics of logging practices in mobile apps : a case study on F-Droid		
[P20]	Li *et al.* (2019)	DLFinder : Characterizing and detecting duplicate logging code smells		
[P21]	Chen & Jiang (2019)	Extracting and studying the Logging-Code-Issue-Introducing changes in Java-based large-scale open source software systems		
[P22]	Li *et al.* (2020a)	A Qualitative study of the benefits and costs of logging from developers perspectives	2020	
[P23]	Kim *et al.* (2020)	Automatic recommendation to appropriate log levels		
[P24]	Bharkad & Chavan (2020)	Optimizing root cause analysis time using smart logging framework for Unix and GNU/Linux based operating system		
[P25]	Obrebski & Sosnowski (2019)	Log based analysis of software application operation		
[P26]	Gholamian & Ward (2020)	Logging statements' prediction based on source code clones		
[P27]	Li *et al.* (2020b)	Where shall we log ? Studying and suggesting logging locations in code blocks		
[P28]	Li *et al.* (2021a)	Studying the practices of logging exception stack traces in open-source software projects	2021	Mise à jour
[P29]	Li *et al.* (2021b)	Studying duplicate logging statements and their relationships with code clones		
[P30]	Li *et al.* (2021c)	Deeplv : Suggesting log levels using ordinal based neural networks		
[P31]	Alves & Paula (2021)	Identifying Logging Practices in Open Source Python Containerized Application Projects		
[P32]	Gholamian (2021)	Leveraging code clones and natural language processing for log statement prediction		
[P33]	Ouatiti *et al.* (2022)	An empirical study on log level prediction for multi-component systems	2022	
[P34]	Patel *et al.* (2022)	The sense of logging in the Linux kernel		
[P35]	Zhang *et al.* (2022)	Studying logging practice in test code		
[P36]	Mastropaolo *et al.* (2022)	Using deep learning to generate complete log statements		
[P37]	Bogatinovski *et al.* (2022)	Qulog : Data-driven approach for log instruction quality assessment		
[P38]	Tang *et al.* (2022b)	A tool for rejuvenating feature logging levels via git histories and degree of interest		
[P39]	Liu *et al.* (2022)	Tell : log level suggestions via modeling multi-level code block information		
[P40]	Tang *et al.* (2022a)	Automated evolution of feature logging statement levels using git histories and degree of interest		
[P41]	Mastropaolo *et al.* (2024)	Log statements generation via deep learning : Widening the support provided to developers	2024	
[P42]	Foalem *et al.* (2024)	Studying logging practice in machine learning-based applications		

Après avoir appliqué la méthode de l'effet boule de neige avant le 8 mars 2024 sur l'ensemble des 27 études sélectionnées en 2021, et en suivant les mêmes critères d'inclusion et d'exclusion, nous avons obtenu 15 nouvelles études pertinentes (Figure 3.2). Ainsi, la mise à jour a enrichi l'ensemble initial de 27 études avec 15 nouveaux travaux, portant le total à **42 études** pertinentes (Tableau 3.1).

3.1.2 RÉSULTATS

DISTRIBUTION DES ÉTUDES INCLUSES

Observation #1 : *La recherche sur les niveaux de sévérité des logs a augmenté ces dernières années.*

Le plus grand nombre de publications incluant les niveaux de sévérité des logs a été publié au cours des sept dernières années : cinq études en 2017, cinq études en 2018, cinq études en 2019, six études en 2020, cinq études en 2021 et huit études en 2022 (Figure 3.3). Ces 34 études représentent environ 81% de nos études incluses et montrent un intérêt croissant pour le sujet.

La plupart des études incluses traitent des niveaux de sévérité en général, les présentant pour illustrer et clarifier les processus de journalisation dans leur ensemble. D'autres études abordent les niveaux de sévérité comme un aspect de leur recherche sur les pratiques de journalisation [P02, P09, P12, P19], ou dans l'étude des instructions de logs [P13, P16, P26]. Certaines études traitent de problèmes spécifiques liés aux niveaux de sévérité, où journaliser [P04, P10], quel niveau de sévérité choisir [P07], et la recommandation automatique des niveaux de sévérité [P23, P27].

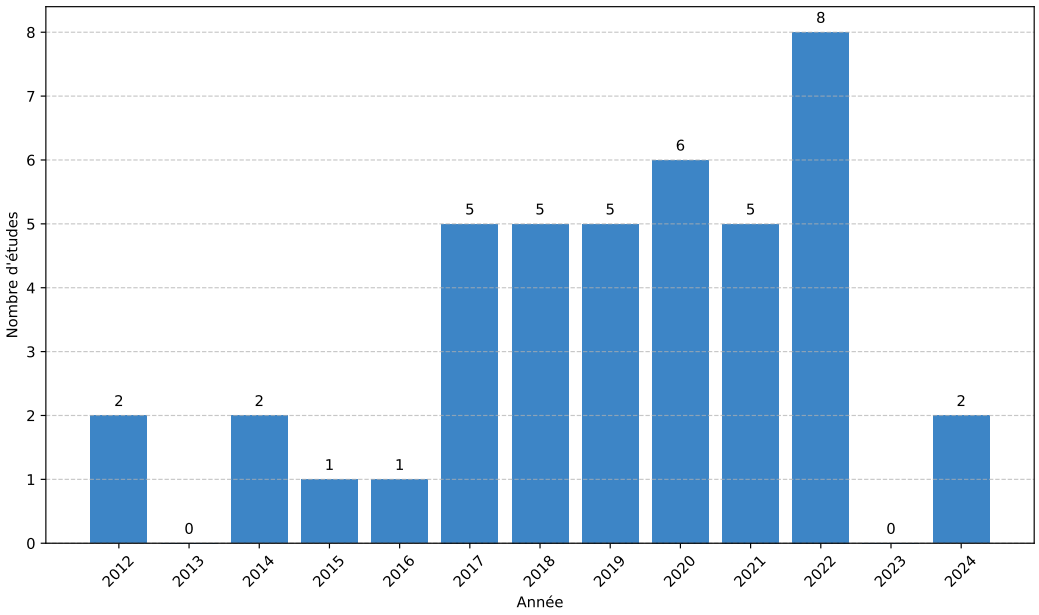

FIGURE 3.3 : Nombre d'études sur les niveaux de sévérité des logs par année.
© Eduardo Mendes de Oliveira, 2024

PRÉSENCE DES NIVEAUX DE SÉVÉRITÉ DES LOGS

> ***Observation #2 :*** *Error, Debug, Info, Warn, Fatal et Trace sont les niveaux qui se démarquent dans la recherche sur les niveaux de sévérité des logs.*

À l'exception de l'étude P32, toutes les études discriminent un ensemble de niveaux de sévérité des logs (par exemple, *Debug*, *Info*, *Warn*, etc.) ; 37 études (88%) mentionnent au moins quatre niveaux de sévérité. En revanche, seulement neuf études (21%) proposent des définitions ou des descriptions pour les niveaux de sévérité. Comme le montre la figure 3.4, il est possible de distinguer deux groupes de niveaux de sévérité : les plus mentionnés et les moins mentionnés. Le premier groupe, composé des niveaux *Error* (38), *Debug* (37), *Info* (36), *Warn* (34), *Fatal* (27) et *Trace* (23), représente 88% des mentions, tandis que le second groupe, formé par les niveaux *Critical*, *Notice*, *Verbose*, *Finer*, *Finest*, *Alert*, *Exception*, *Fine*, *Severe*,

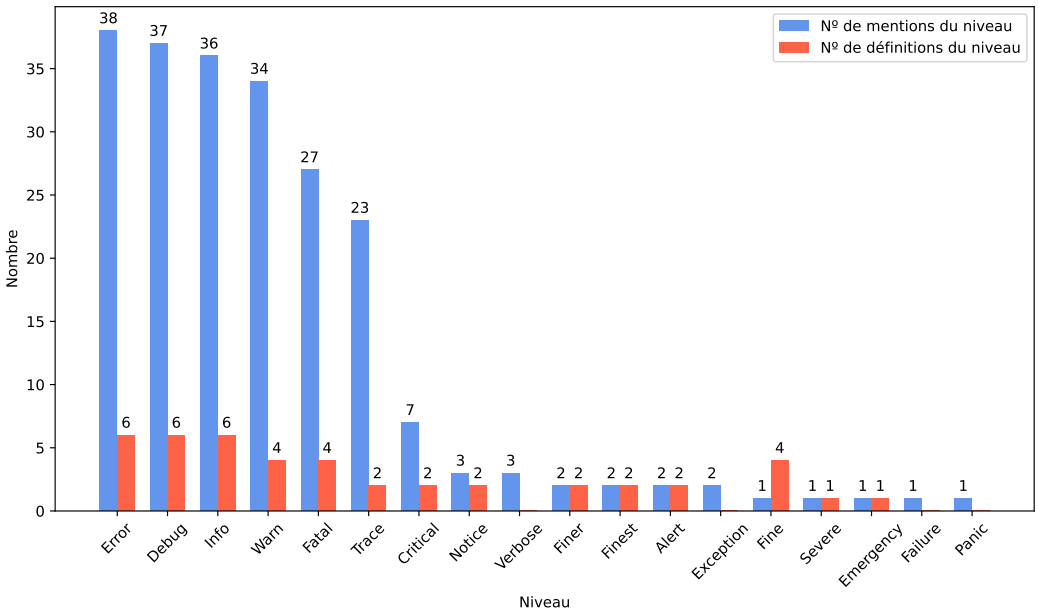

FIGURE 3.4 : Nombre de mentions vs nombre de définitions sur la littérature.
© Eduardo Mendes de Oliveira, 2024

Emergency, *Failure*, et *Panic*, constitue les 12% restants. Le groupe le plus représentatif en termes de mentions est également celui qui contient le plus grand nombre de définitions.

CATÉGORISATION DES NIVEAUX DE SÉVÉRITÉ DE LOG

Observation #3 : *Les études cherchent à catégoriser les niveaux de sévérité ou les éléments des instructions de journalisation.*

Certaines études proposent des catégorisations liées aux instructions de logs. He *et al.* (2018) regroupent les *descriptions de journalisation* [52] en trois catégories principales : (i) *« description de l'opération du programme »*, (ii) *« description de la condition d'erreur »*, et (iii) *« description de la sémantique de code de haut niveau »*. Les descriptions de la première catégorie semblent liées au niveau de sévérité *Info*, décrivant trois types d'opérations :

52. *« (...) la partie textuelle d'une instruction de log, à l'exclusion des variables »* (He *et al.*, 2018)

opération complète, opération en cours et prochaine opération. La seconde catégorie décrit l'occurrence d'une erreur/exception ; les niveaux de sévérité liés à cette catégorie sont *Error* et *Info*. Dans la troisième catégorie, les descriptions de journalisation décrivent essentiellement le code, par exemple, les variables, les fonctions et les branches, telles que les blocs *« if-else »* ; tous les exemples de cette catégorie utilisent le niveau *Debug*.

Yuan *et al.* (2012b) commentent deux classes de niveaux : *« niveau d'erreur (par exemple, erreur, fatal) (...) et non-erreur (également niveaux non fatals), tels que info et debug »*. De la même manière, Shang *et al.* (2015) commentent deux autres classes en considérant le niveau moyen de journalisation [53] :

> *« Intuitivement, les logs de haut niveau sont destinés aux opérateurs de systèmes et les logs de bas niveau sont destinés au développement. (...) Les logs de niveau supérieur sont typiquement utilisés par les administrateurs et les logs de niveau inférieur sont utilisés par les développeurs et les testeurs. »* (Shang *et al.*, 2015)

Comme exploré dans le chapitre de revue de la littérature, d'autres auteurs identifient des catégories de journalisation ; cependant, elles ne sont pas directement liées à la catégorisation des niveaux de sévérité, mais plutôt à la classification des raisons de journaliser (Zeng *et al.*, 2019), des motivations de changements dans les logs (Li *et al.*, 2017b), à la typologie des messages (He *et al.*, 2018), aux anti-patrons (Chen & Jiang, 2017a), ainsi qu'aux coûts et avantages de la journalisation (Li *et al.*, 2020a).

---

53. Le « *niveau moyen de journalisation* »est une métrique calculée à partir de la transformation de chaque niveau de sévérité des logs en mesures quantitatives (Shang *et al.*, 2015).

DÉFINITIONS DES NIVEAUX DE SÉVÉRITÉ DE LOG

Observation #4 : *Seulement 12% des études sélectionnées ont un ensemble de quatre définitions ou plus pour les niveaux de sévérité des logs.*

Le tableau 3.2 présente les définitions et descriptions trouvées pour les niveaux de sévérité des logs. Cinq études [P02, P12, P14, P24, P34] ont des définitions pour un ensemble fermé de niveaux (quatre niveaux ou plus). Cependant, certaines de ces descriptions peuvent être considérées comme lacunaires ou incomplètes, ne fournissant pas toujours un niveau de détail suffisant pour distinguer clairement les conditions spécifiques associées à chaque niveau. Par exemple, des descriptions telles que « *System is unusable* » ou « *Informational* » manquent parfois de précision ou de contextualisation dans certaines situations. Les quatre autres présentent des descriptions pour seulement un ou deux niveaux afin de contextualiser l'idée de niveau de sévérité des logs [P20, P23, P25, P27].

Le tableau 3.2 regroupe les définitions des niveaux de sévérité dans la littérature. Ci-dessous, nous présentons nos observations sur ceux qui ont été définis à plusieurs reprises dans les études sélectionnées :

(a) *Debug :* Comme le décrit son nom, quatre des définitions associent le niveau *Debug* aux tâches de débogage [P02, P12, P14, P24, P34]. Sa phase cible du processus logiciel est le développement, par conséquent principalement utilisé par les développeurs [P23, P24]. Le niveau *Debug* semble lié à des expressions telles que *« verbeux », « information détaillée », « détails des événements », « utile pour les développeurs »*. Kim *et al.* (2020) le décrivent comme *« largement utilisé pour désigner l'état de la variable. »*

(b) *Trace :* Le niveau *Trace* est décrit comme *« plus détaillé »* que le niveau *Debug* [P12, P14].

TABLEAU 3.2 : Définitions des niveaux de sévérité dans la littérature. Adapté de Mendes & Petrillo (2021).

Niveau	Papier	Citation
Trace	[P12]	« *Trace / Finest : This level designates finer-grained informational events than the 'Debug'* ».
	[P14]	« *and trace (tracing steps of the execution, most fine-grained information)* »
Debug	[P02]	« *debug (i.e., verbose logging only for debugging)* ».
	[P12]	« *Debug / Fine / Finer : This level designates fine-grained informational events that are most useful to debug an application* ».
	[P14]	« *debug (verbose logging only for debugging)* ».
	[P23]	« *Moreover, the log level debug is broadly used to designate the state of the variable during the development phase with the corresponding message* ».
	[P24]	« *Level 7 Debug : Messages at debug level contains more details of events, debug level log messages are more useful for developers and for debugging an application* ».
	[P34]	« *Debug-level messages* ».
Info	[P02]	« *info (i.e., record important but normal events)* ».
	[P12]	« *Info / Config : This level designates informational messages that highlight the progress of the application coarse-grained level* ».
	[P14]	« *info (record important but normal events)* »
	[P24]	« *Level 6 Information : Normal operation messages are at this level, no action is required to take* ».
	[P25]	« *Info level entries describe application operation, e.g. details of creating services* ».
	[P34]	« *Informational* ».
Notice	[P24]	« *Level 5 Notice : Unusual event is mentioned, but not, an error is shown* ».
	[P34]	« *Normal but significant condition* ».
Warn	[P12]	« *Warn / Warning : This level designates potentially harmful situations* ».
	[P24]	« *Level 4 Warning : Warning messages indicate that an error may occur if action is not taken* ».
	[P27]	« *The logging statement is at the warn level, which is the level for recording information that may potentially cause system oddities* ».
	[P34]	« *Warning conditions (default)* ».
Error	[P02]	« *error (i.e., record error events)* ».
	[P12]	« *Error / Severe : This level designates error events that might still allow the application to continue running* ».
	[P14]	« *error (record error events)* ».
	[P20]	« *The logging statement is at the error level, which is the level for recording failed operations* ».
	[P24]	« *Level 3 Error : Occurred error information is shown in this kind of log messages* ».
	[P34]	« *Error conditions* ».
Critical	[P24]	« *Level 2 Critical : Critical level messages, is written in the log file when a critical situation occurs in the normal execution of the system* ».
	[P34]	« *Critical conditions* ».
Alert	[P24]	« *Level 1 Alert : Alert level messages indicate that respective one should be corrected immediately* ».
	[P34]	« *An action must be taken immediately* ».
Fatal	[P02]	« *fatal (i.e., abort a process after logging)* ».
	[P12]	« *Fatal : This level designates very severe error events that will presumably lead the application to abort* ».
	[P14]	« *fatal (abort a process after logging)* ».
	[P23]	« *For example, the log level fatal is used to indicate that a critical problem has occurred around the position of the log statement, where the developer tries to leave an appropriate log message as a clue to treat it later.* ».
Emergency	[P34]	« *System is unusable* ».

*Les niveaux sont présentés de haut en bas, du moins sévère au plus sévère.

(c) *Info :* Les messages de niveau *Info* sont décrits comme *« des événements importants mais normaux »* [P02, P14, P24], utilisés pour mettre en évidence et décrire la progression de l'application [P12, P25] *« à un niveau global »,* dont les circonstances ne nécessitent pas d'action à prendre [P24].

(d) *Warn :* Contrairement au niveau *Info*, les définitions du niveau *Warn* le décrivent comme un niveau de sévérité qui nécessite une action [P24] car il désigne des situations potentiellement dangereuses [P12] capables de causer des problèmes au système [P27], ou pour signaler des incohérences das les opérations [P34].

(e) *Error :* Les définitions du niveau *Error* fournissent peu d'informations au-delà de leur objectif de journaliser les erreurs ou les opérations échouées [P02, P12, P14, P24, P34]. Cependant, pour [P12], le niveau *Error « désigne des événements d'erreur qui pourraient encore permettre à l'application de continuer à fonctionner. »*

(f) *Fatal :* Les expressions utilisées dans les définitions du niveau *Fatal* sont l'abandon des processus ou des applications [P02, P12, P14], des erreurs très graves [P12] et des problèmes critiques [P23].

3.2 CARTOGRAPHIE DES BIBLIOTHÈQUES DE JOURNALISATION

Après avoir cartographié les définitions des niveaux de sévérité dans la littérature, notre objectif dans cette section est d'explorer comment ces niveaux sont implémentés dans les bibliothèques de journalisation largement utilisées. Contrairement à la cartographie de la littérature, où nous avons trouvé des études traitant les niveaux de sévérité de manière fragmentée, nous avons constaté qu'il n'existe pas de cartographie exhaustive centralisant ces niveaux et leurs définitions dans les bibliothèques de journalisation.

Nous avons réalisé une cartographie systématique des bibliothèques les plus pertinentes, cherchant à identifier quels niveaux de sévérité sont appliqués, leurs définitions et comment

ils se comparent entre les différentes bibliothèques. Cette cartographie nous permet de mieux comprendre la diversité et la cohérence des niveaux de sévérité dans les outils de journalisation les plus populaires, contribuant ainsi à l'avancement de la compréhension de la manière dont ces niveaux peuvent être mieux définis et appliqués de manière standardisée. De plus, l'analyse comparative nous aide à identifier des modèles et des incohérences, offrant une vision plus claire de l'utilisation de ces niveaux dans différents contextes et plateformes.

3.2.1 MÉTHODOLOGIE

Nous utilisons l'indice PYLP [54], un classement des langages de programmation, comme point de départ pour la sélection des bibliothèques, en sélectionnant les langages avec une « part de marché »supérieure à 1,0%. Nous avons obtenu **16 langages**, donc nous avons pris ces langages et interrogé Google Search : « `logging library` », en concaténant le nom de chaque langue et utilisé la première page de résultats pour chaque requête. Nous avons trouvé **160 occurrences** (blogs, forums, dépôts de code), et parmi eux, nous avons cartographié **60 bibliothèques**. Nous inspectons les dépôts de code (quand ils sont disponibles), la documentation et les directives des bibliothèques pour appliquer nos critères d'inclusion (LCI) et d'exclusion (LCE) :

- **LCI1 :** la bibliothèque/langage a un ensemble de niveaux de sévérité des logs ;
- **LCE1 :** la bibliothèque ne crée pas des instructions de logs avec des niveaux de sévérité des logs ;
- **LCE2 :** la bibliothèque est sur Github et a moins de 1000 étoiles.

Après avoir appliqué les critères ci-dessus, nous avons obtenu **37 bibliothèques**. Nous avons manuellement ajouté Java Util Logging, en raison de son rôle historique comme solution

54. https ://pypl.github.io/PYPL.html

officielle pour le logging dans Java, un langage fréquemment étudié dans les travaux sur les niveaux de sévérité ; PHP Logging, pour représenter la grande utilisation de PHP malgré la rareté des références spécifiques de bibliothèques de logging ; et Syslog-ng, qui constitue une évolution importante du premier système de journalisation à avoir introduit des niveaux de sévérité. Notre ensemble de données final comprenait **40 bibliothèques** [55] (Tableau 3.3) et **63 documents** provenant du code source, de la documentation et des directives associés à ces bibliothèques.

3.2.2 RÉSULTATS

Le tableau 3.3 présente les 19 niveaux de sévérité identifiés dans 40 bibliothèques de journalisation couvrant 14 langages de programmation. Chaque colonne correspond à un niveau de sévérité, organisé de gauche à droite, du moins grave au plus grave, selon les valeurs numériques et les descriptions fournies par les bibliothèques. Les cellules indiquent la présence du niveau dans la bibliothèque concernée par sa valeur numérique ; en l'absence de cette valeur, une coche (✓) est affichée pour signaler la présence du niveau. Les niveaux considérés comme redondants ou non pertinents pour cette analyse, tels que *All, Off, Notset* ou des alias spécifiques (*e.g., Log4Net_Debug*), ont été exclus. Cette représentation permet de mettre en évidence les niveaux les plus couramment utilisés (par exemple, *Info*, *Warn*, *Error*, *Debug*), ainsi que les variations terminologiques (par exemple, *Warn* et *Warning*), facilitant ainsi l'identification des tendances et des incohérences dans les pratiques de journalisation.

55. Parmi les bibliothèques, trois apparaissent en deux versions (Log4J [L13], versions 1 et 2 ; Loguru versions C++[L27] et Python [L34] ; PHP Logging, versions Linux et Windows [L36])

LANGAGES DE PROGRAMMATION

Les bibliothèques incluses couvrent 14 des 16 langages de programmation sélectionnés. Le nombre le plus significatif de bibliothèques vient de C/C++ avec sept bibliothèques (20%), Java avec six bibliothèques (15%), JavaScript avec cinq bibliothèques (13%), et C# et Golang, tous les deux avec quatre bibliothèques (10%).

DISTRIBUTION DES NIVEAUX PAR BIBLIOTHÈQUE

Observation #5 : *Le nombre le plus bas de niveaux de sévérité dans les bibliothèques est de quatre, et le nombre le plus élevé est de 15.*

Parmi les bibliothèques sélectionnées, 91% ont entre cinq et huit niveaux de sévérité, 39,5% ont six niveaux, 23,3% ont cinq niveaux, 14% ont huit niveaux, et 14% ont sept niveaux. Les bibliothèques présentant le moins de niveaux, Google Glog [L01] et Golang Glog [L02], ont les quatre mêmes niveaux : *Info, Warn, Error*, et *Fatal*. Les bibliothèques avec plus de niveaux sont Log4C [L39] et Log4Net [L40], avec neuf et 15 niveaux de sévérité respectivement.

OCCURRENCE DES NIVEAUX

Observation #6 : *Six niveaux sont présents dans plus de 50% des bibliothèques, parmi eux quatre dans plus de 90% : Info, Warn, Error, Debug, Trace, et Fatal.*

En agrégeant les données du tableau 3.3, nous observons que six niveaux sont présents dans plus de 50% des bibliothèques, dont quatre niveaux sont présents dans plus de 90% : *Info* (100%), *Warn* (98%), *Error* (98%), *Debug* (93%), *Trace* (55%), et *Fatal* (52%).

Observation #7 : *Basic, Config, Emergency, Fault, Fine, Finest, Finer, Severe, Success et Verbose ont une faible occurrence dans les bibliothèques comparés aux autres niveaux de sévérité, égal ou inférieur à 10%.*

Quatre bibliothèques ont un niveau de sévérité unique à elles : *Fault* dans OSLogging [L03], *Config* dans Java Util Logging [L29], *Basic* dans Bolterauer [L32], et *Success* dans Loguru [L34]. Six autres niveaux sont présents seulement dans 10% ou moins des bibliothèques : *Verbose* (10%), *Emergency* (10%), *Finer* (10%), *Finest* (7%), *Fine* (5%) et *Severe* (5%).

NOMBRE DE NIVEAUX DE SÉVÉRITÉ AU FIL DU TEMPS

Observation #8 : *Il n'y a pas de tendance vers une diminution ou une augmentation dans le nombre de niveaux de sévérité des logs dans les bibliothèques.*

La figure 3.5 présente la médiane du nombre de niveaux des bibliothèques par année de publication. Il n'y a pas de variation dans le temps dans le nombre de niveaux de sévérité que les bibliothèques de logs ont fourni. Le seul point hors de la tendance est l'année 2004, qui présente les 15 niveaux de la bibliothèque Log4Net [L40]. Malgré ce nombre élevé, sa documentation informe qu'elle *« catégorise la journalisation en niveaux : DEBUG, INFO, WARN, ERROR et FATAL »* [L40].

LE NIVEAU LE PLUS GRAVE

Le niveau le plus grave dans 48% des bibliothèques est le niveau *Fatal*, suivi par le niveau *Error* (19%), *Critical* (14%), *Emergency* (10%), et *Alert* (5%). Les huit bibliothèques où le niveau *Error* est le plus grave ont cinq niveaux de sévérité des logs.

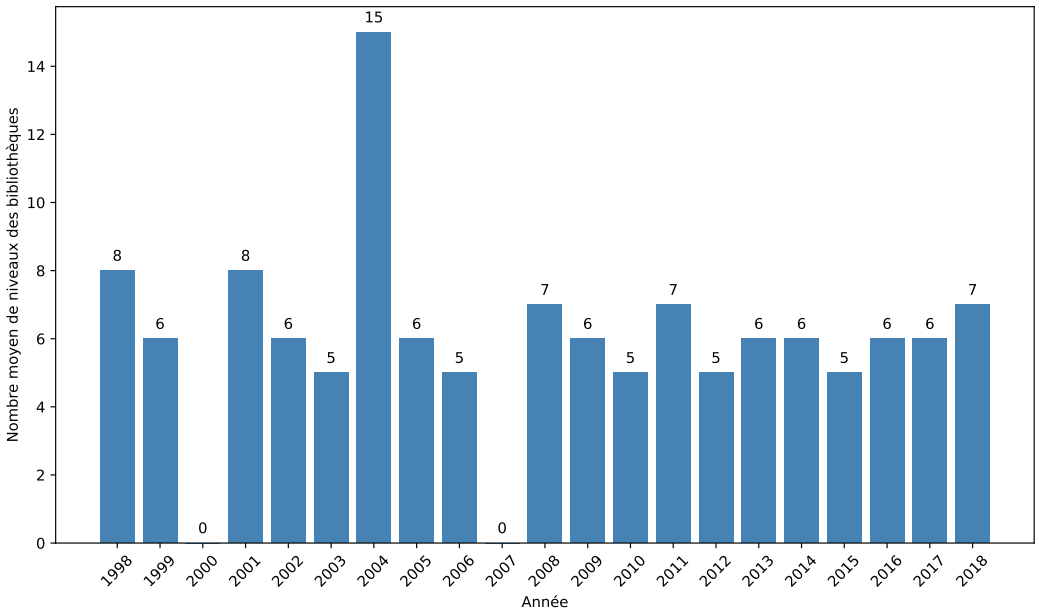

FIGURE 3.5 : Année de publication et nombre de niveaux (La valeur 0 indique l'absence de nouvelles bibliothèques dans l'année concernée). Adapté de Mendes & Petrillo (2021).

VALEURS NUMÉRIQUES

Observation #9 : *Il y a une cohérence dans l'ordre numérique des niveaux de sévérité à travers les bibliothèques de journalisation.*

Dans notre ensemble de données, 38 bibliothèques (95%) utilisent des valeurs numériques associées aux niveaux pour les classer selon leurs degrés spécifiques de sévérité (Tableau 3.3), comme l'indique la citation suivante :

> *« Les niveaux ont une valeur numérique qui définit l'ordre relatif entre les niveaux.»* [L40]

Le tableau 3.3 montre qu'il est possible d'organiser de manière équivalente les niveaux de sévérité de presque toutes les bibliothèques en considérant leurs valeurs numériques, à

TABLEAU 3.3 : Niveaux de sévérité dans les bibliothèques de journalisation. Adapté de Mendes & Petrillo (2021).

#	Bibliothèque	Publication	Nº	Finest	Verbose	Finer	Trace	Debug	Basic	Fine	Config	Info	Success	Notice	Warn	Error	Fault	Severe	Critical	Alert	Fatal	Emergency
[L01]	Google Glog[a]	2015	4									0			1	2					3	
[L02]	Golang Glog[b]	2013	4									0			1	2					3	
[L03]	OSLogging[c,i]	?	5					✓				✓		✓		✓	✓					
[L04]	Rust Lang[d]	2014	5				5	4				3			2	1						
[L05]	Logback[e]	2006	5				5000	10000				20000			30000	40000						
[L06]	SLF4J[e]	2006	5				0	10				20			30	40						
[L07]	Ruby Logger[f]	2003	5					0				1			2	3					4	
[L08]	LogLevel[g]	2013	5				0	1				2			3	4						
[L09]	JS-logger[g]	2012	5				1	2				3			5	8						
[L10]	CocoaLum.[c]	2010	5		4			3				2			1	0						
[L11]	Kotlin-logging[h]	2016	5				✓	✓				✓			✓	✓						
[L12]	SwiftyBeaver[i]	2015	5		0			1				2			3	4						
[L13]	Log4J[e]	1999 (v1) 2014 (v2)	6				5000 600	10000 500				20000 400			30000 300	40000 200					50000 100	
[L14]	Commons Logging[e]	2002	6				1	2				3			4	5					6	
[L15]	Bunyan[g]	2011	6				10	20				30			40	50					60	
[L16]	NLog[j]	2006	6				0	1				2			3	4					5	
[L17]	Python Lang[k]	2002	6					10				20			30	40			50		50	
[L18]	ME Logging	2015	6				0	1				2			3	4			5			
[L19]	Serilog	2013	6		0			1				2			3	4					5	
[L20]	Log4PHP	2009	6				500	10000				20000			30000	40000					50000	
[L21]	PinoJS[g]	2016	6				10	20				30			40	50					60	
[L22]	Log.c[a]	2017	6				0	1				2			3	4					5	
[L23]	C-Logger[a]	2016	6				0	1				2			3	4				.	5	
[L24]	Zlog[a]	2011	6					20				40		60	80	100					120	
[L25]	Go-logging[b]	2013	6					5				4		3	2	1			0			
[L26]	Log4m[l]	2012	6				1	2				3			4	5					6	
[L27]	Loguru[a]	2015	6					6*				5			4	3				1	2	
[L28]	Spdlog[a]	2014	6				0	1				2			3	4			5			
[L29]	Java Util Logging[e]	2002	7	300		400				500	700	800			900			1000				
[L30]	Logrus[b]	2013	7				6	5				4			3	2					1	
[L31]	Uber-go/zap[b]	2016	7					0				1			2	3						
[L32]	Bolterauer[m]	2008	7	7		6			1	5		4			3						2	
[L33]	Swift-log[i]	2018	7				0	1				2		3	4	5			6			
[L34]	Loguru[k]	2017	7				5	10				20	25		30	40			50			
[L35]	Syslog-ng[a]	1998	8					7				6		5	4	3			2	1		0
[L36]	PHP[n]	2001																				
	(on Linux)		8					7				6		5	4	3			2	1		0
	(on Wind.)		8					6				6		6	5	1			1	1		
[L37]	Monolog[n]	2011	8					100				200		250	300	400			500	550	600	
[L38]	Winston[g]	2011	8					7				6		5	4	3			2	1		0
[L39]	Log4C	2002 ?	9				800	700				600		500	400	300			200	100	0	
[L40]	Log4Net[j]	2004	15	10000	10000	20000	20000	30000		30000		40000		50000	60000	70000		80000	90000	100000	110000	120000
				3	4	3	23	39	1	3	1	42	1	10	41	41	1	2	13	8	22	4

[a]C/C++, [b]Golang, [c]Objective-C, [d]Rust, [e]Java, [f]Ruby, [g]JavaScript, [h]Kotlin, [i]Swift, [j]C#, [k]Python, [l]MatLab, [m]VBA, [n]PHP

l'exception des bibliothèques de Python [L17] et Bolterauer [L32]. L17 présente la seule variation dans l'ordre numérique entre *Alert* et *Critical*. L32 a un numérotage du niveau *Debug* différent de toutes les autres bibliothèques. De plus, 60% des bibliothèques (25) ont leurs niveaux triés dans un ordre croissant et 31% (11) dans un ordre décroissant.

Observation #10 : *Trois bibliothèques présentent une redondance dans les valeurs numériques de leurs niveaux de sévérité des logs.*

> *« Deux niveaux ayant la même valeur sont considérés comme équivalents. »* [L40]

Il est possible d'observer des bibliothèques avec la même valeur numérique pour différents niveaux de sévérité, ils sont : (i) les niveaux *Critical* et *Fatal* dans la bibliothèque Python [L17] ; (ii) *Info* et *Notice*, (iii) *Error*, *Critical* et *Alert*, dans la bibliothèque PHP [L36], lors de l'exécution sur le système d'exploitation Windows ; (iv) *Finest* et *Verbose*, (v) *Finer* et *Trace*, (vi) *Debug* et *Fine* dans Log4Net [L40]. Le fait que différents niveaux de sévérité aient la même valeur numérique indique une redondance des niveaux de logs, bien exemplifiée par la citation suivante :

> *« Pourquoi l'interface org.slf4j.Logger n'a-t-elle pas de méthodes pour le niveau FATAL ? L'interface Marker (...) rend le niveau FATAL largement redondant. Si une erreur donnée nécessite une attention au-delà de celle allouée pour les erreurs ordinaires, marquez simplement l'instruction de journalisation avec un marqueur spécialement désigné qui peut être nommé 'FATAL' ou tout autre nom à votre guise. »* [L06]

Cette citation illustre que, dans SLF4J, le niveau *Fatal* est jugé redondant car l'interface *Marker* permet d'ajouter des informations aux messages de log, comme un marqueur nommé « FATAL », sans recourir à un niveau de sévérité distinct. Cette approche introduit une dimen-

sion supplémentaire pour traiter les logs, plus souple que le système classique à cinq niveaux (*Trace*, *Debug*, *Info*, *Warn*, *Error*).

Observation #11 : *Il peut y avoir une similitude sémantique dans l'utilisation des niveaux Trace et Debug.*

L'*« interface Marker »* est une option fournie par les bibliothèques SLF4J [L06] et Log4J [L13] pour ajouter plus de contexte à une instruction de log et éviter la redondance, ce qui permet d'utiliser uniquement les niveaux de logs nécessaires.

Le niveau *Trace* est présent dans 55% des bibliothèques sélectionnées, cependant, en analysant les notes de version de ces bibliothèques, dans au moins trois d'entre elles, le niveau *Trace* n'était pas présent dans les premières versions. Il a été ajouté à Log4J [L13] en 2005, à SLF4J [L06] en 2007 et à JS-Logger [L09] en 2018. Selon la page FAQ de SLF4J, le niveau *Trace* était utilisé dans plusieurs projets

> *« pour désactiver la sortie de journalisation de certaines classes sans avoir besoin de configurer la journalisation pour ces classes. (...) Ainsi, dans de nombreux cas, le niveau TRACE portait une signification sémantique similaire à DEBUG. »* [L06]

DÉFINITIONS

Observation #12 : *Severe, Critical, Alert, Fatal et Emergency ont des descriptions similaires concernant des événements d'erreur graves dans les bibliothèques.*

Observation #13 : *Dans les bibliothèques, nous avons trouvé 19 niveaux de sévérité. Malgré la diversité de la nomenclature, les concepts de niveaux sont cohérents à travers les différentes bibliothèques, suggérant une convergence vers des concepts de granularité supérieure.*

TABLEAU 3.4 : Définitions des niveaux de sévérité des bibliothèques. Adapté de Mendes & Petrillo (2021).

Niveau	Bibliothèque	Citation
Fine*, Verb.	[L09]	« *All of FINE, FINER, and FINEST are intended for relatively detailed tracing. The exact meaning of the three levels will vary between subsystems, but in general, FINEST should be used for the most voluminous detailed output, FINER for somewhat less detailed output, and FINE for the lowest volume (and most important) messages.* »
	[L12]	« *These levels designate fine-grained informational events that are most useful to debug an application* ».
Verb.	[L14]	« *Verbose is the noisiest level, rarely (if ever) enabled for a production app* ».
Trace	[L04]	« *Designates very low priority, often extremely verbose, information* ».
	[L05]	« *The TRACE level designates informational events of very low importance* ».
	[L06]	« *The TRACE Leve l designates finer-grained informational events than the DEBUG level* ».
	[L13]	« *A fine-grained debug message, typically capturing the flow through the application* ».
	[L14]	« *(...) more detailed information. Expect these to be written to logs only* ».
	[L15]	« *Logging from external libraries used by your app or very detailed application logging* ».
	[L16]	« *For trace debugging ; begin method X, end method X* ».
	[L18]	« *Logs that contain the most detailed messages. (...) may contain sensitive application data. (...) should never be enabled in a production environment* ».
	[L40]	« *The Trace level designates fine-grained informational events that are most useful to debug an application* ».
Debug	[L04]	« *Designates lower priority information* ».
	[L06]	« *The DEBUG Level designates fine-grained informational events that are most useful to debug an application* ».
	[L05]	« *The DEBUG level designates informational events of lower importance* ».
	[L13]	« *A general debugging event* ».
	[L14]	« *Detailed information on the flow through the system. Expect these to be written to logs only* ».
	[L15]	« *Anything else, i. e. too verbose to be included in 'info' level* ».
	[L16]	« *For debugging ; executed query, user authenticated, session expired* ».
	[L17]	« *Houston, we have a %s, 'thorny problem'* ».
	[L18]	« *Logs that are used for interactive investigation during development. (...) should primarily contain information useful for debugging and have no long-term value* ».
	[L35]	« *The message is only for debugging purposes* ».
	[L36]	« *debug-level message* ».
	[L40]	« T*he Debug level designates fine-grained informational events that are most useful to debug an application* ».
	[L14]	« *Debug is used for internal system events that are not necessarily observable from the outside, but useful when determining how something happened* ».
Info	[L04]	« *Designates useful information* ».
	[L05]	« *The INFO level designates informational messages highlighting overall progress of the application* ».
	[L06]	« *The INFO level designates informational messages that highlight the progress of the application at coarse-grained level* ».
	[L13]	« *An event for informational purposes* ».
	[L14]	« *Interesting runtime events (startup/shutdown). Expect these to be immediately visible on a console, so be conservative, and keep to a minimum* ».
	[L15]	« *Detail on regular operation* ».
	[L16]	« *Normal behavior like mail sent, user updated profile etc* ».
	[L17]	« *Houston, we have a %s, 'interesting problem'* ».
	[L18]	« *Logs that track the general flow of the application. These logs should have long-term value* ».
	[L19]	« *(...) things happening in the system that correspond to its responsibilities and functions. (...) observable actions the system can perform* ».
	[L29]	« *INFO is a message level for informational messages. Typically INFO messages will be written to the console or its equivalent. So the INFO level should only be used for reasonably significant messages that will make sense to end users and system administrators.*"
	[L35]	« *The message is purely informational* ».
	[L36]	« *informational message* ».
	[L40]	« *The Info level designates informational messages that highlight the progress of the application at coarse-grained level* ».
Notice	[L03]	« *Captures information that is essential for troubleshooting problems. For example, capture information that might result in a failure* ».
	[L35]	« *The message describes a normal but important event* ».
	[L36]	« *normal, but significant, condition* ».
	[L40]	« *The Notice level designates informational messages that highlight the progress of the application at the highest level* ».

TABLEAU 3.5 : Définitions des niveaux de sévérité des bibliothèques (Partie 2 suite). Adapté de Mendes & Petrillo (2021).

Niveau	Bibliothèque	Citation
Warn	[L04]	« *Designates hazardous situations* ».
	[L05]	« *The WARN level designates potentially harmful situations* ».
	[L06]	« *The WARN level designates potentially harmful situations* ».
	[L13]	« *An event that might possible lead to an error* ».
	[L14]	« *Use of deprecated APIs, poor use of API, 'almost' errors, other runtime situations that are undesirable or unexpected, but not necessarily 'wrong'. Expect these to be immediately visible on a status console* ».
	[L15]	« *A note on something that should probably be looked at by an operator eventually* ».
	[L16]	« *Something unexpected ; application will continue* ».
	[L17]	« *Houston, we have a %s, 'bit of a problem'* ».
	[L18]	« *Logs that highlight an abnormal or unexpected event in the application flow, but do not otherwise cause the application execution to stop.* »
	[L29]	« *WARNING is a message level indicating a potential problem. In general WARNING messages should describe events that will be of interest to end users or system managers, or which indicate potential problems* ».
	[L35]	« *Warning conditions / The message is warning* ».
	[L36]	« *warning conditions* ».
	[L40]	« *The Warn level designates potentially harmful situations* ».
Error	[L04]	« *Designates very serious errors* ».
	[L05]	« *The ERROR level designates error events which may or not be fatal to the application* ».
	[L06]	« *The ERROR level designates error events that might still allow the application to continue running* ».
	[L13]	« *An error in the application, possibly recoverable* ».
	[L14]	« *Other runtime errors or unexpected conditions. Expect these to be immediately visible on a status console* ».
	[L15]	« *Fatal for a particular request, but the service/app continues servicing other requests. An operator should look at this soon(ish)* ».
	[L16]	« *Something failed ; application may or may not continue* ».
	[L17]	« *Houston, we have a %s, 'major problem'* ».
	[L18]	« *(...) highlight when the current flow of execution is stopped due to a failure. These should indicate a failure in the current activity, not an application-wide failure* »
	[L35]	« *The message describes an error* ».
	[L36]	« *error conditions* ».
	[L40]	« *The Error level designates error events that might still allow the application to continue running* ».
Severe	[L40]	« *The Severe level designates very severe error events* ».
Critical	[L17]	« *Houston, we have a %s", 'major disaster'.* »
	[L18]	« *Logs that describe an unrecoverable application or system crash, or a catastrophic failure that requires immediate attention* »
	[L35]	« *The message states a critical condition* ».
	[L36]	« *critical conditions* ».
	[L40]	« *The Critical level designates very severe error events. Critical condition, critical* ».
Alert	[L35]	« *Action must be taken immediately* »
	[L36]	« *action must be taken immediately* ».
	[L40]	« *The Alert level designates very severe error events. Take immediate action, alerts* ».
Fatal	[L06]	« *The FATAL level designates very severe error events that will presumably lead the application to abort* ».
	[L13]	« *A severe error that will prevent the application from continuing* ».
	[L14]	« *Severe errors that cause premature termination. Expect these to be immediately visible on a status console* ».
	[L15]	« *The service/app is going to stop or become unusable now. An operator should definitely look into this soon* ».
	[L16]	« *Something bad happened ; application is going down* ».
	[L40]	« *The Fatal level designates very severe error events that will presumably lead the application to abort* ».
Emerg.	[L35]	« *The message says the system is unusable* ».
	[L36]	« *system is unusable* ».
	[L40]	« *The Emergency level designates very severe error events. System unusable, emergencies* ».

Les tableaux 3.4 et 3.5 présentent les définitions et descriptions trouvées pour les niveaux de sévérité des logs dans les bibliothèques [56]. Ensuite, nous commentons ces définitions des niveaux de sévérité, en soulignant les particularités et les similitudes :

(a) *Debug :* Selon les bibliothèques, le niveau *Debug* décrit des informations détaillées [L14, L15, L40] et de faible priorité/importance [L04, L05], qui aident dans les activités de débogage [L13, L16, L36, L40]. [L15] décrit *Debug* comme *« trop verbeux pour être inclus dans le niveau 'info' »,* suggérant une similitude dans le but du niveau. Une seule des bibliothèques utilise le mot *« problème »* pour décrire ce niveau [L17].

(b) *Trace :* Le niveau *Trace* est décrit avec les mêmes caractéristiques que *Debug* mais approfondit la faible priorité : *« très faible priorité »* [L04], *« très faible importance »* [L05].

(c) *Info :* Le niveau *Info* désigne des comportements normaux [L16], des opérations régulières [L15], et leurs messages *« soulignent (globalement) la progression de l'application »* [L05, L40] *« à un niveau grossier »* [L40]. Commons Logging [L14] conseille de les garder au minimum, car les messages à ce niveau généreront immédiatement des données. Java Util Logging [L29] souligne qu'ils ont de la valeur pour les utilisateurs finaux et les administrateurs système. Python [L17] utilise également le mot *« problème »* pour référer à ce niveau.

(d) *Warn :* Le niveau *Warn* est décrit comme une *« situation dangereuse »* [L04], mettant en évidence un problème potentiel [L05, L17, L29, L40] qui pourrait conduire à une erreur [L13]. Il est également décrit comme des *« quasi erreurs »* [L14], considérant que l'application fonctionne toujours bien que de manière inattendue [L14, L16]. Les

56. Alors que les items *g)* et *h)* regroupent les définitions par similitude sémantique, l'item *i)* les regroupe par faible occurrence, malgré leurs buts distincts.

opérateurs/utilisateurs finaux/gestionnaires de système devraient probablement être intéressés par les messages à ce niveau [L15, L29].

(e) *Notice :* Le niveau *Notice* ressemble au niveau *Info* dans trois définitions : il décrit des événements normaux [L35, L36] et souligne la progression de l'application [L40]. Cependant, selon OSLogging [L03], il peut décrire des échecs potentiels, rapprochant le niveau *Notice* du niveau *Warn*.

(f) *Error :* Les expressions utilisées pour décrire le niveau *Error* sont *« problème majeur »* [L17], *« erreur très sérieuse »* [L04], *« conditions inattendues »* [L14]. De plus, il y a aussi des descriptions que l'événement enregistré peut ou non interrompre le fonctionnement de l'application [L05, L16, L40]. Quoi qu'il en soit, même s'il n'arrête pas l'application dans son ensemble, il peut entraver le bon déroulement d'une demande particulière [L06]. En cas de logs de ce niveau, un opérateur doit en être informé dès que possible [L15].

(g) *Severe, Critical, Alert, Fatal, Emergency :* Dans les définitions des bibliothèques, cinq niveaux apparaissent liés à des événements d'erreur très graves : *Severe* [L40], *Critical* [L40], *Alert* [L40], *Fatal* [L13, L14, L40], et *Emergency*. Les définitions du niveau *Critical* laissent penser qu'un *« désastre »* s'est produit [L08]. De plus, le niveau *Critical* nécessite une action immédiate [L35, L36, L40]. Pour le niveau *Fatal*, on trouve plus de descriptifs pour l'événement : cela *« empêche l'application de continuer »* [L13], cela *« cause une terminaison prématurée »* [L14], l'application *« va s'arrêter »* [L15, L16], *« conduit à l'abandon »* [L40] ou *« devient inutilisable »* [L15].

(h) *Finest, Verbose, Finer, Fine :* Les descriptions pour ces niveaux apparaissent dans trois bibliothèques. Dans JS-logger [L9], elles sont décrites de manière similaire, sans termes précis pour distinguer leurs différences, telles que *« le plus volumineux détail »*,

« quelque peu moins détail », et *« le volume le plus bas »* de sortie. Pour SwiftyBeaver [L12], ces niveaux sont utiles pour déboguer une application.

(i) *Basic, Config, Success, Fault :* Ces quatre niveaux sont uniques dans quatre bibliothèques distinctes. Selon Bolterauer [L32], *Basic* est un alias pour le niveau *Debug*. Dans Java Util Logging [L29], le niveau *Config* décrit des messages *« destinés à fournir une variété d'informations de configuration statique, pour aider dans le débogage des problèmes. »* Loguru [L34] est la seule bibliothèque à offrir le niveau de sévérité *Success*, mais elle ne fournit pas de définition. Numériquement, il se situe entre les niveaux *Info* et *Warn*. Pour [L03], le niveau *Fault « capture des informations sur les fautes et les bugs dans votre code »* ; la bibliothèque, pour autant que nous le sachions, ne fournit pas de valeurs numériques pour les niveaux.

3.3 CARTOGRAPHIE DU POINT DE VUE DES PRATICIENS

Après avoir cartographié les définitions des niveaux de sévérité dans la littérature et dans les bibliothèques de journalisation, cette section se concentre sur le point de vue des praticiens, tel qu'il est exprimé sur un site de questions-réponses largement utilisé par les développeurs. Cette analyse vise à comprendre comment les niveaux de sévérité sont perçus et appliqués par les praticiens dans leur pratique quotidienne. Ce complément au mappage théorique et technique permet de mieux saisir les défis rencontrés dans la compréhension et l'usage des niveaux de sévérité dans des environnements réels.

3.3.1 MÉTHODOLOGIE

Dans les directives de Garousi *et al.* (2019), l'importance de l'information contextuelle dans l'étude suggère l'inclusion de la littérature grise. Par conséquent, nous avons adopté une

recherche automatique sur *Stack OverFlow*[57], *« un forum majeur où les praticiens postent des questions et discutent de problèmes techniques »* (Garousi *et al.*, 2016), comme stratégie de recherche pour capturer la vue des praticiens. Notre requête de recherche était : « `log levels` », en utilisant le filtre « `is:question` ». Nous avons trouvé **742 résultats** (à la date de rédaction : juin 2021). Ensuite, nous avons appliqué les critères d'inclusion (ACI) et d'exclusion (ACE) :

- **ACI1 :** la question/réponse explique quand utiliser au moins cinq des niveaux de journalisation.
- **ACI2 :** la question/réponse doit avoir au moins deux votes.
- **ACE1 :** la question/réponse n'est pas originale (copiée d'une autre source telle que des bibliothèques de logs ou des RFCs).
- **ACE2 :** la question/réponse consiste en des messages exemplifiant les caractéristiques des niveaux de logs.
- **ACE3 :** la question n'est pas approuvée et est fermée par Stack Overflow.

Après avoir appliqué les critères, nous avons retenu **quatre questions** avec **neuf réponses pertinentes** (Tableau 3.6). Bien que le nombre de questions et de réponses retenues puisse paraître limité à première vue, celles-ci couvrent les six niveaux de sévérité les plus fréquemment rapportés dans la littérature et dans les bibliothèques de journalisation. Ensemble, elles nous fournissent au moins 40 définitions ou descriptions distinctes de ces niveaux, permettant ainsi une analyse qualitative représentative des perceptions et usages des praticiens dans des contextes concrets de développement logiciel.

57. https://stackoverflow.com/

TABLEAU 3.6 : Questions sélectionnées sur Stack Overflow.
Adapté de Mendes & Petrillo (2021).

#	Titre	URL	Réponses
[QSO1]	*« When to use the different log levels »*	https://bit.ly/2SQhCE8	[ASO1] [ASO2] [ASO3]
[QSO2]	*« Logging levels - Logback rule-of-thumb to assign log levels »*	https://bit.ly/3hNei5d	[ASO4] [ASO5] [ASO6] [ASO7]
[QSO3]	*« Difference between logger.info and logger.debug »*	https://bit.ly/3hgKKhg	[ASO8]
[QSO4]	*« How to use log levels in Java »*	https://bit.ly/3wa1UBn	[ASO9]

3.3.2 RÉSULTATS

Observation #14 : *Les niveaux de sévérité discutés dans les réponses sélectionnées corroborent les niveaux les plus cités et définis dans la littérature évaluée par les pairs et les bibliothèques de logging, respectivement.*

Dans les réponses sélectionnées de StackOverflow, six niveaux de sévérité des logs sont décrits, parmi lesquels les plus discutés sont *Debug* (9), *Error* (9), *Warn* (8) et *Info* (8) ; les deux autres niveaux sont *Trace* (5) et *Fatal* (3).

DÉFINITIONS

Le tableaux 3.7 et 3.8 présentent les définitions et descriptions trouvées pour les niveaux de sévérité de cette étape. Ci-dessous, nous présentons nos observations sur les interprétations fournies par les praticiens, en soulignant les convergences, les divergences et les nuances dans la compréhension et l'utilisation de chaque niveau de sévérité.

(a) *Debug, Trace :* Contrairement aux deux premières sources de notre cartographie, la similarité observée entre les niveaux de sévérité n'est pas aussi frappante dans les

TABLEAU 3.7 : Définitions des niveaux de sévérité de Stack Overflow. Adapté de Mendes & Petrillo (2021).

Niveau	Réponse	Citation
Trace	[ASO1]	« *Only when I would be tracing the code and trying to find one part of a function specifically.* »
	[ASO2]	« *Trace is by far the most commonly used severity and should provide context to understand the steps leading up to errors and warnings* ».
	[ASO3]	« *The TRACE messages are intended for developers when they don't need to log state variables* ».
	[ASO4]	« *We don't use this often, (...) extremely detailed and potentially high volume logs that you don't typically want enabled even during normal development (...)* »
	[ASO5]	« *Trace is something i have never actually used* ».
Debug	[ASO1]	« *Information that is diagnostically helpful to people more than just developers (IT, sysadmins, etc.)* ».
	[ASO2]	« *We consider Debug < Trace. (...) we discourage use of Debug messages (...) this makes log files almost useless (...)* »
	[ASO3]	« *The DEBUG messages are intended for developers when they need to log state variables* ».
	[ASO4]	« *(...) any message that is helpful in tracking the flow through the system and isolating issues, especially during the development and QA phases (...)* »
	[ASO5]	« *Debug means that something normal and insignificant happened ; (...)* »
	[ASO6]	« *Shouldn't be used at all (and certainly not in production) (...)* »
	[ASO7]	« *variable contents relevant to be watched permanently* ».
	[ASO8]	« *If you want to print the value of a variable at any given point, you might call Logger.debug* ».
	[ASO9]	« *As the name says, debug messages that we only rarely turn on (...)* ».
Info	[ASO1]	« *Generally useful information to log (service start/stop, configuration assumptions, etc). (...) I want to always have available but usually don't care about under normal circumstances. This is my out-of-the-box config level* ».
	[ASO2]	« *This is important information that should be logged under normal conditions such as successful initialization, services starting and stopping or successful completion of significant transactions (...)* »
	[ASO3]	« *The INFO messages are intended for system operators and describe expected states* ».
	[ASO4]	« *Things we want to see at high volume in case we need to forensically analyze an issue. System lifecycle events (system start, stop) go here. (...) Typical business exceptions can go here (...)* »
	[ASO5]	« *Info means that something normal but significant happened ; the system started, the system stopped, (...)* »
	[ASO6]	« *Anything else that we want to get to an operator.(...) log message per significant operation (...)* »
	[ASO7]	« *used in functions/methods first line, to show a procedure that has been called or a step gone ok, (...)* »
	[ASO9]	« *Anything that we want to know when looking at the log files, e.g. when a scheduled job started/ended (...)* »
Warn	[ASO1]	« *Anything that can potentially cause application oddities, but for which I am automatically recovering. (...)* »
	[ASO2]	« *This MIGHT be problem, or might not. (...) Viewing a log filtered to show only warnings and errors may give quick insight into early hints at the root cause of a subsequent error. Warnings should be used sparingly so that they don't become meaningless. (...)* »

TABLEAU 3.8 : Définitions des niveaux de sévérité de Stack Overflow (Partie 2 suite). Adapté de Mendes & Petrillo (2021).

Niveau	Réponse	Citation
Warn	[ASO3]	« *The WARN messages are intended for system operators when the process can continue in an unwanted state* ».
	[ASO4]	« *An unexpected technical or business event happened, customers may be affected, but probably no immediate human intervention is required. (...) Basically any issue that needs to be tracked but may not require immediate intervention* ».
	[ASO5]	« *Warn means that something unexpected happened, but that execution can continue, perhaps in a degraded mode ;(...) Something is not right, but it hasn't gone properly wrong yet - warnings are often a sign that there will be an error very soon* ».
	[ASO6]	« *This component has had a failure believed to be caused by a dependent component (...). Get the maintainers of THAT component out of bed* ».
	[ASO7]	« *not-breaking issues, but stuff to pay attention for. Like a requested page not found* ».
	[ASO9]	« *Any message that might warn us of potential problems, (...)* »
Error	[ASO1]	« *Any error which is fatal to the operation, but not the service or application (...) These errors will force user (administrator, or direct user) intervention (...)* ».
	[ASO2]	« *Definitely a problem that should be investigated. SysAdmin should be notified automatically, but doesn't need to be dragged out of bed (...)* »
	[ASO3]	« *The ERROR messages are intended for system operators when, despite the process cannot continue in an unwanted state, the application can continue* ».
	[ASO4]	« *The system is in distress, customers are probably being affected (or will soon be) and the fix probably requires human intervention. The »2AM rule » applies here-if you're on call, do you want to be woken up at 2AM if this condition happens ? If yes, then log it as 'error'* ».
	[ASO5]	« *Error means that the execution of some task could not be completed ; (...) Something has definitively gone wrong* ».
	[ASO6]	« *This component has had a failure and the cause is believed to be internal (...). Get me (maintainer of this component) out of bed* ».
	[ASO7]	« *critical logical errors on application, like a database connection timeout. Things that call for a bug-fix in near future* ».
	[ASO8]	« *When responding to an Exception, you might call Logger.error* ».
	[ASO9]	« *Any error/exception that is or might be critical. Our Logger automatically sends an email for each such message on our servers* ».
Fatal	[ASO1]	« *Any error that is forcing a shutdown of the service or application to prevent data loss (or further data loss)* ».
	[ASO2]	« *Overall application or system failure that should be investigated immediately.(...) wake up the SysAdmin. (...) this severity should be used very infrequently (...)* »
	[ASO3]	« *The FATAL messages are intended for system operators when the application cannot continue in an unwanted state* ».

réponses sélectionnées de StackOverflow. Il serait préférable d'utiliser *Debug* plutôt que *Trace* pour une partie des réponses [ASO1, ASO4, ASO5], et l'inverse pour une autre partie [ASO2, ASO6, ASO9]. Pour ASO3, les deux niveaux sont destinés aux développeurs, *Debug* étant celui qui enregistre les valeurs des variables. Pour ASO1, le niveau *Trace* est utilisé pour trouver un morceau de code spécifique, tandis que le niveau Debug est classé comme *« utile à des personnes plus que juste aux développeurs. »*

(b) *Info :* Parmi les niveaux discutés dans les réponses sélectionnées, le niveau *Info* a la plus grande convergence dans les définitions présentées. Toutes les réponses le décrivent comme un enregistrement d'opérations qui commencent et/ou se terminent, décrivant *« des situations normales mais significatives »* d'un système logiciel [ASO1, ASO2, ASO5, ASO6], c'est-à-dire, *« des situations attendues »* [ASO3]. ASO4 souligne que ce niveau décrit également des exceptions typiques, et selon ASO6, les opérateurs sont le public des messages *Info*.

(c) *Warn :* Comme dans les bibliothèques, les réponses sélectionnées décrivent les messages du niveau *Warn* comme des problèmes potentiels/événements inattendus [ASO1, ASO2, ASO4, ASO5, ASO9] qui peuvent causer des complications pour le système [ASO1, ASO5], et doivent donc être observés. Malgré ces événements, le système reste en fonctionnement [ASO1, ASO3, ASO5], sans nécessité d'intervention humaine immédiate [ASO4]. Pour ASO3, les opérateurs sont le public intéressé par ce niveau de sévérité.

(d) *Error, Fatal :* Il y a également une divergence à ce niveau. Pour quatre des réponses [ASO1, ASO2, ASO3, ASO6], le niveau *Error* indique une défaillance qui n'a pas arrêté l'exécution du système mais qui devrait être enquêtée par les opérateurs du système [ASO2, ASO3]. Cependant, pour deux autres réponses, le degré de sévérité est plus critique, et la *« personne intéressée »* devrait « sortir du lit » [ASO4, ASO5]. Ce degré de sévérité est le même qui est attribué au niveau *Fatal* par ASO1, ASO2, et ASO3 : des

erreurs se produisent qui forcent l'application à *« s'arrêter »* et nécessitent une action immédiate.

3.4 SYNTHÈSE DES NIVEAUX DE SÉVÉRITÉ DES LOGS

En analysant nos trois sources, nous avons constaté une cohérence dans les valeurs numériques des niveaux, une similarité sémantique dans leurs définitions, ainsi qu'une faible occurrence de certains niveaux dans les bibliothèques. Tenant compte de ces similarités sémantiques, des redondances numériques et de la fréquence d'apparition des niveaux de sévérité dans les bibliothèques, nous avons abstrait les 19 niveaux de sévérité en six niveaux principaux. Ces six niveaux représentent l'état de pratique actuel de la journalisation et peuvent être associés à quatre finalités distinctes en matière de sévérité des logs. Ci-dessous, nous expliquons les étapes de cette synthèse, comme illustré dans la figure 3.6.

3.4.1 ABSTRACTION DES NIVEAUX DE SÉVÉRITÉ DES LOGS

Finest, Verbose, Finer, Trace, Debug, Basic, Fine, Config. Les niveaux *Finest* et *Verbose* sont présents dans six bibliothèques. L29 et L32 fournissent le niveau *Finest* ; L10, L12 et L19 fournissent le niveau *Verbose*, et L40 fournit les deux niveaux. Outre la similarité sémantique entre leurs définitions/descriptions dans six bibliothèques, dans L40 ils ont la même valeur numérique. Ces faits suggèrent qu'ils peuvent être fusionnés en un même niveau. Dans la figure 3.6, nous avons choisi de fusionner vers la sévérité avec la plus haute occurrence. Nous avons effectué le même processus pour les niveaux *Finer* et *Trace*, ainsi que pour les niveaux *Debug* et *Fine*. Le niveau *Basic*, présent uniquement dans L32, est amalgamé avec le niveau de sévérité *Debug* dans sa documentation et fusionné à *Debug*. Les niveaux *Verbose* et *Trace* résultants ont encore des définitions avec une similarité sémantique substantielle,

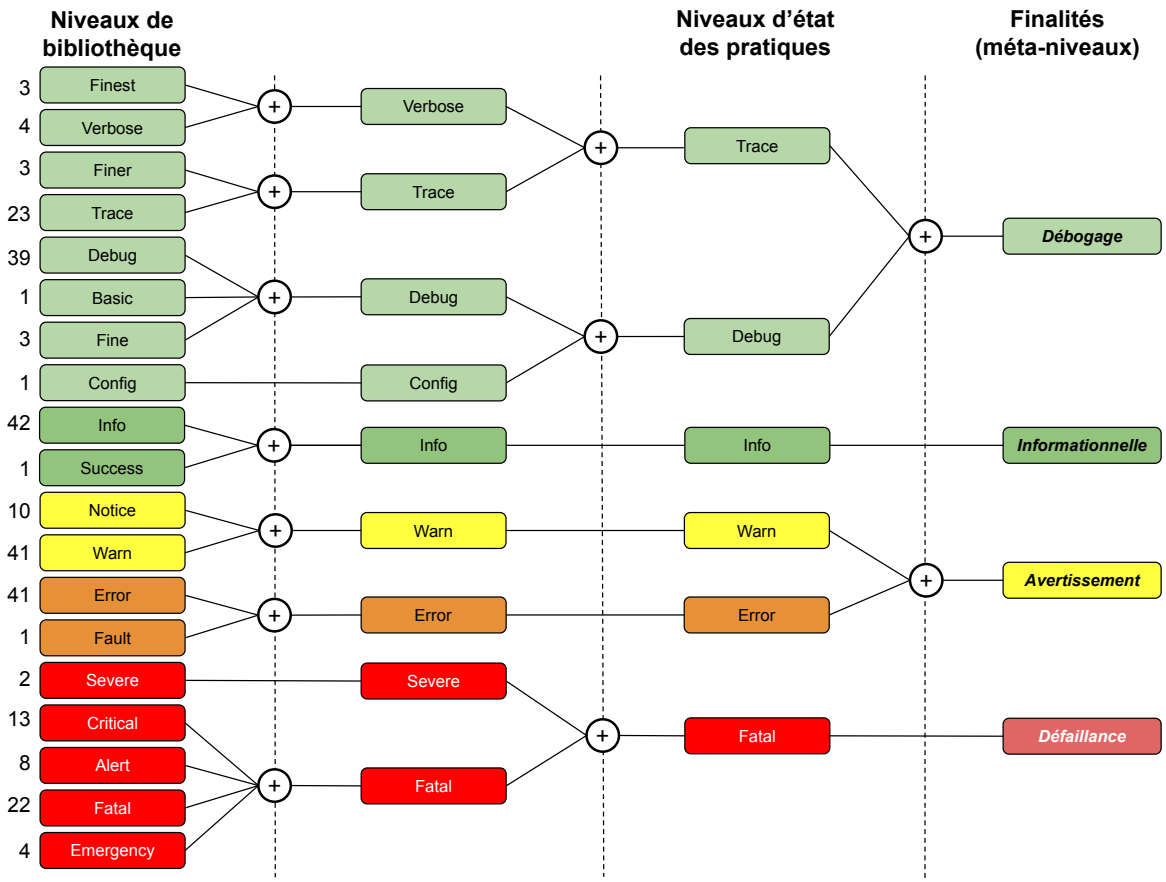

FIGURE 3.6 : Abstraction des niveaux de sévérité des logs. (Les numéros à gauche du niveau indiquent l'occurrence de chaque niveau dans les bibliothèques sélectionnées). Adapté de Mendes & Petrillo (2021).

donc nous avons abstrait le niveau de *Verbose* à *Trace*. Le niveau *Config*, de faible occurrence, fournit des informations pour le débogage [L29] ; il a été fusionné avec le niveau *Debug*.

Info, Success. Nous avons observé que le niveau *Success* a une valeur numérique entre les niveaux *Info* et *Warn* dans L34. De plus, le niveau *Success* a une faible occurrence et n'a pas de définition. Nous l'avons fusionné avec *Info* car il va à l'encontre de l'objectif de *Warn*.

Notice, Warn. Dans les bibliothèques, le niveau *Notice* a une faible occurrence. De plus, il ressemble au niveau *Info* tout en se rapprochant du niveau *Warn*. Considérant la combinaison de sa définition trouvée dans la littérature et la nomenclature des niveaux, nous avons fusionné le niveau *Notice* avec le niveau *Warn*.

Error, Fault. Le niveau *Fault* est un autre niveau de faible occurrence. Pour L03, le niveau *Fault* décrit des bugs dans les systèmes logiciels en fonctionnement et, de plus, il se classe avant le niveau de sévérité *Critical*. Par conséquent, nous avons fusionné *Fault* avec *Error*.

Fatal, Alert, Critical, Emergency, Severe. Les niveaux *Fatal*, *Critical*, *Alert* et *Emergency* sont les niveaux les plus graves sur 81% des bibliothèques sélectionnées, et leurs définitions ont une similarité sémantique prononcée. Dans notre abstraction, le niveau *Fatal* est le niveau qui décrit la situation des défaillances et il a la plus grande importance parmi les niveaux analysés (*Fatal*, *Critical*, *Alert*, *Severe* et *Emergency*). Les niveaux *Fatal* et *Critical* ont la même valeur dans L17. Finalement, le niveau *Severe* a une faible occurrence, mais sa définition a une similarité sémantique avec le niveau *Fatal*. Ces faits nous ont conduits à fusionner *Fatal*, *Critical*, *Alert*, *Severe* et *Emergency* dans le niveau *Fatal*.

3.4.2 DÉFINITIONS SYNTHÉTISÉES

Nous avons effectué le processus de synthèse, aboutissant à six niveaux de sévérité des logs de pratique courante : *Trace, Debug, Info, Warn, Error, Fatal*. Ainsi, en considérant les résultats obtenus à partir des trois sources, nous avons synthétisé des définitions combinées pour les six niveaux abstraits comme presenté dans le tableau 3.9.

Note méthodologique

Bien que des exemples concrets pourraient illustrer l'application de chaque niveau de sévérité, cette thèse s'est volontairement limitée à une analyse conceptuelle et empirique fondée sur trois sources principales : la littérature, les bibliothèques de journalisation et les points de vue des praticiens. Fournir des exemples issus de systèmes réels aurait nécessité une méthodologie complémentaire impliquant l'analyse de traces d'exécution ou de journaux d'applications en production, ce qui aurait dépassé le périmètre de ce travail. De plus, des exemples spécifiques pourraient induire une couverture partielle des usages possibles, compromettant la portée généralisable du cadre proposé. Cette lacune ouvre toutefois la voie à des recherches futures sur la contextualisation et la mise en œuvre pratique des niveaux de sévérité.

3.4.3 FINALITÉS DES NIVEAUX DE SÉVÉRITÉ DES LOGS

Nous avons observé une convergence des niveaux de sévérité des logs à travers les trois sources après les processus d'abstraction et de synthèse, et nous avons noté quatre principaux finalités (méta-niveaux) pour les niveaux de sévérité des logs :

Finalité de débogage : (en anglais, *debugging purpose*) elle décrit les niveaux utilisés pour enregistrer les états des variables et les événements internes au comportement d'un

TABLEAU 3.9 : Définitions des niveaux de sévérité synthétisées.
© Eduardo Mendes de Oliveira, 2024

Niveau de sévérité	Description
Trace	Suit de manière générale les états des variables et les événements dans un système logiciel.
Debug	Décrit les états des variables et les détails sur des événements intéressants et les points de décision dans le flux d'exécution d'un système logiciel, ce qui aide les développeurs à enquêter sur les événements internes du système.
Info	Décrit des événements normaux, qui informent sur la progression attendue et l'état d'un système logiciel.
Warn	Décrit des situations potentiellement dangereuses causées par des événements et des états inattendus. Ces situations doivent être observées, même si elles n'interrompent pas l'exécution du système logiciel.
Error	Décrit l'apparition d'un comportement inattendu d'un système logiciel. Ces situations doivent être enquêtées, même si elles n'interrompent pas l'exécution du système logiciel.
Fatal	Décrit des événements critiques qui entraînent la défaillance d'un système logiciel.

système logiciel. Elle regroupe *Debug* et *Trace*, tandis que *Trace* extrapole les caractéristiques de *Debug* de décrire les variables et les événements.

Finalité informationnelle (en anglais, *informational purpose*) : elle décrit les niveaux utilisés pour enregistrer le comportement attendu d'un système logiciel.

Finalité d'avertissement (en anglais, *warning purpose*) : elle décrit les niveaux utilisés pour avertir du comportement inattendu d'un système logiciel. Elle regroupe les niveaux *Error* et *Warn* parce que les deux indiquent des problèmes (ou des problèmes potentiels) qui devraient être enquêtés, mais ne interrompent pas l'exécution du système.

Finalité de défaillance (en anglais, *failure purpose*) : elle décrit les niveaux utilisés pour enregistrer les défaillances d'un système logiciel.

3.5 DISCUSSION

Dans cette section, nous discutons les principaux résultats de notre étude sur les niveaux de sévérité dans les trois sources étudiées. Nous abordons les problèmes liés à la diversité excessive des niveaux, au manque de clarté dans leurs définitions et à l'utilisation de valeurs numériques pour distinguer ces niveaux. Nous proposons également des recommandations pour simplifier et standardiser l'utilisation des niveaux de sévérité afin d'améliorer les pratiques de journalisation.

3.5.1 IL EXISTE UNE VARIÉTÉ EXCESSIVE DE NIVEAUX DE SÉVÉRITÉ PARMI LES BIBLIOTHÈQUES DE JOURNAUX

68% des niveaux de sévérité des journaux peuvent être considérés comme des spécialisations « prosaïques » des 32% de niveaux de sévérité les plus représentatifs selon la littérature évaluée par les pairs, les bibliothèques de journaux et le point de vue des praticiens. La spécialisation excessive des niveaux peut rendre difficile le choix d'un niveau approprié et impacter la quantité de données générées ainsi que leur fiabilité.

Nous recommandons de maintenir une nomenclature standard pour les niveaux : *Trace, Debug, Info, Warn, Error* et *Fatal*. De plus, les développeurs devraient limiter le nombre de niveaux de sévérité. Ainsi, nous recommandons d'utiliser uniquement ces six niveaux de sévérité. Nous recommandons également de créer une politique d'utilisation des niveaux de sévérité, avec des exemples pratiques pour guider efficacement le choix des niveaux de sévérité.

3.5.2 IL Y A UN MANQUE DE PRÉCISION DANS LES DÉFINITIONS DES NIVEAUX DE SÉVÉRITÉ DES JOURNAUX

Nous avons observé un manque de précision dans les définitions des niveaux de sévérité des journaux. Par exemple, il existe des niveaux de sévérité dans les définitions des bibliothèques sans distinction de buts spécifiques, distingués uniquement par des adjectifs et des superlatifs. Ce manque de précision peut entraîner une mauvaise compréhension des niveaux de sévérité et nuire aux pratiques de journalisation. Ainsi, nous suggérons que les créateurs de bibliothèques de journaux fournissent des définitions précises et sans ambiguïté en tenant compte des finalités des niveaux de journalisation.

3.5.3 LES VALEURS DES NIVEAUX DE SÉVÉRITÉ DES JOURNAUX AIDENT À COMPRENDRE LA DISTINCTION ENTRE LES NIVEAUX

La grande majorité des bibliothèques de journaux utilisent une valeur numérique associée aux niveaux de sévérité. En l'absence de définitions précises, ces valeurs clarifient la proposition des créateurs concernant le degré de sévérité de chaque niveau. Nous suggérons que les créateurs de bibliothèques de journaux continuent de fournir ces valeurs. Nous suggérons également que ces valeurs soient ascendantes, en tenant compte de l'ordre entre les six niveaux de sévérité de l'état de la pratique, du moins sévère au plus sévère : *Trace, Debug, Info, Warn, Error* et *Fatal*.

3.5.4 EXPLORER LES FONCTIONNALITÉS DES BIBLIOTHÈQUES DE JOURNALISATION

L'utilisation d'un ensemble limité de niveaux peut entraîner des difficultés lorsqu'il est nécessaire d'identifier des données de journal spécifiques. Nous recommandons que les déve-

loppeurs explorent les fonctionnalités de la bibliothèque telles que l'interface « Marker » qui peut ajouter une sémantique aux niveaux de journal utilisés en pratique. Nous recommandons également d'étudier les paramètres de la bibliothèque de journalisation pour éviter de résoudre avec des niveaux de sévérité ce qui pourrait être résolu avec des pratiques de configuration.

3.5.5 LA RECHERCHE SUR LES NIVEAUX DE SÉVÉRITÉ DES JOURNAUX A AUGMENTÉ CES DERNIÈRES ANNÉES

La communauté de la journalisation reconnaît que le choix du niveau de sévérité des journaux peut être difficile et peut impacter les systèmes en développement et en production. Par conséquent, ces dernières années, les niveaux de sévérité des journaux ont été plus généralement étudiés. Malgré cet intérêt croissant, les chercheurs n'ont pas exploré ce que sont et devraient être les niveaux de sévérité dans nos pratiques de journalisation.

Du point de vue du génie logiciel, il est nécessaire de mener cette discussion sur les finalités des niveaux de sévérité des journaux et combien de niveaux de sévérité sont nécessaires pour générer des données de journal fiables.

3.5.6 DÉFINITIONS DES NIVEAUX ET DES FINALITÉS COMME BASE POUR DES GUIDES

La revue de littérature a révélé de nombreuses catégorisations des pratiques de journalisation, mais celles-ci restent généralement d'une portée plus large, laissant une lacune dans la description spécifique des niveaux de sévérité. Cependant, ces catégories soutiennent notre proposition de finalités pour les niveaux de sévérité, en renforçant la nécessité de définitions précises. Par exemple, les catégories de messages définies par He *et al.* (2018), comme opérations du programme (qui, bien qu'elle ne recommande pas explicitement le niveau *Info*,

l'utilise en exemple), et la catégorie suivie de l'état d'exécution proposée par Li *et al.* (2020a) s'accordent avec notre finalité informationnelle. De même, les catégories *débogage* (identifiée par Zeng *et al.* (2019)) et *assistance au dépannage* ainsi que *journalisation comptable* de Li *et al.* (2020a) correspondent à notre finalité de débogage.

Ces travaux soulignent également les difficultés rencontrées par les développeurs dans le choix des niveaux de sévérité, ainsi que l'absence de directives claires pour faciliter ce processus.Cette situation peut mener à des incohérences entre le niveau de sévérité choisi et la logique sous-jacente à cette décision Zeng *et al.* (2019). En réponse, nous proposons que les définitions des niveaux et finalités de sévérité présentées ici soient utilisées comme base pour élaborer des lignes directrices précises. Nos définitions reposent sur les six niveaux les plus récurrents identifiés dans nos résultats et fournissent une orientation claire pour les pratiques de journalisation, adaptée aux besoins réels des environnements de développement et de production.

3.6 MENACES À LA VALIDITÉ

Voici les principales menaces à la validité liées à la recherche décrite dans ce chapitre, ainsi que les mesures prises pour les atténuer.

Validation des définitions et des finalités des niveaux de sévérité. Dans cette étude, nous ne validons pas les définitions et les finalités par rapport aux entrées de journaux réelles. Des études empiriques doivent être réalisées pour observer l'adhésion de ces définitions et propositions aux pratiques de journalisation dans des systèmes logiciels réels, ainsi que des expériences contrôlées pour évaluer l'efficacité de ces définitions.

Cartographie des bibliothèques de journalisation. Le fait que notre travail ne couvre pas un ensemble exhaustif de bibliothèques est un facteur qui peut réduire la validité des

résultats. Cependant, nous avons cherché à obtenir un ensemble représentatif de celles-ci. De plus, nous avons appliqué des critères d'inclusion et d'exclusion bien définis et validés, en sélectionnant uniquement les bibliothèques pour augmenter la validité de nos résultats.

3.7 REMARQUES FINALES

Dans ce chapitre, nous avons présenté une cartographie de l'état de l'art et de l'état des pratiques concernant les niveaux de sévérité des logs. Il convient de souligner que cette étude est la première à évaluer systématiquement les définitions fournies pour les niveaux de sévérité des logs. À partir de trois sources distinctes (littérature évaluée par les pairs, bibliothèques de journalisation et points de vue des praticiens), nous avons analysé empiriquement les définitions, descriptions et documentations relatives à ces niveaux.

En résumé, nous avons analysé 19 niveaux de sévérité à partir de 42 études et 40 bibliothèques de journalisation. Nos résultats ont révélé une redondance et une similarité sémantique entre les différents niveaux. Cependant, ils ont également montré une tendance à converger vers un total de six niveaux. En outre, nous avons observé une cohérence dans l'ordre des différents niveaux entre les différentes bibliothèques, chaque niveau étant associé à des finalités spécifiques.

Ces résultats (cartographie, définitions et finalités) fournissent des preuves permettant de créer des lignes directrices pour le choix des niveaux de sévérité des logs, visant à améliorer la fiabilité des données. De plus, les créateurs de bibliothèques de journalisation peuvent utiliser cette étude pour optimiser leur processus de conception. Enfin, nos recommandations sur les niveaux de sévérité visent à renforcer la qualité des données de journalisation et à soutenir l'amélioration des pratiques dans la conception des bibliothèques de logs.

En outre, ces mêmes résultats serviront de fondement à la proposition d'heuristiques qui sera développée dans le chapitre 5.

CHAPITRE IV

AJUSTEMENTS DES NIVEAUX DE SÉVÉRITÉ DES LOGS

Ce chapitre se concentre sur l'analyse des ajustements des niveaux de sévérité des logs dans les systèmes logiciels *open-source*.

Dans l'étude de Yuan *et al.* (2012b), il a été observé qu'environ 33% des modifications apportées aux instructions de log sont réalisées après une analyse approfondie a posteriori, comme réaffirmé dans les études de Chen & Jiang (2017b) et Zeng *et al.* (2019). Parmi ces changements, 26% impliquent des ajustements de niveau de sévérité. Cela révèle que, loin de représenter une modification arbitraire, ces changements traduisent souvent un « affinage » ou un « réajustement », reflétant une réflexion plus poussée sur une décision initiale, ou la prise en compte de retours partagés, par exemple, dans les rapports d'incidents. Ces ajustements sont souvent nécessaires en raison des défis inhérents à la sélection des niveaux de sévérité des logs. Parmi ces défis, on trouve le manque de connaissances sur l'utilisation des journaux en production, la difficulté à évaluer la criticité des événements enregistrés et l'absence de directives pratiques pour guider les développeurs dans ce processus.

Notre objectif principal dans ce chapitre est de répondre à deux questions de recherche :

1. *Quels ajustements de niveaux de sévérité des logs se produisent entre les différentes versions d'un système ?*
2. *Pourquoi ces ajustements sont-ils effectués ?*

Pour répondre à ces questions, nous avons adopté une méthodologie en deux phases. La première phase, *descriptive*, examine les dépôts de systèmes Java *open-source* afin d'identifier les ajustements de sévérité entre différentes versions. Cette phase vise à fournir une compréhension globale des tendances et des motifs des ajustements effectués. La seconde

phase, *explicative*, se penche sur les raisons sous-jacentes à ces ajustements en analysant les descriptions et commentaires des incidents [58] des « *commits* » associés aux ajustements de sévérité. Cette analyse nous permet de comprendre les motivations des développeurs lorsqu'ils modifient les niveaux de sévérité des logs et d'identifier les facteurs qui guident ces ajustements.

Les résultats de ces analyses montrent que la plupart des ajustements de niveaux de sévérité se situent à l'intersection entre les environnements de développement et de production. Nous avons également observé une certaine mutabilité dans les niveaux de sévérité, qui peuvent être ajustés en fonction de la maturité du système et des besoins en production. En l'absence de directives formelles, l'expérience des développeurs et la théorie de la journalisation jouent un rôle central dans ces décisions. Ces résultats serviront de base pour la conception des heuristiques qui seront détaillés dans le chapitre suivant.

Ce chapitre est organisé comme suit : nous présentons notre méthodologie en deux phases dans la section 4.1. La section 4.2 présente les résultats de la phase descriptive (phase 1), et la section 4.3 présente les résultats de la phase explicative (phase 2). Dans la section 4.4, nous discutons des résultats des deux phases, et nous présentons les menaces à la validité dans la section 4.5. Enfin, la section 4.6 conclut le chapitre avec des remarques finales.

---

58. Dans cette thèse, les termes « rapport d'incident » et « incident » sont utilisés de manière interchangeable

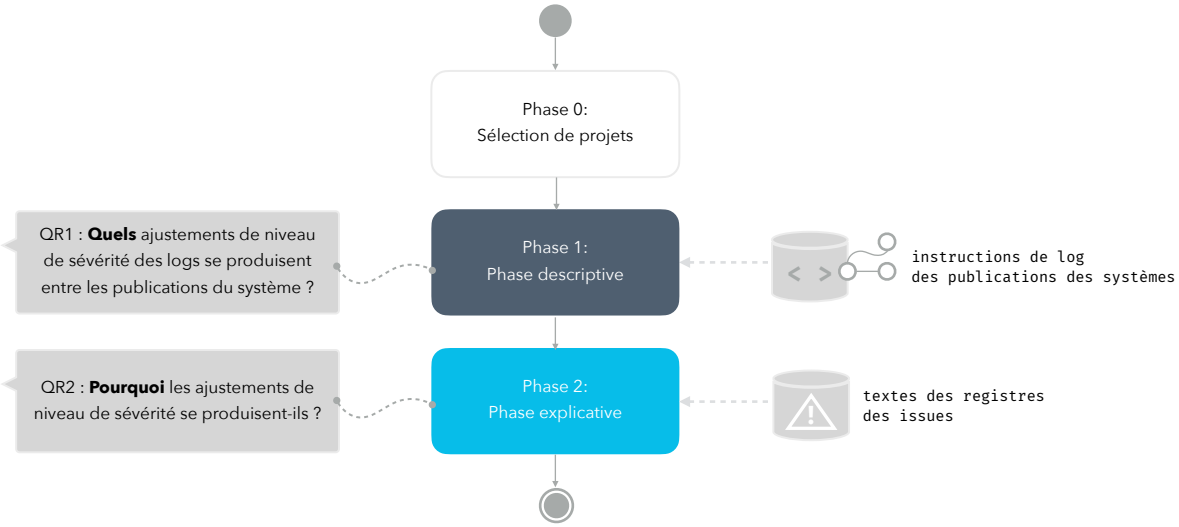

FIGURE 4.1 : Méthodologie en deux phases.
© Eduardo Mendes de Oliveira, 2024

4.1 MÉTHODOLOGIE

Notre stratégie utilise un grand nombre de publications (« *releases* »)[59] des logiciels sélectionnés pour construire une vue étendue des ajustements, en utilisant une méthodologie en deux phases (Figure 4.1), précédée d'une sélection initiale de projets (phase 0).

Dans la *phase descriptive*, nous examinons le niveau de sévérité des logs de manière statique (code source) pour trouver un aperçu de la distribution du niveau de sévérité des logs par publication et des ajustements présents dans chaque logiciel sélectionné. Dans la *phase explicative*, nous examinons les textes des incidents (« *issues* ») de *commit* de Jira[60] pour chaque ajustement du niveau de sévérité des logs associé à la première phase pour trouver les explications des développeurs.

59. Dans ce travail, nous utilisons les termes *version* et *publication* comme suit : *Publication* fait partie d'une *Version* : par exemple, la publication du système 2.1.2 est une publication de la version 2.*, tout comme la publication 3.3.4 est une publication de la version 3.*.

60. Jira (https://www.atlassian.com/software/jira) est un logiciel d'Atlassian qui s'intègre avec Git. Il donne plus de contexte aux *commits*, branches, tags et pull requests.

Ce travail fait partie d'une publication (Mendes *et al.*, 2024), dont le premier auteur est le candidat au doctorat. Afin de garantir l'objectivité dans l'analyse des incidents, des collaborateurs externes ont contribué à certaines étapes de cette phase, en particulier lors de la phase explicative.

4.1.1 SELECTION DES PROJETS (PHASE 0)

Nous décrivons d'abbord comment nous avons sélectionné les projets pour ce travail et la méthodologie en deux phases.

CRITÈRES DE SÉLECTION DES PROJETS

Pour sélectionner les systèmes logiciels de notre étude, nous avons appliqué les critères d'inclusion (SCI) et d'exclusion (SCE) suivants :

- **SCI1 :** il doit exister des preuves documentées dans la littérature académique (publications de journaux ou de conférences) d'ajustements dans les niveaux de sévérité des logs du logiciel.
- **SCI2 :** Le logiciel doit être *open-source*. Les logiciels *open-source* nous permettent d'accéder au code source, à la documentation publiquement disponible et au suivi des incidents.
- **SCI3 :** Le code source du logiciel doit être disponible sur GitHub. Cette accessibilité facilite l'extraction et l'analyse des ajustements de sévérité des logs.
- **SCI4 :** Le logiciel doit avoir des incidents enregistrés sur Jira : Jira consigne des informations détaillées sur les incidents logiciels, y compris les discussions des développeurs sur les *commits*.

- **SCE1 :** Incompatibilité des instructions de logs du logiciel avec les modèles utilisés par SLogAnalyzer. C'est l'outil que nous utilisons pour analyser les instructions de logs. Si les instructions de logs du logiciel sont incompatibles avec les modèles de SLogAnalyzer, il serait difficile de réaliser une analyse cohérente et précise.
- **SCE2 :** Activité limitée ou documentation éparse des incidents dans le dépôt Jira du logiciel : les projets actifs avec des incidents bien documentés sont nécessaires pour garantir un ensemble de données riche pour l'analyse. Les projets avec une activité limitée ou une documentation éparse ne fourniraient pas suffisamment de données pour une étude complète.

Pour répondre au premier critère d'inclusion, nous avons identifié au moins neuf études pertinentes qui documentent les ajustements des niveaux de sévérité des logs (Shang *et al.*, 2015; Chen & Jiang, 2017a,b; Zhao *et al.*, 2017; Li *et al.*, 2017b; Kabinna *et al.*, 2018; Li *et al.*, 2020a, 2021a; Zhang *et al.*, 2022). Après application de nos critères de sélection, environ 60 systèmes logiciels ont été jugés conformes aux conditions initiales d'inclusion. Cependant, en raison de la nature chronophage de la curation et de l'analyse manuelles, il n'était pas possible d'évaluer tous ces systèmes. Par conséquent, nous avons fait un choix en nous basant principalement sur l'impact et la popularité des projets, et nous avons finalement concentré notre analyse détaillée sur trois systèmes *open-source* largement utilisés. Il s'agit des projets suivants.

Hadoop. La bibliothèque logicielle Apache Hadoop [61] fournit un cadre pour le traitement de grands ensembles de données à travers des grappes d'ordinateurs en utilisant des paradigmes de programmation simples. Conçue pour s'étendre des serveurs individuels à des milliers, elle met l'accent sur le calcul local et le stockage. Au lieu de dépendre du matériel à haute

61. https://hadoop.apache.org/

disponibilité, Hadoop est conçu pour gérer les échecs au niveau de la couche application, assurant un service résilient même avec des ordinateurs potentiellement peu fiables.

HBase. Apache HBase [62] est une base de données non relationnelle, distribuée et versionnée qui offre un accès en lecture/écriture aléatoire en temps réel aux « *Big Data* ». Le projet vise à héberger de grandes tables – comprenant des milliards de lignes et des millions de colonnes – sur des grappes de matériel standard. Modélisé d'après le Bigtable de Google, HBase offre des capacités similaires à Bigtable en utilisant Hadoop et HDFS comme infrastructure de stockage.

Kafka. Apache Kafka [63] est une plateforme *open-source* spécialisée dans le streaming d'événements distribués. De nombreuses entreprises l'utilisent pour les *pipelines* de données, l'analyse de streaming, l'intégration de données et les applications critiques.

4.1.2 PHASE DESCRIPTIVE (PHASE 1)

Cette phase vise à décrire les ajustements dans les niveaux de sévérité des logs entre les versions des systèmes. Pour ce faire, nous avons obtenu des données de logs de versions adjacentes et les avons comparées. Le processus décrit peut être vu dans la figure 4.2 et nous fournirons une description détaillée ci-dessous.

Nous avons analysé uniquement les versions stables et écarté celles désignées comme candidates à la publication (« *release candidates* »). Le tableau 4.1 montre les systèmes sélectionnés et le nombre de versions et de publications qui font partie de cette étude.

62. https://hbase.apache.org/
63. https://kafka.apache.org/

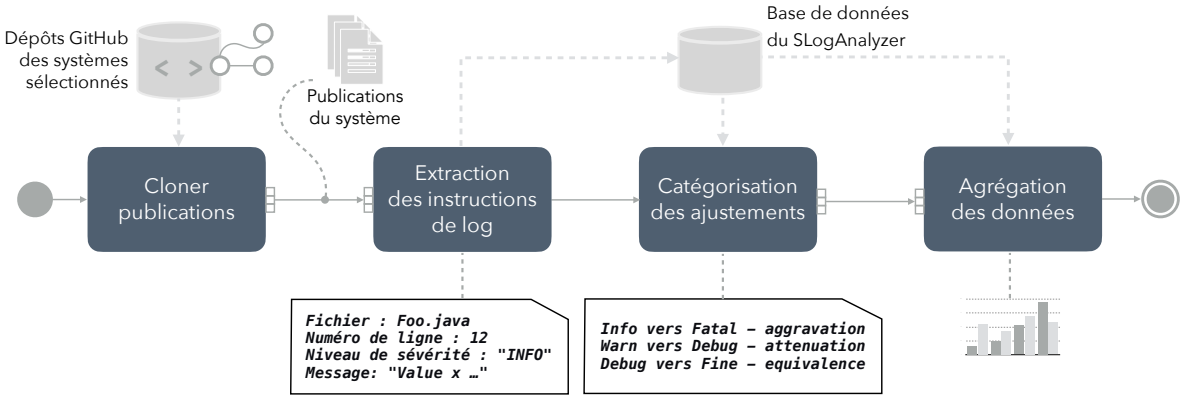

FIGURE 4.2 : Phase descriptive.
© Eduardo Mendes de Oliveira, 2024

TABLEAU 4.1 : Systèmes *open-source* sélectionnés.
© Eduardo Mendes de Oliveira, 2024

#	Système	Versions	Publications	Bibliothèque de journalisation
[S1]	Hadoop	[0.*, 1.*, 2.*, 3.*]	131	Commons Logging, SLF4J + Log4J
[S2]	Kafka	[0.*, 1.*, 2.*, 3.*]	60	Log4J
[S3]	HBase	[0.*, 1.*, 2.*]	204	Log4J
	Total	11	395	

Tout d'abord, nous avons cloné les publications des systèmes choisis en utilisant SLogAnalyzer (Vasconcellos, 2023). Nous avons récupéré tous les fichiers de chaque version sur GitHub pour extraire le code lié aux instructions de logs. Cette extraction incluait des détails tels que le *niveau de sévérité, le message, le nom du fichier, le numéro de ligne*, *un extrait de code environnant* pour le contexte et d'autres informations satellites. En nous concentrant sur les instructions de log, nous utilisons le SLogAnalyzer pour détecter les ajustements du niveau de sévérité des logs (aggravations, atténuations et équivalences) entre différentes publications des systèmes logiciels. Nous avons également ajouté un degré à chaque ajustement représentant la distance du *niveau de sévérité initial* au *niveau de sévérité final* ; les ajustements de niveaux équivalents reçoivent le degré 0 (zéro).

TABLEAU 4.2 : Explication des ajustements de niveau de sévérité des logs.
© Eduardo Mendes de Oliveira, 2024

Catégorie	Exemples d'ajustement	Degré
Atténuation	Debug > Trace	1
	Info > Trace	2
	Fatal > Info	3
Équivalence	Fine ≡ Debug	0
	Finer ≡ Debug	0
	Warning ≡ Warn	0
Aggravation	Trace < Debug	1
	Debug < Warn	2
	Info < Fatal	3

« > » désigne un ajustement vers un niveau de sévérité inférieur,
« < » désigne un ajustement vers un niveau de sévérité supérieur,
et « ≡ » désigne un ajustement vers un niveau de sévérité équivalent.

En utilisant les données traitées à travers SLogAnalyzer, nous avons agrégé les données des ajustements. Ce traitement nous a en outre permis de quantifier et de comprendre la distribution de ces ajustements à travers les différentes publications analysées. Nous avons également compilé un résumé des métriques pour chaque publication, détaillant le nombre total d'instructions de logs, les fichiers affectés par les ajustements, les messages uniques et la distribution des niveaux de sévérité des logs.

4.1.3 PHASE EXPLICATIVE (PHASE 2)

Sur la base des ajustements de niveau de sévérité des logs obtenus dans la phase précédente, nous avons analysé les textes des rapports des incidents Jira associés à ces ajustements pour enquêter et comprendre leurs causes. La relation entre les ajustements, les fichiers GitHub et les incidents Jira est représentée dans la figure 4.3.

FIGURE 4.3 : Relation entre ajustements, fichiers GitHub et incidents Jira. Un fichier sur GitHub peut contenir des ajustements et peut être associé à un ou plusieurs incidents dans Jira. Inversement, un incident Jira est lié à un ou plusieurs fichiers sur GitHub. © Eduardo Mendes de Oliveira, 2024

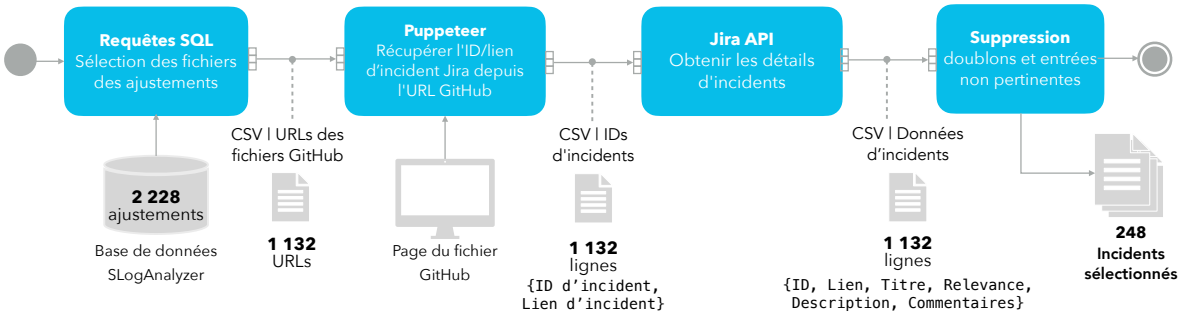

FIGURE 4.4 : Flux de processus pour la sélection d'incidents Jira à étudier. © Eduardo Mendes de Oliveira, 2024

PROCESSUS DE SÉLECTION D'INCIDENTS JIRA

Dans la figure 4.4, nous résumons les étapes de sélection d'incidents Jira à étudier, et ci-dessous, nous décrivons chacune d'elles.

À partir de la base de données SLogAnalyzer, en utilisant des requêtes SQL [64], nous avons filtré les fichiers de chaque système logiciel sélectionné pour obtenir l'ensemble correspondant aux ajustements de niveau de sévérité des logs. Ensuite, en utilisant Puppeteer [65], une bibliothèque JavasScript servant à automatiser l'interaction avec des pages web, nous avons

64. Disponible dans notre kit de reproductibilité (Mendes de Oliveira, 2024b)

65. https://pptr.dev/

récupéré l'*ID d'incident Jira* et son *lien associé* (URL) à partir de la page du fichier GitHub, pourvu qu'ils soient disponibles.

Dans l'étape suivante, nous avons utilisé l'API Jira [66] pour recueillir le *titre*, la *description*, et les *commentaires* pour chaque incident. Nous avons collecté uniquement les incidents résolus. Le champ *l'importance du résumé* est dérivé en appliquant une expression régulière au champ *commentaires* pour déterminer si l'incident contient des preuves d'ajustements du niveau de sévérité des logs. L'expression régulière utilisée à cette étape est « `(fine|trace|debug|info|warn|error|fatal |logging| log | logs | logger |log level|severity level|change level|change severity|slf4j|log4j| logback|noisy|verbose|spammy|overload|fail)` ». Elle s'explique ainsi :

- *« logging », « log », « logs », « logger », « log level »,* et *severity « level »* sont des mots courants qui peuvent être présents dans les discussions sur le logging
- des mots-clés comme *« fine », « trace », « debug », « info », « warn », « error »,* et *« fatal »* sont également inclus pour correspondre aux niveaux de sévérité
- les mots-clés pour identifier les noms des bibliothèques de logs utilisées par les systèmes sont *« slf4j », « log4j »,* et *« logback »*
- nous incluons également des expressions liées aux ajustements tels que *« change level »,* et *« change severity »*
- des mots-clés comme *« noisy », « verbose », « spammy », « overload »,* et *« fail »* représentent des mots communs dans les incidents de logs (Li *et al.*, 2020a; Yang *et al.*, 2021).

Après avoir collecté les commentaires, nous avons entrepris un processus de nettoyage des données qui comprenait l'extraction des balises de marquage de texte (*e.g.,*color, noformat,

66. https://developer.atlassian.com/cloud/jira/software/rest/

panel). Pour obtenir notre ensemble final d'incidents sélectionnés, nous avons manuellement supprimé tous les doublons, reconnaissant qu'un seul incident Jira peut être lié à plusieurs fichiers GitHub, et, finalement, nous avons filtré les entrées non pertinentes basées sur le champ de pertinence du résumé. Les étapes de ce flux de travail sont entièrement reproductibles et disponibles pour une exploration plus approfondie dans notre répertoire [67].

ANALYSE DES INCIDENTS JIRA

Nous avons réparti la conduite de l'analyse des incidents entre le doctorant et deux collaborateurs externes pour garantir l'objectivité du processus.

1. Identification des étiquettes et des explications. Le doctorant a mené l'ensemble du processus d'analyse des rapports d'incident des projets Hadoop, HBase et Kafka. Cela inclut l'identification des explications derrière les ajustements de niveau de sévérité, la formulation d'heuristiques potentielles et l'attribution systématique d'étiquettes aux cas. Afin de renforcer la fiabilité des résultats, 20% des cas ont été sélectionnés comme échantillon de validation. Cet échantillon a été analysé de manière indépendante par Marcelo Vasconcellos, étudiant de maîtrise en informatique à l'UQAC, suivant la méthodologie définie par le doctorant. Cette étape a permis d'établir une compréhension commune des critères utilisés et de confirmer la reproductibilité de l'analyse. Le doctorant a ensuite poursuivi l'analyse de l'ensemble des cas restants. Les quelques divergences observées ont été discutées jusqu'à atteindre un consensus. En cas de désaccord persistant, un second relecteur externe a été consulté pour arbitrer. Voici quelques exemples d'étiquettes et d'explications :

67. https://figshare.com/projects/Adjustments_Research/212807

- Étiquettes : *« Non lié à l'ajustement », « N'explique pas la raison de l'ajustement », « ERROR verbeux », « Événement géré par le client », « Masquage de DEBUG en INFO », « N'affecte pas le service normal »*.
- Explications : *« Info causant beaucoup de bruit », « Peut probablement être changé pour journaliser au niveau DEBUG à la place », « il ne prend aucune action mais est journalisé en mode erreur », « Passage des API de journalisation à slf4j dans hadoop-mapreduce-client-app »*.

2. Détermination des principales catégories. Pour dériver les cinq principales catégories, nous avons suivi une approche pour affiner et regrouper de manière itérative les étiquettes et les explications. Les étapes étaient les suivantes :

- Le doctorant a identifié des étiquettes et des explications ayant une plus grande fréquence et pertinence pour les ajustements de niveau de sévérité et a proposé des catégories initiales. Exemples de catégories initiales : *« Niveaux de sévérité inappropriés pour la situation », « Problèmes silencieux », « Incohérences dans la pratique du projet », « Exceptions extraordinaires », « Journalisation verbeuse », « Classification erronée »*.
- Affinement par discussion : Le doctorant et un collaborateur ont regroupé les résultats, identifiant les similitudes et les principales tendances dans l'ajustement des niveaux de sévérité par rapport aux catégories initiales. Par exemple, la plupart des ajustements liés aux entrées de journal répétitives étaient motivés davantage par la prise de conscience de l'énorme quantité de données produites par le système (*« Journalisation verbeuse »*) que par le contenu généré par l'instruction de journal. Dans les cas d'ajustements comme celui-ci, basés sur l'intuition et la pratique, nous les avons regroupés sous la catégorie Expérience. Ce processus itératif a aidé à fusionner les thèmes chevauchants et à diviser les groupes larges en catégories plus spécifiques lorsque cela était nécessaire.

TABLEAU 4.3 : Données d'extraction des fichiers, messages distincts et instructions de log pour chaque système

Système	Fichiers	Messages distincts	Instructions de log
Hadoop	2 701	16 877	32 046
Hbase	1 568	8 755	17 278
Kafka	410	2 769	5 557
Total	4 679	28 401	54 881

- Validation contre les incidents : Le doctorant et un collaborateur ont validé chaque catégorie en la comparant à l'ensemble des incidents sélectionés pour s'assurer qu'elle représentait avec précision les raisons sous-jacentes des ajustements de sévérité.

À la fin de cette analyse, nous avons obtenu un ensemble d'incidents Jira étiquetés selon leur pertinence par rapport à l'ajustement de la sévérité et un aperçu des principales raisons derrière ces ajustements.

4.2 RÉSULTATS DE LA PHASE DESCRIPTIVE (PHASE 1)

Cette section décrit les résultats quantitatifs de notre analyse des instructions de log à travers 395 versions des projets sélectionnés. Dans cette phase, nous avons analysé plus de 54 000 instructions de log réparties dans près de 5 000 fichiers source (Tableau 4.3) pour établir un panorama des niveaux de sévérité des logs et de leurs ajustements. Nous présentons la répartition des niveaux de sévérité tout au long des versions, les données détaillées sur les ajustements de sévérité, et les totaux pour les catégories et chaque ajustement trouvé.

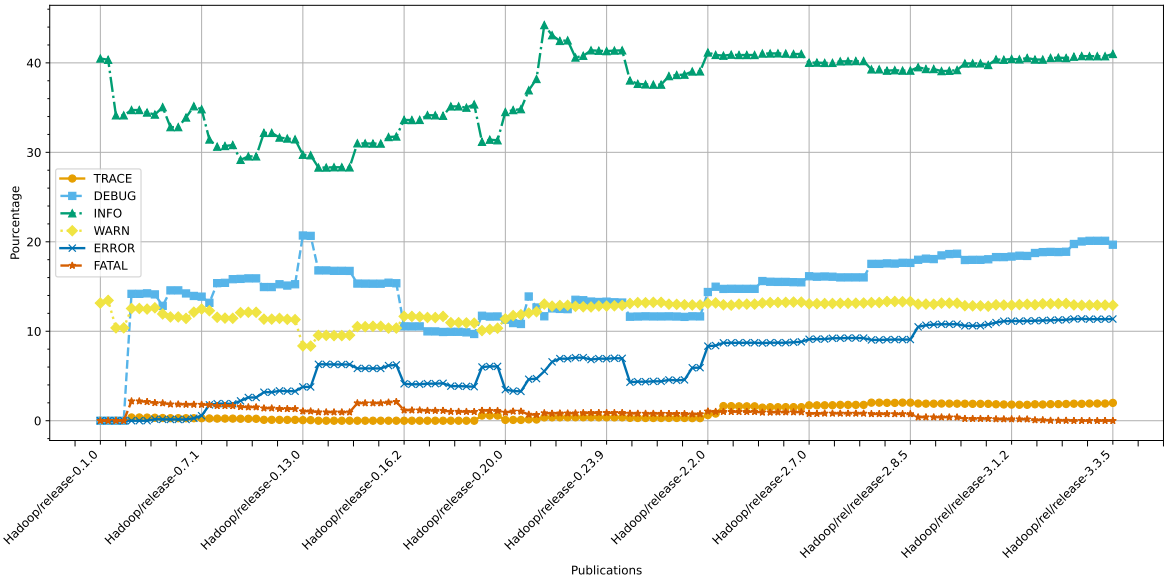

FIGURE 4.5 : Répartition des niveaux de sévérité des logs sur Hadoop.
Données provenant des 130 versions de Hadoop,
avec 11 étiquettes affichées pour une meilleure clarté.
© Eduardo Mendes de Oliveira, 2024

4.2.1 DISTRIBUTION DU NIVEAU DE SÉVÉRITÉ DES LOGS PAR PUBLICATION

Les graphiques suivants (Figures 4.5, 4.6, et 4.7) fournissent un aperçu de l'évolution de la répartition des niveaux de sévérité des logs dans les systèmes analysés au fil du temps en pourcentages. Ces pourcentages représentent la proportion de chaque niveau de sévérité par rapport au nombre total d'entrées de log pour chaque version. Par exemple, une valeur de 40 pour le niveau de sévérité *Debug* dans une version particulière signifie que 40% de toutes les entrées de log pour cette version étaient classées comme *Debug*. Ils montrent la fréquence de chaque niveau de sévérité à travers les versions sélectionnées, ce qui aide à comprendre les motifs et les tendances. Les points principaux sont :

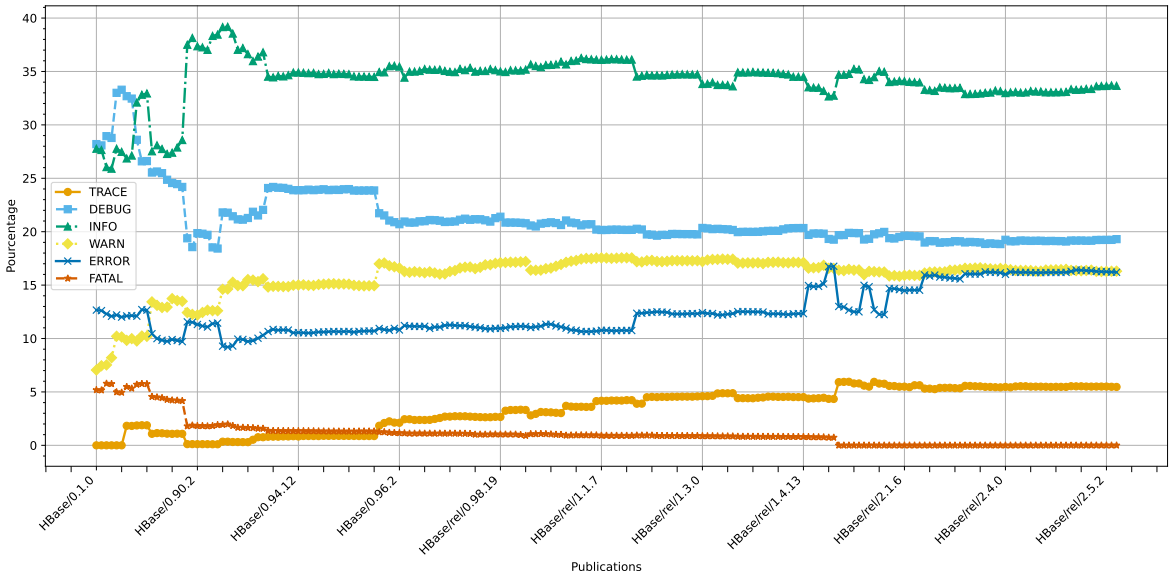

FIGURE 4.6 : Répartition des niveaux de sévérité des logs sur HBase.
Données provenant des 203 versions de HBase,
avec 11 étiquettes affichées pour une meilleure clarté.
© Eduardo Mendes de Oliveira, 2024

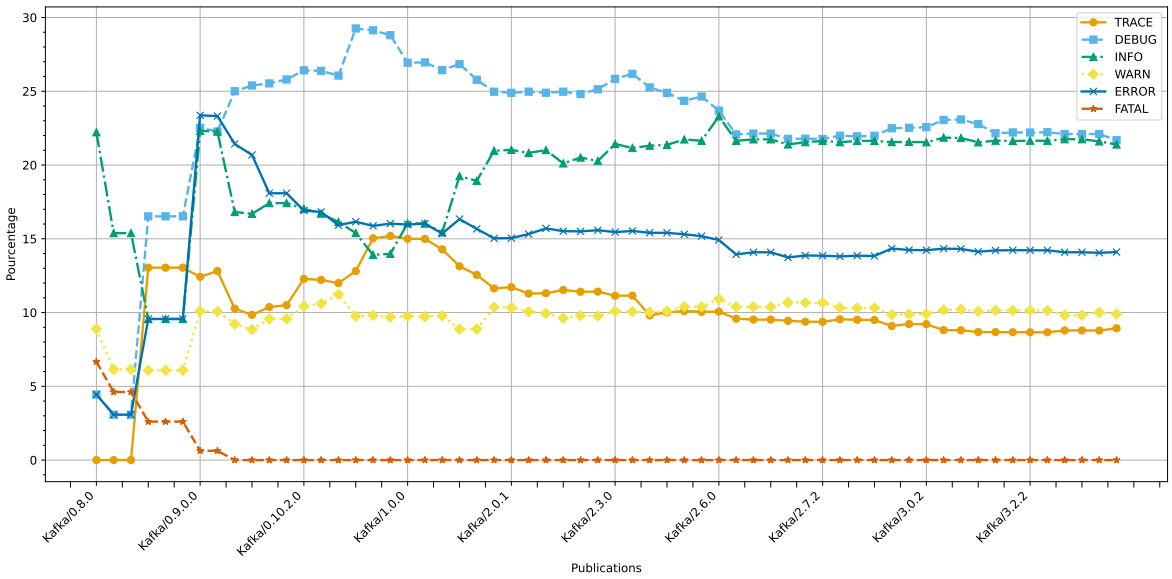

FIGURE 4.7 : Répartition des niveaux de sévérité des logs sur Kafka.
Données provenant des 49 versions de Kafka,
avec 10 étiquettes affichées pour une meilleure clarté.
© Eduardo Mendes de Oliveira, 2024

- les instructions de logs classifiées avec le niveau de sévérité *Info* prévalent dans la plupart des publications, soulignant leur rôle dans l'enregistrement du comportement standard des systèmes. L'exception est dans les publications de Kafka, où la fréquence de la sévérité *Info* est similaire à celle des instructions *Debug* ;
- les instructions de log *Debug* arrivent en deuxième position et suggèrent l'importance de la journalisation comme outil de débogage pour les développeurs ;
- en troisième position, les instructions de log *Warn* enregistrent une fréquence relativement constante, suggérant une certaine régularité dans les incidents ou événements nécessitant une attention ;
- le niveau de log *Error* est le plus grave dans les publications les plus récentes des trois systèmes. Dans Hadoop et HBase, les instructions de log qui utilisent ce niveau forment le quatrième plus grand groupe, tandis que dans Kafka, il est le troisième. En regardant les deux premiers systèmes, cela suggère que les systèmes sont sains. Dans Kafka, d'autre part, cela peut suggérer un nombre plus élevé de comportements susceptibles de défaillance ;
- *Trace*, bien que moins répandu dans Hadoop et HBase, est plus présent dans les publications de Kafka que dans les autres systèmes ;
- les logs *Fatal* sont le niveau de sévérité des logs le moins présent, ce qui peut suggérer l'intégrité des opérations du système, ou que la bibliothèque de logs utilisée n'a pas ce niveau de sévérité. C'est le cas avec Hadoop, qui utilise SLF4J, présentant 5 niveaux de sévérité, ne comprenant pas *Fatal* [68], comme une façade pour la bibliothèque Log4J. Auparavant, la façade était Commons Logging qui a 6 niveaux de sévérité, y compris *Fatal*.

68. www.slf4j.org/api/src-html/org/slf4j/spi/LocationAwareLogger.html

4.2.2 APERÇU DES AJUSTEMENTS DU NIVEAU DE SÉVÉRITÉ DES LOGS

L'analyse des ajustements du niveau de sévérité des logs révèle la nature dynamique des pratiques de journalisation à travers différentes versions de logiciels. Nous catégorisons ces ajustements en trois types principaux : *aggravation*, *atténuation* et *équivalence*. Chaque catégorie reflète un choix stratégique des développeurs, impactant l'informativité et le diagnostic du log. Ci-dessous, nous fournissons une analyse présentant la proportion et l'impact de chaque type d'ajustement sur les pratiques de journalisation.

Nous avons identifié **35 types d'ajustements** au total : 14 types d'atténuation, 16 types d'aggravation et 5 types d'équivalence (Tableau 4.4). Lors de l'analyse des données, nous avons trouvé **plus d'instances d'atténuation que d'aggravation**. Spécifiquement, sur les 2 228 items que nous avons examinés, nous avons vu 1 333 occurrences d'atténuation (environ 60%), 760 occurrences d'aggravation (environ 34%) et 135 occurrences d'équivalence (environ 6%). En ce qui concerne le degré d'ajustement, les ajustements d'un degré représentent 67,50% du total, tandis que les ajustements de deux degrés comptent pour 21,72% (Figure 4.9).

Les ajustements principaux dans les niveaux de logs, particulièrement en termes d'atténuation et d'aggravation, sont notablement prévalents entre les niveaux *Debug - Trace* et *Info - Debug*, comme illustré dans la figure 4.8. Ces ajustements sont caractérisés par un degré de 1, c'est-à-dire la distance entre deux niveaux de sévérité consécutifs.

HADOOP

Nous avons évalué 130 publications de quatre ensembles de versions de Hadoop : 67 de la version *0.**, 10 de la version *1.**, 33 de la version *2.**, et 20 de la version *3.** (Tableau 4.5).

TABLEAU 4.4 : Catégories d'ajustements sur le niveau de sévérité.

#	Catégorie	Ajustement	Degré	Total
1	Atténuation	Debug > Trace	1	295
2		Info > Trace	2	57
3		Info > Debug	1	463
4		Info > Fine	1	2
5		Warn > Trace	3	10
6		Warn > Debug	2	93
7		Warn > Info	2	64
8		Error > Trace	4	4
9		Error > Debug	3	46
10		Error > Info	2	41
11		Error > Warn	1	66
12		Fatal > Info	3	1
13		Fatal > Warn	2	7
14		Fatal > Error	1	184
15	Équivalence	Fine ≡ Debug	0	55
16		Finer ≡ Debug	0	16
17		Finest ≡ Trace	0	2
18		Warning ≡ Warn	0	51
19		Severe ≡ Fatal	0	11
20	Aggravation	Trace < Debug	1	133
21		Trace < Info	2	24
22		Trace < Warn	3	9
23		Trace < Error	4	3
24		Debug < Info	1	282
25		Debug < Warn	2	69
26		Debug < Error	3	29
27		Fine < Info	1	7
28		Finer < Info	1	3
29		Info < Warn	2	74
30		Info < Warning	1	2
31		Info < Error	2	48
32		Info < Fatal	3	3
33		Warn < Error	1	52
34		Warn < Fatal	2	7
35		Error < Fatal	1	15
			Total	2 228

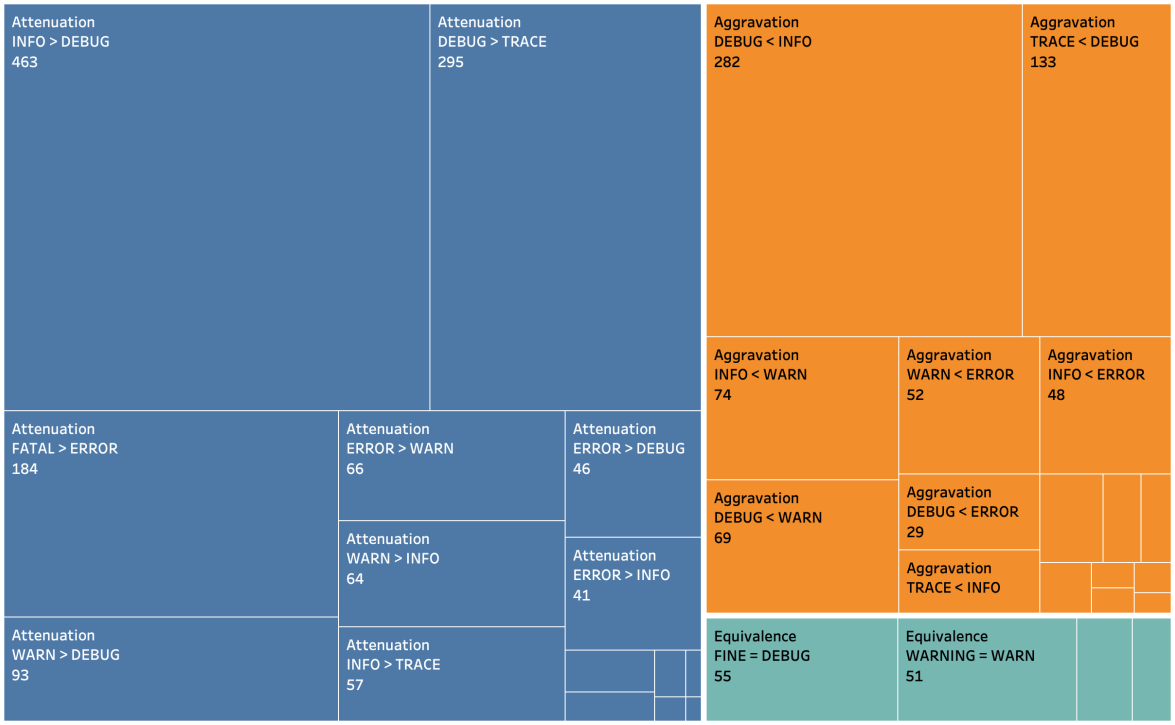

FIGURE 4.8 : Ajustements des niveaux de sévérité sur les systèmes sélectionnés.
© Eduardo Mendes de Oliveira, 2024

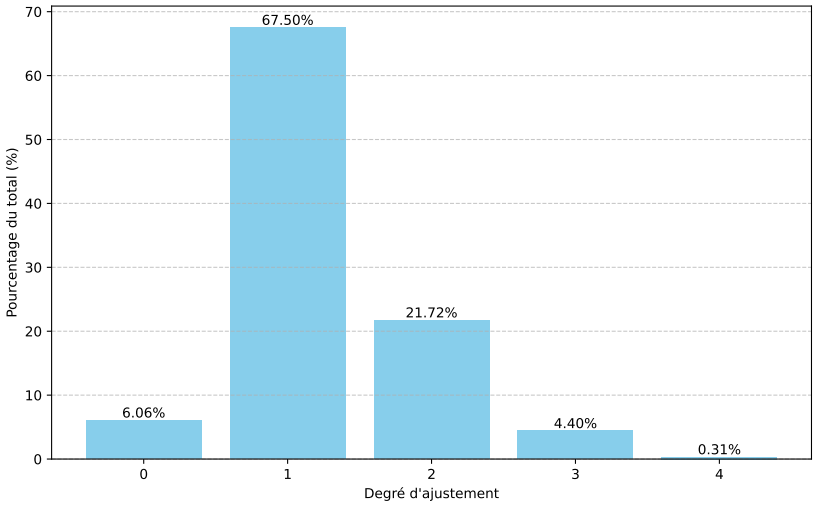

FIGURE 4.9 : Pourcentage d'ajustement par degré.
© Eduardo Mendes de Oliveira, 2024

TABLEAU 4.5 : Ajustements sur Hadoop.
© Eduardo Mendes de Oliveira, 2024

Version	Publications	Ajustements	Atténuation	Équivalence	Aggravation
0.*	67	395	127	135	133
1.*	10	13	6	0	7
2.*	33	379	290	0	89
3.*	20	250	164	0	86
Total	**130**	**1 037**	**587**	**135**	**315**

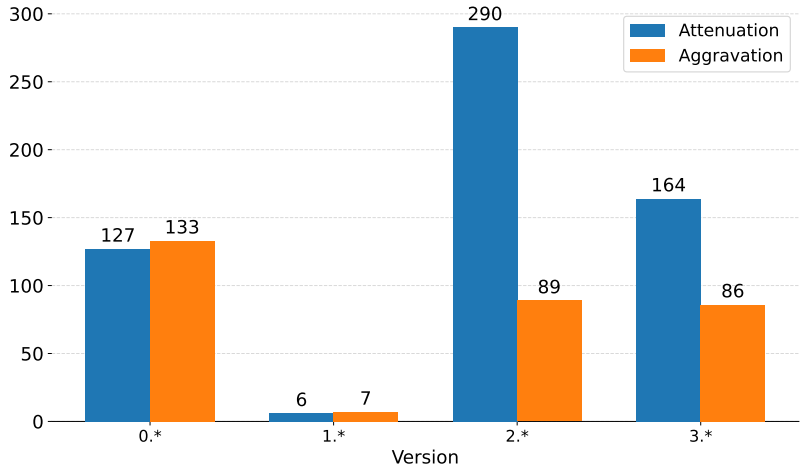

FIGURE 4.10 : Atténuations et aggravations des niveaux de sévérité sur Hadoop.
© Eduardo Mendes de Oliveira, 2024

Les 1 037 ajustements de Hadoop sont 587 atténuations, 315 aggravations et 135 équivalences. Les équivalences sont principalement causées par changement de la bibliothèque de journalisation (Voir l'annexe A pour la liste complète des changements).

En se basant sur la figure 4.10, nous avons surtout remarqué des ajustements d'atténuation dans le niveau de sévérité des logs à mesure que les versions de Hadoop évoluaient. Les ajustements avec le plus d'occurrences sont *Info* > *Debug* (degré 1, 303 occurrences), *Fatal* >

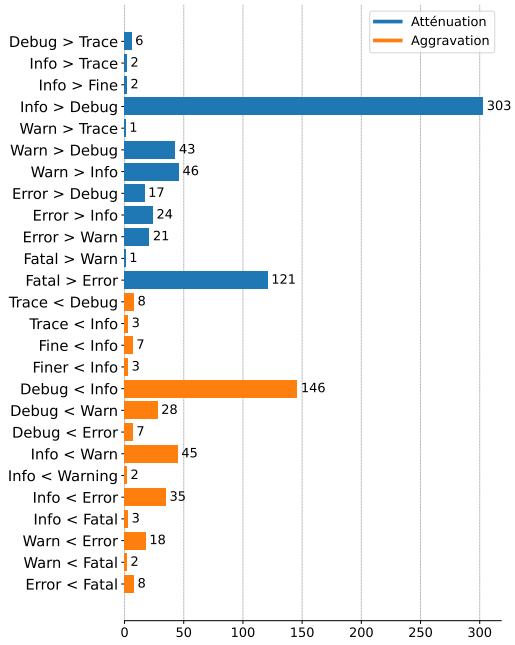

FIGURE 4.11 : Occurrences d'ajustement sur Hadoop.
© Eduardo Mendes de Oliveira, 2024

Error (degré 1, 121 occurrences) et *Debug* < *Info* (degré 1, 146 occurrences) (Figure 4.11). L'ajustement *Fatal* > *Error* (degré 1) se produit uniquement dans les versions 2.* et 3.*.

HBASE

Nous avons évalué 203 publications de trois groupes de versions de HBase : 89 des versions *0.**, 58 des versions *1.** et 56 des versions *2.** (Tableau 4.6).

Les 972 ajustements de HBase sont 637 atténuations, 335 aggravations et aucune équivalence. En se basant sur la figure 4.12, nous remarquons que dans toutes les versions, il y a plus d'atténuations que d'aggravation dans les ajustements du niveau de sévérité des logs. Les ajustements avec le plus d'occurrences sont *Debug* > *Trace* (degré 1, 259 occurrences),

TABLEAU 4.6 : Ajustements sur HBase.
© Eduardo Mendes de Oliveira, 2024

Version	Publications	Ajustements	Atténuation	Équivalence	Aggravation
0.*	89	330	224	0	106
1.*	58	271	178	0	93
2.*	56	371	235	0	136
Total	**203**	**972**	**637**	**0**	**335**

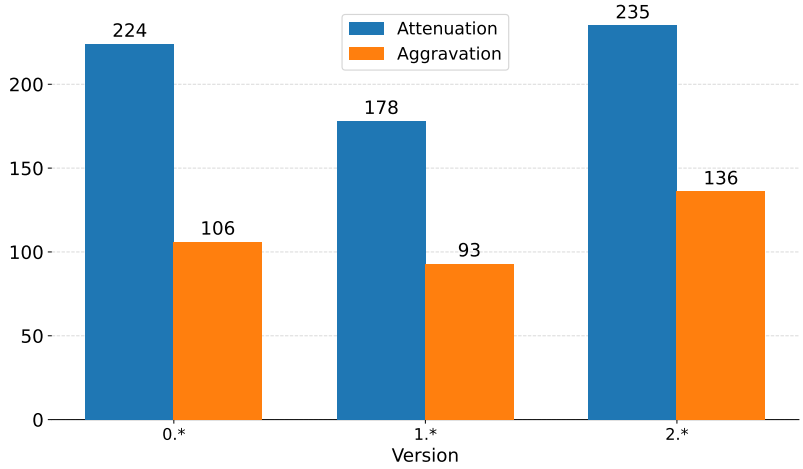

FIGURE 4.12 : Atténuations et aggravations des niveaux de sévérité sur HBase.
© Eduardo Mendes de Oliveira, 2024

Info > *Debug* (degré 1, 117 occurrences), *Trace* < *Debug* (degré 1, 96 occurrences), *Debug* < *Info* (degré 1, 90 occurrences) et *Fatal* > *Error* (degré 1, 63 occurrences) (Figure 4.13).

KAFKA

Nous avons évalué 49 publications de quatre ensembles de versions de Kafka : 7 de la version *0.**, 5 de la version *1.**, 23 de la version *2.** et 14 de la version *3.** (Tableau 4.7).

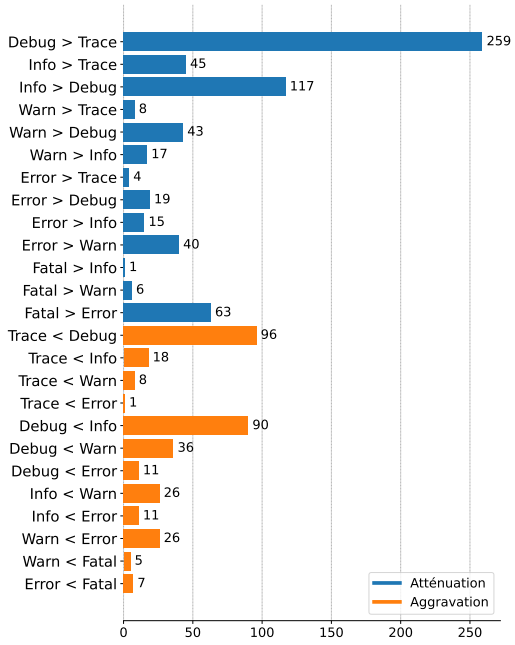

FIGURE 4.13 : Occurrences d'ajustement sur HBase.
© Eduardo Mendes de Oliveira, 2024

Les 219 ajustements de Kafka sont 109 atténuations, 110 aggravations et aucune équivalence. En observant la figure 4.14, nous remarquons que dans le premier ensemble de versions (0.*), il y a plus d'atténuations et dans les trois derniers ensembles de versions, il y a plus d'ajustements d'aggravations du niveau de sévérité des logs. Les ajustements avec le plus d'occurrences sont *Debug* < *Info* (degré 1, 46 occurrences), *Info* > *Debug* (degré 1, 43 occurrences), *Debug* > *Trace* (degré 1, 30 occurrences) et *Trace* < *Debug* (degré 1, 29 occurrences) (Figure 4.15).

4.3 RÉSULTATS DE LA PHASE EXPLICATIVE (PHASE 2)

Dans cette section, nous présentons les résultats de la phase explicative de notre étude, en nous concentrant sur les raisons des ajustements des niveaux de sévérité dans les journaux de

TABLEAU 4.7 : Ajustements sur Kafka.
© Eduardo Mendes de Oliveira, 2024

Version	Publications	Ajustements	Atténuation	Équivalence	Aggravation
0.*	7	62	52	0	10
1.*	5	22	4	0	18
2.*	23	103	42	0	61
3.*	14	32	11	0	21
Total	**49**	**219**	**109**	0	**110**

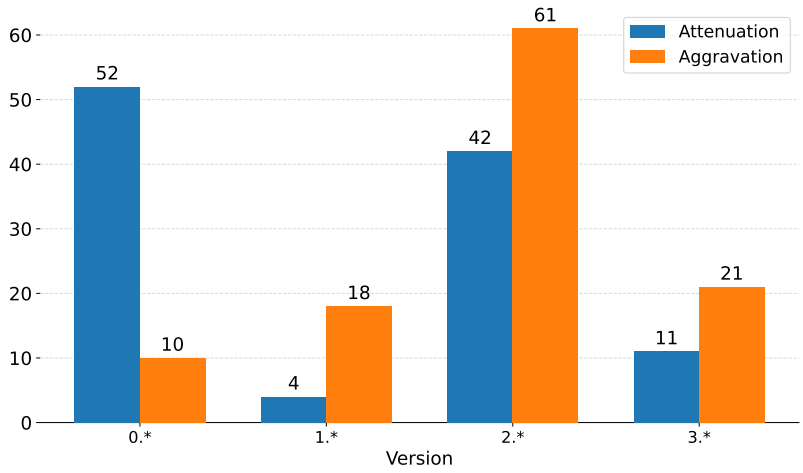

FIGURE 4.14 : Atténuations et aggravations des niveaux de sévérité sur Kafka.
© Eduardo Mendes de Oliveira, 2024

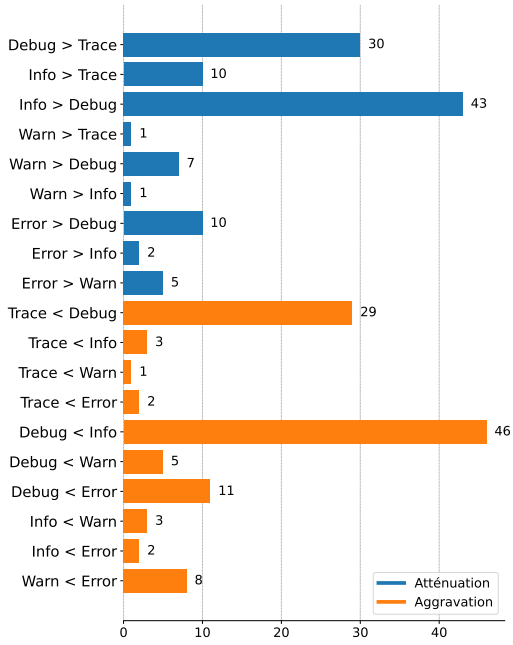

FIGURE 4.15 : Occurrences d'ajustement sur Kafka.
© Eduardo Mendes de Oliveira, 2024

Hadoop, HBase et Kafka. Premièrement, nous présentons les données quantitatives analysées, allant des ajustements identifiés lors de la phase 1 aux incidents qui ont effectivement contribué des données à l'étude. Les sujets sont organisés en cinq catégories principales d'ajustements :

- basés sur les principes fondamentaux ;
- de pratique historique ;
- dictés par l'expérience ;
- basés sur des directives ;
- et provoqués par un changement de bibliothèque de journalisation.

Chaque catégorie est soutenue par des rapports d'incidents spécifiques illustrant la logique derrière ces ajustements, fournissant un aperçu des divers facteurs influençant les pratiques de journalisation.

TABLEAU 4.8 : Nombres d'extraction de la phase explicative.
© Eduardo Mendes de Oliveira, 2024

Métriques	Hadoop	HBase	Kafka	Total
Ajustements	1 037	972	219	2 228
Fichier avec des ajustements	502	515	115	1 132
Fichiers avec des incidents Jira associés	483	514	99	1 096
Fichiers sans incidents Jira associé	19	1	16	36
Incidents Jira (sans doublons)	366	326	85	776
Incidents Jira à investiguer	135	94	19	248
Incidents Jira sur les ajustements	77	36	6	119

La sous-section finale synthétise les intuitions tirées des données analysées et propose des observations préliminaires et des heuristiques potentielles qui pourraient guider les futurs ajustements de sévérité des logs.

4.3.1 APERÇU DES DONNÉES

Les 2 228 ajustements de la phase 1 peuvent être retracés à 1 132 fichiers à travers les systèmes sélectionnés, soit 502 pour Hadoop, 515 pour HBase, et 115 pour Kafka. Lors de la recherche des incidents associés à ces fichiers, en éliminant les doublons et les entrées non pertinentes, on obtient 248 incidents Jira : 135 pour Hadoop, 94 pour HBase et 19 pour Kafka. Après une analyse manuelle, excluant les incidents non liés aux ajustements ou ceux qui, bien qu'étant liés, n'expliquaient pas la raison de l'ajustement, notre ensemble final se compose de 119 rapports d'incidents liés aux ajustements, avec 77 provenant de Hadoop (Annexe A), 36 de HBase (Annexe B), et six de Kafka (Annexe C). Nous avons détaillé ces chiffres dans le tableau 4.8.

Tout au long de notre enquête, nous avons identifié cinq catégories distinctes d'ajustements :

- **Ajustements des principes fondamentaux :** Ces ajustements se basent sur les principes fondamentaux identifiés dans notre synthèse des niveaux de sévérité des logs (chapitre 3, section 3.4), c'est-à-dire les définitions consolidées et les finalités associées à chaque niveau. Ils visent à corriger des incohérences entre le niveau attribué à une instruction de log et l'intention sémantique attendue. Par exemple, une instruction peut être marquée à tort comme critique (*e.g., Error* ou *Fatal*), alors qu'elle devrait plutôt refléter une situation normale (niveau *Info*). Ce type d'ajustement est essentiel pour assurer l'adéquation entre la gravité perçue d'un événement et sa réelle importance.
- **Ajustements des pratiques historiques :** Les ajustements de cette catégorie proviennent d'une analyse de l'historique du code et des incidents du projet. Par exemple, si certains niveaux de sévérité des logs ont été ajustés dans des scénarios spécifiques dans le passé, des ajustements similaires pourraient être appliqués dans les situations actuelles pour maintenir la cohérence et l'adhésion aux pratiques historiques du projet.
- **Ajustements guidés par l'expérience :** Contrairement aux catégories précédentes, ces ajustements résultent de l'expérience pratique des développeurs et des opérateurs système dans le dépannage et le diagnostic des défauts. Ils reflètent une compréhension approfondie des besoins réels en logs, lorsque des informations spécifiques peuvent devenir plus critiques au fil du temps ou dans certains contextes. Par exemple, une instruction log initialement considérée comme de faible niveau de sévérité (*Trace* ou *Debug*) peut être élevée à un niveau supérieur (*Info* ou *Warn*) en fonction de l'expérience, sachant que ces informations sont souvent précieuses pour le dépannage d'incidents spécifiques.
- **Ajustements basés sur des directives :** Ici, le projet prescrit des ajustements par la documentation officielle, qui fournit des directives sur les pratiques de journalisation.

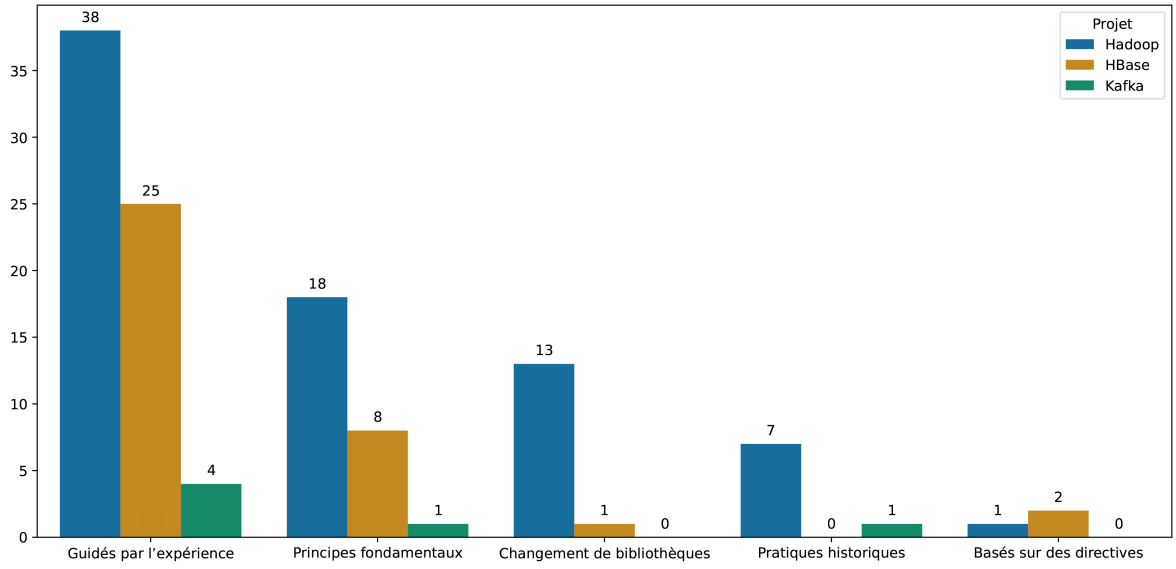

FIGURE 4.16 : Catégories d'ajustement par système.
© Eduardo Mendes de Oliveira, 2024

- **Ajustements dus au changement de bibliothèques de logs :** Ajustements causés par le changement de bibliothèque de journalisation avec des ensembles de niveaux de sévérité différents.

La figure 4.16 présente un résumé des incidents Jira associés aux catégories d'ajustement pour chaque système sélectionné. Nous les discutons dans ce qui suit.

4.3.2 AJUSTEMENTS DES PRINCIPES FONDAMENTAUX

Cette catégorie couvre les ajustements liés à un manque de respect pour les principes fondamentaux de chaque niveau de sévérité. Elle inclut également les cas qui montrent un manque de compréhension de l'importance d'un événement ou d'une information dans les activités d'analyse du comportement du système lors du choix du niveau de sévérité initial.

CONCEPTION ERRONÉE DES NIVEAUX DE SÉVÉRITÉ DES LOGS

Dans les systèmes sélectionnés, nous avons fréquemment rencontré des preuves d'erreurs dans le choix des niveaux de sévérité des logs. Cette preuve est présente dans les discussions expliquant pourquoi l'instruction de log ne correspond pas au niveau de sévérité initialement attribué. Par exemple, nous avons trouvé des cas où un ajustement du niveau de sévérité pour *Info* se produit délibérément, uniquement pour faciliter le débogage sur YARN-1839 :

> *Debug* < *Info* [69]
> *« Changé certains niveaux de débogage liés à NMToken en info afin de faciliter le débogage ».*

Dans un autre incident (HDFS-14759), le niveau de sévérité d'une instruction de log contenant uniquement des valeurs d'état est réinitialisé à *Debug* après avoir été changé en *Info* sans discussion préalable, indiquant l'inadéquation du premier ajustement du niveau de sévérité.

Nous avons également trouvé des valeurs très spécifiques enregistrées au niveau *Info*, ce qui rappelle un type de surdétail (MAPREDUCE-5766, YARN-1022), et même des valeurs qui ne semblent manquer à aucun processus logiciel, comme décrit dans le incident YARN-10369 :

> *Info* > *Debug*
> *« Nous avons changé cela en DEBUG en interne il y a des années et cela ne nous a pas manqué ».*

69. Chaque citation dans cette section est précédée du type d'ajustement du niveau de sévérité dont elle discute.

Certaines des discussions ont révélé l'intention de maintenir certains niveaux de sévérité pour l'analyse ou le débogage malgré qu'ils soient inappropriés pour la situation, comme le présente l'instruction suivante d'un ajustement d'atténuation sur MAPREDUCE-5766 :

> *Info > Debug*
> *« Je comprends le désir de déplacer les demandes de ping et les messages de la JVM demandant des tâches en debug, mais voulons-nous vraiment déplacer les mises à jour de statut en debug ? Ils sont particulièrement utiles pour le débogage . . . »* [70]

Tous ces incidents illustrent le dilemme des développeurs dans la catégorisation de la sévérité des logs : équilibrer le besoin d'informations détaillées pendant le débogage avec le risque d'encombrer le log avec trop de détails.

Cependant, cette méconnaissance des niveaux de sévérité des logs ne se limite pas aux niveaux *Debug* et *Info*. Elle se manifeste également lorsque les développeurs classent des messages bénins comme des incidents (HDFS-6998) ou des événements d'erreur que le client peut gérer et qui ne nécessitent pas d'action immédiate, mais qui sont classés comme des erreurs (*Error*) au lieu d'avertissements (*Warn*), ce qui serait plus approprié (HADOOP-10274, HDFS-14760) :

> *Error > Warning*
> *« . . . à ce stade, notre installation hdfs veut s'assurer qu'aucune 'ERROR' n'est consignée s'il ne s'agit pas vraiment d'une erreur qui devrait/pourrait être actionnée ».* (HDFS-14760)

70. Veuillez noter que les citations de Jira dans cet article sont des reproductions fidèles des textes originaux. En tant que telles, elles peuvent contenir des erreurs grammaticales et des constructions de langage maladroites inhérentes au matériel source.

INFORMATIONS PRÉCIEUSES

Nous avons identifié des ajustements lorsque des informations précieuses de dépannage étaient cachées du journal de l'environnement de production en raison de l'utilisation d'un niveau *Debug*, comme le montre HADOOP-12789 :

> *Debug < Info*
> *« Savoir exactement quel classpath ApplicationClassLoader a est une information cruciale pour le dépannage. Nous devrions le journaliser au niveau INFO ».*

Cette phrase montre un manque de compréhension du comportement du système et de l'importance de l'information lorsque le niveau de sévérité a été initialement choisi. Dans le même contexte de dépannage, les événements provoquant des erreurs d'exécution n'ont pas laissé d'indices clairs sur le problème survenu, ce qui a conduit à une escalade de *Warn* à *Error* pour faciliter la compréhension des comportements inattendus :

> *Warn < Error*
> *« La socket est fermée silencieusement sans qu'aucune information ne soit donnée, et une exception incompréhensible sera lancée. ».* (HADOOP-1034)

Identifier quelles informations sont plus critiques que d'autres est important pour éviter des conséquences non intentionnelles dans les données de log. C'est le cas avec HADOOP-14987, un incident qui a été ouvert pour améliorer la journalisation à des fins de dépannage. Dans cet incident, les développeurs parviennent à un consensus sur l'ajustement d'une seule instruction de log de *Debug* à *Info* dans un groupe d'instructions *Debug*. À première vue, l'instruction de log pourrait être confondue avec une autre entrée de niveau *Debug* en raison de son

contexte environnant. Cependant, cet ajustement ciblé ajoute des informations significatives au journal de production sans créer de verbosité excessive.

4.3.3 AJUSTEMENTS DES PRATIQUES HISTORIQUES

De nos jours, les systèmes de gestion des incidents (IMS) comme Jira jouent un rôle essentiel dans la création d'une base de connaissances en compilant les changements de code pendant le développement et l'évolution des systèmes logiciels. Cette base de données devient un outil précieux pour les développeurs, non seulement pour les consultations lors de l'implémentation de nouveaux changements, mais aussi pour soutenir les discussions concernant les pratiques historiques.

Ce principe s'applique également à l'ajustement des niveaux de sévérité des logs. Généralement, les discussions dans ce contexte sont plus brèves, car elles reflètent souvent d'incidents précédemment débattus. Ci-dessous, nous mettons en évidence des extraits de certains incidents qui illustrent ce *modus operandi*.

Dans les deux premières situations, nous trouvons une comparaison directe de la même situation avec différents niveaux de sévérité :

Debug < Error

« … et chacun d'eux contient une instruction de log, la plupart d'entre eux définissent le niveau de journalisation à ERROR. Cependant, lorsqu'il capture RuntimeErrorException à la ligne 348 (r1839798), le niveau de journalisation de l'instruction de log dans cette clause catch est DEBUG … » (HADOOP-15942)

Warn < Error

« Après avoir regardé tout le projet, j'ai trouvé qu'il y a 8 extraits de code simi-

laires assignant le niveau ERROR, lorsqu'une InvalidStateTransitionException se produit dans doTransition(). Et il n'y a que 3 endroits choisissant le niveau WARN dans les mêmes situations. Par conséquent, je pense que ces 3 instructions de log devraient être assignées au niveau ERROR pour rester cohérent avec d'autres extraits de code ». (YARN-9349)

Plutôt que d'analyser le contexte du message et de l'instruction de log, le développeur préfère se fier à la fréquence d'un niveau de sévérité par rapport à un autre dans la même situation. Parfois, il y a des situations où le niveau attribué à un événement est clairement incorrect, comme un niveau *Error* donné à un événement qui n'affecte pas le fonctionnement normal du système. Dans ces cas, la décision sur comment gérer l'événement peut être basée davantage sur les pratiques existantes du système plutôt que sur un raisonnement logique :

Error > Info
« Je pense que les pratiques de journalisation devraient être cohérentes dans des contextes similaires … En examinant davantage de code, il y a effectivement des incohérences : certaines fonctions write() utilisent le niveau de log info, create() utilise error log, tandis que d'autres utilisent les niveaux warn et info ». (HDFS-14340)

4.3.4 AJUSTEMENTS GUIDÉS PAR L'EXPÉRIENCE

Cette catégorie inclut les ajustements motivés par l'observation des données de log générées par les systèmes logiciels. Parmi eux se trouvent certaines des raisons les plus évidentes d'ajuster le niveau de sévérité des logs, ainsi que des exceptions extraordinaires qui semblent nier les principes fondamentaux du log.

JOURNALISATION EXCESSIVE

La principale cause des ajustements du niveau de sévérité des logs est la production d'entrées de log excessives, que nous appelons *« journalisation verbeuse »*. Nous avons trouvé des incidents impliquant la verbosité pour les niveaux *Error*, *Warn*, *Info* et *Debug*. Une caractéristique spécifique à cette situation est que les ajustements sont toujours atténués vers le niveau *Debug*, sauf lorsque le niveau source est *Debug*, qui s'atténue vers le niveau *Trace*.

À l'exception de l'atténuation *Debug* > *Trace*, la motivation pour les ajustements de ce type vise à éviter de surcharger la production de données de log, ce qui impacte la performance du système et nécessite des stratégies pour persister de grandes quantités de données. La quantité excessive de logs produits est explicitement mentionnée dans les extraits des incidents ci-dessous :

Info > *Debug*

« . . . nous obtenons ce log imprimé environ 50 000 fois par seconde par thread ».
(KAFKA-13037)

Info > *Debug*

« Je constate des fichiers de sortie de resourcemanager de 8go et de gros fichiers log, voyant beaucoup de logs répétés (par exemple chaque appel rpc ou événement) on dirait que nous sommes trop verbeux à quelques endroits ».
(MAPREDUCE-3692)

Warn > *Debug*

« Chaque enregistrement d'entrée contribuera à une ligne de ce type de log,

conduisant à ce que la plupart des tâches aient un syslog > 1GB ». (HADOOP-11085)

Warn > Debug

« Nous avons vu un cas où ce message de log a été généré des millions de fois dans un court intervalle de temps, ralentissant le NameNode jusqu'à presque le paralyser. Il pourrait probablement être modifié pour être consigné au niveau DEBUG ». (HDFS-11054)

Error > Trace

« . . . à la fin du test, le nombre de lignes a bien atteint 1M dans le cluster cible. ». (HBASE-12419)

Les dernières citations présentent différentes métriques pour enregistrer une production excessive de logs en production : *« 50 000 fois par seconde »*, des fichiers de sortie entre 1 Go et 8 Go de données, *« générés des millions de fois »*. Cette génération excessive peut impacter les tâches qui consomment ces données en production.

Dans ce même contexte, nous avons trouvé des incidents qui visent à résoudre le problème de journalisation excessive en production grâce à un réglage *Info > Debug*. Cependant, il y a une résistance de la part des développeurs qui argumentent que si l'ajustement reste, des informations importantes pour le débogage seront perdues, suggérant que le débogage se fait dans un environnement de production (MAPREDUCE-3692), comme décrit précédemment.

Cependant, comme mentionné précédemment, nous rencontrons également des situations dans lesquelles l'instruction de log *Debug* est considérée verbeuse, rendant les données si « obscures » qu'elles rendent le débogage difficile :

Debug > Trace

« . . . journalise trop, tellement que cela obscurcit tout le reste de ce qui se passe ». (HBASE-7214)

AJUSTEMENTS EXCEPTIONNELS

Nous avons trouvé des preuves que l'ajustement des niveaux de sévérité des logs est une décision stratégique qui varie avec l'étape de développement du système. Nous avons observé des instances où des messages typiquement associés au niveau *Debug* étaient temporairement élevés à un niveau de sévérité plus élevé. Cette approche diffère de la méthode *« béquille »* dans le débogage, discutée précédemment sous *Ajustements des Principes Fondamentaux*.

Par exemple, lors des étapes initiales de développement du système, certaines informations peuvent être vraiment importantes pour le suivi et sont donc consignées à un niveau plus grave, comme *Info* ou *Error*, pour garantir leur visibilité. À mesure que le système se stabilise, ces logs peuvent générer des données excessives ou devenir moins critiques, justifiant un retour au niveau *Debug*.

Cette pratique est évidente dans MAPREDUCE-4614, où les développeurs ont décidé de garder une instruction *Debug* au niveau *Info* temporairement pour la stabilisation du système :

Debug < Info

« . . . peut-être que nous pouvons les avoir au niveau INFO pendant la stabilisation du système, et les changer en DEBUG plus tard. Sinon, le niveau DEBUG + un changement dans le niveau de log du client devrait fonctionner . . . Je suis ok avec INFO pendant la stabilisation ». (MAPREDUCE-4614)

Nous avons trouvé une situation similaire dans YARN-2704, où un log *Debug* a été initialement proposé pour être élevé à *Info* mais finalement ajusté à *Error*. La motivation derrière cette décision reflète une compréhension de la fréquence du log et de l'importance de sa visibilité pour une analyse efficace :

> *Debug < Error*
> *« Concernant les logs de niveau info/debug, ces logs sont tous des logs à faible fréquence, par défaut c'est seulement 1 jour à la fois (intervalle de renouvellement). Et il est tellement plus facile de déboguer au niveau info qu'au niveau debug. Peut-être au niveau info pendant la stabilisation de cette fonctionnalité ? »*
> (YARN-2704)

Dans ce cas, la décision d'aggraver la sévérité des instructions était motivée par la reconnaissance que des instructions rares mais importantes pourraient être négligées si elles restaient au niveau *Debug* au milieu d'un grand volume d'entrées de log.

COMPRÉHENSION DU FONCTIONNEMENT ET DE L'UTILISATION DU SYSTÈME

L'expérience de l'utilisateur lorsqu'il interagit avec les fonctionnalités d'un système contribue également au processus d'ajustement des logs, par exemple, comprendre quelles étapes doivent être suivies pour qu'un processus système soit réussi :

> *Info < Warn*
> *« Si les utilisateurs ne créent pas tous les sujets avant de démarrer une application de flux, ils pourraient obtenir des résultats inattendus . . . De plus, cette PR change*

le niveau de log de INFO à WARN lorsque les métadonnées n'ont pas de partitions pour un sujet donné ». (KAFKA-6802)

Dans ce cas, l'ajustement du niveau de sévérité des logs aidera à identifier la raison de comportements possiblement inattendus dans le flux de l'application en raison de ressources obligatoires non créées par l'utilisateur.

Un autre exemple de l'expérience avec le système est l'incident HDFS-14760 demandant initialement une atténuation de *Error* > *Info*. Cependant, un développeur clarifie la compréhension des conséquences de l'événement, et l'instruction de log passe au niveau *Warn* :

Error > *Warn*
« Je pense qu'au moins le niveau WARN est justifié, étant donné que c'est un problème qui pourrait potentiellement amener les répertoires à dépasser leur quota ou à être bloqués même si leur quota n'est pas encore atteint ». (HDFS-14760)

EXCEPTIONS DUES AU CODE HÉRITÉ DANS LES SYSTÈMES EN ÉVOLUTION

Au fur et à mesure que les systèmes évoluent, les nouveaux composants logiciels doivent souvent communiquer avec des composants hérités préexistants. Cette évolution, bien que nécessaire, peut parfois entraîner des échecs de communication entre les nouvelles et anciennes interfaces, pouvant provoquer des perturbations dans l'ensemble du processus.

Un exemple notable de ce défi est documenté dans l'incident KAFKA-5704. Kafka est une plateforme de streaming distribuée ; dans ce cas, les anciennes versions des courtiers (*brokers*) Kafka ne pouvaient pas gérer les nouvelles demandes de vérification et de création de

sujets manquants, résultant en une *UnsupportedVersionException* qui interrompait l'ensemble du processus. La solution proposée dans l'incident reflète une compréhension nuancée de l'évolution logicielle et des pratiques de journalisation efficaces :

> *Error > Debug*
> *« Nous devrions probablement juste l'attraper, consigner un message, et laisser les choses se poursuivre . . . Ce changement gère l'UnsupportedVersionException en consignant un message de debug et en ne faisant rien ».* (KAFKA-5704)

Cette approche implique d'attraper l'exception et de consigner un message au niveau *Debug*, une stratégie qui équilibre le besoin de suivi des erreurs avec la nécessité de maintenir la fonctionnalité du système. En optant pour un niveau de log moins verbeux, les développeurs montrent une décision éclairée, priorisant la continuité du système tout en conservant une trace d'incident discrète, accessible pour les diagnostics futurs. Cette approche, particulièrement pertinente pour les exceptions prévues et les erreurs non critiques, résonne avec les conclusions de Li *et al.* (2021a), qui soulignent que l'utilisation de niveaux de log moins verbaux, comme *Debug* ou *Info*, est préférable pour éviter la surcharge de logs en production et garantir une journalisation efficace sans compromissions de performance.

4.3.5 AJUSTEMENTS BASÉS SUR DES DIRECTIVES

Une directive peut être un document écrit, ou elle peut également être l'un des incidents initiaux du projet qui cherche à établir la norme pour la manière de consigner.

Une directive de journalisation a été proposée dans la version *0.3.0* de Hadoop via l'incident HADOOP-211, approuvant une approche globale et unifiée de la journalisation. Les aspects clés de cette directive se concentrent sur l'établissement d'un format de journalisation

cohérent dans tous les sous-systèmes de Hadoop. Cette proposition vise à améliorer l'analyse et le débogage des logs en s'assurant que les logs sont structurés et souligne l'importance d'une annotation claire dans les entrées de logs, y compris les horodatages, les niveaux de journalisation et la traçabilité vers le sous-système d'origine. Cette initiative représente un pas significatif dans la normalisation des pratiques de journalisation au sein de Hadoop.

Il existe également des références à des conventions [71] non respectées dans les incidents de HBase, comme dans cet ajustement pour réduire la quantité de logs générés par défaut :

> *Debug > Trace*
> *« Nous devrions amener le serveur REST à être au même niveau que les conventions de niveau de log du RS (RegionServer). Les requêtes individuelles doivent être consignées uniquement au niveau TRACE ».* (HBASE-15954)

4.3.6 AJUSTEMENTS LIÉS AU CHANGEMENT DE BIBLIOTHÈQUE DE JOURNALISATION

Les bibliothèques de logs ont différents niveaux de sévérité. Par conséquent, lorsqu'on passe d'une bibliothèque de journalisation à une autre, il n'y a aucune garantie que les niveaux de sévérité actuels seront maintenus. Cet événement s'est produit, par exemple, dans Hadoop, où au moins deux substitutions de bibliothèque de logs ont eu lieu : d'abord de Java Util Logging (sept niveaux de sévérité) à Apache Commons Logging (six niveaux de sévérité) puis d'Apache Commons Logging à SLF4J (cinq niveaux de sévérité). Pour cartographier les niveaux entre les différentes bibliothèques, nous avons utilisé les concepts d'équivalence entre les niveaux de sévérité présentés dans le chapitre 3. Ces circonstances ont entraîné les ajustements suivants :

71. Nous n'avons pas pu trouver d'enregistrement de ces conventions.

- Ajustements d'équivalence : *Fine* ≡ *Debug*, *Finer* ≡ *Debug*, *Finest* ≡ *Trace*, *Warning* ≡ *Warn*, *Severe* ≡ *Fatal*. Ces mises à jour se réfèrent au passage de Java Util Logging à Apache Commons Logging. Chaque mise à jour indique une équivalence des niveaux de sévérité entre les deux bibliothèques.
- Ajustement d'atténuation : *Fatal* > *Error*. La deuxième occurrence la plus élevée de mise à jour dans Hadoop, *Fatal* > *Error* (65), est causée par le remplacement du code par la bibliothèque SLF4J dans les instructions de logs, qui a seulement cinq niveaux de sévérité, ne comprenant pas *Fatal* — son niveau le plus grave est *Error* ; la bibliothèque précédemment utilisée était Apache Commons Logging.

4.3.7 OBSERVATIONS PRÉLIMINAIRES ET HEURISTIQUES POTENTIELLES

Tout au long de l'analyse des ajustements dans la phase 2, nous avons trouvé des indices qui suggèrent des modèles sur la manière dont les niveaux de sévérité sont attribués et ajustés pendant le développement et les opérations logicielles. Nous regroupons ces indices dans le tableau 4.9 et y référons comme des *« observations préliminaires »* (OP).

En complément des observations, nous avons également trouvé certains fragments textuels dans les incidents avec une nature plus impérative, contenant des recommandations directes concernant l'ajustement du niveau de sévérité des logs. Nous référons à ces textes, regroupés dans les tableaux 4.10 et 4.11, comme des *« heuristiques potentielles »* (HP) car ils suggèrent une applicabilité directe.

Chaque entrée de tableau (Tableaux 4.9, 4.10 et 4.11) représente le contenu filtré des discussions trouvées dans les incidents Jira sélectionnés.

---

71. Les citations de Jira dans cet article sont des reproductions fidèles des textes originaux.

TABLEAU 4.9 : Observations préliminaires.
© Eduardo Mendes de Oliveira, 2024

#	Texte	Incident Jira
[OP1]	Il existe des cas d'ajustements de *Debug* à *Info* exclusivement pour faciliter le processus de débogage.	MAPREDUCE-5766, HDFS-14759, YARN-1839
[OP2]	Les instructions de logs consistant uniquement en valeurs d'état sont fréquemment associées au débogage.	HDFS-14759
[OP3]	Les logs qui détaillent excessivement les événements et états étaient généralement classifiés comme des logs de débogage, indiquant une préférence possible pour une granularité plus fine à ce niveau de gravité.	MAPREDUCE-5766, YARN-1022
[OP4]	Attribuer des erreurs gérées par le client à des niveaux destinés à signaler des défaillances du système peut ne pas être approprié, car ces erreurs pourraient être plus convenablement consignées à un niveau moins sévère.	HADOOP-10274, HDFS-14760
[OP5]	Les informations vitales devraient éviter d'être enregistrées aux niveaux de gravité de finalité *Débogage* pour garantir que les événements essentiels ne soient pas obscurcis et restent facilement identifiables.	HADOOP-1034, HADOOP-12789, HADOOP-14987, KAFKA-6802
[OP6]	L'importance et la rareté des informations consignées devraient être directement proportionnelles au niveau de gravité choisi ; des événements plus critiques et moins communs justifient des niveaux de gravité plus élevés.	HADOOP-12789, YARN-2704
[OP7]	Les entrées de logs qui se produisent fréquemment sont mieux adaptées aux niveaux de gravité de finalité *Débogage*, facilitant une analyse ciblée sans submerger le log avec des informations moins critiques.	KAFKA-13037, MAPREDUCE-3692, HADOOP-11085, HDFS-11054, HBASE-12419, HBASE-7214, HBASE-15954
[OP8]	Le niveau de gravité choisi pour une déclaration de log peut changer à mesure que le système mûrit, reflétant le changement dans l'importance et la fréquence des événements consignés.	MAPREDUCE-4614, HDFS-14760, YARN-2704
[OP9]	Les comportements normaux et bénins ne sont pas compatibles avec les niveaux des finalités d'Avertissement et d'Échec.	HDFS-6998, MAPREDUCE-4570

TABLEAU 4.10 : Heuristiques potentielles.
© Eduardo Mendes de Oliveira, 2024

#	Ajustement	Heuristique potentielle	Incident Jira
[HP1]	*Warn<Fatal*	*« Les journaux de ReplicationPeer au niveau WARN interrompent le serveur au lieu de le faire au niveau FATAL. ».*	HBASE-7037
[HP2]	*Warn<Error*	*« Seule l'IOException est interceptée et enregistrée (au niveau warn). Chaque Throwable devrait être enregistré au niveau error. Par exemple : une RuntimeException se produit dans la méthode writeBlock(). L'exception ne sera pas enregistrée, mais simplement ignorée. Le socket est fermé silencieusement et rien ne se passe, une exception incompréhensible sera lancée dans le DFSClient envoyant le bloc [...] ».*	HADOOP-1034
[HP3]	*Info<Warn*	*« Je passerais l'info de la ligne 80 de JettyBugMonitor à un niveau warning, car elle est enregistrée une seule fois et pourrait désactiver une fonctionnalité que l'utilisateur pense avoir. [...] J'ai changé "info" en "war" comme vous l'avez suggéré dans le commit. ».*	MAPREDUCE-3184
[HP4]	*Debug<Info*	*« Ce message devrait être au niveau INFO. Cela ne se produit pas souvent et il est important de savoir quels plugins décident des suppressions de fichiers tout le temps. ».*	HBASE-23250
[HP5]	*Error>Warn*	*« Ainsi, l'erreur peut ne pas vraiment être une erreur si le code du client peut la gérer. ».*	HADOOP-10274
[HP6]	*Error>Info*	*« Il y a un log warn avant cette exception. Le code fonctionne toujours bien même avec l'exception lancée. [...] Oui, le code fonctionne toujours bien dans les trois situations. [...] Ainsi, il semble que la première déclaration de log devrait être assignée à un niveau inférieur, INFO. ».*	HDFS-14340
[HP7]	*Error>Warn, Error>Info*	*« ClassFinder enregistre des messages d'erreur qui ne sont pas exploitables, donc ils ne font que distraire [...] Un correctif simple qui change les lignes en warn et en information en fonction de la probabilité qu'il soit réellement utile de les examiner. ».*	HBASE-9120
[HP8]	*Warn>Debug*	*« Nous devrons changer le niveau de log en debug pour éviter une telle journalisation excessive. ».*	HADOOP-11085
[HP9]	*Warn>Debug*	*« Il existe de nombreuses exceptions que le client peut gérer automatiquement [...] En attachant un patch qui change le log warning en un log debug [...] »*	HADOOP-13552
[HP10]	*Warn>Debug*	*« Si cela est vraiment inoffensif, pourquoi le journalisons-nous ? [...] Il suffit de le changer en debug ? D'accord. »*	HBASE-25642
[HP11]	*Info>Debug*	*« Le niveau DEBUG est approprié pour les messages enregistrés fréquemment, mais le message d'arrêt est consigné une seule fois et devrait être au niveau INFO, tout comme le message de démarrage. »*	HDFS-10377

TABLEAU 4.11 : Heuristiques potentielles (Partie 2 suite).
© Eduardo Mendes de Oliveira, 2024

#	Ajustement	Heuristique potentielle	Incident Jira
[HP12]	*Info>Debug*	*« C'est pourquoi nous avons le niveau de journalisation debug, n'est-ce pas ? Les développeurs peuvent l'activer s'ils veulent voir toutes les transitions AM, mais cela n'inflige pas des journaux volumineux aux utilisateurs normaux. »*	MAPREDUCE-3692
[HP13]	*Info>Debug*	*« Nous obtenons ce log imprimé environ 50 000 fois par seconde par thread [...] Je suis complètement d'accord pour dire qu'enregistrer cela au niveau INFO à chaque itération est totalement inapproprié. Je ne l'ai simplement pas souligné à l'époque, car je pensais que quelqu'un déposerait un ticket si cela dérangeait vraiment les gens. Et nous y sommes. »*	KAFKA-13037
[HP14]	*Debug>Trace*	*« CleanerChore journalise trop, au point d'obscurcir tout le reste qui se passe. [...] Les journaux les plus proéminents ont été réduits au niveau trace, donc si vous avez vraiment besoin de niveau info, vous pouvez y accéder (utile pour le débogage), mais cela ne devrait pas apparaître dans la plupart des cas. ».*	HBASE-7214
[HP15]	*Debug>Trace*	*« Des messages comme celui-ci peuvent s'accumuler considérablement dans les journaux du regionserver lorsque un cluster gère des tables vides. [...] Étant donné que de nombreux déploiements fonctionnent avec DEBUG comme niveau de log par défaut, surtout lorsqu'il s'agit de résoudre d'autres problèmes de production, ce message est mieux enregistré au niveau TRACE. ».*	HBASE-16220
[HP16]	*Debug>Trace*	*« Nous devrions amener le serveur REST à être au même niveau que les conven- tions de niveau de log du RS (RegionServer). Les requêtes individuelles doivent être consignées uniquement au niveau TRACE[...] »*	HBASE-15954

4.4 DISCUSSION

Dans cette section, nous discutons des résultats et des principales conclusions de notre étude selon les deux phases de notre méthodologie.

4.4.1 PHASE DESCRIPTIVE (PHASE 1)

Lors de la première phase, nous avons présenté des données quantitatives fournissant une vue d'ensemble des ajustements des niveaux de sévérité des logs dans les systèmes sélectionnés. Voici les principales conclusions de cette phase.

LA DISTRIBUTION DES NIVEAUX DE SÉVÉRITÉ RENFORCE L'IDÉE DES FINALITÉS DE SÉVÉRITÉ

La répartition en pourcentage des niveaux de sévérité dans les systèmes sélectionnés au fil des versions renforce le concept des finalités de sévérité présenté dans le chapitre 3. Les quatre niveaux de sévérité les plus présents dans les systèmes étudiés, à savoir *Info*, *Debug*, *Warning*, et *Error* (dans cet ordre), vont dans le sens des finalités de log proposées.

Comme les multiples niveaux de sévérité trouvés dans l'ensemble des bibliothèques de journalisation étudiées convergent vers les quatre niveaux de sévérité les plus utilisés dans les systèmes étudiés, que ce soit *Info*, *Debug*, *Warning*, et *Error* (dans cet ordre), ceux-ci correspondent aux finalités proposées : *informationnelle*, *débogage*, *avertissement*, et *défaillance*, respectivement.

MAJORITÉ DES AJUSTEMENTS ENTRE NIVEAUX NON LIÉS À DES ERREURS

En utilisant les catégories de modifications de niveaux de sévérité proposées par Yuan *et al.* (2012b), nos résultats s'accordent davantage avec ceux de Chen & Jiang (2017b), où la majorité des ajustements de sévérité concernent des événements non liés à des erreurs (85,7% dans notre étude et 76% dans celle de Chen & Jiang (2017b)). En contraste avec les découvertes de Yuan, où les ajustements liés aux erreurs prédominent, cela peut indiquer que, dans les projets étudiés, les développeurs effectuent des ajustements pour améliorer le suivi et le contrôle d'informations, au-delà du simple rapport des erreurs critiques.

De plus, en tenant compte des autres catégories de journalisation suggérées par Fu *et al.* (2014) et He *et al.* (2018), qui se concentrent sur la description d'événements de défaillance, il devient opportun de développer des catégories additionnelles prenant en compte l'utilisation des niveaux de sévérité au-delà du contexte de pannes.

IL Y A UNE TENDANCE À AMENER LES LOGS EXCLUSIVEMENT DESTINÉS AU DÉBOGAGE VERS L'ENVIRONNEMENT DE PRODUCTION

Info et *Debug* sont les niveaux de sévérité prédominants dans les instructions de log des systèmes sélectionnés, ainsi que les ajustements impliquant ces deux niveaux : *Info* > *Debug* et *Debug* < *Info*, dans cet ordre. Dans certains cas, les développeurs peuvent choisir de classer les messages *Debug* comme *Info* afin de capturer des informations plus détaillées dans les environnements de production. Cependant, cette pratique peut poser des défis potentiels, tels que la génération excessive de logs qui obscurcissent les événements critiques ou ajoutent une charge inutile au système. Bien que cet ajustement puisse être utile à des fins de débo-

gage, il souligne la nécessité de directives plus structurées pour équilibrer la verbosité et les performances dans la journalisation en production.

IL EST DIFFICILE DE DISTINGUER LES INSTRUCTIONS DE LOGS ENTRE LES NIVEAUX DE SÉVÉRITÉ ADJACENTS

Les ajustements de sévérité d'un seul degré, qu'ils soient atténuants ou aggravants, constituent la majorité des ajustements observés. Cette constatation soutient l'idée que la distinction entre les niveaux de sévérité adjacents est subtile et peut même être biaisée, rendant le choix du niveau de sévérité complexe. Comme le notent Li *et al.* (2017a), les développeurs ajustent fréquemment les niveaux entre degrés adjacents dans un effort pour affiner la granularité des logs sans provoquer de surcharge.

Cependant, cette difficulté à faire des distinctions précises peut également entraîner des incohérences, en particulier en l'absence de directives claires pour l'usage de chaque niveau. Les ajustements fréquents entre *Info* et *Debug*, souvent sans raison technique bien définie, indiquent que des interprétations individuelles influencent ces choix. Cette tendance met en évidence le besoin de clarifier les rôles de chaque niveau de sévérité pour minimiser les ambiguïtés et renforcer la cohérence dans la gestion des logs.

IL Y A UNE TENDANCE À ATTRIBUER INITIALEMENT DES NIVEAUX DE SÉVÉRITÉ EXCESSIVEMENT ÉLEVÉS AUX INSTRUCTIONS DE LOGS

Il y a une prévalence des ajustements dans la catégorie d'atténuation. Cette prévalence suggère que les développeurs ont tendance à attribuer des niveaux de sévérité plus élevés que nécessaire, ce qui explique le besoin de réduire ces niveaux au fil du temps. Cependant,

cela pourrait aussi indiquer une tendance à percevoir certaines instructions de logs comme moins graves dans les versions ultérieures ou que, lors de l'écriture d'un nouveau code, il est nécessaire d'observer son exécution davantage que lorsqu'il devient mature, possiblement avec moins de bogues.

Comme l'observent Li *et al.* (2020a), les développeurs procèdent souvent à des ajustements réactifs des niveaux de journalisation, notamment lors de l'intégration finale, pour aligner la sévérité des logs avec les besoins en production. Cependant, lorsqu'ils ne sont pas liés à des pratiques de révision continue, ces ajustements peuvent introduire des débits techniques en accumulant des instructions de log dont la sévérité est inadéquate pour leur contexte réel d'utilisation.

4.4.2 PHASE EXPLICATIVE (PHASE 2)

Dans cette phase, nous nous sommes concentrés sur les données qualitatives, en analysant les textes des incidents liés aux ajustements des niveaux de sévérité des logs trouvés pendant la phase 1 afin de découvrir les raisons qui ont guidé ces ajustements. Nous présentons ci-dessous les principales conclusions.

PRÉVALENCE DES AJUSTEMENTS GUIDÉS PAR L'EXPÉRIENCE ET LES PRINCIPES FONDAMENTAUX DE LA JOURNALISATION

Nos résultats montrent que la prévalence des ajustements basés sur l'expérience et les principes fondamentaux de journalisation reflète une combinaison de stratégies adaptatives et de bonnes pratiques dans le choix des niveaux de sévérité.

La phase 2 révèle que les ajustements basés sur l'expérience constituent la majorité des ajustements dans les trois projets. Cette tendance suggère que l'expérience pratique dans le dépannage et l'exploitation des systèmes influence de manière significative les décisions concernant les niveaux de sévérité des logs. Comme l'observent Li *et al.* (2020a), les développeurs utilisent souvent des stratégies improvisées pour ajuster la journalisation, de manière proactive, en déterminant où et quand enregistrer des logs, ou de manière réactive, en ajustant les niveaux de sévérité lors de l'intégration finale.

Cette approche intuitive et flexible se reflète dans notre étude, où une grande partie des ajustements observés répond aux exigences des environnements de production et aux besoins en informations des développeurs. Dans les situations de dépannage, par exemple, les développeurs ajustent souvent les niveaux pour mettre en évidence des événements critiques ou atténuer des messages trop verbeux, facilitant ainsi une gestion plus efficace des journaux. Bien que ces ajustements soient souvent improvisés, ils illustrent une utilisation pragmatique de la journalisation pour s'adapter aux conditions spécifiques de chaque projet, tout en améliorant la qualité des entrées de logs.

Par ailleurs, bien que les ajustements guidés par l'expérience soient plus nombreux, les ajustements basés sur les principes fondamentaux de journalisation ont également une présence substantielle. Cela indique un nombre significatif d'ajustements orientés sur des pratiques techniques pour aligner les instructions de logs avec le niveau le plus approprié, visant à maintenir la cohérence et la clarté des instructions de logs.

FAIBLE INCIDENCE DES AJUSTEMENTS BASÉS SUR LES DIRECTIVES ET LES PRATIQUES HISTORIQUES

Les ajustements basés sur des directives et des pratiques historiques sont les moins fréquents dans notre étude, ce qui pourrait indiquer que les directives formelles et les précédents historiques jouent un rôle relativement mineur dans les processus décisionnels pour les ajustements des logs, par rapport à l'expérience pratique et aux principes fondamentaux de journalisation. Cependant, en l'absence de telles directives, les pratiques historiques de journalisation documentées dans les incidents Jira remplissent souvent le rôle de documentation implicite pour le projet.

Ces résultats s'alignent avec les observations de Li *et al.* (2021b), qui mettent en lumière les incohérences de niveaux de log causées par la duplication d'instructions et les clones de code. Dans notre étude, nous avons également observé que, face à des événements similaires, les développeurs préfèrent souvent se fier à la fréquence d'un niveau de sévérité utilisé dans des contextes antérieurs, ce qui favorise une uniformité apparente mais peut masquer des problèmes sous-jacents de classification des niveaux.

Ce comportement pragmatique, bien que visant la cohérence, présente le risque de créer des *code smells*, notamment lorsque des événements non critiques sont enregistrés avec des niveaux élevés, contribuant ainsi à une surcharge de journaux. En intégrant des pratiques historiques sans reconsidérer chaque cas spécifique, ces ajustements montrent qu'une documentation et des directives formelles sur les niveaux de log pourraient aider à atténuer les incohérences dans l'utilisation des niveaux de sévérité, assurant une journalisation plus efficace et adaptée aux besoins de chaque environnement.

PRÉVALENCE DES AJUSTEMENTS ENTRE LES NIVEAUX INFO ET DEBUG

La phase 2 a révélé que la raison des ajustements entre les niveaux *Debug* et *Info*, principalement des atténuations, n'est pas guidée par les fondements de la journalisation. En même temps, ils ne peuvent pas non plus être justifiés par l'expérience. Les instructions de logs au niveau *Debug* tendent à être aggravées à *Info* avec l'excuse qu'il serait plus facile de déboguer au niveau *Info*. Cela est tout à fait inapproprié, car cet ajustement peut causer une surcharge de données, démontrant une mauvaise compréhension des objectifs des niveaux de sévérité des logs.

AJUSTEMENTS STRATÉGIQUES PASSÉS

Certains logs ont été ajoutés dans les versions précédentes des logiciels pour relever des défis spécifiques à cette période. Cependant, à mesure que le logiciel évolue, la pertinence de ces enregistrements peut diminuer ou leur verbosité peut devenir excessive. Réviser et ajuster périodiquement ces logs peut aider à maintenir l'efficacité du système de journalisation.

PRODUCTION EXCESSIVE DE LOGS DUE À DES INSTRUCTIONS DE NIVEAU *DEBUG* MAL CLASSIFIÉES

Un grand nombre d'ajustements ont été effectués pour réduire la production excessive d'entrées de logs en environnement de production, que nous appelons *« journalisation verbeuse »*. Ce type de problème a déjà été signalé, comme dans l'étude de Li *et al.* (2017b), qui documente un cas d'atténuation du niveau *Info* vers *Debug* en raison de « bruit excessif ». Notre étude révèle des situations similaires impliquant les niveaux *Error* et *Warn* pour des raisons comparables. Dans chacun des cas identifiés, la solution a consisté à abaisser les niveaux de

sévérité à ceux destinés à la *finalité de débogage*, indiquant ainsi une classification initiale inadaptée des niveaux de sévérité. Bien que ces logs verbeux puissent être utiles pendant le développement ou le débogage, ils tendent à devenir une surcharge dans les environnements de production, ce qui suggère la nécessité de stratégies plus rigoureuses pour la gestion des niveaux de sévérité.

NIVEAUX PLUS SÉVÈRES POUR LES PROBLÈMES SILENCIEUX

Les problèmes silencieux peuvent être difficiles à diagnostiquer en raison de la rareté des logs associés à des niveaux de sévérité plus verbeux. Identifier et gérer correctement ces scénarios aide à améliorer les tâches de dépannage. Utiliser un niveau de sévérité plus élevé peut être crucial dans ces cas.

LE NIVEAU DE SÉVÉRITÉ DES LOGS EST MUTABLE

La phase 2 révèle également que dans le développement de logiciels, ajuster les niveaux de logs est une décision stratégique qui varie en fonction de l'étape de développement du système. Nous avons observé des cas où des instructions de logs portant des données de niveau *Debug*, satellites à des événements associés à des niveaux de sévérité plus élevés, ont été temporairement escaladées à un niveau de sévérité plus élevé. Cette approche diffère des ajustements « forcés » de *Debug* à *Info* lors du débogage, discutés précédemment dans la section précédente, en augmentant arbitrairement la sévérité d'une instruction de log, ce qui n'ajoute pas de valeur aux instructions de logs d'autres niveaux de sévérité.

Cette mutabilité nous a conduits à comprendre que dans une instruction de log, le niveau de sévérité n'est pas entièrement dicté par son message. Plutôt, le niveau de sévérité peut

varier à mesure que le système se développe et évolue, même si le message lui-même reste le même. À chaque version du logiciel, la sévérité peut s'atténuer ou s'aggraver en fonction des circonstances dans lesquelles l'instruction de log est appliquée.

Cette observation est en partie soutenue par l'approche proposée par Tang *et al.* (2022a), qui utilise l'historique Git et un modèle de degré d'intérêt (*Degree of Interest* - DOI) pour identifier dynamiquement les zones du code qui nécessitent plus d'attention. Le système ajuste alors automatiquement les niveaux de sévérité des logs associés, reflétant ainsi l'intérêt croissant des développeurs pour des portions spécifiques du code à mesure que l'importance des fonctionnalités évolue — même lorsque le contenu du message de log reste inchangé.

4.5 MENACES À LA VALIDITÉ

Nous décrivons ci-dessous les principales menaces à la validité de la recherche exposée dans ce chapitre, ainsi que les stratégies employées pour les minimiser.

4.5.1 VALIDITÉ INTERNE

Les menaces à la validité interne concernent la rigueur (et donc le degré de contrôle) de la conception de l'étude.

Méthode de sélection des instructions de log. La détection des instructions de log en utilisant SLogAnalyzer emploie des expressions régulières ; par conséquent, certaines d'entre elles peuvent être absentes de l'ensemble utilisé pour chaque système. Pour atténuer ce problème, SLogAnalyzer utilise un *pipeline* bien défini pour extraire les instructions, incluant le nettoyage des données non pertinentes, la capture de la structure de chaque fichier source, l'identification des blocs de code maximaux présents, puis l'application d'une expression

régulière. Cette méthode a permis d'extraire 54 881 instructions de log au total à travers les systèmes étudiés.

Suivi des ajustements. La comparaison des instructions de log entre les versions peut avoir incorrectement identifié des ajustements de sévérité de log en associant des extraits de code non correspondants. Cependant, le *pipeline* SLogAnalyzer identifie les blocs dans lesquels chaque instruction de log est située, comparant les instructions et les blocs qui les contiennent. De plus, nous avons validé manuellement les 2 228 ajustements associés aux incidents utilisés pour nos analyses afin d'augmenter la validité.

Méthode de sélection des incidents. Les incidents ont également été filtrés en utilisant une expression régulière, ce qui pourrait entraîner la perte de certains incidents pertinents concernant les ajustements de niveau de sévérité ou les pratiques de journalisation. Avant de recourir à cette approche, nous avons testé certaines techniques basées sur trois algorithmes d'apprentissage automatique sur un ensemble et analysé les résultats. Dans ce cas, l'expression régulière s'est avérée plus efficace pour sélectionner un groupe plus large d'incidents pour une analyse ultérieure, ce qui augmente la validité des résultats. Ce processus a abouti à une sélection finale de 248 incidents, répartis entre les projets analysés.

Ensemble de publications stables. Nous avons limité notre étude aux publications stables des systèmes sélectionnés, ce qui peut nous avoir fait manquer certaines tendances concernant les ajustements de sévérité de log dans les « ajustements intra-versions ». Pour atténuer ce problème, inspirés par la méthode de *backward* des revues littéraires systématiques (Keele, 2007), nous avons également analysé les incidents cités dans les descriptions et les commentaires des incidents sélectionnés.

Biais dans l'analyse des données. Le processus d'interprétation des textes des incidents Jira comporte une part inévitable de subjectivité, ce qui peut introduire des biais dans les résultats. Pour atténuer cela, nous avons adopté une approche systématique d'analyse des données. Les incidents ont été examinés indépendamment par le doctorant et un collaborateur, afin de limiter les biais individuels. Ensuite, des discussions ont permis de rapprocher les interprétations, et les résultats ont été revus pour garantir leur cohérence, augmentant ainsi la validité. De plus, tous les incidents ont été rigoureusement évalués par au moins deux personnes.

Fiabilité des sources de données. L'exactitude et l'exhaustivité des descriptions et des commentaires des incidents dans Jira peuvent influencer considérablement nos conclusions. Il y a un risque que des interprétations erronées surviennent en raison de descriptions incomplètes ou trompeuses des incidents. Pour atténuer ce problème, nous avons exclu de notre évaluation les incidents qui manquaient d'explications sur l'ajustement de la sévérité, malgré le fait que le titre était explicitement sur le sujet.

4.5.2 VALIDITÉ EXTERNE

Les menaces à la validité externe sont tous les facteurs au sein d'une étude qui réduisent la généralité des résultats.

Diversité des bibliothèques de journalisation. Nous nous concentrons sur un éventail limité de 40 bibliothèques de journalisation, ce qui peut ne pas représenter l'ensemble des pratiques de journalisation. Cette limitation pourrait affecter l'exactitude de notre catégorisation des niveaux de sévérité. Cependant, nous avons appliqué une méthodologie pour sélectionner un ensemble représentatif de bibliothèques et avons appliqué des critères validés d'inclusion, améliorant ainsi la validité de nos résultats.

Diversité des systèmes sélectionnés. Nos conclusions sont basées sur un ensemble spécifique de trois systèmes Java *open-source*, qui peuvent ne pas représenter d'autres systèmes logiciels ou langages de programmation. Notre objectif est de comprendre l'état de la pratique basé sur un ensemble significatif de versions de systèmes fermés et de générer des heuristiques qui peuvent contribuer en tant que guides dans le processus de journalisation. Notre objectif n'est pas de trouver des résultats généralisables à toutes les situations de choix de niveau de sévérité des logs. De plus, nous appliquons des critères d'inclusion et d'exclusion validés et bien définis, en sélectionnant uniquement des incidents résolus pour augmenter la validité de nos résultats. Dans les travaux futurs, nous tenterons de confirmer nos conclusions en recherchant un large éventail de systèmes et, par conséquent, de textes des incidents.

4.6 REMARQUES FINALES

Dans ce chapitre, nous avons analysé les ajustements des niveaux de sévérité des logs dans les systèmes logiciels *open-source*. Cette analyse a révélé plusieurs tendances importantes, telles que la prévalence des ajustements entre les niveaux *Debug* et *Info*, la mutabilité des niveaux de sévérité en fonction de la maturation des systèmes, et la forte influence de l'expérience des développeurs sur les décisions d'ajustement.

Nous avons également constaté que les ajustements sont souvent dictés par des facteurs contextuels, tels que les exigences de l'environnement de production, la verbosité excessive des logs ou les changements de bibliothèques de journalisation. En l'absence de directives formelles, les développeurs s'appuient principalement sur leur expérience et des pratiques historiques pour guider ces ajustements.

Ces résultats serviront de base à la formulation d'heuristiques dans le chapitre suivant, qui visera à fournir des recommandations pratiques pour améliorer la classification des niveaux

de sévérité des logs et soutenir les développeurs dans l'amélioration de la qualité des logs produits.

CHAPITRE V
HEURISTIQUES POUR CHOISIR LES NIVEAUX DE SÉVÉRITÉ DE LOGS

Ce chapitre représente la phase prescriptive de notre étude, visant à répondre à la question de recherche suivante : *Comment améliorer la classification des niveaux de sévérité des logs ?* En s'appuyant sur les résultats empiriques des chapitres 3 et 4, qui ont fourni respectivement une cartographie de l'état de l'art des niveaux de sévérité et une analyse des ajustements de sévérité dans des systèmes *open-source*, nous formulons un ensemble d'heuristiques pour guider les développeurs dans leurs décisions concernant la classification et l'ajustement des niveaux de sévérité des logs.

Les résultats des phases précédentes révèlent des tendances et des patrons dans les pratiques actuelles de journalisation. Notamment, nous avons observé que la classification des niveaux de sévérité est souvent influencée par des contextes opérationnels et des expériences subjectives, plutôt que par des lignes directrices formelles. De plus, les ajustements récurrents des niveaux de sévérité entre *Debug* et *Info*, la mutabilité des niveaux de sévérité en fonction de la maturité des systèmes et l'absence de directives explicites soulignent la nécessité d'une approche plus structurée pour aider les praticiens dans ce processus décisionnel.

Ainsi, ce chapitre présente un ensemble de 24 heuristiques pour guider la sélection, la révision et l'ajustement des niveaux de sévérité des logs. Ces heuristiques, basées sur une analyse approfondie, fournissent des recommandations pratiques et applicables pour les ingénieurs de développement et d'exploitation, les aidant à prendre des décisions éclairées sur la classification des niveaux de sévérité. L'objectif est d'améliorer la qualité des logs produits, en assurant qu'ils soient à la fois pertinents pour le dépannage et optimisés pour ne pas surcharger les systèmes de production.

Ce chapitre est organisé comme suit : la section 5.1 présente une synthèse des concepts clés liés à la classification des niveaux de sévérité des logs, ainsi qu'un aperçu des finalités et des environnements concernés. Ensuite, la méthodologie utilisée pour générer les heuristiques est présentée en détail dans la section 5.2. Les résultats des heuristiques dérivées sont discutés dans la section 5.3. La section 5.4 propose une discussion approfondie sur l'application et les implications de ces heuristiques. Nous abordons ensuite les menaces à la validité de l'étude dans la section 5.5. Enfin, la section 5.6 conclut le chapitre avec des remarques finales sur les contributions et les perspectives futures.

5.1 PRINCIPES FONDAMENTAUX DU NIVEAU DE SÉVÉRITÉ DES LOGS

Avant de présenter la méthodologie et les résultats spécifiques de ce chapitre, il est important de rappeler certains des principes essentiels concernant les niveaux de sévérité des logs, déjà abordés dans les chapitres précédentes. Ces principes servent de fondation théorique à la génération des heuristiques et facilitent une meilleure compréhension des concepts sous-jacents à la classification et aux ajustements des niveaux de sévérité. En particulier, nous résumons ici la convergence des multiples niveaux de sévérité vers une structure plus abstraite, qui sera utilisée dans l'élaboration des heuristiques.

5.1.1 CONVERGENCE DES NIVEAUX DE SÉVÉRITÉ DES LOGS

La variation de la nomenclature des niveaux de sévérité, la redondance des niveaux pour les objectifs de journalisation et des valeurs numériques nous a conduit à proposer que les 19 niveaux de sévérité peuvent converger vers des niveaux d'abstraction plus élevés. Nous avons identifié une tendance vers une convergence en six niveaux de sévérité, à savoir : *Trace, Debug, Info, Warn, Error* et *Fatal*, qui peuvent être catégorisés en quatre méta-niveaux, que

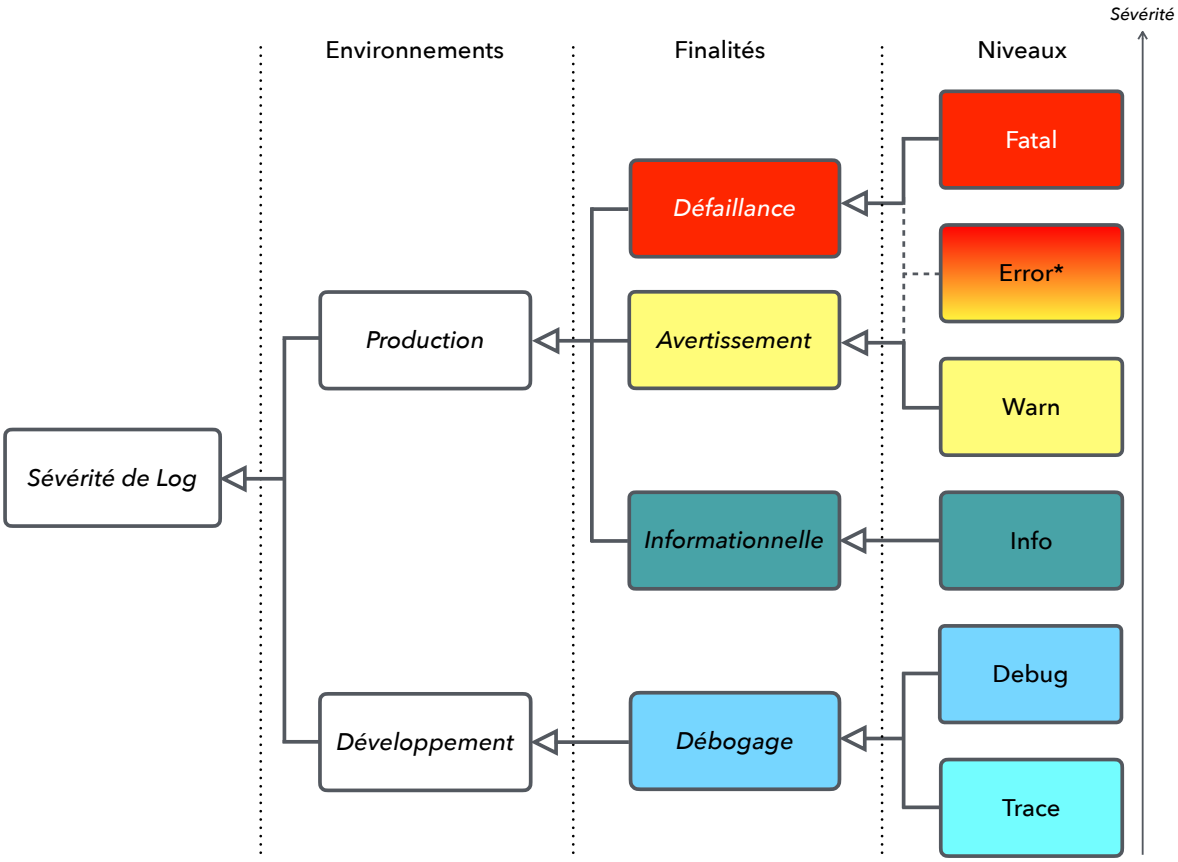

FIGURE 5.1 : Taxonomie des niveaux de sévérité des logs.
Les six niveaux de sévérité des logs sont divisés en deux catégories de gauche à droite, la première concernant l'environnement et la deuxième concernant la finalité.
***Selon la bibliothèque de journalisation utilisée,**
la classification de la finalité du niveau *Error* varie entre *avertissement* et *défaillance*.

nous appelons *finalités des niveaux de sévérité des logs*. La *finalité de débogage* englobe les niveaux axés sur les états des variables et les comportements internes du système. La *finalité informationnelle* enregistre le comportement attendu du logiciel. La *finalité d'avertissement* met en évidence des problèmes à examiner sans arrêter l'exécution du système. Enfin, la *finalité de défaillance* englobe les niveaux les plus sévères dédiés à la journalisation des défaillances du système.

De plus, les niveaux de sévérité des logs ont deux environnements cibles principaux : les environnements de développement et de production (Liu *et al.*, 2019). Alors que les tâches de développement peuvent bénéficier des entrées de log générées à partir de tous les niveaux, l'environnement de production ne nécessite qu'un sous-ensemble de ces entrées. Notamment, le niveau *Info* marque la limite des niveaux de sévérité supérieurs destinés à l'environnement de production. Nous présentons tous ces concepts concernant les niveaux de sévérité des logs à travers une taxonomie dans la figure 5.1.

Ces niveaux de sévérité des logs doivent former un ordre total au sens mathématique, chaque niveau étant considéré comme strictement plus grave que le précédent — en d'autres termes, il ne devrait pas y avoir deux niveaux ayant la même sévérité. Cependant, certains niveaux de sévérité provenant de différentes bibliothèques de journalisation peuvent correspondre à différents niveaux de finalité dans notre taxonomie.

5.2 MÉTHODOLOGIE

Après avoir décrit et exploré les ajustements des niveaux de sévérité des logs dans le chapitre 4, lors de la phase prescriptive, nous employons une approche inspirée de la Théorie Ancrée (Glaser & Strauss, 2017) pour extraire et formaliser les heuristiques des niveaux de sévérité des logs.

5.2.1 JUSTIFICATION DE L'APPROCHE BASÉE SUR LA THÉORIE ANCRÉE

La Théorie Ancrée (Glaser & Strauss, 2017) est une approche de recherche qualitative qui se concentre sur le développement de théories à partir des données collectées. Selon Goulding (2002), elle est particulièrement utile pour les recherches cherchant à prévoir et à expliquer les comportements. Elle se distingue par l'absence d'hypothèses de recherche préétablies, permettant aux concepts d'émerger directement du contenu analysé.

Dans le cadre de cette étude, cette approche a été choisie car il n'existait pas de cadre théorique prédéfini pour guider l'identification des heuristiques de niveaux de sévérité. L'objectif était de faire émerger, de manière inductive, des logiques et motifs d'ajustements observés dans les rapports Jira, en analysant les justifications apportées par les développeurs. Cela nous a permis de formaliser des heuristiques ancrées dans les pratiques réelles, reflétant les situations concrètes rencontrées dans les projets logiciels. Ce processus comprend plusieurs étapes clés :

1. Codage ouvert : les données sont désagrégées en unités significatives pour identifier des thèmes et des concepts.
2. Codage axial : les relations entre les catégories émergentes sont reconnues et examinées pour mieux comprendre les interactions.
3. Codage sélectif : les catégories sont intégrées pour produire une théorie cohérente, en se concentrant sur les éléments centraux qui relient les différents concepts.

Le processus de comparaison constante aide à ancrer la théorie dans les données, garantissant ainsi que les résultats soient étroitement liés aux informations fournies dans les sources analysées. Cela permet de s'assurer que la théorie développée est non seulement

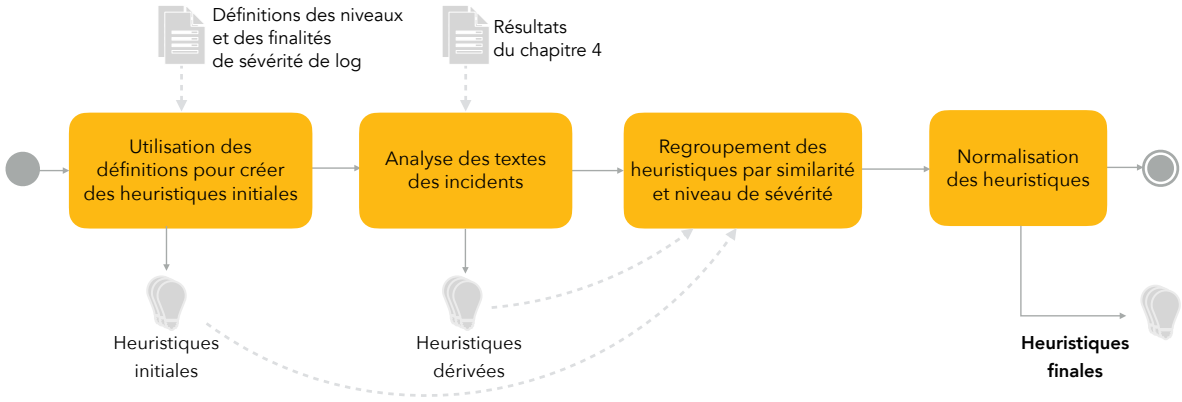

FIGURE 5.2 : Flux de processus pour dériver et normaliser les heuristiques du niveau de sévérité des logs.
© Eduardo Mendes de Oliveira, 2024

pertinente, mais également fondée sur des éléments empiriques concrètes, facilitant une compréhension plus approfondie des choix et des stratégies des développeurs.

5.2.2 GÉNÉRATION D'HEURISTIQUES

Nous utilisons des données de deux sources principales :

- la synthèse des niveaux de sévérité des logs (Section 3.4), présentée dans le chapitre 3 ;
- les résultats obtenus dans le chapitre 4, qui visait à comprendre les raisons des ajustements de sévérité au niveau des logs.

Les étapes de la géneration d'heuristiques (Figure 5.2) sont détaillées ci-dessous :

1. *Étape 1 : Génération d'heuristiques initiales* (codage ouvert). En se basant sur des définitions des niveaux et finalités de sévérité préalablement établies (Section 3.4.2), résumées dans la figure 5.1, le processus de génération des heuristiques est amorcé.
2. *Étape 2 : Analyse des textes des incidents* (codage ouvert).

(a) *Identification des heuristiques potentielles.* Cette étape, nous avons extrait les heuristiques potentielles (HP) du chapitre 4 (Section 4.3) à partir des problèmes analysés, lorsque disponibles. En général, nous recherchions des phrases impératives qui transmettent de bonnes pratiques et qui pourraient être associées à des motifs présents dans d'autres problèmes.

(b) *Enregistrement des observations préliminaires.* À partir des résultats de la catégorisation des ajustements du chapitre 4 (Section 4.3), nous avons enregistré des observations préliminaires générales sur les ajustements des niveaux de sévérité des logs comme apprentissages de cette phase. Chaque observation préliminaire (OP) est associée aux problèmes qui y sont liés. Par exemple :

- [OP1] : *« Il y a des instances d'ajustements de Debug à Info exclusivement pour faciliter le processus de débogage. »* (MAPREDUCE-5766, HDFS-14759, FIO-1839, MAPREDUCE-3692)

3. *Étape 3 : Regroupement des heuristiques par similarité et niveau de sévérite.*

(a) *Analyse collaborative et élaboration initiale d'heuristique* (codage axial) : Nous avons examiné chaque HP et OP, et nous avons conçu des heuristiques dérivées sur la base de leur combinaison. Nous avons analysé les différences dans les heuristiques identifiées pour réduire la liste aux heuristiques initiales.

(b) *Affinement et validation* (codage sélectif) : Nous avons également affiné les heuristiques initiales et les heuristiques dérivées en les faisant correspondre aux incidents associés et aux observations préliminaires pour nous assurer que plusieurs instances soutiennent chaque heuristique. Nous avons utilisé des descriptions statistiques des HP, OP et incidents comme indicateur de la validité de chaque heuristique. Par exemple, H9, qui stipule que *« les informations importantes ne conviennent pas pour les niveaux de sévérité à des fins de débogage »*, est soutenue par HP10 et

OP5 et est liée à plusieurs problèmes tels que HADOOP-12789 et HADOOP-1034. À cette étape, nous avons également éliminé les heuristiques redondantes.

4. *Étape 4 : Normalisation des heuristiques.*

   (a) *Transition vers la forme impersonnelle.* Dans un premier temps, les heuristiques ont été formulées sur la base des observations et des motifs identifiés. Ces formulations initiales n'étaient pas standardisées et étaient parfois exprimées sous forme impérative. Par la suite, chaque heuristique a été révisée pour être présentée de manière impersonnelle, afin de renforcer l'objectivité et la neutralité des recommandations.

   (b) *Conformité à la RFC 2119.* Afin de renforcer la clarté, la cohérence et l'applicabilité des heuristiques proposées, nous avons formulé chaque recommandation en suivant les termes de la RFC 2119 (Bradner, 1997). Cette spécification fournit une terminologie précise — notamment les expressions DOIT, NE DOIT PAS, DEVRAIT, NE DEVRAIT PAS et PEUT — qui permet de qualifier le niveau d'obligation ou de recommandation associé à une directive. L'adoption de cette terminologie assure que les exigences exprimées dans nos heuristiques puissent être interprétées sans ambiguïté et appliquées de manière cohérente dans des contextes logiciels variés.

   (c) *Révision finale et documentation.* L'ensemble final d'heuristiques a été révisé et documenté. La fréquence des questions liées à chaque heuristique a également été observée, fournissant un aperçu de sa base empirique.

Cette approche itérative a contribué à ce que les heuristiques soient basées sur les données quantitatives et qualitatives de notre étude.

5.3 RÉSULTATS

Dans cette section, nous présentons les résultats prescriptifs de notre étude, où nous transformons les observations et motifs identifiés dans les principes fondamentaux du niveau de sévérité des logs (Section 3.4), résumés dans la figure 5.1, et les résultats du chapitre 4, en heuristiques pratiques pour les décisions de niveau de sévérité des logs. Nous synthétisons un ensemble d'heuristiques qui reflètent les complexités et les nuances des ajustements de la sévérité des logs, fournissant une procédure structurée permettant de prendre des décisions éclairées dans la classification de la sévérité des logs.

La relation entre les heuristiques finales, les heuristiques potentielles, les observations préliminaires et les incidents étudiés est détaillée dans le tableau 5.1. Ce tableau présente une vue d'ensemble des heuristiques, en incluant pour chaque heuristique les heuristiques potentielles (HP), les observations préliminaires (OP) et les incidents qui les soutiennent. De plus, la colonne « Fréq(HD/HB/K) » indique les fréquences des incidents relatifs aux projets Hadoop (« HD »), HBase (« HB ») et Kafka (« K »). Notamment, les quatre premières heuristiques, qui concernent l'environnement cible des niveaux de sévérité, ont été soutenues exclusivement par les principes fondamentaux.

5.3.1 HEURISTIQUES BASÉES SUR LES PRINCIPES FONDAMENTAUX

À partir des principes fondamentaux, nous classons les niveaux de sévérité en deux catégories, d'abord concernant l'environnement puis concernant leur objectif selon la figure 5.1. Nous générons des heuristiques spécifiques pour chaque catégorie qui aident à classer les instructions de logs dans cette catégorie. Cependant, considérant que l'ensemble des niveaux de sévérité des logs dépend de la bibliothèque utilisée [72], nous présentons des heuristiques

72. Par exemple, alors que le niveau le plus grave dans une bibliothèque de journalisation est le niveau *Fatal*, il peut être le niveau *Error* dans une autre.

TABLEAU 5.1 : La relation entre les heuristiques, les heuristiques potentielles (HP), les observations préliminaires (OP) et les incidents Jira.
© Eduardo Mendes de Oliveira, 2024

Heuristique	HPs associées	OPs associées	Incidents associés	Fréq HD/HB/K
[H1]	-	-	-	-
[H2]	-	-	-	-
[H3]	-		HDFS-14759, MAPREDUCE-5766, YARN-1022	3/0/0
[H4]	-	-	-	-
[H5]	-	-	HADOOP-10274, HADOOP-1034, HADOOP-10657, HADOOP-96, HDFS-14238, HDFS-14395, HDFS-14521, HDFS-14759, HDFS-14760, HDFS-6998, MAPREDUCE-3348, MAPREDUCE-4570, MAPREDUCE-5766, MAPREDUCE-7063, YARN-1022, YARN-10369, YARN-1608, YARN-1839, HBASE-20665, HBASE-7037, HBASE-9120, HBASE-23047, HBASE-14042, HBASE-8940, HBASE-10906, HBASE-20447, KAFKA-9540	18/8/1
[H6]	-	[OP1]	HDFS-11593, HDFS-14759, MAPREDUCE-3692, MAPREDUCE-5766, MAPREDUCE-7063, YARN-1608, YARN-1839, HBASE-20554, HBASE-20665	7/2/0
[H7]	-	[OP2]	HDFS-14759, MAPREDUCE-5766, YARN-1022	3/0/0
[H8]	-	[OP4]	HADOOP-10274, HADOOP-13552, HADOOP-17597, HDFS-14760	4/0/0
[H9]	-	[OP5]	HADOOP-1034, HDFS-11593, HADOOP-12789, HBASE-13675, HADOOP-14987	5/0/0
[H10]	-	[OP8]	HDFS-14760, MAPREDUCE-4614, YARN-2704, KAFKA-5704	3/0/1
[H11]	[HP1, HP2]	-	HADOOP-1034, HBASE-7037	1/1/0
[H12]	[HP5]	[OP4]	HADOOP-10274, HDFS-14760, HBASE-10906, HBASE-14042, HBASE-16589, HBASE-27391, KAFKA-9540	2/4/1
[H13]	[HP6]	-	HADOOP-17597, HDFS-6998, HBASE-27391	2/1/0
[H14]	[HP7]		HADOOP-17597, HDFS-6998	2/0/0
[H15]	[HP7]	[OP4]	HADOOP-10274, HADOOP-17597, HDFS-14760, HBASE-9120, HBASE-26189, HBASE-27391	3/3/0
[H16]	[HP3]	-	HDFS-14760, KAFKA-5704, KAFKA-6802	1/0/2
[H17]	[HP4, HP11]	[OP6]	HADOOP-10657, HADOOP-12789, YARN-2704, HBASE-23250	3/1/0
[H18]	[HP4, HP11]	[OP6]	HADOOP-10657, HADOOP-12789, MAPREDUCE-3184, YARN-2704, HBASE-5582, HBASE-7037, HBASE-8940, HBASE-23250, KAFKA-6802	4/4/1
[H19]	[HP11]	-	HDFS-10377	1/0/0
[H20]	[HP8, HP11]	[OP7]	HADOOP-7858, HADOOP-8075, HADOOP-8932, HADOOP-9135, HADOOP-9582, HADOOP-10015, HADOOP-11085, HADOOP-15441, HADOOP-17836, HADOOP-18065, HADOOP-18574, HDFS-8659, HDFS-9906, HDFS-10377, HDFS-13692, HDFS-15197, MAPREDUCE-3265, MAPREDUCE-3692, MAPREDUCE-3748, YARN-1892, YARN-2213, YARN-3350, YARN-4115, YARN-5693, YARN-7727, YARN-8459, YARN-10997, HBASE-12419, HBASE-12539, HBASE-12461, HBASE-15582, HBASE-15954, HBASE-24524, HBASE-25483, HBASE-25556, HBASE-25642, HBASE-26443, HBASE-27588, KAFKA-4829, KAFKA-13037, KAFKA-13669	27/11/3
[H21]	[HP8, HP13]	[OP2, OP3]	HDFS-11593, HDFS-14759, MAPREDUCE-5766, YARN-1022	4/0/0
[H22]	[HP8, HP12]	[OP7]	HADOOP-7858, HADOOP-10015, HADOOP-10343, HADOOP-11085, HADOOP-16708, HDFS-11054, MAPREDUCE-3692, HBASE-12419, HBASE-15954, HBASE-20554, KAFKA-13037	7/3/1
[H23]	[HP12, HP15]	[OP2, OP3, OP7]	HADOOP-11085, HADOOP-18574, HDFS-11054, MAPREDUCE-3692, HDFS-14759, MAPREDUCE-5766, YARN-1022, HBASE-7214, HBASE-9371, HBASE-12419, HBASE-12461, HBASE-15954, HBASE-16220, HBASE-20701, HBASE-20770, HBASE-23687, HBASE-27079, KAFKA-13037	7/9/1
[H24]	[HP15, HP16]	[OP3, OP7]	HADOOP-10015, HADOOP-11085, HADOOP-18574, HDFS-11054, MAPREDUCE-3692, MAPREDUCE-5766, YARN-1022, HBASE-7214, HBASE-9371, HBASE-12419, HBASE-12461, HBASE-15954, HBASE-16220, HBASE-20701, HBASE-20770, HBASE-23687, HBASE-27079, KAFKA-13037	7/10/1

visant la finalité du niveau (*débogage*, *informationnelle*, *avertissement*, et *défaillance*) plutôt que le niveau lui-même.

HEURISTIQUES D'ENVIRONNEMENT

L'heuristique initiale sépare les instructions de logs en deux groupes : les phrases ciblant l'*environnement de développement* du système (*finalité de débogage*) et les phrases ciblant l'*environnement de production* (*finalités informationnelle, avertissement et de défaillance*). Les instructions de logs de chaque groupe ont des publics cibles différents. Ceci, en plus d'influencer le choix du niveau de sévérité des logs, impacte également la construction de leurs messages. Par conséquent, la construction des messages de logs est un facteur clé pour atteindre une classification de sévérité précise.

Par exemple, un message composé uniquement de valeurs d'état ou de variables peut être efficace pour aider à résoudre des problèmes spécifiques de l'environnement de développement ; ces détails ciblés soutiennent les développeurs dans l'identification et la résolution des problèmes durant le processus de développement. Cependant, pour les gestionnaires de systèmes dans un environnement de production et qui doivent analyser le comportement des systèmes sans les avoir développés, les messages doivent être écrits clairement afin de les aider à resoudre les problèmes. Ces messages doivent être rédigés dans le but de fournir une interprétation rapide et précise des événements enregistrés, aidant à identifier et à résoudre efficacement les problèmes.

Heuristique #1 : *Les niveaux Trace et Debug DOIVENT classifier les instructions de logs ciblant exclusivement le processus de développement.*

Heuristique #2 : *Les niveaux Info, Warn, Error et Fatal DOIVENT classifier les instructions de logs ciblant l'environnement de production.*

Heuristique #3 : *Les instructions de logs destinées à l'environnement de développement PEUVENT se concentrer sur des aspects spécifiques des états du système, tels que les valeurs des variables, sans nécessiter le contexte généralement nécessaire pour une compréhension plus large.*

Heuristique #4 : *Les instructions de logs destinées à l'environnement de production DOIVENT avoir des messages rédigés en phrases bien structurées et claires, visant à faciliter la compréhension humaine.*

Ensuite, nous passons à la deuxième catégorie d'heuristiques, relative à la finalité des niveaux de sévérité des logs.

HEURISTIQUE DES FINALITÉS

Le deuxième groupe d'heuristiques est généré en fonction des finalités des niveaux de sévérité des logs : *débogage*, *informationnelle*, *avertissement* et *défaillance*. Ces quatre finalités englobent les intentions pour tous les niveaux de sévérité dans les bibliothèques de logs, et suggèrent les principales raisons pour lesquelles nous enregistrons, en encadrant les instructions de logs de manière objective. Nous les présentons de la finalité la plus grave à la moins grave, car une sévérité plus élevée se traduit par un but plus objectif.

Les quatre finalités de journalisation (*débogage*, *informationnelle*, *avertissement* et *défaillance*) englobent l'intention pour tous les niveaux de sévérité dans les bibliothèques de logs et suggèrent la raison principale pour laquelle nous enregistrons en encadrant les instructions de logs de manière objective. Ces finalités ont généré l'heuristique suivante.

Heuristique #5 : *L'intention derrière la journalisation DOIT être analysée lors de la sélection des niveaux de sévérité, en s'assurant qu'ils sont choisis selon les finalités prévues (informationnelle, débogage, avertissement et défaillance). Les finalités DEVRAIENT guider les choix des niveaux de sévérité pour maintenir la clarté et l'utilité des données de log.*

5.3.2 HEURISTIQUES DE LA PHASE 2 | HEURISTIQUES ISSUES DES APERÇUS

L'investigation sur les causes de l'ajustement du niveau de sévérité des logs a également contribué aux heuristiques, à savoir :

Heuristique #6 : *Les niveaux Debug NE DOIVENT PAS être masqués en Info, car cela peut générer des logs verbeux.*

Heuristique #7 : *Les instructions de logs composées uniquement de valeurs d'état DEVRAIENT être classifiées selon les niveaux de la finalité débogage.*

Heuristique #8 : *Les erreurs que le client peut gérer NE DOIVENT PAS être classifiées avec des niveaux de sévérité de finalité de défaillance.*

Heuristique #9 : *Les informations importantes NE DEVRAIENT PAS être classifiées avec les niveaux de sévérité de finalité de débogage, car elles peuvent rendre difficile la constatation de l'existence de l'événement.*

Heuristique #10 : *Les instructions de logs ajustées stratégiquement dans les versions précédentes DEVRAIENT être révisées périodiquement pour s'assurer qu'elles restent pertinentes et ne contribuent pas à un débordement de logs. Le niveau de sévérité approprié pour une instruction de log PEUT changer à mesure que le système évolue.*

5.3.3 HEURISTIQUES DE LA PHASE EXPLICATIVE | HEURISTIQUES POTENTIELLES

Dans cette section, nous proposons des heuristiques basées sur les extraits de problèmes présentés dans le tableau 4.10.

HEURISTIQUES LIÉES AUX EXCEPTIONS

Par nature, il y a une tendance à lier les exceptions aux niveaux de sévérité de la finalité de *défaillance*, comme *Fatal* et *Error*. Cependant, il est important de considérer le contexte et les conséquences des exceptions, car de nombreux cas ne résultent pas en des situations critiques pour le système. C'est cependant le cas dans d'autres situations.

HP1 et HP2 sont des exemples d'exceptions causant des interruptions logicielles, telles que des arrêts de serveur (décrits dans HP1) ou des fermetures de sockets (décrites dans HP2), qui devraient interrompre d'importantes opérations logicielles. Malgré cela, ces instructions de logs avaient un niveau de sévérité moins critique et, par conséquent, les deux ont subi une aggravation de sévérité, passant d'un niveau d'*avertissement* à un niveau de *défaillance*. De plus, les deux sont impliqués dans le lancement d'exceptions de la classe Throwable [73], décrites dans d'autres problèmes Jira comme des événements alignés avec les niveaux de la finalité de *défaillance*, ajoutant des exemples concrets d'application.

Heuristique #11 : *Les instructions de logs indiquant un arrêt ou un échec du serveur ou des fermetures de socket DOIVENT être alignées sur les niveaux de la finalité de défaillance.*

73. « La classe Throwable est la superclasse de toutes les erreurs et exceptions dans le langage Java » Oracle (2018).

Nous allons énumérer certains cas dans lesquels les exceptions ne sont pas si critiques du point de vue du niveau de sévérité des logs :

- une exception a été lancée, mais l'utilisateur peut gérer la situation générée lui-même [HP5] ;
- une exception a été lancée, mais le « code » continue de bien fonctionner [HP6] ;
- une exception a été lancée, mais la situation causée par celle-ci ne nécessite pas d'action immédiate pour que le système reste fonctionnel [HP7].

À partir des situations énumérées ci-dessus, nous pouvons inférer de nouvelles heuristiques :

Heuristique #12 : *La sévérité DOIT être classifiée avec des niveaux de finalité de défaillance seulement si c'est un véritable problème qui impacte le fonctionnement du code et ne peut pas être traité par l'utilisateur ou le client.*

Heuristique #13 : *Si une exception est levée, mais que le système continue de fonctionner normalement sans impact sur ses fonctionnalités, alors l'instruction de log NE DEVRAIT PAS être classifiée avec un niveau de finalité de défaillance.*

Heuristique #14 : *Si une instruction de log rapporte un problème mais que le système ne s'arrête pas, elle DEVRAIT être classifiée comme un niveau de finalité d'avertissement plutôt qu'un niveau de finalité de défaillance.*

Heuristique #15 : *Dans les événements d'exception qui ne nécessitent pas d'action immédiate, cet événement NE DEVRAIT PAS être classifié pour le niveau de finalité de défaillance. Des niveaux de sévérité des logs moins sévères DOIVENT être envisagés.*

DÉSACTIVATION DES FONCTIONNALITÉS

HP3 rapporte un cas sur un événement qui désactive une fonctionnalité que les utilisateurs peuvent penser avoir accès. Plus précisément, il s'agit de la routine de détection de threads bloqués dans une classe. Cette fonctionnalité a été désactivée par défaut en raison de son comportement incertain — les développeurs soulignent que la méthode utilisée peut produire des résultats peu fiables ou coûteux en fonction de la machine virtuelle Java ou du système d'exploitation (MAPREDUCE-3184). Nous pouvons considérer cette situation comme un événement inattendu, comme décrit dans la finalité *d'avertissement*. Cependant, l'événement a été initialement consigné avec le niveau de sévérité *Info* malgré le fait qu'il soit décrit comme de faible fréquence et puisse passer inaperçu. Par conséquent, un ajustement de *Info* à *Warn* a été proposé pour alerter les utilisateurs de la désactivation de cette fonctionnalité potentiellement importante.

L'ajustement dans ce cas était une aggravation pour *avertissement*, et à partir de là nous pouvons suggérer les heuristiques suivantes :

> ***Heuristique #16 :*** *Si un événement peut désactiver une fonctionnalité attendue par les utilisateurs, cet événement DOIT être classifié avec un niveau de sévérité de finalité d'avertissement.*

ÉVÉNEMENTS INHABITUELS

Poursuivant l'idée que des informations importantes peuvent passer inaperçues en fonction du niveau de sévérité utilisé, HP4 décrit une situation concernant la suppression de fichiers, les classant comme «toujours importants», mais qui est au niveau Debug.

HP11 décrit une situation similaire concernant les instructions de log d'arrêt, qui sont enregistrées une seule fois et auraient une sévérité correspondant à un message de démarrage. Comme décrit avant, les instructions qui décrivent les débuts et fins d'événements attendus par le système conviennent pour le niveau de sévérité de finalité *informationnelle*.

Heuristique #17 : *Les événements inhabituels NE DEVRAIENT PAS être classifiés avec des niveaux de finalité de débogage. Des niveaux de sévérité des logs plus sévères DEVRAIENT être considérés.*

Heuristique #18 : *Les instructions de logs rapportant des événements rares mais significatifs DEVRAIENT être classifiées avec les niveaux de finalité informationnelle et d'avertissement.*

Heuristique #19 : *Les messages de démarrage et d'arrêt attendus DEVRAIENT être classifiés avec le niveau de sévérité de finalité informationnelle.*

LOGS FRÉQUENTS ET DÉTAILLÉS

Parmi les heuristiques potentielles dans le tableau 4.10, nous avons trouvé des indications répétées que les messages fréquents – c'est-à-dire celles apparaissant plusieurs fois par minute ou à chaque boucle d'exécution – conviennent pour les niveaux de sévérité de finalité de *débogage* [HP8, HP9, HP10, HP11, HP12, HP13]. En même temps, ils sont totalement inappropriés pour le niveau de finalité *informationnelle* [HP13], car ils peuvent surcharger les utilisateurs avec des messages qui ne sont pas critiques [HP12], en plus de rendre difficile la compréhension des logs destinés à l'environnement de production [HP14]. Comme décrit dans HP13, les messages fréquents peuvent être associés à des instructions de logs dans des blocs d'itération pour détailler les événements et états survenus, ce qui explique leur haute incidence.

Heuristique #20 : *Les niveaux de finalité de débogage DOIVENT classifier les messages qui sont fréquemment enregistrés.*

Heuristique #21 : *Les logs de résumé et les logs détaillés DEVRAIENT correspondre aux niveaux de finalité de débogage.*

Heuristique #22 : *Les utilisateurs NE DOIVENT PAS être submergés par des logs excessifs dans des circonstances normales, à moins qu'il ne soit critique pour l'utilisateur d'être conscient de l'événement.*

Dans ce contexte, nous devons également considérer le niveau de sévérité choisi pour la production de messages dans un environnement de production ; la norme est le niveau de sévérité *Info*. Cependant, dans HP15, nous voyons des déploiements avec le niveau *Debug* par défaut. Ce choix nous amène à considérer la sévérité des événements et états plus détaillés qui devraient être enregistrés, que ce soit au niveau *Debug* ou *Trace*.

Heuristique #23 : *Lorsque la norme pour la génération de logs dans un environnement de production est le niveau Info, le niveau Debug DEVRAIT être préféré pour des informations détaillées. Si la norme est le niveau Debug, le niveau Trace DEVRAIT être envisagé afin de ne pas générer des logs Debug verbeux.*

Nous devrions également considérer comment choisir entre les niveaux *Debug* et *Trace* lors d'occasions de logs fréquents, car même le même processus de débogage peut devenir obscur face à des logs verbeux [HP15]. Dans ces situations, nous pouvons recourir à l'heuristique suivante :

Heuristique #24 : *Dans les cas de logs verbeux qui rendent le processus de débogage peu clair, le niveau de sévérité des instructions les plus proéminentes DEVRAIT être atténué au niveau de sévérité Trace.*

5.4 DISCUSSION

La phase prescriptive de notre recherche met en lumière les défis liés au choix des niveaux de sévérité des logs. Au cours de notre processus de dérivation des heuristiques à partir des observations des phases précédentes, nous avons rencontré des problèmes reflétant la dynamique des développeurs lors de la création et de l'ajustement des niveaux de sévérité des logs.

ADAPTER LES MESSAGES DE LOG AUX ENVIRONNEMENTS CIBLES

Notre ensemble d'heuristiques considére l'importance d'adapter les instructions de log à l'environnement cible, qu'il s'agisse de l'environnement de développement ou de production. Au cours de notre étude, il est apparu clairement que la pertinence d'une entrée de log est liée au public visé. En écho aux observations de Shang *et al.* (2015), où « les logs de haut niveau sont destinés aux opérateurs de systèmes et les logs de bas niveau aux développeurs et testeurs », nous notons que les développeurs ont souvent besoin de détails granulaires pour le débogage, tandis que les administrateurs système nécessitent des logs clairs et concis pour résoudre efficacement les problèmes. Ces deux aspects des logs sont traduits par les heuristiques environnementales, qui soulignent la nécessité de construire des messages compatibles avec le niveau de sévérité choisi.

ADAPTATION À LA FINALITÉ DES NIVEAUX DE SÉVÉRITÉ

Notre proposition de regrouper les niveaux de sévérité des logs par finalité - *débogage, informationnelle, avertissement,* et *défaillance* - fournit un guide pour classifier les niveaux de sévérité des logs. Cette approche respecte la diversité des ensembles de niveaux de sévérité à

travers les différentes bibliothèques de log, qui observent que les bibliothèques de journalisation utilisent des taxonomies de niveaux de sévérité variées. De plus, il met en avant que le concept des finalités de log est plus important que les noms spécifiques des niveaux, qui divergent selon les bibliothèques. Nos heuristiques basées sur les finalités facilitent la sélection des niveaux de sévérité en se concentrant sur une approche plus globale.

NATURE DYNAMIQUE DES NIVEAUX DE SÉVÉRITÉ

Nos résultats soulignent également l'importance de comprendre la mutabilité à laquelle les niveaux de sévérité sont soumis en fonction de l'étape de développement du logiciel et de l'évolution de ses fonctionnalités. L'étude de Ding *et al.* (2015) reflète la pratique courante où des instructions de log sont laissées dans le code après leur écriture initiale, avec des ajustements de niveau souvent réalisés lors de l'intégration finale du code. En tant que mesure stratégique, nous avons observé qu'il est valide d'aggraver temporairement les niveaux de sévérité pour aider à résoudre les périodes instables et que certaines instructions de log peuvent devenir moins sévères d'un point de vue stratégique à mesure que le système évolue.

DÉFIS DE LA JOURNALISATION EXCESSIVE

La journalisation excessive, observée par Yuan *et al.* (2012b); Chen & Jiang (2017a); Hassani *et al.* (2018) comme une cause courante de surcharge en production (Zeng *et al.*, 2019; Li *et al.*, 2020a), s'est révélée être le contexte le plus fréquent d'ajustements dans notre étude. Parfois, une mesure stratégique temporaire peut devenir un *modus operandi* indésirable, causant des effets non souhaités tels qu'une production excessive de messages. Sur cette base, nous dérivons des heuristiques qui reconnaissent la fine ligne entre stratégie temporaire et sur-journalisation. Nos heuristiques conseillent des niveaux de *débogage* pour les détails

récurrents et des niveaux de sévérité plus élevés pour les événements qui, bien que rares, peuvent être significatifs pour élucider les problèmes.

LE RÔLE DE L'EXPÉRIENCE ET DES CONNAISSANCES

L'expérience a prouvé qu'elle joue un rôle fondamental dans les discussions sur les ajustements de la sévérité des logs. Elle reflète une connaissance approfondie et quotidienne de la journalisation tout en démontrant une absorption et une réflexion constante sur les principes fondamentaux de la journalisation. Le fait de retrouver, dans les rapports d'incidents, des traces de doutes partagés et de recherches de solutions pratiques s'est révélé précieux pour alimenter notre réflexion et affiner les heuristiques proposées.

LES OBSERVATIONS PRÉLIMINAIRES COMME BASES DES HEURISTIQUES

Enfin, les observations préliminaires qui ont émergé de notre analyse des ajustements du chapitre 4 ont été essentielles pour développer notre ensemble final d'heuristiques. À partir de ces observations, nous avons pu identifier des patrons d'ajustements de niveaux de sévérité, que nous avons classés en plusieurs catégories, telles que les ajustements des principes fondamentaux et les ajustements guidés par l'expérience.

En résumé, la phase prescriptive de notre étude a produit un ensemble ample d'heuristiques qui abordent à la fois les spécificités et les principes plus larges de la classification des niveaux de sévérité des logs.

5.5 MENACES À LA VALIDITÉ

Notre étude propose un ensemble d'heuristiques fondées sur des observations empiriques et des analyses qualitatives, dans le but de guider le choix et l'ajustement des niveaux de sévérité des logs. Cependant, nous reconnaissons que certaines menaces à la validité peuvent affecter la généralisation et la reproductibilité de ces heuristiques.

5.5.1 VALIDITÉ INTERNE

La validité interne de l'étude peut être compromise par certaines limitations méthodologiques. Tout d'abord, la sélection des heuristiques a été influencée par l'interprétation des données issues des rapports d'incidents et des logs, ce qui peut avoir introduit un biais interprétatif. Bien que les heuristiques aient été fondées sur des observations récurrentes et une analyse de motifs cohérents, certaines décisions de classification de la sévérité peuvent refléter des contextes spécifiques aux projets analysés. Cependant, dans les trois systèmes étudiés, il est observé qu'un rapport d'incident est créé pour chaque commit réalisé. Par conséquent, nous espérons que nos heuristiques, en grande partie dérivées de ces rapports, reflètent fidèlement les intentions des développeurs lorsqu'ils ajustent les niveaux de sévérité des instructions de log.

Une autre menace pour la validité interne réside dans la généralisation des observations issues d'un sous-ensemble de projets. Bien que l'ensemble des projets *open-source* soit bien établi, la diversité des pratiques et le niveau de maturité varient entre les systèmes, ce qui peut avoir influencé la dérivation des heuristiques. Dans les cas où les ajustements de sévérité reflètent des préférences individuelles ou des politiques organisationnelles informelles, les recommandations peuvent ne pas s'étendre et se généraliser de manière entièrement précise à d'autres systèmes similaires. Pour atténuer cette limitation, la construction des heuristiques

a suivi un processus itératif combinant différentes perspectives des rapports, telles que les heuristiques potentielles et les observations préliminaires.

5.5.2 VALIDITÉ EXTERNE

Pour la validité externe, la principale menace réside dans l'applicabilité des heuristiques à d'autres contextes de développement. Notre analyse s'est concentrée sur des systèmes *open-source* spécifiques à une seule organisation, ce qui pourrait ne pas se traduire directement dans des contextes d'entreprise ou des systèmes fermés. Les développeurs d'autres systèmes logiciels ou d'autres organisations peuvent adopter des pratiques de journalisation et des considérations différentes. De plus, les niveaux de sévérité et leur nomenclature peuvent varier sensiblement entre les bibliothèques et les frameworks de journalisation, nécessitant ainsi une adaptation des heuristiques aux spécificités de ces environnements.

Les trois projets *open-source* étudiés, bien qu'ils bénéficient d'un développement et d'une maintenance actifs, ne garantissent pas nécessairement une conformité totale aux meilleures pratiques de journalisation. Cependant, le choix de se concentrer sur des rapports d'incidents résolus vise à atténuer ce biais, car il fournit une vision plus fiable des ajustements de sévérité qui ont été appliqués avec succès et validés par des développeurs expérimentés. En outre, ces projets sont largement utilisés dans l'industrie comme composants de solutions développées par d'autres entreprises, imposant une exigence de qualité élevée aux systèmes étudiés.

Par ailleurs, bien que notre travail se concentre sur des instructions de log associées à des niveaux explicites de sévérité, il convient de noter que cette structure est largement adoptée par les bibliothèques de journalisation courantes. En effet, notre analyse (chapitre 3) a identifié une convergence entre les niveaux proposés dans 19 bibliothèques distinctes, dont

une majorité implémente au moins six niveaux de base : *Trace*, *Debug*, *Info*, *Warn*, *Error* et *Fatal*. Cette uniformité suggère une possibilité de généralisation raisonnable des heuristiques proposées, même dans des environnements logiciels différents de ceux analysés.

Cependant, certains types de logs, tels que ceux utilisés à des fins de sécurité ou de surveillance bas-niveau, peuvent ne pas intégrer de niveaux de sévérité hiérarchisés. Dans ces cas, l'application directe des heuristiques proposées devient limitée. Ainsi, notre contribution vise principalement les systèmes de journalisation où les niveaux de sévérité constituent une partie intégrante de la conception et de l'interprétation des logs.

Des études futures pourraient explorer l'application de ces heuristiques dans des systèmes plus récents ou dans des domaines spécifiques afin de vérifier l'adaptabilité de ces recommandations.

5.6 REMARQUES FINALES

Pour gérer efficacement les coûts et les avantages de la journalisation, Li *et al.* (2020a) recommandent une attribution appropriée des niveaux de sévérité des logs, soutenue par une évaluation continue et une refactorisation régulière de ces niveaux. Dans cet esprit, les heuristiques proposées dans ce chapitre visent à guider la classification, la révision et l'ajustement des niveaux de sévérité des logs. Bien que ces heuristiques ne couvrent pas tous les cas possibles, elles fournissent une base pratique pour traiter une large gamme des instructions de log et permettent de mieux différencier les niveaux de sévérité.

Ces propositions offrent un cadre pour une prise de décision plus structurée et adaptée aux environnements de développement et de production. Elles sont destinées à servir de cadre flexible pour les chercheurs et praticiens, tout en étant suffisamment précises pour orienter les choix dans la classification des instructions de log.

Par ailleurs, il convient de noter que ces heuristiques ne sont pas destinées à être appliquées de manière systématique ou mécanique dans le flux de développement quotidien. Elles visent plutôt combler une lacune des guides de style existants, en particulier lorsqu'aucune directive organisationnelle n'est disponible. Leur valeur réside dans leur potentiel à favoriser une meilleure cohérence dans les pratiques de journalisation.

Dans la suite de ce travail, nous discuterons des implications plus larges de ces résultats et examinerons les perspectives futures qu'ils ouvrent.

CONCLUSION

Le niveau de sévérité des logs est essentiel pour déterminer la pertinence et la visibilité des entrées de log, tant dans les environnements de développement que de production. Le choix correct de ces niveaux a un impact direct sur la qualité des données de journalisation générées, influençant la capacité des développeurs et des administrateurs système à diagnostiquer le comportement du système et à enquêter sur les pannes de manière efficace. Cependant, cette sélection peut s'avérer complexe en raison des multiples contextes d'utilisation et des différentes interprétations des niveaux de sévérité. Face à ces enjeux, nous avons cherché à développer des connaissances et fournir des outils pour mieux soutenir la prise de décision en ce qui concerne la sélection des niveaux de sévérité des logs dans les systèmes logiciels.

Initialement, nous avons présenté les concepts de base qui sous-tendent cette thèse dans le chapitre 1, y compris le rôle des logs dans les systèmes logiciels et des niveaux de sévérité. De plus, nous avons exploré l'évolution des bibliothèques de journalisation, les pratiques de logging et l'utilisation des outils de suivi des incidents, en mettant l'accent sur Jira.

Dans le chapitre 2, nous avons conduit une revue de la littérature, en mettant en lumière la recherche en cours sur les niveaux de sévérité des logs. D'une part, les pratiques de journalisation analysées offrent des perspectives sur les ajustements des niveaux de log effectués par les développeurs, sans toujours fournir de cadres pratiques pour orienter ces décisions. D'autre part, les approches automatisées, bien qu'utiles pour prédire ou corriger les niveaux de sévérité, manquent de considération des motivations sous-jacentes des développeurs.

Comme nous avons identifié des lacunes dans la compréhension des niveaux de sévérité, nous avons approfondi cet aspect au chapitre 3, nous appuyant sur les définitions existantes dans la littérature, les bibliothèques de journalisation et un forum de développeurs. Notre étude est pionnière en cartographiant systématiquement les sources sélectionnées et en analysant de

manière empirique les définitions, descriptions et documentations relatives aux niveaux de sévérité des logs.

En résumé, nous avons analysé 19 niveaux de sévérité issus de 42 études et de 40 bibliothèques de journalisation. Nos résultats montrent qu'il existe une redondance et une similarité sémantique entre certains niveaux. Toutefois, nous avons également observé une tendance à la convergence vers six niveaux principaux. De plus, nous avons constaté une cohérence dans l'ordre des niveaux entre les différentes bibliothèques, chaque niveau étant associé à des finalités spécifiques. À partir de cela, nous avons proposé une catégorisation, appelée « *finalités de la journalisation* », afin de mieux comprendre et structurer la pluralité des niveaux de sévérité des logs. Ces finalités servent de guide pour la prise de décision lors du choix du niveau de sévérité. Nous avons également formulé des définitions claires et nuancées pour les six niveaux les plus couramment rencontrés dans notre étude.

Bien que ces définitions et finalités représentent une première étape dans la compréhension du processus de sélection du niveau de sévérité, dans le chapitre 4 nous avons cherché à approfondir cette question en investiguant les ajustements de sévérité effectués par les développeurs. Nous avons ainsi analysé 376 versions de trois systèmes logiciels *open-source*, comprenant plus de 50 000 instructions de log. Pour cela, nous avons adopté une méthodologie en deux phases, afin d'examiner les niveaux de sévérité à travers les ajustements réalisés au cours du développement et de l'évolution des systèmes. Notre analyse du code source et des rapports d'incidents associés à ces systèmes confirme l'existence des quatre finalités principales de la journalisation : *débogage, informationnelle, avertissement* et *défaillance*.

L'observation de ces ajustements nous a permis de les regrouper en cinq grandes catégories : ajustements basés sur des principes fondamentaux, ajustements dictés par des pratiques historiques, ajustements fondés sur l'expérience, ajustements guidés par des directives, et ajus-

tements induits par des modifications des bibliothèques de journalisation. Nous avons utilisé des extraits de rapports d'incidents des systèmes analysés, fournissant ainsi plusieurs exemples pour mieux comprendre la nécessité et la logique derrière les ajustements des niveaux de sévérité des logs.

Nos résultats soulignent la prévalence des ajustements basés sur l'expérience par rapport à ceux fondés sur des théories de journalisation, la nécessité d'ajustements de sévérité pour éviter une surproduction de données, une tendance à la convergence (notamment entre les niveaux *Debug* et *Info*), ainsi que l'ajustement de la sévérité en fonction de la maturité du logiciel.

À partir de ces observations, nous avons proposé un ensemble d'heuristiques pour la classification, la révision et l'ajustement des niveaux de sévérité des logs. Bien que non exhaustives, ces heuristiques, combinées aux définitions des niveaux et des finalités, constituent une base solide pour classifier les instructions de log selon leur sévérité. Elles sont suffisamment flexibles pour s'adapter à une grande diversité d'instructions de log, tout en étant suffisamment précises pour offrir des distinctions significatives entre les différents niveaux de sévérité. Nous anticipons que ces heuristiques seront une ressource précieuse pour les chercheurs et les praticiens du domaine.

CONTRIBUTIONS PRINCIPALES

Les contributions issues de ce travail couvrent à la fois des aspects théoriques et empiriques de la journalisation. Elles visent non seulement à améliorer la compréhension des niveaux de sévérité des logs, mais aussi à fournir des outils pratiques pour guider les décisions des développeurs et opérateurs de systèmes. Voici un résumé des principales contributions :

- Une cartographie des niveaux de sévérité dans la littérature existante, permettant d'éclairer les différences et similitudes entre les définitions théoriques et pratiques.
- Une correspondance des niveaux de sévérité dans les bibliothèques de journalisation, mettant en lumière la diversité existante et les besoins de standardisation.
- Un cadre conceptuel pour les niveaux de sévérité, identifiant les niveaux essentiels et clarifiant les finalités spécifiques pour chaque niveau.
- Une description des tendances et des motifs récurrents dans les ajustements de sévérité observés dans les systèmes étudiés.
- Une analyse des motivations qui sous-tendent ces ajustements, expliquant pourquoi les développeurs modifient les niveaux de sévérité au cours du cycle de vie des systèmes.
- Un ensemble d'heuristiques destinées à aider les développeurs et les opérateurs à choisir et ajuster les niveaux de sévérité dans les environnements de développement et de production.

LIMITATIONS ET TRAVAUX FUTURS

Les contraintes et les mesures d'atténuation liées à la recherche menée dans cette thèse ont été abordées à la fin de chaque chapitre. Nous proposons ci-dessous un résumé des principales limitations générales ainsi que des orientations pour des recherches futures.

Dans cette étude, nous n'avons pas validé directement les définitions et finalités des niveaux de sévérité avec les entrées de log réelles. Des études empiriques supplémentaires sont nécessaires pour observer l'adhésion de ces définitions aux pratiques réelles de journalisation dans les systèmes logiciels, ainsi que des expériences contrôlées pour évaluer l'efficacité de ces définitions.

Un autre point important est que la cartographie des bibliothèques de journalisation ne couvre pas l'ensemble des bibliothèques existantes, ce qui peut réduire la représentativité des résultats. Cependant, nous avons cherché à sélectionner un ensemble représentatif de bibliothèques en appliquant des critères d'inclusion et d'exclusion bien définis, garantissant ainsi la robustesse des résultats obtenus.

Concernant la sélection des instructions de log, nous avons utilisé un outil automatisé (SLogAnalyzer), qui identifie ces instructions à l'aide d'expressions régulières. Cela peut entraîner l'omission de certaines instructions, mais nous avons atténué ce risque en utilisant un pipeline bien structuré pour extraire les instructions pertinentes, ainsi qu'en incluant des vérifications manuelles.

La comparaison des instructions de log entre différentes versions des systèmes peut également avoir identifié incorrectement des ajustements de sévérité, en associant des segments de code non correspondants. Pour réduire ce risque, nous avons réalisé des validations manuelles des ajustements liés aux incidents signalés, garantissant ainsi une plus grande précision dans nos analyses.

Les incidents utilisés dans l'étude ont été sélectionnés à l'aide d'expressions régulières, ce qui peut avoir exclu certains cas pertinents concernant les ajustements de sévérité ou les pratiques de journalisation. Cependant, nous avons opté pour cette méthode après avoir testé d'autres techniques et observé que l'expression régulière sélectionnait un groupe plus large pour une analyse ultérieure.

Une autre limitation réside dans la restriction de l'analyse aux versions stables des systèmes étudiés. Cela peut nous avoir fait manquer certaines tendances liées aux ajustements de sévérité au sein des versions. Pour atténuer cet impact, nous avons également analysé les incidents mentionnés dans les commentaires et les descriptions d'autres occurrences connexes.

L'interprétation des incidents signalés est intrinsèquement subjective et peut introduire des biais dans les résultats. Pour réduire ce risque, nous avons adopté une approche systématique dans laquelle différents auteurs ont analysé les incidents de manière indépendante, avant de revisiter ensemble les résultats pour aligner les interprétations et minimiser les distorsions possibles.

Enfin, une caractéristique importante de cette étude est que les trois systèmes analysés proviennent tous de l'organisation Apache, ce qui présente une limitation potentielle en termes de diversité des pratiques de journalisation. En tant qu'organisation bien établie dans la communauté *open-source*, on peut supposer qu'il existe une culture et des directives de développement spécifiques, susceptibles d'influencer de manière cohérente ses pratiques de gestion des logs et de niveaux de sévérité dans l'ensemble de ses projets. En conséquence, les heuristiques que nous avons formulées reposent en grande partie sur des pratiques partagées entre ces systèmes, qui peuvent ne pas refléter les méthodes et politiques de journalisation adoptées par d'autres grandes organisations.

Si nous avions étudié des systèmes développés par d'autres organisations, comme Google, Meta ou Netflix, les résultats pourraient présenter des variations significatives. Comme Apache, ces organisations ont probablement des approches spécifiques pour le développement, y compris la journalisation, adaptées à leurs exigences particulières de performance, de sécurité et de maintenance. Par exemple, dans des environnements de production intensifs comme ceux de Netflix, les ajustements de sévérité peuvent être davantage orientés vers l'optimisation des performances que vers un débogage détaillé, ce qui pourrait limiter l'applicabilité de certaines de nos heuristiques.

Il est donc possible que les pratiques de journalisation dans d'autres contextes requièrent des heuristiques additionnelles ou différentes. Des études complémentaires pourraient ainsi

bénéficier de l'inclusion de projets variés, issus de multiples organisations, afin de valider ou d'adapter les heuristiques proposées et de renforcer leur applicabilité dans divers contextes.

Enfin, bien que nos heuristiques aient été formulées à partir de systèmes *open-source* reposant sur des bibliothèques de journalisation avec niveaux de sévérité explicites, leur généralisation à d'autres types de logs reste limitée. Certains environnements, tels que les systèmes de journalisation orientés sécurité ou bas niveau, peuvent ne pas adopter cette structuration hiérarchique. Ainsi, notre contribution s'applique principalement aux systèmes où les niveaux de sévérité constituent une partie intégrante de l'architecture de journalisation. L'existence d'une convergence des niveaux dans la majorité des bibliothèques analysées suggère néanmoins un potentiel de généralisation à des environnements variés.

À la lumière des résultats de cette recherche, plusieurs pistes s'ouvrent pour des travaux futurs visant à approfondir et valider nos conclusions, tels que :

- Validation expérimentale des heuristiques par une étude contrôlée : Une expérimentation pourrait être menée dans un environnement contrôlé où deux groupes de développeurs seraient invités à classifier le niveau de sévérité de plusieurs instructions de logs, basées sur des cas réels ou simulés. Le premier groupe disposerait des heuristiques proposées, tandis que le second s'appuierait uniquement sur son expérience ou des recommandations générales. Les résultats permettraient de mesurer la cohérence des classifications, la dispersion des niveaux choisis et la conformité par rapport à une classification attendue. Ce protocole offrirait une première validation quantitative de l'impact des heuristiques sur la qualité des décisions de journalisation.
- Étude de cas appliquée dans un projet logiciel réel : Pour évaluer la faisabilité et l'efficacité des heuristiques dans un contexte concret de développement logiciel, une étude de cas pourrait être réalisée dans une organisation ou un projet *open-source*. L'objectif serait d'appliquer systématiquement les heuristiques lors de l'assignation des

niveaux de sévérité, puis de mesurer l'impact sur la qualité des logs produits — par exemple, en évaluant la réduction des logs inutiles en production, la clarté des journaux générés ou la rapidité de diagnostic lors d'incidents. Une comparaison avant/après fournirait des indicateurs concrets pour soutenir l'adoption des heuristiques.

- Sondage auprès de praticiens pour valider l'utilité perçue : Une enquête ciblée pourrait être réalisée auprès de développeurs et d'ingénieurs en exploitation afin de recueillir leur opinion sur la clarté, la pertinence et la complétude des heuristiques proposées. Un questionnaire structuré pourrait demander, pour chaque heuristique, si celle-ci aurait influencé leurs décisions passées ou si elle serait adoptée dans leur pratique actuelle. Cette validation externe serait essentielle pour juger de l'acceptabilité des heuristiques dans la communauté et leur potentiel d'intégration dans les guides de style de journalisation.
- Une analyse quantitative des logs générés par les systèmes étudiés comparés à nos résultats sur la distribution des niveaux de sévérité : Nous proposons une analyse approfondie des logs produits par les tests unitaires des systèmes étudiés, comparés aux résultats obtenus sur la distribution des niveaux de sévérité, étant donné le défi que représente le partage de données de log par des entreprises réelles en raison de la sensibilité des informations. Dans le cas des systèmes étudiés, comme Hadoop par exemple, l'organisation Apache met à disposition ses tests et les directives nécessaires pour les exécuter, y compris pour les versions antérieures. Grâce à l'existence de conteneurs comme Docker, il est possible de simuler les environnements requis pour exécuter ces tests, répondant ainsi aux exigences de versions spécifiques de langage. Plusieurs scénarios pourraient être explorés dans ce contexte, en configurant la production de logs pour tous les niveaux de sévérité ou uniquement pour les niveaux destinés à la production. Une évaluation préliminaire a déjà révélé une disparité notable dans la production de logs au niveau *Debug* lorsqu'on active tous les niveaux de sévérité, représentant environ 95%

d'un ensemble de près de 600 000 lignes. En revanche, en considérant uniquement les niveaux destinés à la production, environ 96% des entrées étaient classées au niveau *Info*. Ces résultats initiaux soulignent l'importance du logging dans le développement et la maintenance des systèmes. Une analyse plus approfondie de cette disparité fournirait des informations sur les pratiques de journalisation et leur optimisation pour différents contextes.

- Développement d'une théorie formelle des niveaux de sévérité de logs : Enfin, une avenue intéressante serait de considérer l'attribution d'un niveau de sévérité comme un problème d'optimisation, où il s'agirait de maximiser la pertinence des données de log tout en minimisant l'impact sur les ressources (par exemple, stockage et traitement). Cette théorie formelle, basée sur nos définitions des niveaux et des finalités de sévérité, ainsi que les heuristiques, pourrait fournir un cadre théorique pour aider les développeurs à équilibrer la quantité d'information consignée et la performance globale du système, en adaptant la production de logs aux besoins spécifiques de chaque contexte d'utilisation.

BIBLIOGRAPHIE

Adkins, H., Beyer, B., Blankinship, P., Lewandowski, P., Oprea, A. & Stubblefield, A. (2020). *Building secure and reliable systems : Best practices for designing, implementing, and maintaining systems*. Sebastopol, CA: O'Reilly Media. Repéré à https://google.github.io/building-secure-and-reliable-systems/raw/toc.html

Alves, M. & Paula, H. (2021). Identifying logging practices in open source Python containerized application projects. Dans *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, pp. 16–20. doi: 10.1145/3474624.3474631

Anu, H., Chen, J., Shi, W., Hou, J., Liang, B. & Qin, B. (2019). An approach to recommendation of verbosity log levels based on logging intention. Dans *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 125–134. IEEE. doi: 10.1109/ICSME.2019.00022

Atlassian (2024). *Jira Software – Project Management Tool*. Repéré le 5 avril 2024, à https://www.atlassian.com/software/jira

Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M. & Bier, L. (1998). Clone detection using abstract syntax trees. Dans *Proceedings of the International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 368–377. IEEE. doi: 10.1109/WCRE.2006.18

Bharkad, V. S. & Chavan, M. K. (2020). Optimizing Root Cause Analysis Time Using Smart Logging Framework for Unix and GNU/Linux Based Operating System. Dans *ICT Systems and Sustainability* (pp. 519–528). Springer. doi: 10.1007/978-981-15-0936-0_55

Bogatinovski, J., Nedelkoski, S., Acker, A., Cardoso, J. & Kao, O. (2022). Qulog : Data-driven approach for log instruction quality assessment. Dans *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pp. 275–286. doi: 10.1145/3524610.3527906

Bradner, S. (1997). *RFC2119 : Key words for use in RFCs to indicate requirement levels* publication n° 2119. Internet Engineering Task Force (IETF). Repéré le 28 octobre 2024, à https://www.rfc-editor.org/rfc/rfc2119.txt

Breiman, L. (2001). Random Forests. *Machine Learning*, *45*, 5–32. doi:

10.1023/A:1010933404324

Cândido, J., Aniche, M. & van Deursen, A. (2021). Log-based software monitoring : a systematic mapping study. *PeerJ Computer Science*, *7*, e489. doi: 10.7717/peerj-cs.489

Charmaz, K. (2006). *Constructing grounded theory : A practical guide through qualitative analysis*. London: SAGE Publications.

Chen, B. & Jiang, Z. M. (2017). Characterizing and detecting anti-patterns in the logging code. Dans *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 71–81. IEEE. doi: 10.1109/ICSE.2017.15

Chen, B. & Jiang, Z. M. (2020). Studying the use of java logging utilities in the wild. Dans *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, pp. 397–408. IEEE. doi: 10.1145/3377811.3380408

Chen, B. & Jiang, Z. M. J. (2017). Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation. *Empirical Software Engineering*, *22*(1), 330–374. doi: 10.1007/s10664-016-9429-5

Chen, B. & Jiang, Z. M. J. (2019). Extracting and studying the Logging-Code-Issue-Introducing changes in Java-based large-scale open source software systems. *Empirical Software Engineering*, *24*(4), 2285–2322. doi: 10.1007/s10664-019-09690-0

Chowdhury, S., Di Nardo, S., Hindle, A. & Jiang, Z. M. J. (2018). An exploratory study on assessing the energy impact of logging on android applications. *Empirical Software Engineering*, *23*(3), 1422–1456. doi: 10.1007/s10664-017-9545-x

Cinque, M., Cotroneo, D. & Pecchia, A. (2012). Event logs for the analysis of software failures : A rule-based approach. *IEEE Transactions on Software Engineering*, *39*(6), 806–821. doi: 10.1109/TSE.2012.67

Ding, R., Zhou, H., Lou, J.-G., Zhang, H., Lin, Q., Fu, Q., Zhang, D. & Xie, T. (2015). Log2 : A cost-aware logging mechanism for performance diagnosis. Dans *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 139–150. Repéré le 13 mai 2021, à https://www.usenix.org/system/files/conference/atc15/atc15-paper-ding.pdf

El-Masri, D., Petrillo, F., Guéhéneuc, Y.-G., Hamou-Lhadj, A. & Bouziane, A. (2020). A systematic literature review on automated log abstraction techniques. *Information and Software Technology*, *122*, 106276. doi: 10.1016/j.infsof.2020.106276

Foalem, P. L., Khomh, F. & Li, H. (2024). Studying logging practice in machine learning-based applications. *Information and Software Technology*, *170*, 107450. doi: 10.1016/j.infsof.2024.107450

Fowler, M., Beck, K., Brant, J., Opdyke, W. & Robert, D. (1999). *Refactoring : improving the design of existing code*. USA: Addison-Wesley Longman Publishing Co., Inc. doi: 10.5555/311424

Fu, Q., Zhu, J., Hu, W., Lou, J.-G., Ding, R., Lin, Q., Zhang, D. & Xie, T. (2014). Where do developers log ? an empirical study on logging practices in industry. Dans *Companion Proceedings of the 36th International Conference on Software Engineering*, pp. 24–33. doi: 10.1145/2591062.2591175

Garousi, V., Felderer, M. & Mantyla, M. (2016). The need for multivocal literature reviews in software engineering : complementing systematic literature reviews with grey literature. Dans *Proceedings of the 20th international conference on evaluation and assessment in software engineering*, pp. 1–6. doi: 10.1145/2915970.2916008

Garousi, V., Felderer, M. & Mäntylä, M. V. (2019). Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, *106*, 101–121. doi: 10.1016/j.infsof.2018.09.006

Gholamian, S. (2021). Leveraging code clones and natural language processing for log statement prediction. Dans *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1043–1047. IEEE. doi: 10.1109/ASE51524.2021.9678596

Gholamian, S. & Ward, P. A. (2020). Logging statements' prediction based on source code clones. Dans *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pp. 82–91. doi: 10.1145/3341105.3373845

Gholamian, S. & Ward, P. A. (2021). A comprehensive survey of logging in software : From logging statements automation to log mining and analysis. *arXiv preprint arXiv :2110.12489*. Repéré le 23 avril 2022, à https://arxiv.org/pdf/2110.12489

Git (2024). *Git - git-diff Documentation*. Repéré le 15 octobre 2024, à https://git-scm.com/docs/git-diff

Glaser, B. & Strauss, A. (2017). *Discovery of grounded theory : Strategies for qualitative research*. New York: Routledge.

Gomathy, M., Devi, V. K. & Meenakshi, D. (2014). Developing an error logging framework for Ruby on Rails application using AOP. Dans *2014 World Congress on Computing and Communication Technologies*, pp. 44–49. IEEE. doi: 10.1109/WCCCT.2014.19

Goulding, C. (2002). *Grounded Theory : A Practical Guide for Management, Business and Market Researchers*. London: SAGE Publications.

Goutte, C. & Gaussier, E. (2005). A probabilistic interpretation of precision, recall and F-score, with implication for evaluation. Dans *European conference on information retrieval*, pp. 345–359. Springer. doi: 10.1007/978-3-540-31865-1_25

Gülcü, C. (2003). *The complete Log4J manual*. Montreux: QOS. ch.

Hassani, M., Shang, W., Shihab, E. & Tsantalis, N. (2018). Studying and detecting log-related issues. *Empirical Software Engineering*, *23*(6), 3248–3280. doi: 10.1007/978-3-540-31865-1_25

He, P., Chen, Z., He, S. & Lyu, M. R. (2018). Characterizing the natural language descriptions in software logging statements. Dans *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 178–189. IEEE. doi: 10.1145/3238147.3238193

He, S., He, P., Chen, Z., Yang, T., Su, Y. & Lyu, M. R. (2021). A survey on automated log analysis for reliability engineering. *ACM computing surveys (CSUR)*, *54*(6), 1–37. doi: 10.1145/3460345

He, S., Zhu, J., He, P. & Lyu, M. R. (2020). Loghub : A large collection of system log datasets towards automated log analytics. *arXiv preprint arXiv :2008.06448*. Repéré à https://arxiv.org/abs/2008.06448

Kabinna, S., Bezemer, C.-P., Shang, W. & Hassan, A. E. (2016). Logging library migrations : A case study for the apache software foundation projects. Dans *Proceedings of the 13th International Conference on Mining Software Repositories*, pp. 154–164. doi: 10.1145/2901739.2901769

Kabinna, S., Bezemer, C.-P., Shang, W., Syer, M. D. & Hassan, A. E. (2018). Examining the stability of logging statements. *Empirical Software Engineering*, *23*(1), 290–333. doi: 10.1007/s10664-017-9518-0

Keele, S. (2007). *Guidelines for performing systematic literature reviews in software engineering*. Citeseer. Repéré à https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=3893625c4366d9cefb1ff647a149a0495ef470ca

Kim, T., Kim, S., Park, S. & Park, Y. (2020). Automatic recommendation to appropriate log levels. *Software : Practice and Experience*, *50*(3), 189–209. doi: 10.1002/spe.2771

Kramer, O. (2013). K-nearest neighbors. Dans *Dimensionality reduction with unsupervised nearest neighbors* (pp. 13–23). Berlin, Heidelberg: Springer. doi: 10.1007/978-3-642-38652-7_2

Laplante, P. A., Werghi, N., Kuszmavl, C. L., Verhof, C., Henderson-Sellers, B., Ganley, J. L., Sommerville, I., Omondi, A. R., Guan, L., Gori, M. *et al.* (2017). *Dictionary of computer science, engineering and technology*. Boca Raton, FL: CRC Press.

Leffler, S. J., Karels, M. & McKusick, M. K. (1984). *Measuring and improving the performance of 4.2 BSD* publication nº UCB/CSD-84-218. University of California at Berkeley. Repéré le 26 août 2021, à https://www2.eecs.berkeley.edu/Pubs/TechRpts/1984/CSD-84-218.pdf

Li, H., Chen, T.-H. P., Shang, W. & Hassan, A. E. (2018). Studying software logging using topic models. *Empirical Software Engineering*, *23*(5), 2655–2694. doi: 10.1007/s10664-018-9595-8

Li, H., Shang, W., Adams, B., Sayagh, M. & Hassan, A. E. (2020). A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering*, *47*(12), 2858–2873. doi: 10.1109/TSE.2020.2970422

Li, H., Shang, W. & Hassan, A. E. (2017). Which log level should developers choose for a new logging statement ? *Empirical Software Engineering*, *22*(4), 1684–1716. doi: 10.1109/SANER.2018.8330234

Li, H., Shang, W., Zou, Y. & Hassan, A. E. (2017). Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, *22*(4), 1831–1865. doi: 10.1109/SANER.2018.8330233

Li, H., Zhang, H., Wang, S. & Hassan, A. E. (2021). Studying the practices of logging exception stack traces in open-source software projects. *IEEE Transactions on Software Engineering*, *48*(12), 4907–4924. doi: 10.1109/TSE.2021.3129688

Li, Z., Chen, T.-H. & Shang, W. (2020). Where shall we log ? studying and suggesting logging locations in code blocks. Dans *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 361–372. doi: 10.1145/3324884.3416636

Li, Z., Chen, T.-H., Yang, J. & Shang, W. (2019). DLFinder : characterizing and detecting duplicate logging code smells. Dans *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 152–163. IEEE. doi: 10.1109/ICSE.2019.00032

Li, Z., Chen, T.-H., Yang, J. & Shang, W. (2021). Studying duplicate logging statements and their relationships with code clones. *IEEE Transactions on Software Engineering*, *48*(7), 2476–2494. doi: 10.1109/ICSE.2019.00032

Li, Z., Li, H., Chen, T.-H. & Shang, W. (2021). DeepLV : Suggesting log levels using ordinal based neural networks. Dans *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1461–1472. IEEE. doi: 10.1109/ICSE43902.2021.00131

Lin, Q., Zhang, H., Lou, J.-G., Zhang, Y. & Chen, X. (2016). Log clustering based problem identification for online service systems. Dans *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 102–111. IEEE. doi: 10.1145/2889160.2889232

Linux (2024). *Linux Kernel Documentation | printk() basics*. Repéré le 15 octobre 2024, à https://www.kernel.org/doc/html/next/core-api/printk-basics.html

Liu, J., Zeng, J., Wang, X., Ji, K. & Liang, Z. (2022). Tell : log level suggestions via modeling multi-level code block information. Dans *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 27–38. doi: 10.1145/3533767.3534379

Liu, Z., Xia, X., Lo, D., Xing, Z., Hassan, A. E. & Li, S. (2019). Which variables should I log ? *IEEE Transactions on Software Engineering*, pp. 2012–2031. doi: 10.1109/TSE.2019.2941943

Lonvick, C. (2001). *RFC3164 : The BSD Syslog Protocol*. RFC Editor. Repéré le 20 janvier 2024, à https://tools.ietf.org/html/rfc3164

Lupu, E. (2010). *Under the hood of Java strings*. Repéré le 27 septembre 2024, à https://blog.eyallupu.com/2010/09/under-hood-of-java-strings.html

Ma, L. & Zhang, Y. (2015). Using Word2Vec to process big text data. Dans *2015 IEEE International Conference on Big Data (Big Data)*, pp. 2895–2897. IEEE. doi: 10.1109/BigData.2015.7364114

Maplesden, D., Tempero, E., Hosking, J. & Grundy, J. C. (2015). Performance analysis for object-oriented software : A systematic mapping. *IEEE Transactions on Software Engineering*, *41*(7), 691–710. doi: 10.1109/TSE.2015.2396514

Mastropaolo, A., Ferrari, V., Pascarella, L. & Bavota, G. (2024). Log statements generation via deep learning : Widening the support provided to developers. *Journal of Systems and Software*, *210*, 111947. doi: 10.1016/j.jss.2023.111947

Mastropaolo, A., Pascarella, L. & Bavota, G. (2022). Using deep learning to generate complete log statements. Dans *Proceedings of the 44th International Conference on Software Engineering*, pp. 2279–2290. doi: 10.1145/3510003.3511561

Mathew, A., Amudha, P. & Sivakumari, S. (2021). Deep learning techniques : An overview. Dans A. E. Hassanien, R. Bhatnagar, & A. Darwish (Éds.). *Advanced machine learning technologies and applications*, pp. 599–608., Singapore. Springer Singapore. doi: 10.1007/978-981-15-3383-9_54

Mendes, E. & Petrillo, F. (2021). Log severity levels matter : A multivocal mapping. Dans *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pp. 1002–1013. IEEE. doi: 10.1109/QRS54544.2021.00109

Mendes, E., Vasconcelos, M. M., Petrillo, F. & Hallé, S. (2024). From description to prescription : Unraveling log Sseverity adjustments in open source software. *Journal of Systems and Software*. Sous révision.

Mendes de Oliveira, E. (2024). *Dataset pour le cadre conceptuel des niveaux de sévérité des logs*. Dataset, doi: 10.6084/m9.figshare.27620220.v1

Mendes de Oliveira, E. (2024). *Exploratory dataset on log severity adjustments in open-source software*. Dataset, doi: 10.6084/m9.figshare.26253776.v2

Miranskyy, A., Hamou-Lhadj, A., Cialini, E. & Larsson, A. (2016). Operational-log analysis for big data systems : Challenges and solutions. *IEEE Software*, *33*(2), 52–59. doi: 10.1109/MS.2016.33

Murphy-Hill, E., Zimmermann, T., Bird, C. & Nagappan, N. (2014). The design space of bug fixes and how developers navigate it. *IEEE Transactions on Software Engineering*, *41*(1), 65–81. doi: 10.1109/TSE.2014.2357438

Myerson, J., Green, L. & Warusawitharana, M. (2001). Area under the curve as a measure of discounting. *Journal of the experimental analysis of behavior*, *76*(2), 235–243. doi: 10.1109/TSE.2014.2357438

Obrebski, D. & Sosnowski, J. (2019). Log based analysis of software application operation. Dans *International Conference on Dependability and Complex Systems*, pp. 371–382. Springer. doi: 10.1007/978-3-030-19501-4_37

Oliner, A., Ganapathi, A. & Xu, W. (2012). Advances and challenges in log analysis. *Communications of the ACM*, *55*(2), 55–61. doi: 10.1145/2076450.2076466

Oracle (2018). *Throwable (Java Platform SE 8)*. Repéré le 20 novembre 2023, à https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html

Ortu, M., Destefanis, G., Kassab, M. & Marchesi, M. (2015). Measuring and understanding the effectiveness of jira developers communities. Dans *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*, pp. 3–10. IEEE. doi: 10.1109/WETSoM.2015.10

Ouatiti, Y. E., Sayagh, M., Kerzazi, N. & Hassan, A. E. (2022). An empirical study on log level prediction for multi-component systems. *IEEE Transactions on Software Engineering*, *49*(2), 473–484. doi: 10.1109/TSE.2022.3154672

Papineni, K., Roukos, S., Ward, T. & Zhu, W.-J. (2002). Bleu : a method for automatic evaluation of machine translation. Dans *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318. doi: 10.1109/TSE.2022.3154672

Patel, K., Faccin, J., Hamou-Lhadj, A. & Nunes, I. (2022). The sense of logging in the linux kernel. *Empirical Software Engineering*, *27*(6), 153. doi: 10.1007/s10664-022-10136-3

Pecchia, A., Cinque, M., Carrozza, G. & Cotroneo, D. (2015). Industry practices and event logging : asessment of a critical software development Process. Dans *Proceedings of the International Conference on Software Engineering*, Vol. 2, pp. 169–178. doi: 10.1109/ICSE.2015.145

Rong, G., Gu, S., Zhang, H., Shao, D. & Liu, W. (2018). How is logging practice implemented in open source software projects ? a preliminary exploration. Dans *2018 25th Australasian Software Engineering Conference (ASWEC)*, pp. 171–180. IEEE.

Roudjane, M. (2023). *Détection des écarts de tendance et analyse prédictive pour le traitement des flux d'événements en temps réel*. (Thèse de doctorat). Repéré à https://constellation.uqac.ca/id/eprint/9191/

Roudjane, M., Rebaïne, D., Khoury, R. & Hallé, S. (2021). Detecting trend deviations with generic stream processing patterns. *Information Systems*, *101*, 101446. doi: 10.1016/j.is.2019.101446

Sage, K. (2019). *Concise Guide to Object-Oriented Programming : An Accessible Approach Using Java*. Undergraduate Topics in Computer Science. Cham, Switzerland: Springer. doi: 10.1007/978-3-030-13304-7

Salfner, F., Tschirpke, S. & Malek, M. (2004). Comprehensive logfiles for autonomic systems. Dans *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, p. 211. IEEE. doi: 10.1109/ipdps.2004.1303243

Schmidt, K., Phillips, C. & Chuvakin, A. (2012). *Logging and log management : the authoritative guide to understanding the concepts surrounding logging and log management*. Waltham, MA: Syngress. doi: 10.1016/C2010-0-65241-2

Shang, W., Nagappan, M. & Hassan, A. E. (2015, 1). Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, pp. 1–27. doi: 10.1007/s10664-013-9274-8

Singhal, A. *et al.* (2001). Modern information retrieval : A brief overview. *IEEE Data Engineering Bulletin*, *24*(4), 35–43. Repéré à http://sites.computer.org/debull/A01dec/issue1.htm

Sommerville, I. (2015). *Software engineering* (10th). Harlow: Pearson Education Limited.

Sommerville, I. (2020). *Engineering software products : an introduction to modern software engineering / Ian Sommerville.* Hoboken, NJ: Pearson.

Tan, Pang-Ning, T., Steinbach, M. & Kumar, V. (2005). *Introduction to data mining*. Boston, MA: Pearson.

Tang, Y., Spektor, A., Khatchadourian, R. & Bagherzadeh, M. (2022). Automated evolution of feature logging statement levels using git histories and degree of interest. *Science of Computer Programming*, *214*, 102724. doi: 10.1016/j.scico.2021.102724

Tang, Y., Spektor, A., Khatchadourian, R. & Bagherzadeh, M. (2022). A tool for rejuvenating feature logging levels via Git histories and degree of interest. Dans *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering : Companion Proceedings*, pp. 21–25. doi: 10.1016/j.scico.2021.102724

Vasconcellos, M. M. (2023). *Vers l'évaluation de la stabilité des systèmes à l'aide de la densité des logs* [Mémoire de maîtrise]. Repéré à https://constellation.uqac.ca/id/eprint/9327/

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. & Polosukhin, I. (2017). Attention is all you need. Dans *Advances in neural information processing systems*, Vol. 30. Repéré à https://proceedings.neurips.cc/paper_files/paper/2017

Wilks, D. S. (2011). *Statistical methods in the atmospheric sciences*. Boston: Academic press. doi: 10.1016/C2017-0-03921-6

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. Dans *18th Int. Conference on Evaluation and Assessment in Software Engineering (EASE '14)*, pp. 38 :1–38 :10. doi: 10.1145/2601248.2601268

Wohlin, C., Mendes, E., Felizardo, K. R. & Kalinowski, M. (2020). Guidelines for the search strategy to update systematic literature reviews in software engineering. *Information and software technology*, *127*, 106366. doi: 10.1016/j.infsof.2020.106366

Xu, J., Cui, Z., Zhao, Y., Zhang, X., He, S., He, P., Li, L., Kang, Y., Lin, Q., Dang, Y. *et al.* (2024). UniLog : Automatic Logging via LLM and In-Context Learning. Dans *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pp. 1–12. doi: 10.1145/3597503.3623326

Yang, N., Schiffelers, R. & Lukkien, J. (2021). An interview study of how developers use execution logs in embedded software engineering. Dans *2021 IEEE/ACM 43rd International Conference on Software Engineering : Software Engineering in Practice (ICSE-SEIP)*, pp. 61–70. IEEE. doi: 10.1109/icse-seip52600.2021.00015

Yao, K., Li, H., Shang, W. & Hassan, A. E. (2020). A study of the performance of general compressors on log files. *Empirical Software Engineering*, *25*(5), 3043–3085. doi: 10.1007/s10664-020-09822-x

Yuan, D., Park, S., Huang, P., Liu, Y., Lee, M. M., Tang, X., Zhou, Y. & Savage, S. (2012). Be conservative : Enhancing failure diagnosis with proactive logging. Dans *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pp. 293–306. USENIX Association. Repéré à https://www.usenix.org/conference/osdi12/technical-sessions/presentation/yuan

Yuan, D., Park, S. & Zhou, Y. (2012). Characterizing logging practices in open-source software. Dans *2012 34th International Conference on Software Engineering (ICSE)*, pp.

102–112. IEEE. doi: 10.1109/ICSE.2012.6227202

Yuan, D., Zheng, J., Park, S., Zhou, Y. & Savage, S. (2012). Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)*, *30*(1), 1–28. doi: 10.1145/2248487.1950369

Yuan, Y., Shi, W., Liang, B. & Qin, B. (2019). An Approach to Cloud Execution Failure Diagnosis Based on Exception Logs in OpenStack. Dans *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 124–131. IEEE. doi: 10.1109/CLOUD.2019.00031

Zeng, Y., Chen, J., Shang, W. & Chen, T.-H. P. (2019). Studying the characteristics of logging practices in mobile apps : a case study on F-Droid. *Empirical Software Engineering*, *24*(6), 3394–3434. doi: 10.1007/s10664-019-09687-9

Zhang, H., Babar, M. A. & Tell, P. (2011). Identifying relevant studies in software engineering. *Information and Software Technology*, *53*(6), 625–637. doi: 10.1016/j.infsof.2010.12.010

Zhang, H., Tang, Y., Lamothe, M., Li, H. & Shang, W. (2022). Studying logging practice in test code. *Empirical Software Engineering*, *27*(4), 83. doi: 10.1007/s10664-022-10139-0

Zhao, X., Rodrigues, K., Luo, Y., Stumm, M., Yuan, D. & Zhou, Y. (2017). Log20 : Fully automated optimal placement of log printing statements under specified overhead threshold. Dans *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 565–581. doi: 10.1145/3132747.3132778

APPENDICE A

HADOOP ISSUES

#	ID	URL	Titre
#1	HADOOP-96	https ://issues.apache.org/jira/browse/HADOOP-96	name server should log decisions that affect data : block creation, removal, replication
#2	HADOOP-211	https ://issues.apache.org/jira/browse/HADOOP-211	logging improvements for Hadoop
#3	HADOOP-1034	https ://issues.apache.org/jira/browse/HADOOP-1034	RuntimeException and Error not catched in DataNode.DataXceiver.run()
#4	HADOOP-7858	https ://issues.apache.org/jira/browse/HADOOP-7858	Drop some info logging to DEBUG level in IPC, metrics, and HTTP
#5	HADOOP-8075	https ://issues.apache.org/jira/browse/HADOOP-8075	Lower native-hadoop library log from info to debug
#6	HADOOP-8932	https ://issues.apache.org/jira/browse/HADOOP-8932	JNI-based user-group mapping modules can be too chatty on lookup failures
#7	HADOOP-9135	https ://issues.apache.org/jira/browse/HADOOP-9135	JniBasedUnixGroupsMappingWithFallback should log at debug rather than info during fallback
#8	HADOOP-9582	https ://issues.apache.org/jira/browse/HADOOP-9582	Non-existent file to "hadoop fs -conf" doesn’t throw error
#9	HADOOP-10015	https ://issues.apache.org/jira/browse/HADOOP-10015	UserGroupInformation prints out excessive ERROR warnings
#10	HADOOP-10274	https ://issues.apache.org/jira/browse/HADOOP-10274	Lower the logging level from ERROR to WARN for UGI.doAs method
#11	HADOOP-10343	https ://issues.apache.org/jira/browse/HADOOP-10343	Change info to debug log in LossyRetryInvocationHandler
#12	HADOOP-10466	https ://issues.apache.org/jira/browse/HADOOP-10466	Lower the log level in UserGroupInformation
#13	HADOOP-10657	https ://issues.apache.org/jira/browse/HADOOP-10657	Have RetryInvocationHandler log failover attempt at INFO level
#14	HADOOP-11085	https ://issues.apache.org/jira/browse/HADOOP-11085	Excessive logging by org.apache.hadoop.util.Progress when value is NaN

Continué à la page suivante

#	ID	URL	Titre
#15	HADOOP-12789	https ://issues.apache.org/jira/browse/HADOOP-12789	log classpath of ApplicationClassLoader at INFO level
#16	HADOOP-13552	https ://issues.apache.org/jira/browse/HADOOP-13552	RetryInvocationHandler logs all remote exceptions
#17	HADOOP-14539	https ://issues.apache.org/jira/browse/HADOOP-14539	Move commons logging APIs over to slf4j in hadoop-common
#18	HADOOP-14987	https ://issues.apache.org/jira/browse/HADOOP-14987	Improve KMSClientProvider log around delegation token checking
#19	HADOOP-15441	https ://issues.apache.org/jira/browse/HADOOP-15441	Log kms url and token service at debug level.
#20	HADOOP-15942	https ://issues.apache.org/jira/browse/HADOOP-15942	Change the logging level form DEBUG to ERROR for RuntimeErrorException in JMXJsonServlet
#21	HADOOP-17597	https ://issues.apache.org/jira/browse/HADOOP-17597	Add option to downgrade S3A rejection of Syncable to warning
#22	HADOOP-17836	https ://issues.apache.org/jira/browse/HADOOP-17836	Improve logging on ABFS error reporting
#23	HADOOP-18065	https ://issues.apache.org/jira/browse/HADOOP-18065	ExecutorHelper.logThrowableFromAfterExecute() is too noisy.
#24	HADOOP-18574	https ://issues.apache.org/jira/browse/HADOOP-18574	Changing log level of IOStatistics increment to make the DEBUG logs less noisy
#25	HDFS-6085	https ://issues.apache.org/jira/browse/HDFS-6085	Improve CacheReplicationMonitor log messages a bit
#26	HDFS-6998	https ://issues.apache.org/jira/browse/HDFS-6998	warning message 'ssl.client.truststore.location has not been set' gets printed for hftp command
#27	HDFS-8659	https ://issues.apache.org/jira/browse/HDFS-8659	Block scanner INFO message is spamming logs
#28	HDFS-9906	https ://issues.apache.org/jira/browse/HDFS-9906	Remove spammy log spew when a datanode is restarted
#29	HDFS-10377	https ://issues.apache.org/jira/browse/HDFS-10377	CacheReplicationMonitor shutdown log message should use INFO level.
#30	HDFS-10752	https ://issues.apache.org/jira/browse/HDFS-10752	Several log refactoring/improvement suggestion in HDFS

Continué à la page suivante

#	ID	URL	Titre
#31	HDFS-11054	https ://issues.apache.org/jira/browse/HDFS-11054	Suppress verbose log message in BlockPlacementPolicyDefault
#32	HDFS-11593	https ://issues.apache.org/jira/browse/HDFS-11593	Change SimpleHttpProxyHandler#exceptionCaught log level from info to debug
#33	HDFS-13692	https ://issues.apache.org/jira/browse/HDFS-13692	StorageInfoDefragmenter floods log when compacting StorageInfo TreeSet
#34	HDFS-13695	https ://issues.apache.org/jira/browse/HDFS-13695	Move logging to slf4j in HDFS package
#35	HDFS-14238	https ://issues.apache.org/jira/browse/HDFS-14238	A log in NNThroughputBenchmark should change log level to "INFO" instead of "ERROR"
#36	HDFS-14339	https ://issues.apache.org/jira/browse/HDFS-14339	Inconsistent log level practices in RpcProgramNfs3.java
#37	HDFS-14340	https ://issues.apache.org/jira/browse/HDFS-14340	Lower the log level when can't get postOpAttr
#38	HDFS-14395	https ://issues.apache.org/jira/browse/HDFS-14395	Remove WARN Logging From Interrupts in DataStreamer
#39	HDFS-14521	https ://issues.apache.org/jira/browse/HDFS-14521	Suppress setReplication logging.
#40	HDFS-14759	https ://issues.apache.org/jira/browse/HDFS-14759	HDFS cat logs an info message
#41	HDFS-14760	https ://issues.apache.org/jira/browse/HDFS-14760	Log INFO mode if snapshot usage and actual usage differ
#42	HDFS-15197	https ://issues.apache.org/jira/browse/HDFS-15197	[SBN read] Change ObserverRetryOnActiveException log to debug
#43	MAPREDUCE-3184	https ://issues.apache.org/jira/browse/MAPREDUCE-3184	Improve handling of fetch failures when a tasktracker is not responding on HTTP
#44	MAPREDUCE-3265	https ://issues.apache.org/jira/browse/MAPREDUCE-3265	Reduce log level on MR2 IPC construction, etc
#45	MAPREDUCE-3348	https ://issues.apache.org/jira/browse/MAPREDUCE-3348	mapred job -status fails to give info even if the job is present in History
#46	MAPREDUCE-3692	https ://issues.apache.org/jira/browse/MAPREDUCE-3692	yarn-resourcemanager out and log files can get big
#47	MAPREDUCE-3748	https ://issues.apache.org/jira/browse/MAPREDUCE-3748	Move CS related nodeUpdate log messages to DEBUG

Continué à la page suivante

#	ID	URL	Titre
#48	MAPREDUCE-4570	https ://issues.apache.org/jira/browse/MAPREDUCE-4570	ProcfsBasedProcessTree#construct-ProcessInfo() prints a warning if procfsDir/<pid>/stat is not found.
#49	MAPREDUCE-4614	https ://issues.apache.org/jira/browse/MAPREDUCE-4614	Simplify debugging a job's tokens
#50	MAPREDUCE-5766	https ://issues.apache.org/jira/browse/MAPREDUCE-5766	Ping messages from attempts should be moved to DEBUG
#51	MAPREDUCE-6971	https ://issues.apache.org/jira/browse/MAPREDUCE-6971	Moving logging APIs over to slf4j in hadoop-mapreduce-client-app
#52	MAPREDUCE-6983	https ://issues.apache.org/jira/browse/MAPREDUCE-6983	Moving logging APIs over to slf4j in hadoop-mapreduce-client-core
#53	MAPREDUCE-6997	https ://issues.apache.org/jira/browse/MAPREDUCE-6997	Moving logging APIs over to slf4j in hadoop-mapreduce-client-hs
#54	MAPREDUCE-6998	https ://issues.apache.org/jira/browse/MAPREDUCE-6998	Moving logging APIs over to slf4j in hadoop-mapreduce-client-jobclient
#55	MAPREDUCE-7022	https ://issues.apache.org/jira/browse/MAPREDUCE-7022	Fast fail rogue jobs based on task scratch dir size
#56	MAPREDUCE-7063	https ://issues.apache.org/jira/browse/MAPREDUCE-7063	Fix log level inconsistency in CombineFileInputFormat.java
#57	YARN-1022	https ://issues.apache.org/jira/browse/YARN-1022	Unnecessary INFO logs in AMRMClientAsync
#58	YARN-1608	https ://issues.apache.org/jira/browse/YARN-1608	LinuxContainerExecutor has a few DEBUG messages at INFO level
#59	YARN-1839	https ://issues.apache.org/jira/browse/YARN-1839	Capacity scheduler preempts an AM out. AM attempt 2 fails to launch task container with SecretManager$InvalidToken : No NMToken sent
#60	YARN-1892	https ://issues.apache.org/jira/browse/YARN-1892	Excessive logging in RM
#61	YARN-2213	https ://issues.apache.org/jira/browse/YARN-2213	Change proxy-user cookie log in AmIpFilter to DEBUG
#62	YARN-2704	https ://issues.apache.org/jira/browse/YARN-2704	Localization and log-aggregation will fail if hdfs delegation token expired after token-max-life-time
#63	YARN-3350	https ://issues.apache.org/jira/browse/YARN-3350	YARN RackResolver spams logs with messages at info level
#64	YARN-4115	https ://issues.apache.org/jira/browse/YARN-4115	Reduce loglevel of ContainerManagementProtocolProxy to Debug

Continué à la page suivante

#	ID	URL	Titre
#65	YARN-5693	https ://issues.apache.org/jira/browse/YARN-5693	Reduce loglevel to Debug in ContainerManagementProtocolProxy and AMRMClientImpl
#66	YARN-6068	https ://issues.apache.org/jira/browse/YARN-6068	Log aggregation get failed when NM restart even with recovery
#67	YARN-6873	https ://issues.apache.org/jira/browse/YARN-6873	Moving logging APIs over to slf4j in hadoop-yarn-server-applicationhistoryservice
#68	YARN-6957	https ://issues.apache.org/jira/browse/YARN-6957	Moving logging APIs over to slf4j in hadoop-yarn-server-sharedcachemanager
#69	YARN-7047	https ://issues.apache.org/jira/browse/YARN-7047	Moving logging APIs over to slf4j in hadoop-yarn-server-nodemanager
#70	YARN-7243	https ://issues.apache.org/jira/browse/YARN-7243	Moving logging APIs over to slf4j in hadoop-yarn-server-resourcemanager
#71	YARN-7407	https ://issues.apache.org/jira/browse/YARN-7407	Moving logging APIs over to slf4j in hadoop-yarn-applications
#72	YARN-7477	https ://issues.apache.org/jira/browse/YARN-7477	Moving logging APIs over to slf4j in hadoop-yarn-common
#73	YARN-7727	https ://issues.apache.org/jira/browse/YARN-7727	Incorrect log levels in few logs with QueuePriorityContainerCandidateSelector
#74	YARN-8459	https ://issues.apache.org/jira/browse/YARN-8459	Improve Capacity Scheduler logs to debug invalid states
#75	YARN-9349	https ://issues.apache.org/jira/browse/YARN-9349	When doTransition() method occurs exception, the log level practices are inconsistent
#76	YARN-10369	https ://issues.apache.org/jira/browse/YARN-10369	Make NMTokenSecretManagerInRM sending NMToken for nodeId DEBUG
#77	YARN-10997	https ://issues.apache.org/jira/browse/YARN-10997	Revisit allocation and reservation logging

APPENDICE B

HBASE ISSUES

#	ID	URL	Titre
#1	HBASE-5582	https ://issues.apache.org/jira/browse/HBASE-5582	"No HServerInfo found for" should be a WARNING message
#2	HBASE-6023	https ://issues.apache.org/jira/browse/HBASE-6023	Normalize security audit logging level with Hadoop
#3	HBASE-7037	https ://issues.apache.org/jira/browse/HBASE-7037	ReplicationPeer logs at WARN level aborting server instead of at FATAL
#4	HBASE-7214	https ://issues.apache.org/jira/browse/HBASE-7214	CleanerChore logs too much, so much so it obscures all else that is going on
#5	HBASE-8940	https ://issues.apache.org/jira/browse/HBASE-8940	TestRegionMergeTransactionOnCluster-testWholesomeMerge may fail due to race in opening region
#6	HBASE-9120	https ://issues.apache.org/jira/browse/HBASE-9120	ClassFinder logs errors that are not
#7	HBASE-9371	https ://issues.apache.org/jira/browse/HBASE-9371	Eliminate log spam when tailing files
#8	HBASE-10092	https ://issues.apache.org/jira/browse/HBASE-10092	Move to slf4j
#9	HBASE-10906	https ://issues.apache.org/jira/browse/HBASE-10906	Change error log for NamingException in TableInputFormatBase to WARN level
#10	HBASE-12419	https ://issues.apache.org/jira/browse/HBASE-12419	"Partial cell read caused by EOF" ERRORs on replication source during replication
#11	HBASE-12461	https ://issues.apache.org/jira/browse/HBASE-12461	FSVisitor logging is excessive
#12	HBASE-12539	https ://issues.apache.org/jira/browse/HBASE-12539	HFileLinkCleaner logs are uselessly noisy
#13	HBASE-13675	https ://issues.apache.org/jira/browse/HBASE-13675	ProcedureExecutor completion report should be at DEBUG log level
#14	HBASE-14042	https ://issues.apache.org/jira/browse/HBASE-14042	Fix FATAL level logging in FSHLog where logged for non fatal exceptions
#15	HBASE-15582	https ://issues.apache.org/jira/browse/HBASE-15582	SnapshotManifestV1 too verbose when there are no regions
#16	HBASE-15954	https ://issues.apache.org/jira/browse/HBASE-15954	REST server should log requests with TRACE instead of DEBUG

Continué à la page suivante

#	ID	URL	Titre
#17	HBASE-16220	https ://issues.apache.org/jira/browse/HBASE-16220	Demote log level for "HRegionFileSystem - No StoreFiles for" messages to TRACE
#18	HBASE-17540	https ://issues.apache.org/jira/browse/HBASE-17540	Change SASL server GSSAPI callback log line from DEBUG to TRACE in RegionServer to reduce log volumes in DEBUG mode
#19	HBASE-20447	https ://issues.apache.org/jira/browse/HBASE-20447	Only fail cacheBlock if block collisions aren't related to next block metadata
#20	HBASE-20554	https ://issues.apache.org/jira/browse/HBASE-20554	"WALs outstanding" message from CleanerChore is noisy
#21	HBASE-20665	https ://issues.apache.org/jira/browse/HBASE-20665	"Already cached block XXX" message should be DEBUG
#22	HBASE-20701	https ://issues.apache.org/jira/browse/HBASE-20701	too much logging when balancer runs from BaseLoadBalancer
#23	HBASE-20770	https ://issues.apache.org/jira/browse/HBASE-20770	WAL cleaner logs way too much ; gets clogged when lots of work to do
#24	HBASE-21524	https ://issues.apache.org/jira/browse/HBASE-21524	Unnecessary DEBUG log in ConnectionImplementation#isTableEnabled
#25	HBASE-23047	https ://issues.apache.org/jira/browse/HBASE-23047	ChecksumUtil.validateChecksum logs an INFO message inside a "if(LOG.isTraceEnabled())" block.
#26	HBASE-23250	https ://issues.apache.org/jira/browse/HBASE-23250	Log message about CleanerChore delegate initialization should be at INFO
#27	HBASE-23687	https ://issues.apache.org/jira/browse/HBASE-23687	DEBUG logging cleanup
#28	HBASE-24524	https ://issues.apache.org/jira/browse/HBASE-24524	SyncTable logging improvements
#29	HBASE-25483	https ://issues.apache.org/jira/browse/HBASE-25483	set the loadMeta log level to debug.
#30	HBASE-25556	https ://issues.apache.org/jira/browse/HBASE-25556	Frequent replication "Encountered a malformed edit" warnings
#31	HBASE-25642	https ://issues.apache.org/jira/browse/HBASE-25642	Fix or stop warning about already cached block
#32	HBASE-26189	https ://issues.apache.org/jira/browse/HBASE-26189	Reduce log level of CompactionProgress notice to DEBUG
#33	HBASE-26443	https ://issues.apache.org/jira/browse/HBASE-26443	Some BaseLoadBalancer log lines should be at DEBUG level
#34	HBASE-27079	https ://issues.apache.org/jira/browse/HBASE-27079	Lower some DEBUG level logs in ReplicationSourceWALReader to TRACE

Continué à la page suivante

#	ID	URL	Titre
#35	HBASE-27391	https ://issues.apache.org/jira/browse/HBASE-27391	Downgrade ERROR log to DEBUG in ConnectionUtils.updateStats
#36	HBASE-27588	https ://issues.apache.org/jira/browse/HBASE-27588	"Instantiating StoreFileTracker impl" INFO level logging is too chatty

APPENDICE C

KAFKA ISSUES

#	ID	URL	Titre
#1	KAFKA-5704	https ://issues.apache.org/jira/browse/KAFKA-5704	Auto topic creation causes failure with older clusters
#2	KAFKA-4829	https ://issues.apache.org/jira/browse/KAFKA-4829	Improve logging of StreamTask commits
#3	KAFKA-6802	https ://issues.apache.org/jira/browse/KAFKA-6802	Improve logging when topics aren't known and assignments skipped
#4	KAFKA-9540	https ://issues.apache.org/jira/browse/KAFKA-9540	Application getting "Could not find the standby task 0_4 while closing it" error
#5	KAFKA-13037	https ://issues.apache.org/jira/browse/KAFKA-13037	"Thread state is already PENDING_SHUTDOWN" log spam
#6	KAFKA-13669	https ://issues.apache.org/jira/browse/KAFKA-13669	Log messages for source tasks with no offsets to commit are noisy and confusing