Supporting Practitioners in Managing Horizontal and Vertical Dependencies in Large-scale Software Systems

by

Ali ARABAT

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
Ph.D.

MONTREAL, AUGUST 17, 2026

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Mr. Mohammed Sayagh, Thesis supervisor
Department of Software Engineering and IT, École de technologie supérieure

Mr. Souheil-Antoine Tahan, Chair, Board of Examiners
Department of Mechanical Engineering, École de technologie supérieure

Mr. Ali Ouni, Member of the Jury
Department of Software Engineering and IT, École de technologie supérieure

Mr. Mohammad Hamdaqa, External Independent Examiner
Department of Computer Engineering and Software Engineering, Polytechnique Montréal

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON AUGUST 4, 2026

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to the One who granted me strength, patience, and all the means of support that have enabled me to reach this stage of my life.

I am sincerely grateful to my supervisor, Professor Mohammed Sayagh, for his invaluable supervision, inspiring guidance, and constructive feedback throughout my research journey. His continuous support, countless hours of discussions, and careful reviews of my work have been essential to the completion of this thesis.

I am also profoundly thankful to my colleagues and co-authors whose contributions have greatly enriched this work: Professor Jameleddine Hassine, Professor Sarah Nadi, and Professor Artur Andrzejak.

My gratitude extends to the members of the jury for kindly accepting to read, evaluate, and provide feedback on this thesis.

Above all, I am deeply indebted to my mother, father, brother, and sisters, as well as my late sister, for their unwavering emotional and financial support throughout my life. Finally, I wish to thank my friends in Morocco and Canada for their encouragement and the many memorable moments we have shared during this journey.

Assister les développeurs dans la gestion des dépendances horizontales et verticales au sein des systèmes logiciels à grande échelle

Ali ARABAT

RÉSUMÉ

Les systèmes logiciels modernes reposent aujourd'hui sur des composants interconnectés. Un système logiciel peut être composé de plusieurs composants interagissant entre eux, tels que les systèmes fondés sur des plugins, les architectures orientées services (SOA) et les applications à microservices. Ces interactions visent à fournir des fonctionnalités complémentaires qui, ensemble, offrent un service, et peuvent nécessiter des changements coordonnés entre composants, ce que nous désignons par le terme de **dépendances horizontales**. Parallèlement, un composant d'un système logiciel peut utiliser un ensemble de dépendances externes qui se manifestent sous la forme de bibliothèques externes et qui fournissent un ensemble d'utilitaires pour implémenter un service. Par exemple, au lieu de réimplémenter des utilitaires de système de fichiers, les développeurs peuvent facilement importer une dépendance offrant les mêmes fonctionnalités. Les changements apportés à ces bibliothèques externes peuvent nécessiter des adaptations correspondantes dans le projet client, et nous désignons ces relations par le terme de **dépendances verticales**.

Nous définissons concrètement les *dépendances horizontales* comme des dépendances entre les composants et les services d'un même système logiciel, tandis que les *dépendances verticales* sont définies comme des dépendances entre un projet et les bibliothèques tierces spécifiées dans le fichier manifeste du projet (par exemple, package.json). Dans cette thèse, les dépendances horizontales et verticales sont considérées comme deux formes complémentaires de dépendances de maintenance : l'une à l'intérieur de la frontière du projet et l'autre à travers cette frontière.

Bien que les interconnexions horizontales et verticales entre les composants offrent des avantages significatifs, elles introduisent également des défis complexes qui se manifestent par la charge de maintenance liée à la synchronisation entre les composants et à la gestion des dépendances externes. D'un point de vue horizontal, les développeurs doivent fréquemment se coordonner au-delà des limites des composants — en synchronisant les changements avec les équipes travaillant sur d'autres composants, en collaborant sur du code complémentaire réparti sur plusieurs composants pour corriger un bogue ou fournir une nouvelle fonctionnalité, et en s'assurant qu'un changement effectué dans un composant est correctement validé par des tests dans d'autres composants. Cette coordination inter-équipes ajoute des frictions au développement quotidien et, lorsqu'elle est négligée, peut entraîner des échecs de compilation, des fusions partielles et des échecs de pipelines CI/CD. D'un point de vue vertical, les bibliothèques tierces publient fréquemment des mises à jour contenant de nouvelles fonctionnalités, des corrections de bogues et des correctifs de sécurité. Ne pas intégrer ces mises à jour rapidement peut rendre les projets clients instables, introduire des failles de sécurité exploitables et exposer le code source du client à des risques.

Cette thèse étudie les dépendances de maintenance dans les systèmes à grande échelle composés de multiples composants au moyen d'une série d'études empiriques, dans le but de mieux comprendre ces dépendances, de mettre en évidence les défis qu'elles introduisent et de proposer des approches automatisées afin d'aider les praticiens à gérer ces tâches de maintenance. Plus précisément, nous étudions d'abord la manière dont une modification apportée à un composant peut dépendre de changements effectués dans d'autres composants, phénomène également appelé dépendances inter-composants. Cette étude met en lumière la façon dont différents composants, souvent gérés par des équipes distinctes, dépendent les uns des autres, souligne les défis résultant de ces dépendances et dégage des implications pratiques à destination des praticiens et des chercheurs. Nous proposons ensuite une approche fondée sur l'apprentissage automatique et une approche basée sur l'IA agentique, afin d'aider les praticiens à identifier des changements candidats susceptibles d'être liés à un changement de code donné. Enfin, nous étudions les dépendances verticales de maintenance en examinant la gestion des mises à niveau des dépendances tierces en termes de stratégies de mise à niveau, de durée et de pratiques de configuration, afin de fournir aux développeurs et aux chercheurs des recommandations permettant d'améliorer la gestion de ces dépendances.

Nos résultats soulignent l'importance des dépendances de maintenance, tant horizontales que verticales, dans les systèmes logiciels modernes. En conséquence, le système de recommandation proposé peut aider les développeurs dans la maintenance de tels systèmes. Nous discutons également de plusieurs implications issues de nos résultats et présentons des pistes potentielles pour les travaux de recherche futurs.

Mots-clés: maintenance logicielle, modularité, apprentissage automatique, étude empirique, IA agentique

Supporting Practitioners in Managing Horizontal and Vertical Dependencies in Large-scale Software Systems

Ali ARABAT

ABSTRACT

Modern software systems are nowadays based on interconnected components. A software system can be composed of multiple interacting components, such as plugin-based systems, service-oriented architectures (SOA), and microservice applications. These interactions provide complementary features that together offer a service, and may require coordinated changes across components, which we refer to as **horizontal dependencies**. At the same time, a component in a software system can use a set of external dependencies that manifest as external libraries and that provide a set of utilities to implement a service. For example, instead of re-implementing filesystem utilities, developers can easily import a dependency that provides the same features. Changes in such external libraries may require corresponding adaptations in the client project, and we refer to these relationships as **vertical dependencies**.

We concretely define *horizontal dependencies* as dependencies between the components and services of the same software system, whereas *vertical dependencies* are defined as dependencies between a project and the third-party libraries specified in the project's manifest file (e.g., package.json). In this thesis, horizontal and vertical dependencies are treated as two complementary forms of maintenance dependency: one within the project boundary and the other across the project boundary.

While vertical and horizontal component interconnections offer significant benefits, they also introduce complex challenges that manifest in the maintenance burden of synchronizing across components and managing external dependencies. From a horizontal perspective, developers frequently need to coordinate across component boundaries – synchronizing changes with teams working on other components, collaborating on complementary code spread across multiple components to fix a bug or deliver a new feature, and ensuring that a change made in one component is properly validated by tests in other components. This cross-team coordination adds friction to daily development and, when overlooked, can result in broken builds, partial merges, and CI/CD pipeline failures. From a vertical perspective, third-party libraries frequently release updates containing new features, bug fixes, and security patches. Failing to integrate these updates promptly can render client projects unstable, introduce exploitable security flaws, and expose the client's source code to risks.

This thesis investigates the maintenance dependencies of large-scale, multi-component systems through a series of empirical studies, with the goal of better understanding these dependencies, uncovering the challenges they introduce, and proposing automated approaches to support practitioners in managing such maintenance tasks. In particular, we first study how a change in one component can depend on changes in others, also known as cross-component dependencies. This study sheds light on how different components, often owned by different teams, depend on each other, highlights the challenges that arise from these dependencies, and distills practical

implications for both practitioners and researchers. We then propose a machine-learning-based approach and an agentic-AI-based approach to help practitioners identify candidate changes that may be related to a given code change. Finally, we investigate vertical maintenance dependencies by examining how third-party dependency upgrades are managed in terms of upgrade strategies, time, and configuration practices to provide developers and researchers with guidance on helping practitioners better manage their dependencies.

Our findings highlight the importance of maintenance dependencies, both horizontal and vertical, in modern software systems. Accordingly, the proposed recommendation system can help developers in the maintenance of such systems. We also discuss several implications of our findings and outline potential directions for future research.

Keywords: software maintenance, modularity, machine learning, empirical study, agentic-AI

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Modern software systems are structured around multiple interconnected components. Typical examples include plugin-based architectures WordPress (2023), service-oriented architectures (SOA) Gabhart & Bhattacharya (2008), and microservice applications Richardson (2018). These components interact to provide complementary functionality; for example, one component may implement a CPU-consumption API that is invoked by another component. Such interactions can create maintenance relationships in which changes to one component may require coordinated changes to other components within the same software system. In this thesis, we refer to these relationships as **horizontal dependencies**. Software systems also frequently rely on external dependencies, that is, third-party libraries that provide specific functionality. For instance, a project may use a library to manage interactions with the filesystem. Changes in these external libraries, such as upgrades or API modifications, may require corresponding adaptations in the client project. We refer to these relationships as **vertical dependencies**. Consequently, horizontal and vertical dependencies represent two complementary forms of maintenance dependency: one within the project boundary and the other across the project boundary.

Such dependency types have many benefits. For example, Netflix, a popular streaming provider, has thousands of microservices, each serving a unique feature (e.g., billing management). This architectural approach provides managers with a clear, comprehensive overview of the project structure and allows practitioners to develop and maintain their respective components without disrupting other teams. Furthermore, these microservices often leverage third-party dependencies to avoid reinventing the wheel. Consequently, client projects can focus on core functionalities while relying on third-party libraries to provide stable updates, new features, and resolutions for bugs or security vulnerabilities.

Given these advantages, both vertical and horizontal dependencies can introduce significant challenges in practice. Regarding horizontal dependencies, each component is typically

developed and maintained by a dedicated team, which introduces the challenge of synchronizing changes across component boundaries. Developers must coordinate with other teams when their changes affect or depend on code in other components, collaborate on complementary changes spread across multiple components to fix a bug or implement a new feature, and ensure that a change made in one component is adequately tested in the context of other components. When such coordination is overlooked, it can result in broken builds, partial merges, CI/CD pipeline failures, and eventually production failures.

In turn, managing vertical dependencies presents its own set of difficulties. Client projects that incorporate a large number of external libraries must ensure these dependencies are kept up-to-date with upstream releases. Failure to perform these upgrades promptly can lead to unpatched bugs, backward incompatibilities, and the introduction of exploitable security vulnerabilities. Furthermore, projects may encounter "dependency hell", where conflicting version requirements, such as when multiple dependencies rely on different, incompatible versions of the same sub-dependency, can disrupt build processes and introduce runtime errors.

Building on the widely recognized benefits of component-based software architectures, particularly microservices, a substantial body of research has focused on supporting the migration of legacy monolithic systems into modular architectures such as the work of Gouigoux & Tamzalit (2017); Abdellatif *et al.* (2021); Trabelsi, Moha, Guéhéneuc & Geffard (2024a). Such architectures are designed to improve the maintainability aspect by decomposing legacy systems into independent components that can be developed, deployed, and maintained independently from other components/services.

However, a second line of research has challenged the assumption of complete component independence. In fact, prior studies Sampaio *et al.* (2017); Bogner, Fritzsch, Wagner & Zimmermann (2019, 2021) have shown that, even after migration to modern component-based architectures, practitioners continue to face significant challenges arising from

the coupling between different components. Despite these efforts, little is known about the maintenance overhead caused by the dependencies of different components. For example, the extent to which components within the same system co-evolve remains largely unexplored, as does the question of how practitioners can be effectively supported in identifying maintenance dependencies that span team boundaries.

Similarly, on the vertical dependencies, several studies Alfadel, Costa, Shihab & Mkhallalati (2021); Cogo & Hassan (2022); He, He, Zhang & Zhou (2023) have consistently reported that developers delay dependency upgrades despite the growing adoption of dependency management bots intended to streamline the upgrade process. However, little is known about what happens to the bot-suggested upgrades that are not accepted, what their underlying characteristics are, and how we can support and provide practitioners with insights into reducing upgrade delays and enabling them to get the full benefits of dependency management tools.

In this thesis, we study maintenance dependencies, that is, relationships in which a change to one software artifact may require coordinated changes to other artifacts. We distinguish two manifestations of such dependencies according to the project boundary. Horizontal dependencies are maintenance dependencies among components belonging to the same software system, where modifications to one component may propagate to other internal components. Vertical dependencies are maintenance dependencies between a project and external third-party libraries declared in its dependency manifest (e.g., package.json), where upgrading or modifying an external library may require corresponding adaptations in the client project. Thus, horizontal and vertical dependencies represent two complementary forms of change propagation: one within the project boundary and the other across the project boundary. To this end, the thesis addresses the following research objectives:

1. **Quantify horizontal maintenance dependencies** among components of a multi-component system and assess the existence of dependencies when a given component evolves. This

investigation aims to provide insights into how large-scale software systems evolve their component and the efforts required to handle them.

2. **Develop a recommendation system** to aid developers in better maintenance of dependencies among their components. Prior work highlights a lack of automated support for such activity, which remains a challenging and error-prone part of software development. Thus, we would like to design a machine learning-based approach to help developers find most likely related changes within the same or different components given a component change, and a rigorous comparison to an agentic-AI solution.
3. **Quantify vertical maintenance dependencies**, with a focus on how their upgrades are performed by developers. As a software system has dozens of third-party dependencies, not performing timely upgrades can pose serious problems on the system and the maintenance overhead might be burdensome. Therefore, this research objective investigates different strategies of performing the upgrades that are suggested by dependency management bots, and provides implications to practitioners to enhance the maintenance and upgrade of their dependencies.

0.1 Thesis Hypothesis

While component reuse is a common practice in modern software development, it can substantially increase the complexity of maintaining dependencies, many of which may be unexpected and difficult to manage. To this end, it is crucial to better understand the nature and extent of dependencies within large-scale software systems, as well as the common challenges developers face when evolving their components. Building on this motivation, we formulate the following research hypotheses:

To substantiate our research hypotheses, we structure our work into several complementary investigations. We begin by **(1)** conducting an empirical study to characterize the maintenance

> We hypothesize that modern software systems contain a non-trivial amount of maintenance dependencies – both horizontal, where developers must coordinate co-evolving changes across component boundaries, and vertical, where timely upgrades of third-party dependencies are frequently hindered by the associated maintenance overhead – and that such cross-component dependencies can be effectively predicted through a machine learning-based approach, with agentic-AI as a complementary solution.

dependencies that exist among components within a multi-component system. Accordingly, **(2)** we develop two approaches, one based on machine-learning and the second on agentic-AI, to help developers find related changes in advance when they submit their code changes for review. Finally, **(3)** we look at the maintenance of vertical dependencies, focusing on how third-party dependencies are upgraded, the required time, and the bot configuration practices to enhance such upgrades.

0.2 Thesis Contributions

This section presents the main contributions related to understanding and evolving multi-component software. We further discuss the key contributions of each research objective.

0.2.1 Understanding and Quantifying The Horizontal Maintenance Dependencies Between Components

To quantify horizontal maintenance dependencies, we investigate the interdependencies among software changes and components using code review data hosted on the Gerrit platform (i.e., a code review platform used by OpenStack). To this end, we mine the metadata of code changes and identify links between changes through dependency tags, such as `Depends-On`.

Our results reveal that a non-trivial amount of changes depend on each other that belong to different components. These interdependencies increase over time until a major architectural refactoring is undertaken across OpenStack projects.

Furthermore, our qualitative investigation results in a taxonomy comprising 10 high-level categories of cross-component dependencies and identifies 12 distinct abandonment-related reasons associated with dependent changes.

Based on our results, we derive a set of implications for both practitioners and researchers aimed at improving the understanding, management, and mitigation of maintenance overhead arising from cross-component dependencies.

0.2.2 Developing a Recommendation System for Dependent Changes

Building upon the preceding contributions, we introduce two variant approaches designed to assist developers in identifying dependencies within large-scale software systems: one using machine learning and another leveraging agentic AI.

Our evaluation demonstrates a performance trade-off between the two approaches: the ML-based model achieves a good recall, while the latter exhibits higher precision.

Furthermore, the ML approach demonstrates robust predictive performance across both within-component and cross-component settings, although a slightly improved performance is observed in the cross-component setting.

0.2.3 Understanding and Quantifying The Vertical Maintenance Dependencies

This research objective complements our series of empirical investigations to quantify vertical maintenance dependencies. Specifically, we examine multiple aspects of dependency upgrades

suggested by Dependabot and analyze how developers manage these upgrades in practice. To this end, we leverage Dependabot-generated PRs collected from a curated set of software projects.

Our findings indicate that the studied dependency upgrades exhibit a median upgrade time of 6.13 hours, which is more than twice the median time of directly merged PRs (the way it was studied by prior work). Further, the upgrades performed through superseding changes or regular commits experience significantly longer delays than directly merged PRs, where the median upgrade time ranges between 21.9 and 122 days.

At the project level, the situation appears even more pronounced. Specifically, we observe that 84.70% of the analyzed projects complete dependency upgrades within a median time of one week or longer.

Finally, our quantitative analysis of upgrades performed through regular commits reveals that more than 50% of the examined cases involve developers regrouping multiple dependency updates into a single upgrade effort rather than addressing the corresponding PRs individually.

0.3 Thesis Organization

This section outlines the main chapters of this thesis, which are structured as follows:

1. **Chapter 1** presents and critically reviews existing research related to maintenance dependencies within large-scale systems, the challenges developers encounter, existing solutions aimed at improving the maintenance of post-migration software systems, and the management of vertical dependencies.
2. **Chapter 2** addresses the first research objective, focusing on understanding and quantifying horizontal maintenance dependencies within large-scale software systems.
3. **Chapter 3** proposes a recommendation system to assist developers in identifying related changes for a given code change using various machine learning algorithms. The chapter

also presents a rigorous comparison between traditional machine learning techniques and agentic-AI for accomplishing this task.

4. **Chapter 4** complements Chapter 3 by evaluating the performance of our machine learning models in predicting dependencies in the context of a multi-component system.
5. **Chapter 5** complements the previous study presented in Chapter 2 by investigating the management of vertical dependencies through third-party upgrades suggested by dependency management bots.
6. **Chapter 5.11.2.2.4** discusses potential future research directions and concludes the thesis.

0.4 Related Publications

The research conducted for this thesis resulted in the following publications:

- **Ali Arabat**, Mohammed Sayagh, "An empirical study on cross-component dependent changes: A case study on OpenStack," *Empirical Software Engineering*, 2024.
- **Ali Arabat**, Mohammed Sayagh, Jameleddine Hassine, "A recommendation system for predicting dependencies among software changes: Insights from an empirical study on OpenStack," *Empirical Software Engineering*, 2026.
- **Ali Arabat**, Mohammed Sayagh, Sarah Nadi, Artur Andrzejak, "On the time to complete Dependabot-suggested dependency upgrades," submitted to *Empirical Software Engineering*, 2026.

The following publications were performed during the course of this PhD research but are not included as part of the thesis contributions:

- Keheliya Gallaba, **Ali Arabat**, Dayi Lin, Mohammed Sayagh, Ahmed E. Hassan, "Towards Conversational Development Environments: Using Theory-of-Mind and Multi-Agent Architectures for Requirements Refinement," *arXiv*, 2025.

- **Ali Arabat**, Mohammed Sayagh, "Toward Instructions-as-Code: Understanding the Impact of Instruction Files on Agentic Pull Requests," *International Conference on Mining Software Repositories*, 2026.
- Mahmoud Abujadallah, **Ali Arabat**, Mohammed Sayagh, "Understanding the Rejection of Fixes Generated by Agentic Pull Requests - Insights from the AIDev Dataset," *International Conference on Mining Software Repositories*, 2026.

CHAPTER 1

LITERATURE REVIEW

Modern software systems are often composed of several loosely coupled components, each designed, developed, and maintained independently. These components are typically managed by distinct development teams, each following its own release strategy, development workflow, and technology stack. This architectural style offers several advantages over traditional monolithic systems. For example, teams can deliver new features continuously without disrupting other components. Additionally, time-to-market is improved, as teams operate with independent and well-established CI/CD pipelines. The components also maintain clear organizational and technical boundaries, which helps reduce coordination overhead among teams and minimizes technical dependencies when deploying features to production.

Microservices are among the most widely adopted software architectures in industrial settings. For example, Amazon (2025) offers a comprehensive platform to support the development and deployment of microservice-based applications at scale. Similarly, Uber (2018) decomposed its monolithic system into multiple small, loosely coupled, and autonomous services. Common examples of these extracted microservices include billing, payment, and trip management. In addition, Netflix (2025), a leading video streaming company, migrated its video processing pipeline from a monolithic architecture to a microservices-based one. This shift addressed several issues present in its legacy system, such as tight coupling, inflexible monolithic structure, and long release cycles.

Similar to horizontal dependencies, software projects are also characterized by the use of third-party dependencies. For instance, a JavaScript project typically defines a list of dependencies in a dedicated manifest file, i.e., package.json. These dependencies are required for the application to be built and deployed. To streamline the process of upgrading dependencies, several dependency management bots have been developed, such as Dependabot (2025a), Renovate (2025), and Snyk (2025). These dependency management bots are nowadays a core element in most software projects.

Building on these advantages, existing studies Gouigoux & Tamzalit (2017); Abdellatif *et al.* (2021); Trabelsi *et al.* (2024a) have largely focused on supporting the migration from legacy systems, such as monoliths, to modern architectures like microservices. This architectural shift offers significant benefits; for instance, because each component is developed and maintained independently, teams can ship features without disrupting other services. However, a second line of research Sampaio *et al.* (2017); Bogner *et al.* (2019, 2021) has highlighted that achieving complete component independence remains difficult, even after migration. These studies indicate that practitioners continue to face substantial challenges related to inter-component coupling and dependency management, specifically regarding the coordination of changes across components and the management of reliance on different component versions. Furthermore, regarding vertical dependencies, several studies Alfadel *et al.* (2021); Cogo & Hassan (2022); He *et al.* (2023) have consistently reported that developers often delay dependency upgrades, despite the growing adoption of automated tools designed to streamline the process.

In this chapter, we review related work encompassing four key areas: **(1)** the migration of legacy systems to microservices, **(2)** the underlying challenges encountered after migration, **(3)** approaches developed to support the maintenance of such systems, and **(4)** the management of third-party dependencies. Finally, **(5)** we critically analyze existing literature to discuss its major limitations.

1.1 Migration of Legacy Systems to Microservices

Several research efforts have been devoted to proposing approaches that help practitioners decompose monolithic applications into microservices in order to fully leverage the benefits of this architecture Fan & Ma (2017); Gouigoux & Tamzalit (2017); Mazlami, Cito & Leitner (2017); De Lauretis (2019); Colanzi *et al.* (2021); Mparmpoutis & Kakarontzas (2023); Nitin, Asthana, Ray & Krishna (2023); Chen, Li, Tyszberowicz, Liu & Liu (2024); Abdellatif *et al.* (2018, 2020); Abdellatif *et al.* (2021); Trabelsi *et al.* (2022); Trabelsi *et al.* (2024a); Trabelsi, Popa, Pereyrol, Beaulieu & Moha (2024b); Quattrocchi, Cocco, Staffa, Margara & Cugola (2024); Ding, Xu, Feng & Jiang (2024).

Fan & Ma (2017) proposed a migration framework for transitioning existing systems to a microservice architecture. The approach takes into account various tools and steps involved in the software development lifecycle (SDLC), including those related to system design and implementation. To demonstrate its effectiveness, the authors applied the framework to decompose a mobile application called EasyLearn into microservices.

Gouigoux & Tamzalit (2017) presented a set of lessons learned from migrating a software system of a French company to a Web-Oriented Architecture. The findings revealed significant service reuse over time (a core principle of microservice architectures). For instance, the Business Capability Map was reused by multiple services, and 70 migrated services were used in at least two different areas. Another benefit observed was the flexible replacement of services, where a service could be easily and quickly substituted with a newer version. The migration resulted in performance gains, driven by effective service reuse and flexible replacement.

Mazlami *et al.* (2017) proposed a formal model for identifying microservice candidates from an existing monolithic application. The model consists of three main stages—monolith, graph, and microservice, with two transformation steps between them. In the first stage, the monolith is represented as a graph, and in the second, microservice candidates are extracted from this graph-based structure. The authors validated their approach using 21 open-source projects written in three different languages: Python, Java, and Ruby. The evaluation results demonstrated the model's effectiveness in the migration process, showing improved scalability concerning revision history size and reduced team size.

To leverage the scalability and maintainability benefits of microservice architecture, De Lauretis (2019) introduced a five-stage migration strategy tailored for newly developed systems, beginning with functional analysis and ending with microservice creation.

Colanzi *et al.* (2021) empirically studied legacy system modernization practices involving microservices in industry by surveying software professionals from 56 different companies. The findings indicate that 63.6% of the surveyed companies migrated their existing code bases to a

microservice architecture. Key drivers for this decision included improved software maintenance, scalability, ease of deployment, and greater technological flexibility.

Unlike the aforementioned studies, which primarily focused on formal methods and source code as the basis for microservice extraction, Mparmpoutis & Kakarontzas (2023) examined research efforts that leverage the database as a primary artifact for transitioning from monolithic systems to microservices. Their study aimed to understand the degree of automation, the types of artifacts used, and how these artifacts are leveraged in the migration process. Interestingly, the results revealed a lack of fully automated approaches; most studies were found to be either semi-automated, partially automated, or lacked tool support altogether. Furthermore, schema and data analysis from the source code were typically used to identify potential microservice candidates.

Nitin *et al.* (2023) introduced a novel approach called CARGO, which operates on JavaEE applications to partition a monolithic application into microservices. The method leverages various types of internal relationships, including database transactions, call-return paths, and data-flow edges. In an evaluation using five JavaEE monolith-based applications, CARGO was shown to reduce latency and increase the throughput of the resulting microservices by an average of 11% and 120%, respectively.

Similarly, Chen *et al.* (2024) proposed an approach named Mono2MS, which partitions a monolithic application into a microservice architecture. The technique consists of two main components: MFAC and MSPC. The MFAC component performs semantic, static, and dynamic analyses on the monolithic application and generates corresponding views. These views are then fused and passed to the MSPC component. MSPC performs two key operations: it first learns feature embeddings from the fused views and then applies clustering on those embeddings to generate microservice partitions. The results demonstrate the effectiveness of Mono2MS over existing techniques, such as Mono2Micro, based on experiments conducted on five open-source projects of varying sizes.

Abdellatif *et al.* (2018) surveyed with 45 practitioners to better understand the state of practice in legacy system modernization, including modernization types, migration decisions, and target architectures. The findings revealed that most of the examined legacy systems were written in COBOL, Java, and CICS. These systems are typically modernized using various types of input data, such as source code, business process models, and database schemas. However, human expertise remains essential for guiding the migration process. Microservices and RESTful services are commonly chosen as target architectures to increase service reuse, determine appropriate service granularity, and achieve loose coupling among services.

Building on their earlier work investigating legacy system modernization practices Abdellatif *et al.* (2018), Abdellatif *et al.* (2020) introduced a bottom-up, type-aware tool called ServiceMiner to support the modernization of legacy systems to Service-Oriented Architectures (SOA). The proposed approach was evaluated on a large-scale open-source ERP system. The results demonstrated strong performance in identifying key service candidates within the target application, achieving a precision of 77.9%, recall of 66.4%, and F1-score of 71.7%.

Later, Abdellatif *et al.* (2021) conducted an SLR to investigate the state-of-the-art in service identification approaches (SIAs), covering 41 studies. The review examined their inputs, processes, outputs, and usability. The findings indicate that the surveyed SIAs were not evaluated or validated on enterprise-grade software, limiting their practical applicability in industrial settings. Furthermore, no tool support was available to enhance reproducibility or validate the reported results. Additionally, the authors found that most of the reviewed SIAs lack automation and do not provide quality metrics for the identified service candidates.

Trabelsi *et al.* (2022) proposed an approach called MicroMiner to support the migration of legacy systems to microservices. The approach leverages service type classification and machine learning algorithms, and relies on both static and semantic analysis of the source code. In an evaluation on four legacy systems, MicroMiner demonstrated its ability to identify architecturally relevant microservice candidates, achieving a precision of 68.15% and a recall of 77%.

Building upon their prior work, Trabelsi *et al.* (2024a) developed a fully automated approach named Magnet to facilitate the process of microservice identification. While Magnet shares similarities with MicroMiner Trabelsi *et al.* (2022), it leverages the advanced capabilities of Graph Neural Networks (GNNs). Unlike MicroMiner, Magnet is fully automated. In an empirical evaluation on four representative monolithic systems, including Compiere and JForum 3, Magnet achieved a precision of 56% and a recall of 68%.

Trabelsi *et al.* (2024b) developed a novel approach called MicroMatic, which automatically supports the migration of IoT applications to microservices. Similar to Trabelsi *et al.* (2022), MicroMatic leverages two types of artifacts: static analysis of dependencies and semantic analysis of the source code. It also combines machine learning algorithms with service type classification. Evaluation results indicate that MicroMatic can automatically identify architecturally relevant microservices with a precision of 62.5% and a recall of 45.5%.

Quattrocchi *et al.* (2024) developed a semi-automated tool called Cromlech to support the decomposition of microservices. Cromlech operates in three main steps: (1) it scans the system's functionalities and their associated data entities, (2) it formulates the migration process as an optimization problem, and (3) it suggests alternatives for mapping functionalities and data in the target system. Evaluation results indicate that Cromlech can achieve promising decomposition outcomes, as demonstrated through validation on an industrial system.

While most existing studies overlook the resource usage of migrated systems, Ding *et al.* (2024) proposed a novel microservice extraction method called MECE, which considers various influencing factors, such as CPU and memory usage, to evaluate the quality of the extracted services. Post-migration evaluations showed that the microservices produced by MECE achieved performance improvements of up to 46%.

A more recent effort explored the use of Large Language Models (LLMs) for microservice decomposition Alsayed, Dam & Nguyen (2024). The authors employed static analysis techniques, similar to those in Trabelsi *et al.* (2024b), to extract model classes from monolithic systems and construct graphs representing relationships between these classes. Contextual relationship

information was then obtained using biased random walks in combination with LLMs. When compared against two state-of-the-art microservice decomposition techniques, MicroDec demonstrated superior performance across five evaluation metrics.

Although prior studies have proposed various approaches for migrating monolithic systems to microservices, our work differs from this line of research. Instead, we focus on examining the maintenance overhead of post-migration software systems and understanding how they evolve in practice.

1.2 Challenges and Co-evolution of Post-migration Software Systems

Breaking down a single-component system (i.e., a monolith) into a multi-component system (i.e., microservices) is widely regarded as a challenging process due to the complexity of the system being decomposed and the tight coupling between classes and modules, even the deprecation of technologies Fritzsch, Bogner, Wagner & Zimmermann (2019). Despite advances in tooling, human intervention remains essential in this process. Several researchers have identified key microservice evolution challenges Sampaio *et al.* (2017); Bogner *et al.* (2019, 2021); Fritzsch *et al.* (2019); Taibi, Lenarduzzi & Pahl (2020); Tighilt *et al.* (2020).

Sampaio *et al.* (2017) reported that IBM developers encounter significant challenges when evolving their microservices. Specifically, they rely on time-consuming, error-prone manual investigations to reconcile version inconsistencies across services. To address this, the authors propose an evolution model that supports the systematic evolution of microservices.

Bogner *et al.* (2019) conducted 17 semi-structured interviews with microservice practitioners from 10 different companies. The authors enumerated a set of challenges while migrating a system to microservices, among which service splitting and service granularity were reported as the most challenging. Precisely, services with high cohesion and low coupling are difficult to achieve.

Later, through an extensive grey literature review and interviews with industry professionals, Bogner *et al.* (2021) identified service integration and service decomposition (i.e., service cutting) as the most challenging aspects of the microservice migration process. Furthermore, the authors highlighted a lack of automated tools to support the maintenance and evolution of migrated systems, particularly for monitoring service granularity and evaluating service dependencies.

Fritzsch *et al.* (2019) conducted an in-depth qualitative study with 16 software professionals to better understand the intentions, strategies, and challenges involved in microservice migration. The authors reported that many interviewees preferred to re-implement existing systems due to the lack of systematic migration approaches. Consistent with the findings of Bogner *et al.* (2019, 2021), the study also identified service decomposition (i.e., service cutting) as a key technical challenge practitioners face during the migration process. Additionally, organizational challenges, particularly in large teams, were frequently reported.

Taibi *et al.* (2020) conducted an empirical study on microservice anti-patterns by interviewing 13 participants from the International DevOps Conference in Helsinki and 14 from the O'Reilly Software Architecture Conference in London. All participants had at least two years of experience with microservice-based development. The authors identified 20 microservice anti-patterns, classified into two categories: organizational and technical. The study highlighted that Hardcoded Endpoints, Wrong Cuts, and Cyclic Dependencies are among the most critical issues encountered during the re-architecting of traditional systems into microservice architectures.

Tighilt *et al.* (2020) carried out a systematic literature review (SLR) on 27 research papers and examined 67 microservice-based open-source projects. The authors identified a catalog of 16 microservice anti-patterns, which are similar to those proposed by Taibi *et al.* (2020).

Through an empirical study on a plugin-based system (WordPress), Li, Ma & Lu (2020) found that incompatibility issues frequently arise during the co-evolution of WordPress and its plugins. Notably, approximately 32% of the studied incompatibility issues occurred between WordPress and its plugins, among other sources.

Similarly, Assunção, Krüger, Mosser & Selaoui (2023) studied the evolution of 11 open-source microservice-based systems. The authors observed that microservices often evolve around technical aspects, such as library version updates, rather than changes to business logic. Interestingly, their findings reveal that these systems do not evolve in isolation, even though microservices are intended to evolve independently from one another.

Through a multi-vocal literature review of 26 relevant papers on microservice evolution, Pinto-Agüero & Noel (2025) found that dependencies are regarded as one of the most critical factors hindering the evolution of microservices. The findings also revealed that having bounded contexts and finer service granularity can facilitate and accelerate this evolution.

Our work complements the aforementioned studies that examined the challenges and co-evolution of different components within microservice-based systems. Specifically, we aim to gain a better understanding of how components co-evolve in practice. If such co-evolution exists, we seek to identify the underlying maintenance costs associated with managing these dependencies.

1.3 Improving the Maintenance of Multi-component systems

This section reviews the existing related work on (i) predicting dependencies in a software system, (ii) metrics to improve the maintenance of microservices, and finally (iii) linkage detection in code reviews.

1.3.1 Predicting Dependencies in a Software System

Diaz-Pace, Tommasel & Godoy (2018) proposed an approach to predict dependencies among software modules by applying link prediction techniques to analyze the dynamics of module structures as dependency graphs.

Oliva & Gerosa (2012) proposed a method to identify logical dependencies, which are implicit relationships among software artifacts.

Xia, Lo, McIntosh, Shihab & Hassan (2015) introduce CroBuild, a cross-project build co-change prediction approach. The proposed approach learns different classifiers using data from a project and evaluates its performance on another project, and achieves an F1-score of up to 0.408.

Aryani, Perin, Lungu, Mahmood & Nierstrasz (2011) proposed an approach to predict software dependencies by utilizing coupling in domain-level information. Using such information, their approach can predict up to 68% and 77% of source code and database dependencies, respectively.

Yang *et al.* (2021) introduce AID, an approach that evaluates dependency intensity, which is defined as *"how much the status of the callee service influences the caller service"*.

Zagane & Alenezi (2024) leverage a hybrid approach, consisting of traditional SE techniques and deep learning models, to predict co-changes within software systems.

1.3.2 Microservices Maintenance Metrics

Blincoe, Harrison, Kaur & Damian (2019) proposed a novel approach called Reference Coupling, which automatically identifies technical dependencies using a set of heuristics. Their method is capable of uncovering dependencies that are difficult to detect with traditional static analysis techniques.

Zhong, Zhang, Li, Huang & Feitosa (2023) proposed a microservice coupling metric that quantifies the degree of interdependence between microservices. Their metric is based on the number of interfaces (e.g., API-providing classes) in one microservice that are invoked by classes in another.

Likewise, Li, d'Aragona & Taibi (2023) introduced a metric called organizational coupling, which captures collaboration patterns among developers from different microservice teams. This coupling is derived from ownership data and patterns of cross-service collaboration.

1.4 The Management of Third-party Dependencies

Since our work focuses on third-party dependencies, we review (i) studies on dependency upgrades and (ii) studies on Dependabot PRs.

1.4.1 Studies on dependency upgrades

Several studies have examined how developers manage and upgrade their dependencies Kula, German, Ouni, Ishio & Inoue (2018a); Cogo, Oliva & Hassan (2021, 2022); Venturini, Cogo, Polato, Gerosa & Wiese (2023); Weeraddana, Alfadel & McIntosh (2024); Bi, Liang, Zhang, Chen & Wang (2025); He, Vasilescu & Kästner (2025).

Kula *et al.* (2018a) empirically investigated library migration of 2,700 dependencies encompassing 4,600 JavaScript repositories and reported that 81.5% of them still rely on outdated dependencies.

Cogo *et al.* (2021) explored dependency downgrades in the npm ecosystem, revealing that 49% of the studied cases involved replacing acceptable version ranges with older versions. Later, Cogo *et al.* (2022) examined package and release deprecations in npm, finding that 3.7% of packages had at least one deprecated release (i.e., a release with obsolete features), and 66% of those had all of their releases deprecated.

Venturini *et al.* (2023) examined how dependent packages are affected by breaking changes in the npm ecosystem. They found that nearly 12% of the studied dependent packages were impacted when performing non-major upgrades.

Weeraddana *et al.* (2024) studied CI build waste caused by updates to unused dependencies across more than 20K commits from 1,387 JavaScript projects. They observed that updates to unused dependencies account for 55.88% of the CI build time.

Bi *et al.* (2025) investigated the prevalence of outdated dependencies in the npm ecosystem and reported that outdated packages are widespread across dependency chains, potentially affecting client projects.

He *et al.* (2025) analyzed the impact of pinning dependency versions on security and found, counterintuitively, that this practice can increase a project's exposure to vulnerable dependency updates.

1.4.2 Studies on Dependabot PRs.

Several researchers have empirically examined how developers use and respond to dependency update bots Alfadel *et al.* (2021); He *et al.* (2023); Rebatchi, Bissyandé & Moha (2024).

Alfadel *et al.* (2021) conducted the first empirical study on Dependabot (then known as *dependabot-preview*, an older and now unmaintained version) to investigate how developers handle security-related Dependabot PRs. They found that 65.42% of security PRs were merged, while a substantial proportion (50.8%) of the closed security PRs were superseded.

Similarly, He *et al.* (2023) studied how developers respond to Dependabot PRs and found that security-related PRs take longer for developers to merge, close, or respond (first human intervention), yet have a higher merge rate compared to regular PRs.

Along the same line, Rebatchi *et al.* (2024) conducted a large-scale empirical study on security Dependabot PRs and found that they are often merged within one day, while most closed PRs were also due to superseding, consistent with the findings of Alfadel *et al.* (2021).

Other studies have focused on how vulnerabilities are resolved through Dependabot. Mohayeji, Agaronian, Constantinou, Zannone & Serebrenik (2023, 2025) found that projects using CI/CD tools are more likely to accept security updates automatically, and that when these PRs are not merged, developers would manually resolve the vulnerabilities.

Cogo & Hassan (2022) investigated how developers customize Dependabot configurations and observed that developers frequently change options related to notification reduction.

In line with this, Kula (2025) explored different pathways in which AI can help developers mitigate alert fatigue caused by the large volume of Dependabot notifications.

1.5 Critical Analysis and Limitations of Literature

While existing research on the management of horizontal dependencies has largely focused on the migration from traditional to modern architectures, the underlying challenges of these systems, and proposed maintenance metrics to reduce component coupling, there are several limitations to point out. First, prior studies lack empirical investigations of these interdependencies within real-world software projects. In fact, this line of research has not deeply focused on the coupling of different components and how they co-evolve, given the massive volume of changes submitted daily. Second, there is a lack of automated tools to support practitioners in managing these dependencies and detecting them in advance.

Another potential limitation of the reviewed studies is that they do not provide a systematic method for identifying dependencies among the components of complex software systems, such as OpenStack. Instead, these studies mostly discuss insights from industry practitioners without providing sufficient empirical evidence to prove the existence and relevance of such dependencies. While certain coupling metrics have been proposed, they are often limited to microservice-based systems, which limits their applicability to other types of software architectures where components may not interact via microservice-based links.

Furthermore, existing studies suffer from a limitation regarding the selection of subject systems. For instance, most of these studies focused on unmaintained, uncommon, or overly simplistic software projects. Consequently, these studies face significant threats to external validity, as their findings may not generalize to industry-scale projects, which are inherently more complex and architecturally different.

Unlike prior work, this thesis systematically demonstrates how components depend on each other within large-scale software systems, where components are developed and maintained by different teams. Hence, we empirically investigate the prevalence, relevance, and characteristics of these interdependencies within a popular open-source software project. This thesis also introduces two approaches: one using machine learning and a second leveraging agentic AI to assist developers in identifying these interdependencies early in the development phase.

Although research on the management of vertical dependencies has examined third-party dependency upgrades suggested by bots, these studies also have certain limitations. Notably, little is known about what happens to the bot-suggested upgrades that are not accepted, what their underlying characteristics are, to provide practitioners with insights into reducing upgrade delays and enabling them to get the full benefits of dependency management tools. Therefore, our work deepens the understanding of upgrade time from different perspectives, including the type of dependency, the bot configuration, and the content of the upgrade.

CHAPTER 2

AN EMPIRICAL STUDY ON CROSS-COMPONENT DEPENDENT CHANGES: A CASE STUDY ON THE COMPONENTS OF OPENSTACK

Ali Arabat[1] , Mohammed Sayagh[1]

[1] Department of Software Engineering and IT, École de Technologie Supérieure, 1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3

Article published in the journal « Empirical Software Engineering » July 2024

2.1 Abstract

Modern software systems are composed of several loosely coupled components. Typical examples of such systems are plugin-based systems, microservices, and modular software systems. Such types of software systems have several advantages motivating a large body of research to propose approaches for the migration from monolithic software systems to modular architecture (mainly microservices). However, a few prior works investigated how to assist practitioners post-migration. In fact, these studies reported that having independent components is difficult to achieve, leading to several evolution challenges that are still manually handled. In this paper, we conduct an empirical study on OpenStack and its 1,310 projects (aka., components) to better understand how the changes to a given component depend on changes of other components (aka., cross-component changes) so managers can better plan for their changes in a cross-component project, and researchers can design better solutions to help practitioners in such a co-evolution and the maintenance of multi-component software systems. We observe that the concept of ownership exists in the context of OpenStack, as different teams do not share the responsibility over the studied components of OpenStack. Despite that, dependencies across different components are not exceptional but exist in all releases. In fact, we observe that 52,069 OpenStack changes (almost 10% of all the changes) depend on changes in other components. Such a number of cross-component changes continuously increased over different years and releases, up to a certain release in which OpenStack decided to make a major refactoring of its project by archiving over 500 projects. We also found that a good

percentage of cross-component changes (20.85%) end up being abandoned, leading to wasteful synchronization efforts between different teams. These dependent changes occur for different reasons that we qualitatively identified, among which configuration-related (34.64%) changes are the most common, while developers create cross-component changes for testing purposes then abandon such changes as the most prevalent category (38.45%). These cross-project changes lead to collaboration between different teams to synchronize their changes since 24.55% of the pairs of two cross-component changes are made by different developers, while the second change is reviewed by the developer of the first change of the pair (71.63%). Even when a developer makes both changes, that developer ends up working on a project that she/he is less familiar with. Our results shed light on how different components end up being dependent on each other in terms of their maintenance, which can help managers better plan their changes and guide researchers in proposing appropriate approaches for assisting in the maintenance of multi-component systems.

2.2 Introduction

Modern software systems are composed of multiple components (aka, multi-component systems). Typical examples of these systems are plugin-based systems (e.g., WordPress Sayagh, Kerzazi & Adams (2017)), Service-Oriented-Architecture Gabhart & Bhattacharya (2008), and Microservices Richardson (2018). Such software systems are organized as a collection of small and loosely-coupled components, each of which is developed, maintained, and deployed independently from each other Richardson (2018). In fact, in such systems, different teams are expected to work independently on their respective components without synchronizing with other teams working on other components Richardson (2018). Such loosely coupled types of systems lead to a key benefit which is related to the time-to-market (i.e., rapid and safe delivery of the software product Richardson (2018)). In fact, when one component undergoes changes based on the requirements needs, only that component is expected to be deployed into production without impacting the processes of the remaining running services. For instance,

all high-tech companies such as Uber, Amazon, and Netflix implement such a type of software architecture Fowler (2016); Services (2016); Vučković (2020).

Another well-known example of multi-component software systems is OpenStack. It is a popular cloud computing platform that offers a wide range of components to set up a cloud-based infrastructure. Among these components is "Nova", which is a computing service used to create and manage virtual machines and bare-metal servers. "Swift" is another component for storing and retrieving massive data at scale. The diversity and flexibility of these components make OpenStack one of the most popular and powerful cloud computing solutions in the market. As a matter of fact, it is one of the three biggest open-source projects Ubuntu (2022) where it is deployed on over 40M nodes around the world, supported by over 560 leading software companies such as Redhat, and accounting for more than 9,000 community contributors OpenStack (2023).

To benefit from the advantages of multi-component architectures, microservices in particular, several studies Fan & Ma (2017); Furda, Fidge, Zimmermann, Kelly & Barros (2018); De Lauretis (2019) tried to help practitioners migrate from monolithic to microservice systems. For example, Fan & Ma (2017) proposed a process to migrate monolith-based mobile applications into a microservice-based architecture. Furda *et al.* (2018) have discussed different techniques to migrate legacy enterprise systems to microservice wrt., typically encountered challenges (i.e., Multi-tenancy). Similarly, De Lauretis (2019) developed a strategy that supports the migration from monolith to microservice architecture.

With these progresses in the literature, having independent components is difficult to achieve, leading to a tedious co-evolution of components, which is still a manual and non-straightforward activity Sampaio *et al.* (2017); Bogner *et al.* (2019); Lin, Sayagh & Hassan (2023). Sampaio *et al.* (2017) found that IBM has several evolution challenges, among which are the compatibility and consistency between microservice versions. For instance, they found that developers manually identify how their modifications impact other components/services and engage with these other services' developers to identify potential issues. Through a systematic grey literature review and interviews with developers from 10 different companies, a prior work Bogner *et al.* (2019)

found that having a low coupling between services (aka., components) is difficult to achieve. Similarly, we found in a prior work Lin *et al.* (2023) that incompatibility issues that occur between the plugins of a plugin-based system are common and harmful. These incompatibility issues are among the most dominant reasons why WordPress users do not upgrade the version of their plugins [1]. While prior studies point out that maintaining multi-component systems is challenging, no prior work exists on understanding the maintenance of such systems.

As a first step toward assisting practitioners in the evolution of their multi-component systems, we first aim to conduct an empirical study to understand the maintenance dependency (co-evolution) between different components. In particular, we wish to study the prevalence, characteristics, and impact of the co-evolution of different components. The impact is in terms of engaged practitioners since such a co-evolution engages different teams and in terms of wasted resources manifested as abandoned changes of the co-evolution between components. Our empirical results will motivate and guide future studies to develop approaches for assisting practitioners in the evolution of multi-component systems. Similarly to prior studies on the co-evolution between different artifacts Sampaio *et al.* (2017); Bogner *et al.* (2019); Lin *et al.* (2023); Assunção *et al.* (2023), our results will also help practitioners better estimate the required efforts for maintaining their multi-component systems.

To better understand the co-dependency/evolution of different components of OpenStack and shed light on the *prevalence*, *characteristics*, and *impact* of the co-evolution among different components, we conduct a quantitative and qualitative study on the changes of OpenStack and its 1,310 projects. In this paper, we focus on OpenStack since it has many components (over 1,310) and OpenStack respects the ownership concept as no component is shared among different teams of OpenStack and even the projects that are maintained by the same team are maintained by different developers, as we found from a preliminary study (PQ). Thus, OpenStack is an ideal case study with a large number of components, each of which is maintained by different teams. Furthermore, a large body of research studied OpenStack in different domains, such as bug assignment Tong, Ying, Xiaoyong, Hongyan & Zhonghai (2015),

[1] https://wordpress.org/news/2009/10/plugin-compatibility-beta/

micro-collaboration Foundjem, Constantinou, Mens & Adams (2022a), and infrastructure-as-code Bessghaier, Sayagh, Ouni & Mkaouer (2023). In particular, we address the following research questions:

RQ1. How prevalent is the co-evolution of different components? We observe that OpenStack has 52,069 cross-component changes, which is 9.7% of all the changes and 38.45% of changes with a dependency (either within/across components). Moreover, a change to one component may ripple through a large number of components up to 224.

RQ2. How does the prevalence of cross-component changes evolve?

All OpenStack releases involve cross-component changes and such changes was increasing up to the "rocky" release, right after three releases ("stein", "train", and "ussuri"), they archived a large number (501) of projects. After which the number of cross-component (i.e., 2,151 and 863 were the number of cross-component changes in the "victoria" and "zed" releases, respectively) changes exhibited a gradual decrease. Yet, cross-component changes are still common in the recent releases of OpenStack.

RQ3. How often do cross-component changes end up being abandoned? A good percentage of cross-component changes end up being abandoned. In fact, 20.85% of the studied cross-component changes end up being abandoned. In addition, our findings indicate that 25.77% of the investigated cross-service dependencies have also been abandoned.

RQ4. What are the characteristics of cross-component changes? Our qualitative study has shown that the examined cross-component dependencies are related to different categories (10), while developers abandon such pairs of dependent changes for a variety of reasons (12). For instance, the *Configuration* category is the most dominant cause accounting for 34.64% of the qualitatively studied cross-component changes. Moreover, *Test* is the most widespread category for which developers abandon cross-component changes covering over a third (38.45%) of the studied changes.

RQ5. Who is responsible for addressing cross-component changes? While 24.55% of the examined cross-component dependencies were carried out by different developers, a large percentage (71.63%) of such developers often develop one change and participate in the review of the other change of a pair of cross-component changes. Moreover, just 8.65% of cross-component dependent changes were not reviewed by any shared developer, suggesting collaboration between different teams.

Our study ends with a set of recommendations to help practitioners in the maintenance and evolution of their components and guide researchers in studying cross-component dependencies. We recommend **managers** not to neglect cross-component changes as they occur frequently and over all the releases rather than exceptionally. We also shed light on when to expect such co-changes through qualitative analysis. We also recommend **managers** to select appropriate resources to develop two cross-component dependent changes as having the same developer on both changes can be slower and push the developer of both dependent changes to contribute to a project she/he is less familiar with. We also recommend **practitioners** not to consider migrating from a monolithic to a modular architecture as a lifelong solution, but restructure their architecture when the dependencies among components increase. Finally, our results suggest several take-home messages for **researchers**, among which is to develop mechanisms to identify impacted components by a given change ahead of time, leverage the causes of cross-component dependent changes that we identified in RQ4 to develop better solutions, identify what other components to test given a change to one component (according to RQ4) and identify when it is necessary to refactor a multi-component system to reduce the maintenance dependency.

To facilitate replication of our study, we provide a replication package [2].

The remainder of this paper is structured as follows: Section 2.3 sheds light on the background and related works about the co-evolution of multi-component systems. Section 2.4 provides an overview of the data collection and methodology. Section 2.5 provides answers to our research questions. Section 2.6 presents a set of implications derived from our study findings. Section 2.7

[2] https://github.com/aliarabat/OpenStack/tree/stable

discusses the threats to the validity of our observations. Section 2.8 concludes the paper and discusses potential future research directions.

2.3 Related Work

While our study focuses on the maintenance of multi-component systems so they can better benefit from the advantages of modular architecture, the closest work to our study is hence the migration from monolithic to multi-component systems, in particular, microservices (Section 2.3.1), the evolution of multi-component systems (Section 2.3.2), the maintenance of a software's dependencies which can be seen as external components to a software system (Section 2.3.3), and finally the coupling evaluation in multi-component systems is reported in Section 2.3.4.

2.3.1 Migration from monolith to microservice architecture

Many research efforts Fan & Ma (2017); Gouigoux & Tamzalit (2017); Mazlami *et al.* (2017); Sampaio *et al.* (2017); Richardson (2018); De Lauretis (2019); Fritzsch *et al.* (2019); Li *et al.* (2020) have been performed on migrating a monolithic application to a microservice architecture, so a software system can better benefit from the modularity of the microservice architecture. For instance, microservice-based applications comprise a collection of small, independently maintained and deployable services. Each of these services is owned by a small team of developers; hence, communication between different teams is eventually reduced to ensure the safe and rapid release of the software product Richardson (2018). Mazlami *et al.* (2017) developed a formal model to help with the microservice extraction process from an existing monolithic application. De Lauretis (2019) leveraged a strategy of five different steps, starting from function analysis, through microservice creation, all the way to defining a microservice architecture. Sampaio *et al.* (2017) proposed a service evolution model to support practitioners in the evolution of microservice-based systems. Research efforts have been devoted to help migrating a legacy system to a microservice-based on particular techniques, such as SDLC Fan & Ma (2017), and Strangler Fig pattern Li *et al.* (2020). Furthermore, Gouigoux & Tamzalit (2017) highlighted three major issues derived from an industry case

study when migrating from monolith to microservice, namely, granularity, deployment, and orchestration perspectives. Similarly, Fritzsch *et al.* (2019) presented intentions, strategies, and challenges during the microservice transition process based on a qualitative study on 14 systems across different domains, and 16 interviews with practitioners from 10 software companies. Trabelsi *et al.* (2022); Abdellatif *et al.* (2021) proposed a taxonomy to modernize legacy object-oriented systems to SOA through a set of SE (i.e., service identification) techniques. Later, the authors reported through a survey with 45 practitioners the absence of automation tools supporting the transition from legacy systems to SOA Abdellatif *et al.* (2018). They also built a strategy called "ServiceMiner" Abdellatif *et al.* (2020) which is a bottom-up code-based approach aimed at practically helping the identification of potential services in a legacy system. Similarly, Trabelsi *et al.* (2022) developed a microservice identification technique named "MicroMiner" to support such a transition process. Finally, we refer to an existing systematic literature review for more details about the migration of monolithic systems to microservices Abgaz *et al.* (2023).

Although this line of research suggested techniques, strategies, and models to support the transition from monolith systems to a distributed microservice architecture, such studies do not examine the maintenance of the multi-component systems in general and microservices in particular post-migration. In fact, according to prior studies Sampaio *et al.* (2017); Bogner *et al.* (2019), practitioners still face challenges after migrating their software systems related to the interdependency between different microservices. Thus, our study complements this line of research as we consider the maintenance of the system already migrated.

2.3.2 Co-evolution of components in a multi-component system

The closest line of research to our paper studied the evolution of multi-component systems Sampaio *et al.* (2017); Bogner *et al.* (2019); Lin *et al.* (2023); Assunção *et al.* (2023). Sampaio *et al.* (2017) observed that most of the developers in IBM proceed with manual investigation to manage the evolution of different microservices versions, such a manual task is challenging and error-prone. Similarly, Bogner *et al.* (2019) reported that having independence between different components of a microservice-based system is still an ever-present challenge in

the industry. As such, interviewees stated that ensuring low coupling and high cohesion between services is not a straightforward task. Similarly, Lin *et al.* (2023) found that incompatibility issues during the co-evolution of the WordPress platform and its plugins are common. Hence, 32% of such incompatibilities occur between WordPress and its plugins. Assunção *et al.* (2023) identified three major types of changes during the co-evolution of microservices which are *technical*, *service*, and *miscellaneous*. Thus, the authors concluded that most of the studied microservice-based systems often co-evolve in terms of technical matters (i.e., updating libraries) rather than business concerns.

Our work complements this line of research that pointed out the problem of the co-evolution of different components. For instance, our work considers this line of research as a motivation to empirically understand the co-evolution of different components of a multi-component system. In our study, we wish to understand the co-evolution between two components quantitatively and qualitatively, so managers can better plan their changes and researchers can better understand such a co-evolution so they can develop appropriate solutions that minimize the interactions between different components and/or predict it in advance.

2.3.3 Dependencies in software ecosystems

A large body of research studied the evolution and impact of dependencies in popular software package management systems such as NPM Kula, Ouni, Germán & Inoue (2017); Cogo *et al.* (2021); Chowdhury, Abdalkareem, Shihab & Adams (2022) and Maven Raemaekers, van Deursen & Visser (2014); Jezek, Dietrich & Brada (2015); Soto-Valero, Harrand, Monperrus & Baudry (2021). Kula *et al.* (2017) evaluated the effect of the so-called *micro-package* on the overall npm ecosystem. The authors reported that micro-packages are more frequent in the ecosystem and have a long chain of dependencies, hence, suggesting developers be aware of such potential issues. Cogo *et al.* (2021) investigated the phenomena of dependency downgrades in the same ecosystem. Dependency downgrade (49%) is a common practice among developers according to the derived rationales (i.e., reactive and preventive). Similarly, Chowdhury *et al.* (2022) found that trivial packages (i.e., packages implementing basic features)

are being centrally adopted by a large number of projects, hence, they observe that 16% of the packages in the ecosystem are trivial packages. Such trivial packages can impart around 29% of the packages within the ecosystem. Relating to *maven* package manager, Raemaekers *et al.* (2014) studied the practical use of *semantic versioning*. Therefore, roughly a third of the investigated library releases involve breaking changes. The authors recommend developers consider using semantic versioning as it provides convenient support when performing library upgrades. Jezek *et al.* (2015) observed that incompatibility issues are common between a program and its libraries. Interestingly, Soto-Valero *et al.* (2021) found that bloated packages (i.e., packages that are unnecessary in runtime) in maven are common. In particular, the authors reported that 15.4% and 57% of, respectively, inherited dependencies and transitive dependencies are declared bloated.

Our work differs from this line as we investigate how different components of the same system are co-maintained and how a change in one component depends on another change in another component still of the same software system. In fact, our focus is on horizontal dependencies between the components of the same software system, rather than a system and external artifacts that it uses.

2.3.4 Coupling Evaluation in multi-component systems

Researchers have also examined many different approaches to measuring the coupling degree between components within a multi-component system Candela, Bavota, Russo & Oliveto (2016); Sousa, Bigonha & Ferreira (2019); Zhong *et al.* (2023); Li *et al.* (2023). Such coupling techniques can help practitioners quantify the dependencies among different components, which may pinpoint architecture refactoring strategies. To this end, Candela *et al.* (2016) carried out a quantitative and qualitative study to better understand the main factors determining software remodularization. Hence, through an extensive study on 100 open-source projects and a survey with 29 developers, the authors found that the refactoring efforts based solely on cohesion/coupling are not sufficient to reach a high-quality remodularised system. Furthermore, Sousa *et al.* (2019) investigated the evolution of the coupling, which is the level of dependence

among modules. The authors identified that legacy classes (aka., classes that are introduced in the first version of the system) highly impact the coupling growth. Zhong *et al.* (2023) proposed a coupling metric that measures the degree of coupling among microservices. For instance, their proposed coupling is calculated based on the number of interfaces (i.e., a class with a set of APIs) in the first microservice that are invoked by the classes of the other microservice and vice versa. Moreover, Li *et al.* (2023) proposed another coupling technique, called organizational coupling, which measures the collaboration between developers belonging to different microservice teams. Such a coupling is mainly derived through microservice ownership (i.e., identifying microservice teams) and cross-service collaboration (i.e., how frequently developers contribute to the other microservices).

Even though the aforementioned studies proposed a variety of approaches to evaluate the coupling between different components, such coupling metrics are code-centric metrics. We instead focus on the coupling between components in terms of maintenance activities. We define two components are dependent if they co-evolve. These studies do not also focus on understanding the maintenance of different components, and how they evolve in practice.

2.4 Methodology

Since we study the co-evolution between different components, we need to identify a case study with multiple components with enough data to allow us to systematically identify related changes. Therefore, the objective of this section is to discuss our case study OpenStack (Section 2.4.1), how we identify different components of OpenStack (Section 2.4.2), how we identify the component associated with each change and related changes (Section 2.4.3), how we identify user bots (Section 2.4.4), and the metrics that we used to answer our research questions (Section 2.4.5). Note that each research question has its own approach subsection that further details how we answered each research question.

2.4.1 OpenStack as a case Study

To answer our research questions, we study OpenStack since it has an explicit way of declaring dependencies between different changes, including those that are related to different projects. We also use OpenStack, similar to a large number of prior studies Tong *et al.* (2015); Yamato, Katsuragi, Nagao & Miura (2015); Niwa, Kasuya & Kitahara (2017); Teixeira & Karsten (2019); Zheng, Feng, Yu, Yang & Wu (2019), as a large-scale open-source software system. Finally, we also consider OpenStack as it has a large number of projects (aka., components) that are distributed over different services. For example, *Nova* is a compute service that has 12 projects, including a project called *nova*[3], *nova-powervm*[4], and *python-novaclient*[5]. Another example is *Swift*, an object storage service containing several related projects, such as *swift-bench*[6] and *swift-specs*[7].

2.4.2 Identification of the components of OpenStack

From the OpenStack documentation and the history of changes, we initially identified 61 services that together represent 1,310 distinct projects. Each of the 61 services has a team for which it is responsible, while that team has the same name as the service it is responsible for [9]. Note that 54 services are explicitly mentioned in the documentation as services [10] that are shown in Figure 2.1, while we derive the remaining seven services from the teams' names, such as the service *Requirements* which is a transversal service for configuring dependencies. Additionally, we refined the obtained services by removing *eight* services that do not have any change in Gerrit to avoid any misleading conclusions. We then extract the 1,310 projects from the history

[3] https://opendev.org/openstack/nova

[4] https://opendev.org/openstack/nova-powervm

[5] https://opendev.org/openstack/python-novaclient

[6] https://opendev.org/openstack/swift-bench

[7] https://opendev.org/openstack/swift-specs

[8] https://www.openstack.org/software/

[9] https://releases.openstack.org/index.html#teams

[10] https://www.openstack.org/software/

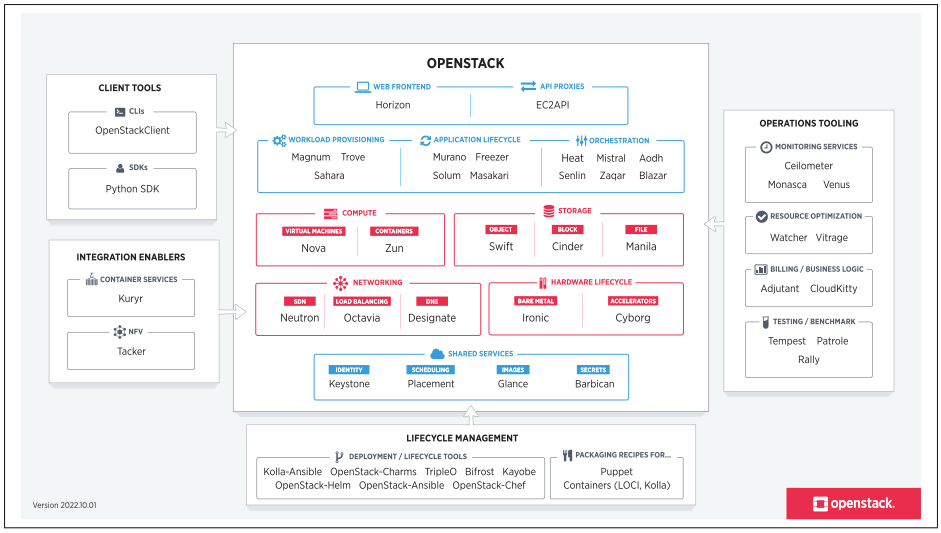


Figure 2.1 A landscape of various OpenStack services obtained from OpenStack documentation homepage[8]

of changes in OpenStack [11]. Then, we map each project to its associated service by leveraging the following heuristic:

- **H1:** We first look at the project names that were modified by each team responsible for a service (i.e., note that a team has the same name for the service it is responsible for). For example, the projects *Openstack/nova* and *Openstack/os-vif* are two projects that belong to the *Nova* service [12] [13].

Our classification ends up with **53 services** and **587 out of 1,310 classified projects**. Note that the non-classified projects are excluded only from the cross-service-related changes and not from the cross-project changes. Cross-service changes represent two changes that belong to two completely different services among the 53 services. Cross-project changes represent the changes that belong to different projects among the 1,310 projects.

[11] https://releases.openstack.org/index.html

[12] https://releases.openstack.org/teams/nova.html

[13] https://governance.openstack.org/tc/reference/projects/nova.html

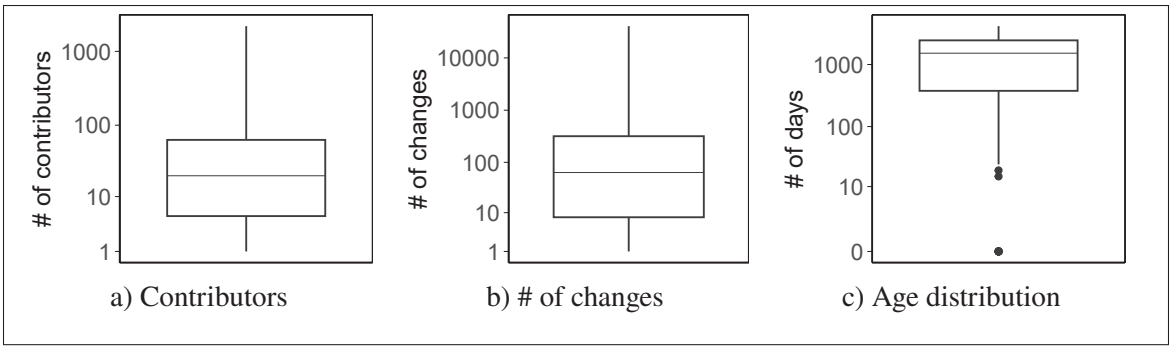


Figure 2.2 Key characteristics of the studied OpenStack projects regarding the distribution of (a) **contributors**, (b) **changes**, and (c) **age** in days. All Figures were represented in logarithmic scale

Our studied projects have a median of 20 human contributors (excluding bots), 63.5 changes, and a median age of 1,518.5 days, as shown in Figure 2.2. Our analysis covers a large number of changes amounting to 516,509.

2.4.3 Related changes identification

To identify cross-project and cross-service related changes, we first map each change to its corresponding project and service, then identify changes that are related to each other.

2.4.3.1 Mapping changes to components

We map each change to its corresponding project using the tag *"Repo"* as shown in the example in Figure 2.3. We then leverage the mapping of projects to their corresponding services using the mapping obtained following the approach discussed in Section 2.4.2.

To identify dependent changes, we leverage the following tags that are mentioned in the changes' description:

- **Depends-On:** is a tag that can be added to a change "A" to indicate on which other changes "A" depend. For example, the change in Figure 2.4a depends on the change *844490*. We

[14] https://review.opendev.org/c/openstack/neutron/+/874797

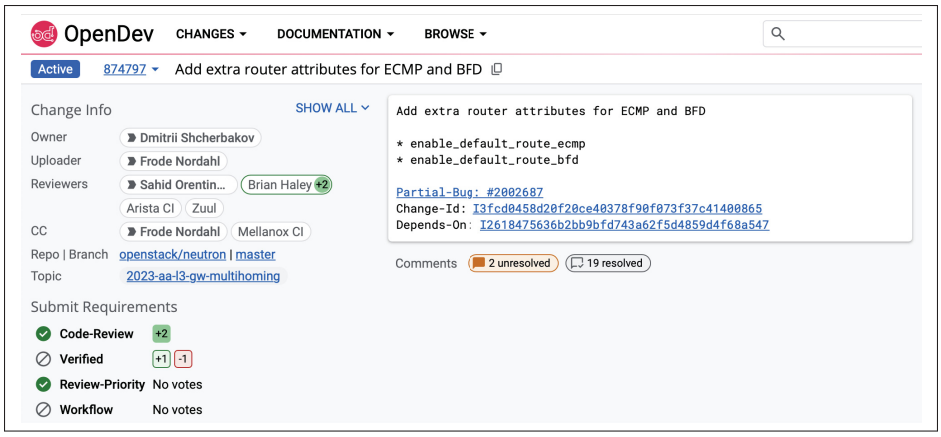
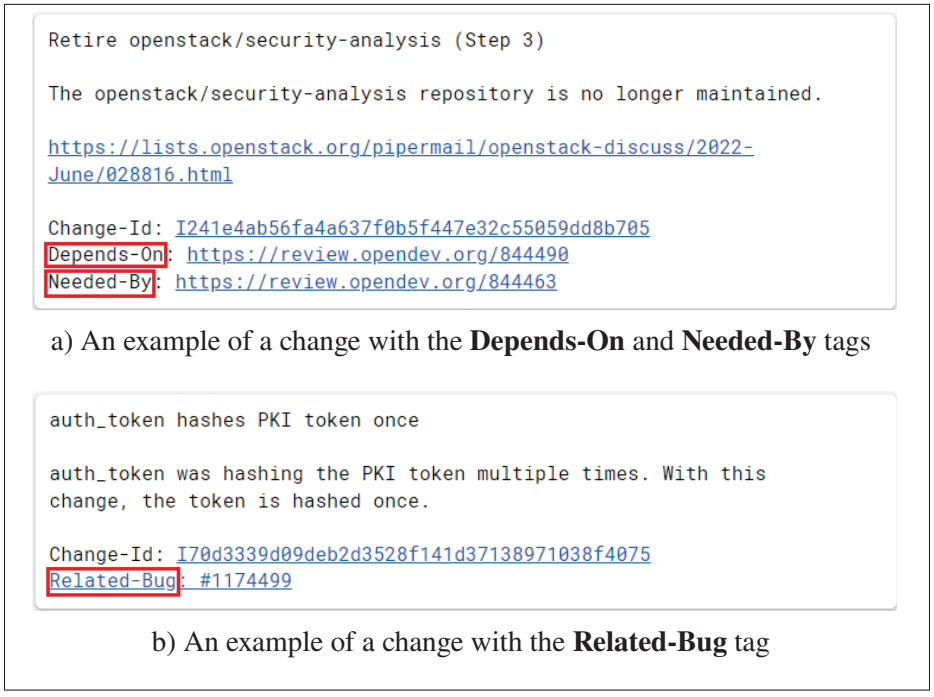


Figure 2.3 An example of an OpenStack change with a set of information considered in our analysis[14]



a) An example of a change with the **Depends-On** and **Needed-By** tags

b) An example of a change with the **Related-Bug** tag

Figure 2.4 Example of changes with tags that are used to identify dependent changes

apply the following regular expression `Depends-On:\s[a-zA-Z0-9/\.\:\+\-\#]{6,}` to extract the "Depends-On" id.

- **Needed-By:** is a tag that mentions that a change "A" is needed by another change "B". For example, the change in Figure 2.4a is needed by the change *844463*. Similarly, we

leverage `Needed-By:\s[a-zA-Z0-9/\.\:\+\-\#]{6,}` regular expression to obtain the "Needed-By" id.

- **Related-Bug:** we also consider that changes for the same bug are related. To identify the bug to which the change is related, we leverage the "Related-bug" tag. To extract "Related-bug", we use the pattern `Related-Bug:\s#\d+`. Figure 2.4b shows a change related to bug *1174499*.
- **Change-Id:** we also consider two related changes if they share the same "Change-Id".

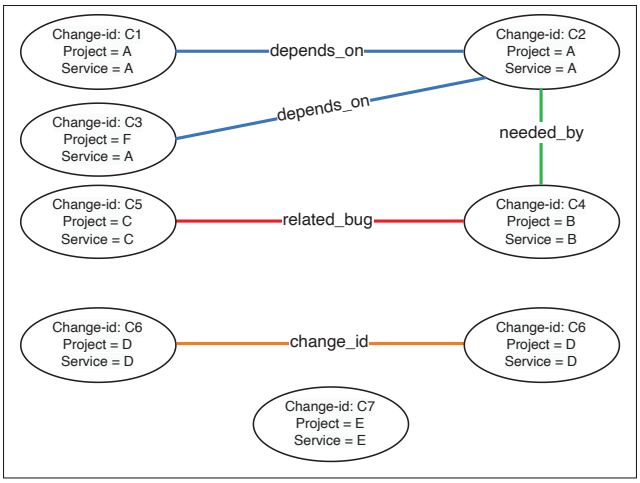


Figure 2.5 An example of dependencies between OpenStack changes where nodes are changes made to components and edges are the relationships between them

We then leverage the previous tags to build a graph of change dependencies and identify cross-project and cross-service related changes. An illustration of such a graph of dependencies is shown in Figure 2.5.

2.4.4 Bots Identification

Note that to identify whether a change owner within OpenStack is a human developer or a bot, we leverage an attribute called `tags` indicated in the user-related information. Therefore, a user bot is assigned the value `["SERVICE_USER"]`. Listing 2.1 provides a concrete example of the Zuul bot which automates a wide range of code review practices such as test execution and running gate jobs.

Listing 2.1: A real example of the Zuul bot

```
{
  "_account_id": 22348,
  "name": "Zuul",
  "username": "zuul",
  "tags": ["SERVICE_USER"]
}
```

2.4.5 Collected Data

To answer our research questions, we collect all the changes of OpenStack and extracted the data shown in Table 2.1. The same table also shows in which research question we used a given data, and we further detail how we answer each research question in the approach of the research question itself.

Table 2.1 Various information used to address each research question

Information	Usage	Description
Number	RQ1-2-3-4-5	Unique ID of the change.
Project	PQ, RQ1-2-3-4-5	The project on which the change is made.
Team	PQ, RQ1-2-3-5	The team associated with the change.
Project type	PQ, RQ2	The type of the project (i.e., modified, non-modified, or archived).
Status	PQ, RQ1-2-3-4-5	The status of the change (i.e., open, abandoned, and merged).
Created	RQ2	The creation date of the change.
Insertions	RQ5	The number of inserted lines in the change.
Deletions	RQ5	The number of deleted lines in the change.
Reviewers	PQ, RQ5	The reviewers (account IDs) who participated in the review process of the change.
Messages	RQ4-5	The messages that were exchanged during the review process of the change.
Revisions	RQ5	Different revisions of the change.
Release	RQ2	The release on which the change was made.
Owner	PQ, RQ5	Original author (account ID) of the change.
Owner nature	PQ, RQ1-2-3-4-5	Whether the change is made by a human or service bot.
Commit message	RQ3-4	The commit message of the change.

PQ. Do OpenStack teams have ownership over the components that they maintain?

Motivation: The goal of this preliminary research question is to empirically understand to which extent ownership of different services and projects exists in OpenStack. In particular, before investigating how different projects and services co-change and whether there is a need for approaches that help with the co-evolution of different projects/services, we first aim to investigate whether OpenStack teams have ownership over different OpenStack projects and services suggesting that these projects and services are expected to evolve independently. To do so, we will investigate whether there is an intersection between different OpenStack teams regarding the projects they are responsible for and modify, whether projects share developers, and whether developers and reviewers contribute to different projects or stick to one project.

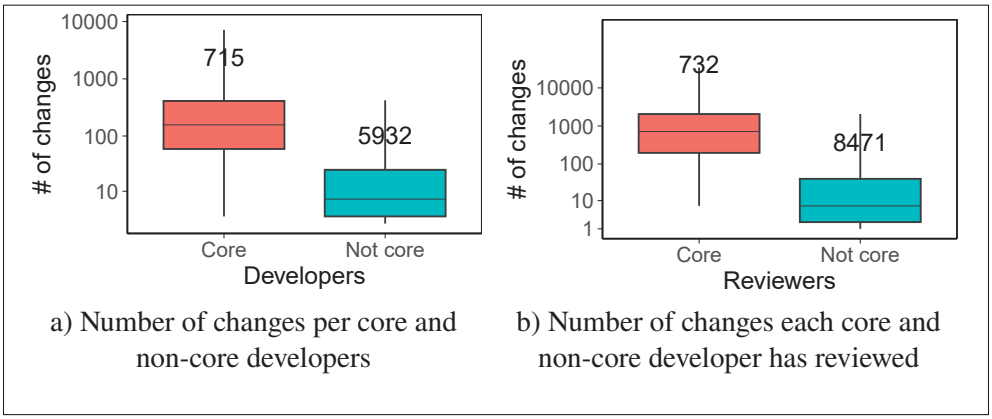
Approach: To investigate whether *different teams have ownership over different projects*, we extract the teams and the projects that each team modified in the past according to the documentation of OpenStack [15]. We then measure the intersection in terms of projects between each pair of teams. For example, we evaluate whether a pair of teams modified the same projects in the previous releases. The smaller the intersection between the teams, the more the ownership concept is respected in the context of the OpenStack projects, and these projects are expected to evolve independently.

As the previous analysis is to measure the intersection between different teams, we also measure whether *individual developers contribute to different projects* or whether they stick to the project their respective team is responsible for. In particular, we quantify for each pair of two projects the number of unique developers in each of the two projects and the number of developers who contributed to both projects. Similarly, we quantify for each pair of projects the number of reviewers who reviewed changes for just one of the two projects or both projects. We repeat the same experiment at the service level. For this last analysis, we collect all the changes of OpenStack as discussed in Section 2.4.3, the developers of these changes, the reviewers of these changes, and on which project such a change is made. We exclude casual contributors (defined

[15] https://releases.openstack.org/index.html#teams

below), so they do not bias our analysis. We also analyze whether core contributors are more likely to contribute to different projects than ordinary contributors, which are defined as follows:

- **Casual contributors**: We exclude casual contributors from our analysis. Casual contributors are developers who have performed at most one change to OpenStack, as defined by Pinto, Steinmacher & Gerosa (2016) and used by Batoun, Yung, Tian & Sayagh (2023). We exclude such contributors from our analysis since they make just one change; hence, by definition, they never contributed to more than one OpenStack project. In other words, we exclude them to avoid biasing our findings on the percentage of developers who contributed to more than one project.
- **Core developers**: these are the main developers of a project as listed in the OpenStack documentation [16].
- **Contributors**: these developers have more than one change in OpenStack and are not part of the core developers.



a) Number of changes per core and non-core developers

b) Number of changes each core and non-core developer has reviewed

Figure 2.6 The comparison of the distribution of OpenStack changes between core vs. non-core (a) **developers** and (b) **reviewers**. The number of individual developers is placed on top of the boxplots

Our dataset consists of 715 core and 5,932 non-core developers, and 732 and 8,471 core and non-core developers who reviewed changes. The core and non-core developers have a median of 157 and 7 OpenStack changes, respectively. Similarly, core and non-core reviewers participate

[16] https://review.opendev.org/admin/groups

in a median of 714 and 7 changes, respectively. Figure 2.12 shows the number of changes each developer and reviewer pushed/reviewed within OpenStack.

Table 2.2 The distribution of OpenStack projects among different releases notes' teams along with their intersection percentages

Team pair	Common projects	Intersection (%)
Neutron - Octavia	*octavia* and *neutron-lbaas*	6.06%
Documentation - Oslo	openstack-doc-tools and openstackdocstheme	4.54%
Horizon - Manila	manila-ui	2.7%

Results: 99.76% (i.e., 1,972 out of 1,975 pairs) of the investigated pairs of OpenStack teams do not share any project. We observe that only three pairs of teams (shown in Table 2.2) share one to two projects. While these teams share projects, we observe from the documentation that these pairs of teams did not release the same versions of their shared projects. For example, versions 0.5.0 to 0.5.2 and 0.9.0 to 0.9.2 of the "octavia" project were released by the "Neutron" team, while the "Octavia" team released other versions of the same project. Similarly, we observe that the "manila-ui" project was originally maintained by the "Horizon" team up to the OpenStack release "Mitaka". Starting from the following release (i.e., "Newton"), the "manila-ui" project was maintained by the "Manila" team. The remaining shared projects show similar observations. Thus, even when two teams share a project, they do not maintain it simultaneously. One possible explanation is that OpenStack tried to promote the ownership principle of different teams over their corresponding projects. Consequently, only one team at a time is held responsible for the development of such a project [17] [18]. Our results suggest that the ownership of different teams over different projects exists for the OpenStack project.

While 60.43% and 98.79% of our studied pairs of projects and services, respectively, have at least one developer who contributed to both projects or services of the pair, such an intersection across projects/services is not high. While 39.57% of our studied pairs of projects do not share any developers, we observe 1.19% of the studied pairs are developed by the same

[17] https://specs.openstack.org/openstack/designate-specs/specs/juno/zone-migration-between-tenants.html

[18] https://docs.openstack.org/keystone/latest/getting-started/architecture.html#projects

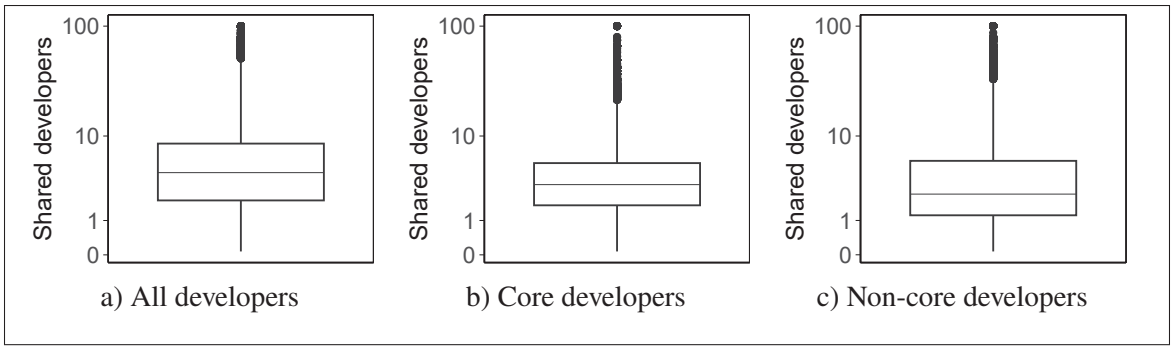


Figure 2.7 Distribution of developers' intersection for (a) **all**, (b) **core**, and (c) **non-core** developers between the studied OpenStack pairs of projects

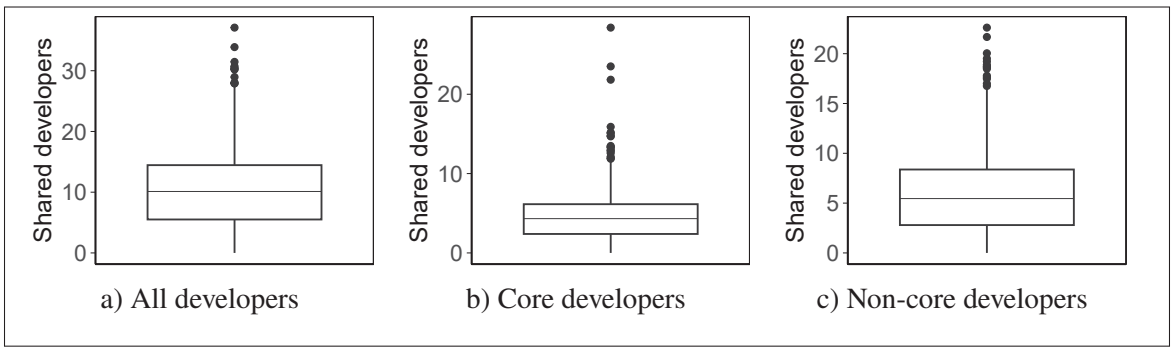


Figure 2.8 Distribution of developers' intersection for (a) **all**, (b) **core**, and (c) **non-core** between the studied OpenStack pairs of services. Note that the intersections are defined in a percentage format

developers (i.e., both projects of the pair share 100% of the developers). By focusing on the 60.43% of the pairs of projects with at least one developer in common, we observe that these pairs share a median of just 2.83% of their respective developers (3.12% and 2.38% core and non-core, respectively), as shown in Figure 2.7. We observe similar results for the pairs of services. We observe that 98.79% (1,808 out of 1,830) pairs of services share at least one developer. For the pairs having an intersection in terms of developers, they share a median of 10.21% of their respective developers (a median of 4.38% and 5.59% of core and non-core developers, respectively), as shown in Figure 2.8.

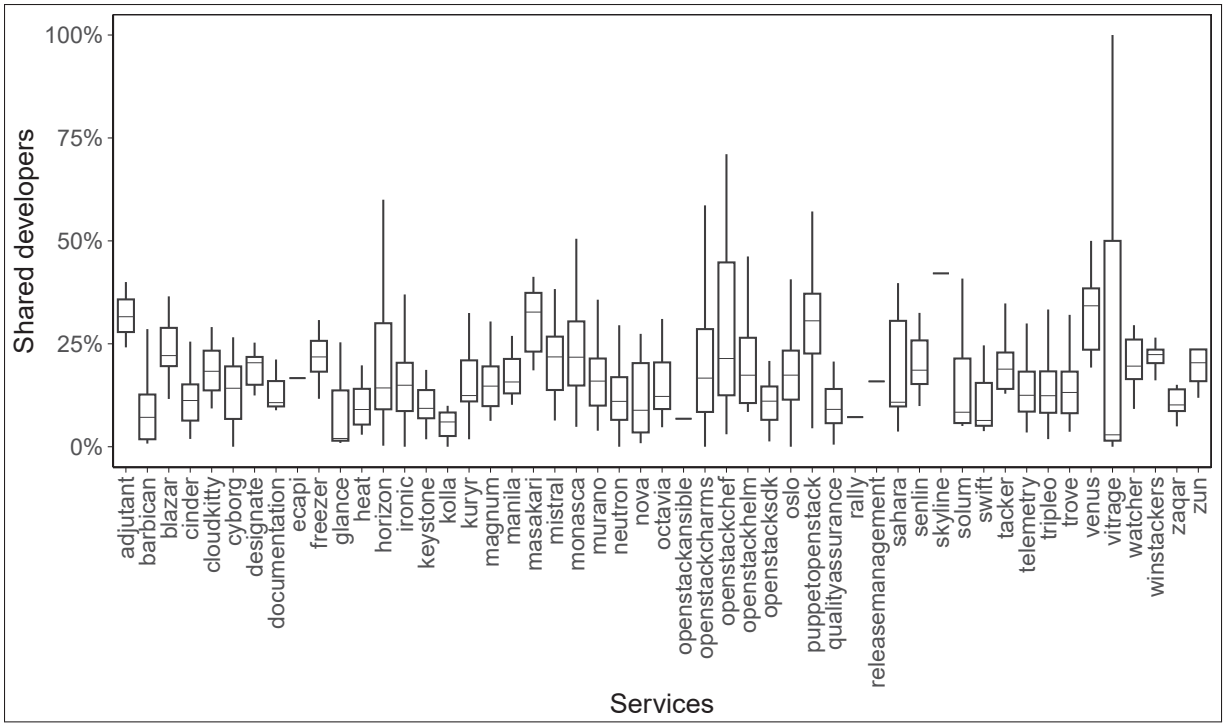


Figure 2.9 Distribution of developers' intersection between pairs of projects within the same service. Note that four services ("requirements" and "shorlets") were not plotted because they contain only one project

Additionally, projects of the same service have shared developers, but such an intersection is not high, as shown in Figure 2.9. The pairs of projects within the same service share a median percentage of developers as low as 2% ("glance" service) and as much as 42.10% (skyline service). In contrast, the maximum shared developers intersection is related to the "skyline" service, which can be expected since that service was just introduced and contains only two projects. In addition, we also observe that services related to logging ("venus"), admin ("adjutant"), and infrastructure ("puppet-openstack" and "masakari") tasks involve a median of developers' intersection ranging from 30% to 34% as shown in the same figure. A potential explanation for such high numbers compared to other services is that these services are for configuration and transversal to the whole OpenStack, having potentially a high dependency on each other.

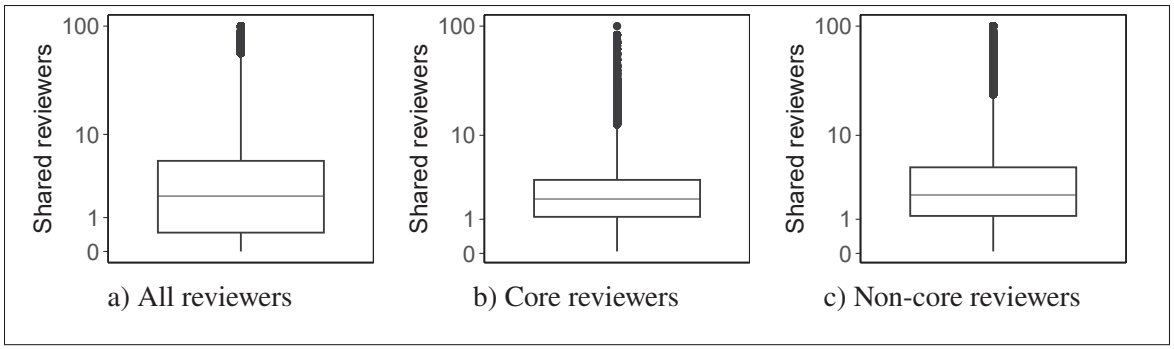


Figure 2.10 Distribution of reviewers' intersection for (a) **all**, (b) **core**, and (c) **non-core** between the studied OpenStack pairs of projects

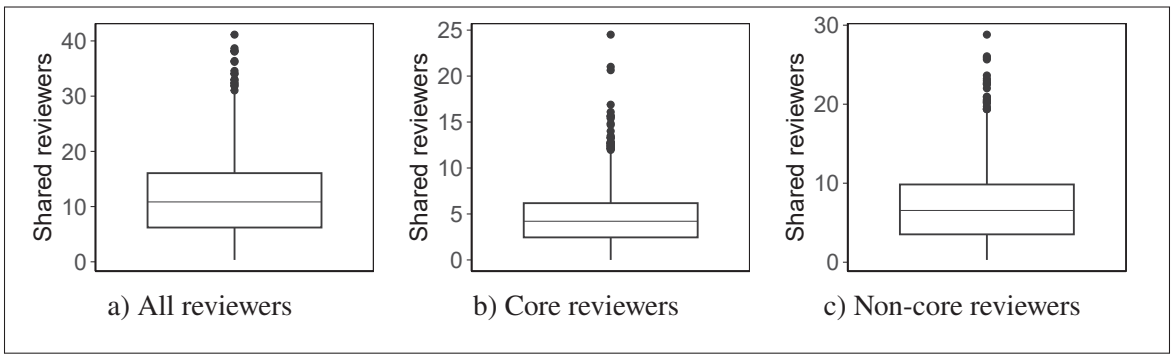
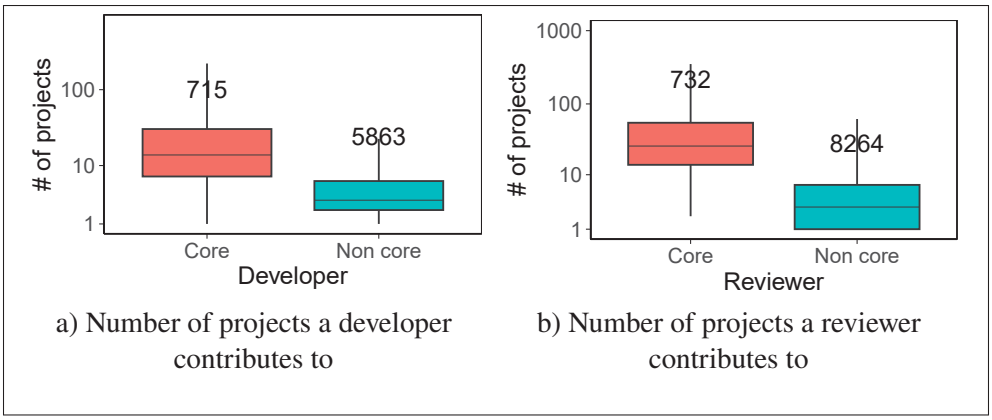


Figure 2.11 Distribution of reviewers' intersection for (a) **all**, (b) **core**, and (c) **non-core** between the studied OpenStack pairs of services. Note that the intersections are defined in percentage format

Our results hold for the reviewers as 78.77% and 100% of the pairs of projects and services have at least one reviewer in common, but such an intersection is not high. We observe that the projects with at least one reviewer in common have a median of 3.17% reviewers in common, among which are 2% and 2.22% core and non-core contributors as shown in Figure 2.10. Note that for core and non-core reviewers' intersection, we consider just the pairs of projects with at least one core and one non-core contributor, respectively, who reviewed changes for both projects. Similarly, the pairs of services with at least one reviewer share a median of 10.84% (4.35% and 6.54% core and non-core) of their respective reviewers as shown in Figure 2.11.



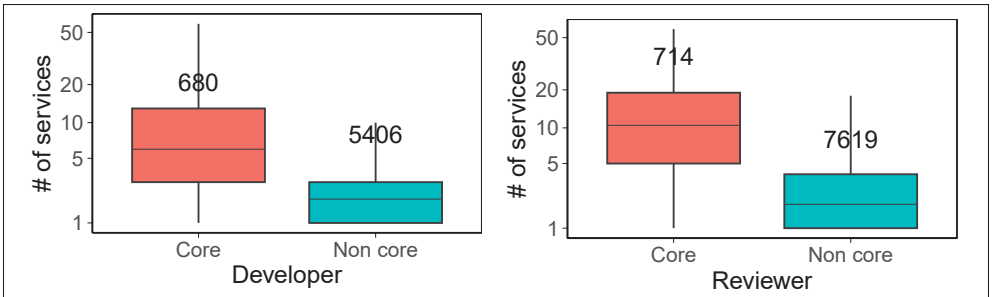
a) Number of projects a developer contributes to

b) Number of projects a reviewer contributes to

Figure 2.12 Number of OpenStack projects a (a) **developer** and **reviewer** contribute to, broken down into core and non-core sets. The number of individual developers is placed on top of boxplots

79.18% of the studied developers contribute to at least two OpenStack projects, with over half (62.56%) contributing to at least two services. On a median, developers participate in three different OpenStack projects. While 20.82% tend to focus on only one project, we observe that a developer contributes to a maximum of 671 projects. The core developers have contributed to a median of 14 projects, whereas non-core developers have been involved in a median of three projects. Our 8,996 studied reviewers participated in a median of three projects. The core developers (732 reviewers) reviewed changes for a median of 26 projects, whereas the non-core reviewers (8,264 reviewers) were involved in a median of three OpenStack projects. These findings are depicted in Figure 2.12a and Figure 2.12b.

Our results hold for how developers and reviewers contribute to different services. Both developers and reviewers contribute to a median of two services. 40.8% (2,373 out of 5,815) of the OpenStack developers tend to focus on only one service. As it can be expected that core developers, with their large expertise, contribute to multiple services of OpenStack, we observed that they contribute to a median of five OpenStack services. Interestingly, even non-core developers contribute to a median of two services (Figure 2.13a), while less than half (44.23%) of non-core developers participate in only one service. We observe similar results for the number of services reviewers review. 40.62% of the reviewers focus on reviewing projects related to



a) Number of services a developer contributed to

b) Number of services a reviewer contributes to

Figure 2.13 Number of OpenStack services a (a) **developer** and **reviewer** contribute to, broken down into core and non-core sets. The number of individual developers is placed on top of the boxplots

a single service. We also observe that core developers review a median of eight services and non-core developers review a median of two OpenStack services (as presented in Figure 2.13b). Figure 2.13 highlights the number of services each developer and reviewer participated in within OpenStack.

Summary of PQ

While the teams at a high level are responsible for different projects and projects do not share a large percentage of common developers and reviewers, even for projects for the same service, developers and reviewers of these teams often contribute to multiple projects and services, creating a potential inter-dependency between these projects and changes in one project/service requiring changes in another one.

2.5 Empirical Results

RQ1. How prevalent is the co-evolution of different components?

Motivation: While the preliminary research question shows that developers contribute to multiple projects, we aim by this research question to investigate if such dynamism is creating a changing dependency between different components. Quantifying these changes will help

managers better understand the existence of changes, hence better planning for future changes and suggesting future studies to develop mechanisms to reduce such a dependency across different components to have better ownership of developers over their projects.

Approach: To quantify the dependencies across multiple projects/services, we first identify whether a change has a dependency on another change by leveraging the four tags (i.e., depends-on, needed-by, related-bug, and change-id) and methodology discussed in Section 2.4. Then, we map each change to its project (the Repo tag in Figure 2.3) to identify whether two dependent changes are related to the same project/service (aka., within-project/service changes) or they belong to two different projects (aka., cross-project/service changes). Thus, using these discussed tags and projects, we build a change dependency graph composed of changes represented as nodes and dependencies represented as the edges between nodes. A node contains the change-id and its corresponding repository. Figure 2.5 shows an example of our graph of dependencies.

From the graph, we identify the number of changes with a dependency to quantify the number of changes with a *dependency* on another project/service (aka., cross-project/service change). From our previous example of Figure 2.5, we have 7 out of 8 changes with a dependency. Among the seven changes, 4 changes ("C2", "C3", "C4", and "C5") are dependent on another component, and three ("C1" and and the two changes with "C6") are single-component changes. From our mapping between project and services obtained following the approach discussed in our methodology (Section 2.4.2), we also count the number of cross-service changes. We have three cross-service and four within-service changes from our running example.

We also quantify the number of projects/services involved in a set of related changes (aka., chain of changes). Since the more projects involved in a set of related changes, the more synchronization and communications should take place; hence the more challenging that change is. Therefore, leveraging our graph of dependent changes, we also quantify the number of projects/services involved in a chain of related changes. For example, four projects (i.e., A, B, C, and F) are related to the chain of cross-component changes shown in our running example.

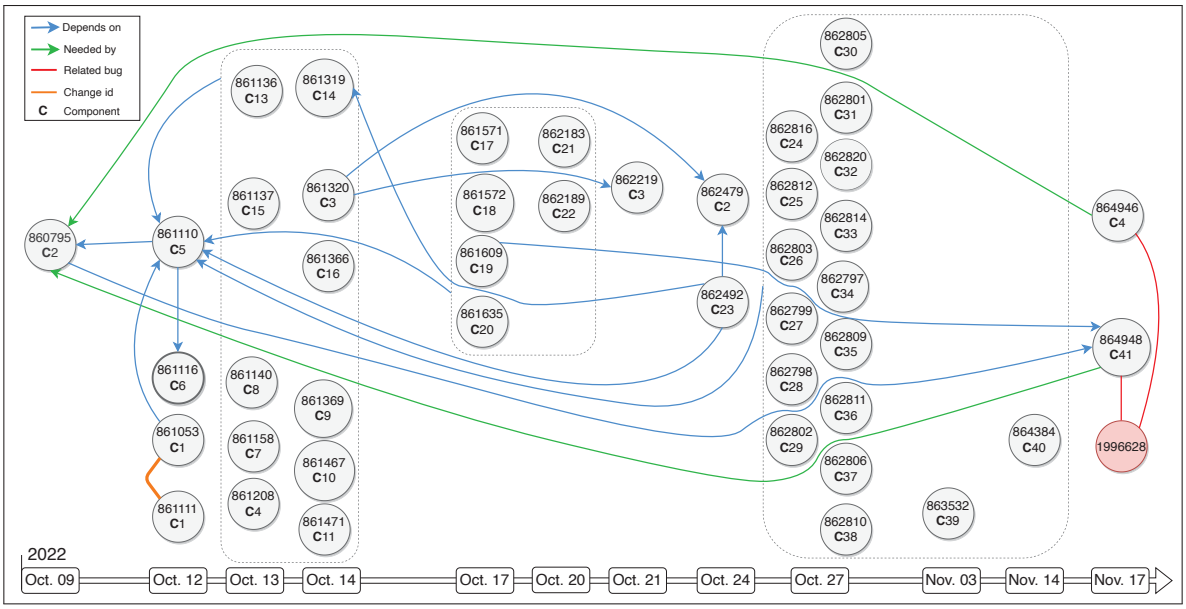
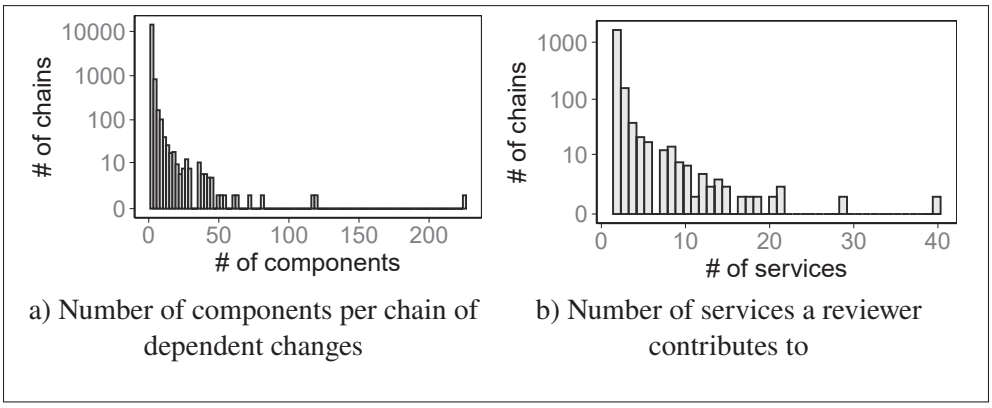


Figure 2.14 A real example of a chain of 44 OpenStack dependent changes involving 40 distinct projects. Most of the relations among them are obtained through "Depends-On" tag (43 out of 48), then 2 for "Related-Bug" and "Needed-By", and "Change-Id" with only one relation. Note that nodes grouped inside a grey-dotted box share the same dependency. Red node is introduced bug

Results: 52,069 and 10,295 changes are cross-project and cross-service dependent, respectively. For instance, 23.49% (a total of 165,033 out of 537,387) of the whole changes of OpenStack depend on other changes. Among these changes, 31.5% (52,069 out of 165,033) of our studied changes depend on at least one change that belongs to a different project. 6.32% of the dependent changes cross the borders of different services, which might be more challenging as these dependent changes are across different services that provide different features, so they are less expected to be dependent, as is the case for projects of the same service. Figure 2.14 shows an example of a modification to the "devstack" project (which creates and initiates a full OpenStack environment) that resulted in an upgrade of the running Ubuntu release (from "focal" to "jammy") of its nodeset (i.e., a node typically runs an OpenStack service, e.g., compute node). This change required a change to the "tempest" project (an OpenStack testing framework) to update the version of its nodeset operating system to "jammy". Consequently, most of the changes (34 out of 44 changes highlighting a message like *"testing jobs on Ubuntu Jammy*

(22.04)"), including tests, were performed against the change on "devstack" across various projects (e.g., "neutron").



a) Number of components per chain of dependent changes

b) Number of services a reviewer contributes to

Figure 2.15 The number of chains of dependencies involving a certain number of (a) **projects** and (b) **services**

The cross-project and cross-service dependent changes occur in a chain of dependent changes that can involve more than two projects in 20.7% of our studied chains and a median of 2 and 1 participating projects and services, respectively. 26.05% (15,467 out of 59,375) and 5.06% (3007 out of 59,375) of our studied chain of dependencies have at least one cross-project and one cross-service change dependency, respectively. While the majority (79.33%) of our studied chains of changes with a cross-project dependency involve two projects, as shown in Figure 2.15a, a chain can have up to 224 distinct projects. Similarly, we observe that 84.83% of our chain of changes with at least one cross-service dependency have two services and a maximum of 40 services, as shown in Figure 2.15b. Our results hint that dependent changes can involve multiple projects requiring further coordination between different teams.

Given the chain containing 224 OpenStack components (199 out of 231 dependent changes are merged), OpenStack has implemented a new project dependency management approach where the "requirements" team maintains and controls all versions of OpenStack libraries that each project may depend on. However, different OpenStack projects may use dependency versions different from each other, which could not be achieved by the global dependency configuration leveraged by the "requirements" project. Thus, based on these cross-component

dependent changes, each project team had to create its dependency management file, called "lower-constraints.txt", containing the lowest versions of dependent libraries. Consequently, each project is, by now, more flexible in specifying the desired versions of libraries they depend on without impacting the others. Similarly, the same example involves the highest number of services, amounting to 42.

Summary of RQ1

We observe that cross-project and cross-service change dependencies are common (52k over 537k studied changes), leading to a chain of dependent changes with a median of two and up to 224 distinct projects. Hence, **OpenStack should consider dependent changes in their planning as a change to one component may propagate through several components.**

RQ2. How does the prevalence of cross-component changes evolve?

Motivation: The goal of this research question is to investigate whether cross-project and cross-service dependent changes frequently occur over time and across releases, so managers should always book time and resources for such changes. Our findings will also shed light on whether such cross-project and cross-service change dependencies are getting more important over time, potentially leading to more interconnections between different projects and teams.

Approach: To determine how cross-project and cross-service dependent changes evolve over time and across releases, we leverage the cross-project/service changes (i.e., changes that are related to another change that belongs to another project/service) used to answer the previous research question. We identify the creation dates of the release related to each dependent change. To identify the release on which a given change is made, we look up if a change is made in a branch whose name starts with the "stable/" prefix followed by the release name or pick up the release issued right after that change if it is made on the main branch.

We quantify the evolution of changes over different years and different releases. We quantify the number of cross-project dependent changes created yearly from 2011 to 2022. We also

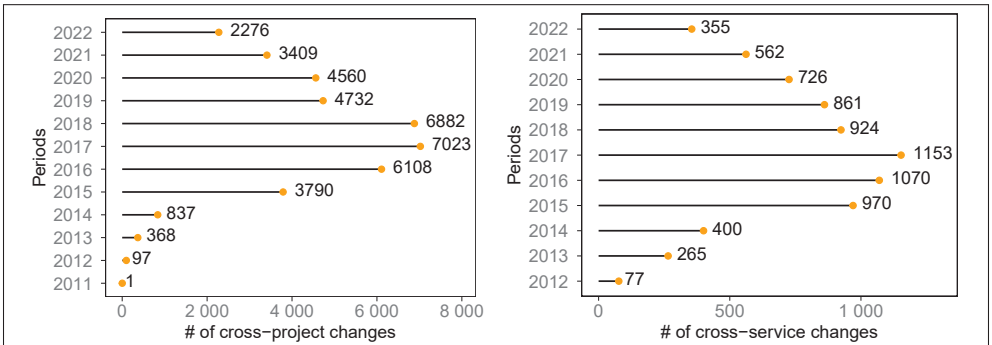
quantify the number of dependent cross-project changes associated with each release. Similarly, we applied the same approach for cross-service changes over time and across releases. Note that for some dependent changes that are across different releases, we increment both releases since they both have a cross-project/service-dependent change.

We also investigate whether there is a relation between the number of co-changes and the complexity of a release in terms of the number of modified projects in that release from one side, and the complexity of OpenStack in terms of the number of existing projects from the other side. In particular, we investigate whether more projects modified in a release are associated with a higher number of cross-project dependent changes. To do so, we compute the number of projects modified during each release according to the changes obtained from the OpenDev platform. We also examine if the number of archived projects in a release/year would lead to a decrease in cross-project changes. To draw such an analysis, we split projects into the three following categories:

- **Archived projects:** represent projects that are no longer maintained by OpenStack and have at least one change. Hence, they become dead projects. To identify archived projects, we download previously-archived OpenStack repositories from GitHub [19]. Each repository has an archival statement at the beginning of the page. Thus, we apply a regular expression pattern to retrieve the archival dates. We then transform the resulting dates into a proper date format for further processing. For instance, the "Bandit" archived project involves the following message at the beginning of the page in a yellow-background box: *"This repository has been archived by the owner on Jun 26, 2020. It is now read-only."*. We then filter projects that have at least one change.
- **Modified projects:** are the set of projects that were modified by OpenStack developers in a release/year. To identify modified projects, we leverage OpenStack changes in each year/release and the associated projects to each of these changes.
- **Non-modified projects:** indicates projects that are not modified in the current release/year, and modified at least once and not archived in the previous releases/years.

[19] https://github.com/openstack-archive

We also quantify the evolution of dependencies between projects, which is measured as the number of projects that a given project depends on across different years/releases. For example, project X depends on two other projects in the first release, whereas it depends on four projects in the second release, etc. To do so, we use dependent changes involving multiple projects and count the number of dependent projects associated with each project. For instance, a change X in component "A" and Y in another component "B", and both of them were created in the same year. Hence, "A" and "B" will each have one dependent project. Note that if X and Y were created in different years/releases, then they are not considered dependent projects. Additionally, we apply the same procedure for cross-project dependencies across releases, where only dependencies happening during the same release are counted. To achieve such consistency throughout this paper, only merged changes are included in our study.



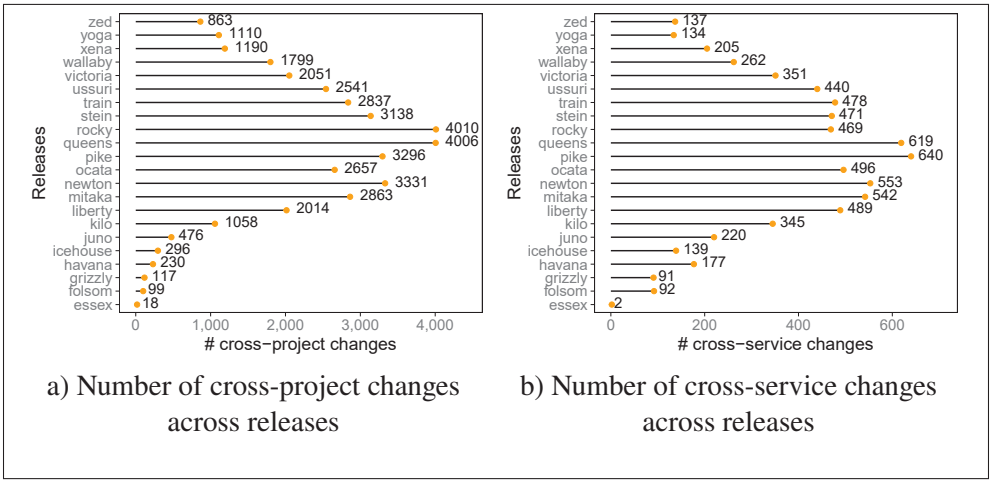
a) Number of cross-project changes over time

b) Number of cross-service changes over time

Figure 2.16 Evolution of (a) **cross-project** and (b) **cross-service** changes over time. Both figures follow the same trend

Results: Cross-project and cross-service changes have grown superlinearly over the first five years of the OpenStack project. We observe that cross-project changes are gaining more importance over time. For example, the number of reported changes that depend on changes in other projects increased exponentially, particularly, between the 2011 (only one cross-project dependent change) and 2016 (6,108 cross-project changes) periods, as shown in Figure 2.16a. Afterward, we observed a gradual increase in the number of cross-project changes in 2017 (7,023) which recorded the highest rate, then remained fairly stable in 2018 (6,882). Over the

following years, we observe a slow drop in the number of cross-project dependent changes. We found similar patterns for cross-service changes over time, except in 2011 where one can observe no dependencies between changes involving different services, as illustrated in Figure 2.16b.

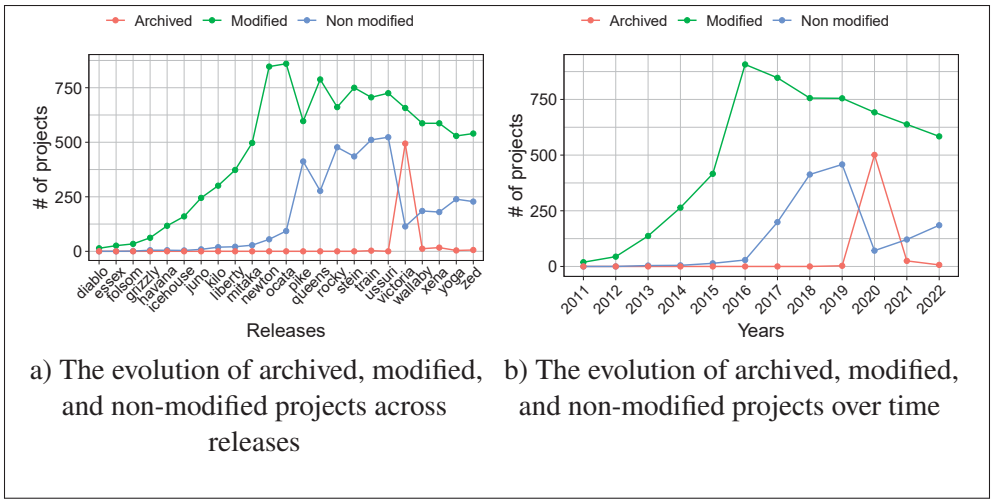


a) Number of cross-project changes across releases

b) Number of cross-service changes across releases

Figure 2.17 Evolution of (a) **cross-project** and (b) **cross-service** changes across releases. The first four releases were omitted given the absence of related cross-project/service changes

Over its earliest stages, OpenStack has seen an explosive increase in the number of both cross-project and cross-service changes with "rocky" and "queens" releases having the highest proportion, respectively. We observe a steady growth of changes having dependencies with changes across multiple projects, as shown in Figure 2.17a. More precisely, between the "essex" release, which had 18 cross-project changes, and "icehouse" release with 296. Starting from the "juno" release, the number of cross-project dependent changes has been going through the roof up to the "newton" release (3,331), while it has remained less or more stable during the following two releases. The two subsequent releases namely, "queens" and "rocky", had 4,006 and 4010 (the highest proportion of recorded cross-project changes), respectively. Right after the "stein" release, we observe that cross-project dependent changes have gradually dropped off over the eight recent releases up to "zed" (which came out on October 5th, 2022). We observe similar trends for cross-service changes evolution across different OpenStack releases, except for some stable growths among certain pairs (e.g., "rocky" through "ussuri"), as shown in Figure 2.17b.



a) The evolution of archived, modified, and non-modified projects across releases

b) The evolution of archived, modified, and non-modified projects over time

Figure 2.18 Evolution of different types of OpenStack projects (a) **across releases** and (b) **over time**

The amount of cross-project changes is related to the load of changes in a given year/release as the more projects modified in each OpenStack release, the more cross-project changes are. Cross-project changes and the number of projects modified in each OpenStack release are highly correlated. As illustrated in Figure 2.18a, OpenStack has been continuously modifying more projects across releases, leading to a higher proportion of cross-project changes. Such correlation is confirmed through the Spearman rank correlation coefficient where the value marks a strong positive correlation ($\rho = 0.86$) for cross-project changes and continuously modified projects. For instance, the number of changed projects has doubled between "essex" (26 projects) and "grizzly" (62 projects) releases, producing 18 and 131 cross-project dependent changes, respectively. Such projects are constantly being changed until recording relative stability in "newton" and "ocata". Then, a slow decay is observed over the recent releases (e.g., "zed" had 540 modified projects). Additionally, one can still observe that archived projects were also a source of such cross-project changes decrease. For instance, Figure 2.18b shows the highest number of archived projects ever across all OpenStack history, recording 501 dead projects in 2020. In the following two years, OpenStack has archived 25 and 7 projects respectively. Consequently, Figure 2.16a reveals a noticeable decrease in the number of cross-project changes from 5,875 in 2020 to 4,335 in 2021. Moreover, cross-project changes and archived projects

between 2019 and 2022 follow similar trends, as demonstrated in Figures 2.16a and 2.18b, respectively.

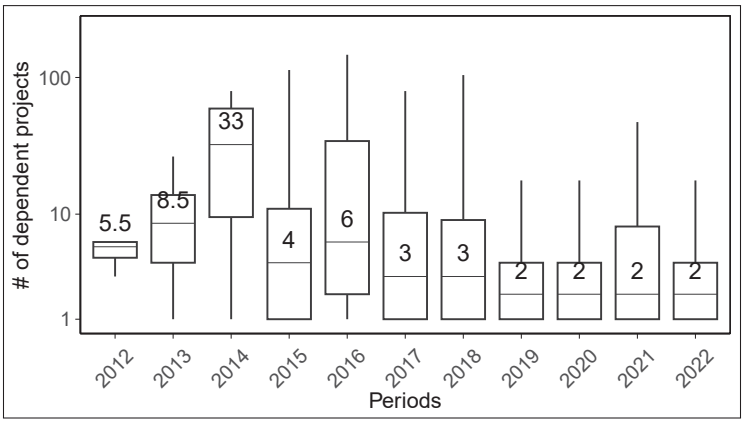


Figure 2.19 Number of projects a given project depends on yearly. The medians are placed above the horizontal median lines

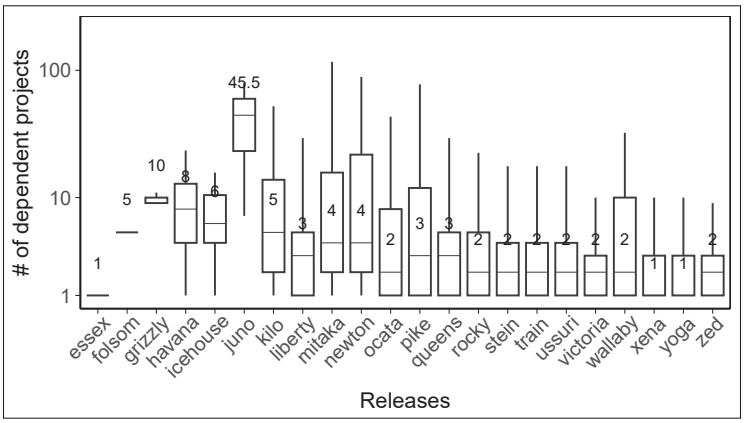


Figure 2.20 Number of projects a given project depends on across releases. The medians are placed above the horizontal median lines

The number of projects that an OpenStack project depends on has been steadily increasing but has remained relatively stable over the last years and releases. Our observations suggest that OpenStack projects tend to rely more on the functionalities provided by other projects, resulting in higher levels of inter-project dependencies, as shown in Figure 2.19. For instance, in 2011, there was no change in one project depending on a change in another project (not shown in the figures). However, three years later, the number of dependent projects for a given project soared up to a median of 33 projects, representing the highest peak of the graph. Over the last

couple of years (2015 to 2022), a project has had a median dependency on two to six other projects, indicating that OpenStack has reduced the potential connections between projects. Nonetheless, we observe that some projects still have a significant number of dependencies, with a maximum of 216 for "openstack-zuul-jobs" in 2018, while "project-config" was the most predominant in the following four stages from 2015 to 2017. Similar patterns were observed across different OpenStack releases, with noticeable stability in the number of dependent projects during the latest releases (a median of two dependent projects), especially from the "rocky" to "wallaby" releases, as depicted in Figure 2.20.

Summary of RQ2

Over time and across releases, there has been a continuous increase in changes that depend on multiple projects and services within OpenStack. Additionally, the number of dependencies decreased after archiving a large number of projects. **Therefore, it is important for OpenStack to be mindful of these interdependencies when issuing new releases. Our results also shed light on the importance of restructuring a project even when it has a multi-component architecture.**

RQ3. How often do cross-component changes end up being abandoned?

Motivation: While RQ1 reveals that cross-project dependencies are common, the goal of this research question is to investigate to what extent such efforts end up being wasted as abandoned changes, similarly to prior work Khatoonabadi, Costa, Abdalkareem & Shihab (2023). We assume that cross-project changes as they involve different teams might be more costly to be wasted as compared to with-component changes or changes that involve just one developer. Understanding the prevalence of such abandoned changes will motivate future studies to better deal with these types of changes.

Approach: Since abandoned changes can be a wasted effort according to prior work Khatoonabadi *et al.* (2023), we wish to quantify to which extent cross-component changes are wasted. To identify abandoned changes, we leverage a tag called status –shown in Figure 2.3 on the left hand of the change identifier. Such a tag can typically have one of the

following statuses: *active*, *abandoned*, or *merged*. In this research question, we quantify the number of abandoned changes that are cross-component changes (i.e., have a dependency on a change in another component) according to changes that depend on other changes within the same component and changes without any dependency.

Results: The percentage of cross-project/service abandoned changes is three times the same percentage for within-project/service changes. Regarding the changes, the abandoned cross-project changes (20.85%) are three times greater than changes (7.9%) that depend on others within the same project, yet still lower than ordinary changes (21.9%) that do not depend on any other change. Similarly, we observe that abandoned cross-service changes (25.77%) are roughly three times that of within-service changes (9.71%). Related to the chains of dependencies, we observe that 4.12% (2,497 out of 59,375) of all studied chains of dependencies were found to be completely abandoned. Interestingly, the proportion of abandoned chains involving one project (20.79%) is twice that of chains of changes belonging to multiple projects (10.24%). On the other hand, we noticed that the percentage of abandoned chains related to cross-service dependencies (10.74%) is three times higher compared to within-service-related chains (3.06%).

Such an abandoned chain of dependencies contains at least two changes and a maximum of 35 related changes. For example, 34 changes depending on the change "210380" [20]. By abandoning "210380", all the other 34 changes were abandoned too, which would not be the case for ordinary changes with no dependencies. The change "210380" aimed to move a file (intall_modules.sh) from the root directory into the "tools" directory. However, the developers of this change found out it would trigger a large amount of work, so they eventually dropped it as well as all its 34 related changes, stating the following comment *"Yes, my preference would be to drop this patch unless there is a good reason for it."*.

[20] https://review.opendev.org/c/openstack/puppet-openstack-integration/+/210380

Summary of RQ3

A good percentage of cross-project changes end up being abandoned, leading to a waste of effort on a whole chain of changes. Among the abandoned changes, some are designed to be abandoned for testing a change on other projects. **Hence, OpenStack should accordingly develop new mechanisms to detect abandonment of a cross-project change, so that such time and effort are not wasted. Our results suggest future work to design approaches to prioritizing components to test for a given change.**

RQ4. What are the characteristics of cross-component changes?

Motivation: This research question complements the previous ones through a deep qualitative understanding of the co-evolution among different components, such that practitioners can better plan for their changes in multi-component systems and research can design appropriate approaches to deal with the evolution of multi-component systems. To do so, we aim to conduct two qualitative analyses on (1) why cross-component changes occur and (2) why cross-component changes end up being abandoned. As we observe in RQ1 and RQ2 that cross-component changes are prevalent, we wish to understand the characteristics and the reasons behind these changes. As we also surprisingly observed in RQ3, practitioners abandon cross-component changes and as abandoned changes are wasted efforts according to prior work Khatoonabadi *et al.* (2023), we wish to understand to which extent it is the case for cross-component abandoned changes, whether practitioners need approaches to help them prevent abandoned changes, and why practitioners create a cross-component change that gets abandoned.

Approach: To qualitatively investigate why cross-component changes occur and why cross-component changes end up being abandoned, we select two randomly statistically representative samples (confidence level of 95% and confidence interval of 5%) of, respectively, **381 merged** and **364 abandoned** cross-component changes.

For both analyses, we went through each change (e.g., merged cross-component change for the 1st analysis and abandoned cross-project changes for the 2nd analysis) and investigated the change's title, description, and modified lines. If the relationship between a pair of changes is not

clear enough, we, therefore, read through the discussion messages exchanged during the review process. The authors of this paper individually analyzed the 381 and 364 cases. After analyzing each case, the authors wrote a brief description (i.e., *"Add a text description on two files to warn the significance of libraries declaration order"*) about the reason for a cross-component change or abandoned cross-component change. Afterward, we performed **card sorting** technique (i.e., a commonly used technique when conducting qualitative analysis Spencer (2009)) to classify each change under a given category. For the card sort technique for the 1st analysis (merged cross-component changes), we defined our labels during the card sorting technique, while in the 2nd analysis, we adopted the labels of Wang, Xia, Lo & Li (2019) with a few differences. We used their categories since they studied abandoned changes in different projects, including OpenStack. As we observe during the labeling of some categories that can be further split into subcategories, we define the categories and sub-categories shown in Table 2.3 and Table 2.4.

Table 2.3 Classification of abandoned changes

Category	**Definition**
Test	Changes that were designed for testing other changes and marked as not to be merged/reviewed.
Duplicate	Changes that were identical to other changes.
Lack of Feedback	Changes that did not have any activity for a long time.
Superfluous	Changes that were not needed at all.
Give Up	Developer gave up on finishing the change because it is time-consuming or laborious.
New Work	Developer of the change moved to another change.
Transfer	Changes that were transferred from one place to another.
Merge Conflict	Changes that were abandoned due to merge conflicts.
Contributor Operation	Changes that were abandoned because of the contributor's operation.
Complicated Change	Changes that were hard to resolve or need to be decomposed into smaller changes.
Incomplete/Wrong Fix	Changes that were incorrect/incomplete
Other	Changes that did not belong to any of the aforementioned categories.

Table 2.4 Subcategories of *Test*, *Duplicate*, *Transfer*, and *Contributor Operation*

Category	**Subcategory**	**Definition**
Duplicate	Already Done	The problem was solved in another change.
	Integrated	Change was part of/integrated into another change.
	Suboptimal solution	Better solutions were proposed in other changes.
Transfer	Project	Change was transferred from one project to another.
	Branch	Change was transferred from one branch to another.
Operation	Wrong Project	Developer pushed the change into the wrong project.
	Wrong Author	The author of the change was wrongly assigned.

Table 2.5 The main reasons for relationships between two OpenStack cross-project dependent changes

Reason	Description	Example	Count (%)
Configuration	Two dependent changes involve configuration tasks by adding, updating, and integrating config files, parameters, and gate jobs (i.e., Zuul pipeline) and using them in the second project, or even deleting the same config options from both projects.	The "499438" change modifies a config related file ("scenario001-multinode.yaml") which was used to execute a task in the "490376" dependent change.	**132 (34.64%)**
Dependency	Dependencies also involve upgrading (i.e., upgrade a library–in the "requirements" project–to enable the use of newer functionalities in the 2nd project), removing (i.e., fixing a bug by removing an extension, or remove a no longer needed dependency), initializing an extension or library (i.e., add configuration options related to a project in the two projects, such as project name, version, and description...etc)	The "544025" change required a specific functionality only exists in the newer version of "gabbi" (i.e., 1.42.1) which forced the "requirements" project ("563173" change) to upgrade it globally.	73 (19.16%)
Code enhancement	Code improvement involve modification to code chunks and properties that are shared between two changes. Such quality tasks include fixing bugs, removing unsupported features...etc.	"473158" removes unnecessary code (i.e., control statement) where its business logic was implemented in the "473081" change.	48 (12.59%)
New features	New features often involve the implementation of API endpoints in the 1st project, hence consuming them in the 2nd project.	The "288392" exposes a set of API(s) while the "254280" dependent change implements its tests.	18 (4.78%)
Docs	This category is designed to provide documentation and more pieces of information to the end user by adding the same content in both dependent projects.	"118694" and "118729", which are cross-project dependent, stated the following message: "warn against sorting requirements", to warn users on the importance of the libraries order in which they are declared.	12 (3.14%)
Renaming	This category consists of renaming either the same files/source code in the dependent changes or property in the 1st change and exploiting it in the 2nd change.	The "620303" change renamed a config option (i.e., "tripleo-ci-centos-7-scenario001-standalone"), while the "619508" dependent change employed it in different places of changed files.	10 (2.62%)
Tests	This type of dependency implements tests-related modules and methods in the 1st change while consuming them in 2nd change.	The "281369" added a new module ("PuppetOpenstackSpecHelpers") dedicated for testing purposes, whereas the "281376" change imported it within the changed files.	6 (1.57%)
Refactoring	Such a reason include refactoring the code across the dependent changes.	The "644929" and "659850" changes perform a wide range of refactoring such as reaming files and as the title of the later change indicates.	6 (1.57%)

Continued on next page

Table 2.5 – *Continued from previous page*

Reason	Description	Example	Count (%)
Moving resources	This category involves moving files or code chunks between dependent projects.	A file called "test-install-haproxy.yml" was deleted from the "475040" change and added to the "476865" change.	5 (1.31%)
Others	Represent the instances that were not easy to identify the reason for their relationships.	N/A	71 (18.63%)

Results: The most frequent reason for cross-component changes is *configuration*, followed by *dependency management*, and *code enhancement*. Table 2.5 shows the primary reasons derived from the 381 qualitatively analyzed OpenStack cross-project dependencies. We observe that relationships between two dependent changes designed for configuration purposes are the most prevalent, accounting for 34.64%. For example, changes 536579 and 536588 involve the removal of the same configuration options (i.e., "tripleo-ovb-check" and "tripleo-ovb-experimental") from a configuration file ("layout.yaml") in both projects. *Dependency management* is the second most common reason for such a relationship, constituting 19.16% of the observed cross-project dependent changes. This type of relationship often involves tasks such as upgrading, removing, or initializing internal or third-party libraries. For instance, the introduction of a new project called "os-service-types" into the OpenStack ecosystem led the *487112* and *487177* dependent changes to, respectively, adding a file containing the project description, and adding the version of the same project to the global dependency management file ("global-requirements.txt"). Additionally, *code enhancement* is another widespread type of cross-project change representing 12.59%. The main purpose of this type involves improving existing code scattered across dependent changes. We observe a wide variety of rationales that are less frequent than the aforementioned categories, including *new features*, *docs*, *renaming*, *tests*, *refactoring*, and *moving resources* where each corresponds respectively to 4.78%, 4.14%, 2.62%, 1.57%, 1.57%, and 1.31% as shown in Table 2.5. However, 18.63% of the manually studied cross-project dependencies were classified into the others' category, making it difficult to determine the exact reason for the relationship between them.

Table 2.6 Various categories of abandoning OpenStack cross-project changes

Category	**Example**	**Count (%)**
Test	Change *843091* was designed to test the Zuul job, hence mentioning the following description ("DNM: Testing tempest pin on stable/ussuri").	**140 (38.45%)**
Duplicate		50 (15.42%)
Already Done	Change *238222* upgraded the version of the "keystonemiddleware" library. After a while, the developer discovered that it had already been done in another repository leading to its abandonment.	29 (8.00%)
Integrated	Change *597291* was abandoned because it has been integrated into another change. Thus, the developer clearly states "*Moved to [600858]*".	14 (3.84%)
Suboptimal Solution	Change *864803* shows an instance of a developer abandoning it because a better solution was provided. The developer concluded such abandonment with the following message "*[864838] is the right one it seems*".	13 (3.58%)
Lack of Feedback	Change *250648* was abandoned by a core reviewer due to inactivity for more than three years. The core reviewer finished the change with "*Abandoning due to inactivity [...]*".	46 (12.63%)
Superfluous	Change *630736* took more than six weeks of review time. Right after, the developer found that such a change was irrelevant, which eventually ended up being abandoned and left the following comment "*Unnecessary changes committed*".	40 (11.00%)
Give Up	Change *445681* was abandoned because its corresponding developer had no sufficient resources to keep working on it. As such, the change was concluded with the following message "*I haven't found resources to continue the efforts on it but I'll probably restore this work later*".	11 (3.02%)
New Work	Change *471479* was abandoned because the developer and reviewers agreed to move forward with another approach. Consequently, the developer concluded this "*After conversation on IRC, i will abandon the usage of delorean-repo and start using DIB_YUM_REPO_CONF*".	11 (3.02%)
Transfer		10 (2.75%)
Project	Change *310042* was transferred from *openstack/ironic* project to *x/ironic-staging-drivers* as "*moved to ironic_staging_drivers [325974]*" depicts.	8 (2.20%)
Branch	Change *810078* was transferred from *master* branch to *feature/r1* as also stated in the last comment "*Cherry-picked to feature/r1 as : [811951]*".	2 (0.55%)
Merge Conflict	Change *772099* was abandoned because there were merge conflicts between certain files as commented by the developer "*The following files contain Git conflicts: [ovsdb_monitor.py] and [requirements.txt]*".	5 (1.37%)
Contributor Operation		4 (1.09%)
Wrong Project	Change *299851* was submitted by its developer to the wrong project which resulted in its abandonment as follows "*Wrong project*".	3 (0.82%)
Wrong Author	Change *563970* was assigned the wrong author as this comment "*wrong author*" exhibits.	1 (0.27%)
Complicated Change	Change *100791* expresses the need to separate the concerns into multiple changes as one of the reviewers proposed the following "*Agree with [...] that the namespace wrapper should be in a separate change.*". After that, the same reviewer said that the current state of the change would be poorly developed as follows "*Try not to let random changes creep into your patch.*"	3 (0.82%)
Incomplete/Wrong Fix	Change *676814* was imperfect because the developer has set out a new direction to take as revealed in "*taking this change set another direction.*".	3 (0.82%)

Continued on next page

Table 2.6 – *Continued from previous page*

Category	Example	Count (%)
Other	N/A	35 (9.61%)

OpenStack developers abandon cross-project changes for many different reasons, among which *Test* is the most widespread purpose for such abandonment. From Table 8, we observe that changes that were abandoned for *Testing* purposes are the most dominant (38.45%), followed by *Duplicate* changes (15.42%), then changes with a *Lack of Feedback* (12.63%). Additionally, changes that were abandoned, but not needed at all, are classified under the *Superfluous* category and are also common (11%). Interestingly, we notice a similar percentage of abandoned changes for which their respective developers gave up on finishing and continued the work on other changes which are, respectively, referred to as the *Give Up* and *New Work* categories. The other remaining abandonment categories, namely *Transfer*, *Merge Conflict*, *Contributor Operation*, *Complicated Change*, and *Incomplete/Wrong Fix* have percentage distribution values in the range of 0.82% to 2.75%. The *Other* category involves abandoned changes that did not belong to any of the previously described categories. While Wang *et al.* (2019) found that the most common reason for abandoning changes is duplicate changes; we find that it is the second category for cross-component changes. Duplicate changes can be seen as wasted resources that one needs to prevent. However, for cross-component changes, we observe that the most important category is for testing purposes, which one does not need to prevent. Instead, developers explicitly create a change that is meant to be abandoned since it is just for testing purposes. Furthermore, Wang *et al.* (2019) found that the testing category accounts for just 4.39% of their studied changes. We also think that there is a lack of synchronization among different teams, which leads to a Duplicate category of abandoned changes. As shown in the example of Table 2.6, a change was already implemented in a different project. Such wasted resources are at the level of different components, which can be more difficult to identify on a large scale of components rather than just already implemented changes within the same component.

Summary of RQ4

Based on our qualitative analysis, we identified a variety of reasons for a pair of cross-component dependent changes, among which the Configuration-related changes are the most frequent purpose (34.64%). Furthermore, we explored various categories for which OpenStack developers decide to abandon a given cross-component change. Thus, our findings indicate that changes that were abandoned for testing purposes are the most prevalent (38.45%), followed by Duplicate (15.42%), and then Lack of activity (12.63%). As a result, **we recommend developers take these findings into account when making a change across components in the future. We also recommend that researchers design approaches to better handle cross-project changes.**

RQ5. Who is responsible for addressing the cross-project/service changes?

Motivation: The goal of this research question is to examine the resources in terms of developers and reviewers required for cross-project/service-dependent changes. A pair of two dependent changes made by different developers requires different team members to engage and discuss their changes, whereas two dependent changes in different components made by the same developer require that developer to be familiar with both projects. Thus, we determine whether dependent changes are developed by different developers. If not, whether a developer responsible for a cross-project dependency has the same maturity over both projects or whether they end up working on projects they are not familiar with. Similarly, we evaluate to which extent OpenStack reviewers need to collaborate with other team members to review changes that are cross-project/service-dependent.

Approach: To determine whether dependent changes are made by the same or different developers, we leverage dependencies involving different projects/services that are obtained following the approach discussed in the methodology section 2.4. We identify the developers of cross-project/service-dependent changes and quantify the number of dependent pairs that share the same developer vs. different developers.

For changes carried out by the same developer, we evaluate the expertise of that developer in both projects to see whether developers end up contributing to projects they are less familiar with. Particularly, we would like to know if a developer changing two projects is the same developer, an expert in both projects, or if he/she is an expert in one project and end up working on a project he/she is less familiar with. To this end, we measure the developers' maturity in the involved projects as the sum of changed lines of code that a developer made in a project over the total number of changed lines of code in the same project (at the creation time of a studied cross-component change), according to the same approach used by Li *et al.* (2023). For example, for two dependent changes A and B that are made in different projects "X" and "Y" by the same developer. We count the number of changed lines of code I that the developer had in the "X" project before and including the change A and the number of changed lines of code J that he/she had in the "Y" project before and including the change B. After that, we identify the min and max values, as respectively being, the minimum and maximum number of changed lines of code a developer had in the pair of changes in a studied cross-project dependency. Hence, we compare the maturity of OpenStack developers on projects they are more familiar with over projects they are less familiar with of a dependency as $max(I, J)/min(I, J)$. Consequently, this allows us to know how much experience a developer has in the project with more familiarity compared to the experience he/she has in the project which is less experienced on. Note that we also report $max(I, J)$ and $min(I, J)$ separately to quantify the experience in both projects of a dependency.

Moreover, we study how different cross-project dependent changes are made by different developers vs. those made by the same developer in terms of the complexity of changes and review efforts which are measured using the three following metrics:

- **Code churn:** We use this metric similarly to a prior work Jiang & Adams (2015), to better understand the number of changed lines of code of two dependent changes, and see how significant it is among those made by different developers and dependent changes made by the same developer. To do so, code churn is identified by the number of changed lines in a change (i.e., deleted and newly added lines of code).

- **Duration:** We also compare merge duration between dependent changes made by the same vs. different developers. We define the duration as the time between the creation of change and the time that change is merged.
- **Revision:** Similarly, we count and compare the number of revisions of each two dependent changes when they are developed by the same developer vs. different developers.

We leverage these previous metrics to compare pairs of dependent **C**hanges made by the **S**ame developers (CS_1, CS_2) vs. pairs of dependent **C**hanges made by **D**ifferent developers (CD_1, CD_2). We define CS_1 and CD_1 as the source of a dependency and CS_2, CD_2 as the target of a dependency. The target change is the one with the "Depends-On" tag, and the other change is the source when two dependent changes are linked with the "Depends-On" tag. For the "Needed-By" tag, we identify changes containing that tag as the source, whereas the other one is marked as the target. Although the "Related-Bug" and "Change-id" do not mention such meaningful relation between two changes in the same way as the two previous tags. Hence, we define the source as the change created first, while the change created later is considered the target.

Similarly, we assess the level of cooperation among OpenStack reviewers who address cross-project and cross-service dependencies. To accomplish this, we first identify the reviewers involved in each dependent change. Then, we compute the percentage of common reviewers among the studied cross-project/service-dependent changes for each pair of dependent changes. Note that we do not consider dependent changes that are published by user bots, and similarly to the previous RQ, we consider only merged changes.

Table 2.7 Number of cross-project/service dependencies addressed by the same and different developers

	Same developers	**Different developers**
Cross-project	35,556	11,572
Cross-service	5,911	2,944

Results: 24.55% and 33.24% of our studied cross-project and cross-service dependencies, respectively, were developed by different developers. Table 2.7 shows the number of cross-project/service dependencies that are made by the same vs. different developers. Although

24.55% of the studied cross-project dependencies were tackled by distinct developers, we observe that 71.63% of these dependencies have at least one developer participating in the review process of the second change, whereas both developers of 27.44% of the pairs of cross-project dependencies were involved in reviewing the other projects changes. For example, the dependent changes *841489* (in "Requirements" project) and *841712* (in "Neutron" project) were, respectively, developed by R. A. and S. K., whereas R. A. participated in the review of *841712* and S. K. in the review of *841489*. Similar observations were perceived for cross-service dependencies. By focusing on dependent changes whose developers participate in reviewing the other pair of projects, 77.06% of these dependent changes involve developers voting during the review process, while 21.12% include developers providing specific comments. However, we do not observe any case where a developer participates in the review of the second project by uploading a new version of a change (or revision). Note that there are also other kinds in which a reviewer contributes to the change review process. For example, we notice from the two dependent changes *831935* and *844680*, that the author of the *831935* change was added as a reviewer in *844680*.

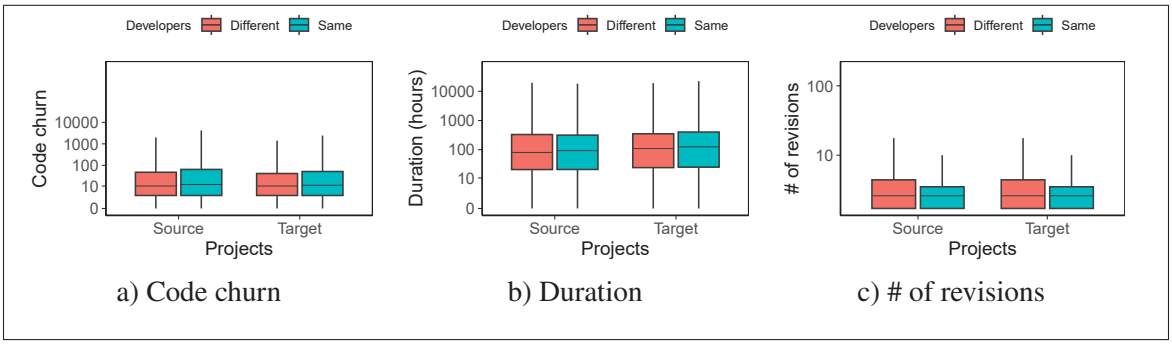


Figure 2.21 Comparison of (a) **code churn**, (b) **duration**, and (c) **number of revisions** between same and different developers addressing "Source" (1st change) and "Target" (2nd change) of a dependency

Although dependent changes made by the same or different developers have the same code churn, the 2nd change of a dependency is 1.14 (125.8/109.92) times slower when the pair of changes are made by the same developers. We observe that two cross-project dependent changes ("Source" and "Target") that are made by the same developer have, respectively, a median code churn of

Table 2.8 Wilcoxon rank sum test for code churn, duration, and the number of revisions, wrt., source, and target. Note that significant $p - values$ (i.e., $p - value \ll 0.05$) are displayed in bold text format

Metric	**Source**	**Target**
Code churn	$\mathbf{3.895 \times 10^{-6}}$	0.1948
Duration	0.6749	$\mathbf{6.177 \times 10^{-8}}$
# of revisions	$< \mathbf{2.2 \times 10^{-16}}$	$< \mathbf{2.2 \times 10^{-16}}$

12 and 11, whereas dependent changes that are made by different developers have a median code churn 10 in both changes, as shown in Figure 2.21a. Furthermore, we notice that the median duration of a review change of two dependent changes made by the same developers is greater than that of the different developers, accounting for 93.05 and 125.8 hours, and 80.5 and 109.92 hours, respectively, as shown in Figure 2.21b. For example, the target (*531138*) of the following dependent changes (*531196* and *531138*), which are made by the same developer, was merged in 125.04 hours (5.2 days). Moreover, we observe that the median number of revisions worked out in the review process of the "Source" and "Target" changes are the same for the same and different developers, as shown in Figure 2.21c. Table 2.8 shows how such metrics are statistically significantly different from each other between the same and different developers.

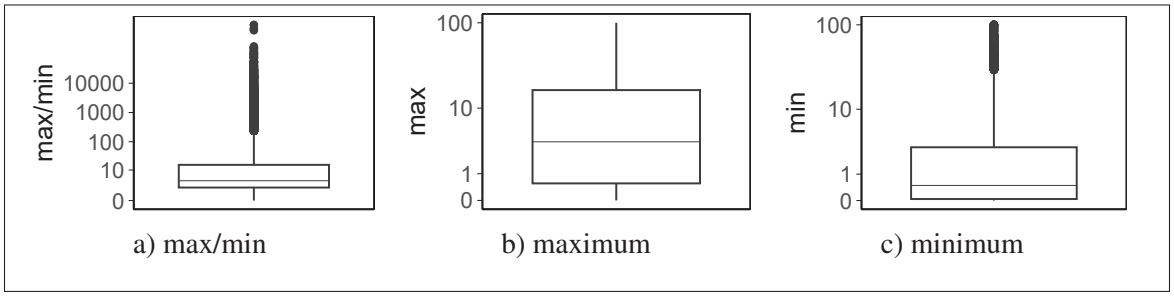
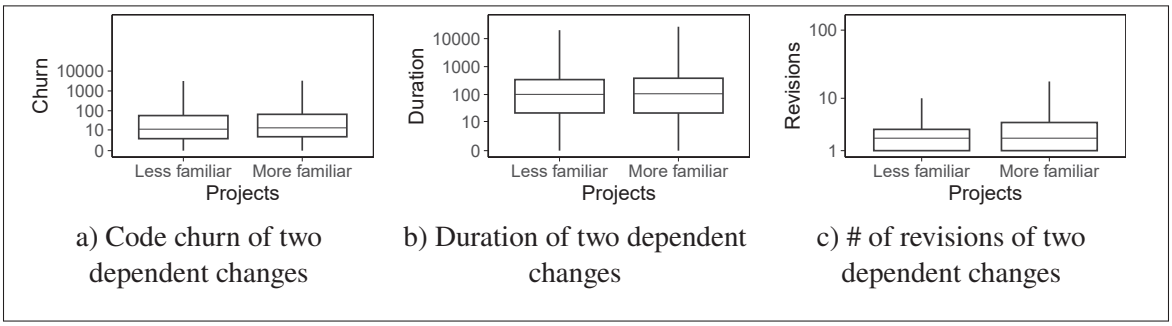


Figure 2.22 Distribution of (a) **experience** of a developer denoted as the division of maximum and minimum values of a developer's maturity in a pair of projects, (b) **maximum** and (c) **minimum** values of developer maturity in one project

Developers addressing both cross-project and cross-service dependent changes, have a median of three times and eight times more experience in one project over the other one at the time of the cross-project and cross-service change, respectively. According to

Figure 2.22a, we notice that developers have substantial experience in one project compared to the other one, suggesting that they end up working on a project they are not as familiar with as the first project. Developers can have a median of 3.58% experience in the project they are more familiar with, while 0.4805% for projects they are less familiar with, as shown in Figures 2.22b and 2.22c, respectively. Note that the observed differences (i.e., between min(I, J) and max(I, J)) are statistically significantly different (Wilcoxon rank sum test, $p\text{-}value = 2.2 \times 10^{-16} \ll 0.05$). Interestingly, 20.17% of studied cross-project dependencies reveal that such developers made no changes in at least one of the projects' pairs. A possible explanation for our observation is that developers are responsible for testing the impact of their changes on other components and fixing any issues in other components accordingly.



a) Code churn of two dependent changes

b) Duration of two dependent changes

c) # of revisions of two dependent changes

Figure 2.23 (a) **code churn**, (b) **duration**, and (c) **number of revisions** of two dependent changes developed by the same developer broken down into less and more familiar projects

The median code churn and duration are relatively higher for projects a developer is familiar with, while the median number of revisions is similar among less and more familiar projects. We observe that projects a developer has more experience highlight a median code churn of 13 compared to the one he/she is less experienced in (a median of 11), as shown in Figure 2.23a. Moreover, we notice, from Figure 2.23b, that it takes a median of 106.42 hours to get changes merged of the project a developer is familiar with compared to 101.2 hours to the project he/she is less familiar with. Wilcoxon rank test shows that the time it takes to get changes merged into the project with less familiarity vs. those with more familiarity is statistically significantly different (p-value = 2.2e-16 « 0.05). However, we observe that the median number of revisions

(a median of 2) is equal across projects in which developers are less and more experienced, as depicted in Figure 2.23c.

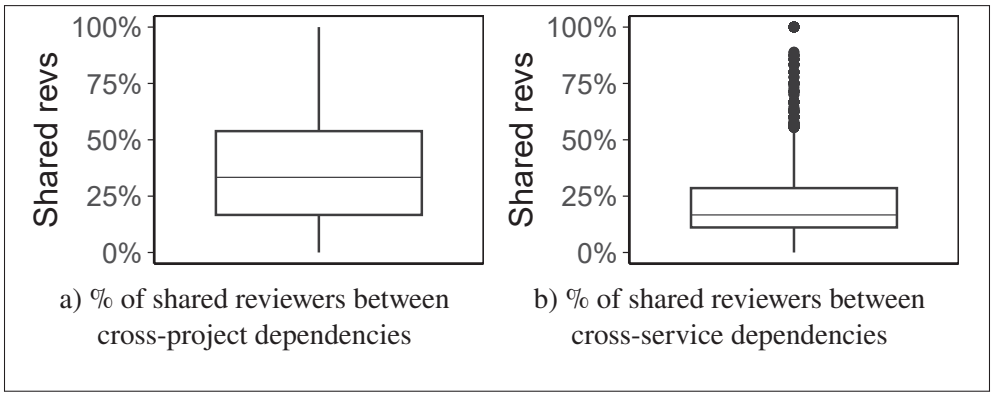


Figure 2.24 Percentage of shared reviewers between (a) **cross-project** and (b) **cross-service** dependencies

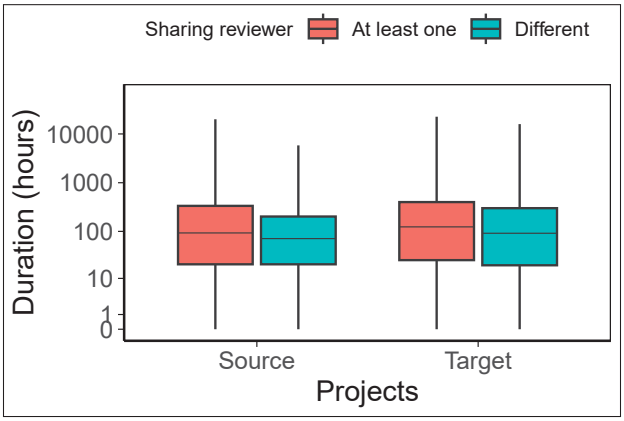


Figure 2.25 Comparison of merge duration between two cross-project dependent changes sharing no reviewers and those sharing at least one reviewer

8.65% and 12.03% of our investigated cross-project and cross-service dependencies, respectively, were not reviewed by any common reviewer. While the examined cross-project dependencies share a median of 33.33% of their respective reviewers, we observe that 8.65% of these dependencies did not reveal any common reviewer. Similarly for services, even though dependencies involving different services share a median of 16.66% of their corresponding reviewers, 12.03% of these dependencies are not carried out by any shared reviewer, as depicted in Figure 2.24. Furthermore, we observe that 49.05% of such cross-project dependencies sharing no

reviewer were created in different releases. Interestingly, we observe that cross-project-dependent changes are quickly merged when they share no common reviewer vs. sharing developers. For instance, two dependent changes, sharing at least one reviewer, are merged after a median of 92.81 hours (source) and 123.83 hours (target); however, when two dependent changes do not share any common reviewer, they are merged after a median of 70.81 hours (source) and 91.39 hours (target) as shown in Figure 2.25. Additionally, we observe that the merge duration for dependent changes sharing vs. not sharing a reviewer is statistically significant (Wilcoxon rank sum test, source: $p\text{-}value = 3.131 \times 10^{-14} \ll 0.05$; target: $p\text{-}value = 2.2 \times 10^{-16} \ll 0.05$).

Summary of RQ5

While nearly a quarter of cross-project dependencies are made by different developers, these developers frequently take part in the review process of the second change. However, we notice that when developers are involved in both projects of a dependency, they tend to make more significant modifications in the project they are less experienced with, as opposed to the one they are familiar with. **Our results suggest that managers choose the right contributor for cross-project change, as it can have an impact on the time to develop two dependent changes.**

2.6 Implications

In this section, we recommend a set of implications from our results to practitioners and researchers for better maintenance of multi-component systems.

2.6.1 Implications for Practitioners:

We recommend managers do not neglect in their planning and resource estimation that a change might require modifying multiple components that are managed by different teams and leverage our qualitative findings for hints on the types of changes that might involve multiple components. The changes that require modifying multiple components can be related to the complexity of the releases, the types of projects, and types of changes that we identified

in our qualitative study. In fact, we observe that different projects are managed by different teams, suggesting that these components are less likely to be maintained by different teams in an independent way (PQ). However, as we observe in RQ1, cross-component-changes represent a large number of changes and such a type of dependent change occurs in all of our studied releases as shown in RQ2. We also want to shed light that a change might trigger a whole chain of changes, while such a chain implies just two components, a chain can have a larger number of components (up to 224 components). At the same time, we observe that a good number of cross-project changes just end up being abandoned (RQ3). Thus, our results suggest managers not neglect the fact that a change might imply other changes in other components with a good chance (20.85%) might end up being abandoned.

We also want to shed light on the fact that the more complex a release is, the higher the chance for cross-project changes (RQ2). Leveraging our qualitatively obtained reasons for cross-component changes can also help practitioners identify changes that are more likely to trigger changes in other components. For example, upgrading the configuration is likely to introduce changes in other components. Thus, practitioners need to consider more resources to implement such a type of change. Hence, this is aligned with our prior work Sayagh & Adams (2015); Sayagh *et al.* (2017) in which we found that cross-component configuration errors are common and difficult to identify.

Our results suggest managers consider that certain types of projects are more likely to be co-changed, hence planning more time and resources for the modification of these projects. For instance, we observe that a larger portion of cross-project changes is related to configuration (RQ4). Besides, even after refactoring OpenStack by archiving multiple projects, certain projects were more likely to be part of cross-project changes such as openstack-zuul-jobs and project-config (RQ2), which are for the DevOps pipeline and configuration. In fact, it is interesting to note that OpenStack tried to have ownership over its dependencies by moving the declaration of dependencies to the Requirement project, while such a move created more cross-project changes as an upgrade to a dependency needs to go through that requirement project. Each upgrade might also create maintenance overhead in other components and increase

the chance of cross-project changes (as shown in the large example of cross-project changes in RQ1). We suggest practitioners follow our mining software repositories approach to identify the components that are more likely to be involved in cross-component changes and especially suggest practitioners leverage tags that indicate cross-component dependent changes such as the "depends-on" tag.

To minimize the amount of cross-component changes, we suggest practitioners refactor the architecture of their system even when it is a multi-component system. For instance, having a modular architecture might not be a lifelong solution, as such we recommend practitioners consider restructuring their components with a large number of cross-project changes. For instance, we observe that the number of cross-component changes was increasing over different releases to reach a peak at the version Rocky and Stein, while such a number continuously decreased after archiving a large number of projects in the Victoria release. That large refactoring of projects also reduced the number of projects a given project depends on.

Our results suggest practitioners choose the right resources to develop cross-project dependent changes. In fact, we observe that when two dependent changes are made by the same developer, that developer ends up working on a project that he/she is less familiar with. As RQ5 indicates, a developer is three times more experienced in one project (less familiarity) than the other one (more familiarity). When the pair of dependent changes are made by different developers, the developer of the first change participates in the review process of the second change. We also recommend managers consider the same developer for both changes when possible, as dependent changes made by the same developers are more likely to be quickly merged compared to dependent changes made by different developers (RQ5).

2.6.2 Implications for Researchers:

Our results suggest future studies recommend components that are likely to co-change with a given component. As we found in RQ1 and RQ2, the cross-component changes are prevalent and increasingly evolving over time, whereas each of the components is maintained

by a different team, as confirmed by our PQ. To assist practitioners in the planning of their changes, we suggest future studies develop approaches that leverage different techniques, such as machine learning and different metrics, including the purpose of changes to predict what other components one needs to change.

Our results suggest researchers develop solutions that recommend which other components to test given one component change. For instance, we observe in RQ2 that a good number of abandoned cross-project changes are meant to be abandoned as they were for testing the impact of a change in a given component on other components. We also observe that the majority (38.45%) of qualitatively analyzed abandoned cross-project changes are related to the Test category. Thus, we recommend researchers propose models to predict what components to test when changing a given component. We believe that such recommendation systems can also help the integration and deployment pipeline, as instead of testing the whole system, one might prioritize testing components that are more likely to be impacted by changing a given component.

Our results suggest researchers develop approaches that recommend when an architecture refactoring is required to minimize the cross-component maintenance dependencies. In fact, as we observe that the number of co-changes increases up to the time when OpenStack archived a large number of projects, we recommend future studies to develop approaches that can suggest when a project needs to be restructured. For instance, most of the studies (as discussed in related work 2.3) focus on migrating a monolithic system to microservices, while migration can still occur afterward. In practice, developers should be notified when there is an unexpected number of co-changes involving a given project. A large pool of prior work proposed various coupling measurements between components of a software system including microservice-level coupling Zhong *et al.* (2023) and organizational coupling Li *et al.* (2023). Such a coupling metric can be used as an indicator for refactoring needs.

Our results suggest researchers investigate the impact of different components' refactoring strategies on the co-evolution of the components of a multi-component system. We find that

the number of cross-component changes increases over different releases up to a certain time in which OpenStack archived a large number of projects. From then, the number of cross-component changes decreased over time. While we cannot directly conclude that reducing the number of components will impact the number of cross-component changes, we suggest future studies to investigate the impact of different projects' refactoring techniques on the number of cross-component changes. Among the refactoring techniques we suggest is the removal of small projects and the combination of similar projects or projects that frequently co-change. One way to achieve this is to leverage different clustering algorithms such as KNN and determine how related projects can be combined according to their characteristics, which we plan to investigate in future work.

Our results suggest researchers extend existing approaches related to wasting efforts due to code duplication or abandoned changes to the level of cross-component systems changes. Similarly to Wang *et al.* (2019), we observe that duplicate changes are a common reason for cross-component abandoned changes (15.42%). Yet, we observe cases in which the duplication itself is cross-component. For example, we observe cases in which a change was abandoned since it was already implemented in another component. Hordijk, Ponisio & Wieringa (2009) stated that source code duplication is a major factor impacting software quality. Hence, we suggest future studies on code duplication to extend their work to the cross-component level. We also suggest future studies to re-evaluate the model of Khatoonabadi *et al.* (2023) in the identification of cross-component abandoned changes and evaluate whether the model is as expected on these cross-component changes compared to single-component duplicated changes.

2.7 Threats to Validity

2.7.1 External Validity

Our study and a large number of previous studies AlOmar, Chouchen, Mkaouer & Ouni (2022); Foundjem *et al.* (2022a); Bessghaier *et al.* (2023); Wang, Thongtanunam, Kula & Matsumoto (2023) that focused on OpenStack share the same threat to validity related to the generalizability

of our results to other multi-component software systems. We focus on OpenStack since it is a well-known project, studied by a large body of research, has a large number of projects that better represent complex multi-component systems, and especially has a well-defined way of declaring dependencies between changes that allows us to systematically identify cross-project and cross-service changes. We also recommend future work and practitioners replicate our study on their multi-component systems to better understand the co-evolution of their components and better manage such an evolution.

2.7.2 Internal Validity

An internal threat to validity is related to our manual analysis. Our manual analysis can be biased with the selected sample of cross-project/service changes. To mitigate such a risk, we select a random sample. Also, studying more cross-project changes can reveal more types of cross-project changes. We encourage future studies to replicate our manual analysis on another sample and other projects to identify more possible reasons for cross-project/service-dependent changes.

2.8 Conclusion

Modern software systems are composed of multiple components (aka., multi-component systems) that are expected to evolve independently from each other for better flexibility and agility. However, previous studies found that reaching a completely independent evolution is difficult to achieve. Yet, we observe from our empirical study on OpenStack that even if the components are maintained by different teams (i.e., the concept of ownership), these components depend on each other, leading to cross-project and even cross-service-dependent changes. Such dependent changes are common, as we observe 52,069 cross-project changes in OpenStack. We also observe that such a number is continuously evolving up to a certain point in which OpenStack decided to archive a large number of projects. With that, the number of cross-project-dependent changes is still relevant. We quantitatively and qualitatively observe from our study that these dependent changes are for different reasons, including configuration-related changes, moving

code to the right location, and testing the impact of a change on other components, whereas developers often create a change for testing purposes on other components then abandon such a change. Furthermore, we observe that both changes of a pair of cross-project changes can be made by the same developer, who ends up working on a project she or he is less familiar with, or the two different changes are by different developers, yet they collaborate on reviewing the second change. Our study has several implications for practitioners and researchers. Among these implications is that managers should always pay attention to cross-project-dependent changes in their planning, and our qualitative analysis provides a hint into when such a cross-project or service change might occur, and when a cross-component change might end up being abandoned. Our results shed light on the importance of restructuring a project to minimize the dependency between different components. For researchers, we recommend building approaches that predict co-changes by taking into account the possible reasons for such change dependencies, when an architecture needs to be restructured, what components need to be tested given a change in a component and recommend the appropriate developer and reviewer for cross-project changes.

Data Availability Statement

The data and corresponding code used to carry out this work are publicly made available in the following GitHub repository: https://github.com/aliarabat/OpenStack.

CHAPTER 3

A RECOMMENDATION SYSTEM FOR PREDICTING DEPENDENCIES AMONG SOFTWARE CHANGES: INSIGHTS FROM AN EMPIRICAL STUDY ON OPENSTACK

Ali Arabat[1] , Mohammed Sayagh[1] , Jameleddine Hassine[2]

[1] Department of Software Engineering and IT, École de Technologie Supérieure,
1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3
[2] Department of Computer Science, Université du Québec à Montréal,
405 Sainte-Catherine, Montréal, Québec, Canada H2L 2C4

Article published in the journal « Empirical Software Engineering » June 2026

3.1 Abstract

As software systems grow in complexity, accurately identifying and managing dependencies among changes becomes increasingly critical. For instance, a change that leverages a function must depend on the change that introduces it. Establishing such dependencies allows CI/CD pipelines to build and orchestrate changes effectively, preventing build failures and incomplete feature deployments. In modern software systems, dependencies often span multiple components across teams, creating challenges for development and deployment. They serve various purposes, from enabling new features to managing configurations, and can even involve traditionally independent changes like documentation updates. To address these challenges, we conducted a preliminary study on dependency management in OpenStack, a large-scale software system. Our study revealed that a substantial portion of software changes in OpenStack over the past 10 years are interdependent. Surprisingly, 51.08% of these dependencies are identified during the code review phase-after a median delay of 5.06 hours-rather than at the time of change creation. Developers often spend a median of 57.12 hours identifying dependencies, searching among a median of 463 other changes. To help developers proactively identify dependencies, we propose a semi-automated approach that leverages two ML models. The first model predicts the likelihood of dependencies among changes, while the second identifies the exact pairs of dependent changes. Our proposed models demonstrate strong performance, achieving average

AUC scores of 88.45% and 96.46%, and Brier scores of 0.201 and 0.243, respectively. We find that our solution is able to find the right dependency for a given change within a list of 10 recommended changes in 67.19% of the evaluated cases. While we compare our solution to an Agentic-AI equipped with regex-based search tools, the machine learning-based approach still outperforms the agentic-AI one, suggesting the need for further studies to investigate how to improve the agentic-AI solution by equipping it with additional tools, prompt engineering techniques, and/or fine-tuning. Furthermore, this study provides actionable recommendations for practitioners, researchers, and tool builders working with large-scale software systems.

3.2 Introduction

Large-scale software systems employ various techniques and solutions to enable parallel development and feature addition without disrupting the development, integration, or deployment pipelines. For instance, the OpenStack[1] system undergoes continuous changes, with a median of 136 changes per day. Similarly, other large-scale systems such as Google Chromium[2] experience a high volume of changes, with new submissions occurring every few minutes.

The vast number of continuously submitted changes to the pipeline must be synchronized and ordered correctly to prevent breaking the integration and deployment pipelines or even introducing incomplete features and bugs into production. This process is facilitated by various tools, such as OpenStack's Zuul[3] CI/CD platform. In this platform, developers must tag their changes when they depend on other changes by using specific tags in the description of their changes. Typical examples of such tags include the "Depends-On" and "Needed-By" tags as illustrated in Figures 3.2 and 3.3. These tags indicate dependencies between changes, ensuring that a failure in an earlier step automatically excludes the dependent change from the integration and deployment pipelines. This approach helps prevent potential bottlenecks, build failures, and even the inadvertent deployment of bugs.

[1] https://www.openstack.org/

[2] https://github.com/chromium/chromium

[3] https://zuul.openstack.org/builds

While a large body of research focuses on the maintenance and evolution of software systems, little attention has been given to the dependencies among software changes. Existing studies primarily examine software dependencies at the file or class level Xia *et al.* (2015), but none specifically investigate dependencies from the perspective of software changes. The closest related work is a recent study by Arabat & Sayagh (2024), which empirically analyzed dependencies among changes across different components. Yet, their focus is on cross-component change relationships, which do not necessarily imply dependencies, such as changes addressing the same bug or sharing the same change ID. Furthermore, their study did not propose any solutions for predicting dependencies among software changes, particularly changes in large-scale software systems.

In this paper, we present the first empirical study on dependencies among software changes by analyzing 49,129 dependent changes in OpenStack. We define dependencies as interrelated changes linked through two key tags: "Depends-On" and "Needed-By" Arabat & Sayagh (2024). Our focus on OpenStack aligns with a significant body of prior research Foundjem, Constantinou, Mens & Adams (2022b); Bessghaier *et al.* (2023); Arabat & Sayagh (2024), including the work of Arabat & Sayagh (2024). We focus on OpenStack because it is a widely used and actively maintained software system, making it an ideal subject for studying the synchronization of concurrent changes. Moreover, to the best of our knowledge, OpenStack is the only system that provides a systematic mechanism for declaring dependencies among changes, offering a unique opportunity for structured analysis.

Our study is conducted in two phases. The first phase focuses on gaining both quantitative and qualitative insights into the dependencies among changes, while also assessing the challenges associated with identifying these dependencies. The second phase introduces two prediction models designed to assist practitioners in effectively identifying dependent changes.

As our work aims to study and predict dependencies among software changes in OpenStack, the key contributions of this research are five-fold as follows:

- We conduct both quantitative and qualitative investigations into the dependencies among OpenStack's software changes, providing insights into the challenges and characteristics of dependency management in large-scale systems.
- We propose a semi-automated approach that leverages two ML models to assist practitioners in identifying dependencies among software changes, offering a practical solution to improve efficiency in dependency detection.
- We propose and evaluate an agentic-AI-based solution to find the right dependency of a change and compare it to our machine learning models.
- To facilitate the replication of our study, we have made the replication package available at the following link: https://github.com/aliarabat/change-predictor.
- We present a comprehensive discussion of the findings, offering actionable recommendations for practitioners, researchers, and tool builders to enhance dependency-based detection tools and further the understanding of dependency management in complex software systems.

The paper is structured as follows. Section 3.3 presents our motivational example. Section 3.4 presents the closest work to our study. Section 3.5 presents our data collection process. Section 3.6 provides our preliminary study on dependent changes. Section 3.7 examined the research questions and an overview of our methodology design. Section 3.8 presents the evaluation results of the proposed models. Section 3.9 provides a detailed implication of this work. Section 3.10 presents our threats to validity before concluding the paper in Section 3.11.

3.3 Motivating Example

To better motivate the need for a mechanism to identify dependencies among software changes using state-of-the-art ML classifiers, it is valuable to examine a real-world scenario where a missed dependency caused significant delays and posed risks to the CI/CD pipeline and deployment process. This example underscores the importance of accurately and timely identifying software change dependencies, which is the primary goal of this paper.

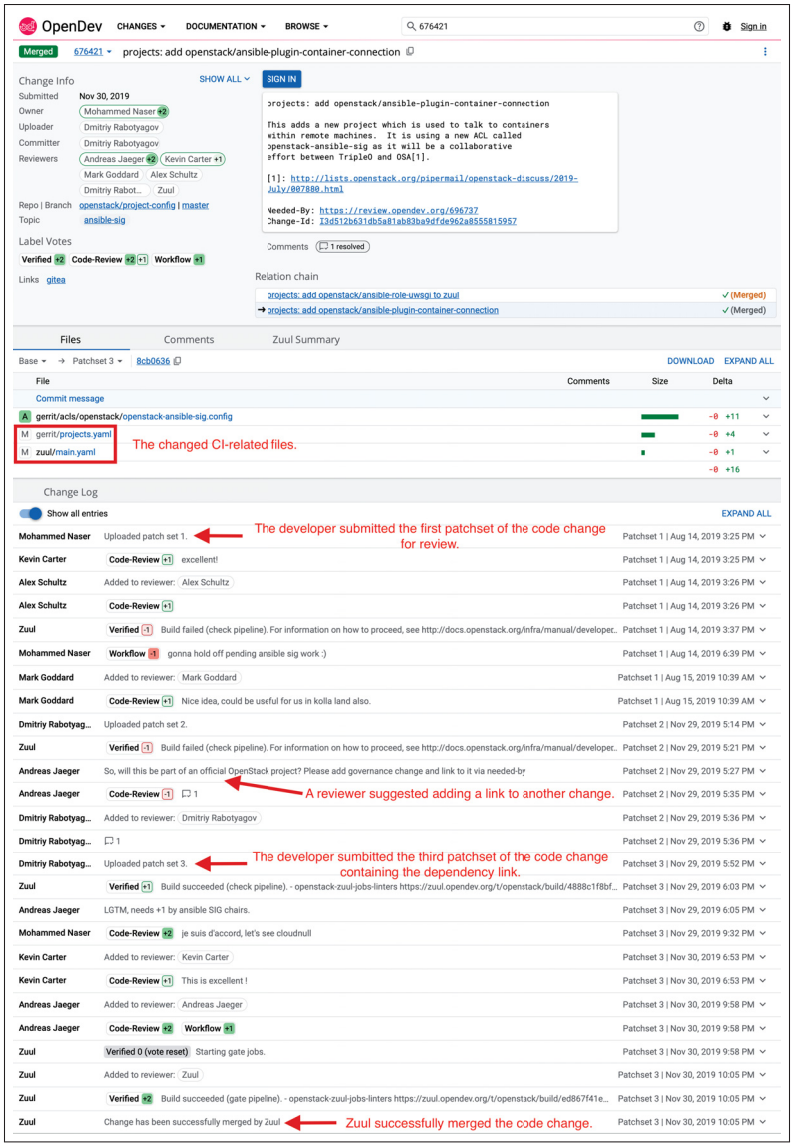


Figure 3.1 A typical example of a code change with a missed dependency link

According to OpenStack, developers are required to add dependency tags such as “Depends-On” or “Needed-By” in the footer of their commit messages when a code change relies on another change in a different branch or even a different repository (i.e., cross-repository dependencies[4]). This ensures that changes are merged in the correct order, and in case a change does not pass the build it gets withdrawn with all of its dependencies. Declaring dependencies is also

[4] https://docs.opendev.org/opendev/infra-manual/latest/developers.html#cross-repository-dependencies

important for developers to merge complete changes that depend on each other and avoid merging incomplete code that if passed the build can theoretically lead to the deployment of an incomplete implementation. Moreover, a change may depend on multiple other changes[5]. In such cases, each corresponding dependent change must be listed separately using its respective link.

In the scenario, illustrated in Figure 3.1, a developer submitted a change, identified as 676421, for review. The purpose of the change was to add metadata for a new OpenStack project, *"ansible-plugin-container-connection"*, to the *"project-config"* repository. The *project-config* repository contains essential configuration files that are crucial for setting up and deploying OpenStack infrastructure [6]. The submitted change involved modifications to two CI tools, *Zuul* and *Gerrit*. However, Zuul failed to deploy such a change.

Since the introduced changes pertained to an official OpenStack project, one of the reviewers identified the issue and proposed adding a link to a dependent change, 696737, using the "Needed-By" dependency tag. The reviewer left the following comment: *"So, will this be part of an official OpenStack project? Please add governance change and link to it via needed-by."*. As a result, it took the developer approximately 107 days (until the submission of the third patchset) to address this feedback and add the missing dependency. Once the missing dependency was added, the same reviewer approved the change with the comment: *"This is excellent!"*. Figure 3.1 highlights an illustrative example of the corresponding code change with a missed dependency link. It also shows the entire review process, from submitting the first patchset of the code change, through adding a dependency link, to successfully merging the code change by Zuul. Note that the commit message, shown in Figure 3.1, belongs to the code change's latest patchset (i.e., third).

This example highlights the critical need for automated tools that can assist developers in identifying dependencies early in the software development lifecycle (SDLC). Delays, such as those described above, not only disrupt CI/CD pipelines but also increase the risk of deploying

[5] https://docs.opendev.org/opendev/infra-manual/latest/developers.html#multiple-changes

[6] https://github.com/openstack/project-config

incomplete or faulty features into production. By leveraging the machine learning models proposed in this paper, practitioners can proactively detect dependencies, ensuring a smoother and more efficient integration and deployment process.

3.4 Related Work

As our study focuses on predicting dependencies among changes in OpenStack, the most closely related research falls into three areas: (1) prediction of dependencies in a software system, (2) linkage detection in code reviews, where linked patches are identified through heuristics such as IDs, URLs, textual content or file location; and (3) investigations of technical dependencies. We review each of these lines of prior work below.

3.4.1 Predicting Dependencies in a Software System

Diaz-Pace *et al.* (2018) proposed an approach to predict dependencies among software modules by applying link prediction techniques to analyze the dynamics of module structures as dependency graphs. Oliva & Gerosa (2012) proposed a method to identify logical dependencies, which are implicit relationships among software artifacts. Xia *et al.* (2015) introduce CroBuild, a cross-project build co-change prediction approach. The proposed approach learns different classifiers using data from a project and evaluates its performance on another project, and achieves an F1-score of up to 0.408. Aryani *et al.* (2011) proposed an approach to predict software dependencies by utilizing coupling in domain-level information. Using such information, their approach can predict up to 68% and 77% of source code and database dependencies, respectively. Yang *et al.* (2021) introduce AID, an approach that evaluates dependency intensity, which is defined as *"how much the status of the callee service influences the caller service"*. Additionally, Zagane & Alenezi (2024) leverage a hybrid approach, consisting of traditional SE techniques and deep learning models, to predict co-changes within software systems.

While prior research has made significant strides in proposing approaches to predict dependencies within software systems, certain limitations remain inherent to this line of work. A key limitation

is the predominant focus on monolithic software systems, which differ substantially from the multi-component systems widely adopted in industrial settings. These multi-component systems introduce additional complexity, as dependencies often span across components maintained by distinct teams. Another limitation is the emphasis on directly related dependencies, such as those involving files, classes, or modules. Modern software systems, however, are often loosely coupled, making the identification of dependencies significantly more challenging. To address these gaps, we propose a semi-automated approach aimed at assisting developers in effectively predicting and managing software change dependencies within the context of OpenStack, a large-scale, multi-component system extensively used in the industry. While our prior work Arabat & Sayagh (2024) examined cross-component changes, this study broadens the scope by considering all types of dependencies, including both within- and cross-component changes.

3.4.2 Understanding and Detecting Linkages in Modern Code Review

Researchers have conducted empirical studies to explore the impact of linked reviews on code review analytics Hirao, McIntosh, Ihara & Matsumoto (2019), the detection of linkages among patches Wang, Kula, Ishio & Matsumoto (2021a); Wang *et al.* (2023), and the practice of sharing links in MCR Wang, Xiao, Thongtanunam, Kula & Matsumoto (2021b).

For example, Hirao *et al.* (2019) examined the impact of linked reviews on code quality across six open-source projects. Their findings revealed that the proportion of linked reviews (i.e., a code review that includes in its description or in the comments a reference to another code review, either by mentioning its ID or providing its URL) ranges between 3% and 25%. Additionally, they identified five categories of review links (i.e., 231,341 links among 1,466,702 code reviews), the categories for which they developed two classifiers. Wang *et al.* (2021a) analyzed the impact of linkages between two patches in code review and proposed two techniques to detect such linkages. Their results indicated that patches with linkages are reviewed faster than patches without any linkages, while the combination of textual and file-location similarity improves the detection of patch linkages. Later, Wang *et al.* (2023) studied the collaboration practices

when a link between two patches is identified (i.e., cross-patch collaboration) in OpenStack. The authors found that patch links, when posted, often serve to share information (211 cases), rather than requesting collaboration, which accounts for only 57. Similarly, Wang *et al.* (2021b) investigated link-sharing practices in code reviews and their purposes. The authors identified 19,268 reviews containing 39,686 links, finding that 93% (OpenStack) and 80% (Qt) of these links were internal references. Additionally, source code and bug reports were frequently cited in reviews. The study also identified seven use cases for link sharing, with contextual reference being the most common.

While prior research has explored the use of patch links and proposed solutions to detect them, our work differs from these efforts by focusing on predicting dependencies among changes, rather than at a granular level such as patches.

3.4.3 Technical Dependencies

Another line of research has explored technical dependencies (i.e., cross-project dependencies) within large-scale software ecosystems Blincoe *et al.* (2019). For example, Blincoe *et al.* (2019) proposed a novel approach called "Reference Coupling" to automatically find technical dependencies based on certain heuristics. Typical examples of heuristics include references/links to other projects mentioned on developer social interaction platforms, like issues and pull requests. The authors demonstrated that their method can uncover technical dependencies that are difficult to find using existing static analysis techniques.

While the proposed method can detect technical dependencies, its effectiveness when combined with source code analysis techniques remains unclear. In contrast, our study focuses on predicting software change dependencies using popular machine learning algorithms, rather than relying on social interactions among developers.

3.5 Data Collection

To analyze the dependencies among OpenStack changes, we begin by downloading OpenStack changes, identifying changes with a dependency, and building pairs of dependent changes. The collected data is then utilized to address each of the research questions (RQ) outlined in the corresponding approach section.

3.5.1 Dataset

OpenStack is a widely adopted open-source cloud infrastructure platform, extensively used by major tech companies across the industry Arabat & Sayagh (2024). It offers a comprehensive suite of services to support the design and development of various cloud-based applications. For example, the "nova" component facilitates the creation of compute instances, such as virtual servers. In this study, we select OpenStack as a case study for three key reasons. First, it has been extensively analyzed in the literature from various perspectives AlOmar *et al.* (2022); Bessghaier *et al.* (2023); Arabat & Sayagh (2024), making it a well-established and well-documented research subject. Second, it is a long-lived, open-source software system consisting of over 1,000 components, providing a rich dataset for in-depth analysis Arabat & Sayagh (2024). Third, it provides a systematic and explicit mechanism for identifying dependencies among code changes Arabat & Sayagh (2024), which aligns with the goals of our research. For our analysis, we leverage software changes submitted to the Gerrit platform, where developers typically propose, review, and merge changes into the OpenStack codebase.

We collect code review changes submitted to OpenDev [7], a code review platform that hosts over 1,300 OpenStack projects from 2011 to 2024. Given the OpenDev server's restriction of retrieving a maximum of 500 changes per request, we acquire the data in batches to ensure comprehensive collection. To address our research questions, we focus solely on merged and abandoned changes, excluding the ones that are still open. Our final dataset consists of 578,826 merged changes and 133,246 abandoned changes. For each change, we gather key attributes,

[7] https://review.opendev.org/

including its subject, description, project, owner, reviewers, creation date, added lines, deleted lines, and the different patch sets–each representing a revision of the change. Each revision consists of the source code and the files that were changed in that revision.

3.5.2 Build OpenStack Dependencies

According to the OpenStack documentation [8], there are two methods to link changes: "Depends-On" and "Needed-By". Therefore, we leverage these tags, which can be defined in a change's description, to identify dependencies among changes.

Figure 3.2 An example of a change with the **Depends-On** tag

- **Depends-On:** Depends-on is to tag that a change depends on another change. We leverage the "Depends-On" tag to identify dependent changes. The change with the tag depends on the description and is considered as *"the target" change*, whereas the change mentioned in the tag is *"the source"* change. Figure 3.2 shows an example of two dependent changes where the current change (aka., target) depends on the *815147* change (aka., source). We apply the following regular expression `Depends-On:\s[a-zA-Z0-9/\.\:\+\-\#]{6,}` to extract the related "Depends-On" change id.
- **Needed-By:** We follow a similar approach to identify dependent changes through the "Needed-By" tag. However, we consider the change whose description has the tag as *"the source"* change, while the mentioned change in the tag is *"the target"* change. We leverage `Needed-By:\s[a-zA-Z0-9/\.\:\+\-\#]{6,}` regular expression to retrieve the right

[8] https://docs.openstack.org/project-team-guide/repository.html

Figure 3.3 An example of a change with the **Needed-By** tag

"Needed-By" change id. The change represented in Figure 3.3 is needed by the change *918979*.

In our study, we define a dependent change as either a source or target change within a dependency relationship. Our final dataset comprises 49,129 dependent changes and 37,897 pairs of dependent changes. The pairs of dependent changes include 27,745 and 10,152 pairs that are made by the same developer and different developers, respectively. These dependencies are categorized based on their identification tags, with 37,152 dependencies recognized through the *Depends-On* tag and 745 dependencies identified using the *Needed-By* tags.

3.6 Preliminary Study

This section investigates the importance of dependent changes in OpenStack by studying their prevalence over time, the challenges associated with identifying them, and gaining qualitative insights into the types of changes that commonly require dependencies. This analysis aims to determine the necessity of solutions to assist practitioners in effectively identifying dependencies. The findings will highlight the need for appropriate tools and mechanisms to streamline the identification process of dependent changes. Our empirical study is guided by the following three preliminary research questions (PRQ):

- **PRQ1**. How prevalent are dependent changes?
- **PRQ2**. How quickly do developers identify dependent changes?
- **PRQ3**. What are the categories of changes with dependencies?

PRQ1. How prevalent are dependent changes?

Motivation: The goal of this research question is to understand whether dependent changes are prevalent and whether such prevalence changes over time, so dependencies among changes require further attention from future studies.

Approach: We study the prevalence of dependent changes over the years by examining their occurrence among merged changes. For each year, we calculate the proportion of changes with dependencies relative to the total number of changes in the merged category. Additionally, we quantify both the number of changes that a given change depends on and the number of changes that depend on it. Identifying dependencies in such changes can be particularly challenging and time-consuming, potentially leading to inefficiencies and resource loss.

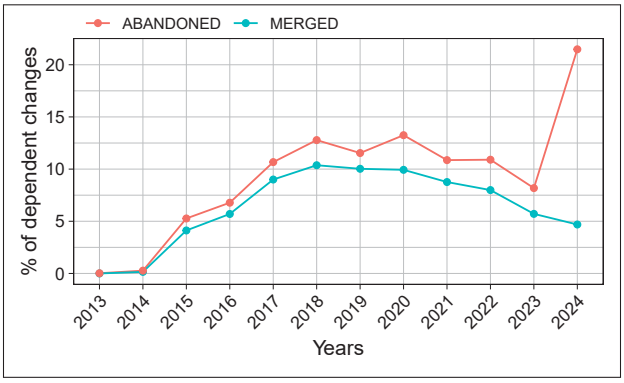


Figure 3.4 The yearly evolution of OpenStack merged and abandoned dependent changes

Results: Since 2015, the proportion of changes with dependencies has ranged between 4% and 11% for merged changes, and between 5% and 22% for abandoned changes, as illustrated in Figure 3.4. In the early stages of OpenStack development, particularly in 2013, the percentage of dependent changes was minimal. However, a notable surge was observed from 2014 onward. Between 2018 and 2022, the trend stabilized with minor fluctuations, reflecting a mature phase characterized by balanced growth. Despite the seemingly low percentages, they account for a substantial total of 38,114 merged changes out of 578,933 changes, respectively. The recent decline in dependent changes can be attributed to the extensive project refactoring and archiving efforts within OpenStack, as identified in recent research Arabat & Sayagh (2024).

Additionally, we notice a spike in abandoned dependent changes in early 2024. These changes are typically abandoned because they are created only for testing purposes, as 87.17% of them contain the *DNM* (i.e., Do Not Merge) keyword, consistent with prior work Arabat & Sayagh (2024). Furthermore, our findings indicate that the number of changes a given change depends on ranges between 1 and 76, with a median of one, while the number of changes that depend on a given change ranges between 1 and 249, with a median of one.

PRQ2. How quickly do developers identify dependent changes?

Motivation: This research question aims to assess the effort required to detect dependencies by measuring the time taken before their identification. Detecting a dependency at the time of change creation suggests an easy and straightforward identification process. In contrast, identifying dependencies during code review or after a build failure indicates that developers may have initially overlooked them, requiring a reviewer's suggestion or a failed build to prompt their addition. It is important to note that undetected dependencies theoretically can potentially introduce bugs, not only during the code review, but also in production, where critical components for a change may remain unmerged and undeployed. However, due to the lack of a systematic approach to identifying such issues in production, our study focuses on dependencies identified during the code review process or as a result of build failures.

Approach: To measure the time required for identifying dependencies among two changes, we first identify in which patch a dependency (i.e., "Dep-ends-On" or "Needed-By") is added. We then compare the date of that patch to the closest creation date of the two dependent changes. That is the minimum among (1) the identification of the dependency and the creation of the first change (aka., source) and (2) the identification of the dependency and the creation of the second change. Such a minimum value is the time taken to identify the dependent change. In particular, the minimum time refers to the earliest point in time when the dependent relationship could have been recognized through explicit tagging. While this does not guarantee the exact moment of human discovery, it helps quantify the potential effort, in terms of time, that a developer would need to examine to detect a dependency. Figure 3.5 illustrates the dependency identification

time between a pair of changes. The left scenario shows when a change with a tag is created *before* the dependent change, while the right scenario depicts when the change mentioning the tag is created *after* the dependent one.

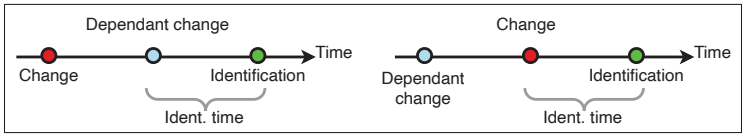


Figure 3.5 Different scenarios for dependency identification timing

We also quantify the lag between two dependent changes and the amount of changes that exist between them. The higher the lag and the number of changes in between, the more one has to dig into different changes to find their dependencies, hence the more effort required to find a dependency. We emphasize that lag is not considered a defect or shortcoming. Instead, it reflects the natural discovery process and helps characterize the effort involved in identifying dependencies.

To quantify how often dependencies are identified following a build failure, we search in the code review comments for the keyword $Build\_failed$. Figure 3.6 shows a concrete example of a build failure caused by the Zuul pipeline. We extract the date on which a dependency was identified and compare the dates of dependency identification to the first build failure.

On top of the lags between changes, we measure the lag between changes made by the same developer compared to pairs of changes made by different developers.

Results: Dependencies among changes are identified during the code review process for 51.08% of the merged changes, after a minimum, median, and maximum of 0, 5.06, and 12,791 hours from the creation of the last change of a pair of dependent changes, respectively. An example of dependencies that are identified during the code review is in the change *672416* [9], in which a reviewer suggested *"Blocking [the merge] until [672463] and [672477] [are] merge[d]"*. We also observe that the amount of dependent change with a pipeline

[9] https://review.opendev.org/c/openstack/python-octaviaclient/+/672416/3

build failure is common in OpenStack, accounting for 64.67% of dependent changes, among which 40.76% of the examined dependent changes have the declaration of dependency after the build failure.

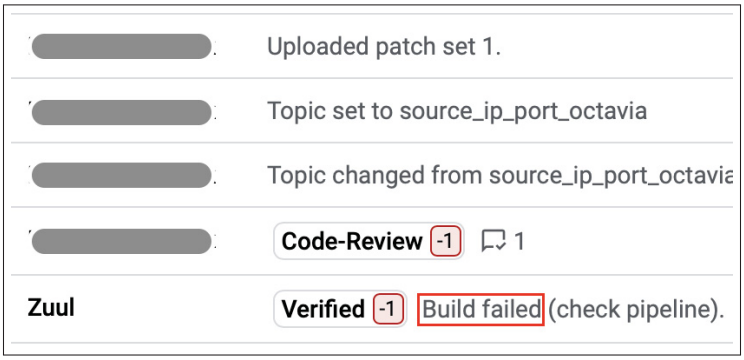


Figure 3.6 An example of a dependent change with a Zuul pipeline build failure

Two dependent changes have a median lag of 57.12 hours and 463 changes in between. OpenStack developers need to look up for a dependent change created before a median of 57.12 hours, which is the time difference between the *source* and *target* changes of a pair. Such lag can be as low as 0 hours and as high as 61,999.54 hours. Additionally, two related changes involve a minimum, median, and maximum of 1, 463, and 522K potential changes in between, respectively. For instance, the lag between the "812389" and "813013" changes has a lag of 57 hours, while they have 407 changes in between. Our results hint at the efforts needed to look for a dependent change and how likely one can easily miss a dependency among the plethora of potential changes.

The median lag between dependent changes made by different developers is 10 times the ones made by the same developers. We observe that the two distributions are statistically different (Mann-Whitney U rank test, $p - value < 0.05$, Cliff's Delta = 0.449). The shorter median for same-developer changes suggests they are typically identified more promptly, which is expected since developers tend to link their changes more efficiently. However, the lag for changes made by different developers can be ten days (median). In addition, 32.74% of the dependent changes made by the same developer have a lag higher than 5 days. This may indicate a risk that a developer forgets their prior changes and consequently misses adding the right

dependency. Apart from the dependencies among changes of the same developer, identifying dependencies made by different developers can be substantially difficult due to the coordination between different developers who have to declare dependencies between changes that are far apart from each other and among a large number of changes made between two dependent changes.

PRQ3. What are the categories of changes with dependencies?

Motivation: This research question aims to qualitatively understand whether dependent changes are more related to certain types of changes than others and to quantitatively understand the types of dependent changes for which developers add a dependency during the code review. Such an understanding is important to propose guided solutions for managing dependencies.

Approach: To qualitatively derive the types of changes with a dependency, we select a random sample of 379 OpenStack dependent changes that have a dependency tag (i.e., "Depends-On" or "Needed-By"). For each change, the first author thoroughly examines the description, the modified files, the added lines, the different revisions of the change, and particularly in which revision the dependency tag was added. We define a dependency as missed if it was added in the second or later revisions. Notably, some of these missed dependencies may also have been identified through mechanisms described in PRQ2, such as reviewer feedback or pipeline build failures. Similar to prior work Arabat & Sayagh (2024), we leverage the categories shown in Table 3.1, which describe the purpose of dependent changes.

To ensure unbiased classification, the second and third authors independently annotated a total of 100 cases (50 each), following the predefined coding schema. The raters classified each change into predefined categories (Table 3.1 identified in our prior work Arabat & Sayagh (2024)) based on a thorough examination of commit descriptions, modified files, revisions, and dependency tags. To quantify inter-rater agreement, we computed Cohen's Kappa coefficient Cohen (1960), which measures the level of agreement between raters while accounting for the possibility of chance agreement. The resulting Cohen's Kappa score of 0.644 indicates substantial agreement McHugh

(2012) among the raters, as it falls within the commonly accepted range of 0.6 to 0.8. This level of agreement supports the consistency of our classification schema. Minor discrepancies were resolved through discussion, which further refined the categorization criteria. Given the relatively high agreement, which reflects the simplicity and clarity of the classification, the remaining classifications were performed by the 1st author alone.

Table 3.1 A detailed overview of various reasons for which developers miss adding a dependency

Category	# missing deps	Total deps	% of missing
Configuration	43	110	39.09%
Dependency	28	55	50.91%
Refactoring	26	60	43.33%
New features	25	48	52.08%
Tests	23	52	44.23%
Code enhancement	14	26	53.85%
Documentation	8	13	61.54%
Moving resources	3	11	27.27%
Renaming	2	11	18.18%

Results: Dependent changes, while they can be related to different categories of changes, the configuration is among the most common types of changes with a dependency, as shown in Table 3.1. The types of changes with a dependency range between configuration, adding, upgrading, or downgrading 3rd party dependencies, refactoring, adding new features, etc. Interestingly, we also observe that documentation is among the categories of changes with a dependency, which might not be expected to have dependencies on other changes.

All the categories of related changes have missing dependencies, indicating no special challenges related to one category compared to the other. While from our previous research question, we observe that the percentage of dependent changes with a dependency during the code review is 51%, we observe that four of our studied categories of dependent changes are close to or above that percentage. 61.54% of the documentation dependencies are added during the code review, which might be the least expected category of changes to break a CI/CD pipeline. For example, the change 228530 [10] is a simple HTML change, while that change

[10] https://review.opendev.org/c/openstack/openstack-manuals/+/228530

depends on another change 228488 [11]. That is because this last change (i.e., 228488) creates a folder (using the "mkdir" command in a configuration file) in the server of documentation, which is used as a link in the change 228530. Thus, this last change should not be deployed without change 228488, which creates the folder; therefore, the dependency had to be added. Note that for certain cases, a dependency is added during the code review because a developer is working on both changes at the same time. For example, while the "850215" [12] change has a dependency on the "858919" [13] change in the 19th revision, the developer was working on such changes at the same time.

Summary

Dependent changes represent thousands of changes, which still have similar relevance in recent years. The dependencies are found for a significant percentage of dependent changes (51.08%) during the code review or following a build failure (41.58%). Dependencies can occur for different types of changes, from the ones that are likely to be dependent, such as configuration and 3rd party dependencies, to less expected types such as the documentation. **Our results suggest the need for approaches to assist practitioners in detecting dependencies as early as the creation of the change.**

3.7 An ML-based Approach to Predicting Software Change Dependencies

This section presents the rationale for research questions and the methodology employed to evaluate our machine learning-based approach for predicting software change dependencies. We first describe how our proposed approach would fit in practice. We also explore the rationale behind each research question, focusing on the challenges of identifying dependencies during code reviews and after build failures. Subsequently, we provide a detailed overview of our methodology, encompassing feature extraction, data construction, model training, and evaluation.

[11] https://review.opendev.org/c/openstack/training-guides/+/228488

[12] https://review.opendev.org/c/openstack/charm-octavia/+/850215

[13] https://review.opendev.org/c/openstack/charm-octavia/+/858919

3.7.1 Research Questions

In this section, we discuss the rationale for each research question to evaluate our ML-based approach to predict software change dependencies. In particular, we detail each research question below.

RQ1: What is the performance of our ML models in predicting software change dependencies?

The identification of dependent changes during code reviews or after build failures highlights the challenges practitioners face in managing dependencies, particularly given the need to examine a large volume of changes. This research question evaluates various machine learning models designed to assist in this process by predicting dependencies. Specifically, it investigates two models: the *dependent-change predictive* model, which predicts whether a change has a dependency, and the *dependent-pair predictive* model, which predicts whether two changes are dependent. These models aim to streamline the dependency identification process, reducing manual effort and improving the efficiency and reliability of the software development process.

RQ2: What features have the greatest impact on predicting software change dependencies?

Understanding the key features that influence the probability of dependency predictions is essential for improving the interpretability and reliability of the proposed approach. This research question aims to identify the most impactful features contributing to the prediction outcomes and analyze how these features affect the likelihood of dependencies. Such insights provide a deeper understanding of the model's decision-making process, helping to refine the models and guide practitioners in making informed decisions when managing software change dependencies in a large-scale system.

RQ3. How does the performance of agentic AI compare with that of our machine learning models?

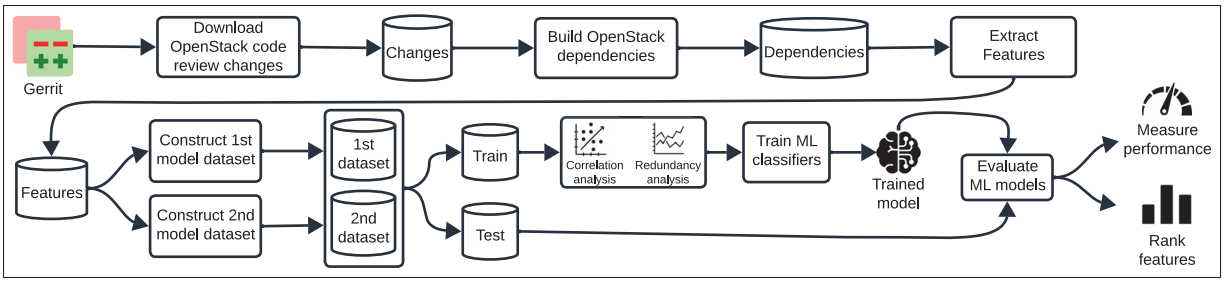


Figure 3.7 An overview of our methodology design

With the advent of Large Language Models (LLMs), particularly Agentic-AI systems, intelligent agents are increasingly being integrated as active teammates throughout the software development lifecycle. In this context, we aim to rigorously evaluate the capability of Agentic-AI in the task of dependency identification and to systematically compare its performance against traditional machine learning algorithms.

3.7.2 Methodology

This section provides a detailed overview of our methodology, which is based on two machine learning models to predict dependent changes. In particular, our approach consists of two steps: (1) predicting which changes have a chance to be dependent, then (2) predicting which exact pair of changes depend on each other. We set out to split the problem into two sub-problems because using a single model would be costly, both in terms of the number of links to build among all the changes and the expense of collecting metrics for a large number (potentially billions) of change pairs. In other words, linking each change to all of its prior changes, even if a change has no chance of being dependent, would result in an excessive number of change pairs. Moreover, having a large number of unrelated changes would increase false positive rates. Therefore, we first reduce the scope of the changes by predicting only the changes that are likely to be related to another change, and then identify the exact pairs of dependent changes.

We expect developers to leverage our approach following the workflow shown in Figure 3.8. A developer leverages our first model to assist her identifying if her change “A” is likely to depend on another change. The prediction is reviewed by the user who then leverages our second model

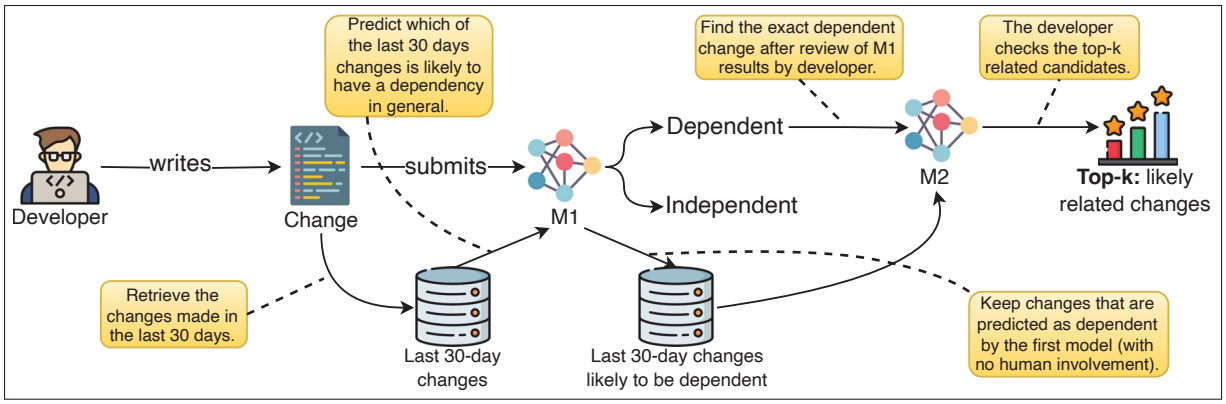


Figure 3.8 A practical overview of the two-model solution

to identify the right change on which her given change "A" depends. To do so, we link the change of the input (change "A") to the list of changes made in the last 30 days, which are predicted by the first model as likely to participate in a dependency. Each of the constructed pairs is provided to the second model that predicts its probability of being dependent. We sort these potential changes using the predicted probabilities and exclude any change whose probability is less than 50%. That way, the second model reports a reduced list of changes to inspect instead of randomly searching among all the hundreds or thousands of changes to find a dependency.

In particular, our study design consists of six steps, as illustrated in Figure 3.7. Note that the data collection process is further detailed in Section 3.5. Hence, we discuss each step of our evaluation design as follows.

Table 3.2 Dimensions and their respective features used to evaluate various machine learning classifiers in our study. Note that Pair-related features were used to evaluate the second model

Dim.	**Feature**	**Description**
Change	*insertions*	# of added lines of code.
	deletions	# of deleted lines of code.
	code_churn	The sum of inserted and deleted lines of code.
	num_directory_files	The # of directories touched by the changes.

Continued on next page

Dim.	Feature	Description
	is_non_functional	Is the change functional?
	has_feature_addition	Does the change involve a new feature?
	is_corrective	Does the change fix an issue?
	is_merge	Is the change a merge?
	is_preventive	Is the change preventive?
	is_refactoring	Is the change about refactoring?
Developer	*num_cro_pro_cha_owner*	The # of cross-project changes the developer had in the project.
	num_wthn_pro_cha_owner	The # of within-project changes the developer had in the project.
	num_whole_cha_owner	The # of changes the developer had in all projects.
	pctg_cro_pro_cha_owner	The % of cross-project changes the developer had in the project.
	num_pro_cont_owner	The # of projects the developer contributed to.
	num_pro_cha_owner	The # of changes the developer had in the project.
	pctg_dep_chan_owner	The # of dependent changes divided by all dependent changes of the developer.
Project	*project_age*	The age of the project in days.
	num_dep_proj_last_mth	The # of projects with which the change's project interacted in the last month.
	num_cro_pro_cha_lst_mth	The # of cross-project changes in change's project made in the last month.
	num_cro_pro_chan	The # of cross-project changes in the project.
	num_wthn_pro_cha	The # of within-project changes in the project.
	pctg_cro_pro_chan	The % of cross-project changes in the project.
	num_whole_wthn_pro_cha	The total # of within-project changes.
File	*num_file_changes*	The # of changes containing any modified files.

Continued on next page

Dim.	Feature	Description
	num_file_types	# of modified file types.
	num_dev_mod_files	The average # of developers who modified file.
	(label[1])_mod_fil_dep_cha	The %, min, med, and max # of modified files in dependent changes.
	file_type[2]	Type of modified files (src_code, config, tests, docs, build_ci, scripts).
Text	*subject_length*	The length of the change title.
	description_length	The length of the change body description.
	subject_word_count	The # of words contained in the title of the change.
	description_word_count	The # of words contained in the body description of the change.
Pair	*msg_similarity_score*	The cosine similarity between the descriptions.
	n_common_reviewers	The # of common reviewers.
	pctg_shrd_file_tkns	The % of shared file tokens.
	pctg_shrd_desc_tkns	The % of shared description tokens (words).

(i) Feature extraction: We collect 36 and 82 features (shown in Table 3.2) to train our first and second ML models, respectively. Our features are inspired by prior work Chouchen, Ouni, Olongo & Mkaouer (2023) and extended with features related to the dependencies, such as the number of prior dependencies for the first model and the similarity between two changes for the second model. The obtained features are divided into six different dimensions related to the change itself, the developers of changes, the project status when the change is made, the files that were changed, textual features representing the description of the change, and pair-related features that compare two changes. Note that we do not consider the Pair dimension for the first model. For the second model, we consider all the features for the source change and the same

[1] Label refers to pctg, min, median, and max.

[2] File type includes src_code, config, tests, docs, build_ci, and scripts.

features for the target change, on top of which we add the Pairs dimension. Our features are measured for each change at its creation time. In other words, since a change can have multiple patches, we only consider the first patch as the principal change. That is because we expect users to use our models as soon as they create a change. We further discuss each dimension and how it contributes to predicting software change dependencies as described below:

- **Change:** characterizes the nature and complexity of an individual change. Prior research found that changes with high churn are more likely to introduce defects Göçmen, Cezayir & Tüzün (2025). Such defects could propagate to other components if not well tested. Features like *insertions*, deletions, and *code_churn* provide insights into the extent of modifications, which can indicate the complexity and impact of the change. The *is_merge*, *is_preventive*, and *is_refactoring* features help identify the purpose of the change, which can influence its relationship with other changes. For instance, changes addressing the same bug might share important information, hence increasing the chance of their dependencies.
- **Developer**: describes the experience of the developer on the project. Many research studies have used the *developer* experience in many SE tasks such as software vulnerabilities Shin, Meneely, Williams & Osborne (2011). Hence, developers with extensive involvement might introduce changes that are more interdependent due to their broader understanding and influence on the system. *num_cro_pro_cha_owner* and *num_wthn_pro_cha_owner* indicate the extent of a developer's involvement across and within projects, respectively. The *pctg_dep_chan_owner* feature, which measures the percentage of dependent changes a developer has made, directly indicates the developer's tendency to create interdependent changes. This can be crucial for predicting dependencies, as developers with a history of creating dependent changes might hold this pattern.
- **Project:** is designed to distinguish the history and evolution of the project (repository). *project_age* and *num_dep_proj_last_mth* denote the age of the project in days, while *num_dep_proj_last_mth* indicates the number of projects with which the project of the change co-changed in the last month. Similarly, *num_cro_pro_chan* represents the number of cross-project changes the change's project had in the past, whereas *pctg_cro_pro_chan* is defined as the percentage of cross-project changes relative to the dependent changes in the

change's project. The *num_cro_pro_cha_lst_mth* and *num_wthn_pro_cha_lst_mth* features indicate recent cross-project and within-project changes, respectively, which can suggest areas of the project that are currently active and potentially interdependent.

- **File:** focuses on the modifications made to individual files. According to Zimmermann Zimmermann, Zeller, Weissgerber & Diehl (2005), files that frequently change together are more likely to change in the future. *num_file_changes* and *num_file_types* provide information about the diversity and extent of file modifications, which can indicate the potential impact and dependency of changes. Changes affecting a large number of files or file types might be more likely to be dependent on other changes. The *avg_num_dev_mod_files* feature, which calculates the average number of developers modifying any file of the change, can indicate the collaborative nature of the change, suggesting potential dependencies. Furthermore, we consider the modified files in the change. Specifically, we check whether a change contains tests, docs, CI/build, config, source code, or script files using a set of keywords, such as the presence of the "test" word in the list of modified files.
- **Text:** analyzes the textual content of change descriptions. *subject_length*, *description_length*, and *subject_word_count* provide insights into the detail and complexity of the change descriptions, which can indicate the nature and impact of the changes. More detailed descriptions might suggest more complex changes that are more likely to depend on other changes.
- **Pair:** describes the relationship between two changes. *n_common_reviewers* represents the number of common reviewers between the two projects. Changes involving the same or a similar set of developers might be more likely to be dependent due to shared knowledge and context. The *msg_similarity_score* feature, which measures the similarity score between the embeddings of two descriptions. Such a similarity can capture mostly likely related context, such as the implementation of a new feature.

We note that similarity-based metrics (i.e., *msg_similarity_score*) are measured using a "all-MiniLM-L6-v2" sentence embedding model, which we use to obtain the embedding (i.e., numerical vector representation) of each change in a pair of changes. Afterward, we leverage

the cosine similarity on the two embeddings of a pair to measure the similarity score of a pair of changes.

(ii) Data construction: To train our first model, we use the merged and abandoned changes of OpenStack and exclude those that are still open. We keep both the merged and abandoned since they can both have dependencies. We exclude the open changes since they can still have a dependency added during the code review process and before being merged or abandoned. We leverage the features discussed above as independent features (aka., the inputs used to make the prediction), and whether a change has a dependency, as our dependent feature (aka., output being predicted).

To train and test our second model, we first build pairs of changes. For the training set, each change is considered as a target change, then linked with its dependency made in the last 30 days. We also randomly select a random sample of pairs of changes that are independent for the negative sample of data to train the model. To construct the testing set, we select a random sample of 378 dependent changes per testing fold and link them to changes that are both (1) made in the last 30 days and (2) are predicted by the first model as potentially dependent. Zhou, Chen & Lipton (2022) stated that choosing a sliding window is often a good approach when the dataset is large. We chose 30 days as a time window for two main reasons. First, we aim to capture the most recent changes contributed within the last month. Another reason is related to the cost of constructing billions of change pairs and the cost of constructing pair metrics.

(iii) Correlation and redundancy analysis: To train and test our models and to avoid any data leakage, we leverage a **time-aware 10-fold cross-validation** technique, similar to prior studies Shehab, Hamou-Lhadj & Gunda (2023); Chouchen *et al.* (2023), as summarized in Figure 3.9. Specifically, we use the first fold for training and the second for testing. Then, we use the first two folds for training and the third one for testing. We continue this process by repeatedly increasing the number of training folds and using the following fold for testing. Note that the folds are created based on the list of changes. From each training set or testing set, we then construct the pairs of changes as discussed earlier.

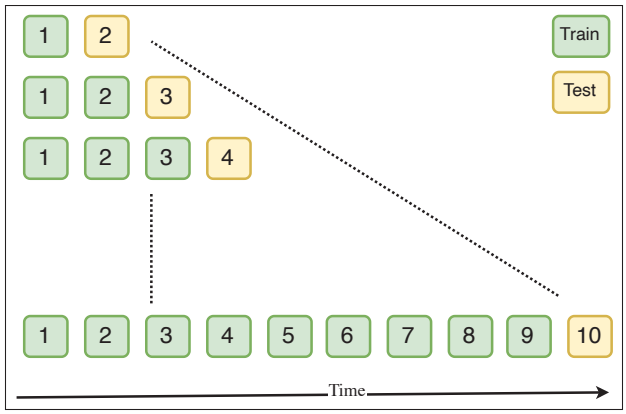


Figure 3.9 10-fold time-aware cross-validation process leveraged to evaluate the performances of various ML classifiers

For each training dataset, we first remove correlated and redundant features since collinearity can impact the interpretation of our models according to Jiarpakdee, Tantithamthavorn & Hassan (2019). We remove features that have a Spearman correlation above 0.7 and redundant features with a threshold of 0.9, similar to prior work Ouatiti, Sayagh, Kerzazi & Hassan (2023). Since we leverage 10 folds and each fold can have its own correlated and redundant features, we perform the correlation and redundancy analysis manually, trying to always remove the same features to have a consistent interpretation of the model. For example, if features A and B are correlated in the first and second folds, we remove A from both folds for consistent analysis.

(iv) Model evaluation: After removing collinearity, we evaluate different machine learning algorithms for our two models, namely **Random Forest, XGBoost, AdaBoost, Extra Trees, and Multilayer perceptron**, which are widely used by prior studies (e.g., Chouchen *et al.* (2023); Zhang, Lo, Xia & Sun (2015); Tantithamthavorn, McIntosh, Hassan & Matsumoto (2017); Chen *et al.* (2020)). Hence, the performance of each classifier is quantified in each iteration, resulting in a total of 10 performance metrics. The final performance score for each classifier is calculated as the average of these performance metrics across the 10 folds. The performance metrics we use in our study are the **AUC (Area Under Curve) and Brier Score**, which are standard performance metrics widely used by prior work Ouatiti *et al.* (2023) and are more suitable for imbalanced data Tantithamthavorn *et al.* (2017). The AUC measures the

discrimination ability of a model, whereas the Brier Score measures how accurately a model predicts the correct class. The higher the AUC, the better the model, with an AUC of 50% representing a random guess. The lower the Brier Score, the better the model. Furthermore, we measure the performance of the first model by reporting the precision-recall curve, which is useful to determine the tradeoff between precision and recall with respect to various thresholds. Precision and recall are expected to change according to the prediction threshold. The higher the threshold, the fewer false positives and hence higher precision. The lower the threshold, the more instances are predicted as positive instead of negative, reducing the chance of false negative and increasing the true positives, hence increasing the recall. According to the context, developers can decide on the right threshold. For instance, to avoid any build failure caused by missing dependencies, one can reduce the threshold to increase the recall. On the other side, under time constraint, one can look for only relevant changes and increase the threshold to improve the precision. In this paper, we evaluate the first model using the AUC, Brier Score, precision-recall curve.

Additionally, we measure the performance of the second model using two key metrics: *top-k precision* and *top-k recall*. We choose these performance metrics since we aim to recommend a set of changes sorted from the change that is more likely to depend on the user's change to the one that has the lowest chance of being related to the same user's given change. As such, the changes predicted by our second model are sorted according to their respective probabilities. We then report top-k precision and top-k recall. Top-k recall indicates the number of times our second model is able to predict the right dependent change within the first k predicted changes. The top-k precision indicates the average amount of false positives reported within the k changes. The higher the top-k recall and top-k precision for lower k is the better the performance of our model. In this paper, we report on top-1, top-3, top-5, top-7, and top-10. Note that it is expected for the top-k recall to increase with the increase of k as we report more candidates among which we might find the correct changes, hence increasing the chances of predicting the correct change. The top-k recall, in other words, is not expected to drop with the increase of k. On the opposite, for the top-k precision, we expect it to drop with the increase of k as we introduce

more candidates with the goal of finding a single source change (most of the changes have only one source dependency). For instance, the top-10 can be 10% in the best case when the top-10 recall is 100% as 9 instances are false positives, by definition and one (the change that one is looking for) is the true positive. In this paper, we also exclude any pair of changes predicted with a probability less than 50%. For example, if four changes are predicted with a probability higher than 50% and six changes with less than 50%, we report only the top 4 changes for all the top-k precision and recall, including for measuring the top-10 precision and recall.

(v) Ranking of the most important features: We extract the most important features from each of the 10 trained models of RQ1 (i.e., using the 10-fold time-aware models) to obtain 10 feature importance scores for the first model and 10 for the second model. These feature importance scores are obtained using the default feature importance of the classifier (the classifier's `feature_importance` attribute). To calculate these features' scores, the model leverages the Mean Decrease Accuracy (aka., gini importance), which undergoes two main steps: (1) creates the same instance of the test set and shuffles values of a particular feature, (2) measures the performance on the modified test set and compares with the original one. This process is performed for each feature. We then summarize these 10 rankings into one final ranking using the Scott-Knot algorithm Tantithamthavorn, McIntosh, Hassan & Matsumoto (2019), similar to prior work Chouchen *et al.* (2023); Ouatiti *et al.* (2023).

(vi) Impact of each feature: We examine how each feature impacts the model's predictions by studying whether increasing the value of a feature increases or decreases the predicted probability by a model, similar to prior work Ouatiti *et al.* (2023). For instance, a feature A has a positive impact on the prediction if increasing the value of A increases the predicted probability of the model. It has a negative impact if the probability decreases with the increase in the value of feature A. In particular, we consider the following steps to measure the impact of each of our features. In the paper, we report our results on the top 10 features, where we put the impact of all of the features in our replication package.

- We create an artificial instance where all the features are set to their median values, and then predict the probability P_i using our first model.

- We then increase the value of one feature (e.g., feature A) by its corresponding standard deviation (median + 1sd), while keeping all the other features at their median values.
- We predict the probability P_t of the new instance using our first model.
- We investigate the impact of increasing our studied feature (e.g., A) on the model as $\frac{P_t - P_i}{P_i}$. Since we have 10 models (based on the 10-fold time-aware cross-validation), we report the minimum, median, and maximum impact in our results.
- We re-conduct the same experiment for all features. We also follow the same process for our second model.
- We report the median impact scores across the 10 folds for the first and second models.

3.8 Evaluation Results

In this section, we discuss the answers to the two research questions to evaluate the performance of our models in predicting dependencies among software changes.

RQ1. What is the performance of our ML models in predicting software change dependencies?

Motivation: Since a non-trivial amount of dependent changes are found during the code review or have at least one build failure before the identification of the dependency and since one has to look into a large amount of change to spot potential dependencies, we propose and evaluate in this RQ machine learning algorithms for assisting practitioners in the identification of dependent changes through two different models: *dependent-changes predictive* (i.e., whether a change has a dependency or not) and *dependent-pairs predictive* (i.e., whether two changes are dependent) models.

Approach: The evaluation of the first model is performed on the entire test dataset, whereas the second model is evaluated on the constructed testing data discussed in the previous section. In particular, we select a sample of dependent changes for the second model and link each of them to the changes that satisfy two conditions: (1) made within the last 30 days and (2) are predicted as potentially dependent by the first model. Each change predicted by the first model

with a probability higher than 50% is a change likely to participate in a dependency. Since predicting dependent changes made by the same developer may be easier than predicting those made by different developers, we conduct three separate evaluations that distinguish the types of dependencies (i.e., same-developer vs. different-developer changes). Specifically, we assess the performance of the second model on: (1) pairs of changes made by different developers, (2) pairs of changes made by the same developer, and (3) all pairs of changes. This allows us to assess whether the model is efficient in capturing dependencies among changes made by different developers, which is expected to be more challenging and typically exhibits higher lag compared to the identification of changes made by the same developer. To train and test our models, we follow the methodology discussed in Section 3.7.2.

To better understand which dimensions of features (shown in Table 3.2) contribute to the performance of our model, we investigate **the explanatory power and contribution of each dimension**. We do so by conducting two experiments: (1) we evaluate a model trained only on the features of one dimension, and (2) we evaluate the model with all features but the features of the same studied dimension. For this purpose, we leverage Random Forest and XGBoost classifiers to perform these experiments since they are the best-performing algorithms. A dimension has a significant explanatory power when its related model has an AUC greater than the random guess (i.e., AUC = 0.5), as reported by prior work Lee *et al.* (2020). Similarly, a model without a high drop in the AUC when a dimension is omitted suggests the low impact of that dimension, hence a low contribution toward the performance of our models.

Table 3.3 Performance metrics of the first model in terms of AUC and Brier score

Classifier	AUC (%)	Brier Score
XGBoost	**88.45**	**0.201**
AdaBoost	87.43	0.215
RF	87.42	0.221
MLP	74.49	0.325
ET	59.05	0.532

Results: Our first model is able to predict dependent changes with an AUC of 88.45% and a Brier Score of 0.201 with the XGBoost algorithm, demonstrating an excellent prediction

performance, as shown in Table 3.3. The XGBoost and AdaBoost show slightly different performances that are as low as a difference of 1.02%, as the two models have an AUC of 88.45% and 87.43%, respectively. Similarly, Random Forest exhibits a comparable AUC of 87.42%, not far from XGBoost. The lowest performance is observed with the Extra Trees (ET), which reaches an AUC of just 59.05%, still higher than a random guess baseline (AUC of 50%). Our results hold for the Brier Score, where the AdaBoost, Random Forest, and XGboost models have the best scores, with a Brier score of 0.221, 0.215, and 0.201, respectively, representing a low difference of just 0.02. We first focus on the AUC and Brier score, which are the best fit for unbalanced datasets Tantithamthavorn, Hassan & Matsumoto (2020); Ouatiti *et al.* (2023).

The precision-recall curves show similar trends in most of the testing folds, as shown in Figure 3.10. For example, the precision and recall of the XGBoost model are 21% and 92%, respectively, with the prediction threshold of 50%. Increasing the threshold to 90% increases the precision to 37.6% in favor of a recall that drops to 60.3%. As such, depending on the context, users can adjust their threshold to identify whether their change is likely to have a dependency or not. For instance, decreasing the threshold would lead to more efforts to inspect potential changes, while increasing the threshold can miss relevant dependencies and lead to potential build failures. Yet, our model is already able to exclude a good amount of changes, which can reduce human efforts on finding the right dependency, as further discussed in Figure 3.11. Note that the low precision is expected in an unbalanced problem. In fact, we find that the model is able to exclude a large number of negative changes. For example, with the threshold 50%, the model is able to successfully exclude a median of 40,168 independent changes and successfully predicts a median of 4,188 positive changes with 19,060 false positive changes. Increasing the threshold to 90% the model is able to correctly identify a median of 3,012 dependent changes with 4,961 false positives. With the same threshold, the model successfully excludes a large portion of noise (independent changes), which consists of a median of 54,057 negative changes. Figure 3.11 shows the frequency of TP, TN, FP, and FN rates achieved by the first model using XGBoost.

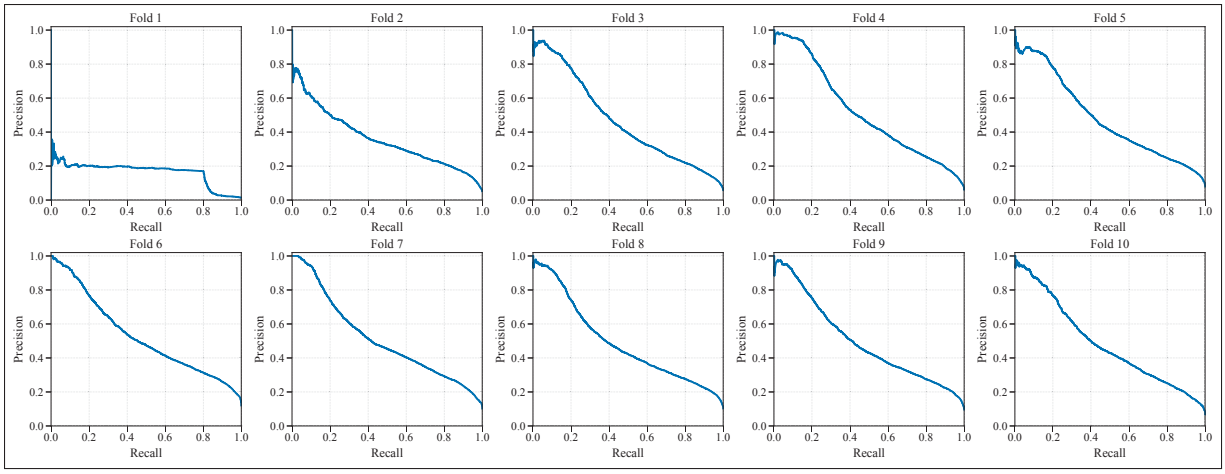


Figure 3.10 The precision-recall curves for the 10 folds using the XGBoost classifier

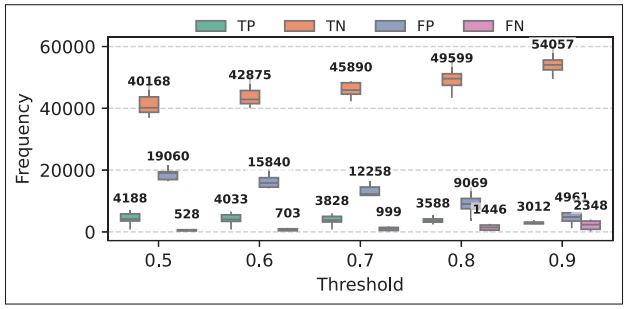


Figure 3.11 The frequency of TP, TN, FP, and FN for the first model using XGBoost

Our second model demonstrates promising predictive performance, achieving an AUC of up to 96.46%, as detailed in Table 3.4. This evaluation identifies whether the second model is able to find a single dependent change within thousands of changes (an average of 4,715 changes made per month in OpenStack), which measures the amount of effort that our model can reduce by guiding developers into a focused list of changes to inspect to find a dependency. Among classifiers, XGBoost achieves the highest AUC of 96.46%, with AdaBoost and Random Forest performing comparably. From a usability perspective (top-k recall), one is expected to find its dependent change within a focused list of 10 candidate changes in 67.19% of the cases with XGBoost. As the model outputs a maximum of 10 changes within the top-10, it is expected that in the best case, only one of the reported changes is correct, with the others considered as false

Table 3.4 The performance of the second model, including AUC, Brier Score, top-k precision, and top-k recall across ML classifiers and developer settings. Note that top-1 precision and top-1 recall will always yield the same value, as each candidate list contains only a single instance (i.e., one pair of changes)

Model	Dev Type	Metric	k=1	k=3	k=5	k=7	k=10	AUC	Brier
AdaBoost	Same	Precision	48.11	31.19	25.92	23.75	22.16	84.62	0.312
		Recall	48.11	70.61	78.17	82.57	85.92		
	Different	Precision	21.91	12.73	9.03	6.97	5.58	93.31	0.242
		Recall	21.91	35.82	41.10	43.92	50.08		
	All	Precision	31.58	17.25	12.15	9.47	7.23	95.56	0.243
		Recall	31.58	48.81	55.70	60.10	64.70		
ET	Same	Precision	15.04	16.51	17.65	18.14	18.62	59.17	0.672
		Recall	15.04	28.44	38.41	44.96	52.43		
	Different	Precision	2.07	1.00	0.74	0.73	0.59	63.77	0.507
		Recall	2.07	3.00	3.68	4.94	5.77		
	All	Precision	12.75	6.02	4.05	3.18	2.45	67.15	0.510
		Recall	12.75	17.64	19.54	21.26	23.18		
MLP	Same	Precision	25.19	21.12	19.92	19.36	18.98	59.36	0.787
		Recall	25.19	42.50	51.43	57.22	63.68		
	Different	Precision	1.88	1.35	1.44	1.49	1.32	75.26	0.422
		Recall	1.88	4.04	7.20	10.27	12.68		
	All	Precision	7.46	5.33	4.23	3.70	3.16	84.20	0.428
		Recall	7.46	15.34	19.62	23.23	27.40		
RF	Same	Precision	40.63	29.30	25.29	23.43	22.01	83.08	0.534
		Recall	40.63	65.23	74.90	80.39	85.46		
	Different	Precision	11.54	8.07	6.58	5.50	4.60	93.92	0.265
		Recall	11.54	23.19	30.90	35.37	41.92		
	All	Precision	22.67	13.25	10.16	8.25	6.59	95.93	0.269
		Recall	22.67	37.41	46.35	51.53	57.36		
XGBoost	Same	Precision	48.85	31.57	26.21	23.89	22.19	85.05	0.702
		Recall	48.85	71.76	79.53	83.76	87.68		
	Different	Precision	21.04	12.66	9.54	7.65	6.08	93.94	0.305
		Recall	21.04	36.12	43.99	48.51	54.30		
	All	Precision	32.29	17.75	12.72	9.97	7.65	96.46	0.312
		Recall	32.29	49.69	58.18	62.52	67.19		

positives. As such, a top-10 precision of 7.65% is not low but expected. We also find that the model is better for changes made by the same developer compared to changes made by different developers. However, developers can find the dependent change that is made by a different developer in more than half of the cases (top-10 recall of 54.30%) within the top-10 changes. We believe that this is beneficial for developers who, instead of looking at thousands of changes made in the entire OpenStack system, made up of hundreds of projects Arabat & Sayagh (2024), will inspect only a focused list of 10 changes to find their relevant dependent change. In an

average of 21.04% of the cases, the dependent change made by a different developer can be even in the top-1.

Even when extending the time window to 60 days (searching within the last 60 days instead of 30 days), we generally do not observe significant differences in the performance of the second model across different developer settings. In particular, we observe that the second model achieves an average AUC of up to 95.69% when the 30-day time window is used to select prior changes, while the AUC when extending the cutoff to 60 days is 95.85%. Focusing on XGBoost, as the best classifier, the top-k recall at top-10 is 75.09% when the timeframe is 30 days, and 73.31% for 60 days, a difference of just 1.78%. Their corresponding top-1 precisions are 38.79% and 40.21%, respectively. When the pairs are made by the same developer, the top-10 recalls for the 30-day and 60-day timeframes are 85.36% and 85.77% (a difference of 0.41%). When pairs are made by different developers, we observe that the top-10 recall drops by roughly 10% when extending the timeframe from 30 days (56.82%) to 60 days (46.67%), whereas the top-1 precision drops by 5% (from 25% to 20%). Exceptionally, AdaBoost has the best top-10 recall of 48.89% when the chosen timeframe is 60 days, while 30-day shows 29.55%. Our findings indicate that even when extending the time cutoff from 30 to 60 days, the performances are generally not significantly different, with few fluctuations across developer settings and certain classifiers. Note that this evaluation of 60 days is made on the sample coming from the last testing fold.

While the *Pair*- and *Developer*-related features individually contribute significantly to model performance, combining all feature dimensions yields the best predictive results. The *Pair* and *Developer* dimensions-capturing developer experience and characteristics of two potentially dependent changes-have a substantial impact on both models. Specifically, the first and second models achieve AUC scores of 83.82% and 81.67%, respectively, when trained solely on *Developer*-related features. Specifically, the second model achieves an AUC of 94.95% when using only *Pair*-related features, as shown in Table 3.5. Furthermore, incorporating all features leads to consistent performance gains. For instance, the first model outperforms its *Developer*-only counterpart by 4.63%. Using all features, the second model shows AUC

Table 3.5 Performance metrics in terms of AUC of the first and second models when using/not using a given dimension and when all dimensions are considered. *Dim.* stands for dimension name and *Dim. incl.* stands for dimension included

Dim.	Dim. incl.	Model 1	Model 2
Change	Only	61.31%	63.80%
	Without	87.65%	96.24%
Text	Only	71.04%	60.53%
	Without	86.53%	96.24%
Developer	Only	83.82%	81.67%
	Without	74.10%	96.07%
Project	Only	60.40%	63.25%
	Without	87.43%	95.96%
File	Only	63.26%	53.25%
	Without	87.64%	96.25%
Pairs	Only	N/A	94.95%
	Without	N/A	77.90%
All	All	88.45%	96.46%

improvements of approximately 14.79% over developer *Developer*-related features and 1.51% over *Pair* features. These findings highlight that, although individual feature dimensions, such as *Pair* or *Developer*, offer strong predictive signals, their combination provides a more comprehensive performance, leading to improved prediction of dependent and pairs of dependent changes.

Summary of RQ1

Our evaluation shows that both the first and second models demonstrate promising performance based on AUC and Brier scores. Our findings indicate that using Random Forest and XGBoost algorithms is effective for predicting dependent changes and pairs of dependent changes, respectively. Yet, the second model performs well when considering different developer settings. Moreover, all dimensions significantly contribute to the high AUC performances, with a higher impact from the *Developer* and *Pair* dimensions. **Our obtained high performances suggest that developers could adopt our proposed approach to identify dependent and dependent-pair changes in advance.**

RQ2. What features have the greatest impact on predicting software change dependencies?

Motivation: The goal of this research question is to better understand the most important features that influence the probability of our predictions and how such a probability is impacted by each feature.

Approach: To identify the most important features, we focus on the XGBoost classifier for the first and second models, as it shows the best performance for the second model and the second best for the first model, with a tiny difference with Random Forest (according to RQ1). Our analysis consists of two steps: (1) ranking the features from the most to the least important ones and (2) identifying whether each feature positively or negatively impacts the prediction. Note that we report the median impact across the 10 folds. For a detailed explanation, we refer the readers to Section 3.7.2. To conduct statistical tests, we use the Mann-Whitney U test Mann & Whitney (1947) to compare dependent and non-dependent changes. Furthermore, we also apply the Bonferroni correction Bonferroni (1936) to control family-wise error rates.

Results: Changes with a larger amount of deleted lines that aim to implement new features inherently amplify the probability of changes to be predicted as dependent, while changes that are for refactoring or fixing bugs decrease such a probability, as shown in Table 3.6. We observe that complex changes with a high number of deleted lines (i.e., *deletions*) increase the probability of a change being predicted as dependent. For example, the "425300" change, as a dependent change, removes more than 7.9K lines of code. Such an observation can be explained by the fact that removing lines might have a larger impact on the system that needs to be updated according to the removed lines. We also observe that the number of added lines (i.e, *insertions*) has a positive impact at the beginning of the project, while a negative impact later as models are trained on more data. On the other hand, the "242556" [14] change, as a non-dependent change with 100 lines of code, adds a simple *"check to avoid storing samples with None or not numerical volumes"*, thus does not depend on other changes. Adding new features (i.e., *has_feature_addition*) is more likely to have an impact on other changes since a large change

[14] https://review.opendev.org/c/openstack/ceilometer/+/242556

Table 3.6 Ranking of the top 10 most important features of the first model and their impacts. ↗ denotes positive effect size, while ↘ is negative

Feature	Ranking	Median impact	p-value (Bonferroni)	Effect size
ratio_dep_chan_owner	1	37.251	0	large (↗)
cross_project_changes_owner	2	0.099	0	medium (↗)
max_num_mod_file_dep_cha	3	3.010	0	medium (↗)
project_changes_owner	4	-0.755	5.81×10^{-6}	negligible (↗)
min_num_mod_file_dep_cha	4	1.688	0	small (↗)
pctg_cross_project_changes_owner	4	0.000	0	medium (↗)
ft_src_code	4	0.000	0	negligible (↘)
deletions	5	0.479	3.16×10^{-180}	negligible (↘)
whole_changes_owner	6	-0.014	0	small (↗)
whole_within_project_changes	6	0.172	0	small (↗)
is_corrective	7	-0.155	2.24×10^{-233}	negligible (↘)
is_preventive	8	0.081	0	negligible (↗)
description_length	8	0.540	0	small (↗)
pctg_cross_project_changes	9	0.046	0	small (↗)
num_file_types	10	0.000	6.01×10^{-270}	negligible (↗)
ft_docs	10	-0.041	2.57×10^{-202}	negligible (↘)
ft_config	10	0.000	0	small (↗)
has_feature_addition	10	0.000	0	negligible (↗)
last_mth_dep_proj_nbr	10	0.210	0	medium (↗)
subject_length	10	0.000	1.00×10^{0}	negligible (↗)
ft_scripts	10	0.225	7.18×10^{-214}	negligible (↗)

can be broken down into smaller changes. On the other side, fixing bugs (i.e., *is_corrective*) or refactoring (i.e., *is_refactoring*) might be seen as small changes that decrease the chances of a change being predicted as dependent. To emphasize this, dependent changes have a statistically significantly higher number of added lines and deleted lines in comparison with non-dependent changes. Bonferroni correction shows a p-value of 8.15×10^{-15} and 3.16×10^{-180}, respectively, though both have a negligible effect size.

Changes with lengthier descriptions are more likely to be predicted as dependent. *Description length* is among the top six most important features for predicting whether a change has a dependency. The longer the description of the change, the more likely that the change has a dependency, while the length of the subject is not conclusive, as it has no impact on

Table 3.7 Ranking of the top 10 most important features of the second model and their impacts. Note that features with no impact were omitted

Feature	Ranking	Median impact
msg_similarity_score	1	0.136
ratio_dep_chan_owner_target	2	-0.003
n_common_reviewers	2	0.136
ratio_dep_chan_owner_source	3	0.036
is_preventive_target	4	0.014
pctg_mod_file_dep_cha_target	5	0.014
whole_within_project_changes_source	6	0.047
cross_project_changes_owner_source	6	-0.010
pctg_cross_project_changes_owner_target	7	-0.066
num_shrd_desc_tkns	8	0.088
projects_contributed_owner_source	9	-0.046
is_preventive_source	9	0.009
pctg_cross_project_changes_owner_source	10	-0.020
ft_scripts_source	10	0.010

the prediction of the first model and a negative impact on the second model. We also note that dependent changes are statistically significantly higher in description length than non-dependent changes. Bonferroni correction yields a $p - value < 0.05$ and small effect size, as shown in Table 3.6. Our results suggest the use of description as an important means of the identification of dependencies.

Changes to projects made by developers with a larger experience on OpenStack, yet less experience on the change's project, are more likely to be predicted as dependent, as shown in Table 3.6. Changes that are made by developers who have diverse contributions in terms of projects, as the higher the number of projects one has contributed to (i.e., *projects_contributed_owner*) increases the probability of her change being predicted as dependent. In the same direction, the higher the number of changes one makes across OpenStack projects (i.e., *cross_project_changes_owner*), the higher the probability a change is predicted as dependent. have higher chances of their change being dependent. The Bonferroni correction confirms that this difference is statistically significant, with a $p - value < 0.05$ and a medium effect size. Statistically, we also observe that developers with at least one dependent change contribute to multiple projects more frequently than those with no prior dependent changes.

Recent project interactions might help in identifying dependent changes. In fact, we observe that projects having a higher number of dependencies with other projects (i.e., *pctg_cross_project_changes*) increase the chances of the same project's changes being dependent. Further, we observe that changes to older projects (i.e., *project_age*) are likely to be predicted as dependent, as the increase of a project age decreases the probability of a change being dependent. Such an observation hints at the fact that a dependency can occur when it is gradually being used by other projects. The Bonferroni correction indicates that the age of projects with a dependency is statistically significantly different ($p - value < 0.05$ and small effect size) from the age of projects without a dependency, as illustrated in Table 3.6.

Changes with files that are frequently maintained in a larger number of previous dependent changes are more likely to be predicted as dependent. In particular, the more developers touch files that are frequently modified in the past (i.e., *num_file_changes*) and the more files participated in previous dependent changes (i.e., *max_mod_file_dep_cha*), the more likely a change is predicted as dependent by our first model. For example, the "507176" [15], which is a major migration of the version of Zuul (a gating OpenStack system) from version "2" to "3" required a large amount of infrastructure-related changes touching over 2,652 files. Hence, having such a major migration over one change is not trivial, as it has a higher repercussion that involves other dependent changes. However, we observe no impact on the amount of modified file types on the likelihood of a change being dependent. A similar pattern is indicated by source code (ft_src_code), configuration (*ft_config*), ci/build (*ft_build_ci*), and tests (*ft_tests*) files, while documentation files have less chances to be dependent.

As expected, the changes that are similar in terms of their respective descriptions and have common reviewers are more likely to be predicted as a dependent pair of changes according to the *Pair*-related features. We observe that the more a developer of the change's project (i.e., *n_common_reviewers*) reviews the other project/change, the more chances for dependencies. Pairs of changes that share a higher number of tokens (i.e., *num_shrd_desc_tkns*), or whose descriptions are textually and semantically similar (i.e., *msg_similarity_score*) are more likely

[15] https://review.opendev.org/c/openstack/openstack-zuul-jobs/+/507176

to be predicted as dependent, as shown in Table 3.7. Furthermore, the higher the amount of dependent changes that the developer has in the source project, the higher the likelihood of a dependency (*ratio_dep_chan_owner_source*).

Summary of RQ2

Changes with a higher number of deleted lines, in projects with higher interactions with other projects, and changes made by developers who have contributions to diverse projects are more likely to be predicted as dependent. A similar pair of changes is more likely to be predicted as dependent by our model. **Our results identify the most important features for predicting dependencies, where these features can be further explored by future work to enhance the management of dependencies.**

RQ3. How does the performance of agentic AI compare with that of our machine learning models?

Motivation: With the recent advances of Agentic-AI, we wish to evaluate their ability to search and identify the right dependent change and how such a performance compares to machine learning models. We compare Agentic-AI with machine learning models as two solutions that one can choose from (if Agentic-AI is as performant as machine learning) if she/he has access to large language agentic-AI resources.

Approach: To answer this research question, we first elaborate on how we design our agentic-AI solution, then how to evaluate its performance compared to machine learning models.

Agentic-AI solution: We define an agentic solution that takes a change as input to look for its dependency within the last 30 days (similar to the machine learning solution). Figure 3.12 shows an overview of our proposed Agentic-AI solution. Our agent is expected to define a strategy/plan to search for a dependency using the system and user prompts depicted in Figures 3.13 and 3.14. Such a plan is executed by the same agent using three search tools that we equip the agent with. The three tools search for prior changes using their description, changed source code, and file and directory names, respectively. The input of these tools is regex expressions that

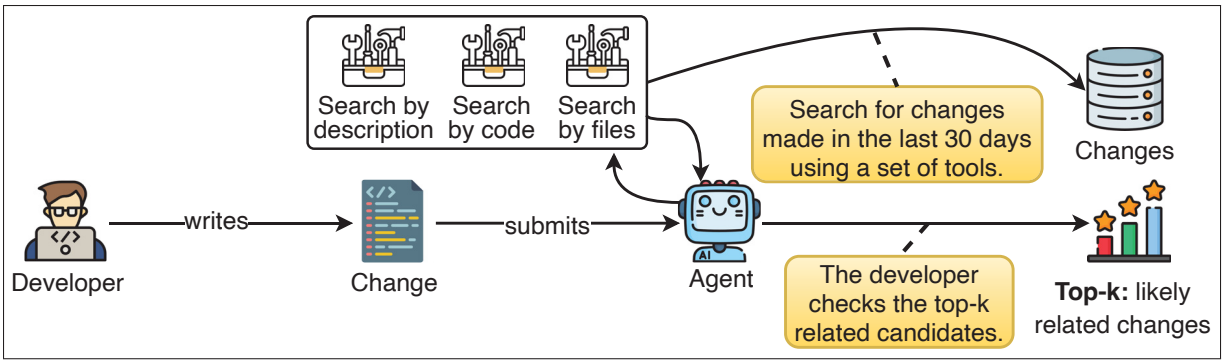


Figure 3.12 An overview of our Agentic-AI solution for finding dependent changes

the agent defines, and the result is the set of k changes that respect the regex expression. The k is by default 5 changes, but the Agent can decide to change that number. With the results of these tools, the agent reflects on their results and decides which of the changes depend on the input. In this paper, we use GPT-5.2 as an agent with reasoning efforts configured to be high and a maximum of ten iterations with the tools. We leverage the agentic-AI solution as the best setting that can leverage large language models. For instance, we cannot evaluate all possible pairs of changes using LLMs, which would not be cost-efficient. Using a simple RAG (Retrieval Augmented Generation) might not be efficient as such a technique is based on textual similarity, and dependent changes are not necessarily similar. However, with agentic-AI, we equip the model with search capabilities through tools that are based on regular expressions. Future studies are encouraged to extend our basic set of tools.

```
You are an expert on identifying dependent source code changes in the
↪ OpenStack software.

Your task is to analyze a specific code change (defined with a commit
↪ message, a set of changed files and the modifications to these files),
↪ and search with a set of given tools for its dependent changes made in
↪ the past.
```

```
# Definition of dependency

Two dependent changes are dependent if they rely on each other, indicating
↪  a relationship in which they are linked and cannot be treated as fully
↪  independent from one another. For example, two changes can depend on
↪  each other when:

1. A change A introduces parameters/features: Change B configures or
↪  enables them.
2. A change A removes support (e.g., Python 2.7): Change B updates
↪  tests/requirements to match.
3. A change A centralizes logic: Change B migrates other components to use
↪  that central logic.
4. A change A modifies an API: Change B implements tests for that API or
↪  updates consumers.
5. A change A deprecates a feature: Change B removes references to the
↪  deprecated feature.

# Available tools

You have access to the following tools that can support your search for
↪  changes that have a dependency on the change given in the input.

## 1. search_changes_using_commit_message_regex
Searches for changes where the commit message matches a regex pattern.
def search_changes_using_commit_message_regex(regex_patt: str, top_k: int =
↪  10) -> List[Dict]

## 2. search_changes_using_changed_code
Searches for changes where the code diff matches a regex pattern.
```

```
def search_changes_using_changed_code(pattern: str, top_k: int = 10) ->
↪  List[Dict]

## 3. get_related_changes_by_file_paths
Finds past changes that modified the same files or directories.
def get_related_changes_by_file_paths(changed_files: List[str], top_k: int
↪  = 10) -> List[Dict]

# Step-by-step Reasoning Process

### Step 1: Analyze the Source Change
- Examine the Commit Message (intent, motivation).
- Analyze Changed Files (architecture touchpoints).
- Analyze Code Diffs (logic additions/removals).

### Step 2: Retrieve Candidates
- Use the provided tools to fetch candidate changes.

### Step 3: Assess Potential Dependency
- Evaluate candidates against the user's provided change.

### Step 4: Assign Probability Score (0-100)
- **80-100**: Strong dependency (e.g., breaking change if missing).
- **50-79**: Moderate relationship (partial dependency).
- **20-49**: Weak/Indirect relationship.
- **0-19**: Unrelated and do not report it in the output.

# Constraint:
- If no dependencies exist, the output should be an empty list.
```

```
- Two semantically similar commit messages or changes are not necessarily
 ↪  dependent.

# Response format
Return **ONLY** a valid JSON array. Do not include markdown formatting,
 ↪  introductions, or conclusions.

**JSON Structure:**
[
    {
    ``number'': 12345, # Change that you identify as having a dependency
     ↪  with the given change.
    ``change_id'': ``Iabcdef123456'', # Change ID
    ``score'': 85,
    ``reason'': "Provide a one-two sentences about your reasoning to
     ↪  justify your decisions",
    }
]
```

Figure 3.13 The system prompt for the Agentic-AI solution

```
## Commit message:
${commit_message}

## Code diff:
${code_diff}
```

Figure 3.14 The user prompt for the Agentic-AI solution

Evaluation: We perform two evaluations that together simulate the two possible scenarios that could happen. A first one is to have a change with a dependency whose developer is looking for the right dependent change. The second case is a change with no dependency, expecting both solutions (machine learning and the agentic-AI) to return an empty list:

- We evaluate the ability of the model to find the right dependent change for changes that already have a dependency. We select a random sample of 378 changes with a dependency from the last testing fold and compare both the Agentic-AI solution and our second machine learning model on the same dataset. For the machine learning solution, we leverage the same approach discussed in the previous section which was performed in RQ1. We link each change to all the changes made in the last 30 days and that are predicted by the first model as dependent.
- We evaluate the ability of the model to report no dependency for changes that do not have one. As such, we simulate a user who does not have a dependency when using our machine learning models and when using the agentic-AI solution. For Agentic-AI, we expect the model to return an empty list. For the machine learning model, we expect our first model to exclude all changes without dependency. If the first model does not report a change as independent, we provide it to the second model and count the number of predicted changes by the second model. For the agentic-AI, we count the number of changes that it outputs. These false positives are wasted human resources that we wish to minimize.

Similarly to RQ1, we also evaluate both the machine learning-based solution and the agentic-AI on pairs of changes made by the same developer, pairs of changes made by different developers, and by combining both sets.

Results: Machine learning models outperform agentic-AI in terms of recall, while the opposite is observed for precision. As shown in Table 3.8 on the changes that have at least one dependency, we observe that the recall of the machine learning model reaches 75% on all the changes, while its agentic-AI counterpart only reports 37.40% of the cases. While for both approaches, the performance is driven by changes made by the same developer, the machine learning solution still reports more than half of the dependencies that are made by different

Table 3.8 The comparison between ML and Agentic-AI performances across developer settings (9th fold sample of dependent changes)

Model	Dev Type	Metric	k=1	k=3	k=5	k=7	k=10
XGBoost	All	Precision	34.64	19.11	13.27	10.42	8.12
		Recall	34.64	54.64	62.50	68.21	75
	Same	Precision	44.96	31.09	26.44	24.79	23.83
		Recall	44.96	69.75	76.05	80.25	85.71
	Different	Precision	15.91	9.85	6.82	6.17	5.91
		Recall	15.91	29.55	34.09	43.18	59.09
LLM	All	Precision	31.30	25.33	25.25	25.26	25.25
		Recall	31.30	36.34	37.14	37.40	37.40
	Same	Precision	55.28	48.49	48.52	48.57	48.57
		Recall	55.28	63.32	64.32	64.82	64.82
	Different	Precision	10.78	9.15	9.15	9.15	9.15
		Recall	10.78	11.76	11.76	11.76	11.76

developers (59%). For agentic-AI, such a recall is only 11.76%. However, the precision is better for agentic-AI. This can be explained by the fact that agentic-AI reports only a few changes, reducing the false positive rates.

While our machine learning-based solution is able to exclude more changes with no dependency, the Agentic-AI reports fewer false positive dependent source changes. We observe that our first model is able to exclude 84.92% of the changes with no dependency. Such a number is only 40.68% for the agentic-AI. These are the changes that are independent from other changes and for which the agentic-AI does not report any change. These are incorrectly predicted as dependent as seen as waste since developers would need to find their dependencies and inspect them. For that purpose, the agentic-AI is better as it reports only a median of two changes to inspect and a maximum of 12. The second model of our machine learning pipeline reports a median of 157 and up to 382 source changes predicted with a probability higher than 50%.

Upon examining a random sample of 20 false positive cases (pairs wrongly predicted as dependent), we find that LLMs tend to predict a dependency whenever a pair of changes modifies the same configuration names, job definitions, or Zuul templates, or even when they share identical commit messages. This behavior can be attributed to our definition of dependency

between two changes. For instance, the agentic-AI solution produced the following reasoning for the dependency between the pairs identified by 829228 (Source) and 830437 (Target): *"Touches the same scenario environment files and is also a DNM/NO MERGE-style test-only change, so it likely relates to the same CI gating/testing workflow around those files"*. Consequently, the agentic-AI solution frequently predicts pairs as dependent if they share common elements. Nonetheless, while these pairs do not constitute actual dependencies, they are contextually and semantically related.

Summary of RQ3

While we observe that agentic-AI has some abilities to search and find dependencies, its performance is still lower than our machine learning model. **We do not advocate for ignoring the agentic-AI in the dependency identification problem, especially with the rise of agents being software development teammates, but we suggest future studies improve our solution, e.g., by the inclusion of more tools, the improvement of the agent itself through fine-tuning, and the search space in which the agent is searching for a dependency.**

3.9 Discussion and Implications

In this section, we provide a set of practical insights and recommendations to practitioners, researchers, and tool builders derived from the findings of this work.

3.9.1 Practitioners

We recommend that practitioners leverage our proposed approach to effectively identify dependencies among software changes. In RQ1, we found that Random Forest is the best-performing machine learning classifier for predicting dependent changes and pairs of dependent changes, achieving an average AUC of 88.45% and 96.46%. Additionally, XGBoost demonstrated the best performance in terms of the Brier score, highlighting its reliability for probabilistic predictions. Given these findings, practitioners are encouraged to adopt our models,

particularly those utilizing Random Forest and XGBoost, to improve the efficiency and accuracy of dependency identification in software projects. Moreover, the effectiveness of Random Forest has been corroborated by a significant body of prior research, which has demonstrated its utility in various software engineering tasks, including log change suggestion Li, Shang, Zou & E. Hassan (2017), credit card fraud detection Aburbeian & Ashqar (2023), and software faults prediction Thomas & Kaliraj (2024). This aligns with our findings, further validating the robustness of this classifier in software change dependency detection. While we proposed a semi-automated approach consisting of two models, one might implement a pipeline where the first model's output would be the input to the second model. Such an approach drives the full automation of software change dependency detection.

We recommend that practitioners explore the contribution of features from different dimensions when training and evaluating models for software change dependency prediction, As observed in RQ1, utilizing features from all dimensions significantly enhances the performance of both models. For instance, when trained on features from a specific dimension, the first model achieves an average AUC ranging between 60.40% and 83.82%, while the second model achieves an average AUC ranging between 53.25% and 94.95%. However, when incorporating features from all dimensions, the first and second models achieve substantially higher AUCs of 88.45% and 96.46%, respectively. This demonstrates that features across different dimensions collectively contribute to the improved predictive performance of the models, reinforcing findings from prior work Chouchen *et al.* (2023). Therefore, practitioners should carefully consider the interplay and contribution of features from various dimensions to maximize model effectiveness. By doing so, they can derive deeper insights into the factors influencing software change dependencies.

We recommend that practitioners invest in specific features that serve as strong indicators of dependent software changes. Our analysis reveals that certain feature dimensions significantly influence the performance of dependency prediction models. For instance, the developer-related features have a substantial positive impact on both models, particularly in the case of the pair dimension for the second model. Specifically, developers with higher contributions

across multiple projects and dependent changes are more likely to be involved in changes with dependencies (as highlighted in RQ2). Additionally, changes with similar commit descriptions are more likely to be related, as they often share a common context or pertain to closely related features. Other critical features include recent project interactions, the presence of specific types of changes (e.g., *is_refactoring*), and developers' expertise levels. Practitioners should leverage these insights when designing tools for predicting software change dependencies. By focusing on these influential features, they can enhance the accuracy and utility of such tools, ultimately improving dependency management in complex software development environments.

3.9.2 Researchers

We recommend that researchers investigate software change dependencies in other large-scale software projects. While the findings of this study, derived from OpenStack, provide valuable insights, they are not directly generalizable to other software systems due to the unique characteristics of each project. Therefore, researchers are encouraged to explore software change dependencies in other large-scale projects to uncover potential variations and patterns specific to those contexts, as we observed (RQ1) a continuous increase in dependencies among changes for seven years in a row. Such investigations would contribute to a broader understanding of how dependencies manifest and evolve across different software ecosystems. Additionally, comparative studies across multiple software systems could provide deeper insights, enabling researchers to identify commonalities, differences, and factors influencing software change dependencies. These efforts could ultimately lead to more robust and universally applicable strategies for managing dependencies in large-scale software projects. A promising avenue for future research is applying our proposed models across multiple software systems to achieve a more comprehensive evaluation. For example, a model could be trained using data from one project and then evaluated using data from another project.

We recommend that researchers propose innovative tools for detecting software change dependencies. While the models proposed in this study demonstrate promising performance in predicting software change dependencies, they can serve as a baseline for future advancements.

Researchers are encouraged to build on our approach and develop novel solutions that further enhance accuracy and robustness in identifying dependencies. Additionally, future research could focus on predicting specific types of dependencies to address domain-specific challenges. For instance, identifying dependencies related to security vulnerabilities would be particularly valuable, given the critical importance of security in software projects Casola, De Benedictis, Mazzocca & Orbinato (2024). Even more, one may want to predict source code or database dependencies Aryani *et al.* (2011). As observed in PRQ3, dependencies can have different purposes such as testing. By targeting such specialized scenarios, researchers can contribute to more comprehensive dependency management solutions tailored to the unique needs of a software system.

We recommend that researchers explore additional feature dimensions to improve the prediction of software change dependencies. Our findings demonstrate that the proposed models achieve higher performance when leveraging features across multiple dimensions (RQ1). Notably, *developer*-related features significantly enhance the performance of both models, while *pair*-related features contribute substantially to the second model. Building on these insights, researchers are encouraged to investigate other feature dimensions that may be context-specific or project-dependent. By evaluating our models with alternative or supplementary feature sets tailored to the unique characteristics of different projects and the data available, researchers can uncover new opportunities to further refine dependency prediction models. This exploration could also lead to a deeper understanding of which feature dimensions are most influential in finding dependent changes.

We recommend that researchers investigate the factors contributing to prolonged dependency identification delays. In PRQ2, we found that approximately 50% of dependent change pairs are identified during code review, with delays spanning up to 12,791 hours. Notably, dependent changes exhibit a median lag of 57.12 hours and 463 intervening changes, suggesting substantial efforts to detect dependencies. Future work should examine how factors like team structure, module ownership, and system complexity could influence dependency identification

timing. Such a study could inform context-aware tooling to streamline dependency management in large-scale collaborative development.

We recommend that future studies improve our agentic-AI solution. While our agentic-AI solution does not show better results than the machine learning models, it already shows some abilities to perform correct predictions. With the advances of agentic-AI as teammates within software development pipelines, we encourage future studies to build on top of our approach. We suggest, for instance, extending the solution with other tools, improving the scope in which we search for the dependent changes, and the agent itself by fine-tuning it and improving the prompt. A combination of agentic-AI and machine learning can also be explored to improve the detection of dependencies among changes. Furthermore, our manual analysis of LLM failures indicates that the model often struggles when a pair of changes modifies the same elements, such as configuration names. Consequently, researchers could attempt iterative refinement of the system prompts (i.e., the definition of dependencies among changes) to improve the precision of predicted pairs. This refinement could incorporate additional resources, including, but not limited to, project specifications, software architecture diagrams, and technical documentation.

3.9.3 Tool Builders

We encourage tool builders to design innovative solutions leveraging the proposed models to facilitate the identification of software change dependencies. To maximize the utility of our models, we recommend that tool builders develop plugins or extensions that integrate seamlessly into existing development platforms, such as Gerrit. These tools should provide real-time predictions of software change dependencies at the moment a change is submitted for review, thereby assisting developers in proactively identifying and addressing dependencies. For example, a plugin for the Gerrit platform could analyze submitted changes and suggest potential dependencies based on our models' predictions. Such integration would not only enhance the review process by reducing the time and effort required to detect dependencies but also improve the overall efficiency of the development workflow. By embedding these capabilities into widely

used platforms, tool builders can significantly aid developers in managing dependencies and minimizing risks associated with undetected changes.

3.10 Threats to Validity

In this section, we identify and discuss the potential threats to validity in our study, including construct validity, internal validity, and external validity.

3.10.1 Construct Validity

A potential threat to construct validity arises from how we measure and define software change dependencies. In our study, dependencies are identified based on a set of predefined features, such as developer activity, commit descriptions, and project interactions. However, these features may not fully capture the complexity and diversity of all possible dependencies that exist in software development. There is a risk that the selected features may not adequately reflect all aspects of change dependencies, leading to incomplete or biased dependency identification.

To mitigate this threat, we based our feature set on established taxonomies and previous studies Chouchen *et al.* (2023), while we define a new set of features that fit our context. Nevertheless, future work could explore additional dimensions of dependencies, such as dependencies arising from external factors (e.g., third-party libraries, system configurations) or implicit dependencies based on developer behavior or collaboration patterns.

Another threat to construct validity is the reliance on “Depends-On” and “Needed-By” tags to identify software change dependencies. Such tags provide an explicit and interpretable form of documenting dependencies among changes; however, they may not capture the full spectrum of implicit or indirect relationships. Although OpenStack offers formal guidelines for using these tags (OpenStack Project Team Guide[16]), their application depends on developers’ discretion, potentially leading to incomplete identification of dependencies. Nevertheless, we focus on developer-annotated dependencies because they are actively and frequently used by OpenStack

[16] https://docs.openstack.org/project-team-guide/repository.html

developers, and we encourage future studies to identify additional ways of dependencies among changes.

Another important construct validity is the timing identification and the definition of missed dependencies. In fact, developers may retroactively annotate dependencies, omit them entirely, or apply them inconsistently across changes. Consequently, while these tags serve as a practical and reliable source for dependency identification, the inferred timing represents an estimated approximation of when a developer would typically find a dependency. Similarly, our identification of missed dependencies is based on whether the tag was added in the initial revision. However, developers may delay tagging intentionally as part of their workflow, rather than due to error or lack of awareness.

3.10.2 Internal Validity

A threat to internal validity is related to the manual classification that can be impacted by the annotators. To mitigate such risk, we first leverage the existing taxonomy of types of changes, the classification is performed by the first author, and the second and 3rd authors were involved to cross-check a subset of cases and achieved a substantial agreement (Cohen's Kappa score of 0.644).

Another potential internal threat to validity is related to sampling bias. Our selected sample might not be representative. However, the sample we study is large enough and similar to other qualitative studies Macklon *et al.* (2023); Bessghaier, Ouni, Sayagh, Chouchen & Mkaouer (2025). Our goal from the qualitative study is to identify existing categories of changes related to dependencies and, within the obtained categories, quantify the number of times developers forget to mention the dependency in their commit messages.

3.10.3 External Validity

A typical external threat to validity concerns the generalizability of our results. While we do not generalize our results to other systems similarly to other empirical studies, we focus on OpenStack

as a popular software system commonly used as the case study for other studies AlOmar *et al.* (2022); Bessghaier *et al.* (2023); Arabat & Sayagh (2024). As discussed previously, we also focus on OpenStack as it has a systematic way of identifying dependencies among changes.

Our second external threat to validity concerns the period frame of 30 days prior to a target change to construct pairs of changes for our second model. While choosing a different time frame could lead to different results, the 30-day period captures a substantial portion of dependencies. Further extending such a time frame will significantly impact the amount of required resources for training a model, which will make our approach less practical. That said, we recommend future studies to explore more time frame windows, up to covering all the possible prior changes to a target, to construct pairs of changes with their features.

3.11 Conclusion

Modern software systems can have a large number of contributors who concurrently submit their changes. While changes can break the integration and deployment pipeline, existing tools have been developed to better synchronize changes. Such synchronization is made through the identification of changes that depend on each other, such that a breaking change would be reduced alongside its dependencies. The identification of dependencies can be made through predefined tags (i.e., “depends-on” and “needed-by”) that are declared in the description of a change.

While a large body of prior work exists on software maintenance and dependencies, a few studies have looked at the dependencies among changes. We observe that such dependencies can occur for changes that are expected to have dependencies, such as configuration, to less likely categories to have a dependency, such as the changes that update the documentation. We also observe that such dependencies are prevalent, and practitioners often add them only during the code review or after a build failure. Even worse, one has to look into a large number of changes (a median of 463) that are made in a median period of 57.12 days.

Thus, we propose a semi-automated approach that developers can use our first model to classify changes that are likely to be dependent. After deciding on dependent changes, developers can use our second model to predict the exact pair of changes that depend on each other. Our models show high performances with an AUC of 88.45% and 96.46% for our first and second models, respectively. We find that our solution is able to find the right dependency for a given change within a list of 10 recommended changes in 67.19% of the evaluated cases. Further, a comparison of ML and agentic-AI on the last testing fold shows that the ML-based solution reaches a top-10 recall of 75%, while the agentic-AI shows only 37.40%. Their corresponding top-10 precision are 8.12% and 25.25%, respectively. The machine learning-based approach still outperforms the agentic-AI, suggesting future studies to improve the capabilities of agents in finding dependent changes. The interpretation of our models sheds light on the importance of the size of the change, the description content, the experience of developers, and the similarity between a pair of changes for predicting dependencies.

Data Availability Statement

The data and corresponding code used to replicate this work are publicly made available at the following link: https://github.com/aliarabat/change-predictor.

CHAPTER 4

EVALUATION OF MACHINE LEARNING MODELS IN MULTI-COMPONENT SOFTWARE SYSTEMS

In Chapter 3, we assessed the performance of our two-step machine learning pipeline and the agentic AI solution in predicting dependencies among software changes. Similarly, we evaluated the performance of these models across different developer settings: (i) when a pair is made by the same developer, (ii) when made by different developers, and (iii) when including all pairs. Consequently, this chapter complements the recommendation system proposed in Chapter 3 by evaluating the performance of our machine learning models in predicting dependencies within multi-component systems, specifically across *cross-component* and *within-component* settings. This chapter builds upon and extends the accepted research paper presented in Chapter 3.

RQ. What is the performance of our machine learning models in cross- and within-component prediction tasks?

Motivation: Our prior work Arabat & Sayagh (2024) indicated that approximately 10% of the studied OpenStack changes are cross-component dependent (i.e., a change C_1 in project A depends on another change C_2 in project B). Therefore, we would like to evaluate the ability of our second model in predicting within-component and cross-component dependent changes. In fact, when a pair of changes belongs to the same component, it is more likely that a developer can easily find the dependent changes. However, given large-scale software systems consisting of a large number of components, it becomes challenging to find the right dependent change among hundreds to thousands of changes per project.

Approach: To answer this research question, we use the same dataset from the experiments conducted in Chapter 3. Specifically, we split the testing data into two subsets based on whether the target change is cross- or within-project. Then, we report the performance of both settings alongside the overall performance when all changes are included.

Results: Our machine learning models perform well in both the cross-component and within-component prediction tasks, with slightly better performance in the cross-component

Table 4.1 The performance of thse second model using XGBoost, including AUC, Brier Score, top-k precision and top-k recall across within- and cross-component prediction settings

Setting	Metric	k=1	k=3	k=5	k=7	k=10	AUC	Brier
Within-component	Precision	24.37	16.03	11.44	9.01	6.63	96.64	0.253
	Recall	24.37	46.16	55.25	60.44	62.60		
Cross-component	Precision	32.97	17.93	12.84	10.05	7.73	96.39	0.324
	Recall	32.97	50.12	58.50	62.73	67.59		
All	Precision	32.29	17.75	12.72	9.97	7.65	96.46	0.312
	Recall	32.29	49.69	58.18	62.52	67.19		

setting, as shown in Table 4.1. We observe that the top-k precision for the within-component setting ranges between 24.37% for $k = 1$ and 6.63% for $k = 10$. Similarly, we observe that when considering the cross-component setting, the performance is slightly better, with a top-1 precision of 32.97% and a top-10 precision of 7.73%. Furthermore, we observe that the top-k recall ranges from 24.37% ($k = 1$) to 62.60% for within-component changes, while the cross-component changes show a top-k recall ranging from 32.97% to 67.59%. These results indicate that the models perform consistently regardless of the type of target change, be it cross-component or within-component.

Summary of RQ

Our machine learning models perform well across both cross-component and within-component settings. Hence, **practitioners can leverage our models to predict dependencies among software changes regardless of whether they are cross- or within-component.**

CHAPTER 5

ON THE TIME TO COMPLETE DEPENDABOT-SUGGESTED DEPENDENCY UPGRADES

Ali Arabat[1] , Mohammed Sayagh[1] , Sarah Nadi[2] , Artur Andrzejak[3]

[1] Department of Software Engineering and IT, École de Technologie Supérieure,
1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3
[2] Department of Computer Science, New York University Abu Dhabi,
Building A1, Office 177, P.O. Box 129188, Abu Dhabi, United Arab Emirates
[3] Department of Computer Science, Heidelberg University,
Im Neuenheimer Feld 205, Room 2/214, 69120 Heidelberg, Germany

Article submitted to the journal « Empirical Software Engineering » January 2026

5.1 Abstract

Modern software systems heavily rely on third-party dependencies, which evolve and publish frequent releases that introduce new features, fix bugs, and security vulnerabilities. When developers are expected to keep their dependencies up-to-date by regularly upgrading them to fully benefit from third-party reuse and promptly addressing bugs and security issues, prior work found that developers tend to delay their upgrades. As such, bots, such as Dependabot, are widely adopted to help developers maintain secure and up-to-date dependencies. Indeed, a prior work found that Dependabot helps developers accelerate the upgrades of their dependencies from an average of 48.99 days to only 25.38 days from the time a dependency's version is released to the time developers upgrade it in their projects. Dependabot proactively generates pull requests (PRs) whenever new dependency versions are available or when known security vulnerabilities are fixed. Prior studies examined the use of Dependabot and the time to accept or close PRs proposed by Dependabot. However, beyond the scope of the PR itself, a dependency flagged by Dependabot may still be upgraded in the codebase after the PR is closed. For instance, one can close multiple Dependabot PRs and perform the upgrades suggested in the PRs through a regular commit or delay the upgrade until the suggested upgrade gets superseded by newer versions. In addition, prior studies did not look into the relation between the upgrade time and other factors such as the content of the proposed upgrade and the Dependabot's configuration. In this paper,

we empirically re-evaluate the time to perform the upgrades suggested by Dependabot and study the association between the upgrade time and the client projects (i.e., how the project customized Dependabot), the content of the upgrade, and the dependency to upgrade. Our results show that upgrades take a median of 6.13 hours, which is 2 times longer than directly merged PRs, and 1.41 times the way it was studied by prior work. Yet, 84.70% of the projects indicate a median upgrade higher than one week, which is worse than overall median time, and that of prior work. We find changes to the Dependabot's configuration that are statistically significantly followed by faster or slower upgrades, among which is the grouping of dependencies to upgrade them at once, ignoring irrelevant, and the changing of the frequency of receiving upgrades from Dependabot. Moreover, development dependencies are typically upgraded faster than production dependencies, while major version upgrades take a significantly longer time than minor and patch version updates, even for upgrades with security-related changes. Our results indicate that critical changes such as security upgrades should be in minor versions, e.g., though patches, need more tests and validations for production dependencies, and need solutions on how to have project-customized Dependabot configuration through agentic-AI capabilities.

5.2 Introduction

Modern software systems heavily rely on third-party dependencies to deliver complex features Decan, Mens & Grosjean (2019). The rise of package management ecosystems has greatly simplified the integration of these dependencies, accelerating development cycles, and promoting code reuse Kula, German, Ouni, Ishio & Inoue (2018b); Latendresse, Mujahid, Costa & Shihab (2023). In fact, popular dependency package managers surrounding various programming languages host thousands to millions of community-created libraries. For instance, npm, the most popular package manager for JavaScript applications, contains over two million packages [1]. This reliance ultimately aims to reduce redundant effort and avoid "reinventing the wheel".

[1] https://www.npmjs.com

However, reliance on third party dependencies introduces notable maintenance challenges which require developers to continuously monitor and update dependencies to address security vulnerabilities and ensure compatibility with evolving requirements Mens & Decan (2024). Delaying upgrades can leave projects exposed to severe risks, as vulnerabilities in a single package may propagate through dependent systems Mens & Decan (2024). For example, in 2021, the widely used "Pac-solver" library Sharma (2021)—with more than three million weekly downloads— contained a critical flaw that allowed remote code execution within NodeJS applications. More recently, npm suffered one of its most serious attacks Henig & Hyde (2025) when impostors posing as the npm support team tricked a maintainer into revealing credentials, allowing them to inject malicious code into libraries downloaded over 2.6 billion times a week.

To mitigate these challenges, dependency management tools Dependabot (2025a); Renovate (2025); Snyk (2025) have been developed and are now widely adopted in software engineering practices. Prominent examples include Dependabot (2025a), Renovate (2025), and Snyk (2025), which automatically scans for outdated or vulnerable dependencies and generates pull requests (PRs) with recommended updates. These tools have become integral to modern DevSecOps workflows, as they substantially reduce the manual effort required to maintain dependencies and enhance software supply chain security by promptly addressing known vulnerabilities Carettoni & Trummer (2023).

Among these tools, Dependabot (2025a) is the most widely adopted due to its native integration with GitHub Rebatchi *et al.* (2024) and its support for several programming languages. Dependabot supports two primary update modes: security updates and version updates. In the security update mode, Dependabot cross-references project dependencies against the GitHub Advisory Database and automatically raises PRs whenever a library version used has fixes for known vulnerabilities Dependabot (2025b). In version update mode, developers configure a ".github/dependabot.yml" file to periodically check for the latest library versions and open PRs based on predefined rules Dependabot (2025c). Once submitted, Dependabot PRs can be merged, closed, or left pending for future actions.

While previous studies Alfadel *et al.* (2021); Rebatchi *et al.* (2024); He *et al.* (2023) examined how Dependabot is used and the time to accept or close its PR, they underestimate the time to update dependencies suggested by Dependabot due to their assumptions. Specifically, they only look into the time taken to merge the Dependabot PRs, while upgrades can occur beyond the scope of that PR and its merge. For example, developers may close a PR but later apply the suggested upgrade through a regular commit. Similarly, Dependabot may supersede an earlier PR with a newer version, where the older PR is closed, yet the dependency is still upgraded through a commit. As a result, the time to perform these upgrades is different from that of a single merged PR.

In this work, we revise the upgrade time, which we quantify as the period between Dependabot's suggestion and the actual completion of the upgrade in the project. In addition, we investigate the time to upgrade and its association with the three factors that define an upgrade, which are: **(1)** the client project and how it is configured to receive upgrades, **(2)** the content of the upgrade itself, and **(3)** the dependency to be upgraded. These associations were not explored by any of the previous studies.

In the context of this work, we conduct an empirical study on Dependabot dependency upgrades in a large set of JavaScript projects, an ecosystem that is both the most widely used on GitHub and examined in prior research Alfadel *et al.* (2021); Rebatchi *et al.* (2024); He *et al.* (2023); Mohayeji *et al.* (2025). We first characterize the lifecycle of Dependabot PRs, detailing the stages they undergo from their creation to their eventual integration or closure. Building on this foundation, we set out to re-evaluate the time to upgrade according to the literature through RQ1-2, while we build a deep understanding around upgrades through RQ3-5 as follows:

- **RQ1. How long does it take to perform the dependency upgrades that are suggested by Dependabot?** This RQ aims to quantify the time elapsed from when Dependabot opens a PR for a dependency to when that dependency is eventually upgraded in the codebase, including cases where the upgrade occurs after the PR is closed. Our results show that the upgrades studied take a median of 6.13 hours, which is 2 times longer than directly merged PRs. 16.91% of the studied upgrades are still performed after the Dependabot PR closure or

after being superseded, and exhibit a median of 30.41 days, which is 252 times longer than directly merged PRs (2.89 hours). Our qualitative analysis reveals that dependency upgrades whose Dependabot PR is closed are frequently upgraded with other dependencies.

- **RQ2. How does the time to upgrade vary across projects?** In this RQ, we study the prevalence of delayed upgrades across different projects to see if delaying upgrades is a niche problem affecting a few projects or a common problem that needs further attention. Our results indicate that 84.70% of the projects perform their upgrades after a median of one week, and 71.24% of the projects have at least 20% of the upgrades whose time to upgrade is higher than one week. Moreover, we observe an increasing trend in external upgrade over the first three years, followed by a gradual decrease onward.
- **RQ3. How do client projects configure Dependabot wrt., fast and slow upgrades?** This RQ examines the Dependabot configuration options that are associated with faster or slower upgrades by statistically comparing the time to upgrade dependencies before and after changing each Dependabot option. Our findings indicate 12 options that when changing them, the time to upgrade statistically changes. Among these options, developers configure Dependabot to avoid unstable or broken upgrades and regroup dependencies to upgrade together. However, we observe that certain options can be associated with faster upgrades for some cases, while the same options can be associated with slower upgrades in other cases. While the confounding variables have no impact on the upgrade speed, our multivariate models indicate that ignoring dependencies can help reduce the upgrade time.
- **RQ4. How does dependency upgrade time vary across different types of dependencies?** This RQ aims to determine the association between the types of dependencies and the time to upgrade them. We identify 25 types of dependencies using the TopicGPT framework, among which development dependencies (i.e.g, *linting*) are upgraded faster than production ones (e.g., *security tools*). This is even confirmed by comparing development and production dependencies in "package.json" file, which reveals a statistically significant difference.
- **RQ5. To what extent is the time to upgrade a dependency associated with the types of changes introduced in its new version?** This research question sheds light on the association between the upgrade time and the content of the upgrade itself across projects

that use Dependabot for *security* and security/version updates. Our findings reveal that delaying upgrades can be risky, as upgrades addressing security concerns take longer upgrade time than non-security upgrades and other types of changes. Further, major upgrades are significantly slower than minor and patch updates. That is, even when major upgrades contain security-related changes.

Our work concludes with a set of implications for researchers, practitioners, and Dependabot maintainers aimed at streamlining dependency upgrades. For instance, future research should consider that (i) multiple Dependabot PRs can participate in upgrading dependencies rather than relying solely on singles merged PRs as well as conducting the same analyses on the project level, as the time it takes is even worse than on the PR level (RQ1 and RQ2), (ii) help developers write project-customized by levering agentic-AI that is capable to learn from developer preferences and suggest appropriate configurations that keep up with the dependency upgrades (RQ3), (iii) researchers leverage our taxonomy of dependency types to examine the dependency management practices, (iv) propose solutions to support the risky upgrades, e.g., related to major versions and security fixes (RQ5). Furthermore, Dependabot could improve the way it proposes upgrades to developers. Dependency maintainers, in turn, should provide security fixes as patches rather than intermixing them with major releases.

We summarize the main contributions of this study as follows:

- To the best of our knowledge, this is the first study to follow Dependabot-suggested upgrades beyond the resolution of a single PR—through PR closure and chains of superseded PRs—to the point the dependency is actually upgraded in the codebase. While prior work Alfadel *et al.* (2021); Rebatchi *et al.* (2024) observed that most closed security PRs are superseded, it focused on security upgrades only and no studies tracked whether and when the underlying dependency is eventually upgraded.
- We are the first to study how the time to upgrade relates to the type of dependency and to the content of the upgrade itself (i.e., the kinds of changes introduced and the semantic versioning level of the new release). These dimensions have not been examined by prior Dependabot studies.

- We are the first to statistically associate individual changes to a project's Dependabot configuration with the resulting time to upgrade. While prior work Cogo & Hassan (2022); Renovate (2025) described how developers customize Dependabot (largely to reduce notifications), it did not relate these customizations to how fast or slow upgrades are subsequently performed.
- We provide a comprehensive replication package[2] containing all scripts and data, enabling future research to fully reproduce the experiments presented in this manuscript.

The remainder of this paper is organized as follows. Section 5.3 describes the background and related work. Section 5.4 describes our methodology. Section 5.5 re-evaluates the time to upgrades. Section 5.6 digs deeper into understanding the upgrades. Section 5.8 discusses implications derived from our findings. Section 5.9 outlines the threats to validity. Section 5.10 concludes the paper.

5.3 Background and Related Work

This section discusses the background and related work for our study. In particular, we discuss how developers can perform an upgrade suggested by Dependabot, how developers can configure Dependabot in their projects, and we discuss the closest studies to our paper on dependency upgrades and Dependabot.

5.3.1 Background

In this subsection, we present how an upgrade proposed by Dependabot can be performed, how we define the time to upgrade, and how developers configure Dependabot. The configuration is a background for our paper as we study how different configurations are associated with the time to upgrade dependencies.

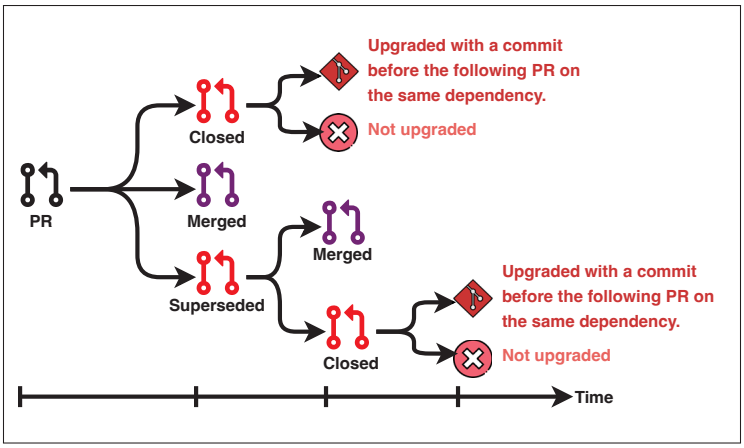


Figure 5.1 Dependabot PR lifecycle

5.3.1.1 Dependabot PR lifecycle

Depending on how Dependabot is configured (next subsection), Dependabot raises PRs for developers to upgrade a dependency whose new version addresses a security vulnerability Dependabot (2025b) or whenever a new version of a dependency is released Dependabot (2025c). A dependency upgrade suggested through a Dependabot PR can be performed either by merging the PR itself or after its closure, as shown in the different ways one can follow to perform an upgrade suggested by Dependabot, illustrated in Figure 5.1. In fact, the typical path explored by previous studies Alfadel *et al.* (2021); He *et al.* (2023); Rebatchi *et al.* (2024) is to directly merge the proposed upgrade. However, developers can still perform the upgrade manually through an ordinary commit after the PR closure. The same first dependabot PR can remain open until the dependency releases a new version. In such a case, dependabot supersedes the first PR by another PR that goes through the same cycle again. Either being directly merged, closed with the upgrade performed in an ordinary commit, or delayed until being superseded by a new Dependabot PR. These superseded PRs, according to Alfadel *et al.* (2021), represent 50.8% of the closed security-related Dependabot PRs.

[2] https://github.com/DaREf-MS/dependency-upgrades

```yaml
updates:
  - package-ecosystem: npm
    ...
    groups:
      production-dependencies:
        applies-to: version-updates
        dependency-type: production
        update-types:
          - minor
          - patch
        ...
      dev-dependencies:
        applies-to: version-updates
        dependency-type: development
        ...
```

Listing 1: A typical example of Dependabot configuration file

While prior studies looked at the time between the creation of a Dependabot and its merging time, we consider **the time to upgrade a dependency proposed by Dependabot as the time between the creation of the first Dependabot PR and the time at which the actual upgrade took place**. That is, either by merging the first PR, manually upgrading the dependency in a regular commit, after a chain of superseeding PRs whose last PR is merged, or the closing of the last PR in the chain with the upgrade performed manually.

5.3.1.2 Dependabot configuration

Dependabot provides a wide range of configuration options to customize the behavior of dependency updates through the ".github/dependabot.yml" file Cogo & Hassan (2022). For instance, developers can use the `groups` option to group multiple dependency updates under a single PR. Such an option typically simplifies the process of performing the upgrades and resolve any potential issues in one place, instead of multiple PRs. Additionally, the `schedule.interval` option can better help adjust the frequency of raising dependency updates, such as increasing the frequency from daily to weekly or monthly when a given project does not perform the upgrades promptly. Listing 1 demonstrates the use of the `groups` option, which groups production and development dependencies independently.

5.3.1.3 Types of npm dependencies

There are typically two types of dependencies within the npm ecosystem: (1) production dependencies and (2) development dependencies Latendresse *et al.* (2023). Production dependencies are those required at runtime when the application is bundled and deployed. They are defined within the `dependencies` object in the `package.json` file. In contrast, development dependencies are used only during the build or development phase. These dependencies are not shipped with the bundled application and, therefore, are not needed at runtime. They are declared under the `devDependencies` section of `package.json`. Furthermore, the resolved versions of `dependencies` and `devDependencies` appear in lock files such as `package-lock.json`, `yarn.lock`, or `pnpm-lock.yaml`. Typical examples of production dependencies include `express`, `react`, and `axios`, while development dependencies include `jest`, `eslint`, and `prettier`.

5.3.2 Related Work

The closest related research falls into two areas: (i) studies on dependency upgrades, and (ii) studies on Dependabot PRs. We review each of these areas below.

(i) Studies on dependency upgrades. Several studies have examined how developers manage and upgrade their dependencies Kula *et al.* (2018a); Cogo *et al.* (2021, 2022); Venturini *et al.* (2023); Weeraddana *et al.* (2024); Bi *et al.* (2025); He *et al.* (2025). Kula *et al.* (2018a) empirically investigated library migration of 2,700 dependencies encompassing 4,600 JavaScript repositories and reported that 81.5% of them still rely on outdated dependencies. Cogo *et al.* (2021) explored dependency downgrades in the npm ecosystem, revealing that 49% of the studied cases involved replacing acceptable version ranges with older versions. Later, Cogo *et al.* (2022) examined package and release deprecations in npm, finding that 3.7% of packages had at least one deprecated release (i.e., a release with obsolete features), and 66% of those had all of their releases deprecated. Venturini *et al.* (2023) examined how dependent packages are affected by breaking changes in the npm ecosystem. They found that nearly 12% of the studied

dependent packages were impacted when performing non-major upgrades. Weeraddana *et al.* (2024) studied CI build waste caused by updates to unused dependencies across more than 20K commits from 1,387 JavaScript projects. They observed that updates to unused dependencies account for 55.88% of the CI build time. Bi *et al.* (2025) investigated the prevalence of outdated dependencies in the npm ecosystem and reported that outdated packages are widespread across dependency chains, potentially affecting client projects. He *et al.* (2025) analyzed the impact of pinning dependency versions on security and found, counterintuitively, that this practice can increase a project's exposure to vulnerable dependency updates.

While these studies have explored various aspects of dependency management, such as downgrades and deprecations, our work differs by focusing on delayed dependency upgrades, which were suggested by Dependabot.

(ii) Studies on Dependabot PRs. Several researchers have empirically examined how developers use and respond to dependency update bots Alfadel *et al.* (2021); He *et al.* (2023); Rebatchi *et al.* (2024). Alfadel *et al.* (2021) conducted the first empirical study on Dependabot (then known as *dependabot-preview*, an older and now unmaintained version) to investigate how developers handle security-related Dependabot PRs. They found that 65.42% of security PRs were merged, while a substantial proportion (50.8%) of the closed security PRs were superseded. Similarly, He *et al.* (2023) studied how developers respond to Dependabot PRs and found that security-related PRs take longer for developers to merge, close, or respond (first human intervention), yet have a higher merge rate compared to regular PRs. Along the same line, Rebatchi *et al.* (2024) conducted a large-scale empirical study on security Dependabot PRs and found that they are often merged within one day, while most closed PRs were also due to superseding, consistent with the findings of Alfadel *et al.* (2021).

Other studies have focused on how vulnerabilities are resolved through Dependabot. Mohayeji *et al.* (2023, 2025) found that projects using CI/CD tools are more likely to accept security updates automatically, and that when these PRs are not merged, developers would manually resolve the vulnerabilities. Cogo & Hassan (2022) investigated how developers customize

Dependabot configurations and observed that developers frequently change options related to notification reduction. In line with this, Kula (2025) explored how AI techniques could help developers mitigate alert fatigue caused by the large volume of Dependabot notifications.

While prior work has offered valuable insights into how developers use Dependabot, it has notable limitations. First, existing studies have looked at the time to merge or close Dependabot PRs, while we look at the upgrades that were originally suggested by Dependabot. They also looked at the delays from two dimensions: (1) PRs, (2) GitHub actions, while our work primarily looks at the *delayed upgrades* and further investigates the following dimensions: (1) the changes made in the suggested upgrade, (2) the types of dependencies to upgrade, (3) the configuration of Dependabot, (4) and upgrades made within and after the PR closure.

5.3.3 Positioning of this study

Most existing research on Dependabot Alfadel *et al.* (2021); He *et al.* (2023); Rebatchi *et al.* (2024) has studied PRs in isolation, underestimating the actual time to perform the upgrades. Specifically, Alfadel *et al.* (2021) analyzed merged and closed security PRs individually, without considering that an upgrade can be performed through more than one PR. Similarly, He *et al.* (2023) examined the time to close and merge Dependabot PRs and the extent to which Dependabot can automate dependency upgrades. Further, Rebatchi *et al.* (2024) also focused largely on security Dependabot PRs to investigate their popularity, how developers deal with vulnerabilities, and their management.

Our study focuses on the re-evaluation of the time to upgrade. First, we consider that a dependency upgrade suggested by Dependabot may be addressed through one or more PRs (PR lifecycle vs atomic PR) as well as after PR closure. This perspective refines our understanding of the upgrade process, as treating PRs in isolation can lead to underestimating the actual time required to complete an upgrade. To the best of our knowledge, this study is the first to characterize these various upgrade paths (see Section 5.3.1.1), particularly those where the original PR is closed but the upgrade still occurs in the codebase. This shift in perspective

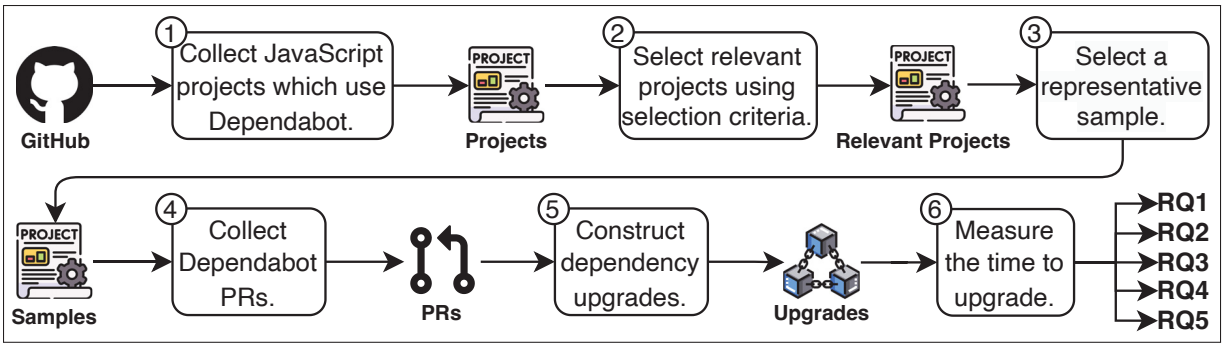


Figure 5.2 An overview of our methodology design to study delayed Dependabot dependency upgrades

from "PR-centric" to "Upgrade-centric" allows us to provide a more accurate and nuanced measurement of Dependabot's actual utility in real-world projects.

Another key contribution of our work is the exploration of upgrade time relative to the type of dependency, the Dependabot configuration changes, and the content of the upgrade.

5.4 Methodology

In this section, we describe the data collection process to answer our research questions. Our methodology consists of six main steps, as illustrated in Figure 5.2.

(i) Collect JavaScript projects that use Dependabot. We choose JavaScript as the focus of our work, because it is the most popular programming language Rebatchi *et al.* (2024); Mohayeji *et al.* (2025), which has also been the primary focus of several research studies Alfadel *et al.* (2021); He *et al.* (2023); Rebatchi *et al.* (2024); Mohayeji *et al.* (2025). Furthermore, Dependabot is widely used in JavaScript projects, as a prior study He *et al.* (2023) has shown that JavaScript dominates the studied Dependabot PRs. We begin by identifying JavaScript projects that leverage Dependabot to manage dependency upgrades. To do so, we search for PRs authored by Dependabot on JavaScript projects. We take this PR-based approach because GitHub does not provide a direct endpoint for listing projects that use Dependabot. Since GitHub API returns a maximum of 1,000 entries and we cannot crawl all the PRs in one shot, we collect

the PRs created by Dependabot day-by-day from May 23, 2019, to July 7, 2025. For example, we run a query that collects all the Dependabot PRs created on May 23rd, then all the PRs created on May 24th, and up to July 7th, 2025. Note that we were able to collect PRs using our tokens only until May 23rd, 2019, as GitHub imposes certain rate limits on token usage. From these collected PRs, we obtain 11,058 JavaScript projects that use Dependabot.

(ii) Select relevant projects. Next, we filter the real projects with enough history from GitHub by applying certain selection criteria inspired by prior work Alfadel *et al.* (2021); He *et al.* (2023) and extended with additional ones. Hence, we include projects that (i) are written in JavaScript (see the previous step), (ii) have at least 10 stars, (iii) contain at least 20 commits, (iv) have at least five contributors, (v) are not forks, archived, or template repositories, (vi) have at least six months of commit history, (vii) include at least one commit in the past three months (from the date of data collection), (viii) contain at least 10 Dependabot PRs, and (ix) are not toy or educational projects. To filter toy projects, we start with a seed list of keywords such as *tutorial*, *course*, and *academic*, iteratively refining it based on search results. After applying these criteria, we obtain 3,625 relevant repositories created between January 10, 2009, and November 15, 2024.

(iii) Select a representative sample. In this paper, we study a total of 1,044 projects which are obtained by selecting three representative samples as discussed in this paragraph. As we want to study a sample that better represent projects according to their popularity (i.e., number of stars), the amount of maintenance (i.e., number of commits), and the number of contributors (i.e., contributors), we split the data into two sets for each metric based on its median and we select a stratified representative sample (confidence level of 95% and margin error of 5%) of 348 projects. For instance, we split all the 3,625 projects based on the median number of stars into two groups: popular and non-popular projects. We select a representative sample of 348 projects, half of which come from the top category (above the median number of stars) and the other half from the bottom category (below the same median). We repeat the same selection based on the number of commits and a third selection based on the number of contributors. The

total number of selected projects after removing duplicates is 1,044 projects (a union of the three sets of 348 projects).

(iv) Collect Dependabot PRs. We then gather the full set of Dependabot PRs for each of the 1,044 projects selected in (iii). Using the GitHub GraphQL API, we query `is:pr repo:[REPO] author:app/dependabot`, specifying the repository name and filtering for PRs authored by Dependabot. We collect a total of 127,479 Dependabot-authored PRs. To select relevant PRs, we apply certain exclusion criteria similarly to the projects' selection. In particular, we select PRs that changed at least one npm specification file, including: "package.json", "yarn.lock", or "package-lock.json". That way, we limit our analysis to projects that use Dependabot to upgrade npm dependencies. This results in a total of 88,496 PRs involved in 758 JavaScript projects.

(v) Construct dependency upgrades. This step consists of constructing the upgrades from the Dependabot PRs collected in (iv). In this work, an upgrade is defined as the act of bumping a dependency from an older to a newer version (i.e., from "1.0.0" to "1.5.0"). Hence, we classify a PR or a chain of PRs as one upgrade depending on different upgrade pathways discussed in Section 5.3.1.1. As a result, we obtain 71,700 upgrades representing 88,496 PRs. Note that multiple PRs can target the same dependency, particularly in chains of superseded PRs. In such cases, these related PRs propose version bumps on top of the previous PRs. As a result, the number of PRs is higher than the number of unique upgrades. For instance, suppose PRs *A* and *B* propose updates to the dependency `X` from "1.0.0" to "1.3.0" and from "1.3.0" to "1.5.0", respectively. Although there are two PRs, they represent a single upgrade of the dependency, chaining from "1.0.0" to "1.5.0". In this case, only the last PR in the chain determines whether the dependency was upgraded, whereas all prior PRs are closed. Hence, this dataset is used to investigate delayed Dependabot dependency upgrades. Table 5.1 summarizes the key characteristics of the collected projects.

(vi) Measure the time to upgrade. This step consists of measuring the time to perform the upgrades that are suggested by Dependabot. We further explain how we measure the upgrade time for the four types of upgrades identified in Section 5.3.1.1.

Table 5.1 Statistics of the 758 studied JavaScript projects

Metric	Min.	Median	Mean	Max.
Commits	41	818	2,080.46	72,123
Age (days)	336	2,595.50	2,713.24	5,910
Dep. PRs	1	36	116.75	978
PRs	14	323.50	761.06	17,558
Stars	10	95.50	565.03	10,428
Contributors	5	22	55.22	2,761

- **Merged PR:** The Dependabot PR that suggests an upgrade is opened and then merged. We measure the time between the PR's creation and its merge.
- **Closed PR with external upgrade:** The PR is closed, but the dependency is updated directly in the codebase (after PR closure). To identify the corresponding commit, we search between the closure of the PR and the opening of the following Dependabot PR with the same dependency to upgrade, then select the commit that (1) modifies the same dependency and (2) updates it to a version that is equal to or newer than the version proposed in the PR. If there are multiple commits, we select the first one that satisfies the last two criteria. Here, we measure the time from the PR's creation to the commit that upgrades the dependency.
- **Merged superseded PR:** The original PR is superseded by one or more PRs, and the final PR in the chain is merged. In this case, we measure the time between the creation of the first PR and the merge of the last PR in the chain.
- **Closed superseded PR with external upgrade:** The original PR is superseded by one or more PRs, where the last PR in the chain is closed. The dependency is nonetheless upgraded in the codebase through a commit. We follow the same approach used in **Closed PR with external upgrade** to find the commit responsible for the upgrade. Hence, we measure the time from the creation of the first PR to the commit that has the upgrade.

5.5 Re-evaluation of upgrade time

In this section, we re-evaluate the time to perform the upgrades suggested by Dependabot according to the literature. Specifically, we first present the overall upgrade time, discuss the breakdown of different upgrade types, and manually examine whether an upgrade, suggested

by Dependabot, is still performed after PR closure (RQ1). Finally, we study the prevalence of delayed upgrades and the external upgrades across projects (RQ2).

RQ1. How long does it take to perform the dependency upgrades that are suggested by Dependabot?

Motivation: In this research question, our aim is to examine the time it takes for a dependency whose upgrade is suggested by Dependabot to be effectively upgraded. Prior work Alfadel *et al.* (2021); Rebatchi *et al.* (2024); He *et al.* (2023) has primarily quantified the time it takes for Dependabot PRs to be **merged**, rather than the overall time to **complete a dependency upgrade**. Investigating what happens to a dependency goes beyond what prior work has measured. Notably, prior work Alfadel *et al.* (2021); He *et al.* (2023) found that 50.8% of the closed Dependabot PRs were superseded by newer versions of the same dependency, and 65% of surveyed developers reported manually upgrading Dependabot PRs. Therefore, this research question re-evaluates the time to upgrade a dependency, to what extent such a time is aligned with the time to merge Dependabot PRs, studied by prior work Alfadel *et al.* (2021); He *et al.* (2023), and to examine whether a dependency is still upgraded elsewhere after its corresponding PR has been closed. Note that we do not claim that post-closure upgrades are necessarily motivated by Dependabot's suggestion. Instead, we aim to characterize what happens to dependencies whose upgrades are suggested by Dependabot after their associated PRs are closed.

Approach: To address this research question, we use 71,700 upgrades constructed in the data collection process that cover all the types of upgrading strategies along with their calculated upgrade time (Section 5.4).

Once we obtain the time-to-upgrade for each of the above categories, we conduct two analyses structured around the following dimensions:

1. We quantify the upgrade time of all upgrades and compare how such timing varies among the four upgrade types, including merged PRs and upgrades after PR closure/superseding.
2. We manually inspect the upgrades that were performed after PR closure to examine the dependencies suggested by Dependabot after their PRs' closure and determine whether they

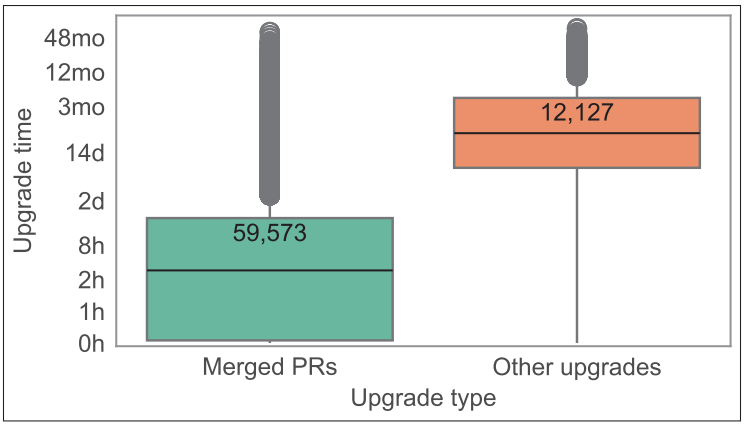


Figure 5.3 The time to upgrade *Merged PRs* and other upgrades (after PR closure or superseded)

were ultimately upgraded. For instance, are developers still interested in upgrades whose associated Dependabot PRs were closed?

To manually investigate what happens to upgrades performed after PR closure (i.e., through a commit), we select a random sample of 362 (confidence level of 95% and margin error of 5%) from the upgrades that were completed after PR closure. Each upgrade was manually analyzed to determine how the upgrades were handled by developers. Specifically, the first author inspects each upgrade by examining the commit messages, review comments, release notes, and any related commits, including the commit introducing the change. He then summarizes each upgrade in one or two sentences, describing the developers' changes, and then classify each case into a specific category based on these summaries following the card sorting technique Spencer (2009) and similarly to previous work Arabat & Sayagh (2024). Another author independently analyzed 200 of the 362 samples. The resulting Cohen's κ inter-rater reliability score is 0.605, indicating moderate agreement between the annotators McHugh (2012). Through multiple rounds of discussion, the annotators review the remaining disagreements and reach a final agreement score of 1.0.

Finding #1: The time from PR creation to the dependency being upgraded in the codebase shows a median of 6.13 hours, which is twice longer than directly merged PRs (the way

Dependabot is studied by prior work Alfadel *et al.* (2021)). Our analysis of Dependabot dependency upgrades, including *Merged PR*, *Closed PR with external upgrade*, *Merged superseded PR*, and *Closed superseded PR with external upgrade*-reveals that the median upgrade time across all cases is 0.26 days (6.13 hours). We find that for the majority of the studied projects (78.10%), the median upgrade time exceeds 6.13 hours. This observation confirms that, for most projects, upgrading a dependency takes longer than the overall median upgrade time. The median time per project ranges between 0 and 1,238 days, with a median of 3.74 days. Moreover, we also observe that dependencies upgraded through superseding or after PR closure require a median of 30.41 days, which is 252 times the duration to perform direct upgrades (a median of just 2.89 hours). Figure 5.3 shows the time it takes to upgrade merged PRs compared to upgrades that are made through an ordinary commit or after being superseded.

Table 5.2 The distribution of Dependabot dependency upgrades. μ denotes the median upgrade time in days

Label	**#**	**%**	μ
Merged PR	59,573	83.09	0.1
Closed PR with external upgrade	5,629	7.85	29
Merged superseded PR	4,971	6.93	21.9
Closed superseded PR with external upgrade	1,527	2.13	122
Total	**71,700**	**100**	-

Finding #2: 16.91% of the studied upgrades after closing Dependabot PRs (aka., through a commit) or after superseding the original PR by a newer version of the same dependency with a median upgrade time range of 21.9-122 days. Specifically, we find that 9.98% of the upgrades occur after closing Dependabot PRs. This type of upgrade can take two forms. The first one is when the first Dependabot PR for a dependency is closed (7.85%) and the upgrade is done via a commit, and shows a median upgrade time of 29 days. The second is when the first PR is superseded by one or more times before being closed (2.13%) and the dependency is upgraded through a commit with a median upgrade time of 122 days. Furthermore, we observe that 6.93% of the upgrades have one or more superseded PRs, while the last PR in the chain is merged within a median upgrade time of 21.9 days. The size of the chain of superseded PRs range between 2 and 85 PRs, with a median of 2. Table 5.2 highlights the breakdown of

frequency and upgrade time of different types of the upgrades studied. The first row shows the upgrade studied by prior work, while the other rows highlight upgrades that were made through an ordinary commit or after being superseded.

Table 5.3 Different changes that developers make after Dependabots PRs' closure

Change type	**#**	**%**
Grouping dependencies	161	44.48
Indirect dependency upgraded when the whole tree is regenerated	68	18.78
Upgraded by other bots	24	6.63
Upgraded in the new release	10	2.76
Required source code changes	8	2.21
Fix bugs	3	0.83
Upgraded to a newer version	2	0.56
Others	86	23.76

Finding #3: When upgrades occur after closing Dependabot PRs, dependencies are regrouped (44.48%), upgraded by other bots (6.63%), or upgraded as part of a new release (2.76%), as shown in Table 5.3. We observe a wide range of practices for performing dependency upgrades after closing Dependabot PRs.

Focusing on the first change type (*Grouping dependencies*), we find that developers often batch multiple dependencies within a single update. This typically occurs when developers combine different types of dependencies, including development, production, and those in locked files, or when the dependencies are development-related, production-related, or belong to the same family. Among these, the first cases are is the most common, which account for 49.06% (79 out of 161), followed by those belonging to the same family (31.05%), then development-related dependencies with 13.04%. For example, `eslint`[3] was upgraded together with `eslint-plugin-jsdoc` in the same commit[4], as both libraries belong to the same "eslint" family. Similarly, the `@eslint/js`[5] dependency was upgraded along with four other development dependencies in a

[3] https://github.com/msys2/setup-msys2/pull/515

[4] https://tinyurl.com/2s397jxy

[5] https://github.com/Chia-Network/cadt/pull/1153

single commit[6]. Additionally, `@babel/types`[7] was upgraded alongside another same-family dependency, `@babel/preset-env`, within the same commit[8].

The second most frequent change made after closing Dependabot PRs is the indirect upgrade of dependencies when the entire dependency tree is regenerated, accounting for 18.78% of the cases. In particular, when a top-level dependency (declared in the *dependencies* section of the package.json file) is upgraded, npm automatically regenerates the associated lock file (e.g., yarn.lock or pnpm-lock.yml), which may result in the upgrade of additional dependencies. For instance, a commit[9] in the *NullVoxPopuli/eslint-plugin-decorator-position* repository upgraded several dependencies in the package.json file, while the "semver-regex" dependency was automatically upgraded in the regenerated lock file (yarn.lock) from "3.1.3" to "3.1.4".

Another change made after closing Dependabot PRs involves other dependency-automation bots performing the upgrade, accounting for 6.63% of the manually analysed upgrades. In particular, 91.66% (22 out of 24) of the bot-initiated upgrades were performed by Renovate, while Depfu and Snyk were each responsible for only one upgrade. For example, Dependabot created a PR in the "archiverjs/node-archiver" project to bump `tar-stream` from "2.2.0" to "3.0.0". However, this dependency was instead upgraded by Renovate[10], leading the developer to close the Dependabot PR. Similarly, Snyk upgraded the `snyk` dependency from "1.465.0" to "1.667.0" in the "denisecase/node-express-mvc" project[11], whereas Depfu upgraded `eslint` from "9.14.0" to "9.15.0" in the "sealninja/react-grid-system" repository[12].

Another change (2.76%) is related to release cycles. Developers batch multiple upgrades as a release approaches. The 20 related PRs are generally opened a median of 1.96 days before the

[6] https://shorturl.at/Vb9VP

[7] https://github.com/mrhenry/core-web/pull/703

[8] https://shorturl.at/GNbKB

[9] https://tinyurl.com/4wakp5b9

[10] https://github.com/archiverjs/node-archiver/pull/633

[11] https://shorturl.at/ICjTg

[12] https://shorturl.at/X6yvl

release and are typically closed on the release day. For example, the `release-it` dependency in the repository *"elwayman02/ember-scroll-modifiers"* was upgraded alongside several other dependencies in both the manifest (`package.json`) and lock (`yarn.lock`) files as part of the project's "4.0.0" release[13].

Furthermore, 2.21% of the cases reveal that developers needed to modify source code-related files. Additionally, less frequent changes included fixing dependency-related issues (0.83%), or modifying the version operator (0.56%) or bump to a newer version than the one suggested by Dependabot. We could not determine the rationale for performing the upgrade externally to Dependabot for 23.76% of the cases.

Summary of RQ1

The studied upgrades show a median upgrade time of 6.13 hours, which is two times longer than directly merged PRs. 16.91% of such upgrades are performed after PR closure or after being superseded, which are upgraded after a median of 30.41 days compared to only 2.89 hours for merged PRs. When upgrades are made through commits, developers tend to group dependencies to upgrade, some are performed by other bots, other dependencies are upgraded through the automatic regeneration of the lock file.

RQ2. How does the time to upgrade vary across projects?

Motivation: The goal of this RQ is to investigate how common delayed upgrades are across projects. In particular, we want to better understand whether delayed upgrade is a common problem or is concentrated within a niche project, making it a less relevant problem.

Approach: To examine how common delayed upgrades are, we perform two primary analyses. The first analysis consists of measuring the upgrade time for each project. We then classify the projects based on their median upgrade time into one of the following categories: (i) Within 1 hour, (ii) Within 1 day, (iii) Within 1 week, and (iv) Beyond 1 hour. The second analysis quantifies the number of projects with upgrades that are made after PR closure. Precisely, we

[13] https://shorturl.at/osR86

quantify the percentage of external upgrades per project and accordingly report the distribution of projects falling into a percent range.

We manually investigate a random sample of 379 upgrades that take longer than one day to perform. We choose a one-day threshold to better understand the underlying reasons why developers defer upgrades to a later time.

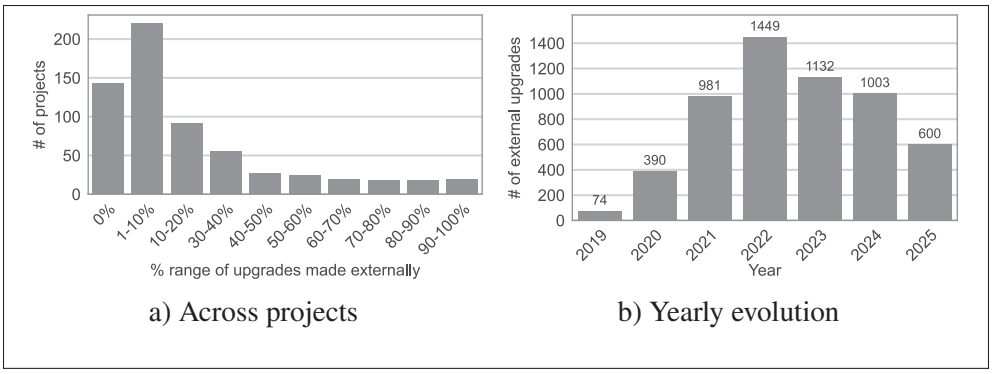
Finally, we study the upgrade time based on the repository type to see any differences in managing dependencies. To do so, we leverage the TopicGPT framework Pham, Hoyle, Sun, Resnik & Iyyer (2024) on the README.md files of the studied projects.

Table 5.4 Distribution of projects across median upgrade time intervals

Upgrade Interval	Definition	#	%
Within 1 hour	Median upgrade time of less than 1 hour.	17	2.24
Within 1 day	Median upgrade time greater than 1 hour and up to 1 day.	42	5.54
Within 1 week	Median upgrade time greater than 1 day and up to 1 week.	57	7.52
Beyond 1 week	Median upgrade time exceeding 1 week.	642	84.70

Finding #1: 84.70% (642 out of 758) of the studied projects have a median upgrade time exceeding one month, while only a small portion (2.24%) perform quick upgrades within a median of one hour. Overall, the investigated projects tend to delay their dependency upgrades rather than merging them promptly. Specifically, only 2.24% of the projects complete their upgrades within a median of one hour. Moreover, 5.54% of the projects upgrade their dependencies within a median of one day and 7.52% within one week. Notably, over one-third of the projects delay their upgrades for more than a median of one week. We also observe that 71.24% of the projects have at least 20% of the upgrades whose time to upgrade is higher than 1 week. Table 5.4 shows the distribution of projects along their median upgrade time interval.

Finding #2: Upgrades performed after PR closure are scattered across 81.13% of the studied projects. We observe that such upgrades appear in most projects to varying extents, as illustrated in Figure 5.4a. Specifically, 29.02% of the projects have between 1% and 10% of the upgrades performed after PR closure, while over 12.13% indicate more than 50% of their



a) Across projects b) Yearly evolution

Figure 5.4 The (a) frequency across projects and (b) yearly evolution of external dependency upgrades performed after closing Dependabot PRs

upgrades were made externally. In contrast, 18.86% of the studied projects do not exhibit any upgrades performed after PR closure.

Table 5.6 Summary of deferred upgrade reasons.

Reason	**Definition**	**#**	**%**
Inactivity	No developer activity for at least one week.	133	35.10
Lack of review approval	Upgrade was performed only after the a reviewer approved the changes.	91	24
CI failure	Deferred caused by build or test failures.	17	4.50
Fixing issues	Resolving related bugs or errors within the current or referenced PR.	3	0.80
Others	Deferral cause could not be clearly categorized.	135	35.60

An extensive analysis of a randomly selected representative sample of 379 deferred upgrades reveals that 35.10% lack developer activity for more than one week before the dependency is upgraded, as depicted in Table 5.6. Furthermore, we observe that 24% require reviewer approval, while 4.50% were deferred due to CI failures, and three cases involve fixing issues in the same or different PR. For instance, the `commander`[14] dependency was not merged until a second reviewer approved the changes made by Dependabot, ten days after the PR was opened. The remaining cases do not provide sufficient information regarding the reasons for their deferral.

[14] https://github.com/tldr-pages/tldr-lint/pull/181

We observe an increasing trend in the number of external upgrades occurring after closing Dependabot PRs between 2019 and 2022, followed by a gradual decrease thereafter. Starting in 2019, we note only a small number of external upgrades, as shown in Figure 5.4b. Such a number grows significantly over the following three years, peaking at 1,449 in 2022. From 2023 onward, they begin to decline, dropping from 1,132 in 2023 to 600 in 2025 (as of July, the date of our data collection).

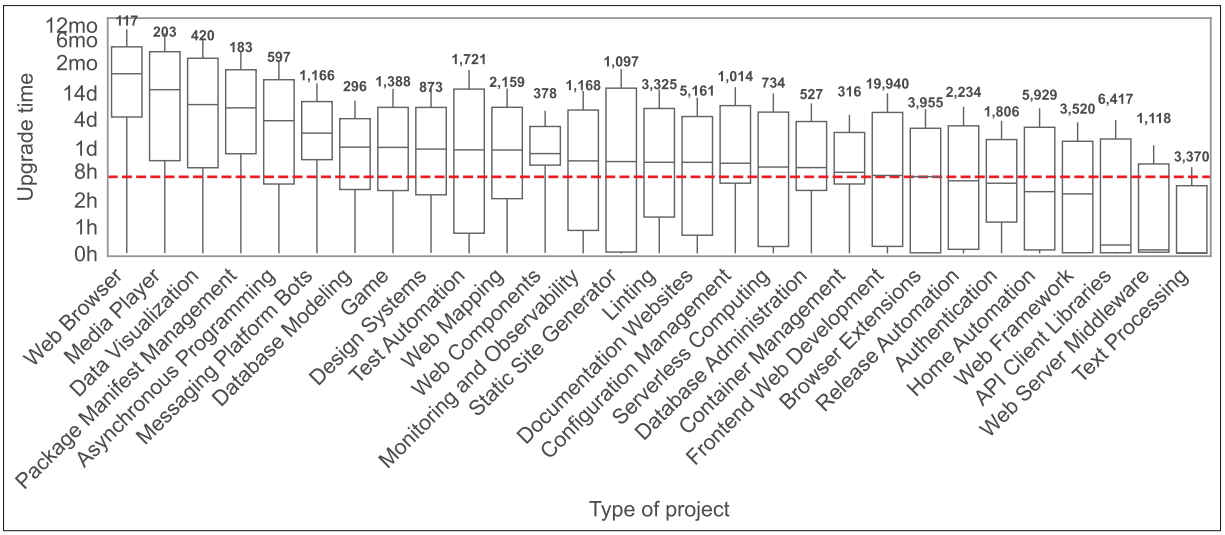


Figure 5.5 The time to upgrade per project type using the TopicGPT framework. The horizontal red line is the median

Finding #3: We identify 29 types of projects, with 21 (72.41%) out of these projects indicating a median upgrade time higher than the overall upgrade time, as illustrated by Figure 5.5. We observe that the identified types of projects using TopicGPT manage their dependencies differently and to varying degrees. In particular, projects dedicated to text processing, web server middleware, and API client libraries upgrade their dependencies rapidly, often in less than 30 minutes. For instance, the "Fdawgs/node-poppler" project (a PDF rendering utility classified under text processing) reveals a median upgrade time of 25.63 minutes. This efficiency is supported by GitHub Actions configured to review opened Dependabot PRs and automatically merge them once all checks pass successfully. Furthermore, the Browser Extension type shows an upgrade time comparable to the overall average at 6.13 hours. For categories ranging from front-end web development to web browsers, the median upgrade

time increases significantly, ranging between 6.48 and 841.22 hours. A typical example is the "AirenSoft/OvenPlayer" project (a JavaScript-based player), which upgrades its dependencies with a median of 24.62 days. These findings highlight the distinct strategies with which different project types manage their dependencies.

Summary of RQ2

Approximately three-quarters of the projects complete their upgrades within a median of one week or beyond, while externally performed upgrades occur in 81.13% of the studied projects. The external upgrades increased substantially over the first three years, showing a gradual decrease thereafter. Further, the identified project types manage their dependencies differently.

5.6 Understanding The Association between the Upgrade Time and Dependabot Configuration

In this section, we want to understand the upgrade time according to the client projects' configurations to Dependabot through RQ3.

RQ3. How do client projects configure Dependabot wrt., fast and slow upgrades?

Motivation: The goal of this research question is to study whether changing how Dependabot is configured can be associated with a faster or slower upgrade. Such an understanding can guide developers on potential options that can improve the freshness of their dependencies and motivate future studies on proposing solutions that guide practitioners on the configuration of Dependabot

Approach: To quantitatively and qualitatively investigate whether developers configure Dependabot to make the upgrades faster or slower, we focus on the 349 out of 758 of the studied repositories with a Dependabot configuration file (i.e., ".github/dependabot.yml"). We then mine the version history of these configuration files using Git, yielding a total of 1,244 changes across the 349 projects.

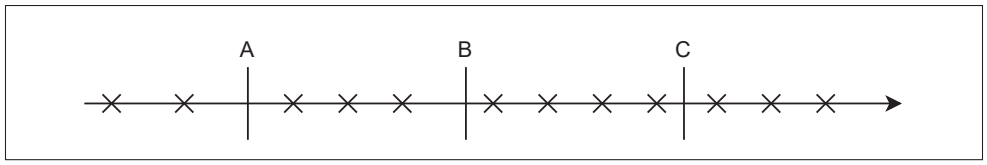


Figure 5.6 Illustration of how upgrade time is measured before and after a configuration change. The illustration shows configuration changes A, B, and C. Each configuration change has dependency upgrades made before and after the configuration change. The time to perform these upgrades is compared

Next, we compare the upgrade durations immediately before (starting from the last change, but before the current one) and after each modification to the configuration file. For instance and as illustrated in Figure 5.6, if a project has three changes to the Dependabot configuration file *A*, *B*, and *C*, we statistically compare (i) the time to upgrade dependencies before *A* with the time to upgrade dependencies between *A* and *B*, (ii) the time to upgrade dependencies between *A* and *B* with that between *B* and *C*, and (iii) the upgrade time between *B* and *C* with that after *C*. The number of upgrades made before the Dependabot configuration changes ranges between 1 and 29 (a median of 2), while the number of upgrades after the configuration change ranges between 1 and 645 (a median of 18).

To assess the statistical significance and magnitude of these differences, we apply the Mann-Whitney U test and calculate Cliff's Delta (along with its effect size) for upgrade times before and after each configuration change. We then focus on changes that show both a statistically significant difference in upgrade time and a medium or large effect size. This process yielded *108 and 115 Dependabot configuration changes* associated with significantly faster and slower dependency upgrades, respectively. We further use the same dataset (108 and 115 changes to Dependabot) to examine whether certain Dependabot changed options are associated with accelerated or slow upgrades.

In the first finding, we report the results of our quantitative analysis, while in each subsequent finding, we discuss the results of our qualitative analysis.

To control for confounding variables, we construct a multivariate logistic regression model. To isolate the impact of the configuration changes, we leverage the significance of the upgrade time (significantly slow vs. significantly fast) as our dependent variable and leverage these features as our confounding variables:

- **project_size**: The total lines of code in the project at the time of changing a Dependabot option.
- **dependency_count**: the number of dependencies in the package manager at the time of changing a Dependabot option.
- **team_size**: The number of active contributors in the last 90 days.
- **team_experience**: The average developer experience in days before the Dependabot changed option.
- **num_recent_commits**: The number of commits in the last 90 days of the Dependabot changed option.

To label the data, we assign a value of 1 to changed options associated with fast upgrades that are statistically significant (p-value<0.05) with a medium or large effect size. Similarly, we assign a value of 0 to changed options associated with slow upgrades that meet the same criteria of statistical significance and effect size. Furthermore, we exclude changed options that appear fewer than five times, as insufficient data points can introduce bias into the model's performance Peduzzi, Concato, Kemper, Holford & Feinstein (1996).

Note: We do not claim a causal relationship between option changes and time to upgrade. Instead, we report statistical associations that may inform developers about configuration options worth considering, based on empirical patterns observed in our study. Importantly, the same configuration change can be associated with both accelerated and slowed upgrades in different projects. These associations should therefore be interpreted strictly as empirical signals rather than deterministic effects: they support hypothesis generation about which options may influence upgrade time.

Table 5.8 Comparison of fast and slow actions that are made to the Dependabot configuration file

Action	Fast (#)	Slow (#)
Config file	31	17
schedule	20	21
open-pull-request-limit	10	15
ignore	12	10
Style	9	7
groups	5	5
versioning-strategy	2	6
target-branch	2	5
labels	1	3
assignees	1	2
reviewers	0	3
commit-message	1	1
package-ecoystem	0	2
Others	14	18

Finding #1: Our quantitative analysis reveals 12 changed options in the Dependabot file that are associated with either fast or slow upgrades, among which five change types are frequently modified across both categories. We find that configuring Dependabot for version updates, specifically when developers initially create the ".github/dependabot.yml" file, is the most common action in our dataset. This action is generally associated with accelerating the upgrades, with 31 instances associated with faster upgrades compared to 17 with slower upgrades (as shown in Table 5.8). We also observe that the `schedule`, `open-pull-requests-limit`, `ignore`, and `groups` options are associated with a similar frequency with fast and slow upgrades, ranging from 20-21, 10-15, 12-10, and 5-5 occurrences, respectively. Conversely, some options appear more frequently in slow upgrades. For example, `versioning-strategy` is changed six times in the slow category compared to twice in the fast one. We observe a similar trend for the `target-branch` option, which specifies the git branch to raise PRs for with 2 and 5 occurrences, respectively. Other options, such as `commit-message`, `package-ecosystem`, `reviewers`, `labels`, and `assignees` do not show strong associations with upgrade speed. However, we find 14 changes associated with fast upgrades and 18 changes with slow upgrades to the Dependabot file are not related to npm, such as changes to GitHub Actions dependencies.

Finding #2: We do not observe any significant differences in upgrade delay when the ignore option is changed. Such an option is used as a preventive step to avoid broken upgrades. Overall, changes to this option are sometimes associated with accelerated upgrades and other

Table 5.9 Distribution of Dependabot changed options that are associated with fast and slow upgrades

Option	Change	Fast (#)	Slow (#)
Config file	Adopt Dependabot	28	15
	Remove Dependabot	3	2
schedule	Increase interval (e.g., from daily to weekly)	12	17
	Add/set time	2	3
	Decrease interval	5	0
	Add/set timezone	1	1
open-pull-request-limit	Increase limit (e.g., from 5 to 20)	8	9
	Decrease limit	2	6
ignore	Unignore dependencies	2	5
	Ignore unstable dependencies/version	4	1
	Ignore ESM-related dependencies	2	2
	Ignore major version upgrades	3	0
	Ignore minor/patch version upgrades	1	1
	Ignore dependency updates	0	1
Style	Fix typos	9	7
groups	Group dev/prod dependencies	2	2
	Group matching name dependencies	1	3
	Group all dependencies	1	0
	Ungroup prod/dev dependencies	1	0
target-branch	Add/update/remove target-branch	2	5
versioning-strategy	Add/update versioning-strategy	2	6
labels	Add/update/remove labels	1	3
assignees	Add/update assignees	1	2
reviewers	Add/update/remove reviewers	0	3
commit-message	Add/update/remove commit-message	1	1
package-ecoystem	Add/remove updates for npm	0	2
Others	Others	14	18

named imports and supersedes older formats like CommonJS[15]. Because the client projects were not yet ready to adopt the new module system, these dependencies were temporarily ignored. We also identify two cases where dependencies were removed from the ignore list, typically because the dependency had become stable, prompting developers to upgrade. Other actions associated with the fast category include ignoring patch, minor, and major updates, ignoring dependencies that do not support version updates, and ignoring a given dependency's updates (i.e., all versions).

[15] https://shorturl.at/AEFee

For changes related to the *slow* category, we observe similar patterns: 2 related to ESM transitions, 1 case of ignoring unstable dependencies, 1 case of ignoring updates for specific dependencies, and 1 instance of ignoring minor/patch version updates. Additionally, we find five cases where dependencies were removed from the ignore list due to becoming stable or even removed entirely from the project manifest (package.json). Overall, this finding highlights that changes are commonly used as a preventive step to avoid issues or broken upgrades. Thus, the dependency is only upgraded when a stable version is released. Table 5.9 illustrates the distribution of the changed Dependabot options, how they are changed, along with their frequencies associated with fast and slow upgrades.

Finding #3: The same `interval` change is sometimes associated with faster upgrades and other times with slow ones. As such, the schedule needs to be adapted according to the project's specific characteristics. For changes associated with **faster** upgrades, we find that 12 out of 20 `schedule`-related changes *increase* this value, i.e., from "daily" to "weekly" or "monthly", or from "weekly" to "monthly". Only five cases involve *decreasing* the interval from "weekly" to "daily", and one case for adding time. For changes related to **slow** upgrades, 17 out of 21 of the `schedule`-related changes correspond to increasing the interval, 3 changes related to the `time` sub-option, and 1 for the timezone. One plausible explanation for increasing the `schedule.interval` option is to reduce the notification fatigue. For example, in the alphagov/govuk-frontend[16] project, Dependabot was initially configured to propose weekly dependency updates. However, developers found this frequency burdensome and increased the interval to monthly updates to better align with release preparations, as the developer stated: *"Weekly updates are becoming burdensome, so we're going to try updating monthly instead and folding Dependabot fixes into preparation for releases."*. This observation suggests that developers may need to adjust the schedule at which Dependabot raises PRs according to various factors, including developer availability, time flexibility, and release cycles.

Finding #4: The `open-pull-requests-limit` option is changed comparably across cases associated with fast and slow upgrades. We observe that 8 of the

[16] https://shorturl.at/LhL4v

`open-pull-requests-limit`-related changes that *increase* this limit are associated with accelerating the upgrades, while 9 are associated with fast upgrades. However, we observe changes that decrease this PR limit associated with slow upgrades outmber those of the fast ones (6 vs. 2). This observation suggests that developers may modify the limit option regardless of whether they intend to accelerate the upgrade process.

Finding #5: Changes to the `groups` option are not significantly associated with either slower or faster upgrades. However, we observe distinct patterns in how dependencies are grouped. For changes associated with slower upgrades, we find that developers may group dependencies based on matching names (e.g., grouping "typescript" with "@types*"[17]), which accounts for three cases and two cases group development/production dependencies. This practice is typically adopted when developers want to ensure production/development dependencies remain consistently up-to-date. For changes associated with faster upgrades, we observe that developers regroup dependencies based on their types (i.e., dev or prod) with five cases, matching name with one instance, or all dependencies with one case as well. We also observe one case related to ungrouping production/development dependencies. Overall, regrouping dependencies can be leveraged as a strategy to simplify maintenance and streamline upgrades, with no observed significant association with upgrade speedup.

Finding #6: We also observe other less frequently changed options associated with fast and slow upgrades. For instance, developers change how Dependabot handles the versioning in 2 and 6 cases related to fast and slow categories, respectively. They also tend to add, update, or remove `versioning-strategy`, `target-branch`, `labels`, `assignees`, `reviewers`, `commit-message`, and `package-ecoystem` with changes ranging between 1 and 6, as shown Table 5.9.

Our multivariate regression models reveal that the five confounding variables, namely team_experience ($p - value = 0.056$), project_size ($p - value = 0.061$), team_size ($p - value = 0.186$), dependency_count ($p - value = 0.265$), and num_recent_commits

[17] https://tinyurl.com/5n8vzder

($p - value = 0.563$), did not have a statistically significant impact on the change in upgrade speed. However, the models indicate that the **adoption of Dependabot** and the use of the **ignore** configuration are strongly associated with a reduction in upgrade delay, with p-values of 0.017 and 0.027, respectively.

Summary of RQ3

Our quantitative analysis identified 12 changed options to Dependabot that are associated with either significantly faster or slower upgrades. While certain options are associated with faster upgrades in some cases and with slower upgrades in others, developers primarily configure Dependabot to reduce notification fatigue, avoid unstable or broken upgrades, and regroup related dependencies.

5.7 Characteristics and Challenges of High-Risk Upgrades

In this section, we want to investigate the characteristics and challenges of upgrading dependencies through the types of dependencies (RQ4), and the upgrades' content, i.e., what has been changed between the proposed version and the one used in a given project (RQ5).

RQ4. How does dependency upgrade time vary across different types of dependencies?

Motivation: The goal of this research question is to examine the association between the types of dependencies and the time required to perform their upgrades. For instance, we aim to validate if risky dependencies (e.g., the ones used in production) are more likely to be upgraded faster than less urgent dependencies (e.g., development dependencies).

Approach: To investigate the types of dependencies being upgraded (i.e., whether a dependency is used for development or production), we classify the upgraded dependencies into meaningful categories (e.g., Compiler) based on their textual descriptions, mainly the repository descriptions provided on GitHub. We also check whether a dependency appears in the `dependencies` object or the `devDependencies` object of the `package.json`. Dependencies listed under

`dependencies` represent production dependencies, whereas those under `devDependencies` denote development-related dependencies.

To classify dependencies based on their descriptions, we adopt TopicGPT Pham *et al.* (2024), a framework designed to generate high-level topics from textual descriptions using Large Language Models (LLMs). We begin by retrieving the repository descriptions for all dependencies via the GitHub API, excluding entries without descriptions-these typically correspond to removed or undocumented repositories. We then prompt the OpenAI gpt-5-mini model to iteratively derive high-level topics using a four-step process: (1) generate an initial list of topics, (2) merge semantically similar topics, (3) assign the refined topics to the dependency descriptions, and (4) revise or adjust topic assignments when necessary. For a complete list of the system prompts employed in this study, please refer to the Appendix 5.11.2 To further improve accuracy, we provide the model with a set of few-shot examples mapping representative dependencies to their expected categories. This procedure results in 28 initial topics (see Appendix 5.11). After manually merging closely related topics (e.g., API and HTTP), we obtain a final set of 25 categories used in our analysis.

To evaluate classification quality, we manually compare a random sample of 100 labeled dependencies against their respective repository descriptions. Similarly, we compare the same sample to the npm package labels (e.g., keywords) to assess their alignment. For instance, if a project has the “security” term or a security tool name, then we map it to the “Security Tool” category generated by the TopicGPT framework. If no keyword is mentioned or differs from the TopicGPT category, we consider the TopicGPT category as a false label. The first experiment yields an accuracy of 93% (93 out of 100 classifications were correct), while the second experiment results in an accuracy of 86%. These results demonstrate that the TopicGPT framework Pham *et al.* (2024) achieves high classification performance. Additionally, we opt for the TopicGPT framework over the npm keywords for two main reasons: (1) labeling all dependencies in our dataset is a time-consuming process, and (2) TopicGPT systematically labels and refines the generated categories following four key steps, which is slightly challenging

to achieve for the npm keywords labeling, and has widely been used in the literature Piper & Wu (2025); Garcia Quevedo, Glaser & Verzat (2026).

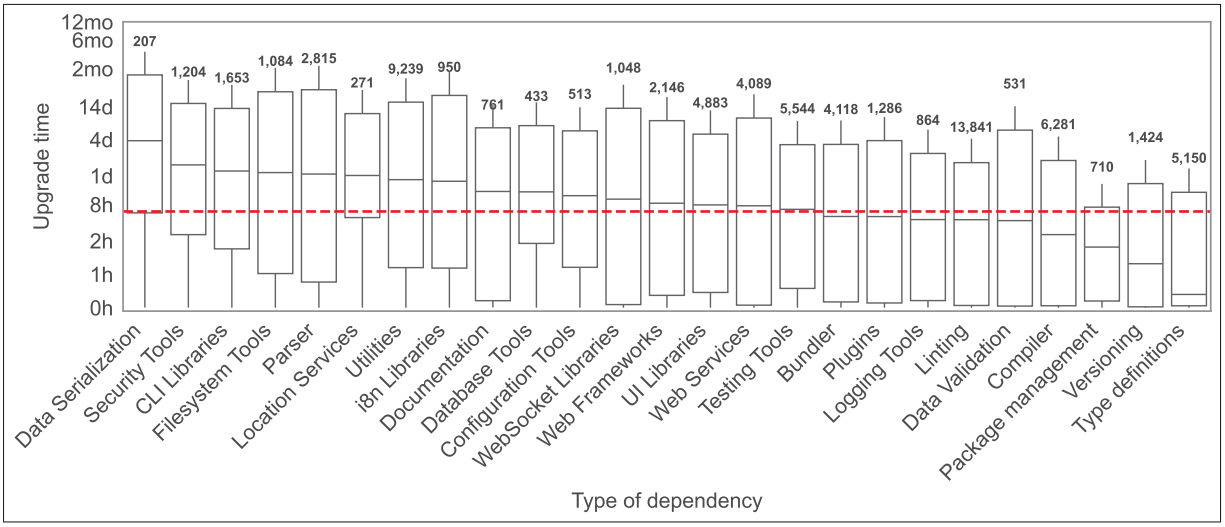


Figure 5.7 The time to upgrade different types of dependencies suggested by Dependabot, identified through the TopicGPT framework. The horizontal red line is the median

Finding #1: Dependencies mainly used in the development phase are being upgraded relatively faster than production ones. Our analysis reveals that the time required to upgrade npm dependencies varies substantially across different dependency types. In particular, dependencies actively used in production environments tend to require more time to upgrade compared to those primarily used during development (see Section 5.3.1.3), such as compilers and linters. Development-related dependencies, including *Bundler*, *Plugins*, *Linting*, *Data Validation*, *Compiler*, *Package management*, *Versioning*, and *Type definitions*, exhibit a median upgrade time shorter than the overall median (6.13 hours). This is also confirmed with the declaration of production and development dependencies in the package.json file. Particularly, a Mann-Whitney test confirms a statistically significant difference between production (a median of 13.78 hours) and development (a median of 4.65 hours) dependency upgrade times ($p\text{-}value < 0.001$), with a small effect size. An example of a development dependency is `babel/babel`, a widely used JavaScript compiler, which was upgraded 4,489 times with a median upgrade time of only 1.12 hours. In contrast, production dependencies tend to be updated more cautiously, especially *Data Serialization*, *Security Tools*, *File System Tools*, *Location*

Services, and *Utilities*, which reveal a median upgrade time between 20.72 and 91.81 hours. For instance, `indutny/elliptic`, the most upgraded dependency in the *Security Tools* category, which is a JavaScript implementation of the elliptic-curve cryptography, shows a median upgrade time of 93.65 hours. Exceptionally, *Data Serialization* (denotes libraries used to serialize and deserialize structured data like JSON and protobuf) indicates the highest median upgrade time of 91.81 hours. Figure 5.7 presents the upgrade time distribution across the 25 identified dependency types.

Summary of RQ4

Our findings reveal 25 different types of dependencies, out of which development-related dependencies (e.g., *linting*) are generally upgraded more quickly than production dependencies (e.g., *Security Tools*).

RQ5. To what extent is the time to upgrade a dependency associated with the types of changes introduced in its new version?

Motivation: The goal of this research question is to better understand the upgrades according to their content. We characterize the content of an upgrade along two dimensions: (1) the types of changes made in the new release and (2) the semantic versioning level of the new release. In particular, we investigate whether upgrades introducing high-risk changes (e.g., security vulnerabilities) are performed more rapidly than upgrades involving less critical changes, since delaying such upgrades exposes the dependent projects to potential security threats. We also shed light on whether security fixes and bug-fixing changes are released as patch versions. When such issues are found, maintainers typically issue urgent patch releases, which may influence how quickly these upgrades are adopted. Our results will provide insights into whether certain upgrades are more challenging (e.g., when security-related changes still take longer to integrate despite their importance) and whether semantic version updates (major, patch, and minor) are swiftly performed.

Approach: To study the types of changes in the upgrade's new release, we identify if there are any bugs, security concerns, new features, tests, or refactored code in the commits between

the original version and the suggested version of the dependency (commits that belong to the dependency). To do so, we adopt a keyword-search approach to systematically label each commit that was made in the proposed release following existing studies Batoun *et al.* (2023); Bosu, Carver, Hafiz, Hilley & Janni (2014); Zhou, Siow, Wang, Liu & Liu (2021); Sun *et al.* (2023). We follow the steps described below.

- **Extract upgrade metadata:** In this step, we leverage the title of the PR to extract the old and new versions of the dependency upgrade suggested by the Dependabot. We leverage the body to identify the repository name related to the dependency.
- **Retrieve commits:** We invoke GitHub API using the key metadata retrieved in the last step, including *old version*, *new version*, and *dependency repository name*. The response is a full list of commits with additional details that were made to the dependency between the two provided versions. We extract the messages of these commits. In case of a superseding chain, we only leverage the last merged PR or closed PR.
- **Label commits:** Since an upgrade can have one or more commits. We therefore assign one or more of the following labels: *security*, *bug*, *feature*, *refactoring*, *deprecate*, *test* to each upgrade. We search in the commits' descriptions for the presence of any keyword related to the previous labels (i.e., security). We adopt the labeling scheme proposed by Batoun *et al.* (2023), which includes bug keywords. To identify security-related commits, we employ security-specific keywords derived from prior work Bosu *et al.* (2014); Zhou *et al.* (2021); Sun *et al.* (2023). Bosu *et al.* (2014) and Zhou *et al.* (2021) developed keyword sets to identify security-related changes in code reviews and commits.

The second dimension examines how frequently each type of change is released as a *major*, *minor*, or *patch* update. For example, we count how many security-related changes are released under each versioning level. We also measure the time it takes to upgrade dependencies for each type of change across different semantic versions.

In each experiment, we analyze projects configured for (1) security updates only and (2) both security and version updates. This distinction allows us to determine whether significant upgrade

delays are unique to a specific configuration, such as security-only projects, or if they affect both groups equally.

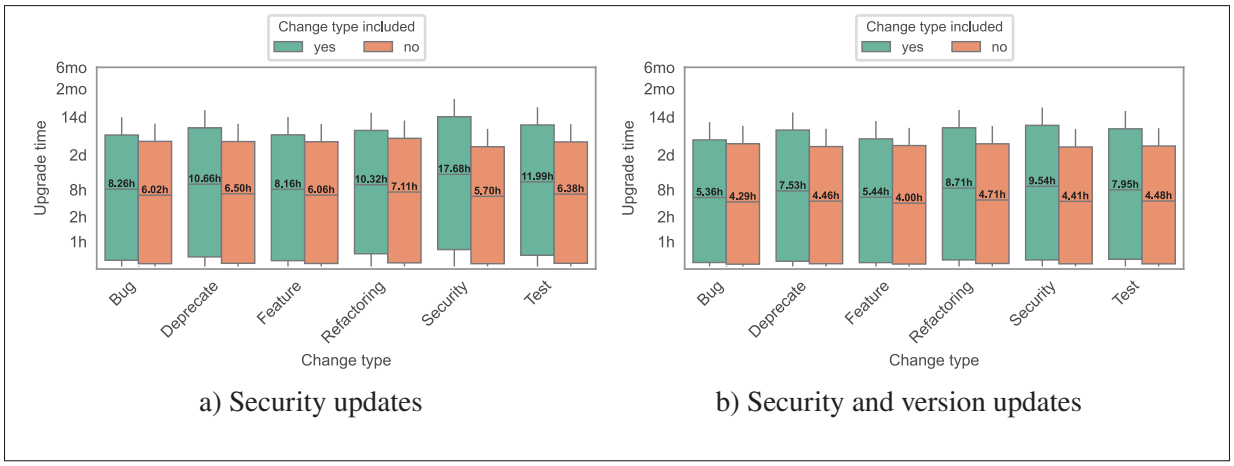


Figure 5.8 The upgrade time across the types of changes in the dependencies' commits for projects that use Dependabot for (a) **security** and (b) **security and version updates**

Finding #1: Upgrades addressing security concerns take longer than non-security upgrades, regardless of the Dependabot configuration usage. Particularly, we observe that, when projects configure Dependabot for security updates only, dependencies that include security-related updates exhibit a median upgrade time of 17.68 hours-approximately three times higher than upgrades with non-security changes, and take even longer than upgrades addressing other types of changes, such as bug fixes, new features, deprecated code removal, refactoring, or tests, as shown in Figure 5.8a. Similarly, we observe the same pattern across projects that configure Dependabot for security and version updates. For example, the security upgrades show a median of 9.54 hours, which is higher than upgrades without security updates (a median of 4.41 hours) and other change types (refactoring, being the second most delayed change type, indicates a median of 8.71 hours), as depicted in Figure 5.8b.

One possible explanation for this is the time needed for deeper analysis to ensure that no regressions are introduced. For instance, in the `ipfs/ipfs-desktop` repository, a developer responding to a Dependabot upgrade[18] noted that the update required deeper investigation and

[18] https://github.com/ipfs/ipfs-desktop/pull/2655

that their tests were not comprehensive enough. As such, the developer remarked: *"I think this requires deeper analysis [...] tests do not cover all features [...]"*. Following Bonferroni correction, the results reveal statistically significant differences ($p\_value < 0.01$) across all change types between the two project groups. However, Cliff's Delta values range from 0.034 to 0.106, representing negligible effect sizes in most instances. A notable exception occurs in security-configured projects, which exhibit a medium effect size for security-related change types.

While we expect projects configured for security-only updates to integrate security patches faster, we observe the opposite. Specifically, projects using Dependabot solely for security updates show a median delay of 17.68 hours to integrate security changes, compared to just 9.54 hours for projects using it for both version and security updates. A Mann-Whitney U test confirms this difference is statistically significant ($p - value \ll 0.01$), however, the effect size is negligible, as illustrated in Figure 5.8.

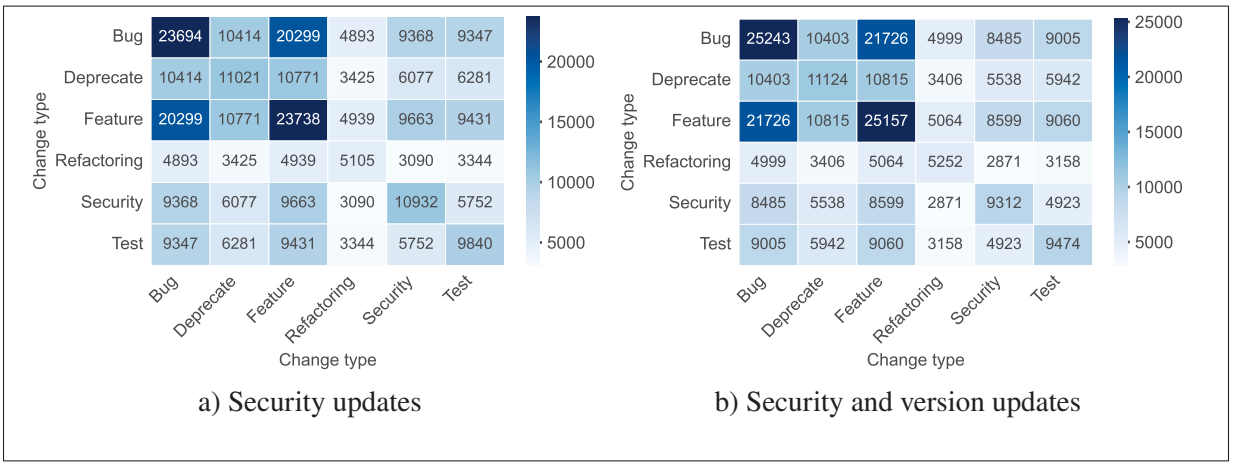


Figure 5.9 The co-occurrence of change types in the upgrades for projects that configure Dependabot for (a) **security** and (b) **security and version updates**

Finding #2: The studied upgrades highlight that security-related changes are often made alongside bug fixes and new feature implementations for both types of Dependabot-configured projects. For security-configured projects, we observe that upgrades addressing bugs in a new release frequently introduce new features (20,299 upgrades) and deprecate existing

code (10,414 upgrades), as shown in Figure 5.9a. This pattern is expected, as bug fixes are commonly integrated with feature enhancements, which in turn render certain code fragments obsolete. Security- and test-related changes also appear in roughly 9.3k upgrades that include bug fixes. Most upgrades containing deprecated code also introduce new features (10,771 upgrades), making this combination the most common among the studied change types. Security changes similarly tend to co-occur with new features (9,663 upgrades), at rates comparable to their co-occurrence with bug fixes. In contrast, security changes appear less frequently with deprecation and test changes (around 5–6k upgrades), and least often with refactoring (3,090 upgrades). Furthermore, refactoring is mainly observed alongside bug fixes and new feature implementations, while test changes most commonly accompany bug fixes and new features. We observe a similar trend across projects that are configured for security and version updates. Overall, the triplet of bug fixes, deprecated code, and new feature implementations represents the most frequent pattern among the examined upgrades, with security changes also predominantly occurring together with bug fixes and new features.

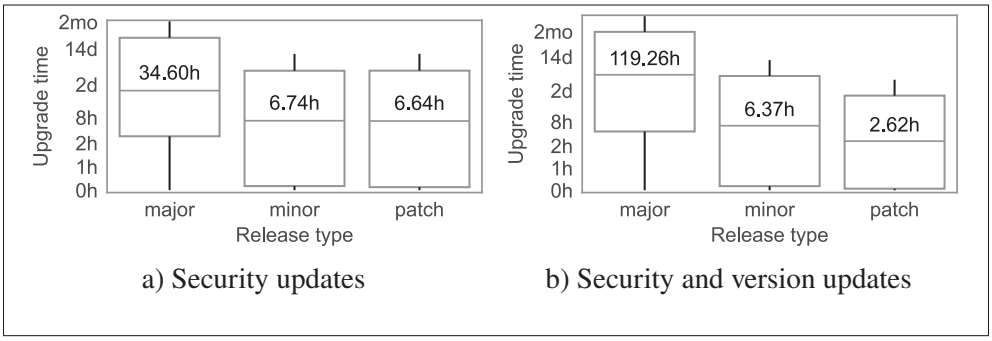


Figure 5.10 The upgrade time per release types when projects configure Dependabot for (a) **security** and (b) **security and version updates**

Finding #3: As expected, developers tend to delay major upgrades more often compared to patch and minor release updates across the two types of projects. We find that dependency updates are substantially more delayed when they involve major version changes, when projects are either configured for security or security and version updates, as illustrated in Figures 5.10a and 5.10b, respectively. Specifically, major upgrades require a median of 34.60 hours (> one day), approximately 5 times longer than minor and patch updates. This pattern suggests that

developers prioritize faster integration of minor and patch releases, which typically introduce new features or backward-compatible bug fixes, while exercising greater caution when upgrading to major versions that may require code refactoring or adaptation to breaking API changes. Additionally, we observe that upgrading major changes is even worse for projects that are configured for security and version updates, since the median time to perform major upgrades is 119.26 hours, the median time for minor updates is 6.37 hours, and patch updates is 2.62 hours. Furthermore, 10.68% of the security-configured projects and 17.15% of the security- and version-configured projects perform more than 10 major upgrades. These projects are typically popular and actively maintained. For instance, the *"openstreetmap/iD"* project (i.e., use Dependabot for security and version updates) exhibits the highest number of major version upgrades, totaling 80 (more than 16k commits), while *"codeceptjs/CodeceptJS"* (e.g., configured for security updates), an end-to-end Node.js testing framework with more than 3k commits, has 46 major version upgrades.

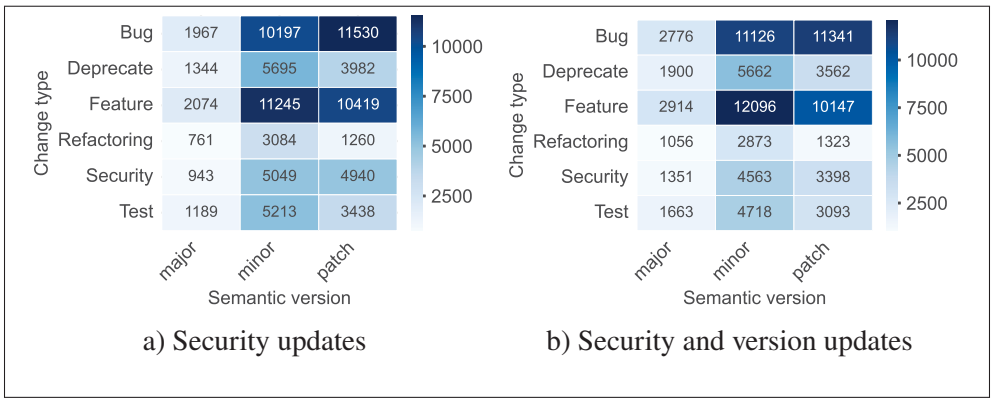


Figure 5.11 The co-occurrence of change types and semantic version levels in the upgrades for projects that configure Dependabot for (a) **security** and (b) **security and version updates**

Finding #4: 5 out of the 6 investigated change types are most often released as minor releases, except bugs as patch releases, while a non-trivial portion of all change types also appears in major releases. For security-configured projects, upgrades involving new features, deprecated code, tests, security fixes, and refactoring are predominantly released as minor version updates, accounting for 11,245, 5,696, 5,213, 5,049, respectively, as shown

in Figure 5.11a. Bug-related changes, however, are more frequently issued as patch releases (11,530) but still appear substantially in minor releases (10,197). This is expected, as bugs are typically fixed and shipped quickly as hot fixes. Despite this trend, a non-negligible number of changes are still shipped as major releases. In particular, security-related changes released as major updates require a median of 3.23 days to upgrade, while the median upgrade time for the remaining five change types ranges between 1.34 and 2.34 days. We observe a relatively similar behaviour in projects that are configured for security and version updates, as depicted in Figure 5.11b.

Summary of RQ5

Delaying upgrades can be risky, as dependencies involving security updates require approximately three times as much time to upgrade compared to non-security upgrades, as well as other types of changes. Developers typically upgrade major releases significantly slower than minor and patch updates, mainly when they are security-related. A similar trend is observed across security-configured and security-and-version-configured projects.

5.8 Discussion and Implications

This section outlines key implications from our findings for researchers, practitioners, and Dependabot maintainers to help developer quickly perform the upgrades. In addition, we clearly discuss how our own implications are positioned with the implication of prior studies on Dependabot Alfadel *et al.* (2021); He *et al.* (2023); Rebatchi *et al.* (2024).

Implication 1: Future studies must consider that an upgrade can be performed through more than one PR or regular commit rather than treating directly merged PRs only. While prior work He *et al.* (2023) found that developers delay upgrading dependencies, we find that the situation is more pronounced. As observed in RQ1, the overall upgrade time is double the duration of directly merged PRs the way it was studied by prior work He *et al.* (2023). Further, delaying upgrades is not a niche problem, as 78.10% of the projects have at least 20% of their upgrades delayed beyond one week (RQ2). Furthermore, while both Alfadel et al. Alfadel

et al. (2021) and Rebatchi et al. Rebatchi *et al.* (2024) recognized that a vast majority of closed Dependabot PRs (50.8% and 86%, respectively) are simply "superseded" by newer Dependabot PRs, they measured time strictly within the lifecycle of an individual PR and did not consider the complete PR lifecycle until the upgrade is performed. As such, we recommend that future work consider that an upgrade may occur through multiple Dependabot PRs rather than a single atomic PR, a distinction that this line of research has underestimated.

Implication 2: We recommend that developers promptly review dependency upgrades that require approval and that Dependabot notifies developers when upgrade PRs remain inactive for an extended period. Our qualitative analysis of deferred upgrades (RQ2) shows that a substantial proportion (24%) were delayed due to missing review approvals, and that more than one-third of these PRs experienced no activity for over one week before eventually being closed by either Dependabot or developers. Dependabot could help reduce prolonged inactivity by alerting developers about delayed upgrade PRs and encouraging timely action to keep dependencies current. To the best of our knowledge, no prior work has examined prolonged inactivity prior to the closure of Dependabot-generated upgrade PRs. Our results suggests a nudging mechanism similar to that of Maddila et al. Maddila *et al.* (2023) to be integrated in into the Dependabot to alert developers about delayed upgrade PRs and encourage timely actions to keep dependencies updated.

Implication 3: We suggest that Dependabot improve the way it proposes upgrades through better grouping. Our qualitative analysis of upgrades performed after Dependabot PR closure (RQ1) shows that in 44.48% of these cases, the dependency was eventually upgraded together with other dependencies. Our configuration analysis (RQ3) also shows that grouping is not uniformly associated with faster upgrades: changes to the *groups* option appear in both faster and slower upgrade cases. Therefore, the implication is not simply that Dependabot should group more dependencies together, as suggested by prior work on developers' desired bot features Alfadel *et al.* (2021); He *et al.* (2023); Rebatchi *et al.* (2024). Rather, Dependabot should better infer which dependencies should be grouped together, and under which project conditions such grouping is likely to facilitate rather than delay the upgrade. For example,

Dependabot could distinguish between dependencies that developers repeatedly upgrade together, dependencies from the same family, and upgrades that require source-code changes, instead of relying only on static grouping rules.

Implication 4: Future studies should support outcome-aware Dependabot configurations by identifying which project signals matter for faster upgrades. He et al. He *et al.* (2023) showed that developers frequently change Dependabot configuration options and Dependabot does not opperate as developers expect. They used these two observation to argue that dependency management bots should be self-adaptive. Our findings complement this perspective by examining what happens after such configuration changes. In RQ3, we do not observe a configuration change that is consistently associated with faster upgrades across projects. Instead, the same options can appear in both faster and slower upgrade cases. For instance, changes to *schedule*, *open-pull-request-limit*, *ignore*, and *groups* are associated with both accelerated and delayed upgrades.

We suggest that self-adaptability should not only rely on broad project context or developer preferences, but should be guided by empirical signals about upgrade outcomes. Our results provide such signals by highlighting configuration options and upgrade patterns that future work can inspect when recommending Dependabot settings. Future studies should investigate techniques that derive project-customized configurations from evidence such as past upgrade time, ignored dependencies, recurring manual upgrades, and dependencies that are co-upgraded. Agentic AI may support this process by generating candidate configurations from these project-specific signals, while keeping developers in control of the final decision Shin *et al.* (2025).

Implication 5: Dependency maintainers should release security fixes as isolated patch-level upgrades whenever possible. He et al. He *et al.* (2023) found that developers tend to rapidly merge security PRs when tests pass, while Rebatchi et al. Rebatchi *et al.* (2024) reported, based on developer survey responses, that major updates raise concerns because of breaking changes and the refactoring effort they may require. Our findings move beyond these PR-level and survey-based observations by showing how security fixes are actually packaged in dependency

releases. In RQ5, we find that security fixes often co-occur with other types of changes, such as bug fixes and new features, and that many security-related upgrades are released as minor or major versions rather than patches. We further find that major releases take longer to complete than minor then patch releases, regardless of whether the upgrade is security-related or not. These results suggest that security fixes may inherit the delay associated with broader and riskier releases when they are bundled with unrelated changes. Therefore, dependency maintainers should, whenever possible, decouple security fixes from new features, broad bug-fix releases, and breaking changes, and provide them as patch-level releases. Dependency management tools could also support this direction by identifying the smallest version change that contains the security fix, allowing developers to address vulnerabilities without unnecessarily adopting broader release changes.

5.9 Threats to Validity

This section discusses the internal and external threats to the validity of our study.

5.9.1 Internal Validity

A first internal threat to validity is the use of keyword-based commit annotation. Although this method may introduce bias, we mitigated it by deriving our keyword set from established, contextually relevant studies, such as security-related keywords proposed by Bosu *et al.* (2014); Zhou *et al.* (2021); Sun *et al.* (2023). Similarly, we leverage the keywords of Batoun *et al.* (2023) to identify the other change types (e.g., bug fixes).

A further threat concerns the interpretation of upgrades made after the closure of their associated Dependabot PRs. We cannot establish that a commit made after the closure of a Dependabot PR was directly motivated by the bot's suggestion, as developers may have acted independently. To mitigate this threat, we restrict the detection window to commits occurring between the PR's closure and the opening of the next Dependabot PR on the same dependency. Furthermore, we frame our goal as characterizing what happens to a dependency after its associated Dependabot

PR is closed, rather than claiming a causal link between the bot's suggestion and the subsequent upgrade.

5.9.2 External Validity

A key limitation regarding external validity is the extent to which our findings generalize beyond the npm ecosystem. Since our study focuses exclusively on JavaScript projects using Dependabot, generalizability to other ecosystems and dependency management tools is inherently constrained. That said, JavaScript is among the most widely adopted languages in both research and industry Rebatchi *et al.* (2024), and prior work Alfadel *et al.* (2021); Sun *et al.* (2023); Rebatchi *et al.* (2024) has already established npm as a well-studied context for Dependabot pull requests — making it a reasonable and representative starting point. Future replications across other languages and tools will be necessary to broaden the applicability of our conclusions.

Another threat to external validity is that we do not claim causality between Dependabot changed options and upgrade speed. Our aim is to provide hints into which options are changed in slow and fast upgrades. Therefore, the observed empirical patterns may inform developers about the configuration options worth considering with respect to upgrade speed, rather than options that actually speed up or slow down the upgrades.

5.10 Conclusion

In this paper, we conducted an empirical study to investigate the extent to which Dependabot dependency upgrades are delayed. Our analysis focused on JavaScript projects that adopt Dependabot as their dependency management tool. Our results have shown that the studied upgrades take a median of 6.13 hours, which is 1.41 times larger than the way it was studied by prior work (a median of 4.32 hours), and even worse at the project level, with 84.70% of the projects showing a median upgrade time higher than one week. We observe various Dependabot configurations that are statistically significantly followed by faster and slower upgrades, among which are regrouping dependencies, ignoring unmaintained ones, and changing the frequency of

receiving upgrades. Development dependencies are upgraded slightly faster than production ones, while major version upgrades take a huge amount of time compared to patch and minor version updates, with security-related changes indicating the largest delay. Our work concludes with a set of implications for improving the management of dependency upgrades. Solutions and tools are needed to assist practitioners in (1) performing upgrades promptly, (2) handling risky upgrades, and (3) helping developers write project-customized Dependabot configurations through agentic-AI capabilities that keep up with dependency upgrades. In turn, Dependabot could improve the way upgrades are proposed, and dependency maintainers should provide security fixes as patches; developers need to act on Dependabot PRs that require approval.

5.11 Appendix

This appendix shows a supplementary table and different system prompts for the TopicGPT framework.

5.11.1 TopicGPT Generated Topics

Table 5.11 The 28 high-level types of dependencies generated by the TopicGPT framework

Topic	**Definition**	**Library**
Utilities	The description describes a cross-platform wrapper around Node's child-process APIs (spawn/spawnSync), which matches Utilities' scope of general-purpose helpers and cross-platform abstractions.	moxystudio/node-cross-spawn
HTTP	The description describes an HTTP server that serves static files and implements HTTP features like Range requests and conditional GET handling, which are specific to the HTTP protocol (e.g., handling request/response headers and RFC behaviors).	pillarjs/send
Parser	This is a library whose primary purpose is parsing semantic-version strings for Node (i.e., converting version text into a structured representation), so it fits the Parser category.	npm/node-semver

Continued on next page

Table 5.11 – *Continued from previous page*

Topic	Definition	Library
API	Implements the Fetch Web API for Node.js — the description explicitly states it brings the Fetch API to Node.js.	node-fetch/node-fetch
CLI	The description identifies an "option parser" specifically for use with yargs, which directly pertains to parsing command-line arguments and building CLIs.	yargs/yargs-parser
Bundler	Matches the description stating the package is a web bundler; it directly aligns with bundling JavaScript and assets for web deployment.	evanw/esbuild
Testing	[1] Testing: The description identifies the project as a test runner for JavaScript, which places it in the Testing category.	karma-runner/karma
Compiler	The description explicitly identifies Babel as a compiler for writing next-generation JavaScript.	babel/babel
Frontend	The description refers to transforming styles using JavaScript plugins, which matches tools for processing and transforming styles for web applications.	postcss/postcss
Realtime communication	The description explicitly describes a WebSocket client and server for Node.js, which is a bidirectional, low-latency real-time communication technology.	websockets/ws
Serialization	Base64 is an encoding/decoding scheme for representing binary data as text, which fits under serialization tools for encoding/decoding data representations.	dankogai/js-base64
Runtime environment	"Standard Library" most closely aligns with runtime-provided APIs and global objects (i.e., the standard library of a JavaScript runtime) rather than a third-party utility or tooling package.	zloirock/core-js
Web framework	The description explicitly identifies the project as a Vue component framework, which aligns with "provides a lightweight foundation for building web applications" — it's a framework for building UI/components with Vue.	vuetifyjs/vuetify
Filesystem	The description states this provides filesystem bindings for tar-stream, matching the topic for file system access and bindings and integrations with archive/stream libraries.	mafintosh/tar-fs

Continued on next page

Table 5.11 – *Continued from previous page*

Topic	Definition	Library
Security	The project is an HTML sanitization tool that cleans user-submitted HTML and preserves only whitelisted elements/attributes, matching the Security category for sanitizing user input to prevent XSS/injection.	apostrophecms/sanitize-html
Internationalization	Timezone support is part of locale/globalization features (date/time handling) which falls under internationalization.	moment/moment-timezone
Linting	This tool's purpose is to run code-style and quality checks (formatters and linters) on files staged for commit, which directly relates to linting workflows.	lint-staged/lint-staged
Database	The description identifies the package as the MongoDB client driver for Node.js, which fits the "Database" category for libraries/drivers that connect to and interact with databases.	mongodb/node-mongodb-native
Logging	The Countly SDK collects and records product analytics events from websites/web applications and sends them to a remote analytics service, which aligns with libraries that record application events and diagnostics (i.e., logging/telemetry).	Countly/countly-sdk-web
Documentation	The project generates API documentation specifically for JavaScript code, matching the Documentation topic which covers "tools for generating and publishing API and developer documentation".	jsdoc/jsdoc
Geolocation	This is an IP-to-location (GeoIP) library for Node.js implementing MaxMind's GeoIP API, so it fits the Geolocation category.	geoip-lite/node-geoip
Configuration	The description explicitly states the module is for programmatic interaction with an nginx config file, matching tools for reading/parsing/modifying server configuration files.	tmont/nginx-conf
Release management	Matches tools for automating releases — version bumping and changelog generation using semantic versioning and Conventional Commits.	conventional-changelog/standard-version
Module loader	The description is about analyzing and producing graphs of module imports/dependencies (CommonJS, AMD, ES6), which relates to resolving and managing module imports and their dependency relationships.	pahen/madge
Type definitions	The description states the repository provides TypeScript type definitions, matching the "Type definitions" topic for declaration files and typings.	DefinitelyTyped/DefinitelyTyped

Continued on next page

Table 5.11 – *Continued from previous page*

Topic	Definition	Library
Plugins	This is an extension/add-on that expands the assets of an existing icon set (Maki), which fits the "Plugins" category for extensions that add or modify functionality of a host project.	rapideditor/ temaki
Validation	This is a library for verifying data against JSON Schema and JSON Type Definition schemas, matching the Validation category for schema-based data verification.	ajv-validator/ajv
Package management	Lerna is primarily a tool for managing and publishing multiple packages from a single repository, which aligns with package management functionality.	lerna/lerna

5.11.2 System prompts

This section presents the various system prompts used to apply TopicGPT to the dependency descriptions (Section 5.11.2.1) and README.md files of the studied client projects (Section 5.11.2.2).

5.11.2.1 Dependency descriptions prompts

5.11.2.1.1 Topic generation

```
You will receive a document and a set of top-level topics from a topic
↪ hierarchy. Your task is to identify generalizable topics within the
↪ document that can act as top-level topics in the hierarchy. If any
↪ relevant topics are missing from the provided set, please add them.
↪ Otherwise, output the existing top-level topics as identified in the
↪ document.
```

```
[Top-level topics]
{Topics}

[Examples]
Example 1: Adding "[1] Web Framework (Node.js)"
Document:
Fast, unopinionated, minimalist web framework for node.

Your response:
[1] Web Framework: provides a lightweight foundation for building web
↪  applications and APIs.

Example 2: Duplicate "[1] Terminal/Console Utility", returning the existing
↪  topic
Document:
Regular expression for matching ANSI escape codes.

Your response:
[1] Terminal/Console Utility: handles ANSI escape codes for colored or
↪  styled console output.

[Instructions]
Step 1: Determine topics mentioned in the document.
- The topic labels must be as GENERALIZABLE as possible. They must not be
↪  document-specific.
- The topics must reflect a SINGLE topic instead of a combination of
↪  topics.
- The new topics must have a level number, a short general label, and a
↪  topic description.
- The topics must be broad enough to accommodate future subtopics.
```

```
Step 2: Perform ONE of the following operations:
1. If there are already duplicates or relevant topics in the hierarchy,
↪  output those topics and stop here.
2. If the document contains no topic, return "None".
3. Otherwise, add your topic as a top-level topic. Stop here and output the
↪  added topic(s). DO NOT add any additional levels.



[Document]
{Document}

Please ONLY return the relevant or modified topics at the top level in the
↪  hierarchy. Your response should be in the following format:
[Topic Level] Topic Label: Topic Description

Your response:
```

5.11.2.1.2 Topic refinement

```
You will receive a list of topics that belong to the same level of a topic
↪  hierarchy. Your task is to merge topics that are paraphrases or near
↪  duplicates of one another. Return "None" if no modification is needed.

Here are some examples:
[Example 1]
Topic List:
<pairs of similar topics>

Your response:
```

```
<topics being merged into an existing topic>

[Example 2]
<pairs of similar topics>

Your response:
<topics being merged into a new topic>

[Rules]
- Each line represents a topic, with a level indicator and a topic label.
- Perform the following operations as many times as needed:
    - Merge relevant topics into a single topic.
    - Do nothing and return "None" if no modification is needed.
- When merging, the output format should contain a level indicator, the
↪  updated label and description, followed by the original topics.



[Topic List]
{Topics}

Output the modification or "None" where appropriate. Do not output anything
↪  else.
[Your response]
```

5.11.2.1.3 Topic assignment

```
You will receive a document and a topic hierarchy. Assign the document to
↪  the most relevant topics the hierarchy. Then, output the topic labels,
↪  assignment reasoning and supporting quotes from the document. DO NOT
↪  make up new topics or quotes.

[Topic Hierarchy]
{tree}

[Examples]
Example 1: Assign "[1] HTTP Client Library" to the document
Document:
A light-weight module that brings the Fetch API to Node.js

Assignment:
[1] HTTP Client Library: Mentions HTTP related words ("...Fetch API...")

Example 2: Assigned "[1] Debugging" to the document
Document:
A tiny JavaScript debugging utility modelled after Node.js core's debugging
↪  technique. Works in Node.js and web browsers

Assignment:
[1] Debugging Utility: Mentions debugging as main functionality of the
↪  library ("...debugging utility...")

[Instructions]
1. Topic labels must be present in the provided topic hierarchy. You MUST
↪  NOT make up new topics.
2. The quote must be taken from the document. You MUST NOT make up quotes.
```

```
[Document]
{Document}

Double check that your assignment exists in the hierarchy!
Your response should be in the following format:
[Topic Level] Topic Label: Assignment reasoning (Supporting quote)

Your response:
```

5.11.2.1.4 Topic correction

```
You will receive a document and a topic hierarchy. Assign the document to
↪  the most relevant topics the hierarchy. Then, output the topic labels,
↪  assignment reasoning and supporting quotes from the document. DO NOT
↪  make up new topics or quotes.

[Topic Hierarchy]
{tree}

[Examples]
Example 1: Assign "[1] HTTP Client Library" to the document
Document:
A light-weight module that brings the Fetch API to Node.js

Assignment:
[1] HTTP Client Library: Mentions HTTP related words ("...Fetch API...")

Example 2: Assigned "[1] Debugging" to the document
Document:
```

```
A tiny JavaScript debugging utility modelled after Node.js core's debugging
↪  technique. Works in Node.js and web browsers

Assignment:
[1] Debugging Utility: Mentions debugging as main functionality of the
↪  library ("...debugging utility...")

[Instructions]
1. Topic labels must be present in the provided topic hierarchy. You MUST
↪  NOT make up new topics.
2. The quote must be taken from the document. You MUST NOT make up quotes.

[Document]
{Document}

{Message}

Double check that your assignment exists in the hierarchy!
Your response should be in the following format:
[Topic Level] Topic Label: Assignment reasoning (Supporting quote)

Your response:
```

5.11.2.2 Client projects prompts

5.11.2.2.1 Topic generation

```
You will receive a document and a set of top-level topics from a topic
↪ hierarchy. Your task is to identify generalizable topics within the
↪ document that can act as top-level topics in the hierarchy. If any
↪ relevant topics are missing from the provided set, please add them.
↪ Otherwise, output the existing top-level topics as identified in the
↪ document.

[Top-level topics]
{Topics}

[Examples]
Example 1: Adding "[1] Software Development (Code Editor)"
Document:
# Acode - Code Editor for Android

<p align="center">
  <img src='res/logo_1.png' width='250'>
</p>

[...]

## • Overview
```

```
Welcome to Acode Editor - a powerful and versatile code editing tool
 ↪  designed specifically for Android devices. Whether you're working on
 ↪  HTML, CSS, JavaScript, or other programming languages, Acode empowers
 ↪  you to code on-the-go with confidence.

## • Features

- Edit and create websites, and instantly preview them in a browser.
- Seamlessly modify source files for various languages like Python, Java,
 ↪  JavaScript, and more.
- Built-in javascript console
- Enjoy multi-language editing support with easy management tools.
- Enjoy a large collections of community plugins to enhance your coding
 ↪  experience.

## • Installation

You can get Acode Editor from popular platforms:

[...]

## • Project Structure

<pre>
Acode/
|
|- src/   - Core code and language files
|
|- www/   - Public documents, compiled files, and HTML templates
|
```

```
|- utils/ - CLI tools for building, string manipulation, and more
</pre>

## • Multi-language Support

Enhance Acode's capabilities by adding new languages easily. Just create a
↪  file with the language code (e.g., en-us for English) in [...]. Manage
↪  strings across languages effortlessly using utility commands:

## • Contributing & Building the Application

See [CONTRIBUTING.md](CONTRIBUTING.md) for detailed instructions.

## • Contributors

<a href="https://github.com/Acode-Foundation/Acode/graphs/contributors">
  <img src="https://contrib.rocks/image?repo=Acode-Foundation/Acode" />
</a>

## • Developing a Plugin for Acode

For comprehensive documentation on creating plugins for Acode Editor, visit
↪  the [repository](https://github.com/Acode-Foundation/acode-plugin).

For plugin development information, refer to: [Acode Plugin
↪  Documentation](https://docs.acode.app/)

## Star History

[<a href="https://star-history.com/#Acode-Foundation/Acode&Date">
```

```
 [...]
</a>]

Your response:
[1] Software Development (Code Editor): It is an Android-based application
 ↪  specifically built for writing and editing code (HTML, CSS, JavaScript,
 ↪  Python, etc.), with features like live preview and plugin support.
 ↪  These characteristics clearly place it in the software development
 ↪  domain as a developer tool.

Example 2: Duplicate "[1] Game (Online Multiplayer)", returning the
 ↪  existing topic
Document:
spyfall
=======

React + Firebase implementation for the card game
 ↪  [Spyfall](http://boardgamegeek.com/boardgame/166384/spyfall)

### Localizations
[...]

Access [crowdin's](https://crowdin.com/project/adrianocola-spyfall) page to
 ↪  request new translations or help translating existing ones.

### Running the project (locally)

- Install [node.js](https://nodejs.org/)
- Clone this project, enter the cloned folder and install dependencies with
 ↪  `npm install`
```

```
- Open a terminal and run the command `npm run emulator`
- Open another terminal and run the command `npm run dev-emulator`
- Access it at `http://localhost:5173`

### Running the project (deploy fo Firebase)

- Install [node.js](https://nodejs.org/)
- Clone this project, enter the cloned folder and install dependencies with
↪  `npm install`
- Login to firebase with `npx firebase login`
- Setup a new firebase project with the `npx firebase init` command and
↪  follow the instructions (select only the feature `database`. Use the
↪  default values for everything and don't overwrite anything)
- Create a new web app with the command `npx firebase apps:create WEB`. The
↪  command will output the created `App Id`
- Execute the command `npx firebase apps:sdkconfig WEB <created app id>` to
↪  get the complete app configuration
- Create a copy of the file `.env.sample` named `.env.development` and fill
↪  it with the firebase configuration. You don't need to fill all fields.
- Access your firebase project in the [firebase
↪  console](https://console.firebase.google.com/) and enable anonymous
↪  authentication (Authentication → Sign-in method → Anonymous)
- Deploy firebase database security rules `npx firebase deploy --only
↪  database`
- Run the project with `npm run dev` and access it at
↪  `http://localhost:5173`

Your response:
```

```
[1] Game (Online Multiplayer): The project is a digital implementation of
↪  the Spyfall card game using React and Firebase, designed to be played
↪  through a web interface. Its primary purpose is entertainment through
↪  gameplay, which categorizes it in the game domain.

Example 3: Add "[1] Logistics"
Document:
# EasyPost Node Client Library

[...]

EasyPost, the simple shipping solution. You can sign up for an account at
↪  <https://easypost.com>.

NOTE: This library is intended to be used in a backend Node service and not
↪  in a frontend Javascript project via the browser.

## Install

**NOTE:** If you are using @easypost/api prior to v5 and a version of Node
↪  less than 6.9, you will need to install and include a polyfill, such as
↪  `babel-polyfill`, and include it in your project:

You can alternatively download the various built assets from this project's
↪  [releases page](https://github.com/EasyPost/easypost-node/releases).

### Compatibility

#### v4 and Earlier
```

```
- `require('@easypost/api/easypost.8-lts.js')` (Node 8.9+)
- `require('@easypost/api/easypost.6-lts.js')` (Node 6.9+)
- `require('@easypost/api/easypost.legacy.js')` (Node 0.10+)

#### v5

If using @easypost/api v5, you can require the base project which is built
↪  on Node v10 - v16

#### v6

If using @easypost/api v6, you can require the base project which is built
↪  on Node v12+

### Note on ES6 Usage

You can import specific versions of the compiled code if you're using later
↪  versions of Node and using @easypost/api prior to v5.

#### timeout

Time in milliseconds that should fail requests.

#### baseUrl

Change the base URL that the API library uses. Useful if you proxy requests
from a frontend through a server.

#### useProxy
```

```
Disable using the API key. Useful if you proxy requests from a frontend
↪  through
a server.

#### superagentMiddleware

Function that takes `superagent` and returns `superagent`. Useful if you
↪  need
to wrap superagent in a function, such as many superagent libraries do.

#### requestMiddleware

Function that takes a superagent `request` and returns that request. Useful
↪  if
you need to hook into a request:

### HTTP Hooks

If you would need or want to view what requests are being made to the
↪  EasyPost API behind the scenes, you can provide functions to listen for
↪  the request and/or responses. This can be done with helper functions on
↪  the created client.

**Note:** You cannot edit or otherwise affect the requests using these
↪  hooks. They are purely for debugging or logging purposes.

## Documentation

API documentation can be found at: <https://docs.easypost.com>.
```

```
Library documentation can be found on the web at:
↪ <https://easypost.github.io/easypost-node/> or by building them locally
↪ via the `just docs` command.

Upgrading major versions of this project? Refer to the [Upgrade
↪ Guide](UPGRADE_GUIDE.md).

## Support

New features and bug fixes are released on the latest major release of this
↪ library. If you are on an older major release of this library, we
↪ recommend upgrading to the most recent release to take advantage of new
↪ features, bug fixes, and security patches. Older versions of this
↪ library will continue to work and be available as long as the API
↪ version they are tied to remains active; however, they will not receive
↪ updates and are considered EOL.

For additional support, see our [org-wide support
↪ policy](https://github.com/EasyPost/.github/blob/main/SUPPORT.md).

## Development

### Typescript Definitions

Starting with `v5.3.0`, this project has Typescript definitions included.

#### Typescript Exclusions

- We do not provide a DefinitelyTyped version of these definitions at this
↪ time
```

```
- Predefined packages (see [Carrier
↪  Metadata](https://docs.easypost.com/docs/carrier-metadata) in our docs
↪  for more details)
- Carrier service levels (see [Carrier
↪  Metadata](https://docs.easypost.com/docs/carrier-metadata) in our docs
↪  for more details)
- Carrier list (see [Carrier
↪  Types](https://docs.easypost.com/docs/carrier-types) in our docs for
↪  more details)

### Testing

The test suite in this project was specifically built to produce consistent
↪  results on every run, regardless of when they run or who is running
↪  them. This project uses [Pollyjs](https://github.com/Netflix/pollyjs)
↪  (AKA: VCR) to record and replay HTTP requests and responses via
↪  "cassettes". When the suite is run, the HTTP requests and responses for
↪  each test function will be saved to a cassette if they do not exist
↪  already and replayed from this saved file if they do, which saves the
↪  need to make live API calls on every test run. If you receive errors
↪  about a cassette expiring, delete and re-record the cassette to ensure
↪  the data is up-to-date.

**Sensitive Data:** We've made every attempt to include scrubbers for
↪  sensitive data when recording cassettes so that PII or sensitive info
↪  does not persist in version control; however, please ensure when
↪  recording or re-recording cassettes that prior to committing your
↪  changes, no PII or sensitive information gets persisted by inspecting
↪  the cassette.
```

```
**Making Changes:** If you make an addition to this project, the
 ↪  request/response will get recorded automatically for you. When making
 ↪  changes to this project, you'll need to re-record the associated
 ↪  cassette to force a new live API call for that test which will then
 ↪  record the request/response used on the next run.

**Test Data:** The test suite has been populated with various helpful
 ↪  fixtures that are available for use, each completely independent from a
 ↪  particular user **with the exception of the USPS carrier account ID**
 ↪  (see [Unit Test API Keys](#unit-test-api-keys) for more information)
 ↪  which has a fallback value of our internal testing user's ID. Some
 ↪  fixtures use hard-coded dates that may need to be incremented if
 ↪  cassettes get re-recorded (such as reports or pickups).

#### Unit Test API Keys

The following are required on every test run:

- `EASYPOST_TEST_API_KEY`
- `EASYPOST_PROD_API_KEY`

Some tests may require an EasyPost user with a particular set of enabled
 ↪  features such as a `Partner` user when creating referrals. We have
 ↪  attempted to call out these functions in their respective docstrings.
 ↪  The following are required when you need to re-record cassettes for
 ↪  applicable tests:

- `USPS_CARRIER_ACCOUNT_ID` (eg: one-call buying a shipment for
 ↪  non-EasyPost employees)
- `PARTNER_USER_PROD_API_KEY` (eg: creating a referral user)
```

```
- `REFERRAL_CUSTOMER_PROD_API_KEY` (eg: adding a credit card to a referral
↪  user)

Your response:
[1] Logistics: Provides tools and APIs for managing shipping operations,
↪  including creating, purchasing, and tracking shipments within backend
↪  systems.

[Instructions]
Step 1: Determine topics mentioned in the document.
- The topic labels must be as GENERALIZABLE as possible. They must not be
↪  document-specific.
- The topics must reflect a SINGLE topic instead of a combination of
↪  topics.
- The new topics must have a level number, a short general label, and a
↪  topic description.
- The topics must be broad enough to accommodate future subtopics.
Step 2: Perform ONE of the following operations:
1. If there are already duplicates or relevant topics in the hierarchy,
↪  output those topics and stop here.
2. If the document contains no topic, return "None".
3. Otherwise, add your topic as a top-level topic. Stop here and output the
↪  added topic(s). DO NOT add any additional levels.

[Document]
{Document}

Please ONLY return the relevant or modified topics at the top level in the
↪  hierarchy. Your response should be in the following format:
[Topic Level] Topic Label: Topic Description
```

```
Your response:
```

5.11.2.2.2 Topic refinement

```
You will receive a list of topics that belong to the same level of a topic
↪ hierarchy. Your task is to merge topics that are paraphrases or near
↪ duplicates of one another. Return "None" if no modification is needed.

Here are some examples:
[Example 1]
Topic List:
- E-commerce
- Payment Systems
- FinTech

Your response:
- Commerce: E-commerce, FinTech, Payment Systems.

[Example 2]
- Web application
- Android app

Your response:
- Software development: Web application, Android app.

[Rules]
- Each line represents a topic, with a level indicator and a topic label.
- Perform the following operations as many times as needed:
```

```
    - Merge relevant topics into a single topic.
    - Do nothing and return "None" if no modification is needed.
- When merging, the output format should contain a level indicator, the
↪  updated label and description, followed by the original topics.


[Topic List]
{Topics}

Output the modification or "None" where appropriate. Do not output anything
↪  else.
[Your response]
```

5.11.2.2.3 Topic assignment

```
You will receive a document and a topic hierarchy. Assign the document to
↪  the most relevant topics the hierarchy. Then, output the topic labels,
↪  assignment reasoning and supporting quotes from the document. DO NOT
↪  make up new topics or quotes.

[Topic Hierarchy]
{tree}

[Examples]
Example 1: Assign "[1] Software Development" to the document
Document:
# Acode - Code Editor for Android

<p align="center">
```

```
  <img src='res/logo_1.png' width='250'>
</p>

[...]

## • Overview

Welcome to Acode Editor - a powerful and versatile code editing tool
↪  designed specifically for Android devices. Whether you're working on
↪  HTML, CSS, JavaScript, or other programming languages, Acode empowers
↪  you to code on-the-go with confidence.

## • Features

- Edit and create websites, and instantly preview them in a browser.
- Seamlessly modify source files for various languages like Python, Java,
↪  JavaScript, and more.
- Built-in javascript console
- Enjoy multi-language editing support with easy management tools.
- Enjoy a large collections of community plugins to enhance your coding
↪  experience.

## • Installation

You can get Acode Editor from popular platforms:

[...]

## • Project Structure
```

```
<pre>
Acode/
|
|- src/   - Core code and language files
|
|- www/   - Public documents, compiled files, and HTML templates
|
|- utils/ - CLI tools for building, string manipulation, and more
</pre>

## • Multi-language Support

Enhance Acode's capabilities by adding new languages easily. Just create a
 ↪  file with the language code (e.g., en-us for English) in [...] and
 ↪  include it in [...]. Manage strings across languages effortlessly using
 ↪  utility commands:

## • Contributing & Building the Application

See [CONTRIBUTING.md](CONTRIBUTING.md) for detailed instructions.

## • Contributors

<a href="https://github.com/Acode-Foundation/Acode/graphs/contributors">
  <img src="https://contrib.rocks/image?repo=Acode-Foundation/Acode" />
</a>

## • Developing a Plugin for Acode
```

```
For comprehensive documentation on creating plugins for Acode Editor, visit
 ↪  the [repository](https://github.com/Acode-Foundation/acode-plugin).

For plugin development information, refer to: [Acode Plugin
 ↪  Documentation](https://docs.acode.app/)

## Star History

<a href="https://star-history.com/#Acode-Foundation/Acode&Date">
 [...]
</a>

Assignment:
[1] Software Development: Mentions Android-based application related words
 ↪  ("...designed specifically for Android devices...")

Example 2: Assign "[1] Game" to the document
Document:
spyfall
=======

React + Firebase implementation for the card game
 ↪  [Spyfall](http://boardgamegeek.com/boardgame/166384/spyfall)

### Localizations
[...]

Access [crowdin's](https://crowdin.com/project/adrianocola-spyfall) page to
 ↪  request new translations or help translating existing ones.
```

```
### Running the project (locally)

- Install [node.js](https://nodejs.org/)
- Clone this project, enter the cloned folder and install dependencies with
↪  `npm install`
- Open a terminal and run the command `npm run emulator`
- Open another terminal and run the command `npm run dev-emulator`
- Access it at `http://localhost:5173`

### Running the project (deploy fo Firebase)

- Install [node.js](https://nodejs.org/)
- Clone this project, enter the cloned folder and install dependencies with
↪  `npm install`
- Login to firebase with `npx firebase login`
- Setup a new firebase project with the `npx firebase init` command and
↪  follow the instructions (select only the feature `database`. Use the
↪  default values for everything and don't overwrite anything)
- Create a new web app with the command `npx firebase apps:create WEB`. The
↪  command will output the created `App Id`
- Execute the command `npx firebase apps:sdkconfig WEB <created app id>` to
↪  get the complete app configuration
- Create a copy of the file `.env.sample` named `.env.development` and fill
↪  it with the firebase configuration. You don't need to fill all fields.
- Access your firebase project in the [firebase
↪  console](https://console.firebase.google.com/) and enable anonymous
↪  authentication (Authentication → Sign-in method → Anonymous)
- Deploy firebase database security rules `npx firebase deploy --only
↪  database`
```

```
- Run the project with `npm run dev` and access it at
↪  `http://localhost:5173`

Assignment:
[1] Game: Mentions card game implementation as main functionality of the
↪  library ("...implementation for the card game...")

Example 3: Assign "[1] Automotive" to the document
Document:
This adapter integrates BMW vehicles into ioBroker using the new BMW
↪  CarData API with OAuth2 authentication and real-time MQTT streaming. It
↪  provides comprehensive vehicle data monitoring for all BMW models
↪  linked to your BMW account.\n\n## Data Updata while charging\n\nWhile
↪  charging it can happens that the battery level is not updated via
↪  stream because the car is sleeping/standby when turn on the car the
↪  data will be updated. You can trigger an update via API.

Assignment:
[1] Automotive: Mentions BMW vehicles in the document ("...integrates BMW
↪  vehicles...").

[Instructions]
1. Topic labels must be present in the provided topic hierarchy. You MUST
↪  NOT make up new topics.
2. The quote must be taken from the document. You MUST NOT make up quotes.

[Document]
{Document}

Double check that your assignment exists in the hierarchy!
```

```
Your response should be in the following format:
[Topic Level] Topic Label: Assignment reasoning (Supporting quote)

Your response:
```

5.11.2.2.4 Topic correction

```
You will receive a document and a topic hierarchy. Assign the document to
↪ the most relevant topics the hierarchy. Then, output the topic labels,
↪ assignment reasoning and supporting quotes from the document. DO NOT
↪ make up new topics or quotes.

[Topic Hierarchy]
{tree}

[Examples]
Example 1: Assign "[1] Software Development" to the document
Document:
# Acode - Code Editor for Android

<p align="center">
  <img src='res/logo_1.png' width='250'>
</p>

[...]

## • Overview
```

```
Welcome to Acode Editor - a powerful and versatile code editing tool
 ↪  designed specifically for Android devices. Whether you're working on
 ↪  HTML, CSS, JavaScript, or other programming languages, Acode empowers
 ↪  you to code on-the-go with confidence.

## • Features

- Edit and create websites, and instantly preview them in a browser.
- Seamlessly modify source files for various languages like Python, Java,
 ↪  JavaScript, and more.
- Built-in javascript console
- Enjoy multi-language editing support with easy management tools.
- Enjoy a large collections of community plugins to enhance your coding
 ↪  experience.

## • Installation

You can get Acode Editor from popular platforms:

[...]

## • Project Structure

<pre>
Acode/
|
|- src/   - Core code and language files
|
|- www/   - Public documents, compiled files, and HTML templates
|
```

```
|- utils/ - CLI tools for building, string manipulation, and more
</pre>

## • Multi-language Support

Enhance Acode's capabilities by adding new languages easily. Just create a
 ↪  file with the language code (e.g., en-us for English) in [...] and
 ↪  include it in [...]. Manage strings across languages effortlessly using
 ↪  utility commands:

## • Contributing & Building the Application

See [CONTRIBUTING.md](CONTRIBUTING.md) for detailed instructions.

## • Contributors

<a href="https://github.com/Acode-Foundation/Acode/graphs/contributors">
  <img src="https://contrib.rocks/image?repo=Acode-Foundation/Acode" />
</a>

## • Developing a Plugin for Acode

For comprehensive documentation on creating plugins for Acode Editor, visit
 ↪  the [repository](https://github.com/Acode-Foundation/acode-plugin).

For plugin development information, refer to: [Acode Plugin
 ↪  Documentation](https://docs.acode.app/)

## Star History
```

```
<a href="https://star-history.com/#Acode-Foundation/Acode&Date">
 [...]
</a>

Assignment:
[1] Software Development: Mentions Android-based application related words
↪  ("...designed specifically for Android devices...")

Example 2: Assign "[1] Game" to the document
Document:
spyfall
=======

React + Firebase implementation for the card game
↪  [Spyfall](http://boardgamegeek.com/boardgame/166384/spyfall)

### Localizations
[...]

Access [crowdin's](https://crowdin.com/project/adrianocola-spyfall) page to
↪  request new translations or help translating existing ones.

### Running the project (locally)

- Install [node.js](https://nodejs.org/)
- Clone this project, enter the cloned folder and install dependencies with
↪  `npm install`
- Open a terminal and run the command `npm run emulator`
- Open another terminal and run the command `npm run dev-emulator`
- Access it at `http://localhost:5173`
```

```
### Running the project (deploy fo Firebase)

- Install [node.js](https://nodejs.org/)
- Clone this project, enter the cloned folder and install dependencies with
 ↪  `npm install`
- Login to firebase with `npx firebase login`
- Setup a new firebase project with the `npx firebase init` command and
 ↪  follow the instructions (select only the feature `database`. Use the
 ↪  default values for everything and don't overwrite anything)
- Create a new web app with the command `npx firebase apps:create WEB`. The
 ↪  command will output the created `App Id`
- Execute the command `npx firebase apps:sdkconfig WEB <created app id>` to
 ↪  get the complete app configuration
- Create a copy of the file `.env.sample` named `.env.development` and fill
 ↪  it with the firebase configuration. You don't need to fill all fields.
- Access your firebase project in the [firebase
 ↪  console](https://console.firebase.google.com/) and enable anonymous
 ↪  authentication (Authentication → Sign-in method → Anonymous)
- Deploy firebase database security rules `npx firebase deploy --only
 ↪  database`
- Run the project with `npm run dev` and access it at
 ↪  `http://localhost:5173`

Assignment:
[1] Game: Mentions card game implementation as main functionality of the
 ↪  library ("...implementation for the card game...")

Example 3: Assign "[1] Automotive" to the document
Document:
```

```
This adapter integrates BMW vehicles into ioBroker using the new BMW
↪ CarData API with OAuth2 authentication and real-time MQTT streaming. It
↪ provides comprehensive vehicle data monitoring for all BMW models
↪ linked to your BMW account.\n\n## Data Updata while charging\n\nWhile
↪ charging it can happens that the battery level is not updated via
↪ stream because the car is sleeping/standby when turn on the car the
↪ data will be updated. You can trigger an update via API.

Assignment:
[1] Automotive: Mentions BMW vehicles in the document ("...integrates BMW
↪ vehicles...").

[Instructions]
1. Topic labels must be present in the provided topic hierarchy. You MUST
↪  NOT make up new topics.
2. The quote must be taken from the document. You MUST NOT make up quotes.

[Document]
{Document}

{Message} Double check that your assignment exists in the hierarchy!
Your response should be in the following format:
[Topic Level] Topic Label: Assignment reasoning (Supporting quote)

Your response:
```

Data Availability Statement

The data and corresponding code used to carry out this work are publicly made available in the following GitHub repository: https://github.com/DaREf-MS/dependency-upgrades.

CONCLUSION AND RECOMMENDATIONS

6.1 Summary

In this thesis, we conducted a series of empirical studies to better understand the maintenance overhead of horizontal dependencies within large-scale software systems. In addition, we introduced and evaluated two approaches: one based on machine learning and a second leveraging agentic AI, aimed at predicting interdependencies among components of a software system. Finally, we empirically examined how vertical maintenance dependencies are managed through quantitative and qualitative analyses. Accordingly, we formulated the following research hypothesis:

> We hypothesize that modern software systems contain a non-trivial amount of maintenance dependencies – both horizontal, where developers must coordinate co-evolving changes across component boundaries, and vertical, where timely upgrades of third-party dependencies are frequently hindered by the associated maintenance overhead – and that such cross-component dependencies can be effectively predicted through a machine learning-based approach, with agentic-AI as a complementary solution.

Based on our research findings, we can confirm our research hypothesis that a non-trivial number of software changes cross component and team boundaries, where components may be developed and maintained by developers from different teams. Our thesis has also revealed several challenges associated with these interdependent changes, particularly in terms of coordination and synchronization efforts they require, potentially leading to delays and wasted development resources.

To overcome these challenges, we introduced and evaluated two automated approaches: one based on machine learning and a second using agentic-AI, to assist developers in finding dependencies among handered to thousands of software changes. Therefore, the evaluation results demonstrate promising performance, with the machine learning approach achieving higher recall, while the agentic AI approach exhibits higher precision. Similarly, the machine learning models perform well in a cross-component setting.

Finally, we empirically investigated how vertical dependencies are managed and outlined their associated challenges, particularly anlyzing the upgrade time from different perspectives with deep qualitative studies. Our results indicated that upgrading dependencies can take much time compared the way it was studied by prior work, and even worse for cases after superseding or through regular commits. Consequently, developers do not always perform these upgrades promptly, often due to their associated maintenance efforts.

6.2 Future Work

While this research thesis has explored the maintenance overhead caused by coupling among different components of a software system and introduced two automated approaches, one based on machine learning and the other as a lightweight agentic-AI solution, many important research directions remain for future work.

Specifically, since the advent of Large Language Models (LLMs) Radford, Narasimhan, Salimans & Sutskever (2018), researchers have extensively adopted these models across various software engineering tasks, ranging from code generation Du *et al.* (2024), code review Cihan *et al.* (2025), to testing Liu *et al.* (2024). More recently, LLMs have evolved beyond standalone models that simply receive a prompt and generate a response. They are increasingly becoming autonomous agents capable of performing complex tasks that software engineers typically

undertake, such as analyzing GitHub issues, navigating an entire project locally, generating code, validating tests, and committing changes Li, Zhang & Hassan (2025).

With this remarkable shift, we envision future studies to investigate the extent to which these autonomous agents can handle the development of multiple interconnected vertical and horizontal components. For instance, future research directions include answering questions such as: are the autonomous agentic-AI models able to identify the impact of their changes on other components, are they able to handle features and changes that go beyond the scope of a single component, can they safely implement features without breaking external components, and how can multiple AI agents coordinate changes across interconnected components? Investigating these questions could significantly advance the integration of agentic-AI models into dependency management practices.

A second research direction we envision is how developers guide through ACFs (Agent Context Files GitHub Documentation (2025)), such as "AGENTS.md", Agentic-AI models navigate and change complex projects without introducing regressions in external components. Furthermore, we also plan to study what ACF instructions are most effective for managing dependencies across components.

Future studies could focus on developing benchmarks for evaluating dependency-aware AI agents. Such benchmarks could incorporate interconnected components similar to those used in this thesis. Furthermore, we also plan to explore additional approaches used by other large-scale software systems to identify and manage dependencies across projects, and services. Insights from these systems can guide the development of more effective AI-assisted dependency solutions.

Last but not least, as our thesis focuses on OpenStack to study the maintenance overhead caused by horisontal dependencies, a substantial body of prior work AlOmar *et al.* (2022); Foundjem

et al. (2022a); Bessghaier *et al.* (2023); Wang *et al.* (2023) focused on OpenStack, and that, they share the same threat to validity related to the generalizability of our results. However, we focused on OpenStack since it is a popular open-source software system involving a large number of projects and services, studied by a large body of previous studies, and especially has a well-defined way of declaring dependencies among changes that allows us to systematically identify cross-project and cross-service dependent changes. Hene, we recommend future work and practitioners replicate our study on their multi-component systems to better understand the co-evolution of their components and better manage such an evolution.

LIST OF REFERENCES

Abdellatif, M., Hecht, G., Mili, H., Elboussaidi, G., Moha, N., Shatnawi, A., Privat, J. & Guéhéneuc, Y.-G. (2018). State of the Practice in Service Identification for SOA Migration in Industry. *Service-Oriented Computing*, pp. 634–650.

Abdellatif, M., Tighilt, R., Moha, N., Mili, H., El Boussaidi, G., Privat, J. & Guéhéneuc, Y.-G. (2020). A Type-Sensitive Service Identification Approach for Legacy-to-SOA Migration. *Service-Oriented Computing*, pp. 476–491.

Abdellatif, M., Shatnawi, A., Mili, H., Moha, N., Boussaidi, G. E., Hecht, G., Privat, J. & Guéhéneuc, Y.-G. (2021). A taxonomy of service identification approaches for legacy software systems modernization. *Journal of Systems and Software*, 173, 110868. doi: 10.1016/j.jss.2020.110868.

Abgaz, Y., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., Jackson, G., Yilmaz, M., Buckley, J. & Clarke, P. (2023). Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review. *IEEE Transactions on Software Engineering*, 1-32. doi: 10.1109/TSE.2023.3287297.

Aburbeian, A. M. & Ashqar, H. I. (2023). Credit Card Fraud Detection Using Enhanced Random Forest Classifier for Imbalanced Data. *Proceedings of the 2023 International Conference on Advances in Computing Research (ACR'23)*, pp. 605–616. doi: 10.1007/978-3-031-33743-7_48.

Alfadel, M., Costa, D. E., Shihab, E. & Mkhallalati, M. (2021). On the Use of Dependabot Security Pull Requests. *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 254-265. doi: 10.1109/MSR52588.2021.00037.

AlOmar, E. A., Chouchen, M., Mkaouer, M. W. & Ouni, A. (2022). Code Review Practices for Refactoring Changes: An Empirical Study on OpenStack. *Proceedings of the 19th International Conference on Mining Software Repositories*, (MSR '22), 689-701. doi: 10.1145/3524842.3527932.

Alsayed, A. S., Dam, H. K. & Nguyen, C. (2024). *MicroDec: Leveraging Large Language Models for Microservice Decomposition*.

Amazon. (2025). What are microservices? | AWS. Retrieved on 2025-07-09 from: https://tinyurl.com/yv78dz8p.

Arabat, A. & Sayagh, M. (2024). An empirical study on cross-component dependent changes: A case study on the components of OpenStack. *Empirical Software Engineering*, 29(5), 109. doi: 10.1007/s10664-024-10488-y.

Aryani, A., Perin, F., Lungu, M., Mahmood, A. N. & Nierstrasz, O. (2011). Can We Predict Dependencies Using Domain information? *2011 18th Working Conference on Reverse Engineering*, pp. 55-64. doi: 10.1109/WCRE.2011.17.

Assunção, W. K., Krüger, J., Mosser, S. & Selaoui, S. (2023). How do microservices evolve? An empirical analysis of changes in open-source microservice repositories. *Journal of Systems and Software*, 204, 111788. doi: 10.1016/j.jss.2023.111788.

Batoun, M. A., Yung, K. L., Tian, Y. & Sayagh, M. (2023). An Empirical Study on GitHub Pull Requests' Reactions. *ACM Trans. Softw. Eng. Methodol.*, 32(6). doi: 10.1145/3597208.

Bessghaier, N., Sayagh, M., Ouni, A. & Mkaouer, M. W. (2023). What Constitutes the Deployment and Runtime Configuration System? An Empirical Study on OpenStack Projects. *ACM Trans. Softw. Eng. Methodol.*, 33(1). doi: 10.1145/3607186.

Bessghaier, N., Ouni, A., Sayagh, M., Chouchen, M. & Mkaouer, M. W. (2025). Towards understanding code review practices for infrastructure-as-code: An empirical study on OpenStack projects. *Empirical Software Engineering*, 30(4), 106. doi: 10.1007/s10664-025-10654-w.

Bi, F., Liang, C., Zhang, Y., Chen, Y. & Wang, W. (2025). Uncover the Risks of Outdated Dependencies in Software Supply Chains: Insights from the npm Ecosystem. *Engineering of Complex Computer Systems: 29th International Conference, ICECCS 2025, Hangzhou, China, July 2–4, 2025, Proceedings*, pp. 403–423. doi: 10.1007/978-3-032-00828-2_22.

Blincoe, K., Harrison, F., Kaur, N. & Damian, D. (2019). Reference Coupling: An exploration of inter-project technical dependencies and their characteristics within large software ecosystems. *Information and Software Technology*, 110, 174-189. doi: 10.1016/j.infsof.2019.03.005.

Bogner, J., Fritzsch, J., Wagner, S. & Zimmermann, A. (2019). Assuring the Evolvability of Microservices: Insights into Industry Practices and Challenges. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 546-556. doi: 10.1109/ICSME.2019.00089.

Bogner, J., Fritzsch, J., Wagner, S. & Zimmermann, A. (2021). Industry practices and challenges for the evolvability assurance of microservices. *Empirical Software Engineering*, 26(5), 104. doi: 10.1007/s10664-021-09999-9.

Bonferroni, C. (1936). Teoria statistica delle classi e calcolo delle probabilita. *Pubblicazioni del R istituto superiore di scienze economiche e commericiali di firenze*, 8, 3–62.

Bosu, A., Carver, J. C., Hafiz, M., Hilley, P. & Janni, D. (2014). Identifying the characteristics of vulnerable code changes: an empirical study. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (FSE 2014), 257-268. doi: 10.1145/2635868.2635880.

Candela, I., Bavota, G., Russo, B. & Oliveto, R. (2016). Using Cohesion and Coupling for Software Remodularization: Is It Enough? *ACM Trans. Softw. Eng. Methodol.*, 25(3). doi: 10.1145/2928268.

Carettoni, L. & Trummer, A. [Accessed on 10-11-2025]. (2023). doyensec.com. Retrieved from: https://www.doyensec.com/resources/Doyensec_Software_Composition_Analysis_Benchmark.pdf.

Casola, V., De Benedictis, A., Mazzocca, C. & Orbinato, V. (2024). Secure software development and testing: A model-based methodology. *Computers & Security*, 137, 103639. doi: 10.1016/j.cose.2023.103639.

Chen, G., Li, C., Tyszberowicz, S., Liu, Z. & Liu, B. (2024). Mono2MS: Deep Fusion of Multi-Source Features for Partitioning Monolith into Microservices. *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, (Internetware '24), 259-268. doi: 10.1145/3671016.3674817.

Chen, H., Ai, H., Yang, Z., Yang, W., Ye, Z. & Dong, D. (2020). An Improved XGBoost Model Based on Spark for Credit Card Fraud Prediction. *2020 IEEE 5th International Symposium on Smart and Wireless Systems within the Conferences on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS-SWS)*, pp. 1-6. doi: 10.1109/IDAACS-SWS50031.2020.9297058.

Chouchen, M., Ouni, A., Olongo, J. & Mkaouer, M. W. (2023). Learning to Predict Code Review Completion Time In Modern Code Review. *Empirical Software Engineering*, 28(4), 82. doi: 10.1007/s10664-023-10300-3.

Chowdhury, M. A. R., Abdalkareem, R., Shihab, E. & Adams, B. (2022). On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages. *IEEE Transactions on Software Engineering*, 48(8), 2695-2708. doi: 10.1109/TSE.2021.3068901.

Cihan, U., Haratian, V., İçöz, A., Gül, M. K., Devran, Ö., Bayendur, E. F., Uçar, B. M. & Tüzün, E. (2025). Automated Code Review in Practice. *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 425-436.

Cogo, F. R. & Hassan, A. E. (2022). Understanding the Customization of Dependency Bots: The Case of Dependabot. *IEEE Software*, 39(5), 44-49. doi: 10.1109/MS.2022.3179484.

Cogo, F. R., Oliva, G. A. & Hassan, A. E. (2022). Deprecation of Packages and Releases in Software Ecosystems: A Case Study on NPM. *IEEE Transactions on Software Engineering*, 48(7), 2208-2223. doi: 10.1109/TSE.2021.3055123.

Cogo, F. R., Oliva, G. A. & Hassan, A. E. (2021). An Empirical Study of Dependency Downgrades in the npm Ecosystem. *IEEE Transactions on Software Engineering*, 47(11), 2457-2470. doi: 10.1109/TSE.2019.2952130.

Cohen, J. (1960). A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1), 37-46. doi: 10.1177/001316446002000104.

Colanzi, T., Amaral, A., Assunção, W., Zavadski, A., Tanno, D., Garcia, A. & Lucena, C. (2021). Are we speaking the industry language? The practice and literature of modernizing legacy systems with microservices. *Proceedings of the 15th Brazilian Symposium on Software Components, Architectures, and Reuse*, (SBCARS '21), 61-70. doi: 10.1145/3483899.3483904.

De Lauretis, L. (2019). From Monolithic Architecture to Microservices Architecture. *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 93-96. doi: 10.1109/ISSREW.2019.00050.

Decan, A., Mens, T. & Grosjean, P. (2019). An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1), 381–416. doi: 10.1007/s10664-017-9589-y.

Dependabot. [Accessed on 2025-09-11]. (2025a). Dependabot quickstart guide. Retrieved from: https://docs.github.com/en/code-security/getting-started/dependabot-quickstart-guide.

Dependabot. [Accessed on 2025-09-11]. (2025b). Dependabot security updates. Retrieved from: https://docs.github.com/en/code-security/dependabot/dependabot-security-updates.

Dependabot. [Accessed on 2025-09-11]. (2025c). Dependabot version updates. Retrieved from: https://docs.github.com/en/code-security/dependabot/dependabot-version-updates.

Diaz-Pace, J. A., Tommasel, A. & Godoy, D. (2018). Can Network Analysis Techniques Help to Predict Design Dependencies? An Initial Study. *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pp. 64-67. doi: 10.1109/ICSA-C.2018.00025.

Ding, Z., Xu, Y., Feng, B. & Jiang, C. (2024). Microservice Extraction Based on a Comprehensive Evaluation of Logical Independence and Performance. *IEEE Transactions on Software Engineering*, 50(5), 1244-1263. doi: 10.1109/TSE.2024.3380194.

Du, X., Liu, M., Wang, K., Wang, H., Liu, J., Chen, Y., Feng, J., Sha, C., Peng, X. & Lou, Y. (2024). Evaluating Large Language Models in Class-Level Code Generation. (ICSE '24).

Fan, C.-Y. & Ma, S.-P. (2017). Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report. *2017 IEEE International Conference on AI & Mobile Services (AIMS)*, pp. 109-112. doi: 10.1109/AIMS.2017.23.

Foundjem, A., Constantinou, E., Mens, T. & Adams, B. (2022a). A mixed-methods analysis of micro-collaborative coding practices in OpenStack. *Empirical Software Engineering*, 27(5), 120. doi: 10.1007/s10664-022-10167-w.

Foundjem, A., Constantinou, E., Mens, T. & Adams, B. (2022b). A mixed-methods analysis of micro-collaborative coding practices in OpenStack. *Empirical Software Engineering*, 27(5), 120. doi: 10.1007/s10664-022-10167-w.

Fowler, S. J. (2016). *Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization* (ed. 1st). O'Reilly Media, Inc.

Fritzsch, J., Bogner, J., Wagner, S. & Zimmermann, A. (2019). Microservices Migration in Industry: Intentions, Strategies, and Challenges. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 481-490. doi: 10.1109/ICSME.2019.00081.

Furda, A., Fidge, C., Zimmermann, O., Kelly, W. & Barros, A. (2018). Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency. *IEEE Software*, 35(3), 63-72. doi: 10.1109/MS.2017.440134612.

Gabhart, K. & Bhattacharya, B. (2008). *Service Oriented Architecture Field Guide for Executives*. Wiley Publishing.

Garcia Quevedo, D., Glaser, A. & Verzat, C. (2026). Enhancing theorization using artificial intelligence: Leveraging large language models for qualitative analysis of online data. *Organizational Research Methods*, 29(1), 92–112. doi: 10.1177/10944281251339144.

GitHub Documentation. [Accessed: 2025-12-25]. (2025). Adding Repository Custom Instructions for GitHub Copilot. Retrieved from: https://docs.github.com/en/copilot/how-tos/configure-custom-instructions/add-repository-instructions.

Göçmen, I. S., Cezayir, A. S. & Tüzün, E. (2025). Enhanced code reviews using pull request based change impact analysis. *Empirical Software Engineering*, 30(3), 64. doi: 10.1007/s10664-024-10600-2.

Gouigoux, J.-P. & Tamzalit, D. (2017). From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 62-65. doi: 10.1109/ICSAW.2017.35.

He, H., Vasilescu, B. & Kästner, C. (2025). Pinning Is Futile: You Need More Than Local Dependency Versioning to Defend against Supply Chain Attacks. *Proc. ACM Softw. Eng.*, 2(FSE). doi: 10.1145/3715728.

He, R., He, H., Zhang, Y. & Zhou, M. (2023). Automating Dependency Updates in Practice: An Exploratory Study on GitHub Dependabot. *IEEE Transactions on Software Engineering*, 49(8), 4004-4022. doi: 10.1109/TSE.2023.3278129.

Henig, A. & Hyde, C. [Accessed on 10-11-2025]. (2025). Breakdown: Widespread npm Supply Chain Attack Puts Billions of Weekly Downloads at Risk - Palo Alto Networks Blog — paloaltonetworks.com. Retrieved from: https://www.paloaltonetworks.com/blog/cloud-security/npm-supply-chain-attack/.

Hirao, T., McIntosh, S., Ihara, A. & Matsumoto, K. (2019). The review linkage graph for code review analytics: a recovery approach and empirical study. (ESEC/FSE 2019), 578-589. doi: 10.1145/3338906.3338949.

Hordijk, W., Ponisio, M. L. & Wieringa, R. (2009). Harmfulness of Code Duplication - A Structured Review of the Evidence. *Journal of Signal Processing Systems*. Retrieved from: https://api.semanticscholar.org/CorpusID:17418919.

Jezek, K., Dietrich, J. & Brada, P. (2015). How Java APIs Break - An Empirical Study. 65(C), 129-146. doi: 10.1016/j.infsof.2015.02.014.

Jiang, Y. & Adams, B. (2015). Co-Evolution of Infrastructure and Source Code: An Empirical Study. *Proceedings of the 12th Working Conference on Mining Software Repositories*, (MSR '15), 45-55. doi: 10.1109/MSR.2015.12.

Jiarpakdee, J., Tantithamthavorn, C. & Hassan, A. E. (2019). The impact of correlated metrics on the interpretation of defect models. *IEEE Transactions on Software Engineering*, 47(2), 320–331.

Khatoonabadi, S., Costa, D. E., Abdalkareem, R. & Shihab, E. (2023). On Wasted Contributions: Understanding the Dynamics of Contributor-Abandoned Pull Requests-A Mixed-Methods Study of 10 Large Open-Source Projects. *ACM Trans. Softw. Eng. Methodol.*, 32(1). doi: 10.1145/3530785.

Kula, R. G. (2025). Reducing Alert Fatigue via AI-Assisted Negotiation: A Case for Dependabot. *2025 IEEE/ACM International Workshop on Bots in Software Engineering (BotSE)*, pp. 11–12.

Kula, R. G., Ouni, A., Germán, D. M. & Inoue, K. (2017). On the Impact of Micro-Packages: An Empirical Study of the npm JavaScript Ecosystem. *CoRR*, abs/1709.04638.

Kula, R. G., German, D. M., Ouni, A., Ishio, T. & Inoue, K. (2018a). Do developers update their library dependencies? *Empirical Software Engineering*, 23(1), 384-417. doi: 10.1007/s10664-017-9521-5.

Kula, R. G., German, D. M., Ouni, A., Ishio, T. & Inoue, K. (2018b). Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering*, 23(1), 384–417. doi: 10.1007/s10664-017-9521-5.

Latendresse, J., Mujahid, S., Costa, D. E. & Shihab, E. (2023). Not All Dependencies are Equal: An Empirical Study on Production Dependencies in NPM. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, (ASE '22). doi: 10.1145/3551349.3556896.

Lee, D., Rajbahadur, G. K., Lin, D., Sayagh, M., Bezemer, C.-P. & Hassan, A. E. (2020). An empirical study of the characteristics of popular Minecraft mods. *Empirical Softw. Engg.*, 25(5), 3396-3429. doi: 10.1007/s10664-020-09840-9.

Li, C.-Y., Ma, S.-P. & Lu, T.-W. (2020). Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System. *2020 International Computer Symposium (ICS)*, pp. 519-524. doi: 10.1109/ICS51289.2020.00107.

Li, H., Zhang, H. & Hassan, A. E. (2025). The Rise of AI Teammates in Software Engineering (SE) 3.0: How Autonomous Coding Agents Are Reshaping Software Engineering.

Li, H., Shang, W., Zou, Y. & E. Hassan, A. (2017). Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, 22(4), 1831-1865. doi: 10.1007/s10664-016-9467-z.

Li, X., d'Aragona, D. A. & Taibi, D. (2023). Evaluating Microservice Organizational Coupling Based on Cross-Service Contribution. *Product-Focused Software Process Improvement: 24th International Conference, PROFES 2023, Dornbirn, Austria, December 10-13, 2023, Proceedings, Part I*, pp. 435-450. doi: 10.1007/978-3-031-49266-2_30.

Lin, J., Sayagh, M. & Hassan, A. E. (2023). The Co-evolution of the WordPress Platform and Its Plugins. *ACM Trans. Softw. Eng. Methodol.*, 32(1). doi: 10.1145/3533700.

Liu, Z., Chen, C., Wang, J., Chen, M., Wu, B., Che, X., Wang, D. & Wang, Q. (2024). Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions. (ICSE '24).

Macklon, F., Viggiato, M., Romanova, N., Buzon, C., Paas, D. & Bezemer, C.-P. (2023). A Taxonomy of Testable HTML5 Canvas Issues. *IEEE Transactions on Software Engineering*, 49(6), 3647-3659. doi: 10.1109/TSE.2023.3270740.

Maddila, C., Upadrasta, S. S., Bansal, C., Nagappan, N., Gousios, G. & van Deursen, A. (2023). Nudge: Accelerating Overdue Pull Requests toward Completion. *ACM Trans. Softw. Eng. Methodol.*, 32(2). doi: 10.1145/3544791.

Mann, H. B. & Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, 50–60.

Mazlami, G., Cito, J. & Leitner, P. (2017). Extraction of Microservices from Monolithic Software Architectures. *2017 IEEE International Conference on Web Services (ICWS)*, pp. 524-531. doi: 10.1109/ICWS.2017.61.

McHugh, M. L. (2012). Interrater reliability: the kappa statistic. *Biochemia medica*, 22(3), 276–282. doi: 10.11613/BM.2012.031.

Mens, T. & Decan, A. (2024). An Overview and Catalogue of Dependency Challenges in Open Source Software Package Registries. Retrieved from: https://arxiv.org/abs/2409.18884.

Mohayeji, H., Agaronian, A., Constantinou, E., Zannone, N. & Serebrenik, A. (2023). Investigating the Resolution of Vulnerable Dependencies with Dependabot Security Updates. *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pp. 234-246. doi: 10.1109/MSR59073.2023.00042.

Mohayeji, H., Agaronian, A., Constantinou, E., Zannone, N. & Serebrenik, A. (2025). Securing dependencies: A comprehensive study of Dependabot's impact on vulnerability mitigation. *Empirical Software Engineering*, 30(3), 89. doi: 10.1007/s10664-025-10638-w.

Mparmpoutis, A. & Kakarontzas, G. (2023). Using Database Schemas of Legacy Applications for Microservices Identification: A Mapping Study. (ICACS '22). doi: 10.1145/3564982.3564995.

Netflix. (2025). Rebuilding Netflix Video Processing Pipeline with Microservices. Retrieved from: https://tinyurl.com/y9xh8w6t.

Nitin, V., Asthana, S., Ray, B. & Krishna, R. (2023). CARGO: AI-Guided Dependency Analysis for Migrating Monolithic Applications to Microservices Architecture. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, (ASE '22). doi: 10.1145/3551349.3556960.

Niwa, T., Kasuya, Y. & Kitahara, T. (2017). Anomaly detection for openstack services with process-related topological analysis. *20172 13th International Conference on Network and Service Management (CNSM)*, pp. 1-5. doi: 10.23919/CNSM.2017.8255977.

Oliva, G. A. & Gerosa, M. A. (2012). A Method for the Identification of Logical Dependencies. *2012 IEEE Seventh International Conference on Global Software Engineering Workshops*, pp. 70-72. doi: 10.1109/ICGSEW.2012.19.

OpenStack. (2023, Apr, 24). Statistics about OpenStack. Retrieved on 2023-04-24 from: https://openinfra.dev.

Ouatiti, Y. E., Sayagh, M., Kerzazi, N. & Hassan, A. E. (2023). 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.

Peduzzi, P., Concato, J., Kemper, E., Holford, T. R. & Feinstein, A. R. (1996). A simulation study of the number of events per variable in logistic regression analysis. *Journal of clinical epidemiology*, 49(12), 1373–1379. doi: 10.1016/s0895-4356(96)00236-3.

Pham, C. M., Hoyle, A., Sun, S., Resnik, P. & Iyyer, M. (2024, Jun). TopicGPT: A Prompt-based Topic Modeling Framework. *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 2956–2984. doi: 10.18653/v1/2024.naacl-long.164.

Pinto, G., Steinmacher, I. & Gerosa, M. A. (2016). More Common Than You Think: An In-depth Study of Casual Contributors. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 1, 112-123. doi: 10.1109/SANER.2016.68.

Pinto-Agüero, S. & Noel, R. (2025). Microservices Evolution Factors: A Multivocal Literature Review. *IEEE Access*, 13, 88707-88730. doi: 10.1109/ACCESS.2025.3570658.

Piper, A. & Wu, S. (2025). Evaluating Large Language Models for Narrative Topic Labeling. *Proceedings of the 5th International Conference on Natural Language Processing for Digital Humanities*, pp. 281–291. doi: 10.18653/v1/2025.nlp4dh-1.25.

Quattrocchi, G., Cocco, D., Staffa, S., Margara, A. & Cugola, G. (2024). Cromlech: Semi-Automated Monolith Decomposition Into Microservices. *IEEE Transactions on Services Computing*, 17(2), 466-481. doi: 10.1109/TSC.2024.3354457.

Radford, A., Narasimhan, K., Salimans, T. & Sutskever, I. (2018). Improving language understanding by generative pre-training.

Raemaekers, S., van Deursen, A. & Visser, J. (2014). Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 215-224. doi: 10.1109/SCAM.2014.30.

Rebatchi, H., Bissyandé, T. F. & Moha, N. (2024). Dependabot and security pull requests: large empirical study. *Empirical Software Engineering*, 29(5), 128. doi: 10.1007/s10664-024-10523-y.

Renovate. [Accessed on 2025-09-11]. (2025). Automated dependency updates. Multi-platform and multi-language. Retrieved from: https://docs.renovatebot.com.

Richardson, C. (2018). *Microservices Patterns: With examples in Java*. Manning. Retrieved from: https://books.google.ca/books?id=UeK1swEACAAJ.

Sampaio, A. R., Kadiyala, H., Hu, B., Steinbacher, J., Erwin, T., Rosa, N., Beschastnikh, I. & Rubin, J. (2017). Supporting Microservice Evolution. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 539-543. doi: 10.1109/ICSME.2017.63.

Sayagh, M. & Adams, B. (2015). Multi-layer software configuration: Empirical study on wordpress. *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 31-40. doi: 10.1109/SCAM.2015.7335399.

Sayagh, M., Kerzazi, N. & Adams, B. (2017). On Cross-Stack Configuration Errors. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 255-265. doi: 10.1109/ICSE.2017.31.

Services, A. W. (2016, Sep, 22). Introduction to Microservices. Retrieved on 2023-06-23 from: https://www.slideshare.net/AmazonWebServices/introduction-to-microservices-66320469.

Sharma, A. [Accessed on 10-11-2025]. (2021). NPM package with 3 million weekly downloads had a severe vulnerability — arstechnica.com. Retrieved from: https://arstechnica.com/information-technology/2021/09/npm-package-with-3-million-weekly-downloads-had-a-severe-vulnerability/.

Shehab, M. A., Hamou-Lhadj, A. & Gunda, V. S. (2023). JITBoost: Boosting Just-In-Time Defect Prediction using Boolean Combination of Classifiers. *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pp. 95-104.

Shin, J., Tang, C., Mohati, T., Nayebi, M., Wang, S. & Hemmati, H. (2025). Prompt Engineering or Fine-Tuning: An Empirical Assessment of LLMs for Code. *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pp. 490-502. doi: 10.1109/MSR66628.2025.00082.

Shin, Y., Meneely, A., Williams, L. & Osborne, J. A. (2011). Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6), 772-787. doi: 10.1109/TSE.2010.81.

Snyk. [Accessed on 2025-09-11]. (2025). AI-powered platform. Retrieved from: https://snyk.io.

Soto-Valero, C., Harrand, N., Monperrus, M. & Baudry, B. (2021). A comprehensive study of bloated dependencies in the Maven ecosystem. *Empirical Software Engineering*, 26(3), 45. doi: 10.1007/s10664-020-09914-8.

Sousa, B. L., Bigonha, M. A. S. & Ferreira, K. A. M. (2019). Analysis of Coupling Evolution on Open Source Systems. *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*, (SBCARS '19), 23-32. doi: 10.1145/3357141.3357147.

Spencer, D. (2009). *Card sorting: Designing usable categories*. Rosenfeld Media.

Sun, S., Wang, S., Wang, X., Xing, Y., Zhang, E. & Sun, K. (2023, Oct). Exploring Security Commits in Python. *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 171-181. doi: 10.1109/ICSME58846.2023.00027.

Taibi, D., Lenarduzzi, V. & Pahl, C. (2020). Microservices Anti-patterns: A Taxonomy. In Bucchiarone, A., Dragoni, N., Dustdar, S., Lago, P., Mazzara, M., Rivera, V. & Sadovykh, A. (Eds.), *Microservices: Science and Engineering* (pp. 111–128). Cham: Springer International Publishing. doi: 10.1007/978-3-030-31646-4_5.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E. & Matsumoto, K. (2017). An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering*, 43(1), 1-18. doi: 10.1109/TSE.2016.2584050.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E. & Matsumoto, K. (2019). The Impact of Automated Parameter Optimization on Defect Prediction Models. *IEEE Transactions on Software Engineering*, 45(7), 683-711. doi: 10.1109/TSE.2018.2794977.

Tantithamthavorn, C., Hassan, A. E. & Matsumoto, K. (2020). The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models. *IEEE Transactions on Software Engineering*, 46(11), 1200-1219. doi: 10.1109/TSE.2018.2876537.

Teixeira, J. A. & Karsten, H. (2019). Managing to release early, often and on time in the OpenStack software ecosystem. *Journal of Internet Services and Applications*, 10(1), 32. doi: 10.1186/s13174-019-0105-z.

Thomas, N. S. & Kaliraj, S. (2024). An Improved and Optimized Random Forest Based Approach to Predict the Software Faults. *SN Computer Science*, 5(5), 530. doi: 10.1007/s42979-024-02764-x.

Tighilt, R., Abdellatif, M., Moha, N., Mili, H., Boussaidi, G. E., Privat, J. & Guéhéneuc, Y.-G. (2020). On the Study of Microservices Antipatterns: a Catalog Proposal. *Proceedings of the European Conference on Pattern Languages of Programs 2020*, (EuroPLoP '20). doi: 10.1145/3424771.3424812.

Tong, J., Ying, L., Xiaoyong, Y., Hongyan, T. & Zhonghai, W. (2015). Characterizing and Predicting Bug Assignment in OpenStack. *2015 Second International Conference on Trustworthy Systems and Their Applications*, pp. 16-23. doi: 10.1109/TSA.2015.14.

Trabelsi, I., Abdellatif, M., Abubaker, A., Moha, N., Mosser, S., Ebrahimi-Kahou, S. & Guéhéneuc, Y.-G. (2022). From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis. *Journal of Software: Evolution and Process*, e2503. doi: 10.1002/smr.2503.

Trabelsi, I., Moha, N., Guéhéneuc, Y.-G. & Geffard, L. (2024a). Magnet: Method-Based Approach Using Graph Neural Network for Microservices Identification. *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, pp. 1-11. doi: 10.1109/ICSA59870.2024.00009.

Trabelsi, I., Popa, B., Pereyrol, J., Beaulieu, P.-O. & Moha, N. (2024b). MicroMatic: Fully Automated Microservices Identification Approach From Monolithic Systems. *Proceedings of the ACM/IEEE 6th International Workshop on Software Engineering Research & Practices for the Internet of Things*, (SERP4IoT '24), 7-13. doi: 10.1145/3643794.3648283.

Uber. (2018, 7). Microservice architecture - learn, build, and deploy applications. Retrieved on 2025-07-09 from: https://tinyurl.com/yts8aj3k.

Ubuntu. (2022, Mar, 03). OpenStack is dead? The numbers speak for themselves. Retrieved on 2023-04-25 from: https://ubuntu.com/blog/openstack-is-dead.

Venturini, D., Cogo, F. R., Polato, I., Gerosa, M. A. & Wiese, I. S. (2023). I Depended on You and You Broke Me: An Empirical Study of Manifesting Breaking Changes in Client Packages. *ACM Trans. Softw. Eng. Methodol.*, 32(4). doi: 10.1145/3576037.

Vučković, J. (2020). You Are Not Netflix. In Bucchiarone, A., Dragoni, N., Dustdar, S., Lago, P., Mazzara, M., Rivera, V. & Sadovykh, A. (Eds.), *Microservices: Science and Engineering* (pp. 333–346). Cham: Springer International Publishing. doi: 10.1007/978-3-030-31646-4_13.

Wang, D., Kula, R. G., Ishio, T. & Matsumoto, K. (2021a). Automatic patch linkage detection in code review using textual content and file location features. *Information and Software Technology*, 139, 106637. doi: 10.1016/j.infsof.2021.106637.

Wang, D., Xiao, T., Thongtanunam, P., Kula, R. G. & Matsumoto, K. (2021b). Understanding shared links and their intentions to meet information needs in modern code review:. *Empirical Software Engineering*, 26(5), 96. doi: 10.1007/s10664-021-09997-x.

Wang, D., Thongtanunam, P., Kula, R. G. & Matsumoto, K. (2023). An Exploration of Cross-Patch Collaborations via Patch Linkage in OpenStack. *IEICE Transactions on Information and Systems*, E106.D(2), 148-156. doi: 10.1587/transinf.2022MPP0002.

Wang, Q., Xia, X., Lo, D. & Li, S. (2019). Why is my code change abandoned? *Information and Software Technology*, 110, 108-120. doi: 10.1016/j.infsof.2019.02.007.

Weeraddana, N. R., Alfadel, M. & McIntosh, S. (2024). Dependency-Induced Waste in Continuous Integration: An Empirical Study of Unused Dependencies in the npm Ecosystem. *Proc. ACM Softw. Eng.*, 1(FSE). doi: 10.1145/3660823.

WordPress. (2023, Jun, 08). WordPress: Popular content management system. Retrieved on 2023-06-08 from: https://wordpress.org.

Xia, X., Lo, D., McIntosh, S., Shihab, E. & Hassan, A. E. (2015). Cross-project build co-change prediction. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 311-320. doi: 10.1109/SANER.2015.7081841.

Yamato, Y., Katsuragi, S., Nagao, S. & Miura, N. (2015). Software Maintenance Evaluation of Agile Software Development Method Based on OpenStack. *IEICE Transactions on Information and Systems*, 98(7), 1377-1380. doi: 10.1587/transinf.2015EDL8049.

Yang, T., Shen, J., Su, Y., Ling, X., Yang, Y. & Lyu, M. R. (2021). AID: Efficient Prediction of Aggregated Intensity of Dependency in Large-scale Cloud Systems. *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 653-665. doi: 10.1109/ASE51524.2021.9678534.

Zagane, M. & Alenezi, M. (2024). Enhancing Software Co-Change Prediction: Leveraging Hybrid Approaches for Improved Accuracy. *IEEE Access*, 12, 68441-68452. doi: 10.1109/ACCESS.2024.3399101.

Zhang, Y., Lo, D., Xia, X. & Sun, J. (2015). An Empirical Study of Classifier Combination for Cross-Project Defect Prediction. *2015 IEEE 39th Annual Computer Software and Applications Conference*, 2, 264-269. doi: 10.1109/COMPSAC.2015.58.

Zheng, W., Feng, C., Yu, T., Yang, X. & Wu, X. (2019). Towards understanding bugs in an open source cloud management stack: An empirical study of OpenStack software bugs. *Journal of Systems and Software*, 151, 210-223. doi: 10.1016/j.jss.2019.02.025.

Zhong, C., Zhang, H., Li, C., Huang, H. & Feitosa, D. (2023). On measuring coupling between microservices. *Journal of Systems and Software*, 200, 111670. doi: 10.1016/j.jss.2023.111670.

Zhou, H., Chen, Y. & Lipton, Z. C. (2022). Model Evaluation in Medical Datasets Over Time. Retrieved from: https://arxiv.org/abs/2211.07165.

Zhou, Y., Siow, J. K., Wang, C., Liu, S. & Liu, Y. (2021). SPI: Automated Identification of Security Patches via Commits. *ACM Trans. Softw. Eng. Methodol.*, 31(1). doi: 10.1145/3468854.

Zimmermann, T., Zeller, A., Weissgerber, P. & Diehl, S. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6), 429-445. doi: 10.1109/TSE.2005.72.