Understanding, Analysis, and Handling of Software Architecture Erosion

Li, Ruiyin

DOI:
10.33612/diss.833424077

university of groningen

Understanding, Analysis, and Handling of Software Architecture Erosion

PhD thesis

to obtain the degree of PhD at the
University of Groningen
on the authority of the
Rector Magnificus Prof. J.M.A. Scherpen
and in accordance with
the decision by the College of Deans.

This thesis will be defended in public on

Tuesday 9 January 2024 at 14.30 hours

by

Ruiyin Li

born on 11 October 1994
in Hubei, China

Supervisors
Prof. P. Avgeriou
Prof. P. Liang

Assessment Committee
Prof. A. Capiluppi
Prof. A.N. Chatzigeorgiou
Prof. H. Muccini

The research reported in this thesis has been conducted in the Software Engineering and Architecture group of the Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence of the University of Groningen, The Netherlands.

Understanding, Analysis, and Handling of Software Architecture Erosion
Ruiyin Li

To my parents

Abstract

Architecture erosion reflects the tendency of an implemented architecture of a software system to gradually diverge from the intended architecture. Empirical evidence from numerous studies has demonstrated that architecture erosion significantly impacts various aspects of software development, maintenance, and evolution. This is because, architecture erosion often tends to occur imperceptibly and accumulates over time, making its repair challenging, costly, and sometimes even impossible. Therefore, timely detection and remediation of architecture erosion become crucial.

One way to manage architecture erosion is by identifying its early *symptoms*, such as lack of modularity and various architectural smells. By identifying and managing these symptoms and their evolution, developers can gain insights into the software system's health and take proactive measures like architecture refactoring. This early warning mechanism not only aids in repairing eroded architecture, but also helps in understanding, identifying, analyzing, and optimizing software architecture, ultimately improving software product quality. However, despite several research studies investigating the architecture erosion phenomenon, the current state of the art has certain shortcomings. Specifically, there is a lack of comprehensive understanding of the nature of architecture erosion, it is unclear which symptoms of architecture erosion are the most common, and there is a lack of effective methods to identify these symptoms. Hence, **the main research objective of this thesis is to establish a landscape of architecture erosion, investigate common erosion symptoms, and propose feasible approaches to identify and handle architecture erosion**.

To achieve the stated objective, we first need to obtain a landscape of the architecture erosion phenomenon and its current state of research in the literature. To this end, we conducted a systematic mapping study that covers the literature spanning

from January 2006 to May 2019 for a comprehensive understanding of architecture erosion, including its definitions, symptoms, causes, and consequences. The main results show that (1) "*architecture erosion*" is the most frequently-used term followed by "*architecture decay*". Four perspectives (i.e., violation, structure, quality, and evolution) regarding the definition of architecture erosion are worthy of investigation in both research and practice, along with the respective four types of erosion symptoms. (2) Non-technical reasons contribute to architecture erosion alongside technical reasons. Architecture erosion has negative impacts on software quality attributes, and practitioners can advocate for management intervention to prioritize addressing architecture erosion and prevent potential system failures. (3) Approaches and tools for detecting and addressing architecture erosion are categorized into 19 and 35 categories, respectively, with consistency-based and evolution-based approaches being commonly mentioned.

Subsequently, in order to gain insights into the state of practice, we conducted an empirical study that aimed to explore how developers perceive and discuss the phenomenon of architecture erosion. To collect relevant information from the perspective of practitioners, we utilized three primary data sources: developers' online communities, surveys, and interviews. This comprehensive approach allowed us to gather diverse perspectives and obtain a deeper understanding of the architecture erosion phenomenon. The findings reveal that despite the absence of dedicated tools for detecting architecture erosion, developers can utilize associated practices (e.g., code review, architecture conformance checking) and tools (e.g., Lattix) to identify the symptoms of architecture erosion. Evidence indicated that the collected measures from practitioners (e.g., architecture assessment, periodic maintenance) can be utilized during architecture implementation to effectively address architecture erosion.

After obtaining a comprehensive understanding of the state of research and practice, we decided to focus on a prevalent practice (i.e., code review) to analyze architecture erosion. Specifically, we chose code review comments, as a type of textual artifact produced in code review, providing a window into analyzing developers' practical understanding of erosion symptoms. We conducted two empirical studies to delve deeper into common erosion symptoms during software development. The first study focused on architecture erosion symptoms in code reviews, analyzing discussions from the Nova and Neutron projects in OpenStack. The results revealed that the most frequently identified erosion symptoms are architectural violation, duplicate functionality, and cyclic dependency. The number of comments on erosion symptoms decreased over time, indicating increased stability in the architecture. Most erosion symptoms were addressed by fixing or abandoning them after review votes. Building upon these findings, the second study further explored the most

frequent violation symptoms, as these are the most immediate symptoms of architecture erosion and they require the most attention from practitioners. We collected and analyzed 606 code review comments from four popular open-source projects (i.e., Nova, Neutron, Qt Base, and Qt Creator) to study violation symptoms. The findings show that developers discuss 10 categories of violation symptoms during code review. The primary measures employed to address violation symptoms are refactoring and removing code, accounting for 90% of the cases, while some symptoms were disregarded by developers.

Considering the limitations of existing tools on identifying symptoms of architecture erosion, we sought to explore the possibility of automatically identifying violation symptoms from textual artifacts in practice. We developed 15 machine learning-based and 4 deep learning-based classifiers using three pre-trained word embeddings to identify violation symptoms of architecture erosion from developer discussions in code reviews. The results indicate that (1) the SVM classifier, utilizing the *word2vec* pre-trained word embedding, achieved the highest performance with an F1-score of 0.779; (2) classifiers employing the *fastText* pre-trained word embedding model achieved favorable performance; (3) classifiers using 200-dimensional pre-trained word embeddings outperformed those using 100 or 300-dimensional models; (4) through employing a majority voting strategy, an ensemble classifier improved performance and surpassed individual classifiers; and (5) practitioners perceive the classifiers' results as valuable, confirming the practical potential of automated identification of violation symptoms.

After proposing a solution for the identification of architecture violations in code reviews, we proceeded to design an approach aimed at mitigating the impact of human factors, and especially the problem of unqualified code reviewers. To this end, we proposed an automated recommendation for code reviewers who are qualified to review architecture violations based on reviews of code changes. Specifically, we utilized three widely adopted similarity detection methods to measure the file path similarity and the semantic similarity of review comments. Through a series of experiments, we evaluated these methods separately and compared them with the baseline approach (RevFinder). The results proved that the common similarity detection methods exhibited acceptable performance scores and outperformed RevFinder in recommending code reviewers for architecture violations. Furthermore, we discovered that the sampling techniques used in recommending code reviewers have an impact on the performance of reviewer recommendation approaches.

Samenvatting

Architectuurerosie weerspiegelt de neiging van een geïmplementeerde architectuur van een softwaresysteem om geleidelijk af te wijken van de bedoelde architectuur. Empirisch bewijs uit talrijke studies heeft aangetoond dat architectuurerosie aanzienlijk invloed heeft op verschillende aspecten van softwareontwikkeling, onderhoud en evolutie. Dit komt omdat architectuurerosie vaak onmerkbaar optreedt en zich in de loop van de tijd ophoopt, waardoor reparatie uitdagend, kostbaar en soms zelfs onmogelijk is. Daarom wordt tijdige detectie en herstel van architectuurerosie cruciaal.

Een manier om architectuurerosie te beheren is door vroegtijdige *symptomen* te identificeren, zoals gebrek aan modulariteit en verschillende architecturale geuren. Door deze symptomen en hun evolutie te identificeren en te beheren, kunnen ontwikkelaars inzicht krijgen in de gezondheid van het softwaresysteem en proactieve maatregelen nemen, zoals architectuurherstructurering. Dit vroege waarschuwingssysteem helpt niet alleen bij het repareren van eroderende architectuur, maar helpt ook bij het begrijpen, identificeren, analyseren en optimaliseren van softwarearchitectuur, en uiteindelijk bij het verbeteren van de softwareproductkwaliteit. Ondanks verschillende onderzoeken naar het fenomeen van architectuurerosie, heeft de huidige stand van de techniek bepaalde tekortkomingen. Specifiek, er is een gebrek aan een alomvattend begrip van de aard van architectuurerosie, het is onduidelijk welke symptomen van architectuurerosie het meest voorkomen, en er is een gebrek aan effectieve methoden om deze symptomen te identificeren. Daarom is **het belangrijkste onderzoeksdoel van deze scriptie het vaststellen van een landschap van architectuurerosie, het onderzoeken van veelvoorkomende erosiesymptomen en het voorstellen van haalbare benaderingen om architectuurerosie te identificeren en aan te pakken**.

Om het gestelde doel te bereiken, moeten we eerst een landschap verkrijgen van

het fenomeen architectuurerosie en de huidige staat van onderzoek in de literatuur. Hiervoor hebben we een systematische mappingstudie uitgevoerd die de literatuur beslaat van januari 2006 tot mei 2019 voor een alomvattend begrip van architectuurerosie, inclusief de definities, symptomen, oorzaken en gevolgen. De belangrijkste resultaten tonen aan dat (1) "*architectuurerosie*" de meest gebruikte term is gevolgd door "*architectuur verval*". Vier perspectieven (d.w.z. schending, structuur, kwaliteit en evolutie) met betrekking tot de definitie van architectuurerosie zijn het waard om te onderzoeken in zowel onderzoek als praktijk, samen met de respectievelijke vier soorten erosiesymptomen. (2) Niet-technische redenen dragen bij aan architectuurerosie naast technische redenen. Architectuurerosie heeft negatieve effecten op de kwaliteitskenmerken van software, en praktijkmensen kunnen pleiten voor managementinterventie om prioriteit te geven aan het aanpakken van architectuurerosie en mogelijke systeemfouten te voorkomen. (3) Benaderingen en tools voor het detecteren en aanpakken van architectuurerosie zijn gecategoriseerd in 19 en 35 categorieën, respectievelijk, met consistentie-gebaseerde en evolutie-gebaseerde benaderingen die vaak worden genoemd.

Vervolgens, om inzicht te krijgen in de praktijk, hebben we een empirische studie uitgevoerd die tot doel had te onderzoeken hoe ontwikkelaars het fenomeen van architectuurerosie waarnemen en bespreken. Om relevante informatie te verzamelen vanuit het perspectief van de beoefenaars, hebben we drie primaire gegevensbronnen gebruikt: online gemeenschappen van ontwikkelaars, enquêtes en interviews. Deze uitgebreide benadering stelde ons in staat om diverse perspectieven te verzamelen en een dieper inzicht te krijgen in het fenomeen van de architectuurerosie. De bevindingen onthullen dat ondanks de afwezigheid van specifieke tools voor het detecteren van architectuurerosie, ontwikkelaars geassocieerde praktijken (bijv. code review, architectuurconformiteitscontrole) en tools (bijv. Lattix) kunnen gebruiken om de symptomen van architectuurerosie te identificeren. Bewijs toonde aan dat de verzamelde maatregelen van beoefenaars (bijv. architectuurbeoordeling, periodiek onderhoud) kunnen worden gebruikt tijdens de implementatie van de architectuur om architectuurerosie effectief aan te pakken.

Na een uitgebreid begrip van de stand van onderzoek en praktijk te hebben verkregen, besloten we ons te richten op een gangbare praktijk (d.w.z. code review) om architectuurerosie te analyseren. Specifiek kozen we voor code reviewopmerkingen, als een soort tekstueel artefact dat wordt geproduceerd in code review, waardoor we een kijkje krijgen in de praktische kennis van ontwikkelaars over erosiesymptomen. We hebben twee empirische studies uitgevoerd om dieper in te gaan op de gangbare erosiesymptomen tijdens softwareontwikkeling. De eerste studie richtte zich op symptomen van architectuurerosie in code reviews, en analyseerde discussies uit de Nova- en Neutron-projecten in OpenStack. De resul-

taten onthulden dat de meest frequent geïdentificeerde erosiesymptomen architecturale schendingen, duplicatie van functionaliteiten en cyclische afhankelijkheden zijn. Het aantal opmerkingen over erosiesymptomen nam in de loop van de tijd af, wat duidt op een verhoogde stabiliteit in de architectuur. De meeste erosiesymptomen werden aangepakt door ze te repareren of te laten varen na review-stemmen. Voortbouwend op deze bevindingen, heeft de tweede studie de meest voorkomende schendingssymptomen verder verkend, aangezien deze de meest onmiddellijke symptomen van architectuurerosie zijn en ze de meeste aandacht van de beoefenaars vereisen. We verzamelden en analyseerden 606 code review-opmerkingen van vier populaire open-source projecten (d.w.z. Nova, Neutron, Qt Base en Qt Creator) om schendingssymptomen te bestuderen. De bevindingen tonen aan dat ontwikkelaars tijdens code review 10 categorieën van schendingssymptomen bespreken. De voornaamste maatregelen om schendingssymptomen aan te pakken zijn refactoring en het verwijderen van code, goed voor 90% van de gevallen, terwijl sommige symptomen door ontwikkelaars werden genegeerd.

Gezien de beperkingen van bestaande tools bij het identificeren van symptomen van architectuurerosie, hebben we gezocht naar de mogelijkheid om automatisch schendingssymptomen te identificeren uit tekstuele artefacten in de praktijk. We hebben 15 machine learning-gebaseerde en 4 deep learning-gebaseerde classificatoren ontwikkeld met behulp van drie voorgeleerde word embeddings om schendingssymptomen van architectuurerosie te identificeren uit discussies van ontwikkelaars in code reviews. De resultaten geven aan dat (1) de SVM-classificator, die gebruikmaakt van de voorgeleerde *word2vec* word embedding, de hoogste prestaties behaalde met een F1-score van 0.779; (2) classificatoren die het *fastText* voorgeleerde word embedding-model gebruiken, presteerden gunstig; (3) classificatoren die 200-dimensionale voorgeleerde word embeddings gebruikten, presteerden beter dan die met 100 of 300-dimensionale modellen; (4) door het gebruik van een meerderheidsstemstrategie verbeterde een ensemble-classificator de prestaties en overtrof individuele classificatoren; en (5) beoefenaars zien de resultaten van de classificatoren als waardevol, wat het praktische potentieel van geautomatiseerde identificatie van schendingssymptomen bevestigt.

Na een oplossing te hebben voorgesteld voor de identificatie van architectuurschendingen in code reviews, gingen we verder met het ontwerpen van een aanpak die gericht was op het beperken van de impact van menselijke factoren, en in het bijzonder het probleem van niet-gekwalificeerde code reviewers. Hiervoor stelden we een geautomatiseerde aanbeveling voor van code reviewers die gekwalificeerd zijn om architectuurschendingen te reviewen op basis van reviews van codeveranderingen. Specifiek hebben we drie veelgebruikte gelijkenisdetectiemethoden gebruikt om de bestandspadgelijkenis en de semantische gelijkenis van review-opmerkingen

te meten. Door middel van een reeks experimenten hebben we deze methoden afzonderlijk geëvalueerd en vergeleken met de basisbenadering (RevFinder). De resultaten toonden aan dat de gangbare gelijkenisdetectiemethoden acceptabele prestatiescores vertoonden en RevFinder overtroffen bij het aanbevelen van code reviewers voor architectuurschendingen. Bovendien ontdekten we dat de samplingtechnieken die worden gebruikt bij het aanbevelen van code reviewers, invloed hebben op de prestaties van reviewer aanbevelingsbenaderingen.

Contents

List of Figures

List of Tables

Acknowledgements

If I were to reflect on my accomplishments in the first half of my life, earning a doctoral degree is definitely one of the significant milestones. Coming to a country as diverse, open, and inclusive as the Netherlands is a decision I will never regret, not to mention that here, I met a lot of lovely friends who left stamps on my life.

First of all, I want to express my sincere and heartfelt gratitude to my supervisor, Professor Paris Avgeriou. His mentorship, guidance, and support have been invaluable in helping me navigate the challenges of conducting research. From the moment he accepted me as his student, he has been a constant source of wisdom, motivation, and support. He has challenged me to think critically, develop my ideas with rigor, and articulate my thoughts with clarity. His feedback and perspective have helped shape my research and refine my thinking. Beyond his academic expertise, he has also shown great care and kindness. I am so grateful for the opportunity to have worked with him.

Meanwhile, I would like to sincerely extend thanks to my co-supervisor, Professor Peng Liang. As one of the helmsmen of my scientific research during my doctoral phase, Peng has not only guided me in academic research and scientific writing but also provided me with care in my life. His rigorous attitude, patience, and humility have deeply influenced me. The road of scientific research never goes smoothly. Whenever I encountered problems and felt lost, he always encouraged me through the examples of domestic and international scholars, helping me regain my confidence and continue my research.

I consider myself fortunate to have completed my Ph.D. under the guidance of these two excellent supervisors. The knowledge, skills, and experiences I have gained during this period will undoubtedly serve me well throughout my life. I am forever grateful for the roles my supervisors have played in shaping me as a researcher, scholar, and the person I am today.

I would like to thank all the members of the Software Engineering and Architecture (SEARCH) group of the Bernoulli Institute: Yikun Li, Jie Tan, Darius Daniel Sas, Cezar Sas, João Paulo Biazotto, Sahar Ahmadisakha, Zaki Pauzi, Tien Rahayu Tulili, Héctor Fabio Cadavid Rengifo, Emeralda Sesari, Edi Sutoyo, Ayushi Rastogi, Vasilios Andrikopoulos, Andrea Capiluppi, Daniel Feitosa, Tijs van der Storm. Thank you for your company, working together, drinking coffee and having lunch together. Special thanks to Mohamed A.M. Soliman, He was my daily supervisor at the beginning of my Ph.D. journey in Groningen and helped me a lot. Besides, I also want to thank the members from Wuhan University for their support and company: Fangchao Tian, Tingting Bi, Zhuang Xiong, Tianlu Wang, Muhammad Waseem, Xueying Li, Xiaofeng Han, Liming Fu, Beiqi Zhang, and other members.

Lastly, but certainly not least, I am also grateful for the unwavering support from my parents. Despite the geographical distance, your emotional support and love have been my pillars of strength. In an era where technology bridges distances, your presence has been a constant source of comfort. You have always been there to lend an ear, offer sage advice, and provide encouragement during challenging times. Thank you for being my steadfast rock throughout my Ph.d. journey and for your firm belief in me.

Ruiyin Li
Groningen
July 18, 2023

Chapter 1
Introduction

Ideally, throughout the life cycle of a software system, its architecture should undergo gradual, "*organic*" modifications to accommodate new requirements and adapt to environmental changes. This implies that the evolution of the architecture should proceed in parallel to the evolution of the software system (Bass et al., 2021). However, in reality, the surge in software complexity and the perpetual fluctuation of requirements tends to lead to architecture erosion over time (Perry and Wolf, 1992).

This chapter elaborates on the main concepts of this Ph.D. thesis, the context of the study, and describes the overall research design. Section 1.1 introduces the concepts of architecture erosion and its symptoms. Section 1.2 presents the potential causes and impact of architecture erosion. Section 1.3 describes some background knowledge regarding mining architecture information from textual artifacts. Section 1.4 provides an overview of the problem statement and its decomposition, as well as the research methods employed in this thesis. Finally, Section 1.5 presents the organization of the remainder of this dissertation.

1.1 Architecture Erosion and Its Symptoms

1.1.1 Architecture and Its Erosion

Software architecture usually refers to the overarching framework of a software system, which describes the structural components of the system (i.e., the architectural elements) and their relationships and attributes that meet system requirements and overall quality. In the past few decades, different definitions of software architecture appeared from different perspectives. For example, Perry and Wolf (Perry and Wolf, 1992) define software architecture as "*a set of architectural elements that have a particular form*"; Booch (Booch, 2005) stated that "an architecture is the set of significant decisions about the organization of a software system"; ISO/IEC 42010 defines software architecture as "*the fundamental concepts or properties of a system in its environment embodied in its elements, their relationships, and in the principles of its design and*

evolution". In this thesis, we adopt a widely used definition (Clements et al., 2010; Bass et al., 2021):

> "*The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.*"

The design of a software system takes into account multiple factors, including requirements, budgetary considerations, time constraints, technical limitations, etc. During the software development life cycle, the emergence of new requirements and issues prompts a series of modifications to the system. The cumulative modifications can gradually compromise the structural integrity of the software system, undermining its fundamental design principles. Consequently, the implemented architecture gradually diverges from the intended architecture, giving rise to what is commonly called *architecture erosion*.

The phenomenon of architecture erosion was initially brought to light in 1992 by Perry and Wolf alongside their definition of software architecture (Perry and Wolf, 1992). Since then, researchers and practitioners increasingly recognized the existence of architecture erosion, employing various terms to describe this phenomenon, such as software erosion (De Silva and Balasubramaniam, 2012; Dalgarno, 2009), design erosion (Baum et al., 2018), design decay (Izurieta and Bieman, 2013), architecture degeneration (Li and Long, 2011), architectural decay (Hassaine et al., 2012), code decay (Bandi et al., 2015), and modular deterioration (Rama, 2010).

As a practical example of architecture erosion, we consider a famous case, namely the Mozilla Web browser (Godfrey and Lee, 2000; van Gurp and Bosch, 2002). Mozilla is an application from Netscape comprised of over 2,000,000 lines of code. Netscape engineers invested six months in assessing the architecture erosion and irreparable state of the initial architecture of the application, as new requirements caused changes that gradually violated the optimal design decisions in the original version. Subsequently, a two-year effort was dedicated to the redevelopment of the Mozilla Web browser, as it was not carefully architected at the beginning. This example implies the significance of addressing architecture erosion, as neglecting it could lead to substantial expenditures of time and labor costs.

1.1.2 Erosion Symptoms

In many cases, by the time developers become aware of the occurrence of architecture erosion, it will have already made a significantly negative impact on software development and maintenance. Consequently, effectively addressing architecture erosion becomes a challenging task, often proving to be exceedingly difficult and

sometimes even impossible (considering the costs) (De Silva and Balasubramaniam, 2012). Hence, a more sensible way is to detect and (at least partially) fix the eroded architecture as early as possible (Macia, Arcoverde, Garcia, Chavez and von Staa, 2012). One feasible means to achieve this, is to identify early-stage *symptoms* of architecture erosion.

Symptoms of architecture erosion have been explicitly investigated in the literature, as signs or indicators of the advent of erosion. For example, common symptoms of architecture erosion include code anomaly agglomeration (Oizumi et al., 2016, 2015), lack of modularity (Macia, Garcia, Popescu, Garcia, Medvidovic and von Staa, 2012), various architectural smells (Le et al., 2016, 2018), etc.

Through the identification of architecture erosion symptoms and monitoring their evolutionary trends, such as changes in density and quantity, developers can gain valuable insights into the overall health status of a software system and then mitigate (or repair) the erosion (De Silva and Balasubramaniam, 2012). This can be achieved, for instance, by measuring the density of erosion symptoms in different modules and taking proactive measures (e.g., through refactoring) when a certain threshold is exceeded. Identifying and analyzing architecture erosion symptoms can serve as an early warning mechanism for software engineers to repair the eroded architecture. More importantly, it can assist researchers and practitioners in better understanding, identifying, analyzing, and optimizing a software architecture, ultimately resulting in improving software product quality.

1.2 Causes and Impact of Architecture Erosion

Software evolution is an inherently dynamic process, characterized by the gradual erosion of system architecture over time (Merkle, 2010). Architecture erosion may occur in each stage of the software development life cycle and has varying impacts on the development speed and the cost of maintenance. Previous studies have investigated the factors leading to architecture erosion (De Silva and Balasubramaniam, 2012; Zhang et al., 2011; Macia, Garcia, Popescu, Garcia, Medvidovic and von Staa, 2012). We briefly introduce three common causes of architectural erosion in the following:

Architectural violations are one of the main causes of architecture erosion, as mentioned by Perry and Wolf when they proposed the concept of architecture erosion (Perry and Wolf, 1992). There are many different types of architecture violations, for example, code changes that violate dependency rules (e.g., communication rules, consistency rules) between different layers of the system architecture (Mendoza et al., 2021; Macia, Arcoverde, Garcia, Chavez and von Staa, 2012). Furthermore, architectural violations occur when the architecture is implemented without

strict adherence to design decisions and architectural guidelines.

The accumulation of *technical debt* is another common cause of architectural erosion. Technical debt is a metaphor that reflects technical compromises that yield short-term benefits but may harm the long-term health of a software system (Avgeriou et al., 2016). For instance, this happens when new products are rushed to market due to product launch time pressure (Merkle, 2010; Mair et al., 2014; Stal, 2014; Bhattacharya and Perry, 2007). Such "shortcuts" may change architecturally relevant elements (e.g., modules) and break architectural integrity (De Silva and Balasubramaniam, 2012; Feilkas et al., 2009), e.g., by breaking encapsulation rules and introducing unneeded dependencies, or deliberately introducing technical debt.

In addition, software development is a knowledge-intensive activity, and *knowledge vaporization* related to development is generally considered to be one of the main causes of architecture erosion. For example, lack of documentation of design decisions and their rationale or assumptions, hinders subsequent architectural enhancements and modifications (Feilkas et al., 2009; Strasser et al., 2014). Developer turnover exacerbates architecture erosion due to lost knowledge about system requirements and architectural decisions (Mair et al., 2014; de Oliveira Barros et al., 2015; Reimanis and Izurieta, 2019).

Empirical evidence from numerous studies has demonstrated that architecture erosion significantly impacts various aspects of software development, maintenance, and evolution (De Silva and Balasubramaniam, 2012). The quality degradation of software systems is a very common consequence, as eroded architecture contributes to the deterioration of system structure, diminished system flexibility, and reduced degree of modularization of systems (Izurieta and Bieman, 2013; De Silva and Balasubramaniam, 2012). In addition, in the case of legacy systems, architecture erosion poses obstacles to developers' comprehension and maintenance of the software (Rama, 2010). Eroded architectures with implicit dependencies raise challenges for testers and maintainers, as preserving these dependencies while making modifications can be arduous (Stal, 2014). Developers may not fully understand the ramifications of breaking such dependencies, setting off a chain reaction of unintended consequences (Uchôa et al., 2020). Besides, the increased complexity of eroded architectures, such as an excess of circular dependencies, significantly impedes development and iteration speed. Activities, such as debugging or adding new functions, can become painstakingly slow or even almost stagnate. Consequently, additional time and labor costs must be invested in activities like maintenance and refactoring, potentially prolonging the software product's time-to-market and exceeding the budgeted development costs (Dalgarno, 2009).

1.3 Mining Architecture Information from Textual Artifacts

The entire software development life cycle is accompanied by generating a large number of textual artifacts, such as source code, code comments, bug reports, documentation, code review comments, developer mailing lists, and so on. These textual artifacts encapsulate a wealth of knowledge pertaining to software development. As architecture design is a knowledge-intensive process, an extensive body of development knowledge relevant to architecture design and implementation can be extracted from these textual artifacts through the application of data mining techniques.

Data mining has emerged as an invaluable and practical approach in the field of software engineering. By extracting and analyzing data from textual artifacts, researchers and practitioners can gain valuable insights and make well-informed decisions throughout the software development life cycle. Data mining has been used in different aspects of software engineering, including defect prediction, requirements analysis, and software traceability.

A specific textual artifact that is used as the main data source in this thesis, is code review comments; these typically document the understanding and discussion of the source code by developers and code reviewers. Code review, as a critical development activity, involves the systematic examination of assigned code to identify defects and enhance software quality (Bacchelli and Bird, 2013). A methodical code review process not only improves the quality of software systems, but also fosters the sharing of development knowledge and prevents the release of unstable and defective products. Over time, code review practices have become increasingly important and have been widely employed in modern software development. These practices have been integrated into modern code review workflows, supported by diverse tools during development. Focusing on code reviews can extract valuable knowledge from unstructured data. The insights from code reviews can empower practitioners and researchers with practical insights, and lead to improved software quality, enhanced decision-making, and more efficient development processes.

1.4 Research Design

1.4.1 Problem Statement

Even though there are currently several research studies that investigated the field of architecture erosion, the current state of the art has three main shortcomings,

elaborated in the following.

First, there is a lack of a systematic overview that comprehensively explores the landscape of the architecture erosion phenomenon. Various terms (e.g., "*architecture erosion*", "*architecture decay*", "*architecture degradation*", and "*architectural degeneration*") were used to describe the architecture erosion phenomenon in previous studies (e.g., van Gurp and Bosch (2002); Mo et al. (2013); Macia, Arcoverde, Garcia, Chavez and von Staa (2012); Hochstein and Lindvall (2005)). However, the meaning of these terms is not always consistent; such an inconsistency might hinder a thorough understanding of architecture erosion and prevent systematic investigation of this phenomenon. Considering the fundamental impact of architecture erosion on software development, especially in terms of maintainability and evolvability (De Silva and Balasubramaniam, 2012), it is imperative for researchers and practitioners to gain a comprehensive understanding of this phenomenon. This includes exploring the underlying causes behind the occurrence of architecture erosion and the corresponding consequences, summarizing feasible countermeasures, and utilizing supported approaches and tools to detect and address architecture erosion systematically.

Second, when studying architecture erosion during development, existing studies tend to primarily focus on source code and rarely utilize textual artifacts, as it is still challenging to effectively search and retrieve architectural knowledge from textual artifacts (Bi, Liang, Tang and Yang, 2018). For example, previous studies have predominantly focused on identifying symptoms of architecture erosion through examining source code to detect and analyze various issues, including architectural smells and dependencies (Oizumi et al., 2019; Le et al., 2018, 2016; Mo et al., 2013; Fontana et al., 2016). However, it is essential to acknowledge that architectural knowledge is not solely contained within source code. While analyzing source code is valuable, there are many architecture concepts, such as design decisions and their rationale that cannot be simply mined from source code (Babar et al., 2009). Therefore, it is worth mining and analyzing practical and useful architectural knowledge pertaining to architecture erosion from textual artifacts. Such information could be supplementary to source code analysis, and using both sources (source code and textual artifacts) can help create a bigger picture of the architecture knowledge that is more than the sum of its parts.

Third, to effectively identify and repair architectural erosion, existing tools suffer from certain limitations, such as limited programming language support, insufficient coverage of architectural erosion symptoms, and restricted functionality and availability (Mumtaz et al., 2021). The code review process and its supporting tools, can effectively complement existing tools and source code analysis techniques regarding the identification and subsequently repair of architectural issues (Balachan-

dran, 2013; Ruangwan et al., 2019). This is because the comments generated during code review provide a wealth of textual information about the architectural changes that developers have identified and discussed during the development process (Paixao et al., 2019). Therefore, code review comments can be examined and leveraged to investigate architecture changes and violations from the viewpoint of the developers.

Summarizing the aforementioned challenges, this thesis addressed the core problem as follows:

> *Current research on architecture erosion is still incomplete and lacks a comprehensive understanding of the nature of architecture erosion. Due to the limitations of existing tools, it is essential and complementary to identify architecture erosion from textual artifacts beyond source code, to gain insights into the practical erosion symptoms and their countermeasures. One textual artifact that is particularly worth exploring for the identification of architectural violations, is code reviews.*

1.4.2 Design Science Framework

Design science was initially introduced by March and Smith (March and Smith, 1995), and refers to the design and investigation of artifacts in context. Later, a design science framework for the field of Software Engineering was developed and refined by Wieringa (Wieringa, 2014). As presented in Figure 1.1, design science concerns two parts: *design* and *investigation*, and they correspond to two kinds of research problems, namely, *design problems* and *knowledge questions*. Specifically, *design problems* necessitate real-world changes and require an analysis of stakeholder goals, whether actual or hypothetical. In contrast, *knowledge questions* do not seek a change in the world; rather, they ask for knowledge about the world as it is (Wieringa, 2014).

The distinction between design problems and knowledge questions often becomes obscured within reports pertaining to design science research. If a request asks for a solution to a design (e.g., design an algorithm to solve certain specific problems), it falls under the category of *design problems*. In these cases, the solutions may not be unique and are evaluated by utilities with respect to the stakeholder goals. If there is a pursuit for answers about *the world* (e.g., whether certain algorithms are useful for certain goals), it belongs to the realm of *knowledge questions*. The answers can be a proposition or a hypothesis, which might be uncertain and require testing through the verification of truth over time.

Design problems are treated by *design cycles* that can be organized in varying ways, and are broken down into three tasks, problem investigation, treatment design, and treatment validation; knowledge questions can be answered by *empirical*

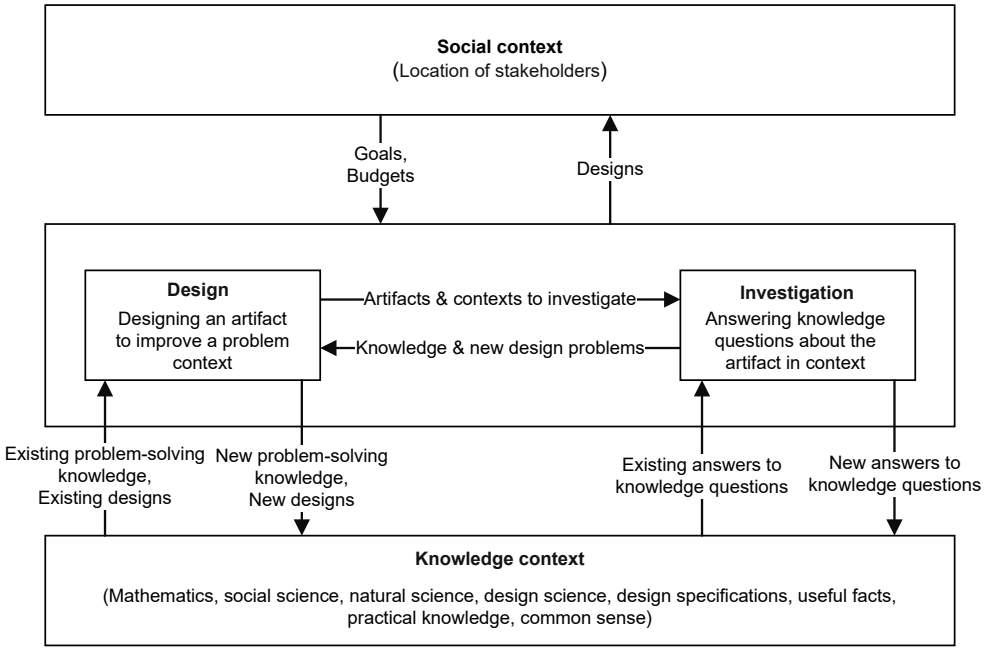

Figure 1.1: The design science framework proposed by Wieringa (Wieringa, 2014)

cycles, utilizing analytical or empirical research methods, such as experiments, case studies, and surveys. Design problems and knowledge questions are usually nested with each other to develop artifacts that can interact with problem contexts to bring about improvements in context. The above characteristics make the design science framework well-suited to design long-term research, such as Ph.D. projects. By decomposing a research question into sub-questions, Ph.D. projects can be divided into fine-grained research questions. In this way, the initial research problem described in Section 1.4.1 can be decomposed into smaller research questions in Section 1.4.3, which become more concrete along with in-depth understanding, and analysis.

1.4.3 Problem Decomposition

This section elaborates on the problem decomposition, that is, how the problem statement is decomposed into design problems and knowledge questions, as well as their interconnections. Figure 1.2 shows the decomposition of the main research problem described in Section 1.4.1. For brevity, both design problems and knowledge questions are marked as *Research Questions* (RQs); for instance, RQ1 denotes the first knowledge question "*What is the current state of the art of architecture ero-*

sion?" and RQ1.1 indicates one of its sub-RQs, "*What is the current understanding of architecture erosion in the literature?*". The thick arrows denote the main research sequence and the thin arrows refer to the further decomposed sub-RQs.

To address the stated problem, the first step is to establish a broad and holistic landscape on the architecture erosion phenomenon for a comprehensive understanding of the nature of architecture erosion; thus, we started our research with the above-mentioned RQ1. To the best of our knowledge, there were no studies that systematically investigate the phenomenon of architecture erosion, such as its definitions, symptoms, causes, and consequences. Therefore, we formulated RQ1.1: "*What is the current understanding of architecture erosion in the literature?*". Subsequently, RQ1.2 (i.e., "*What are the existing supports for addressing architecture erosion in the literature?*") concerns the existing approaches and tools used to identify and address architecture erosion. Finally, RQ1.3 (i.e., "*What are the difficulties and lessons of addressing architecture erosion?*") investigated the challenges and lessons learned on handling architecture erosion, such as dilemmas and trade-offs regarding addressing erosion.

To gain a comprehensive understanding of architecture erosion, it is paramount to not only analyze the existing literature but also to investigate its current state of practice in the industry. This is particularly important as it allows us to ascertain whether a disparity exists between academic research and industry practices. To this end, we formulated RQ2: "*What is the current state of the practice regarding architecture erosion from the developers' perspective?*". RQ2.1 (i.e., "*How do developers discuss architecture erosion in practice?*") investigated how developers perceive this phenomenon and shed light on its manifestation in practice, as well as the potential causes and consequences of architecture erosion based on their own experience. RQ2.2 (i.e., "*What are the existing supports for addressing architecture erosion in practice?*") sought to explore the existing practices and tools to address potential architecture erosion in industrial practice.

The findings from RQ1 and RQ2 unveiled the current status regarding architecture erosion within both research and industry. One of the main findings was that symptoms of architecture erosion have received considerable attention from both researchers and practitioners. More importantly, we found that erosion symptoms can be regarded as useful signs and indicators to identify architecture erosion. However, the existing approaches and tools use source code analysis, and have various limitations in identifying erosion symptoms from source code (Mumtaz et al., 2021). Furthermore, code review is a common means of improving the quality attributes of software systems and combating architecture erosion in practice. Meanwhile, rich and fine-grained discussions on source code and system design are recorded as code review comments, which makes it possible to investigate real-world issues related

to architecture erosion. As mentioned previously, textual artifacts can act as complementary data sources in mining architecture information that cannot be derived from source code. Thus, we decided to investigate erosion symptoms in textual artifacts in order to focus on the developers' understanding of these symptoms in practice. As a type of textual artifact, code review comments contain a large number of implementation discussions during development; thus we decided to focus on this type.

Given the above, to further explore the erosion symptoms from the perspective of practitioners during code reviews, we formulated RQ3: "*How are architecture erosion symptoms discussed in code review comments?*". Subsequently, RQ3.1 (i.e., "*Which symptoms of architecture erosion are frequently identified in code reviews?*") investigated frequently discussed erosion symptoms and raised awareness about them; RQ3.2 (i.e., "*How do architecture erosion symptoms identified in code reviews evolve over time?*") provided valuable insights into the evolution of erosion symptoms throughout the code review process, and consequently contributed to understanding the sustainability and stability of architecture; RQ3.3 (i.e., "*Do the architecture erosion symptoms identified in code reviews get fixed in subsequent code changes?*") focused on the actions that developers might take after identifying erosion symptoms, and explored the extent to which code reviews can assist in removing the identified erosion symptoms and subsequently improving software quality.

One of the main findings of RQ3 revealed that violation symptoms are the most frequently discussed erosion symptoms during code review. Architecture violations are the core reason for architecture erosion, as mentioned by Perry and Wolf (Perry and Wolf, 1992) and many previous studies, e.g., Mendoza et al. (2021); Brunet et al. (2012); Terra et al. (2015); Maffort et al. (2016). They are often gradually accumulated, undermining the quality attributes of systems and impeding their maintainability and sustainability. Thus, we decided to conduct an in-depth study on violation symptoms, by asking RQ4, "*Which violation symptoms are discussed during code review?*". RQ4.1 (i.e., "*What categories of violation symptoms do developers discuss?*") delved deeper into the investigation of categories of violation symptoms (a.k.a architecture violations), which may provide practitioners with valuable insights and guidelines to avoid such violations in practice. In general, violation symptoms in textual artifacts like code reviews are expressed in natural language. In this context, RQ4.2 (i.e., "*How do developers express violation symptoms?*") summarized the linguistic patterns and frequent expressions associated with the identified violation symptoms. Moreover, RQ4.3 (i.e., '*What practices are used by developers to deal with violation symptoms?*") explored how practitioners respond when they encounter violation symptoms during the development process. Specifically, it looks into whether practitioners address the violation symptoms and how they do that.

The first four research questions (RQ1-RQ4) are *knowledge questions* that focus on investigating the architecture erosion phenomenon, including its definitions, symptoms, causes, consequences, and corresponding approaches and tools to address architecture erosion, how practitioners perceive it, and how its symptoms are discussed and handled in practice. Ultimately, the goal of these knowledge questions is to gain a comprehensive understanding of architecture erosion in order to manage it better. By answering RQ3 and RQ4, we established that identifying symptoms of architecture erosion is the prerequisite of its management; but it is labor-intensive and time-consuming to manually identify such symptoms (especially violation symptoms), while no studies have yet explored the use of automated techniques to tackle this problem. Therefore, the next step is to explore the possibility of automatically identifying violation symptoms from textual artifacts in practice; similarly to RQ3 and RQ4 we use code reviews as a data source that is wealthy in architecture violation information. To this end, we formulated RQ5: "*Design classifiers to automatically identify violation symptoms of architecture erosion from code reviews*". RQ5 can be decomposed into a design problem (i.e., RQ5.1 "*Train and select classifiers that can effectively identify violation symptoms from code review comments.*") and a knowledge question (i.e., RQ5.2 "*Are the violation symptoms identified by the trained classifier useful?*"). RQ5.1 aimed to design a solution to automatically and effectively identify architecture violations from code review comments using Machine Learning (ML) and Deep Learning (DL) techniques. In particular, we employed five widely-used ML algorithms and a DL algorithm called TextCNN. These classifiers were trained by leveraging three pre-trained word embeddings (word2vec, fastText, and GloVe). Through a series of experiments, we explored the optimal approach for constructing a high-performing classifier. After that, RQ5.2 was formulated to assess the usefulness of the trained classifier.

The main goal of RQ5 was to leverage automated techniques for the identification of architecture violations during code reviews. However, while these automated techniques can enhance the development and code review process, several crucial human factors still influence code review activities, such as unqualified reviewers and response delays. Hence, we proceeded to design an approach aimed at mitigating the impact of human factors, and particularly finding qualified reviewers. Specifically, we designed a recommendation approach leveraging historical code review data to identify potential, qualified code reviewers, rather than subjectively or randomly inviting code reviewers. Subsequently, we formulated RQ6: "*Recommend qualified code reviewers to handle architecture violations*". RQ6.1 (i.e., "*Explore common similarity detection methods to recommend code reviewers for architecture violations.*") focused on establishing an *expertise* model based on history code review comments and the corresponding file paths, in order to recommend code re-

viewers who are knowledgeable on architecture violations; RQ6.2 (i.e., "*Compare the performance of the proposed approach with existing code reviewer recommendation approaches.*") aimed to compare the proposed approach (i.e., similarity detection methods and their combinations) with an existing approach, RevFinder (Thongtanunam et al., 2015); finally, RQ6.3 (i.e., "*Do the sampling techniques affect the performance of the proposed code reviewer recommendation approach?*") took a further step to investigate whether sampling methods can impact the performance of reviewer recommendation approaches.

1.4.4 Empirical Research Methodology

The previous section decomposes the stated problem into fine-grained knowledge questions and design problems. Each research question is supported by one or more empirical or design cycles, which align with the studies carried out in this Ph.D. project. Table 1.1 maps the empirical research methods to the corresponding research questions, along with the respective sections that describe the empirical study designs in this thesis.

We briefly introduce the empirical methods employed in this thesis.

- **Systematic mapping study** is aimed at providing a comprehensive overview of a specific research area through classification and counting contributions in relation to the categories of that classification (Petersen et al., 2015). A mapping study provides an overview of the scope of the research area, and allows to discover research gaps and trends. In this thesis, a systematic mapping study was employed to answer RQ1, that is, to outline the state of the art on the architecture erosion phenomenon in the literature and unveil the definitions, symptoms, causes, consequences, and corresponding approaches and tools to address architecture erosion.

- **Survey** is a collection of standardized information from a specific population, or some sample from one, usually, but not necessarily by means of a questionnaire or interview (Host et al., 2012). The purpose of a survey is to produce statistics, that is, quantitative or numerical descriptions of some aspects of the study population (Fowler Jr, 2013). In this thesis, surveys were employed to answer RQ2 (personal opinion survey through questionnaires and interviews) and RQ5.2 (technology evaluation survey through questionnaires) with the goal of collecting data for analysis and interpretation.

- **Case study** is an empirical enquiry that investigates a contemporary phenomenon within its real-life context (Yin, 2009). Case studies fit particularly well in software engineering, and they can generate rich and nuanced insights

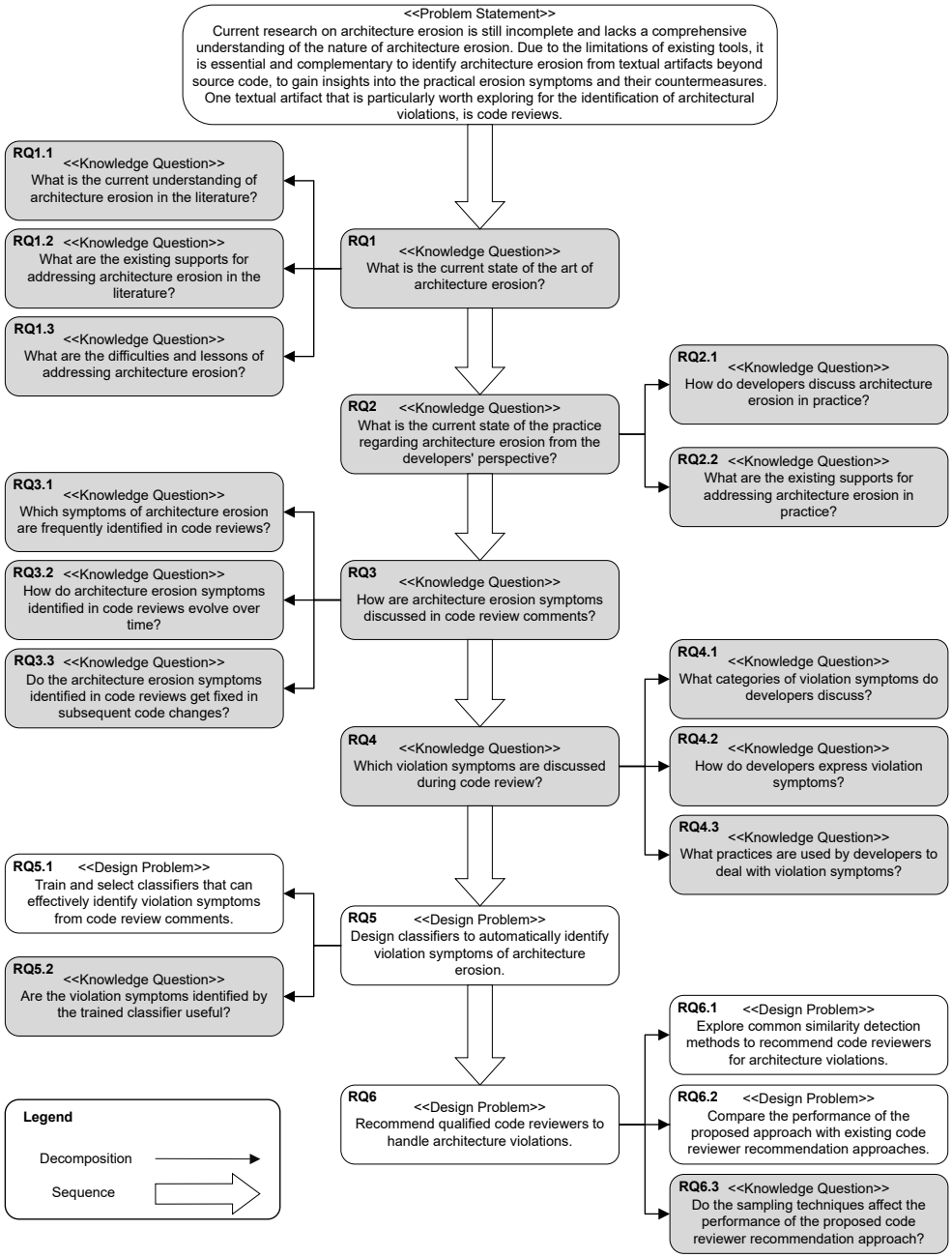

Figure 1.2: Decomposition of the research problem

into software engineering practices, contribute to theory development, and inform decision-making in industry (Host et al., 2012). In this thesis, case studies were used to answer RQ3 and RQ4, with the purpose of investigating the types and evolution trends of erosion symptoms during code review.

- **Experiment** refers to measuring the effects of manipulating one variable on another variable (Host et al., 2012). Experiments in the software engineering domain are suitable to investigate various aspects, including confirming theories and conventional wisdom, evaluating the accuracy of models, and validating measures (Wohlin et al., 2012). In this thesis, experiments were conducted to answer RQ5 and RQ6, in order to determine the effects of different variables (e.g., pre-trained word embeddings and sampling techniques) on the performance of classification models (RQ5) and code reviewer recommendation approaches (RQ6).

Table 1.1: Overview of research methods

RQ	Knowledge Question & Design Problem	Empirical Method	Section
RQ1	What is the current state of the art of architecture erosion?	Systematic mapping study	Section 2.3
RQ2	What is the current state of the practice regarding architecture erosion from the developers' perspective?	Survey	Section 3.3
RQ3	How are architecture erosion symptoms discussed in code review comments?	Case study	Section 4.3
RQ4	Which violation symptoms are discussed during code review?	Case study	Section 5.3
RQ5.1	Train and select classifiers that can effectively identify violation symptoms from code review comments.	Experiment	Section 6.3
RQ5.2	Are the violation symptoms identified by the trained classifier useful?	Survey	Section 6.3
RQ6	Recommend qualified code reviewers to handle architecture violations.	Experiment	Section 7.3

1.5 Overview of the Dissertation

The main body of this dissertation contains six chapters (i.e., Chapters 2 to 7). Table 1.2 shows the research questions and their corresponding chapters, which have been either published in peer-reviewed journals and conferences, or are under review (i.e., Chapter 6). Finally, Chapter 8 concludes the dissertation by summarizing the answers to the six main RQs and further discusses future research directions.

Table 1.2: Overview of dissertation

Research Question	**Chapter**
RQ1: What is the current state of the art of architecture erosion?	Chapter 2
RQ2: What is the current state of the practice regarding architecture erosion from the developers' perspective?	Chapter 3
RQ3: How are architecture erosion symptoms discussed in code review comments?	Chapter 4
RQ4: Which violation symptoms are discussed during code review?	Chapter 5
RQ5: Design classifiers to automatically identify violation symptoms of architecture erosion.	Chapter 6
RQ6: Recommend qualified code reviewers to handle architecture violations.	Chapter 7

Chapter 2 is based on a published paper in the Journal of Software: Evolution and Process (JSEP) (Li, Liang, Soliman and Avgeriou, 2022). This chapter aims to understand and analyze the current state of the art of architecture erosion phenomenon. It covers the literature from January 2006 to May 2019 based on a series of criteria, and the results lay a solid theoretical foundation for follow-up research.

Chapter 3 reports a work accepted by the 29th International Conference on Program Comprehension (ICPC) (Li, Liang, Soliman and Avgeriou, 2021b). This chapter aims at investigating developers' perceptions of architecture erosion in practice, including its causes, consequences, and strategies for its identification and control. Through data source triangulation (Runeson and Höst, 2009), we collected and analyzed qualitative data from six popular developer communities, surveys, and online interviews.

Chapter 4 is based on a peer-reviewed conference paper in the proceedings of the 19th International Conference on Software Architecture (ICSA) (Li, Soliman, Liang and Avgeriou, 2022). This chapter focuses on conducting an empirical study based on discussions regarding architecture erosion symptoms in code reviews. The study

involves collecting and analyzing review comments from the two prominent OpenStack projects, namely Nova and Neutron. The aim of this work is to provide researchers and practitioners with a deeper understanding of commonly discussed erosion symptoms and their evolving trends, along with insights into the actions taken by developers to address them.

Chapter 5 is based on a peer-reviewed journal paper in the Journal of Information and Software Technology (Li et al., 2023d). This chapter focuses on empirically investigating the discussions on violation symptoms of architecture erosion from code review comments. It includes the analysis of violation symptoms in code reviews from four popular OSS projects in the OpenStack (i.e., Nova and Neutron) and Qt (i.e., Qt Base and Qt Creator) communities.

Chapter 6 reports the work that is currently under review in a peer-reviewed journal. This chapter aims at exploring and proposing effective classifiers to automatically identify violation symptoms in code review comments by using several popular ML/DL algorithms based on pre-trained word embeddings. We also surveyed the involved participants who discussed architecture violations in code reviews to validate the usefulness of the classifier.

Chapter 7 is based on a paper published in the 27th International Conference on Evaluation and Assessment in Software Engineering (EASE) (Li et al., 2023a). This chapter focuses on exploring the possibility of using similarity detection methods to recommend code reviewers on architecture violations. Besides, we compared the performance of our approach with one previous approach, and explored the impact of different sampling techniques on the performance of recommendation approaches.

Based on:

Ruiyin Li, Peng Liang, Mohamed Soliman, Paris Avgeriou, (2022) *"Understanding Software Architecture Erosion: A Systematic Mapping Study," Journal of Software: Evolution and Process*, vol. 34, no. 3, e2423, DOI:10.1002/smr.2423

Chapter 2

Understanding Software Architecture Erosion: A Systematic Mapping Study

Abstract

Architecture erosion (AEr) can adversely affect software development and has received significant attention in the last decade. However, there is an absence of a comprehensive understanding of the state of research about the reasons and consequences of AEr, and the countermeasures to address AEr. This work aims at systematically investigating, identifying, and analyzing the reasons, consequences, and ways of detecting and handling AEr. With 73 studies included, the main results are as follows: (1) AEr manifests not only through architectural violations and structural issues but also causing problems in software quality and during software evolution; (2) non-technical reasons that cause AEr should receive the same attention as technical reasons, and practitioners should raise awareness of the grave consequences of AEr, thereby taking actions to tackle AEr-related issues; (3) a spectrum of approaches, tools, and measures has been proposed and employed to detect and tackle AEr; and (4) three categories of difficulties and five categories of lessons learned on tackling AEr were identified. The results can provide researchers with a comprehensive understanding of AEr and help practitioners handle AEr and improve the sustainability of their architecture. More empirical studies are required to investigate the practices of detecting and addressing AEr in industrial settings.

2.1 Introduction

Architecture may exhibit an eroding tendency when changes are accumulated in the software system. As the system evolves, the accumulation of such problems (e.g., architectural violations) can cause the implemented architecture to deviate away from the intended architecture. The phenomenon of divergence between the intended and implemented architecture is regarded as Architecture Erosion (AEr) (Perry

and Wolf, 1992). An eroded architecture can aggravate the brittleness of the system (Perry and Wolf, 1992) and decrease architecture sustainability (Koziolek et al., 2013). For instance, a software system with an eroded architecture may lead to the deterioration of the engineering quality of the system (De Silva and Balasubramaniam, 2012), and make it difficult for developers to understand the internal structure of the system (Perry and Wolf, 1992). Furthermore, AEr might make it very hard to implement new requirements and consequently negatively affect the extensibility of the system (Macia, Arcoverde, Garcia, Chavez and von Staa, 2012).

Due to the severe consequences of AEr, it has been the subject in architecture research since the architecture concept was coined, and the first definition of the concept of AEr was given by Perry and Wolf (Perry and Wolf, 1992) almost 30 years ago. However, the phenomenon of AEr has been described using various terms and definitions in the literature, including software erosion (De Silva and Balasubramaniam, 2012; Dalgarno, 2009), design erosion (Baum et al., 2018), design decay (Izurieta and Bieman, 2013), architecture degeneration (Li and Long, 2011), architectural decay (Hassaine et al., 2012), code decay (Bandi et al., 2015), and modular deterioration (Rama, 2010). Some of these terms focus on erosion at different levels of abstraction, for example, code decay (Bandi et al., 2015) highlights more code anomalies (i.e., at the code level), while modular deterioration (Rama, 2010) concentrates on the modularity of systems (i.e., at the architecture level). These terms indicate that AEr might have an impact on different levels of software systems, and the viewpoints of the aforementioned studies also imply that AEr is a multifaceted phenomenon.

Since the research landscape on this field is diverse and perplexing, there is a need for a comprehensive overview and systematic analysis of the literature on AEr. To this end, we conducted a Systematic Mapping Study (SMS) (Petersen et al., 2015), to investigate the definitions behind the AEr phenomenon, the reasons that incur it, the consequences it imposes, and the research approaches for detecting and handling AEr. This SMS aims at consolidating the existing research results and deriving research directions for future work.

Both SMS and Systematic Literature Review (SLR) are typically employed to survey the literature on a specific topic area. An SLR focuses on investigating, evaluating, and interpreting the available studies related to specific research questions towards a topic or phenomenon (Kitchenham and Charters, 2007), while an SMS provides an overview of a research area to systematically identify and evaluate the evidence in literature (Petersen et al., 2015). One of the main differences between an SMS and an SLR is that an SMS aims to discover the research trends and covers a broad topic in the literature, while an SLR usually has a relatively narrow and in-depth research scope and focuses on specific research questions (Petersen

et al., 2015). Specifically, the AEr phenomenon has an impact on different levels of software systems and affects various aspects in software development (e.g., development activities) (Li, Liang, Soliman and Avgeriou, 2021b). Hence, to establish an overview of this topic area in the context of software development, we decided to conduct an SMS rather than an SLR on the studied topic (i.e., architecture erosion in software development). To this end, we comprehensively summarized the findings of the research questions, discussed potential directions and trends of studies in this domain, provided an overview regarding various aspects related to architecture erosion phenomenon, and proposed a conceptual model of architecture erosion.

The remainder of this chapter is organized as follows: Section 2.2 introduces the context of architecture erosion. Section 2.3 elaborates on the mapping study design and the research questions. Section 2.4 presents the results of each research question. Section 2.5 describes the results of the research questions and their implications for researchers and practitioners. Section 2.6 examines the threats to validity, and Section 2.7 discusses the related work. Section 2.8 summarizes this SMS.

2.2 Context

In this section, we present the context of this SMS by briefly introducing the terms and some characteristics of architecture erosion.

2.2.1 Terms of architecture erosion

As mentioned in Section 2.1, many studies explored the phenomenon of AEr, but they described this phenomenon with different terms. In the seminal paper on software architecture, Perry and Wolf (Perry and Wolf, 1992) coined the concept of architecture erosion and drift and they argued that AEr happens due to the violations of architecture. "Architecture decay" is also a common term used to describe this phenomenon, for example, Behnamghader *et al*. (Behnamghader et al., 2017) explored AEr through a large-scale empirical study of architectural evolution in open-source projects. Dalgarno (Dalgarno, 2009) used "software erosion" to denote the constant internal structural decay of software systems, which manifests that the implemented architecture diverges from the intended architecture. Izurieta and Bieman (Izurieta and Bieman, 2007) used "design decay" to express the deterioration of the internal structure of system design, while they focused more on the decay of design patterns. Besides, some other terms are also used to describe the same phenomenon, such as design erosion (van Gurp and Bosch, 2002), architecture degeneration (Li and Long, 2011), and code decay (Bandi et al., 2013).

In this SMS, we proposed and refined the definition of architecture erosion:

Architecture erosion happens when the implemented architecture violates the intended architecture with flawed internal structure or when architecture becomes resistant to change.

The intended architecture means the conceptual architecture, which is also called *planned architecture*, *as-designed architecture*, and *prescriptive architecture*; the implemented architecture refers to the concrete architecture, which is also named *as-implemented architecture*, *as-built architecture*, *as-realized architecture*, and *descriptive architecture* (Tran and Holt, 1999). Note that, the intended architecture is not a static architecture and it can evolve as new requirements emerge, therefore, the intended architecture refers to the currently desired architecture instead of the initially documented architecture. In a sense, the intended architecture could be regarded as a continuously evolving architecture, and the implemented architecture often evolves at a pace either faster or slower than the intended architecture, influenced by various factors. Besides, AEr does not mean temporary violations of design rules in one or two versions, but refers to a decreasing tendency of the health status of software systems at the architecture level.

2.2.2 Characteristics of architecture erosion

AEr occurs in the software development life cycle, which has the following characteristics:

(1) AEr can exist in different phases of the software development process. AEr in software systems is one of the main reasons and manifestations of software erosion and aging (De Silva and Balasubramaniam, 2012; Parnas, 1994). The maintenance and evolution phases account for a large part of the life cycle of software development, therefore a common view is that erosion starts to creep into a system during these two phases. Actually, the seed of erosion might be sowed in the system once architecture patterns were chosen and AEr may already exist before implementing the detailed design (e.g., Zhang et al. (2011); Reimanis and Izurieta (2019)).

(2) The existence of AEr does not depend on the size of a software system. Some practitioners thought that AEr could only occur in large software systems with complex structures and dependencies, which are harder to understand, but AEr can also exist in small systems (Hochstein and Lindvall, 2005).

(3) AEr has an "incubation period". AEr may have an "incubation period" and be a long-term process (de Oliveira Barros et al., 2015), that is, when the architecture of a software system suffer from erosion intentionally (e.g., incurring various technical debt for short-term benefits) or unintentional (e.g., unconsciously breaking the design principles or architectural constraints), the effects of the potential violations may not emerge immediately, such as drastically decreasing the system

performance. Sometimes, a well-performed software system could already have an eroded internal structure, and the performance is not the only indicator to judge whether AEr happened or not (De Silva and Balasubramaniam, 2012).

(4) Uncertainty in the speed of AEr. Due to the possible "incubation period", there is a conjecture that AEr is a progressive process. Meanwhile, AEr can also happen quickly in software systems (Hochstein and Lindvall, 2005). For example, during the software development process, if newly added components or modifications violate the communication principles among modules specified in design documents, the maintainability of the software system can slide rapidly.

2.3 Mapping study design

This SMS was designed according to the guideline for systematic mapping studies proposed by Petersen *et al*. (Petersen et al., 2015). The design of the SMS is described in the five following sub-sections: (1) research questions, (2) pilot search and selection, (3) formal search and selection, (4) data extraction, and (5) data synthesis.

2.3.1 Research questions

The goal of this SMS formulated based on the Goal-Question-Metric (GQM) approach (Basili et al., 1994) is to **analyze** the primary studies **for the purpose of** analysis and categorization **with respect to** the concept, symptoms, reasons, consequences, detecting, handling, and lessons learned of architecture erosion **from the point of view of** researchers **in the context of** software development. The goal is further decomposed into eight Research Questions (RQs) (see Table 2.1) in order to get a comprehensive view of AEr, and the answers of these RQs can be directly linked to the goal of this SMS.

2.3.2 Pilot search and selection

Before the formal search and selection, we conducted a pilot search and selection to address the potential considerations in the formal SMS (i.e., search terms and time period). These two considerations are elaborated in the following paragraphs.

First, the pilot search can help us settle the appropriate search terms for the formal search (Petersen et al., 2015). As mentioned in Section 2.2.1, we realized that the AEr phenomenon is described in various terms in the literature. Therefore, we selected the synonyms of "erosion" (i.e., topic-related terms) from related studies (see Section 2.2.1) (e.g., "decay", "deterioration", "degradation"), and formulated a trial search string: ("erosion" **OR** "decay" **OR** "degrade" **OR** "degradation" **OR**

Table 2.1: Research questions and their rationale

Research questions	Rationale
RQ1: What are the definitions of architecture erosion in software development? • **RQ1.1: Which terms are used to describe architecture erosion in the definitions?** • **RQ1.2: What perspectives do the architecture erosion definitions concern about?**	Researchers may have different definitions or understanding about the phenomenon of AEr in software development, which are described by various terms and from different perspectives. This might lead to ambiguous interpretation of AEr phenomenon. RQ1 aims to examine which terms are commonly used to define AEr phenomenon and understand AEr phenomenon from different perspectives. By answering RQ1, we can shed light on the common understanding of AEr phenomenon and reach a common definition of AEr.
RQ2: What are the symptoms of architecture erosion in software development?	Various symptoms (e.g., violations of design decisions (Macia, Arcoverde, Garcia, Chavez and von Staa, 2012)) of AEr have been explicitly discussed in the literature, which can be regarded as indicators of AEr. The aim of this RQ is to provide a taxonomy of different AEr symptoms according to their manifestations, and such a taxonomy may provide a base for future work on detecting AEr.
RQ3: What are the reasons that cause architecture erosion in software development?	The phenomenon of AEr does not happen coincidentally, but rather due to specific reasons (e.g., architectural violation (Gurgel et al., 2014)). These have been separately mentioned in literature. By answering this RQ, we want to determine which reasons of AEr are mostly mentioned in the literature. This can increase the awareness for both practitioners and researchers on the reasons of AEr, and consequently support future work on proactively preventing AEr.
RQ4: What are the consequences of architecture erosion in software development?	AEr can have a serious impact on a software system and diverse consequences for different stakeholders (e.g., increasing cost). This RQ aims at identifying the potential consequences of AEr and their impact on software development, which can raise the awareness of related stakeholders on AEr and expose the possible high risks of AEr.

(*Continued*) Table 2.1

Research questions	Rationale
RQ5: What approaches and tools have been used to detect architecture erosion in software development?	Effective approaches and tools have been proposed and used to identify the eroded structure of architecture. Approaches and tools can facilitate the identification of AEr and check the health status of software systems. The answers to this RQ can tell us what feasible approaches and tools can be used to detect AEr, and to some extent inspire researchers to develop new approaches and tools.
RQ6: What measures have been used to address and prevent architecture erosion in software development?	By answering this RQ, we would like to investigate what kinds of measures that have been developed, proposed, and employed in addressing and preventing AEr. The measures may help to prolong the lifetime of systems and save substantial maintenance effort.
RQ7: What are the difficulties when detecting, addressing, and preventing architecture erosion?	The answers to this RQ can shed light on the difficulties and limitations on handling AEr during software development. Additionally, being aware of the difficulties and challenges of handling AEr can help to avoid the pitfalls of AEr and provide a starting point for future research.
RQ8: What are the lessons learned about architecture erosion in software development?	Lessons learned refer to the experience presented in the primary studies on coping with AEr in software development. The answers to this RQ will help researchers and practitioners to obtain such experience and get familiar with the AEr in software development.

"deteriorate" **OR** "deterioration" **OR** "degenerate" **OR** "degeneration") **AND** ("architecture" **OR** "architectural" **OR** "structure" **OR** "structural"). Initially, to retrieve as many related papers as possible, we excluded "software" in the software architecture search terms, but the retrieved results encompassed a large number of irrelevant studies (e.g., biochemistry, civil engineering). Then, we selected determiners (i.e., area-related terms) and added several domain-specific terms, such as "software", "software system", and "software engineering", and formulated another search string: ("software" **OR** "software system" **OR** "software engineering") **AND** ("architecture" **OR** "architectural" **OR** "structure" **OR** "structural") **AND** ("erosion" **OR** "decay" **OR** "degrade" **OR** "degradation" **OR** "deteriorate" **OR** "deterioration" **OR** "degenerate" **OR** "degeneration"). We found that it was precise enough

to use "software" as the domain-specific term. As for the area-related terms, we firstly selected "architecture", "architectural", "structure", "structural", and "system", and then we decided to add "design" as a complementary term after a discussion and a trial search, since the number of retrieved results showed a slight increase, which is not a burden to the selection. More details of the formal search terms are presented in Section 2.3.3.5.

Second, regarding the time period, we initially wanted to choose the search period starting from 1992 when Perry and Wolf published the seminal paper on software architecture and coined the concept of architecture erosion (Perry and Wolf, 1992). We did a trial search in the IEEE Xplore database using the query expression, i.e., (architectural decay **OR** architecture decay **OR** architectural erosion **OR** architecture erosion **OR** architectural degradation **OR** architecture degradation) **AND** software. The results (see Figure 2.1) showed that the total number of papers retrieved was 329, where the number of papers published between 1992 and 2005 was 69 (i.e., 21%) while the rest (i.e., published between 2006 and May 2019) was 260 (i.e., 79%). In addition, in 2006, Shaw *et al*. (Shaw and Clements, 2006) published the milestone paper about the golden age of software architecture in research and practice, which can also be partially reflected from the statistic result in Figure 2.1. We decided to use the Pareto principle (the 80\20 rule) (Kiremire, 2011) to segment the time period, since it is a popular segmentation approach that is widely used in software engineering (e.g., Kiremire (2011); Chaniotaki and Sharma (2021)). Considering the distribution of the trial search results (i.e., following the 80\20 rule) and the milestone paper (Shaw and Clements, 2006), we finally considered 2006 as the starting year of the search period to study architecture erosion. The end search period is settled as May 2019 when we started this SMS.

2.3.3 Formal search and selection

The search strategy is a critical prerequisite to an SMS or SLR, since a well-established search strategy can help researchers to retrieve complete and relevant studies as many as possible. In this section, we firstly describe the process of the formal search and selection, and then we present the search strategy employed in this SMS in three parts: (1) selection criteria, (2) search scope, and (3) search terms. The process of this SMS is illustrated in Figure 2.2.

2.3.3.1 Process

The execution process of this SMS in four phases is shown in Figure 2.2, and we conducted three rounds of study selection (described in Section 2.3.3.3). This SMS

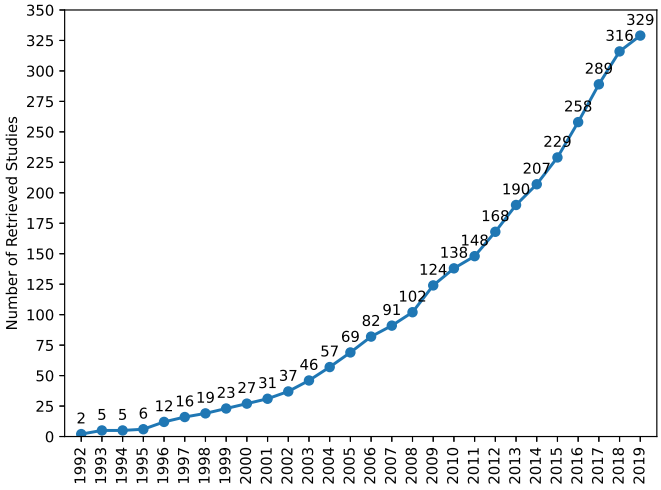

Figure 2.1: Result of the trial search from 1992 to 2019 using the IEEE Xplore database

was conducted by four researchers; the author of this dissertation is the main author and the other researchers are the supervisors.

Phase 1: The author specified the RQs (see Section 2.3.1) and formulated the protocol for this SMS, which were reviewed by the other researchers. Then, a pilot search was conducted to determine the search scope (see Section 2.3.3.4) and search terms (see Section 2.3.3.5).

Phase 2: The author applied the search scope (see Section 2.3.3.4) to conduct the study search in the seven electronic databases (see Table 2.2) and the manual search in the supplementary venues (see Table 2.3). The 1st round selection was conducted in this phase and the included studies of the search results were merged by removing the duplicated studies.

Phase 3: The author conducted the 2nd round selection (by abstract) and the 3rd round selection (by full paper). For the results of the 3rd round selection, any uncertain studies were discussed to reach a consensus between two researchers about whether the studies should be included or not. After that, the author conducted a snowballing in this phase, which followed the same three rounds of selection (see Section 2.3.3.3).

Phase 4: The author extracted data from the selected studies (see Section 2.3.4) according to the data items defined in Table 2.4. Then, we synthesized the extracted data to answer the RQs defined in Section 2.3.1, and we also conducted a pilot extraction to reach an agreement on the controversial data items.

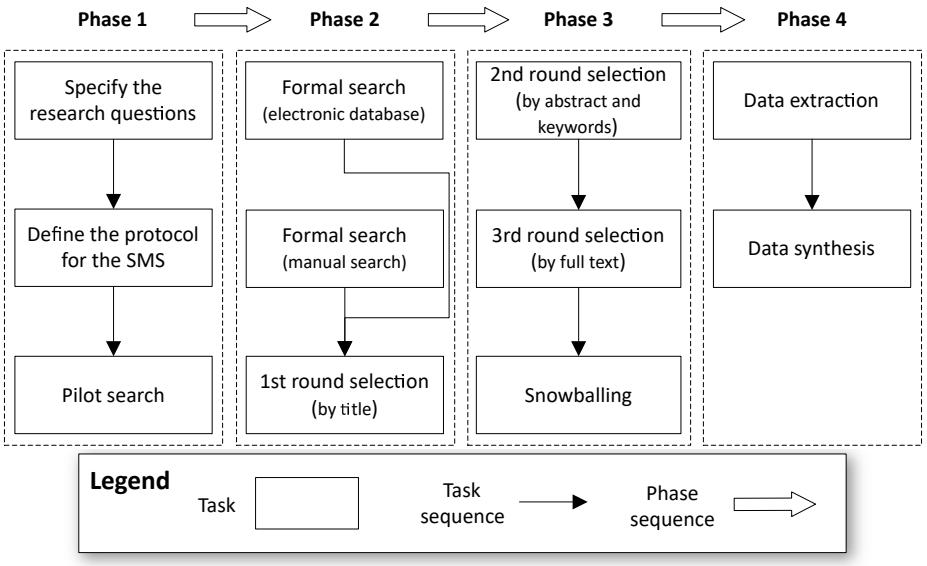

Figure 2.2: The process of this systematic mapping study

2.3.3.2 Selection criteria

Before the study selection, the following inclusion and exclusion criteria were discussed and defined among the researchers after reaching an agreement. The criteria were formulated to select relevant studies for answering the RQs (see Section 2.3.1) in this SMS. Note that, the selection criteria were applied in all study selection tasks, including pilot search, electronic databases, manual search, and snowballing (see Figure 2.2).

(1) Inclusion criteria

- I1: The paper has been peer-reviewed and is available in full text.
- I2: The paper is related to software development and software architecture.
- I3: The paper mentioned the phenomenon or reasons about architecture erosion, as well as the related measures to handle architecture erosion.

(2) Exclusion criteria

- E1: The paper is not written in English.
- E2: The paper is gray literature (e.g., technical reports).

- E3: The content of the paper is only an abstract.
- E4: The paper only mentioned "architecture erosion", but it cannot help to answer the research questions.
- E5: If there are two papers about the same work that were published in different venues (e.g., conference and workshop), the less mature one will be excluded.

2.3.3.3 Details of the study selection

The 1st round selection: The author applied the selection criteria (see Section 2.3.3.2) to identify the primary studies. In the 1st round, the author filtered studies based on the titles to select potential primary studies. In order to mitigate potential bias during study selection, we randomly selected 100 papers as a sample from the search results and a study selection was independently conducted by two researchers. The inter-rater agreement on the 1st round selection in the Cohen's Kappa coefficient (Cohen, 1960) was 0.82. Any uncertain studies (i.e., they could not be decided by the titles) were temporarily included and kept in the 2nd round selection.

The 2nd round selection: The author selected studies left in the 1st round selection by reading their abstracts and keywords. The potentially related studies were selected based on the selection criteria. Similarly to the 1st round selection, to mitigate potential bias, two researchers selected independently the remaining 52 papers left in the sample and the Cohen's Kappa coefficient was 0.64. Note that, the difference in this round is due to one researcher taking a conservative policy in study selection with the purpose of not missing possibly relevant studies. Any uncertain studies (i.e., those that could not be decided by abstracts and keywords) were temporarily included and kept in the 3rd round selection. All the studies from the sample that were included by one researcher and excluded by the other in this round, were eventually excluded in the third round.

The 3rd round selection: The author selected studies by reading the full text of the papers left in the 2nd round selection. The potentially related studies were decided whether they should be finally selected based on the criteria. Again, to mitigate potential bias in this round, two researchers independently read the full-text of the remaining 26 papers left in the sample and the Cohen's Kappa coefficient was 0.92. Note that, in this final round, any uncertain judgments were discussed together and reached an agreement among all the researchers.

Snowballing: To collect more potentially relevant studies about AEr, we employed the "snowballing" method (Wohlin, 2016) to avoid missing any AEr related

studies. Snowballing contains two strategies: backward snowballing and forward snowballing. Backward snowballing refers to the review of the reference list of the included papers, and forward snowballing means to identify the citations to the included papers (Wohlin, 2016). In addition, snowballing is an iterative process: the author checked the references and citations of the studies selected in the 3rd round selection, then re-rechecked the newly included studies by references and citations. This iterative process could stop until there is no any newly selected study. We conducted forward and backward snowballing in Phase 3 (see Figure 2.2), and each iteration entails the aforementioned three rounds of selection (see Section 2.3.3.3). The newly selected studies in the snowballing process were merged into the final results of the study selection.

2.3.3.4 Search scope

(1) Time period

As illustrated in Section 2.3.2, we conducted a pilot search to decide the time period of this SMS. After that, we specified the time period of the published studies between January 2006 and May 2019 (i.e., the starting time of this SMS).

(2) Electronic databases

The electronic databases selected in this SMS are listed in Table 2.2; these are regarded as the most common and popular databases in the software engineering field and the selected databases are considered appropriate to search relevant studies (Chen et al., 2010). We excluded Google Scholar in this SMS, since the precision of the search results was not acceptable and might include many duplicated and irrelevant results. Additionally, the search results of Google Scholar have overlapped with the seven electronic databases and might omit many relevant paper (e.g., newly published studies).

(3) Manual search

To retrieve as many relevant studies as possible, besides the seven electronic databases, we also selected 9 journals, 10 conferences (merged conferences were only counted once, such as WICSA and ICSA), and 2 workshops as supplementary sources to the electronic databases search (see Table 2.3). These supplementary sources are reputable venues that publish research on software engineering and software architecture, and are commonly used in related SMSs and SLRs (e.g., Shahin et al. (2014a); Li et al. (2015)). Note that, the selected journals, conferences, and workshops may not be complete, since they were selected as supplementary rather than exhaustive sources.

Table 2.2: Search string and electronic databases used in this SMS

Search string		
(“software”) **AND** (“architecture” **OR** “architectural” **OR** “system” **OR** “structure” **OR** “structural” **OR** “design”) **AND** (“decay” **OR** “erosion” **OR** “erode” **OR** “degrade” **OR** “degradation” **OR** “deteriorate” **OR** “deterioration” **OR** “degenerate” **OR** “degeneration”)		
#	**Database**	**Search scope in databases**
DB1	ACM Digital Library	Title, Abstract
DB2	EI Compendex	Title, Abstract
DB3	IEEE Xplore	Title, Keywords, Abstract
DB4	ISI Web of Science	Title, Keywords, Abstract
DB5	Springer Link	Title, Abstract
DB6	Science Direct	Title, Keywords, Abstract
DB7	Wiley InterScience	Title, Abstract

Table 2.3: Journals, conferences, and workshops included in this SMS

#	**Journals, Conferences, and Workshops**
J1	ACM Transactions on Software Engineering and Methodology (TOSEM)
J2	IEEE Transactions on Software Engineering (TSE)
J3	Empirical Software Engineering (ESE)
J4	IEEE Software
J5	Information and Software Technology (IST)
J6	Journal of Systems and Software (JSS)
J7	Science of Computer Programming (SCP)
J8	Software Quality Journal (SQJ)
J9	Software Practice and Experience (SPE)
C1	International Conference on Software Engineering (ICSE)
C2	International Conference on Automated Software Engineering (ASE)
C3	ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)
C4	International Symposium on Empirical Software Engineering and Measurement (ESEM)
C5	International Conference on Software Engineering and Knowledge Engineering (SEKE)
C6	European Conference on Software Architecture (ECSA)
C7	Working IEEE/IFIP Conference on Software Architecture (WICSA) International Conference on Software Architecture (ICSA)
C8	IEEE International Conference on Software Maintenance (ICSM) IEEE International Conference on Software Maintenance and Evolution (ICSME)
C9	International Conference on Quality Software (QSIC)

(*Continued*) Table 2.3

#	Journals, Conferences, and Workshops
C10	International Conference on Software Quality, Reliability and Security (QRS) IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER) European Conference on Software Maintenance and Reengineering (CSMR) IEEE Working Conference on Reverse Engineering (WCRE)
W1	Workshop on Software Architecture Erosion and Architectural Consistency (SAEroCon)
W2	International Workshop on Sharing and Reusing Architectural Knowledge (SHARK)

2.3.3.5 Search terms

In this SMS, we defined the search terms according to the PICO criteria (i.e., Population, Intervention, Comparison, and Outcome) (Kitchenham and Charters, 2007). The population in this SMS is the primary studies on software engineering. The intervention is about the topic of "architecture erosion", which can be divided into two types of search terms (i.e., area-related terms and topic-related terms). Through the pilot search and selection (see Section 2.3.2), we chose the area-related terms "architecture", "architectural", "system", "structure", "structural", and "design", and the topic-related terms "decay", "erosion", "erode", "degrade", "degradation", "deteriorate", "deterioration", "degenerate", and "degeneration". Finally, the following query string was used in the formal search. We used Boolean **OR** to join topic-related terms and area-related terms (i.e., synonyms), respectively. we used Boolean **AND** to join the major terms. The Boolean expression for retrieving relevant studies in database search is:

("software") **AND** ("architecture" **OR** "architectural" **OR** "system" **OR** "structure" **OR** "structural" **OR** "design") **AND** ("decay" **OR** "erosion" **OR** "erode" **OR** "degrade" **OR** "degradation" **OR** "deteriorate" **OR** "deterioration" **OR** "degenerate" **OR** "degeneration")

2.3.4 Data extraction

To answer the RQs presented in Section 2.3.1, we extracted and analyzed the data according to the data items (i.e., D8-D16) from each included study. The sixteen data items were discussed and formulated by all the researchers, see Table 2.4, which

also shows the relevant RQs that are supposed to be answered by the extracted data according to the specific data items.

Before the formal data extraction, we discussed the meaning of each data item and the way about how to extract data. To ensure an unambiguous understanding of the data items, a pilot data extraction with five studies was conducted by three researchers. The author extracted data based on the data items from the selected five studies, while another two researchers reviewed the extracted data. Furthermore, we selected another five studies to conduct a sample data extraction by another two researchers independently. By comparing the sample data extraction results from the two researchers, we established that they largely overlap. We provided both the pilot data extraction results and the sample data extraction results in the replication package (Li, Liang, Soliman and Avgeriou, 2021a). Finally, all divergences and ambiguities of the results of the extracted data were discussed together for reaching an agreement. Likewise, in the formal data extraction process, the data extraction was performed by the author and reviewed by another two researchers. Besides, in the process of data synthesis and classification, the extracted data was repeatedly reviewed by another two researchers and any disagreements were discussed between the author and another two researchers. In this way, we can ensure that the extracted data in the formal data extraction process are valid.

2.3.5 Data synthesis

In this process, we conducted data synthesis with the extracted data (see Table 2.4). We employed the descriptive statistics and Constant Comparison method (Adolph et al., 2011) to analyze the qualitative data for answering the eight RQs (see Section 2.3.1). Note that, we employed both descriptive statistics and Constant Comparison as the data analysis methods for answering RQ1, RQ3, RQ4, RQ5, and RQ6. Besides, we also provided examples to clarify the data synthesis results.

Descriptive statistics can provide quantitative summaries based on the initial description of the extracted data; specifically, they were used to analyze the definitions of AEr (i.e., D8), symptoms of AEr (i.e., D9), approaches and tools (i.e., D12 and D13), difficulties (i.e., D15), and lessons learned (i.e., D16). These data can be used to answer RQ1, RQ2, RQ3, RQ4, RQ5, and RQ6, respectively. For instance, we extracted the descriptions of the approaches used to detect AEr from the selected studies to classify the approaches into several categories (see Section 2.4.6) for answering RQ5. We followed the guidelines of Constant Comparison (Adolph et al., 2011; Stol et al., 2016) to analyze the extracted data (see Table 2.5) for answering RQ1, RQ3, RQ4, RQ5, RQ6, RQ7, and RQ8.

In this SMS, Constant Comparison, which is a qualitative data analysis method

Table 2.4: Data items extracted from selected studies

#	Data item name	Description	Relevant RQ
D1	ID	The ID of the study.	Overview
D2	Title	The title of the study.	Overview
D3	Author list	The authors' full names of the study.	Overview
D4	Type of authors	The type of authors (i.e., academia, industry, and both).	Overview
D5	Publication type	The type of the study (i.e., Journal, conference, workshop, or book chapter).	Overview
D6	Publication venue	The name of the venue where the study was published.	Overview
D7	Publication year	The publication year of the study.	Overview
D8	Definition	The definition of architecture erosion.	RQ1
D9	Symptoms	The mentioned symptoms can be regarded as indicators of architecture erosion.	RQ2
D10	Reasons	The reasons that lead to architecture erosion.	RQ3
D11	Consequences	The consequence caused by architecture erosion.	RQ4
D12	Approaches	The approach used to detect architecture erosion.	RQ5
D13	Tools	The tool used to detect architecture erosion.	RQ5
D14	Prevention and remediation	The measure used to address and prevent architecture erosion.	RQ6
D15	Difficulties	The difficulties when detecting, addressing, and preventing architecture erosion.	RQ7
D16	Lessons learned	The lessons learned about architecture erosion.	RQ8

to develop a grounded theory by consistently comparing with the existing findings, was used to generate concepts and categories through a systematic analysis of the qualitative data, including the definitions of AEr (i.e., D8), reasons of AEr (i.e., D10), consequences of AEr (i.e., D11), approaches of AEr detection (i.e., D12), measures of AEr prevention and remediation (i.e., D14), difficulties (i.e., D15), and lessons learned (i.e., D16). The data can be used to answer RQ1, RQ2, RQ3, RQ4, RQ6, RQ7, and RQ8, respectively. The process of Constant Comparison consists of three steps: (1) Initial coding, executed by the author, examining the extracted data line by line to identify the topics of the data. For example, "*Erosion can happen as a result of many different factors, e.g., there can be a lack of architectural documentation, a lack of developer knowledge, ...*" was labelled as "reason" (i.e., D10). (2) Focused coding, executed by the author and reviewed by another two researchers, selecting categories from the most frequent codes and using them to categorize the data. For example, "*architectural erosion is known as having a negative impact on the system quality, such as maintain-*

Table 2.5: Relationship between data items, data analysis method, and research questions

#	Data item name	Data Analysis Methods	RQs
D1-D7	Publication information of selected studies	Descriptive Statistics	Overview
D8	Definition of AEr	Descriptive Statistics & Constant Comparison	RQ1
D9	Symptoms of AEr	Descriptive Statistics	RQ2
D10	Reasons of AEr	Descriptive Statistics & Constant Comparison	RQ3
D11	Consequences of AEr	Descriptive Statistics & Constant Comparison	RQ4
D12	Approaches	Descriptive Statistics & Constant Comparison	RQ5
D13	Tools	Descriptive Statistics	RQ5
D14	Measures of prevention and remediation	Descriptive Statistics & Constant Comparison	RQ6
D15	Difficulties	Constant Comparison	RQ7
D16	Lessons learned	Constant Comparison	RQ8

ability, evolvability, performance, and reliability, ..." is regarded as a type of "quality degradation" and we mapped the collected quality attributes to the software product quality model (i.e., the ISO/IEC 25010 standard (ISO/IEC25010, 2011)). (3) The disagreements on the coding results were checked and settled down by three researchers to reduce potential bias and ambiguity. For example, for answering RQ3, we initially coded a reason that causes AEr "*this problem is at least partially caused by developers' lack of underlying architectural knowledge*" as "*knowledge vaporization*". Then we discussed and agreed that this reason better fits into "*understanding issue*", and finally we got 13 categories of reasons as shown in Table 2.10.

To facilitate the replicability of our SMS, we provide the replication package of this SMS as an online resource (Li, Liang, Soliman and Avgeriou, 2021a). The replication package includes the detailed information of the 73 selected studies (see Appendix A) and the extracted data based on the 16 data items listed in Table 2.4.

2.4 Results

We present the number of searched and selected papers in Section 2.4.1.1, the demographic data of the selected studies in Section 2.4.1.2, and the results of the eight RQs from Section 2.4.2 to Section 2.4.9.

2.4.1 Overview

2.4.1.1 Number of searched and selected papers

The overview of the search, selection, and snowballing results is shown in Figure 2.3. In the study search phase, we collected 41,723 papers, including 20,551 papers from the seven databases (see Table 2.2) and 21,172 papers from the 21 venues (see Table 2.3). In total, 1,220 papers were retained after the first round selection by title, 982 papers were left after removing duplicated papers, 492 papers were kept after the second round selection by abstract, and 68 papers were selected after the third round selection by full text. Then, we conducted snowballing iterations (including forward and backward snowballing) also with the three rounds selection, and 5 more papers were added by snowballing. Ultimately, 73 (i.e., 68+5) papers were finally selected (see Appendix A).

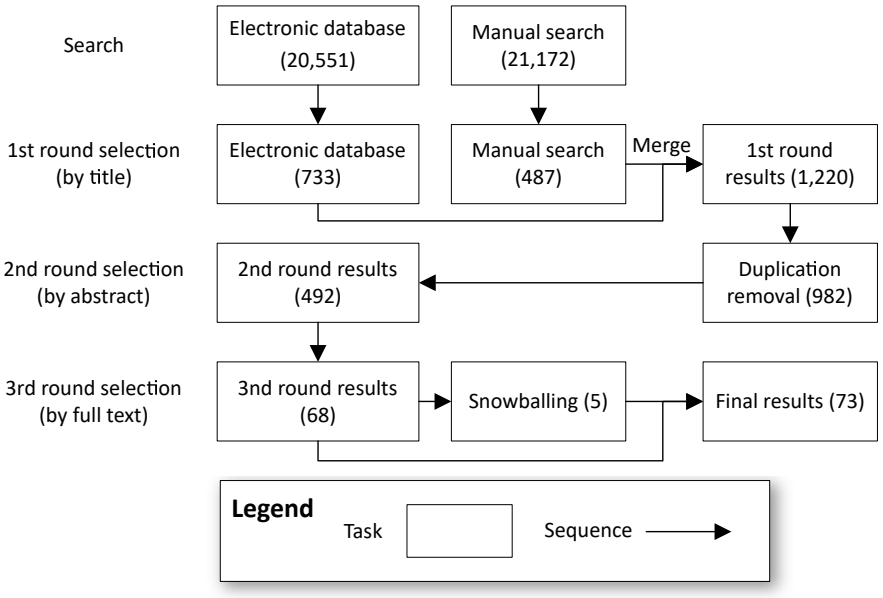

Figure 2.3: Results of search, selection, and snowballing in this SMS

2.4.1.2 Demographic results

We present the statistical information of the selected studies, including types of authors (see Figure 2.4), publication years (see Figure 2.5), and publication venues (see

Figure 2.6).

Figure 2.4 shows that 82.2% of the selected studies (60 out of 73) are authored by researchers from academia, authors of 9.6% of the selected studies (7 out of 73) work in industry, and the rest (8.2%, 6 out of 73) of the selected studies are collaboration outcomes between academia and industry. Additionally, Figure 2.5 shows the distribution of the selected studies over the last 13 years (i.e., from 2006 to 2019), from which we can see a fair amount of attention on architecture erosion from 2011 to 2018 (6-10 papers were published per year) with the peak year around 2013. Note that only two papers published in 2019 were included because we stopped our literature search by May 2019.

As shown in Table 2.6, we list the venues where the selected studies were published, including their names, types, and counts. The 73 studies are published in 49 publication venues (note that merged conferences were only counted once, for example, WICSA and ICSA were counted as one conference). The leading venues are SANER (6 studies including CSMR and WCRE), ICSA (6 studies including WICSA and ICSA), ECSA (5 studies), ICSE (3 studies), JSS (3 studies), and IEEE Software (3 studies). Figure 2.6 provides the distribution of the publication venues of the selected studies. Most of the studies were published in conferences (58.9%, 43 out of 73), followed by journals (27.4%, 20 out of 73), workshops (9.6%, 7 out of 73), and book chapters (4.1%, 3 out of 73).

Table 2.6: Number and proportion of the selected studies

Publication Venue	**Venue Type**	**Count**
International Conference on Software Analysis, Evolution, and Reengineering (SANER) European Conference on Software Maintenance and Reengineering (CSMR) Working Conference on Reverse Engineering (WCRE) Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)	Conference	6
International Conference on Software Architecture (ICSA) Working IEEE/IFIP Conference on Software Architecture (WICSA)	Conference	6
European Conference on Software Architecture (ECSA)	Conference	5
International Conference on Software Engineering (ICSE)	Conference	3
Journal of Systems and Software (JSS)	Journal	3
IEEE Software	Journal	3
International Conference on Modularity (MODULARITY)	Conference	2

(Continued) Table 2.6

Publication Venue	Venue Type	Count
Workshop on Software Architecture Erosion and Architectural Consistency (SAEroCon)	Workshop	2
Software: Practice and Experience (SPE)	Journal	2
IEEE Transactions on Software Engineering (TSE)	Journal	1
Information and Software Technology (IST)	Journal	1
Science of Computer Programming (SCP)	Journal	1
Automated Software Engineering (ASE)	Journal	1
Journal of Object Technology (JOT)	Journal	1
Software Quality Journal (SQJ)	Journal	1
International Journal of Software Engineering and Knowledge Engineering (IJSEKE)	Journal	1
International Journal of Computer, Control, Quantum and Information Engineering (IJCCQIE)	Journal	1
Journal of the Brazilian Computer Society (JBCS)	Journal	1
Electronic Notes in Theoretical Computer Science	Journal	1
Journal of King Saud University-Computer and Information Sciences (JKSUCIS)	Journal	1
Methods and Tools	Journal	1
Relating System Quality and Software Architecture	Book Chapter	1
Agile Software Architecture	Book Chapter	1
Transactions on Foundations for Mastering Change I	Book Chapter	1
Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA)	Conference	1
International Symposium on Foundations of Software Engineering (FSE)	Conference	1
International Conference on Software Maintenance (ICSM)	Conference	1
IEEE International Conference on Program Comprehension (ICPC)	Conference	1
IEEE Working Conference on Mining Software Repositories (MSR)	Conference	1
Asia-Pacific Software Engineering Conference (APSEC)	Conference	1
International Conference on Software Engineering and Knowledge Engineering (SEKE)	Conference	1
International Conference on Software and Systems Reuse (ICSR)	Conference	1
International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)	Conference	1
Annual ACM Symposium on Applied Computing (SAC)	Conference	1

(Continued) Table 2.6

Publication Venue	Venue Type	Count
IEEE Working Conference on Software Visualization (VISSOFT)	Conference	1
International Doctoral Symposium on Components and Architecture (WCOP)	Conference	1
International Conference on Advanced Software Engineering and Its Applications (ASEA)	Conference	1
Annual International Conference on Aspect-oriented Software Development (AOSD)	Conference	1
International Conference on Software Engineering and Data Engineering (SEDE)	Conference	1
IEEE Prognostics and System Health Management Conference (PHM)	Conference	1
Annual India Software Engineering Conference (ISEC)	Conference	1
Conference on Pattern Languages of Programs (PLoP)	Conference	1
Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (SPLASH/OOPSLA)	Conference	1
IEEE Symposium on Computational Intelligence in Dynamic and Uncertain Environments (CIDUE)	Conference	1
International Conference on Advances in ICT for Emerging Regions (ICTer)	Conference	1
International Workshop on Managing Technical Debt (MTD)	Workshop	1
International Workshop on Modeling in Software Engineering (MiSE)	Workshop	1
International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)	Workshop	1
International Conference on Software Architecture Workshops (ICSAW)	Workshop	1
Workshop on Sustainable Architecture: Global Collaboration, Requirements, Analysis (SAGRA)	Workshop	1

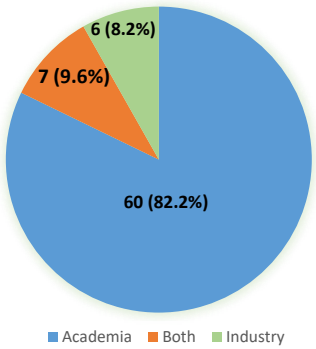

Figure 2.4: Distribution of the selected studies over types of authors

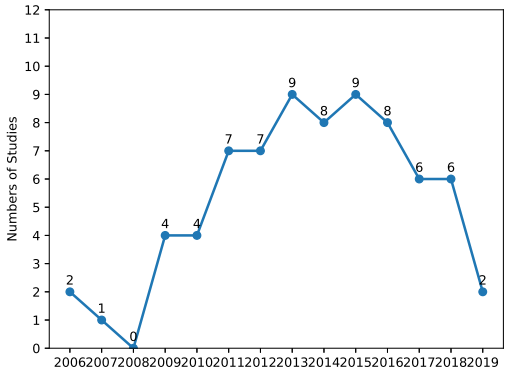

Figure 2.5: Number of the selected studies over time period

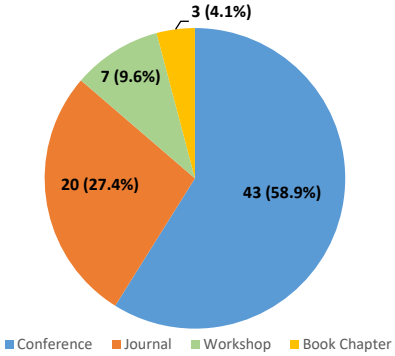

Figure 2.6: Distribution of the selected studies over types of publication

2.4.2 RQ1: What are the definitions of architecture erosion in software development?

2.4.2.1 RQ1.1 Terms of different definitions

As mentioned in Section 2.2.1, the concept of AEr was proposed by Perry and Wolf (Perry and Wolf, 1992) in 1992, which is the earliest study that provided the definition of AEr. In the last decades, many studies described the phenomenon of AEr with different terms, such as architectural decay (e.g., [S64]), architecture degeneration (e.g., [S1]), design erosion (e.g., [S72]), and code decay (e.g., [S28]). Table 2.7 presents all the terms that the selected studies used to describe the phenomenon of AEr. We can see that most studies prefer to use "architecture erosion" to describe this phenomenon, followed by the term "architecture decay". The terms, e.g., "architecture degeneration", "design erosion", "code decay", "software erosion" are also used to represent the phenomenon of AEr. The rest terms are only used in one or two studies to describe the eroded architecture, such as design deterioration (e.g., [S9]) and structure decay (e.g., [S3]).

2.4.2.2 RQ1.2 Perspectives of different definitions

According to the extraction results of data item D8, we analyzed the definitions of AEr and found that researchers described the AEr phenomenon from different perspectives (i.e., understanding AEr phenomenon from different angles). Based on the analysis of the definitions, we classified the definitions into four perspectives (see Table 2.8).

Violation perspective is the most common description of AEr, which denotes that the implemented architecture of a software system violates the design principles or architecture constraints, and consequently results in system problems and the increasing brittleness of the system (Perry and Wolf, 1992). For example, the authors of [S57] stated that "*architectural erosion is the process of introducing changes into a system architecture design that violates the rules of the system's intended architecture*". Violations mainly occur in two phases: (1) Design phase. Developers intentionally or unintentionally change the planned structure of the system architecture due to various reasons (e.g., introducing new design decisions), which gives rise to the violations of original design rules in the design phase of software development (e.g., [S17], [S21]). (2) Maintenance and evolution phase. Developers accidentally or deliberately make modifications (e.g., for short-term benefits, unfamiliar with the dependencies among components) that violate the design time architectural intents during maintenance and evolution activities (e.g., [S47], [S57]).

Table 2.7: Terminologies that the selected studies mentioned to describe architecture erosion

Term	Studies	Count
Architecture erosion	[S2][S3][S4][S6][S7][S8][S12][S13][S14][S18][S20][S22] [S23][S24][S25][S26][S27][S29][S31][S32][S33][S34][S35] [S37][S39][S40][S43][S45][S47][S48][S50][S51][S52][S53] [S55][S57][S58][S60][S61][S63][S65][S66][S67][S69][S70] [S71]	46
Architecture decay	[S7][S11][S15][S16][S17][S19][S21][S37][S42][S44][S49] [S50][S64][S68]	14
Architecture degradation	[S7][S10][S18][S26][S31][S39][S45][S48][S57][S67]	10
Architecture degeneration	[S1][S3][S27][S28][S36][S44][S50]	7
Design erosion	[S1][S9][S15][S18][S55][S62][S72]	7
Software erosion	[S41][S54][S59][S73]	4
Code decay	[S28][S56]	2
Modularity/modular structure deterioration	[S5][S46]	2
design pattern rot/decay/grime	[S9][S30]	2
Architecture deterioration	[S5]	1
Design deterioration	[S9]	1
Structure decay	[S3]	1
Software decay	[S73]	1
Software aging	[S5]	1
System rot	[S72]	1
Design rot	[S7]	1
Code rot	[S73]	1

Structure perspective highlights the structural problems due to constant erosion of the structure of system systems, where the structure of a software system encompasses its components and their relationships to each other (Bass et al., 2021). For example, the authors of [S9] mentioned that "*we define decay as the deterioration of the internal structure of system designs*". From this perspective, the constant erosion of the structure of architecture is derived from the damage to the structure integrity, such as gradually breaking the encapsulation and dependency relationships (e.g., [S9]).

Quality perspective refers to the gradually declining product quality (ISO/IEC25010, 2011) of software systems due to architectural changes or increasing architectural smells (Garcia et al., 2009) (e.g., extraneous connector, ambiguous interface, component overload). A system with an eroded architecture might also have a good runtime performance (De Silva and Balasubramaniam,

Table 2.8: Understanding of architecture erosion phenomenon from four perspectives

Perspective	Description	Studies	Count
Violation	The implemented architecture (as-built) violates the intended architecture (as-planned)	[S6][S7][S9][S12][S13][S14][S15][S16] [S17][S18][S21][S23][S25][S27][S28][S31] [S33][S34][S35][S47][S48][S51][S56][S57] [S58][S60][S63][S65][S71][S73]	30
Structure	Constant erosion of the structure of the system architecture	[S3][S5][S7][S9][S22][S24][S26][S27][S73]	9
Quality	Quality of the software system gradually declines due to architectural changes	[S1][S30][S34][S59][S63][S72]	6
Evolution	Decreasing maintainability makes it hard to change the eroded architecture	[S21][S28][S50]	3

2012) that is one quality attribute of software systems specified in the ISO/IEC 25010 standard (ISO/IEC25010, 2011), while the other quality attributes (e.g., reliability, maintainability) may be negatively affected by AEr. Once the environment or requirements changed, it might bring serious consequences to the software system. For example, the authors of [S34] mentioned that "*we regard erosion as the overall deterioration of the engineering quality of a software system*".

Evolution perspective denotes the loss of flexibility at the architectural level, which manifests a state of architecture that is almost stagnated and becoming harder to add more changes than before. In this perspective, researchers find that eroding architecture continuously reduces the maintainability and evolvability of the architecture (e.g., [S21], [S28]). For example, the authors of [S21] stated that "*when the architecture of a software system allows no more changes to it due to changes introduced in the system over time and renders it unmaintainable*".

We found that certain definitions contain multiple perspectives. For example, the authors of [S21] proposed a definition of AEr from three perspectives: violation, quality, and evolution. The definition of AEr provided by [S21] is "*when concrete (as-built) architecture of a software system deviates from its conceptual (as-planned) architecture [violation perspective] where it no longer satisfies the key quality attributes [quality perspective] that led to its construction OR when architecture of a software system allows no more changes to it due to changes introduced in the system over time and renders it un-maintainable [evolution perspective]*".

2.4.3 RQ2: What are the symptoms of architecture erosion in software development?

AEr shows a tendency to decrease the engineering quality of a system, during which AEr can be detected by miscellaneous signs. Those signs are explicitly mentioned in the literature that can be regarded as *symptoms* and *indicators* of AEr phenomenon, while the symptoms have not been systematically categorized. Therefore, to better understand various symptoms, we collected the AEr symptoms and classified them into four categories as shown in Table 2.9.

Structural symptom includes various structural problems in software systems. Code anomaly agglomeration (e.g., [S17]) and architectural smell (e.g., undesired dependencies [S20], cyclic dependencies [S38]) are the most frequently mentioned structural symptoms. For example, the authors of [S26] mentioned that "*several studies confirmed co-occurrences of code anomalies are effective indicators of architectural degradation symptoms*". Another example is that the authors of [S64] mentioned "*we use the (architectural) smells as indicators of architecture erosion and consequently to analyze better the sustainability of a system*". Moreover, modular problems (e.g., deteriorated modules and lack of modularity [S36]) can also be regarded as signs of AEr phenomenon. For instance, the authors of [S43] stated that "*we ignore this case and consider lack of modularity as a symptom of architecture erosion*".

Violation symptom is a common type of AEr symptoms, which derive from the most common definition of AEr (see Section 2.4.2) and refers to the violations of prescribed design decisions or constraints (e.g., abstraction and encapsulation). The violations of design decisions or constraints are regarded as a symptom of AEr, which is mentioned in ten selected studies (e.g., [S31], [S57]). In [S57], the authors proposed two types of architectural violations: divergence and absence relationship (see Table 2.9).

Quality symptom is comparatively obvious symptom and often manifests in quality attributes of products that can receive attentions of maintainers, such as high defect rate (e.g., [S21]), losing functionality (e.g., [S22]), and gradually decreasing productivity (e.g., [S7], [S53]). For example, the authors of [S21] mentioned that "*some common symptoms of decay in software architecture include poor code quality, unlocalized changes and regressions, high defect rates*".

Evolution symptom covers the symptoms related to the evolution process of software systems. For example, rigidity refers to the tendency that architecture becomes difficult to change, since a simple change might cause a ripple effect across all the coupling classes or components (Martin, 2000) (e.g., [S7]). Another example is that, the authors of [S1] mentioned that "*increasing (change) difficulty in further evolution and maintenance is a sign of code decay*".

Table 2.9: Categories of symptoms of architecture erosion

Type	Subtype	Description	Studies
Structural symptom	Code anomaly agglomeration and architectural smell	Certain structural issues (e.g., code anomaly agglomeration, architectural smells) can be regarded as a sign of AEr	[S5][S7][S17][S18][S20] [S21][S27][S31][S36] [S38][S41][S47][S57] [S64]
	Co-occurrence of code smells	Certain patterns of co-occurring code smells tend to be stronger indicators of AEr	[S1][S26][S27][S36]
	Modularity problem	It refers to the modular issues, such as deteriorated modularization and lack of modularity	[S5][S27][S43]
	Class and modular grime	Grime refers to the increase in the number of harmful relationships (e.g., generalizations, associations, dependencies)	[S9][S41]
	Others	The study has a generic description about structural symptoms not included in the other subtypes	[S1][S21][S22][S26] [S40]
Violation symptom	Divergence	A module or relationship that is in the extracted architecture but not in the intended architecture	[S1][S57]
	Absence relationship	A module or relationship that is in the intended architecture but not the extracted architecture	[S57]
	Others	The study has a generic description about violation symptoms not included in the other subtypes	[S1][S12][S18][S21] [S27][S31][S36][S38] [S57][S69]
Quality symptom	A gradually lower productivity	Decreasing productivity or investing costs cannot receive higher output	[S7][S53]
	Functional erosion	Decreasing functionality in software systems	[S22]

(Continued) Table 2.9

Type	Subtype	Description	Studies
	High defect rate	The high portion of defective elements compared to all items produced	[S21]
Evolution symptom	Rigidity	The tendency for software to be difficult to change, as a change could cause a cascade of subsequent changes in dependent modules	[S1][S7][S21]
	Brittleness	The tendency of the software to break in many places every time it is changed, since the changes might cause break in unexpected ways	[S7]
	Immobility	The inability to reuse software from other projects or from parts of the same project	[S7]
	Increasing correction cost	Gradually increasing cost for correcting architectural defects	[S1]

2.4.4 RQ3: What are the reasons that cause architecture erosion in software development?

In fact, AEr can be caused by various reasons. In Table 2.10, we list the 13 reasons according to the data collected from the selected studies, as well as their subtypes, description, and related studies. The results show that 64.4% (47 out of 73) of the studies explicitly mention the reasons that cause AEr and we classified these reasons into 13 categories (see Table 2.10).

Architecture violation. An architecture gradually erodes when architecture violations are introduced into the system by several ways: (1) Rules governing the dependencies between different levels of the system architecture are violated (e.g., [S31]), such as communication rules, conformance rules. (2) Design decisions and architectural guidelines are not strictly followed (e.g., [S34], [S44], [S46]), such as presentation layer frameworks.

Evolution issue. Architecture could hardly keep intact when evolution happens. As the software maintenance and evolution, certain inappropriate changes

could break the original architectural integrity and introduce superfluous dependencies. Some seemingly innocuous architecture changes (e.g., bug-fixings, refactoring) could also cause ripple effects to the system architecture (e.g., [S36]). Uncontrollable evolution process (e.g., [S58]) might increase the risk of AEr and give rise to software failure. For example, AEr could be also caused by technological evolution (e.g., [S7]), including operating systems, hardware, and programming languages (Fellah and Bandi, 2019).

Technical debt is a metaphor reflecting technical compromises that can yield short-term benefits but may hurt the long-term goal of a software system (Li et al., 2015). For example, new products are hastily released to the market because of time pressure (e.g., [S66]). Some shortcuts can change architecturally relevant elements (such as classes, components, modules) and break architectural integrity (such as breaking the encapsulation rules, introducing undesired dependencies), or intentionally incur technical debt (e.g., [S55]). Such suboptimal operations or techniques might increase potential risks to the system quality (such as reliability, extensibility), thereby leading to AEr. More details of technical debt as a reason for AEr are discussed in Section 2.5.1.3.

Knowledge vaporization could cause a series of problems, for example, a poor understanding of project contexts can hinder knowledge transfer among team members, which is often regarded as one of the reasons that cause AEr. Lack of documentation denotes that the previous knowledge (such as design decisions and their rationales) is not available or documented (such as undocumented design decisions and assumptions), which hinders the subsequent architecture enhancements and modifications (e.g., [S57], [S66]). Besides, developer turnover (Ma et al., 2020) aggravates AEr by losing knowledge about system requirements and architectural decisions (e.g., [S50]).

Requirement issue challenges the architecture sustainability. For example, newly-added or constantly changing requirements may conflict with the intended architectural design decisions (e.g., [S4], [S37], [S45]). Conflicting requirements that are possibly unforeseen in the early stage, and vague requirements may break the consistency and integrity of the intended architecture and negatively influence the maintainability and extensibility of the system architecture (e.g., [S48]).

Understanding issue. Poor understanding of the system architecture stems from various factors, such as poor code readability (e.g., bad code quality, lack of code comments), which may act as a trigger for making wrong changes during maintenance and modification. For example, the authors of [S45] pointed out that AEr can derive from bad practices and developer mistakes when developers change the code without understanding the intended architecture which may be violated. Another example is from [S34] that inadequate understanding of the design documentation

and principles (e.g., due to poor training) may be a trigger of AEr. Moreover, lack of architecture knowledge can also lead to the understanding problem, for example, developers who are not aware of previous design decisions or have poor knowledge about the intended architecture [S45] could cause damages to the original decisions and architectural structure [S39].

Suboptimal design decision is a kind of design defect that often occurs during the design process. Design defects might creep into a system architecture during the design phase, which can be regarded as a hidden danger to system sustainability, as design defects might not be discovered through static analysis (Li, Liang, Soliman and Avgeriou, 2021b). If the target architecture style is initially a suboptimal choice, then trying to fix the design defects later may be impossible (Barney et al., 2012). For example, the authors of [S47] mentioned that many maintenance issues originate from inappropriate design. Another example is that the original architecture is not designed to accommodate potential changes (e.g., low extensibility [S34], [S73]).

Increasing complexity usually happens when there are increasing couplings and oversized architecture elements (such as class, component, module) during evolution. Increasing complexity can reduce the understandability of systems and makes it easier to damage the architectural integrity when changes happen (e.g., [S34], [S56]), since the clean initial architecture cannot be recognized anymore (e.g., [S66]). For example, the authors of [S54] mentioned that the increase in complexity combined with an evident lack of documentation hinders stakeholder to maintain the design aspects of a system, and consequently lead to AEr.

Organization issue. According to Conway's law (Conway, 1968), a bad organization can probably generate bad architecture. Various management problems (e.g., lack of maintenance [S47], [S53]) can generate a bad organization, which finally mirrors negative impact in architecture, including high turnover rate, poor training and education of developers, unfair rewarding and punishment metrics, and incompetent code view process. For example, the authors of [S34] mentioned that poor developer training could be a key trigger of AEr. Consequently, AEr can happen due to various organization and management issues.

Table 2.10: Reasons that cause architecture erosion

Reason	Subtype	Description	Studies
Architecture violation	Architectural rule and constraint	Architecture violation happens by violating architectural guidelines (e.g., dependencies between components)	[S34][S46][S54] [S58]

(Continued) Table 2.10

Reason	Subtype	Description	Studies
	Design decision	Developers mistakenly violate design decisions	[S35][S42][S45]
	Others	The study has a generic description about architecture violation not included in the other subtypes	[S13][S21][S23] [S27][S28][S31] [S32][S50][S62] [S66][S73]
Evolution issue	Inappropriate architecture change	The changes to a system often erode the fundamental characteristics of the original architecture	[S1][S2][S3][S9] [S10][S16][S21] [S26][S31][S37] [S45][S50][S73]
	Uncontrolled evolution process	Uncontrolled evolution process might increase the complexity of system and decrease the maintenance	[S56][S58]
	Process change	Architecture changes due to the business and development process change	[S7][S22]
	Others	The study has a generic description about evolution issue not included in the other subtypes	[S7][S45]
Technical debt	Deadline pressure	Due to deadline pressures, architecture changes or compromises have been taken to system architecture	[S9][S12][S22] [S46][S47][S48] [S53][S62][S66] [S69]
	Others	The study has a generic description about technical debt not included in the other subtypes	[S4][S55][S73]
Knowledge vaporization	Lack of documentation	Incomplete and unavailable documentation and undocumented knowledge	[S34][S36][S37] [S45][S47][S50] [S53][S54][S57] [S66]
	Developer turnover	Knowledge lost due to developer turnover	[S34][S69]
Requirement issue	Unforeseen and conflicting requirement	Unforeseen, conflicting, and vague requirements may bring uncertainties to software development and cause conflicting design decisions	[S4][S12][S34] [S37][S48][S54] [S60][S62][S66] [S69]

(Continued) Table 2.10

Reason	Subtype	Description	Studies
	Constantly changing requirement	Constantly changing requirements could lead to multiple impacts (e.g., dramatically increase the project budget, development schedule)	[S45]
Understanding issue	Poor understanding of intended architecture	Lack of a full understanding of a system may let the system conflict with the original design during evolution	[S25][S34][S37] [S45]
	Lack of architecture knowledge	Developers (e.g., novice) who lacks enough architecture knowledge to support the development and evolution	[S39][S44][S45] [S53]
Suboptimal design decision	Inflexible design	The architecture has low extensibility and is hard to accommodate changes	[S34][S73]
	Others	The study has a generic description about design flaw not included in the other subtypes	[S7][S17][S26] [S36][S47]
Increasing complexity	Design pattern grime	Increasing unrelated artifacts to original design patterns	[S9][S38]
	Others	The study has a generic description about increasing complexity not included in the other subtypes	[S34][S51][S54] [S56][S64][S66]
Organization issue	Poor developer training	Lack of training and education makes it harder to establish uniform programming styles	[S34][S47]
	Lack of maintenance	Lack of long-term maintenance (e.g., commitment) to the projects	[S47][S53]
	Poor code review process	Bad quality of code review cannot ensure the code quality	[S45]
	Increasing workload	Increasing workload might have a negative impact on the development productivity and software quality	[S47]
	Non-architectural-centered practice	Focusing on coding efforts instead of the architecture will doom to suffer architecture erosion	[S61]

(Continued) Table 2.10

Reason	Subtype	Description	Studies
Communication issue	Lack of communication	Lack of communication between stakeholders	[S12][S25][S34] [S47]
	Miscommunication	Miscommunication includes inaccurate information and misunderstanding	[S12][S34]
Environment change	Operational environment change	Application environment changed (e.g., standards changes)	[S9][S73]
	Obsolete software and hardware component	Third party libraries update and become incompatible with the current architecture	[S62]
Lack of architecture tools	-	Manually fixing problems probably contributes to AEr due to lack of architecture and design tools	[S23][S34][S40]
Iterative software development process	-	Modern iterative software development processes (e.g., agile programming methods) may give rise to the occurrence of AEr sooner	[S34]

Communication issue. Both miscommunication and lack of communication can incur AEr during the development process. Miscommunication refers to conveying wrong information between stakeholders, while lack of communication can be unconscious. For example, the authors of [S34] stated that communication is vital during system evolution, which is a common cause of AEr including unavailable or misunderstood design decisions.

Environment change. The retention of outdated technologies and software (or hardware) components can reduce architecture evolvability, where the software does not follow the changes and AEr will be incurred very soon (e.g., [S73]). Therefore, when the environment changes, the system architecture should also be revised to adapt to the new operational environment, otherwise, environment changes can contribute to the deterioration and erosion of system designs (e.g., [S9]).

Lack of architecture tools is a potential cause of AEr for software systems. For example, the authors of [S24] mentioned that AEr might even more severe in systems implemented in dynamic languages, since certain features of dynamic languages (e.g., dynamic invocations and buildings) make developers more likely to

Table 2.11: Reasons of architecture erosion in three categories

Category	Reason	Studies
Technical reason	Architecture violation	[S13][S21][S23][S27][S28][S31][S32][S34][S35] [S42][S45][S46][S50][S54][S58][S62][S66][S73]
	Evolution issue	[S1][S2][S3][S7][S9][S10][S16][S21][S22][S26] [S31][S37][S45][S50][S56][S58][S73]
	Technical debt	[S4][S9][S12][S22][S46][S47][S48][S53][S55][S62] [S66][S69][S73]
	Suboptimal design decision	[S7][S17][S26][S34][S36][S47][S73]
	Increasing complexity	[S9][S34][S38][S51][S54][S56][S64][S66]
	Lack of architecture tools	[S23][S34][S40]
Non-technical reason	Knowledge vaporization	[S34][S36][S37][S45][S47][S50][S53][S54][S57] [S66][S69]
	Understanding issue	[S25][S34][S39][S37][S44][S45][S53]
	Organization issue	[S34][S45][S47][S53][S61]
	Communication issue	[S12][S25][S34][S47]
	Environment change	[S9][S62][S73]
Both	Requirement issue	[S4][S12][S34][S37][S45][S48][S54][S60][S62] [S66][S69]
	Iterative software development process	[S34]

break the planned architecture and these languages suffer from lack of architecture tools.

Iterative software development process (e.g., agile development methods, extreme programming) can give rise to the occurrence of AEr sooner, since they place less emphasis on upfront architectural design (e.g., [S34]). Recent studies show that the agile process may not make a project agile (Sturtevant, 2017), and architecture might become the bottleneck of agile projects and rapidly erode if the developers focus on following the agile process rather than increasing architectural agility.

Table 2.11 shows that the 13 reasons that cause AEr can be further classified into three categories: (1) technical reason, (2) non-technical reason, and (3) both. Technical reason refers to the reasons related to technical application, and non-technical reason denotes the reasons closely related to the human factors in software engineering. Both means that the type consists of both technical and non-technical reasons. The results in Table 2.10 and Table 2.11 show that technical reasons of AEr (46.6%, 34 out of 73 studies) derive from design, implementation, maintenance, and evolution phases; non-technical reasons of AEr (24.7%, 18 out of 73 studies) stem from organization and management issues; 15.1% (11 out of 73) of the studies mentioned both

technical reasons and non-technical reasons. Additionally, the results also indicate that AEr exists in different phases of software development and interacts with many stakeholders.

2.4.5 RQ4: What are the consequences of architecture erosion in software development?

We identified various consequences of AEr mentioned in the selected studies (see Table 2.12). These consequences can be classified into eight categories.

Quality degradation is the most frequently mentioned consequence and AEr can result in various Quality Attributes (QAs) degradation. In this SMS, we mapped the degraded QAs according to the ISO/IEC 25010 standard (ISO/IEC25010, 2011) (see Table 2.12). For example, AEr can give rise to the deterioration of the architectural structure, make the system less flexible (e.g., hard to enhance and extend [S7]), decrease modularity of systems (e.g., reduce the components' cohesion [S50]), and make systems harder to maintain without breaking certain dependencies. For instance, the authors of [S34] mentioned that "*even if a system does not become unusable, erosion makes software more susceptible to defects, incurs high maintenance costs, degrades performance and, of course, leads to more erosion*". Besides, stakeholders may not even be able to understand the ramification of breaking these dependencies.

Architectural defect denotes that an eroded architecture will make the architecture to have more defects (i.e., defect proneness), e.g., anti-patterns [S9], superfluous dependencies [S47], [S73], which in turn is likely to give rise to more erosion. For example, the authors of [S54] mentioned that "*even if a software system does not become completely inoperative, erosion will make the system more predisposed to defects*". Architecture mismatch (Garlan et al., 2009) is also an architecturally relevant defect arose in eroded software systems (e.g., [S47]). The potential problems of architectural mismatch may derive from the assumptions made to the components, connectors, and global architecture structure (Garlan et al., 2009).

Increased cost denotes that rising costs (including time and labor cost) need to be invested into activities like maintenance and refactoring. Moreover, it may increase the time-to-market of software products and bring about the budget of development cost overrun (e.g., [S51], [S69]).

Failure of software projects might be the worst consequence of AEr. Once uncontrollable AEr has led to irreversible effects and irreparable software systems (or become less cost-effective to be maintained), it marks the failure of software projects and the systems need to go through a process of reengineering and redevelopment (e.g., [S34], [S44]). Besides, AEr shorten the lifetime of software projects and cause rapid obsolescence, or trigger bottom-up system reengineering (e.g., [S1]).

Table 2.12: Consequences of architecture erosion

Category	Type	Subtype	Studies
Quality degradation	Maintainability	Modifiability	[S1][S7][S9][S17][S21][S34][S49][S71][S73]
		Modularity	[S9][S12][S43][S46][S49][S51][S52]
		Reusability	[S13][S24][S33][S47][S49][S53]
		Testability	[S9][S30][S47][S51][S52][S73]
		-	[S9][S13][S16][S21][S24][S25][S29][S33][S42][S46][S47][S54][S62][S63][S69]
	Evolvability	-	[S12][S24][S29][S46][S47][S53][S63][S69][S71]
	Conceptual Integrity of the Architecture	Understandability	[S21][S42][S47][S51][S54][S73]
	Performance efficiency	Capacity	[S22]
		-	[S9][S29][S30][S34][S54][S62]
	Portability	Adaptability	[S9][S13][S15][S30][S33][S69]
	Extensibility	-	[S51][S52][S54]
	Reliability	-	[S29][S49]
	Security	-	[S30]
	Usability	Operability	[S29]
		-	[S54]
	Others	-	[S7][S9][S15][S34][S40][S53]
Architectural defect	Brittleness	-	[S11][S32][S48][S73]
	Architecture mismatch	-	[S16][S40][S47]
	Anti-pattern	-	[S9]
	Poor system integrity	-	[S21]
	Others	-	[S1][S9][S15][S21][S34][S54][S71][S73]
Increased cost	-	-	[S34][S38][S48][S49][S51][S52][S53][S54][S69][S73]
Failure of software projects	Shortened system lifetime	-	[S1]

(Continued) Table 2.12

Category	Type	Subtype	Studies
	Others	-	[S7][S13][S33][S34][S44][S54] [S69][S73]
Failure to meet requirements	-	-	[S7][S48][S73]
Software aging	-	-	[S9][S34][S48]
Technical debt	Architectural debt	-	[S49]
	Others	-	[S42][S49]
Organization disintegration	-	-	[S7]

Failure to meet requirements means that AEr can result in the decreasing ability of a system to satisfy the requirements of stakeholders. For example, the authors of [S48] mentioned that the potential consequences of AEr encompass failure to meet the requirements, as changes become difficult to be made on software systems due to various reasons, such as increasing complexity in eroded architecture (e.g., [S7]).

Software aging is a phenomenon caused by AEr that refers to the tendency of performance degradation and failure of software systems, and a general characteristic of this phenomenon is that the systems failure rate will increase and many aging-related errors occur (e.g., erroneous outcomes) (Parnas, 1994; Grottke et al., 2008). For example, the authors of [S9], [S34], [S48] mentioned that the low adaptability and extensibility arose in an eroded architecture could further deteriorate the systems, and AEr could accelerate software aging.

Technical debt would be incurred in software systems that AEr exists and degrades the quality of software systems, where short-term compromises lead to significantly potential hidden danger to software systems regarding fixing bugs or adding new features (e.g., [S49]). For example, the authors of [S42] mentioned that AEr has been shown to incur technical debt and decrease a system's maintainability, therefore, tracking AEr is critical. Besides, the authors of [S49] stated that "*such an (eroded) system incurs a technical debt, where short-term compromises lead to significant long-term problems in terms of reduced ability of fixing bugs or adding new features*".

Organization disintegration denotes that AEr could cause inconsistent management issues, thereby disintegrating the organization. For example, the authors of [S7] argued that the newly hired developers in a development team cannot completely understand the eroded architecture and the workload finally falls on the

original developers who might have to work hard and not resist the stress, which may incur high turnover in the long run.

2.4.6 RQ5: What approaches and tools have been used to detect architecture erosion in software development?

2.4.6.1 Approaches

As shown in Table 2.13, we classified the collected approaches of AEr detection into four categories, which are detailed below.

Consistency-based approach refers to approaches based on the evaluation of architecture consistency (Ali et al., 2018), which aims to align the implemented system with the intended architecture. Consistency-based approaches are one type of the most commonly applied approaches for detecting AEr during the development process. 34.2% (25 out of 73) of the studies propose consistency-based approaches to detect AEr through evaluating architecture consistency, in which around 56.0% (14 out of 25) of the studies use the Architecture Conformance Checking (ACC) techniques to detect AEr through checking the conformance between the implemented architecture and the intended architecture (e.g., [S28], [S38], [S43]). For example, specific architectural rules can be defined and used to compare the implemented architecture against the specified constraints, such as rules of constraints (e.g., [S33], [S45], [S53]), formal representation (e.g., [S13], [S68]), Architecture Description Languages (ADL) (e.g., [S29], [S48]), and Domain-Specific Language (DSL) (e.g., [S14], [S31], [S58]). In addition, other approaches can also be used to evaluate architecture consistency. Reflexion Modelling (RM) technique (Murphy et al., 2001) enables developers to extract high-level models and map the modules to source code elements for building high-level models that capture the intended architecture of the system, thereby identifying AEr. Design Structure Matrix (DSM) can establish a dependency matrix among classes of a system without relying on the higher level components, which enables developers to visually check violations for detecting AEr (e.g., [S49], [S51]).

Evolution-based approach is commonly used to identify AEr by checking different history versions of projects. For example, the authors of [S15] proposed the ADvISE approach to analyze the evolution of the architecture at different levels. The authors proposed a representation approach of architecture based on classes and their relationships, and they analyzed the architectural history at various levels (e.g., classes, triplets, micro-architectures) to identify and measure AEr. Besides, to study architectural changes and erosion, the authors of [S16] conducted an empirical study that they employed the ARCADE approach to analyze the evolution history

of the architecture of 14 open-sourced Apache projects using architectural recovery, architectural change metrics, dependency analysis, and evolution paths.

Defect-based detection is extensively employed to detect AEr, which refers to the review and inspection process of detecting AEr, e.g., through identifying an increasing number of system defects. For example, the authors of [S1] conducted a case study of defect-based measurement and they found that hotspot components contain 50% more defects which contribute most to AEr. Other approaches can also help to detect AEr by identifying the symptoms of AEr (see Section 2.4.3). For example, the authors of [S64] proposed the symptoms of AEr (e.g., interface-based smells, dependency-based smells) and AEr metrics to analyze the AEr trend in software systems.

Decision-based approach aims to detect AEr by capturing important design decisions, including the selected architectural patterns and tactics. For example, the authors of [S37] detected AEr by redefining a collection of architectural patterns. They leveraged the proposed ABC tool to capture design decisions and generate OCL code to specify the constraints of the patterns, in order to detect AEr during the design phase and validate the runtime architecture.

Table 2.13: Approaches used to detect architecture erosion

Category	Approach	Description	Studies
Consistency-based approach	Architecture Conformance Checking (ACC)	Checking whether the implemented architecture is consistent with the intended architecture (e.g., checking architectural violations)	[S13][S14][S28] [S29][S31][S33] [S38][S43][S45] [S48][S53][S58] [S68][S73]
	Reflexion Modelling (RM)	It helps to extract high-level models, map the implementation entities into these models, and compare the two artifacts	[S27][S40][S55] [S57][S65][S69]
	Design Structure Matrix (DSM)	Detecting relations of modules by a creating dependency matrix to visually check for violations	[S49][S51]
	Light-weight sanity Check for Implemented Architectures (LiSCIA)	LiSCIA can help to derive architectural constraints from the implemented architecture and spot potential problems leading to AEr	[S8][S28]

(Continued) Table 2.13

Category	Approach	Description	Studies
	Detection of design pattern grime and rot	Comparing the specified design patterns with realized patterns	[S9]
	Detection of dependencies between architectural components	Analyzing the number and types of dependencies between architectural components	[S11]
Evolution-based approach	Architectural Decay In Software Evolution (ADvISE)	ADvISE aims to analyze architecture evolution at various levels that helps to analyze how and where AEr occurred	[S15][S56][S61]
	Architecture Recovery, Change, And Decay Evaluator (ARCADE)	A software workbench used to conduct architecture recovery, architectural changes metrics, and AEr metrics	[S16][S17]
	MORPHOSIS	Detecting AEr through analyzing evolution scenarios, checking architecture enforcement, and tracking architecture-level code metrics	[S51][S52]
	Profilo	Detecting AEr by analyzing the history and trends of system evolution	[S56]
	Architecture Analysis and Monitoring Infrastructure (ARAMIS)	Detecting AEr by runtime monitoring, interaction validation among architectural components, mapping the captured behavior of architectural elements	[S67]
	Variant Analysis (VA)	Detecting and comparing architectural violations across different versions	[S44]
Defect-based detection	Defect-based measurement	Investigating the defect-fix history and detecting hotspots that contribute to AEr by analyzing certain defects (e.g., interface defects)	[S1][S73]
	Detection of architectural smells	Detecting AEr by detecting architectural smells	[S36][S64]

(Continued) Table 2.13

Category	Approach	Description	Studies
	Software Prognostics and Health Management (PHM) approach	Building a PHM model to identify and predict the health status of software systems based on Discriminant Coordinates Analysis (DCA)	[S3]
	Detection of design pattern grime and rot	Comparing the specified design patterns with realized patterns to identify AEr	[S9]
Decision-based approach	Capturing design decisions about the adopted architectural patterns	Detecting AEr by capturing the most important design decisions about the adopted architectural patterns	[S37]
	Detect and trace architectural tactics	Detecting AEr through identifying and tracing the presence of architectural tactics and keeping developers informed of underlying architecture decisions	[S39]

2.4.6.2 Tools

Versatile tools can facilitate the understanding of the system architecture, and make it more convenient for developers to visualize, identify, track the symptoms of AEr, and finally repair or mitigate AEr. There are 41.1% (30 out of 73) of selected studies that proposed or investigated relevant tools for detecting AEr. In total, 35 tools were collected from the selected studies, and the numbers of open-source tools (51.4%, 18 out of 35) and commercial tools (48.6%, 17 out of 35) are close to each other. Six tools are not available for download now.

The tools have various features and the classification of tools is presented in a similar manner developed for AEr detection approaches according to their main purposes mentioned in the studies. In Table 2.14, we classified the 35 tools into four categories based on their distinctive purposes. The results in the "URL" column were mainly collected from the selected studies, part of which were collected from the Internet. "Not available" means that we cannot find an available URL link regarding whether the tools are still available now.

Conformance checking denotes that this type of tools is used to check archi-

tecture conformance for ensuring the developers and maintainers have followed the architectural edicts set and not eroding the architecture by breaking down abstractions, bridging layers, compromising information hiding, and so on (Bass et al., 2021). Tools in this type support detection of architectural violations, analysis of dependencies, and so forth. For example, Axivion allows developers and architects to conduct dependency analysis, quality metrics, and visualization for checking architecture conformance by identifying and stopping AEr (e.g., [S18], [S34]).

Quality measurement means that tools of this type provide various metrics to evaluate the quality of systems, monitor software systems, trace decisions, etc. For example, Archie is an Eclipse plugin that helps developers know about the impact of modifications and refactorings on sensitive areas of code, thereby contributing to mitigating the long-term problems related to AEr in the system.

History analysis denotes that this type of tools can be used to analyze the evolution trend, trace evolutionary scenarios, and parse the software architecture. For example, Sotograph consists of a suite of tools and supports the analysis of detailed structure, quality, and dependency on different abstraction levels, such as cyclical dependencies and duplicated code blocks (e.g., [S63], [S65], [S66]).

Visualization refers to the tools that employ graphical representation of architectural elements to detect AEr, such as the visualization of anti-patterns and architectural smells (e.g., cyclic dependencies). Visualization techniques contribute to the understanding of the structure of large software systems by graphic or multiple-dimensional representations. For instance, the authors of [S72] demonstrated that Getaviz is an easy-to-use tool for visualizing software evolution, which enables developers to visualize the erosion of system design and architecture, such as the evolution of cyclic dependencies and anti-patterns.

Table 2.14: Tools used to detect architecture erosion

Type	Tool	URL link	Description	Open source	Studies
Conformance checking	Lattix	https://www.lattix.com/products-architecture-issues/	It can be used to extract the architecture dependencies for identifying architectural violations.	No	[S18][S23][S25] [S28][S34][S51] [S52][S63][S65]
	Structure101	https://structure101.com/	It offers visualized views of code organization and helps practitioners to better understand the structure and remove cyclic dependencies.	No	[S13][S18][S23] [S25][S34][S63] [S65][S66]
	Axivion	https://www.axivion.com/en	It supports various dependency analysis and helps stop software erosion, analyze, and recover the methods developed for legacy software by understanding the software architecture.	No	[S18][S34][S51] [S52][S63][S65] [S66]
	Sonargraph	https://www.hello2morrow.com/products/sonargraph	It is a static code analyzer that allows developers to monitor a software system for technical quality, enforce architectural rules (e.g., relationships between layers), and detect violated dependencies.	No	[S18][S23][S25] [S34][S63]
	ArCh	https://wiki.archlinux.org/index.php/Java_package_guidelines	It can be used to check dependencies violations and generate reports about the conformance checking of system architecture.	Yes	[S13][S60][S69]
	ConQAT	https://www.cqse.eu/en/news/blog/conqat-end-of-life/	It can depict all modules as UML components and check the consistency of the defined architectural model.	No	[S23][S63][S68]
	SAVE	Not available	It provides a graphical editor to define the intended architecture and show violations after the evaluation.	No	[S23][S44][S65]
	ARCADE	https://softarch.usc.edu/wiki/doku.php?id=arcade:start	It supports architecture recovery, architectural change and decay metrics, and helps perform different statistical analyses.	Yes	[S11][S16]

(Continued) Table 2.14

Type	Tool	URL link	Description	Open source	Studies
	Dclcheck	https://github.com/rterrabh/pi-dclcheck	It can be used to verify whether the implemented architecture respects the specific constraints.	Yes	[S4][S69]
	Macker	https://sourceforge.net/projects/macker/	It can be used to define a diversity of conformance rules and provide violation reports.	Yes	[S23][S63]
	ArchRuby	https://aserg.labsoft.dcc.ufmg.br/archruby/manual.html	It can generate reports about the illustration of the architectural conformance and visualization processes.	No	[S24]
	DCL2Check	https://github.com/aserg-ufmg/dcl2check	As a plugin for Eclipse IDE, it supports architectural conformance verification and high-level architectural visualization.	Yes	[S35]
	Card	Not available	As a plugin for Eclipse IDE, it can be used to search for UML and Java files, as well as conformance checking.	Yes	[S48]
	ReflexML	Not available	It can be used to create a UML-embedded mapping of architecture models to code and check the consistency (e.g., architecture-to-code traceability information) based on the mapping results.	Yes	[S63]
	dTangler	https://github.com/vladdu/dtangler	It helps conduct conformance checking based on the DSM technique that is complemented with text-based editors to define rules.	Yes	[S23]
	Coverity	https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html	It provides a set of quality metrics and various forms of dependency analysis to check architecture conformance.	No	[S34]

(Continued) Table 2.14

Type	Tool	URL link	Description	Open source	Studies
	STAN	http://stan4j.com/	It helps to detect design flaws, analyze dependencies, and visually understand system structure.	No	[S18]
	Dependometer	https://sourceforge.net/projects/dependometer/	It helps validate dependencies against the logical architecture structuring the system into classes, packages, subsystems, vertical slices, and layers and detects cycles between these structural elements.	Yes	[S63]
	Classycle	https://classycle.sourceforge.net/	It allows to define and analyze the dependencies of classes and packages (especially helpful for finding cyclic dependencies).	Yes	[S63]
	FIS system	https://www.fispro.org/en/	It is a fuzzy rule-based system (a.k.a Fuzzy Inference System) used to detect AEr symptoms and recommend treatments to reduce or mitigate AEr symptoms.	Yes	[S41]
	Klocwork	https://www.perforce.com/products/klocwork	It is a static analysis tool that provides support to architecture visualization in the form of graphs, which helps detect security issues and increase code reliability for ensuring architecture conformance.	No	[S34][S65]
	ABC	Not available	It provides support to design architecture, adopting predefined architectural patterns, capturing design decisions, and detecting AEr.	Yes	[S37]
	SIC	Not available	It helps to extract the descriptive architecture using JavaDoc comments and detect the architecture violations of a codebase change.	Yes	[S45]

(Continued) Table 2.14

Type	Tool	URL link	Description	Open source	Studies
Quality measurement	JDepend	https://github.com/clarkware/jdepend	It allows to automatically measure the quality of a design in terms of its extensibility, reusability, and maintainability to effectively manage and control package dependencies.	Yes	[S18][S34][S65]
	Archie	https://github.com/ArchieProject/Archie-Smart-IDE	As a plugin for Eclipse IDE, it helps to automate the creation and maintenance of architecturally-relevant trace links between code, architectural decisions, and related requirements.	Yes	[S19][S39]
	SourceAudit	Not available	It can measure the maintainability of source code and automatically monitor the source code quality of software system.	No	[S59]
	JArchitect	https://www.jarchitect.com/	It can provide various metrics to analyze the architecture and generate evaluation reports.	No	[S53]
	SonarQube	https://www.sonarqube.org/	It supports inspecting code quality by performing automatic reviews.	No	[S18]
	Xradar	https://xradar.sourceforge.net/	It is an open extensible code report tool and can generate HTML/SVG reports of the systems' current state and development over time.	No	[S63]
History analysis	Sotograph	http://www.hello2morrow.com/products/sotograph	It consists of a number of tools and supports the trend monitoring, quality and dependency analysis on different abstraction levels, as well as the changes in architecture violations between various versions.	No	[S63][S65][S66]
	CppDepend	https://www.cppdepend.com/	(For C++) It is used to measure some basic metrics and trace evolution scenarios to the code and reveal potential ripple effects.	No	[S51][S52]

(Continued) Table 2.14

Type	Tool	URL link	Description	Open source	Studies
	NDepend	https://www.ndepend.com/	(For C#) It is used to measure some basic metrics and trace evolution scenarios to the code and reveal potential ripple effects.	No	[S51][S52]
	EVA	https://github.com/namdy0429/EVA	EVA helps explore, visualize, and understand multiple facets of architectural evolution.	Yes	[S42]
Visualization	Understand	https://scitools.com/	It helps to understand and maintain the poorly documented legacy systems by visualizing and metrics.	No	[S18]
	Getaviz	https://github.com/softvis-research/Getaviz	It is a toolkit that automatically generates the visualization of the eroded part of systems, thereby assisting developers and architects in evaluating, tracing, and combating AEr.	Yes	[S72]

2.4.7 RQ6: What measures have been used to address and prevent architecture erosion in software development?

We find that the measures used to address and prevent AEr encompass preventive and remedial measures. As shown in Table 2.15, we classified the measures used to address AEr into three categories: (1) preventive measure, (2) remedial measure, and (3) both. Preventive measures are usually employed in different architecture activities (e.g., architecture analysis, architecture evaluation), and these measures refer to the proactive countermeasures for avoiding AEr or mitigating the risk of AEr. Remedial measures aim at repairing (or improving) the existing eroded parts of the implementation. Both means the measures help to address AEr including the above-mentioned two measures.

2.4.7.1 Preventive measures

Architecture Conformance Checking (ACC) are employed to review the process-based activities during software development for ensuring the implemented architecture aligns with the intended architecture. The authors of [S23] mentioned that "*checking architecture conformance bridges the gap between high-level models of architectural design and implemented program code, and to prevent decreased maintainability, caused by architectural erosion*", which indicates that ACC can help identify and correct architectural violations and further avoid constant AEr during the software development life cycle. In addition, establishing architecture constraints can be used to support ACC. Architecture constraints refer to the rules for the specification and enforcement of architecture, which help to keep the architecture consistent with the specified restrictions, such as constraints for consistency checking between architecture decisions and component models. Establishing architecture constraints contributes to the reliability and robustness of software systems and reduces AEr risks. For example, the authors of [S4] proposed a Dependency Constraint Language (DCL) to restrict the spectrum of architectural dependencies. Additionally, the authors of [S35] proposed a domain-specific language (i.e., DCL 2.0) with a supporting tool (i.e., DCL2Check) to facilitate modular and hierarchical architectural specifications, which helps the development teams to handle architectural violations for addressing AEr.

Architecture monitoring and evaluation aims at employing means to monitor the health status of software systems and evaluate whether the symptoms of AEr (Garcia et al., 2009), such as architectural smells (e.g., extraneous connector, ambiguous interface, component overload) crept into a system. For example, the authors of [S8] applied LiSCIA, an architecture evaluation method, at different stages

of software development and periodically evaluated the implemented architecture, which can produce evaluation reports and guidelines to improve the implemented architecture, thereby helping to identify and prevent AEr. Besides, dependency analysis is a feasible measure for identifying and determining the interdependence between various entities (e.g., classes, packages, modules, components), which enables developers to understand the relationships and directions of the dependencies. For example, the authors of [S20] proposed a constraints concerning plugin based on a domain-specific language (i.e., DepCoL) to define constraints regarding plugins and feature dependencies. In this way, AEr can be prevented in plugin-based software systems.

Establishing traceable mechanism denotes that available mechanisms should be established to manage and trace software artifacts and support maintenance activities during the software development and evolution process, such as design decisions and tactics. For example, in order to mitigate and avoid pervasive problems of AEr, the authors of [S10] established six trace creation patterns for creating traceability between concrete architectural elements and design decisions, which can keep developers informed about implemented architectural tactics, styles, and design patterns.

Evolving architecture as changes occur refers to the measures with the goal of evolving an architecture when changes to the system happen, such as environment, requirements, and workflows changes. For instance, the authors of [S25] claimed that architecture should be built for satisfying all current requirements and incrementally evolved once requirements changed. Although it might be time-consuming, to some extent these measures are feasible to prevent and minimize AEr. Another example is from [S34], the authors mentioned that "*the contribution of self-adaptation strategies towards preventing architecture erosion relies on minimising human interference in routine maintenance activities*". Self-adaptation techniques are employed to build and manage the systems by following the principles of autonomic computing, in order to respond to possible changes (e.g., requirement and environment changes) and address uncertainty at runtime (Oreizy et al., 1999). With this measure, architectural problems might be detected and resolved early during the development process, which reduces the demands for making fixes afterwards that are potentially error-prone.

Explicitly specifying architecture could partially help to prevent AEr by defining architecture explicitly and exposing underlying design decisions, architecture tactics, and constraints, in order to ensure stakeholders and maintainers completely understand the details of what they will implement and reduce the risk from the implementation phase (e.g., [S19]). For example, the authors of [S29] argued that the AEr control techniques should provide mechanisms for explicitly defining the

intended architecture and checking whether the implemented system conforms to the intended design.

2.4.7.2 Remedial measures

Architecture maintenance is a common measure used to repair eroded architecture with the purpose of maintaining architectural sustainability. Architecture repair aims at using repairing strategies to fix up architectural anomalies, such as synchronizing the implementation with intended architecture and repairing detected violations. For example, the authors of [S18] employed tools (e.g., Sonargraph, Structure 101) to repair certain architecture problems (e.g., design violations, cyclic dependencies). Another example is that, the authors of [S12] proposed an architectural repair recommendation system that provides refactoring guidelines for developers and maintainers to fix architectural violations. Note that, inappropriate architectural modifications (e.g., violating architectural rules) could break the system structure over time, while suitable modifications can help to reverse or minimize AEr for the purpose of architecture optimization. For example, the authors of [S9] optimized the architecture by removing design pattern grime and rot, which can potentially reduce maintenance costs and improve system adaptability, consequently stop and remedy the eroding architecture.

Architecture restoration typically includes reverse engineering techniques to discover and recover architectural structures from software artifacts, as well as speculate the intended architectural design. For example, the authors of [S50] proposed an approach to regenerate architecture and extract architecturally-significant concerns that led to eroded model artifacts, in order to counteract the erosion process.

2.4.7.3 Both

Architecture refactoring aims to improve the structure of a system without changing the external behaviors of the system. For example, the authors of [S46] proposed a multi-objective optimization method for improving the original package structure, preserving the original design decisions, and remedying eroded parts in an architecture. Besides, simplifying architecture is also one of the purposes of architecture refactoring, which aims at deliberately controlling architectural complexity and further simplifying the system during the maintenance and evolution phases. When architectural complexity proliferates towards being uncontrollable, simplifying the system architecture can be useful for understanding the architecture of the system (e.g., [S18], [S69]), reducing the risk of AEr occurrence from the increasing complexity, and decreasing the difficulty of conformance checking.

Visualization is the way of using tools or feasible techniques to represent the architectural structure through visual notations for helping practitioners have better insights into the system design and relationships within/between various components and modules. In design phases, visualizing architecture can help to avoid architecture violations; in the maintenance phase, it can be conducive to repairing various code and architecture anomalies. Hence, this type of measures could be classified into both preventive and remedial measures.

Management optimization denotes that optimizing the management methods of controlling or stopping the constant erosion tendency of the system architecture. For example, the author of [S73] claimed that creating a culture is valued to stop erosion with management support, and this culture is likely to have the characteristics, including emphasizing regular refactoring, clear assignment of responsibilities, sharing architectural knowledge, and frequent communication within development teams.

Table 2.15: Measures used to address architecture erosion

Category	Type	Description	Studies
Preventive measure	Architecture conformance checking	Checking and correcting the inconsistencies between the intended architecture and the implemented architecture	[S2][S4][S6][S13] [S14][S20][S23][S24] [S25][S29][S31][S34] [S35][S40][S48][S52] [S58][S61][S63][S65]
	Architecture monitoring and evaluation	Monitoring and evaluating architectural status (e.g., assessing QAs metrics, dependency analysis)	[S8][S17][S20][S34] [S43][S51][S62][S67] [S70][S73]
	Establishing traceable mechanism	Tracing and managing artifacts (e.g., decisions, tactics) by certain feasible methods and tools (e.g., documentation, links)	[S10][S14][S34][S61]
	Evolving architecture as changes occur	Evolving architecture (and architecture model) when changes occurred (e.g., environment change, requirements change, workflows change)	[S25][S32][S34]
	Explicitly defining architecture	Explicitly defining the intended architecture and exposing related knowledge	[S19][S29][S31]

(Continued) Table 2.15

Category	Type	Description	Studies
Remedial measure	Architecture maintenance	Conducting maintenance activities to repair architectural defects and adapt the system architecture	[S7][S9][S12][S26] [S29][S34][S41][S53] [S57][S60][S73]
	Architecture restoration	Restoring architecture (e.g., reverse engineering, model discovery)	[S50][S53][S54][S71]
Both	Architecture refactoring	Refactoring the codebase (e.g., remodularizing, restructuring)	[S5][S18][S45][S46] [S62][S69]
	Visualization	Architecture visualization can help to understand the architectural structure	[S35][S72][S73]
	Management optimization	Optimizing management skills and improving management commitment	[S73]

2.4.8 RQ7: What are the difficulties when detecting, addressing, and preventing architecture erosion?

The difficulties refer to the obstacles of detecting, addressing, and preventing AEr during the development process. We collected and extracted the difficulties from the selected studies and classified them into three types. We list the main difficulties, types, and descriptions of the difficulties in Table 2.16.

Table 2.16: Classification for the difficulties of detecting, addressing, and preventing architecture erosion

Category	Type	Description	Studies
Detection of AEr	Lack of dedicated techniques and tools	Due to a lack of efficient tools and techniques, it is very hard to manually detect AEr in software systems	[S10][S13][S24] [S27][S29][S31] [S35][S48][S57]

(Continued) Table 2.16

Category	Type	Description	Studies
	Hard to establish mapping relations	Due to various factors, it is hard to establish mapping relations between source code and architectural elements (e.g., components, dependencies)	[S10][S26][S29] [S36][S37][S39] [S47][S51][S57]
	Limitation of detection approaches	Due to the specification of domains and software models, some methods can hardly be generalized to detect AEr in different development scenarios	[S9][S13][S14] [S33][S35][S38] [S48]
Handling of AEr	Hard to keep the architectural consistency	When the system evolves over time, it is hard to keep the implemented architecture aligns with the intended architecture	[S6][S18][S20] [S45][S60][S71]
	Labor and time constraint	It is usually hard to manually keep the architecture consistent with source code and completely remove AEr due to the labor and time constraints	[S20][S45][S53] [S54]
	Lack of broad understanding of software systems	Hard to get access to a broad understanding of the whole system for repairing AEr	[S60][S69]
Others difficulties	Verification	Hard to validate the effects of AEr repairing	[S46]
	Assessing the cost of stopping erosion	Hard to evaluate the cost to stop AEr in software systems	[S73]

Detection of AEr. Studies in this category can be classified into three types as shown in Table 2.16). (a) *Lack of dedicated techniques and tools/plug-ins for detecting AEr*. For example, in [S24] and [S29], the authors mentioned that there was a lack of tools specifically geared towards addressing AEr. (b) *Hard to establish a mapping relation*. For instance, due to various reasons (e.g., lack of documentation, developer turnover), the authors of [S10] mentioned that it could be very difficult to know about the original design decisions for establishing mapping relations be-

tween source code and architectural elements (e.g., components, dependencies). When the correlation of this kind of understanding is missing and it will become the obstacle for identifying their impact on AEr. (c) *Limitation of detection approaches*. Due to the specialization of domains and software models, some of the existing approaches can hardly be reused in the typical software development. For example, the authors of [S13] claimed that the specification of consistency constraints depended on the syntax and definitions of specialized models, and it may need to repeatedly define all the models where AEr might occur. This kind of redundancy constrains the reusability of this type of methods and the extensive use for detecting AEr in industry.

Handling of AEr. Even if erosion is found in software systems, it is still a challenge to handle AEr. (d) *Hard to keep the architectural consistency*. For example, the authors of [S18] mentioned that keeping the intended architecture consistent with the implemented architecture is still very hard for software engineers in maintenance phases, since engineers must cope with obsolescence and maintenance when systems evolve. (e) *Labor and time constraints*. Manually ensuring the architecture keep consistent with the implementation of a system might be quite labor-intensive and time-consuming. For example, the authors of [S20] stated that manually ensuring the consistency between the implementation and the dependency model is a laborious, time-consuming, and error-prone task, even for smaller systems. (f) *Lack of broad understanding*. Lack of a broad and in-depth understanding of the whole system is also an obstacle for completely removing and repairing erosion (e.g., [S60], [S69]). Hence, it is a challenge for practitioners to find optimal ways to fix the eroded architecture.

Other difficulties. (g) *Verification*. In most of the cases, maintainers are not the original developers, for example, the authors of [S46] mentioned that this situation increased the difficulties for optimizing the original modularization. Meanwhile, re-modularizing the system structure might obtain new modularizations that are different from the original structure, and it is hard to validate the effect of such kind of erosion repairing. (h) *Assessing the cost of stopping erosion*. Besides, the authors of [S73] mentioned that it is hard to assess the cost of stopping AEr to software systems and convey the cost to non-technical stakeholders.

2.4.9 RQ8: What are the lessons learned about architecture erosion in software development?

The selected studies provide numerous and valuable lessons on the handling of AEr. The lessons discussed in most of the selected studies are about the experience on handling AEr. Generally, a lesson could be one or more sentences, or a paragraph,

we collected more than 200 lessons learned from the selected studies. Moreover, we classified the collected lessons into five types below and provide representative examples of each type in Table 2.17.

Table 2.17: Lessons learned about architecture erosion in software development

Lessons learned	Description	Studies
Tackling of AEr	(1) Detecting and removing deviations might not efficiently mitigate erosion [S1]. (2) Dealing with AEr after it has happened is more difficult and costly, and AEr should be handled once it has been introduced in a system [S8].	[S1][S7][S8][S11][S16] [S17][S21][S27][S28] [S31][S34][S37][S40] [S41][S44][S45][S46] [S47][S48][S51][S52] [S53][S54][S55][S57] [S60][S62][S69][S70]
Manifestation of AEr	(1) The authors found that the erosion process often intertwined with the drift process [S31]. (2) The authors learned that eroded architecture might not be observed by performance [S34].	[S1][S7][S11][S17][S22] [S27][S28][S30][S31] [S34][S35][S41][S47] [S49][S51][S53][S55] [S56][S57][S62][S64] [S68][S69][S73]
Detection of AEr	(1) Identifying the hotspots in the architecture can help to identify erosion [S1]. (2) The detection strategies of erosion must rely on certain captured architectural information [S27].	[S1][S22][S27][S31] [S33][S34][S35][S36] [S49][S51][S52][S53] [S57][S60][S63][S65] [S66][S68][S69][S73]
Understanding of AEr	(1) The more the software system evolves, the greater possibility of erosion happens [S21]. (2) The authors mentioned that AEr cannot be avoided completely [S69].	[S1][S7][S9][S13][S21] [S33][S34][S35][S44] [S45][S47][S53][S55] [S56][S57][S60][S66] [S69]
Prevention of AEr	(1) Establishing a culture and offering supportive management is crucial for preventing erosion [S7]. (2) The authors suggested that minimizing human interference in routine maintenance activities contributes to erosion prevention when employed self-adaptation strategies [S34].	[S7][S11][S13][S34] [S51][S53][S61][S73]

Tackling of AEr. The lessons in this type contain the experience on how to tackle erosion in the software development life cycle. For example, in [S41] and [S51], the authors mentioned that blindly optimizing one symptom of AEr might make other symptoms worse. Therefore, it is necessary for developers to understand the relationships between AEr symptoms and maintenance activities.

Manifestation of AEr. This type of lessons refers to the manifestation of eroded

architecture in software development, such as characteristics and impact on performance. For example, in [S1], the authors mentioned that not all deviations (e.g., short-term deviations only in few versions) in an architecture manifest erosion trend.

Detection of AEr. The lessons in this type are about the experience on detecting erosion in the software development process. For example, the authors of [S31] found that the drift process often intertwined with the erosion process, and detection of drift symptoms might help to detect erosion symptoms or vice versa.

Understanding of AEr. This type of lessons contains how practitioners understand AEr in software development, such as characteristics and attributes of eroded architecture in software development. For example, the authors of [S7] mentioned that almost all projects would suffer from erosion sooner or later unless taking some effort to overcome it.

Prevention of AEr. The lessons in this type include the findings about which means are effective in preventing AEr. For example, the authors of [S13] found that AEr might also occur in model-driven software development. Hence, formulating and regularly checking the architectural rules with the model-implementation process can help to prevent architecture from sliding into erosion during the software development life cycle.

2.5 Discussion

In this section, we analyze the results of each RQ (see Section 2.5.1) and discuss their implications for researchers and practitioners (see Section 2.5.2), respectively. To demystify the AEr phenomenon, we first provide a conceptual model of AEr (see Figure 2.7) to present our findings and the relationships between various aspects (i.e., definitions, symptoms, reasons, and consequences) of AEr according to the results of this SMS in Section 2.4. This conceptual model acts as a panorama to understand the nature of the AEr phenomenon.

Regarding each aspect shown in Figure 2.7, we discuss their elements in Section 2.5.1. The conceptual model shows that the understanding of AEr (i.e., definition perspective) consist of four perspectives (see Section 2.4.2), and presents the types of AEr symptoms (see Section 2.4.3) that can be considered as the manifestation of AEr. On the other side, the AEr symptoms can directly or indirectly give rise to diverse consequences to software projects. Moreover, the conceptual model highlights that both technical and non-technical reasons that can trigger the occurrence of AEr (see Section 2.4.4), as well as the negative effects caused by AEr (see Section 2.4.5).

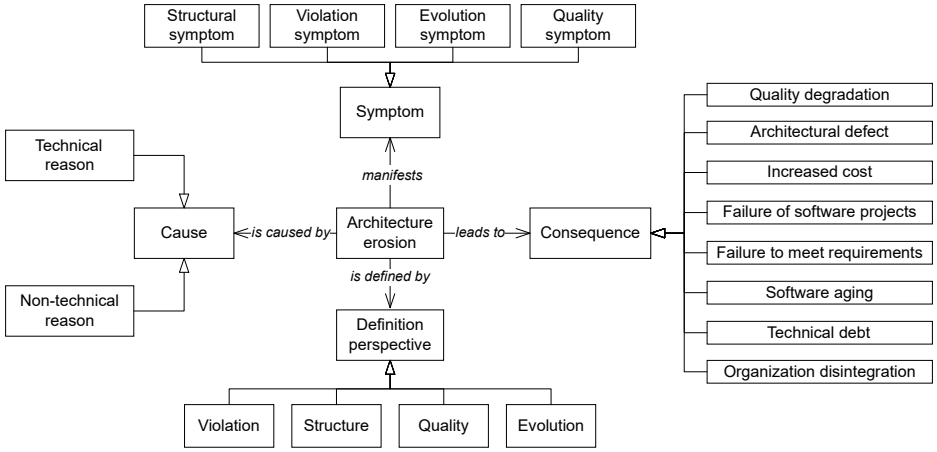

Figure 2.7: A conceptual model of architecture erosion according to the RQ results

2.5.1 Analysis of results

2.5.1.1 Definition of architecture erosion

Section 2.4.2 reveals that the concept of AEr has been defined in different terms and perspectives. Table 2.8 shows that only 54.8% (40 out of 73) of the studies proposed a definition of the AEr phenomenon. According to the terms listed in Table 2.7, some of these terms are general concepts, which refer to the constant decay of the system structure (at the system level), such as software aging (e.g., [S5]), software erosion (e.g., [S41]), design rot (e.g., [S7]), system rot (e.g., [S72]), and modular/modularity deterioration (e.g., [S46]); certain terms are used to describe software anomalies at the code level, such as code decay and code rot (e.g., [S28]); some other terms focus on the AEr phenomenon at the architectural level, and the architectural problems are described as architecture degradation or architecture degeneration, which represents the introduction of architecturally-relevant problems and continuous decline of architectural modularity (e.g., [S27], [S31], [S45], [S57], [S67]). Moreover, according to the extracted definitions of AEr, we found that architecture erosion, architecture decay, and architecture deterioration have the same meaning when describing the AEr phenomenon (e.g., [S15], [S16], [S17], [S21]), which refers to the architectural deviation between the intended and implemented architecture, and both the intended and implemented architecture may change with the changing demands of stakeholders.

Besides, certain definitions of AEr are not included in this SMS, since the publication years of these studies are not within the time period of our SMS. For example,

Jaktman *et al*. (Jaktman et al., 1999) defined AEr as "*structure of a software architecture to be eroded when the software becomes resistant to change or software changes become risky and time consuming*", and this definition focuses on the structure perspective (see Table 2.8); Wang *et al*. (Wang et al., 2019) defined that "*architecture erosion is a phenomenon that occurs when architecture quality is decreased with software evolution*", and this definition focuses on the quality perspective (see Table 2.8). The two definitions above can also support that AEr is more than a simple phenomenon about the violations of intended architecture, and the relationships between the four perspectives are worth further investigation. Furthermore, the AEr definitions in four perspectives imply that AEr occurred at different levels of abstraction, and this finding also shows that AEr is a multifaceted phenomenon. The results of our recent study from the practitioners' perspective (Li, Liang, Soliman and Avgeriou, 2021b) also indicate that AEr is a multifaceted phenomenon and practitioners describe this phenomenon from four perspectives: structure, quality, maintenance, and evolution. Interestingly, we noticed that there is only a slight difference in the perspectives of AEr between academia and industry. Compared to the violation perspective identified in this SMS, practitioners care more about the maintenance perspective of AEr. One potential reason could be that the maintainers may not be the original architects in practice, thus practitioners discuss more about the maintainability issues caused by AEr, while academia might focus more on the nature of AEr. Considering this, we provide a refined definition of AEr according to the results (i.e., the four perspectives) of our SMS: **architecture erosion happens when the implemented architecture violates the intended architecture with flawed internal structure or when architecture becomes resistant to change**.

> ***Finding 1****: AEr is a multifaceted phenomenon, which is often described from four perspectives: violation, structure, quality, and evolution.*
> ***Finding 2****: we provide a refined definition of AEr according to the results (i.e., the four perspectives) of our SMS: architecture erosion happens when the implemented architecture violates the intended architecture with flawed internal structure or when architecture becomes resistant to change.*

2.5.1.2 Symptoms of architecture erosion

Table 2.9 shows that the symptoms of AEr can be mapped into the perspectives of AEr phenomenon (see Section 2.4.2). We found that 32.9% (24 out of 73) of the selected studies mention the symptoms of AEr, and nearly half of them (41.7%, 10 out of 24) regard violations of design decisions, principles (e.g., abstraction, encapsu-

lation), or constraints as the symptoms of AEr. Moreover, most of the symptoms (70.8%, 17 out of 24) are related to structural anomalies, especially some architectural smells (e.g., undesired dependencies).

According to Table 2.9, we observed that structural symptoms receive more attention among the majority of symptoms. One potential reason could be that it is relatively easy to detect structural issues by employing various tools or visualizing the dependency relationships of components. Besides, compared to individual structural symptoms, the agglomerations of structural symptoms are even stronger indicators of AEr. For example, Oizumi *et al*. (Oizumi et al., 2016, 2015) studied the impact of code anomaly agglomeration (i.e., a group of inter-related code anomalies) on software architecture and the results show that more than 70% of architectural problems are related to code anomaly agglomerations. Their studies confirm that code anomaly agglomerations are better than individual code anomalies to indicate the presence of architectural problems. Besides, Microservices Architecture (MSA), as a popular architectural style used to increase resilience to AEr (Chen, 2018), can also show certain AEr symptoms (such as architectural smells). For example, Mumtaz *et al*. (Mumtaz et al., 2021) presented several service-oriented and service-specific architectural smells (e.g., Shared Libraries, Missing Package Abstractness) in MSA, which are a typical type of structural symptoms of AEr (see Table 2.9). Considering the popularity of using MSA in industry in recent years, the potential shortcomings and risks of MSA related to AEr need to be further investigated.

In addition, we found that it is a challenge to adequately quantify various AEr symptoms, though several studies (e.g., [S16], [S41], [S26]) attempted to propose metrics and tools for detecting AEr symptoms. For example, ARCADE mentioned in [S16] is a tool used to analyze and quantify different structural symptoms (e.g., architectural smells, dependency information) for given systems, which is employed to identify and recover the eroded architecture (e.g., Schmitt Laser et al. (2020); Garcia et al. (2022)). A potential drawback of entirely relying on structural symptoms might ignore other AEr symptoms (e.g., evolution symptoms). Besides, note that, although quality degradation (as a quality symptom) could emerge alongside continuously eroding architectures, not all quality degradation is related to architectural issues; for example, inconsistent coding styles may not have an impact on architectural integrity but lead to reduced readability and maintainability. Quantifying the AEr symptoms can help to recognize AEr and its degree in software development. Although a wide variety of studies mentioned diverse AEr symptoms, there is a lack of diversity about AEr symptoms in the empirical validations. For example, we did not find many in-depth validations of which symptoms are strong indicators of AEr and how to detect those symptoms.

***Finding 3**: AEr symptoms can be classified into four categories: structural symptom, violation symptom, evolution symptom, and quality symptom, where structural symptoms receive the most attention.*

2.5.1.3 Reasons of architecture erosion

Every architecture will undergo erosion sooner or later as long as the evolution happens (Merkle, 2010), which is a process of increasing entropy. In a broad sense, AEr does not arise spontaneously and the root reason of AEr comes about through *change*. According to Table 2.10 and Table 2.11, we observed that the reasons of AEr are more than from technical factors, but also related to non-technical factors. The findings show that *architecture violation* (24.7%, 18 out of 73 studies), *evolution issue* (23.3%, 17 out of 73 studies), and *technical debt* (17.8%, 13 out of 73 studies) are the three main technical reasons of AEr, and *knowledge vaporization* (15.1%, 11 out of 73 studies) is the most frequently mentioned non-technical reason. One possible reason is that the maintenance and evolution phases account for a large part of the life cycle of software development, and these frequently mentioned reasons are relatively easy to be involved in various artifacts and software development activities. *Requirement issue* (15.1%, 11 out of 73 studies) is a common reason of AEr related to both technical and non-technical reasons, since requirements (including functional and non-functional requirements) have a significant impact on system design and evolution, and the requirements-design gap can affect different development activities during the software development life cycle.

Additionally, the reasons of AEr listed in Table 2.10 may have interactive relationships. For example, *increasing complexity* can lead to *understanding issue* (e.g., [S34]), while *understanding issue* may be caused by other reasons, such as *architecture violation* (e.g., [S45]). *Technical debt* might give rise to *suboptimal design decisions*, but not all of the *suboptimal design decisions* are derived from *technical debt* (e.g., specific design constraints with low extensibility). Thus, the potential relationships between the reasons are also worthy of further exploration. We believe that our work can serve as a good starting point for future research on exploring the relationships between the collected reasons.

***Finding 4**: AEr does not arise spontaneously and the root reason of AEr comes about through **change**. AEr is caused both by technical factors (e.g., architecture violation, evolution issue, and technical debt) and non-technical factors (e.g., knowledge vaporization) which are intertwined.*

2.5.1.4 Consequences of architecture erosion

As shown in Table 2.12, we mapped the consequences extracted from the selected studies into eight categories. 50.7% (37 out of 73) of the studies mention or investigate the impact of eroded architecture on software systems, where most of them (83.8%, 31 out of 37 studies) are related to quality degradation issues. The priority of tackling the consequences relies on the severity and degree of impact on different software Quality Attributes (QAs) (e.g., maintainability, evolvability, extensibility), and the significance of these QAs to the companies. For instance, Mozilla Web browser (Godfrey and Lee, 2000), an application from Netscape comprised over 2,000,000 source lines of code, is a frequently mentioned example of AEr in industry. Netscape engineers took six months to conclude that the original architecture layers of this application were eroded and irreparable. However, they spent two years redeveloping Mozilla Web browser and re-architecting the source code and dependencies. For small companies, it might lead to *failure of software projects*, while it might bring about *increased cost* to redevelop this application for big companies like Netscape.

Not surprisingly, the consequences might damage architectural structures and generate software defects. Ultimately, massive labor and time cost have to be invested into the projects for incorporating the new requirements and restructuring the source code, and the worst situation is to rebuild a system from scratch. The results of RQ4 can shed light on the significance of AEr in software development. Knowing about various consequences can draw more attention to AEr phenomenon and further warn researchers and practitioners to avoid the potential consequences of AEr. Notably, the findings indicate that technical debt might not only be one of the AEr reasons, but also one of the AEr consequences. It is worth noting that correlation does not imply causation and the accumulated unpaid technical debt might give rise to AEr with high probability. For example, Mo *et al*. [S49] mentioned that systems with AEr happened can incur technical debt where short-term compromises lead to significant long-term problems (e.g., fixing bugs or adding new features). The findings imply that technical debt might have a feedback loop with negative consequences, including decreasing reliability and increasing complexity. This observation suggests that more empirical work should be performed to investigate the relationships between technical debt and AEr.

***Finding 5**: AEr can lead to various consequences, such as damaging architectural structures and generating software defects. The priority of tackling the consequences of AEr often relies on the severity and degree of impact on different software quality attributes.*
***Finding 6**: Technical debt might not only be one of the AEr reasons, but also one of the AEr consequences, forming a vicious circle between the two phenomena.*

2.5.1.5 Approaches and tools for architecture erosion detection

We found that 54.8% (40 out of 73) of the selected studies proposed or employed approaches for detecting AEr and we classified these approaches into four categories (see Table 2.13). The results of RQ5 reveal that the research effort on AEr detection has mainly focused on the consistency-based approaches (62.5%, 25 out of 40 studies), where the most frequently employed evaluation-based approach is Architecture Conformance Checking (ACC). One potential reason is that ACC is straightforward to assist stakeholders in checking whether the implemented architecture complies with the intended architecture. Moreover, our results show that the evolution-based approaches (22.5%, 9 out of 40 studies) and defect-based approaches (15.0%, 6 out of 40 studies) receive similar concerns from the literature. One reason may be that the diversity of source code analysis tools makes the two types of approaches more applicable to different software projects.

The findings also show that 39.7% (29 out of 73) of the selected studies mentioned relevant tools that can be employed in detecting AEr (see Table 2.14), where the majority of the tools (57.1%, 20 out of 35 tools) used to detect AEr are designed based on ACC. Additionally, we observed that many tools employed to detect AEr are implemented to support the approaches in Table 2.13 besides ACC, which demonstrates the necessity of close collaboration between academia and industry. Moreover, we noticed that half of the tools are commercial tools and more than half of the tools only support one specific programming language, which could be an obstacle for the tools to be widely employed in industry, especially when multi-programming-language development is booming (Li, Qi, Yu, Liang, Mo and Yang, 2021). Although some tools are not specifically devoted to AEr detection, to some extent these tools can provide support for detecting AEr symptoms, which would be beneficial to identify architectural anomalies at the early stage of AEr occurrence during the software development life cycle.

In our recent work (Li, Liang, Soliman and Avgeriou, 2021b), we found that developers claimed that there are no dedicated tools for detecting AEr. However, according to the results in this SMS, for example, the Axivion tool can help to detect

and stop AEr. One potential reason is that a gap still exists on AEr between research and practice, and developers might not be familiar with the existing approaches and tools in the literature. Therefore, AEr detection approaches and tools can be empirically evaluated through case studies. For instance, it is interesting to see some approaches and tools proposed in the literature, but there is limited empirical evidence regarding their effectiveness and productivity. Hence, we encourage more collaborations between academia and industry in the area of AEr detection and advocate more empirical studies on the validation of existing approaches and tools.

***Finding 7**: The majority of the approaches and tools used to detect AEr are mainly focused on architectural consistency. The diversity of the existing approaches and tools should be increased, since more than half of the tools only support one specific programming language.*
***Finding 8**: A gap still exists on AEr between research and practice. More dedicated approaches and tools should be designed to detect AEr and empirical studies should be conducted to evaluate the existing approaches and tools.*

2.5.1.6 Measures for addressing architecture erosion

We identified and collected the measures for addressing AEr from the selected studies and classified the measures into three categories (see Table 2.15). Table 2.15 shows that most of the preventive measures are employed to prevent AEr by checking, monitoring, and evaluating system architecture. One reason is that establishing and checking the consistency between the specifications and implementation (e.g., using formal methods and traceability mechanisms) can effectively eliminate the possibility of deviating the implemented architecture from the intended architecture. In comparison to preventive measures, remedial measures are maintenance-oriented measures and closely related to reverse engineering and re-engineering techniques, such as architecture recovery. To address AEr in software systems, engineers need to understand the architectural structure (e.g., the layered pattern) and reasons about the changes through the recovered architecture (e.g., restoring architecture views from source code). According to Table 2.15, remedial measures are not as widely employed in practice compared to preventive measures. The reason could be that the extent of architecture recovery is largely dependent on code quality (e.g., code readability) and the availability of architecture knowledge. In addition, if the system has been severely eroded, then maintenance may not be a cost-effective choice; consequently, discarding the eroded system and building a new system from scratch would be a feasible option in terms of return on investment.

Nevertheless, prevention is better than cure. In recent years, Microservices Architecture (MSA), as a cloud-native architecture, has grown in popularity in industry due to its benefits of flexibility, loose coupling, and scalability. To some extent, decomposing the original monolithic architecture into microservices can improve architecture extensibility and increase its resilience to AEr. Moreover, with the rising popularity of MSA, there is a growing number of companies choosing to migrate their existing monolithic applications to MSA through decomposition and service interactions. Migrating a monolithic architecture to a cloud-native architecture like MSA can help engineers maintain large software systems, through increasing flexibility to adopt new technologies, reducing the time-to-market, and managing independent resources for diverse components (Balalaie et al., 2016). For example, Chen (Chen, 2018) reported their practices and experience when they moved their eroded applications to an MSA, and they observed increased deployability, modifiability, and resilience to AEr. The reason is that MSA can create physical boundaries between microservices, since each microservice has its own codebase, development team, and runs in its own containers. Thus, MSA provides better protection against the temptation of breaking the boundaries between services (a typical violation symptom of AEr, see Table 2.9). Therefore, architects can choose well-accepted architecture patterns (e.g., MSA) to gain control of the systems when facing increasing complexity, as well as adhere to design principles that allow the decomposition of a complex system into more understandable chunks (Sturtevant, 2017). Note that, although MSA provides a promising way to deal with complicated architectural issues, MSA is definitely not a silver bullet and it can raise new architectural problems. For example, an experience report from Balalaie *et al*. (Balalaie et al., 2016) indicates potential risks that might be related to AEr, for example, microservices implemented based on different programming languages could lead to major issues, rendering the system unmaintainable.

> ***Finding 9**: Measures used to address AEr include preventive measures (e.g., architecture conformance checking, architecture monitoring and evaluation) and remedial measures (e.g., architecture maintenance and restoration).*

2.5.1.7 Difficulties on handling architecture erosion

As shown in Table 2.16, we classified the difficulties in detecting, addressing, and preventing AEr into three categories. Most of the difficulties presented in Section 2.4.8 are still challenges in the current research and practice of software engineering, but a few difficulties have been partly addressed by the measures presented in

Section 2.4.7, such as establishing the mapping relationships between design and architectural elements. For example, there are still no effective mechanisms to deal with the consistency between System-of-System (SoS) architectural instance models and SoS abstract architecture models (e.g., [S29]). Besides, architects and developers need suitable tool support for AEr detection and the lack of dedicated tools raises difficulties for detecting and addressing AEr. For example, Python has become a popular programming language in recent years[1], while we found that more than half of the tools (see Section 2.4.6.2) only support one specific programming language (e.g., Java) and only a few tools support Python projects. Additionally, the results reflect the significance of understanding and handling AEr. It is critical to raise the awareness of the AEr risks in software systems and identify potential AEr symptoms, since having a holistic understanding of the whole system is the prerequisite for mitigating the negative consequences of AEr.

***Finding 10**: The difficulties of detecting, addressing, and preventing AEr are mainly derived from the detection and handling of AEr, while we still lack effective mechanisms (e.g., dedicated approaches and tools) to handle AEr in various software systems (e.g., SoS, MSA).*

2.5.1.8 Lessons learned on handling architecture erosion

As mentioned in Section 2.4.9, the lessons refer to the experience on handling AEr. The lessons learned on handling AEr are classified into five types, and we provide two examples for each type of lessons learned (see Table 2.17). The findings show that the majority (39.7%, 29 out of 73) of the selected studies mentioned the lessons about the tackling of AEr, followed by lessons about the manifestation of AEr (32.9%, 24 out of 73 studies) and detection of AEr (27.4%, 20 out of 73 studies). One reason is that maintenance and evolution phases account for a large part of the software development life cycle, and AEr phenomenon is most likely to be observed by maintainers and testers during the maintenance and evolution phases. Moreover, we observed that only 11.0% (8 out of 73) of the studies discuss the lessons about AEr prevention. One potential reason is that AEr prevention is relatively difficult and not always feasible. For example, the authors of [S34] mentioned that "*preventing erosion completely is a difficult task and may not be feasible*" and the authors of [S69] mentioned "*in general architecture erosion cannot be avoided completely*". Another reason is that there are various causes related to AEr (see Section 2.4.4) that it is nearly impossible to completely prevent the reasons from happening, especially for

[1] `https://insights.stackoverflow.com/survey/2020`

complex software systems with millions of lines of code. To some extent, AEr prevention means that taking feasible measures to extend the software system life cycle as long as possible. Additionally, we suggest that stakeholders should raise awareness of AEr, create a culture of organizational support, and practitioners should be encouraged to fight for AEr.

> ***Finding 11**: More than 200 lessons are collected from the selected studies that are concentrated on five types: tackling, manifestation, detection, understanding, and prevention of AEr; the majority of these lessons concern tackling AEr.*

2.5.2 Implications

This SMS provides a comprehensive insight into the AEr-relevant studies and the results of the RQs provide significant implications for both researchers and practitioners. In this section, we discuss the implications of the findings and highlight the promising research directions on AEr.

2.5.2.1 Implications for researchers

As shown in Figure 2.4, we found that most (82.2%, 60 out of 73) of the selected studies are solely from academia, and we encourage researchers to seek more collaborations with industrial partners to fill the gap between academia and industry and solve the existing challenges.

According to the results of RQ1 (see Section 2.4.2.1) and their analysis (see Section 2.5.1.1) about the AEr definitions, we highly recommend that both researchers and practitioners can **describe and define the term when they refer to AEr phenomenon and use the common term "architecture erosion"** consistently for minimizing ambiguities and misunderstanding. In addition, the results of RQ1 (see Section 2.4.2.2) and their analysis (see Section 2.5.1.1) also indicate that **AEr is a multifaceted phenomenon** and it manifests not only through architectural violations and structural issues, but also affects the quality and evolution of software systems. Future research should consider the four perspectives of AEr when evaluating architecture and analyzing architecture evolution. For example, different metrics-based evaluations can be conducted from the four perspectives when software engineers are engaged in maintenance and evolution activities.

As discussed in Section 2.5.1.2 (the analysis of the results of RQ2), although a wide variety of studies mentioned diverse AEr symptoms, we notice that there is **a lack of empirical validation of these AEr symptoms**. Researchers can investigate which AEr symptoms are strong indicators of AEr and how to detect and further

quantify the symptoms. For example, researchers can attempt to establish a comprehensive evaluation method by either employing the existing or proposing new approaches and tools to evaluate and quantify the structural symptoms (e.g., architectural smells).

The results of RQ3 (see Section 2.4.4) show that **the occurrence of AEr in software systems may derive from various reasons**, and researchers can empirically investigate the potential relationships between the reasons of AEr. Additionally, we notice that technical debt might be related to part of the technical and non-technical reasons of AEr. We believe that it is valuable to investigate when and how technical debt can induce AEr.

The results of RQ4 (see Section 2.4.5) and their analysis (see Section 2.5.1.4) show that **the prevalence of AEr leads to varying degrees of impact on software systems, and not all types of consequences have received the same attention**. As shown in Table 2.12, researchers are more likely to notice the quality attributes degradation when AEr happens. It would be interesting to empirically study the measures and costs taken to handle the eroded architecture. For example, researchers can conduct case studies with industrial partners to investigate the financial loss of different consequences identified in this SMS brought into the software projects suffered from AEr.

As discussed in Section 2.5.1.5 (the analysis of RQ5 in Section 2.4.6), although our recent work (Li, Liang, Soliman and Avgeriou, 2021b) investigated part of the causes and consequences of AEr from the practitioners' perspective, there remains **a dearth of empirical case studies on exploring the reasons and consequences of AEr** in a real-life industrial setting. Besides, comparative studies about different **approaches for tackling AEr are deficient** in the field of software architecture. Therefore, further research can focus on exploring the potential reasons and consequences of AEr, as well as the effectiveness of relevant approaches for tackling AEr using industrial cases.

Although some approaches and tools (the results of RQ5, see Section 2.4.6) can be useful to detect AEr (e.g., Axivion, Structure101, Lattix), there exists a demand to evaluate the performance of different approaches and tools and verify the capability (e.g., merits and demerits) of these approaches and tools. We encourage researchers to conduct empirical studies for **evaluating these AEr detection approaches and tools**. Besides, researchers can evaluate the approaches and tools reported in Section 2.4.6 and explore their scope, characteristics, and metrics for providing a solid foundation on designing dedicated tools.

As discussed in Section 2.5.1.6 (the analysis of the results of RQ6), there are **scarce research on the measures for addressing AEr in emerging systems, architecture styles, and development methods**, such as SoS, MSA, and DevOps, which is an

interesting and meaningful research field to be explored. For example, researchers can pay more attention to the potential risk of AEr (e.g., erosion symptoms, the gap between designed and implemented architecture) in MSA-based systems.

From the identified **difficulties** (the results of RQ7 in Section 2.4.8 and their analysis in Section 2.5.1.7) and the **lessons learned on handling AEr** (the results of RQ8 in Section 2.4.9 and their analysis in Section 2.5.1.8), we observed that the results provide the challenges in this area that would inspire more academic and industrial collaboration and promising measures for the identification, handling, and prevention of AEr. For example, automatically detecting the AEr symptoms by leveraging machine learning techniques can be an effective and efficient way compared to manual analysis of components and source code.

2.5.2.2 Implications for practitioners

As discussed in Section 2.5.1.1 (the analysis of the results of RQ1), practitioners are encouraged to reach an agreement on the **understanding of AEr phenomenon for reducing the ambiguity of the AEr concept**. For example, we recommend practitioners to use the common term "architecture erosion" to refer to the AEr phenomenon, which helps to minimize the misunderstanding and confusion for reaching a common ground on the description of the AEr phenomenon.

There are diverse technical reasons and non-technical reasons that can give rise to AEr (see Table 2.10 and Table 2.11, the results of RQ3). It is critical for practitioners to realize that certain technical reasons (e.g., technical debt) and non-technical reasons (e.g., knowledge vaporization) of AEr, which may not have a noticeable effect on system quality in a short time, but they can negatively impact architectural understanding and integrity in the long term, thereby leading to AEr. For example, practitioners can attach importance to the non-technical reasons of AEr and cultivate a culture of AEr prevention, such as exposing and discussing architectural changes, improving organization management skills (especially about the training and education of developers to help them be familiar with system architecture).

The analysis of the results of RQ4 in Section 2.5.1.4 indicate that practitioners need to **raise awareness of the grave consequences of AEr and request actions at the management level**, in order to obtain a high priority to tackle AEr related issues and reduce the risk of architecture sliding into erosion and failure. To prevent and repair AEr to some extent, we suggest that practitioners should build the mechanisms that support recording and tracing architectural knowledge (e.g., decisions, assumptions) to source code and architectural changes, and regularly conduct architecture assessment (e.g., architecture conformance checking).

Practitioners should also pay attention to the state of the art of academic research results, since we noticed that **there is a gap between research and practice** (see the analysis of the results of RQ5 in Section 2.5.1.5). Practitioners are encouraged to apply and adapt the proposed approaches and tools according to their needs and project contexts, which can help to identify not only the limitation of those approaches and tools but also the real needs from practitioners for detecting AEr in practice.

It is critical to conduct early diagnoses of AEr in software systems. **The approaches and tools for detecting AEr are still scarce and have some limitations** (e.g., domain/language-specific features). The results of RQ5 (see Section 2.4.6) imply that there exists a demand for dedicated approaches and tools to support AEr detection in practice. Therefore, practitioners are encouraged to collaborate with researchers and employ dedicated approaches and tools for detecting AEr. Such dedicated tools can better support multiple programming languages, detection of the AEr symptoms, and the quantitative visualization of the trend of AEr.

Some measures used to address AEr are specific to the contexts in the studies (see the analysis of the results of RQ6 in Section 2.4.7). When it comes to AEr prevention and remediation, practitioners need to first understand the corresponding contexts of employing the measures, and then adapt the measures according to the specific situations.

It is never too late to fight against AEr. The difficulties (see the analysis of the results of RQ7 in Section 2.4.8) and valuable lessons (see the analysis of the results of RQ8 in Section 2.4.9) we collected can be beneficial for practitioners to better cope with AEr during the development life cycle, avoid the pitfalls of AEr, and further explore the benefits and limitations of applying the measures for addressing AEr in practice. Besides, further industrial evidence is required to validate the benefits of the measures as well as the potential pitfalls.

To sum up, the findings of this mapping study can help practitioners to better understand the practical impact of the AEr phenomenon, supplementing our previous industrial survey on AEr (Li, Liang, Soliman and Avgeriou, 2021b) and empirical study on AEr symptoms identified by code review in OSS projects (Li, Soliman, Liang and Avgeriou, 2022). Specifically, Findings 1, 2, and 3 provide a comprehensive understanding of the AEr phenomenon; Findings 4, 5, and 6 shed light on the potential reasons and consequences of AEr; Findings 7 and 8 show the increasing demand for dedicated AEr detection tools; and Findings 9, 10, and 11 highlight the importance of tackling AEr in practice.

2.6 Threats to validity

In this section, we discuss the potential threats to the validity of our SMS, as well as the measures that we took to mitigate the threats according to the guidelines in (Zhou et al., 2016; Wohlin et al., 2012).

2.6.1 Construct validity

Construct validity concerns whether the theoretical and conceptual constructs are correctly interpreted and measured. In this SMS, the main threats to construct validity include study search and selection:

Study search. There may be relevant studies that were omitted, which will affect the completeness of the retrieved results. To mitigate this threat, we searched seven popular electronic databases (see Table 2.2) that publish software engineering research, and we also conducted a manual search from well-known venues (including conferences, journals, and workshops) closely related to the topic of our SMS, i.e., architecture erosion. Before the formal search, a pilot search was performed to enhance the suitability (e.g., the search terms and time period) and quality of this SMS. Additionally, to ensure the completeness of the study search, we employed the "snowballing" technique (Wohlin, 2016) to include any potentially relevant studies.

Study selection. Whether or not to include relevant studies mainly depends on the understanding of the researchers who were involved in the study search and selection, and this inevitably introduced personal bias due to the limitation of personal knowledge. To minimize the selection bias, (1) we selected 100 papers as a sample, which were selected by two researchers independently in three rounds, to measure their inter-rater agreement on study selection; (2) we also defined a set of inclusion and exclusion criteria regarding the study selection and discussed among four researchers about the uncertain studies for reaching an agreement and reducing the risk that relevant studies were omitted.

2.6.2 Internal validity

Internal validity pertains to the study design that has a potential impact on the results. In this SMS, the main threats to internal validity are concerned with the extracted data and the synthesis of the results:

Data extraction. One potential threat is about the quality of the extracted data, which might have a negative impact on the data synthesis and classification results. Several measures were taken to mitigate the bias of researchers who conducted data extraction. First, to ensure the consistency of data extraction results, we discussed and formulated the data items by four researchers to reach a consensus on the con-

tent of data to be extracted. Moreover, before the formal data extraction, a trial data extraction with five selected studies was performed by the author and checked by another two researchers. Furthermore, we selected another five papers from the selected results to conduct a data extraction by two researchers, independently. Any disagreement was discussed and resolved together for reaching an agreement about the understanding of the data items (specified in Table 2.4). Furthermore, the data extraction results from the author were checked by another two researchers according to the description of each data item.

Data synthesis. The quality of data synthesis may affect the correctness of the answers to the eight RQs (see Section 2.3.1). Different researchers may have their own understanding on the extracted data, for example, the classifications of extracted data. To minimize the personal bias, we conducted continuous discussions about the divergent classifications and any conflicts were discussed until an agreement was reached. Besides, during the data synthesis process, we excluded the extracted data if we could not find any valuable information.

2.6.3 External validity

External validity concerns the extent to which the findings of a study can be generalized. This SMS provides an overview of the state of the art of the studies on AEr phenomenon in software development. Although we took measures to increase the completeness of our study search, there are still limitations and we may miss relevant studies that investigate AEr but are not covered by this SMS. To alleviate the threats to external validity, we provided the search protocol of this SMS (see Section 2.3) that rigorously specifies the execution process of this SMS, and we searched and selected peer-reviewed studies using both popular electronic databases and relevant target venues (i.e., the manual search).

2.6.4 Reliability

Reliability refers to whether the study produces the same results when other researchers replicate it. This threat is related to several factors, such as missing studies, incomplete data extraction, and improper categorization of the extracted data. We only searched and selected relevant studies written in English according to the inclusion and exclusion criteria. Nevertheless, relevant studies might still be omitted for various reasons, such as different terms used to describe the phenomenon of AEr, incompleteness of the selected venues. Additionally, a replication package of this SMS has been made available online (Li, Liang, Soliman and Avgeriou, 2021a) to improve its replicability. With these measures, we strove to ensure that this SMS can be repeated by following the procedure in Section 2.3.

2.7 Related Work

To the best of our knowledge, there are no systematic mapping studies (SMSs) or literature reviews (SLRs) that specifically focus on the AEr phenomenon. However, there are several secondary studies that are close to the topic of our SMS (i.e., architecture erosion). Therefore, we provide a brief overview of these secondary studies in this section.

Baabad *et al*. (Baabad et al., 2020) conducted an SLR with 73 primary studies focusing on the architecture degradation in OSS projects. They performed a coarse-grained SLR that mainly focuses on the reasons and solutions of architectural degradation problems in OSS projects. Their results show that architectural degradation problems, including identifying, addressing, avoiding, and predicting architectural degradation within OSS projects, are still open research issues. Compared to their SLR, our SMS has a broader scope on the AEr phenomenon. We analyzed the difference between various terms of the AEr phenomenon, provided a refined definition of AEr, and systematically analyzed and categorized the reasons, consequences, measures, difficulties, and lessons learned of AEr.

Brogi *et al*. (Neri et al., 2020) recently conducted a multivocal review on the architectural smells possibly violating the design principles of microservices architecture (MSA). The authors identified the design principles of MSA from 54 peer-reviewed papers and grey literature published before the end of January 2019. This multivocal review focuses on the architectural smells on MSA and why the smells violate the design principles, as well as the refactorings used to fix the smells. Besker *et al*. (Besker et al., 2018) conducted an SLR on architectural technical debt. They proposed a unified model of architectural technical debt that provides a new and comprehensive interpretation of architectural technical debt, since negative effects (e.g., architecture erosion) might be caused by architectural technical debt, which is also the result of this SMS (see Section 2.4.4). Bandi *et al*. (Bandi et al., 2013) conducted an SMS with 30 studies investigating the techniques and metrics which have been empirically evaluated for addressing code decay. Their SMS mainly covers the identification and minimization approaches for code decay. Herold *et al*. (Herold et al., 2016) reported the preliminary results of a literature review about architecture degradation and consistency checking. Their results show that empirical evaluation on this field is still missing and experiments and surveys should be complemented for assessing the impact of architectural degradation in practice.

The aforementioned secondary studies have a different focus, and our SMS is the only one that focuses on the AEr phenomenon. To be more specific, we systematically analyzed and categorized the concept, symptoms, reasons, consequences, detecting, handling, and lessons learned of architecture erosion through this SMS.

2.8 Conclusions

Through this SMS, we investigated the understanding, reasons, and consequences of AEr, the available approaches and tools for detecting AEr and the measures for tackling AEr, as well as the difficulties and lessons learned from the selected studies. We searched the papers in seven electric databases and manually checked the papers in 22 related venues published between Jan 2006 and May 2019 on the topic of AEr in software development, and finally selected 73 primary studies for further data extraction and analysis. Based on the extracted data, we got a comprehensive understanding of the concept of AEr and an overview of various aspects of AEr. The main points of this SMS are summarized as follows:

1. Most (82.2%) of the selected studies are from academia, and the studies from conferences (58.9%) and journals (27.4%) make up the large majority of the selected papers. The tendency in Figure 2.5 shows that AEr related studies received a fair amount of attention in the last decade.

2. Among the different terms used to describe the AEr phenomenon, "architecture erosion" is the most frequently-used term followed by "architecture decay". Both practitioners and researchers should clearly define the terms when they mention the AEr phenomenon and better use the common term "architecture erosion" to refer to the phenomenon.

3. AEr manifests not only through architectural violations and structural issues, but also causing problems in software quality and during the evolution of software systems. The four perspectives are worthy of investigation in both research and practice.

4. The AEr symptoms can be classified into four categories according to the four perspectives of AEr, and structural and violation symptoms are the most common symptoms. Additionally, AEr exists not only in maintenance and evolution phases, but may also exist in the design and implementation phases.

5. Apart from technical reasons, non-technical reasons are also significant factors that lead to AEr, which indicates that non-technical reasons (e.g., management and organization issues) cannot be ignored, and AEr can exist in different phases of software development and interacts with various stakeholders.

6. Most of the studies deem that AEr has a negative impact on QAs of software systems (e.g., maintainability, evolvability), while other consequences can also generate varying degrees of impact on systems. Practitioners should

raise their awareness of the grave consequences of AEr and request actions at the management level, in order to obtain a high priority to tackle AEr related issues and reduce the risk of architecture sliding into erosion and failure.

7. The 19 approaches for detecting AEr are classified into four categories, in which consistency-based approaches and evolution-based approaches are the most frequently mentioned approaches. Besides, 35 tools used to detect AEr were classified into four categories according to their distinctive purposes (e.g., checking architecture conformance).

8. The identified approaches, tools, and measures for detecting and addressing AEr are applicable in specific contexts (e.g., constraints or preconditions). Practitioners need to first understand the corresponding contexts of employing the measures, and then adapt the measures according to the specific situations.

9. The difficulties in detecting and tackling AEr mainly derive from the domain and language constraints, absence of dedicated tools, and establishing mapping and alignment between the intended and implemented architecture. These difficulties should receive more attention in future studies.

10. The experience and lessons learned collected from the primary studies about AEr were classified into five categories, which can provide evidence for researchers to future investigation on AEr and help practitioners avoid pitfalls when addressing AEr.

We believe that the results of this SMS could benefit the researchers and practitioners to better understand the nature of AEr, as well as its underlying reasons and consequences, and the findings can provide clues for future research in this area (as discussed in Section 2.5). We encourage more collaboration between researchers and practitioners to close the gap between academia and industry and address the existing challenges. Additionally, we plan to conduct studies on detecting and handling AEr, including the detection of AEr symptoms and the approaches and tools to support the detection at various granularity levels.

Based on:

Ruiyin Li, Peng Liang, Mohamed Soliman, Avgeriou Paris, (2021) *"Understanding Architecture Erosion: The Practitioners' Perceptive," in: Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC)*, Madrid, Spain, 2021, pp. 311-322: IEEE. DOI:10.1109/icpc52881.2021.00037

Chapter 3

Understanding Architecture Erosion: The Practitioners' Perceptive

Abstract

As software systems evolve, their architecture is meant to adapt accordingly by following the changes in requirements, the environment, and the implementation. However, in practice, the evolving system often deviates from the architecture, causing severe consequences to system maintenance and evolution. This phenomenon of architecture erosion has been studied extensively in research, but not yet been examined from the point of view of developers. In this exploratory study, we look into how developers perceive the notion of architecture erosion, its causes and consequences, as well as tools and practices to identify and control architecture erosion. To this end, we searched through several popular online developer communities for collecting data of discussions related to architecture erosion. Besides, we identified developers involved in these discussions and conducted a survey with 10 participants and held interviews with 4 participants. Our findings show that: (1) developers either focus on the structural manifestation of architecture erosion or on its effect on run-time qualities, maintenance and evolution; (2) alongside technical factors, architecture erosion is caused to a large extent by non-technical factors; (3) despite the lack of dedicated tools for detecting architecture erosion, developers usually identify erosion through a number of symptoms; and (4) there are effective measures that can help to alleviate the impact of architecture erosion.

3.1 Introduction

Ideally, during the lifespan of a software system, its software architecture is constantly modified to satisfy new requirements and accommodate changes in the environment; therefore, the evolution of the architecture is aligned with the evolution of the software system (Bass et al., 2021). However, with the increasing complexity

and changing requirements, the brittleness of a software system may increase, and the implementation may deviate from the architecture over time (Perry and Wolf, 1992). This divergence between the intended and the implemented architecture is often called *architecture erosion* (De Silva and Balasubramaniam, 2012).

Architecture Erosion (AEr) is not a new concept. Perry and Wolf (Perry and Wolf, 1992) explained around 30 years ago how a slowly eroding architecture makes it harder for new developers to understand the original system design. The phenomenon of AEr has been described using different terms in the literature, such as architectural decay (Le et al., 2016, 2018), architecture degeneration (Li and Long, 2011), architecture degradation (Macia, Arcoverde, Garcia, Chavez and von Staa, 2012), and design erosion (van Gurp and Bosch, 2002). AEr can significantly affect software development. Specifically, AEr can decrease software performance (Brosig et al., 2014), substantially increase evolutionary costs (Breivold and Crnkovic, 2010), and degrade software quality (Neto et al., 2018; De Silva and Balasubramaniam, 2012). Moreover, in an eroded architecture, code changes and refactorings could introduce new bugs and aggravate the brittleness of the system (Perry and Wolf, 1992). Given such negative consequences, it is critical to investigate how far practitioners are aware of it and how they perceive it in their daily work.

Although AEr has been studied in the literature, there is little knowledge about the current state of practice of AEr from the perspective of developers. Hence, we attempted in this work an in-depth exploration of the viewpoints of developers on the notion of AEr, the causes and consequences of AEr, the used practices and tools for detecting AEr, and the measures to control and prevent AEr. To find developers with knowledge and experience on AEr, we looked into online developer communities, similar to the recent studies (Tahir et al., 2018; Tian et al., 2019; Sinha et al., 2013). These communities cover a wide spectrum of topics on software development, where practitioners share their development experience, ask questions and get responses. Specifically, we collected data about AEr from developer discussions in six popular online developer communities. In addition, we also leveraged the communities to reach out to developers who took part in the discussions, and thus were aware of and experienced on AEr. We then collected their opinions by conducting a mini survey with 10 participants and holding interviews with 4 participants. Finally, we analyzed the collected data from all the data sources (i.e., posts, questions and answers, surveys, interviews) using Constant Comparison (Adolph et al., 2011).

The contributions of this work are threefold: (1) we explored how developers describe the phenomenon of AEr and how they perceive its manifestation; (2) we categorized and analyzed the diverse causes of AEr and the potential consequences to development from the developers' perspective; and (3) we investigated the effec-

tive tools and practices for detecting AEr, and presented the measures suggested by developers to control and prevent AEr.

This chapter is organized as follows: Section 3.2 introduces related work on AEr and online developer communities. Section 3.3 elaborates on the study design. The results of each research question are presented in Section 3.4, and their implications are discussed in Section 3.5. Section 3.6 examines the threats to validity, while Section 3.7 summarizes this work and outlines the directions for future research.

3.2 Related Work

3.2.1 Architecture Erosion

Several studies have explored the AEr phenomenon, which is described in various terms, such as "*architecture erosion*" (De Silva and Balasubramaniam, 2012), "*architecture decay*" (Mo et al., 2013), "*architecture degradation*" (Macia, Arcoverde, Garcia, Chavez and von Staa, 2012), "*software deterioration*" (Land, 2002), "*architectural degeneration*" (Hochstein and Lindvall, 2005), and "*software degradation*" (Bianchi et al., 2001).

Many studies focus on AEr with the purpose of identifying and repairing eroded architectures. For example, Wang *et al.* (Wang et al., 2019) proposed a multilevel method for detecting and repairing AEr based on architecture quality. Le *et al.* (Le et al., 2018) identified AEr by analyzing architectural smells and their relationships between reported issues. De Silva and Balasubramaniam (De Silva and Balasubramaniam, 2012) conducted a comprehensive survey on how to control AEr and the techniques for restoring and repairing eroded architectures, and they found that academic methods had limited adoption in industry. While the aforementioned studies proposed approaches for detecting and repairing AEr, to the best of our knowledge, there are no studies that investigate AEr from the perspective of developers. Therefore, our study differs from the existing studies in that it offers an examination of the phenomenon from the developers' perspective, including its concept, the causes and consequences, as well as the tools and approaches used in practice to detect and control AEr.

3.2.2 Online Developer Communities

There are millions of software practitioners who are active in online developer communities. They exchange knowledge, share experience, and provide an abundance of valuable discussions about software development. Such communities have been extensively utilized to conduct studies in the field of software engineering. For ex-

ample, Stack Overflow, which is the largest and most visited Q&A website, has been used as a source to efficiently identify architecturally relevant knowledge (Soliman et al., 2018), investigate the relationships between architecture patterns and quality attributes (Bi, Liang and Tang, 2018), examine users' behaviors and topic trends (Pal et al., 2012; Wang et al., 2013), study architecture smells (Tian et al., 2019), and explore anti-patterns and code smells (Tahir et al., 2018). In addition, other popular online developer communities are also used for research in software engineering, for example, understanding social and technical factors of contributions in GitHub (Tsay et al., 2014).

Considering the advantages of online developer communities (such as the vast amount of available information, fast responses to questions, and diverse solutions to engineering problems), we decided to use them as our data source. In addition to mining data from these communities, we went one step further and contacted developers from these communities to collect more in-depth data. This allowed us to target subjects that have experience in the topic of AEr, while also to use two more sources (survey and interviews) for the sake of triangulation (Runeson and Höst, 2009) (see Section 3.3).

3.3 Study Design

We formulated the goal of this study based on the Goal-Question-Metric approach (Basili et al., 1994) as follows: **analyze** *the perception of architecture erosion in practice* **for the purpose of** *understanding* **with respect to** *the notion, causes, consequences, detection and control of architecture erosion* **from the point of view of** *developers* **in the context of** *industrial software development*. In the following subsections, we explain the Research Questions (RQs), motivation, and the corresponding research steps and methods used to answer the RQs.

3.3.1 Research Questions

RQ1: How do developers describe architecture erosion in software development?
RQ1.1: Which terms do developers use to indicate architecture erosion?
RQ1.2: How does architecture erosion manifest according to developers?

Rationale: Researchers define AEr differently, such as architectural decay (Le et al., 2016, 2018), architecture degeneration (Li and Long, 2011), architecture degradation (Macia, Arcoverde, Garcia, Chavez and von Staa, 2012) (see Section 3.2.1).

Through this RQ, we investigate what terms and aspects are used by developers (practitioners) to discuss the phenomenon of AEr and how they describe its manifestation in practice. This can provide insights on how practitioners communicate this phenomenon and how they describe it according to their own experience.

RQ2: What are the causes and consequences of architecture erosion from the perspective of developers?

Rationale: There could be many factors leading to AEr, e.g., architecture smells (Le et al., 2018) and violations of architecture decisions (Zhang et al., 2011). Furthermore, an eroded architecture comes with severe consequences to development, such as slowing down development or hampering maintenance. Through this RQ, we intend to identify the potential causes and consequences of AEr in practice. This can help to confirm whether the causes and consequences that have been reported in the literature hold in practice, and whether new ones come to light.

RQ3: What practices and tools are used to detect architecture erosion in software development?

Rationale: Developers employ a number of practices and tools to identify potential architecture erosion in a system, or assess the quality of an eroded architecture. A list of such practices and tools used by practitioners can help other developers make informed decisions when dealing with AEr, and can inspire researchers to develop new approaches and tools.

RQ4: What measures are employed to control architecture erosion in software development?

Rationale: After AEr is detected, it needs to be controlled to prevent it from hurting the system. By answering this RQ, we want to identify the measures taken by developers for addressing AEr in practice and the effect of these measures. Being aware of these measures has a direct benefit to practitioners, as the measures can help to prolong the lifetime of systems, and save substantial maintenance effort (De Silva and Balasubramaniam, 2012). Moreover, categorizing the measures against AEr can provide insights to researchers for refining and extending existing approaches (e.g., Adersberger and Philippsen (2011)).

3.3.2 Research Process

To answer the RQs, we need to collect opinions, knowledge, and experiences about AEr from practitioners. Considering that many practitioners may not be familiar with the concept and terms of AEr, we need an efficient mechanism to find practitioners who have knowledge about AEr.

Online developer communities are regarded as an important platform for practitioners to communicate and share their knowledge and experience (Vassileva, 2008). For instance, Stack Overflow contains 20 million asked questions and 13 million users as of 21 January 2021. Hence, we decided to first collect data from the most popular developer communities and then contact the developers who were involved in the discussions about AEr. In detail, we followed five steps for data collection and analysis (see Figure 3.1), as elaborated in the following paragraphs.

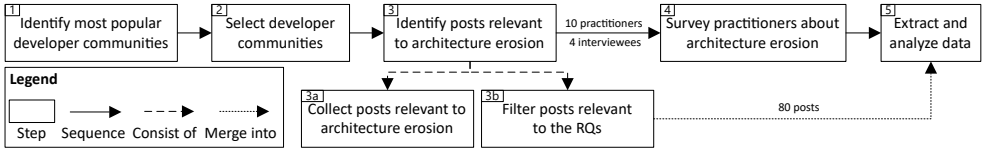

Figure 3.1: Overview of the research process

(1) Identify most popular developer communities

There is no dedicated communities/websites for discussing issues of software architecture. Therefore, we planned to collect data from developer communities, and we first identified the most popular developer communities. We conducted a web search using Google in two steps:

1. *Execute search queries*: We searched in Google using the search terms: "popular/ranking/top/best online developer communities/forums".

2. *Identify websites*: We counted the frequency of websites mentioned in the search results, and ranked them to obtain the most frequently mentioned websites (see Table 3.1). After the eighth most frequently mentioned websites, the rest had much lower frequencies so we considered the top eight. The complete list of websites and the corresponding frequencies are available online (Li, Soliman, Liang and Avgeriou, 2021b).

(2) Select developer communities

We conducted a pilot search on each of the eight most popular developer communities (see Step (1) and Table 3.1). For each developer community, we searched using the terms: "architecture erosion", "architecture degradation", "architecture

Table 3.1: Eight most popular online developer communities

#	Website	URL Link	S[1]	# R[2]	# A[3]
1	Stack Overflow	https://stackoverflow.com/	Yes	3973	39
2	Reddit	https://www.reddit.com/r/programming/	Yes	38	4
3	Dzone	https://dzone.com/	Yes	556	26
4	Hack News	https://news.ycombinator.com/	Yes	625	8
5	GitHub	https://github.com/	No	-	-
6	Stack Exchange	https://www.stackexchange.com/	No	-	-
7	Code Project	https://www.codeproject.com/	Yes	821	2
8	Sitepoint	https://www.sitepoint.com/community/	Yes	61	1
		Total	6	6074	80

[1-3] S = Selected communities, R = Number of retrieved posts, A = Number of analyzed posts

decay". We then checked a sample from the top returned results, and determined their relevance to the topic of "architectural erosion". Based on this pilot search, we selected six developer communities (see Table 3.1), and excluded two (GitHub and Stack Exchange). The justification of excluding GitHub and Stack Exchange is presented below, and we also discussed the threat of this process in Section 3.6.

- When conducting the pilot study on GitHub, there are a large number of search results from *Issues* and *Commits* on GitHub, but a very low percentage of the results relate to AEr (less than 0.1%). Therefore, the data extraction from GitHub would result in a very low return on investment.
- Stack Exchange is a Q&A website on topics in diverse fields, but not limited to software development. Stack Overflow is the largest sub-website of Stack Exchange and contains a number of discussion on software architecture (Soliman et al., 2016). Thus, to avoid duplicated search results, we excluded Stack Exchange and retained Stack Overflow.

(3) Identify posts relevant to architecture erosion

In this step, we collected posts from the selected developer communities (see Step (2) and Table 3.1), which discuss AEr and can provide answers to our RQs (see Section 3.3.1). Specifically, we performed two sub-steps: collecting (3a) and filtering posts (3b), as detailed below.

(3a) Collect posts relevant to architecture erosion

To identify the most suitable terms for capturing posts relevant to AEr, we experimented with several terms within the developer communities. After a pilot search

with several combinations of search terms, we decided to use the search query below: (additional details of the pilot search with other search terms are available online (Li, Soliman, Liang and Avgeriou, 2021b)).

((architecture OR architectural OR structure OR structural) AND (decay OR erode OR erosion OR degrade OR degradation OR deteriorate OR deterioration))

To execute the search query effectively, we used different search engines for each developer community (see Table 3.1), depending on the effectiveness of internal search engines offered by each developer community. For Stack Overflow, Sitepoint, and Hack News, we used their internal search engines. In contrast, Dzone, Reddit, and Code Project lack an effective search engine, and thus we used Google.

For each developer community, we executed the same search query, and removed duplicated search results. We executed the search manually for all the developer communities except for Stack Overflow, for which we built a crawler to automatically execute queries, collect and remove duplicated posts. We identified duplicated Stack Overflow posts by comparing the title of each post with other posts; using the title results in less duplicates, as posts may have different IDs but contain the same information (e.g., Stack Overflow posts with IDs "*36475408*" and "*36475800*"[1]).

(3b) Filter posts relevant to the RQs

We manually checked the collected posts to ensure their relevance to answering the RQs (see Section 3.3.1). To achieve this, we specified the following inclusion and exclusion criteria (Boolean AND is used to connect the inclusion criteria):

1) Inclusion criteria

- **I1**: The post (and its answers) discusses architecture erosion in software development.
- **I2**: The post (and its answers) is relevant to answering the RQs.

2) Exclusion criteria

- **E1**: When the content of two or more posts contain similar information, we exclude the one less relevant to the RQs.

Before the formal data filtering through manual inspection, to reach an agreement about the criteria, we conducted a pilot filtering step with 50 posts randomly-selected from the 3,973 retrieved posts on Stack Overflow, which were checked by two researchers independently. We then calculated the Cohen's Kappa coefficient (Cohen, 1960) of the pilot filtering results to measure the inter-rater agreement

[1]To access the post on Stack Overflow, add the ID number to the URL: https://stackoverflow.com/questions/ID

between the two researchers; this achieved an agreement of 0.728. Any disagreements and uncertain judgments on the posts were discussed between the two researchers until a consensus was reached. Then the author conducted the formal filtering with all the retrieved posts (see Table 3.1), and discussed with another researcher any uncertainties in order to get a consensus. Eventually, we collected in total 80 posts related to our RQs (see Table 3.1). Note that, 3,973 posts were originally retrieved from Stack Overflow, but we excluded a large number of irrelevant posts (e.g., topics about image erosion, array decay/degradation, learning rate decay).

(4) Survey practitioners about architecture erosion

While the data from community posts are valuable, we wanted to enrich them with other data sources; this helps also to achieve data source triangulation (Runeson and Höst, 2009). Thus we conducted a survey and interviews for collecting more data from another two sources (i.e., questionnaires and interviews). The prerequisite of conducting surveys and interviews is to find qualified participants, and collecting posts that discuss AEr (as explained in Step (3)) gives us the chance to identify the practitioners, who are potentially knowledgeable about AEr. These practitioners who were involved in the discussion of the collected posts can provide credible answers to the RQs.

To identify practitioners, we manually inspected the profiles of users, who were involved in the discussion of the 80 collected and filtered posts (from Step (3)). However, some users provided minimal information in their profile about their identity. This prevented us from identifying all potential practitioners. After inspecting all user profiles, we found 38 valid user profiles, from which we could discern their identity and contact information. We further searched for these 38 practitioners on LinkedIn, in order to validate their identity, and gather missing background and contact information. We note that the anonymity of these developers is preserved in both the manuscript and the accompanying dataset (Li, Soliman, Liang and Avgeriou, 2021b).

We contacted the 38 practitioners, and sent each practitioner two invitations. The first one was for answering the survey and the second invited them for participating in a one-to-one interview. 13 out of the 38 practitioners (34.2%) accepted our invitations for either filling out the survey (10 practitioners) or conducting a one-to-one interview (4 practitioners); note that one developer took part in both the survey and the interview. We got a response rate of 34.2%, which is much higher than the general response rate in empirical software engineering research around 5% (Singer et al., 2008). Table 3.2 provides the background information about the 13 practitioners.

Based on the selected posts and our RQs, we used similar questions for both

Table 3.2: Background of the participants

S/I*	EB[1]	YE[2]	YA[3]	YD[4]	Role	Location
S	PhD	44	28	-	Director	UK
S	MSc	15	-	10.3	Senior Consultant	Norway
S	BSc	25	-	25	Senior Software Engineer	UK
S	PhD	20	6.9	6.2	Senior Lead Developer	UK
S	MSc	20	2	14.8	Architect	Spain
S	PhD	54	15.8	6.3	Architecture Consultant	Denmark
S	MSc	16	-	15.3	DevOps Coach	Norway
I	MSc	27	16	-	CEO	USA
I	MSc	20	2.1	15	Lead Software Developer	USA
B	MSc	21	15	6	Software Intelligence Expert	USA
S	PhD	21	7.7	3.9	CEO	USA
S	PhD	16	-	16	Research Scientist	USA
I	MSc	8	2.8	5.2	Senior Software Engineer	India

* S = Survey, I = Interview, B = Both
[1-4] EB = educational background, YE = years of experience in software development, YA = years of experience as architect, YD = years of experience as developer

the customized surveys and interviews (available online (Li, Soliman, Liang and Avgeriou, 2021b)). The questions were designed and refined iteratively by four researchers. We decided to use open-ended questions to allow practitioners to express their own opinions; this is in line with our research goal to determine the practitioners' point of view on AEr. Before sending the survey to practitioners and conducting the interviews, we conducted a pilot survey with 2 developers who had the knowledge about AEr and helped to improve the clarity of the questions. Note that the pilot survey answers were not included in the data for analysis. The survey and interviews are detailed below.

Survey: We sent the questionnaire to the 38 practitioners, and 10 of them accepted our invitation to fill in the questionnaire. To make the questionnaire more relevant to the practitioners and improve the response rate, we sent each practitioner a customized version of the questionnaire (Li, Soliman, Liang and Avgeriou, 2021b), which was aligned with the post that he/she was involved by using the same term of AEr as the practitioner used to prevent misunderstanding. The survey responses were collected by Google Forms and saved in a Word document for later extraction and analysis (see Step (5)).

Interviews: We conducted semi-structured interviews by following the guidelines in software engineering (Hove and Anda, 2005) with 4 practitioners, who accepted our invitation. The interviews were conducted by another researcher (a postdoctoral fellow), and transcribed and analysed by two researchers (see Step (5)). For

Table 3.3: Mapping between the extracted data items and RQs

#	Data Item	Description	RQ
D1	Description of AEr	Terms used to describe AEr and how it manifests	RQ1
D2	Causes of AEr	Potential factors leading to AEr	RQ2
D3	Consequences of AEr	Impacts and ramifications of AEr on development	RQ2
D4	Practices and tools for detecting AEr	Approaches used in practice to identify AEr	RQ3
D5	Measures to control AEr	Ways to address or manage the impact of AEr	RQ4

the interviews, we used similar questions with the survey as a basis. As is usual with semi-structured interviews, additional questions came up during the interview depending on the answers to specific questions; in such cases, we allowed the practitioners to freely express their opinions.

(5) Data extraction and analysis

We used the data items (see Table 3.3) to collect data from three sources, i.e., online developer communities (80 posts), the survey (10 answers), and interviews (4 interviews). The author extracted the data and another researcher subsequently reviewed it. To alleviate personal bias, any uncertainties and ambiguities in the extraction results were discussed between two researchers to reach an agreement.

Regarding data analysis, we used descriptive statistics (for data item D1 and D4) and Constant Comparison (Adolph et al., 2011) (for data items D1-D5) to analyze and categorize the extracted qualitative data. Constant Comparison can be used to generate concepts, categories, and theories through a systematic analysis of the qualitative data. For the five data items, the author coded the data (around 400 annotations) using Constant Comparison and another researcher checked the coding results during the two coding activities (i.e., open coding and selective coding); any divergence in the coding and categorization results were further discussed until the two researchers reached an agreement. To effectively code and categorize data, we employed the MAXQDA tool[2] to help conduct manual data coding, which is a commercial tool for qualitative data analysis. All the data of this study is available online (Li, Soliman, Liang and Avgeriou, 2021b).

[2]MAXQDA, `https://www.maxqda.com/`

3.4 Results

3.4.1 RQ1 - Description of architecture erosion

To answer RQ1.1, we first used descriptive statistics to analyze the frequencies of the used terms. We then categorized the terms used to describe AEr into five types (see Table 3.4). We can see that most developers prefer to use *erode/erosion* to describe AEr. The term *degrade/degradation* comes second, followed closely by *decay*. Another popular term, though not as popular as the first three, is *deteriorate/deterioration*. In addition, we found several terms not included in our search terms are used to represent the AEr phenomenon when we manually checked the extracted data, such as *software rot*, *software entropy*, *software aging*, and *fundamental design flaw*. Note that, some developers used ambiguous or high level terms to describe AEr, like *system degradation*; but it is not clear whether a term like *system degradation* refers to performance degradation or structure degradation of the system. For this reason, we excluded such vague terms.

To answer RQ1.2, we analyzed the context of AEr mentioned by developers and categorized the manifestation of this phenomenon in four perspectives with Constant Comparison.

From the *structure* perspective: the structure of an eroded architecture deviates from the intended architecture. As mentioned by interviewee #1, "*Architecture (erosion) is about the original architecture blueprint got lost when you can't know architecture structure and boundaries anymore*". A number of striking examples were reported. Consider, for example, the violation of design rules about encapsulation that breaks implemented abstract layers and can have long-lasting impact on maintenance (in the interest of performance). A similar problem is the accumulation of cyclic dependencies and increased coupling, both resulting in binding elements together that were intentionally separated by architects. Finally, other mentioned structural manifestations include dead/overlapping code, and obsolete or incompatible third-party libraries.

From the *quality* perspective: an eroded architecture may not meet the original or current non-functional requirements; thus the system quality attributes are degraded. As developer #1 stated "*to make changes error prone, or to no longer meet cross-functional requirements (performance, security, scalability, etc.)*". Our data indicates a negative effect of AEr mostly on reliability (mainly due to increasing error-proneness), performance, and user experience.

From the *maintenance* perspective: an eroded architecture could be harder to understand, fix bugs, and refactor. As developer #2 stated "*it is often very hard to understand the existing architecture, determine the extent of architectural decay, and identify*

Table 3.4: Terms that developers used to describe architecture erosion

Type	Term	Count	
Erode /erosion	Architecture/architectural erode/erosion	20	26
	Structure/structural erosion	3	
	Software erosion	1	
	Project erosion	1	
	Component erosion	1	
Degrade /degradation	Architecture/architectural degrade/degradation	6	16
	Structure/structural degrade/degradation	2	
	Project/product degrade/degradation	3	
	Code degrade	1	
	Design degrade	1	
	Module degrade	1	
Decay	Architecture/architectural decay	8	15
	Project decay	2	
	Software decay	2	
	Design decay	1	
	Package structure decay	1	
	Structural decay	1	
Deteriorate /deterioration	Architecture/architectural deteriorate/deterioration	4	8
	Structure deterioration	2	
	Code quality deterioration	2	
Others	Software/design rot	5	9
	Software entropy	2	
	Software aging	1	
	Fundamental design flaw	1	

architectural smells and metric violations". Our data indicates that increasing complexity and technical debt often become common in eroded architectures, making bug-fixing and refactoring increasingly difficult.

From the *evolution* perspective: an eroded architecture makes it hard or even impossible to plan the next evolution steps, e.g., which features to implement next or which technologies to adopt. As developer #3 pointed out "*the unfortunate side effect of this [erosion] is that it becomes more and more difficult to add new visible features without breaking something*". Developer #3 also commented that, when an architecture erodes over time, "*the stakeholders will notice instability, high maintenance cost and ridiculously high cost for adding or changing features*".

3.4.2 RQ2 - Causes and consequences of architecture erosion

(1) Causes of architecture erosion

In Table 3.5, we list the potential causes leading to AEr, categorized into 12 types.

Inappropriate architecture changes are the most frequently mentioned reason incurring AEr, and often happen in the maintenance and evolution phases in a number of ways: introducing new anomalies (e.g., cyclic dependencies), breaking architectural rules, introducing new architectural principles that are incompatible with existing frequent modifications leading to the accumulation of cyclic dependencies. It is hard to accurately anticipate the side effects of those architectural changes, as developers cannot fully understand which part of the functionality will be implicated. The inappropriate changes to an architecture increase the risk of undermining the architectural integrity (e.g., breaking the encapsulation rules), introducing new bugs and causing architecture smells (e.g., increasing superfluous dependencies).

Unlike inappropriate architecture changes that happen during maintenance, **architecture design defects** often occur in the design phase. It could be regarded as a hidden danger to sustainability, as they often cannot be discovered via static analysis. These flaws in the system architecture, eventually lead to a gap between the intended architecture and the implementation. For example, if the original system has a "bad encapsulation", the implementation is likely to gradually deviate from the intended architecture as dependencies are created with the poorly encapsulated functionality over time. Developer #16 commented that, AEr happens as "*the inherent flaws present in every initial design begin to surface*". Developer #22 mentioned that "*if your core system abstractions are not clean, then the system is destined to degrade*".

Lack of management skills is another common cause of AEr and examples include: assigning incompetent developers to do a job they do not fit, having unreasonable rewarding and punishment metrics in place (that could lead to a high turnover of staff), lacking proper training and education for developers (resulting, for example, in non-uniform coding standards), or lacking long-term strategies for architecture evolution. As developer #15 stated "*some worse or mediocre developers who just are too uncomfortable with creative development and technical decision making are "promoted" to the management. As a result, everyone suffers, ..., it often creates disaster*".

The accumulation of **technical debt** is also a major cause leading to AEr. Technical debt (Li et al., 2015) is a short-term solution that may expedite development, but it might violate architectural principles, hampering refactoring and maintenance in general, and eventually deviating from the target architecture. As developer #4 stated "*the short term benefit of finishing your task now by taking short cuts versus the long term risk of making the code less understandable*".

Disconnection between architects and developers manifests through three scenarios: (1) architects do not adequately monitor the implementation process; (2) developers do not actively participate in the architecting process; or (3) there is complete absence of architect roles. Architects need to guide and monitor the implementation of architecture design and developers also need to understand the rationale

of architectural decisions. As mentioned by developer #5 "*the software changes need special attention (architectural assessment) from software architects. If this does not happen, the architecture could erode or become overly complex*".

Knowledge vaporization is mostly due to developer turnover, and poorly documented architectural knowledge. It is a potential threat to project management. For instance, knowledge is lost when developers leave the team, while new developers may violate architecture design principles due to the lack of knowledge about the current architecture.

Requirements changes challenge the architecture sustainability, e.g., when the architecture is incompatible with the newly-added or changed requirements, developers need to remove some parts from the architecture and add extra "patches"; this often impacts the maintainability and extensibility of the architecture. Note that, this kind of requirements are usually unforeseen and/or unplanned requirements (e.g., increasing demands for storage), that are in conflict with existing design rules and constraints.

Lack of communication has many obvious disadvantages to software development and particularly maintenance. For example, as mentioned by developer #6 "*if some developers isolate themselves from others, this may reduce the communication complexity at the cost of increasing the program complexity*"; consequently this increased complexity makes it harder to implement the intended architecture correctly. Interviewee #2 stated "*communication is key, for everyone on a team, including the architects and builders*".

In many cases, due to quick iterations and releases, developers might ignore long-term architectural strategies. Particularly, **agile development** is considered as a cause of rapid AEr. This is a common and recurring issue faced by agile development teams, as described by developer #7 "*No architecture will stay intact in the face of agile, evolving requirements*". Recent studies also show that agile process may not make a project agile (Sturtevant, 2017) and architecture might become the bottleneck of agile projects. Additionally, **increasing complexity** could slowly degrade the architecture, make codebases less understandable, and gradually make it harder and harder to maintain and evolve the system.

Lack of maintenance is regarded as another cause of AEr. If the architectural components are outdated and maintainers do not constantly refactor and maintain the codebase to keep it tidy and clean (e.g., replacing obsolete third party libraries), the architecture is destined to erode. Finally, there are four less frequently mentioned causes of AEr, including environment change, business process change, business pressure, and treating quality concerns as second-class citizens. For example, when a business process changes, the architecture might become incompatible with the new process (i.e., erosion).

Table 3.5: Causes of architecture erosion

No.	Cause	Type	Count
1	Inappropriate architecture changes	Technical	22
2	Architecture design defects	Technical	15
3	Lack of management skills	Non-technical	13
4	Technical debt	Technical	11
5	Disconnection between architects and developers	Non-technical	10
6	Knowledge vaporization	Non-technical	9
7	Requirements change	Both	9
8	Lack of communication	Non-technical	8
9	Agile development	Technical	8
10	Increasing complexity	Technical	7
11	Lack of maintenance	Both	6
12	Others (environment change, business process change, business pressure, quality concerns as 2nd-class)	Non-technical	9

Table 3.6: Consequences of architecture erosion

No.	Consequence	Count
1	Hard to understand and maintain	20
2	Run-time quality degradation	13
3	Enormous cost to refactor	11
4	Big ball of mud	9
5	Slowing down development	5
6	High turnover rate	3
7	Overall complexity	2

(2) Consequences of architecture erosion

The consequences of AEr are presented in Table 3.6. **Hard to understand and maintain** is the most frequently mentioned consequence. For example, an eroded architecture with implicit dependencies might be difficult to maintain without breaking some dependencies, and developers may not understand the ramification of breaking these dependencies. **Run-time quality degradation** is another major consequence of AEr. Users perceive a compromise in run-time qualities, such as performance, reliability, and user experience.

Eroded architectures may incur an **enormous cost to refactor**. As developer #8 said "*there is a risk that the refactoring cost is significant, possibly even too high to contemplate, resulting in a system is "stuck" in an undesirable form*". Furthermore, due to the increasing complexity of systems (e.g., cyclic dependencies), **slowing down development** can be common during the development and maintenance phase. For

example, time-to-delivery is delayed, while implementing new features and debugging can become extremely slow or even stagnant. In the worst case, an eroded architecture may render the system a **big ball of mud** (Foote and Yoder, 1997), i.e., the system lacks a perceivable architecture. As developer #9 stated "*As with all big ball of muds, the issue doesn't usually make itself apparent until there's some maintenance/enhancement needed*".

An eroded architecture can cause **high turnover rate**, as developers are forced to work on a messy architecture. Developer #10 explained how this affects an organization: "*it's not the software that rots but instead the users and/or organization that decays*". Developer #11 discussed the similarity between the software and the organization: "*Conway's Law would suggest that software architecture mirrors organization structure, so it too would become more brittle*". The high turnover further aggravates AEr, due to losing knowledge about system requirements and design decisions. Additionally, the **overall complexity** of the architecture can increase drastically. Developer #12 stated "*architectural erosion starts to happen as you add capabilities and slowly increase software complexity*". The accumulation of complexity, if left uncontrolled, bears the risk of bringing the entire project to a halt.

3.4.3 RQ3 - Identifying architecture erosion

(1) Tools

To answer RQ3, we collected the practices and tools employed to identify AEr. Table 3.7 lists the 13 tools collected and ranked according to their frequency mentioned by developers. We note that these tools practically identify issues in the architecture that can be considered as symptoms of AEr; in other words, none of the tools claims to specifically identify AEr per se. For example, Lattix (Kumar, 2016) is a commercial tool that allows users to create dependency models to identify architecture issues; NDepend (Kumar, 2016) can help users to find out architectural anomalies; Structure101 (Sangwan et al., 2008) offers views of code organization and helps practitioners to better understand the code structure and dependencies for checking architecture conformance. Some tools are language-specific, such as JDepend (Gopal, 2016) and Archie (for Java), Designite (for C#), while other tools support multiple programming languages.

(2) Practices

Apart from the tools mentioned by developers, we also found several general practices applied to identify AEr, as elaborated in the following paragraphs.

Dependency Structure Matrix (DSM) (Sangal et al., 2005) can be used to visually represent a system in the form of a square matrix. Developer #13 mentioned

Table 3.7: Tools used to detect architecture erosion

No.	Tool	Link	Count
1	Lattix	https://www.lattix.com	10
2	NDepend	https://www.ndepend.com	10
3	Sonargraph	http://www.hello2morrow.com/products/sonargraph	5
4	Structure101	https://structure101.com/products/workspace	4
5	ArchitectureQualityEvolution	https://github.com/tushartushar/ArchitectureQualityEvolution	4
6	JDepend	https://github.com/clarkware/jdepend	3
7	Designite	https://www.designite-tools.com/	3
8	SonarQube	https://www.sonarqube.org	2
9	SonarLint	https://www.sonarlint.org/	1
10	Archie	https://github.com/ArchieProject/Archie-Smart-IDE	1
11	Glasnostic	https://glasnostic.com/	1
12	CodeScene	https://codescene.io/	1
13	CAST	https://www.castsoftware.com/	1

"*DSMs are also a powerful way of setting and visualizing design rules. They make it easy to pinpoint violations to design rules*", which are typical symptoms of AEr. DSMs can visualize dependency relationships between packages (e.g., Mo et al. (2013); Nord et al. (2012)); understanding such dependencies helps detect AEr during the maintenance phase.

Software Composition Analysis (SCA) (Mokni et al., 2016, 2015) refers to the process that provides visibility of open source components in a system. As stated by developer #14, "*Managing a product against software decay can be a nightmare, but again a good SCA tool should be able to take care of that. It should be updated to the developers when a new open-source library becomes available*". Open source components are used in software products across all industries, and SCA can especially help to determine latent obsolete components that can typically cause AEr. There are also some SCA tools available, such as WhiteSource, Snyk, and Sonatype.

Architecture Conformance Checking (ACC) refers to the type of conformance between the implemented architecture and the intended architecture. The detection happens by identifying architectural violations in the implementation. As developer

#3 mentioned "*It is easier to pass a system from a development team to a maintenance team, especially when changes can automatically be checked for architectural conformance. Also new team members need less time to become productive because the code is easier to understand*".

Architecture monitoring refers to using tools to monitor the health of the architecture by detecting architecture issues. This is achieved by various techniques (e.g., Reflexion models (Murphy et al., 1995)) and metrics, such as coupling, size of files, hotspots (D'Ambros et al., 2008)). As developer #13 mentioned "*When design rules are monitored, tight scheduling does not erode the architecture and, if it does, the consequences of time pressure can be tracked (architectural technical debt) and monitored*".

Code review is a continuous and systematic process conducted by developers or architects to identify mistakes in code, such as violations of design patterns. As developer #1 mentioned "*when the team is small enough, using code reviews will be effective enough to prevent architectural erosion*".

Checking the change of architectural smell density is a method employed to detect AEr following the release timeline by statically comparing architectural smells in various versions. As developer #17 stated "*Comparing absolute values of detected architecture smell instances across versions is not a good idea because the size of the code is also changing. Therefore, we compute architectural smell density. It is a normalized metric capturing number of smells per one thousand LOC*". By observing the change rate of architectural smell density (Sharma et al., 2020), developers can find out from which version, the architecture started to deteriorate.

Architecture visualization (Shahin et al., 2014b) aims at representing architectural models and architectural design decisions using various visual notations. It helps to better understand the architecture and its evolution by visualizing the structure, metrics, and dependencies between architecture elements (e.g, components) in a project. Understanding architectural dependencies of a system through visualization is significant to detect the proliferation of violations (Murphy et al., 1995) and further erosion.

3.4.4 RQ4 - Addressing architecture erosion

To answer RQ4, we categorized the measures used to control AEr, as discussed in the following paragraphs.

Architecture assessment refers to the assessment process throughout the life cycle during architecture design (e.g., discovering and addressing the shortcomings of design decisions), during architecture implementation (e.g., monitoring and repairing possible violations of design rules), and during architecture evolution (e.g., choosing appropriate refactoring patterns to fix issues from new requirements or

evaluating the risk of architectural changes). We note that architecture assessment is meant to be followed up with concrete actions; in other words, it is a measure that connects detecting and addressing AEr, as developer #5 mentioned "*Architecture erosion can happen in any software project where the architectural assessments are not part of the development process*".

Periodic maintenance refers to regular activities (e.g., code refactoring, bug-fixing, testing) aimed at keeping a system "clean" and running smoothly. Developer #18 stated "*you can test the architecture regularly every time you make changes to the code. This eliminates the worry about architectural erosion in your software*". Another developer #19 urged "*don't leave unrepaired 'broken windows'. Fix each one as soon as it is discovered*" (examples of "broken windows" are bad designs, wrong decisions, or poor code). If there is insufficient time to conduct maintenance work right away, developers can create a list of pending problems and technical debt, and find a suitable time to pay off the debt and replace the temporary solutions.

Architecture simplification. When architectural complexity proliferates towards being uncontrollable, simplifying the architecture, and deliberately controlling the system size and complexity could be an option worthy of consideration. Developer #12 mentioned "*There needs to be a continuous effort to simplify (refactor) the code. If not, architectural erosion starts to happen as you add capabilities and slowly increase software complexity*". Several developers mentioned migrating to a microservices architecture, as one prominent way to achieve this. Decomposing the original monolithic architecture into many small microservices can, to some extent, improve architectural extensibility and increase its resilience to AEr.

Architecture restructuring is a drastic, yet effective means to control AEr. It is a much more pervasive change that concerns a large part of the architecture, compared to *architecture simplification*, that merely tries to reduce complexity. Developer #19 stated "*Sometimes, the best solution is simply to rewrite the application catering to the new requirements. But this is normally the worst case scenario. The cumbersome solution is to stop all new development, start by writing a set of tests and then redesign and rearchitect the whole solution*". However, restructuring the original architecture to satisfy new requirements and keeping the system running smoothly, may require enormous time and effort, considering the famous example of Mozilla web browser (Godfrey and Lee, 2000).

Organization optimization. Hiring more capable team members might also be an good option to address AEr. As developer #20 stated "*Consideration of people and organizational aspects of the architecture as well as technical aspects. Investment in people: training, study time, mentoring, etc*".

In addition, there are two less frequently mentioned measures: restarting a project or rewriting the architecture from scratch, and avoiding the systems growing

larger than intended by controlling the size and functional diversity deliberately.

3.5 Discussion

3.5.1 Interpretation of Results

RQ1: Terms and manifestation of architecture erosion. The results of RQ1.1 (see Table 3.4) show that most of the developers prefer to use the term "*erode/erosion*" to describe the AEr phenomenon, followed by "*decay*" and other less-frequently used terms. Regarding the results of RQ1.2, we found that developers usually describe the phenomenon of AEr from four perspectives: structure, quality, maintenance, and evolution. All the four perspectives are worth investigating with further research; while there are some literature linking each perspective with AEr (see e.g., Macia, Garcia, Popescu, Garcia, Medvidovic and von Staa (2012); Jaktman et al. (1999) for structure, Wang et al. (2019) for quality, and Brunet et al. (2012) for maintenance and evolution), this investigation needs to be more systematic.

RQ2: Causes and consequences of architecture erosion. We identified 15 causes of AEr (see Table 3.5). Inappropriate architecture changes is the most frequently mentioned reason that leads to AEr; this aligns with recent studies (e.g., Le et al. (2015)) on architectural changes. Table 3.5 reveals that, alongside technical factors, non-technical factors are not trivial. In fact, the non-technical aspects seem to reinforce each other; for example, "*lack of communication*" might induce the "*disconnection between architects and developers*" and "*knowledge vaporization*". The findings corroborate the results in (De Silva and Balasubramaniam, 2012), that good culture of communication and improving management skills are also important for architectural sustainability.

According to Table 3.6, the potential consequences from AEr mirror the four perspectives mentioned in RQ1, and extend them with further effects. Specifically, in addition to the impact on maintenance and evolution, as well as run-time qualities (that match the perspectives of quality, maintenance and evolution), we also found the impact of AEr on development speed, cost of refactoring, and high staff turnover of developers. The findings show the significance of preventing and controlling AEr, and warn that massive cost might be invested in the degraded projects for tackling AEr.

RQ3: Practices and tools for detecting architecture erosion. Developers often employ practices and tools to indirectly detect AEr by indicators or symptoms (e.g., cyclic dependencies, architecture violations). Such practices and tools contribute to the identification of architecture issues that are reported in the literature (e.g., Macia, Garcia, Popescu, Garcia, Medvidovic and von Staa (2012)) and have a well-

established connection to the phenomenon of AEr. Although they do not detect AEr per se, the findings suggest that there is a clear need for the software architecture community to devise dedicated tools on AEr detection.

Regarding the practices presented in Section 3.4.3, our findings suggest that a trend analysis from the quality and evolution perspectives (e.g., architecture monitoring, checking the change of architectural smells) may help the development team better understand the health status of systems. For example, Merkle (Merkle, 2010) found that keeping track of an evolving architecture (especially the changes and trends) can contribute to stopping AEr by using tools like Structure101.

RQ4: Measures taken for controlling architecture erosion. The results indicate that the measures can potentially help to alleviate the impact of AEr and prevent AEr during development. While there are few studies that validate such measures (e.g., Gerdes et al. (2016); Stal (2014)), further evidence is required to attest to their merit as well as potential pitfalls. This evidence is important for convincing management to allocate resources to control AEr, such as conducting architecture evaluation and periodic maintenance, or architecture simplification. Otherwise, tackling AEr may not get priority over the urgent implementation of features and bug fixing. Consequently, developers cannot ignore and do nothing about the appearance and accumulation of system anomalies, and regular inspection and maintenance are indispensable for preventing and tackling AEr. In general, these measures can provide practitioners further guidance regarding architecture management on how to deal with AEr during architecture maintenance.

3.5.2 Implications for Researchers and Practitioners

Terms of AEr: Regarding the terms used for describing the AEr phenomenon, although it is difficult to establish a unified term to be used universally, we do recommend that researchers should define the used terms when they refer to AEr, in order to minimize the ambiguities and misunderstandings. Additionally, practitioners are also encouraged to find common ground on understanding the AEr phenomenon for diminishing the ambiguity of the AEr concept and terms.

Four perspectives of AEr: It indicates that AEr manifests through structural issues, but mostly causes problems when it affects both run-time qualities (e.g., performance or reliability) and design-time qualities (e.g., maintainability and evolvability). Given AEr is a multifaceted phenomenon from the developers' perspective, and researchers can conduct more empirical studies to further investigate the characteristics of AEr. Additionally, the four perspectives of AEr should receive more attention from practitioners in their daily development activities, and these perspectives can be regarded as indicators for perceiving AEr.

Awareness of AEr: The findings can also raise awareness among practitioners about potential causes of AEr that are not intuitive, such as the adoption of agile development. We would thus urge practitioners to pay more attention to the grave consequences of AEr in order to request action at the management level. It is more likely that management will take measures or give a high priority to address AEr, when the aforementioned risks become explicit.

Guidance for software development: Having a good culture of communication and improving management skills (especially about the training and education of developers) are quite significant to developers for understanding the system design and structure. The findings can provide clues for practitioners to reduce the risk of AEr in their design and maintenance activities. Specifically, practitioners need to pay particular attention to the integrity of architecture when architecture changes happen.

Approaches and tools with empirical evidence: There is a need for researchers and practitioners to devise dedicated approaches and tools to detect and address AEr. Moreover, researchers can use the practices and tools reported in this study, to explore the scope, characteristics, and metrics of AEr, in order to provide a solid foundation on those tools. Furthermore, we encourage researchers and practitioners to explore the benefits and limitations by applying the approaches, tools, and measures for addressing AEr in practice.

3.6 Threats to Validity

The threats to the validity of this study are discussed by following the guidelines proposed by Wohlin *et al*. (Wohlin et al., 2012). Internal validity is not considered, since this study does not address any causal relationships between variables.

Construct validity concerns if the theoretical and conceptual constructs are correctly interpreted and measured. In this study, there are two key threats. The first one concerns the search process related to data collection of AEr. To mitigate this threat, we leveraged Google to collect recommended popular online developer communities as many as possible, and excluded duplicate results and searched platforms with qualifiers (e.g., popular "*web/Android/PHP*" communities, best developer communities "*in India*"). Besides, we reviewed papers of AEr and summarized the most frequently-used terms about AEr in the literature (see Section 3.2.1); these terms were used to derive our search string. The second threat lies in the process of manually extracting and analyzing the collected data. To partially mitigate this threat, we did a pilot execution of data filtering, extraction, and coding by the two researchers, for reaching an agreement about all the terms used.

External validity concerns the extent to which we can generalize the research

findings. A relevant threat concerns the representativeness of the selected online developer communities. To reduce this threat, we conducted a comprehensive analysis and selected several top-recommended online developer communities (see Section 3.3.2), including the largest and most widely used Q&A community by developers around the world (e.g., Stack Overflow) and other developer forums.

Reliability refers to the replicability of a study for generating the same or similar results. To alleviate this threat, we specified the process of our study design in a research protocol that can be used to replicate this work; Section 3.3 presents the details of the study design, while the complete information and all instruments and data are available on the replication package (Li, Soliman, Liang and Avgeriou, 2021b). Moreover, pilot studies of data collection, filtering, and survey were conducted to mitigate misinterpretations and biases. Before the formal data analysis, we did a pilot data filtering, extraction, and coding by two researchers. To eliminate personal biases, any conflicts and disagreements were discussed until an agreement was reached. Finally, we obtained a Cohen's Kappa coefficient of 0.728 on the filtering process, which partially reduces this threat.

3.7 Conclusions and Future Work

We conducted an empirical study to explore how developers perceive and discuss the phenomenon of AEr by collecting relevant information on AEr from the perspective of practitioners using three data sources (i.e., communities, surveys, and interviews). The findings can provide practitioners with concrete measures to detect and control AEr, and provide researchers with the challenges of AEr.

We found that most developers described the phenomenon of AEr with terms like "*erode/erosion*", "*decay*", and "*degrade/degradation*". When thinking about AEr, developers consider structural issues, but also the effect on run-time qualities, maintenance and evolution. Furthermore, besides the technical factors, non-technical factors play a big role in causing AEr. Despite the lack of dedicated tools for detecting AEr per se, developers employed associated practices and tools to detect the symptoms of AEr. To some extent, the practices and tools can help practitioners understand architectural structure and identify the eroding tendency of an architecture. Moreover, the identified measures can be employed during architecture implementation for effectively addressing AEr.

In the next step, we plan to detect and prevent AEr (semi-)automatically by establishing a dataset about symptoms of AEr and quantifying the degree of AEr from development artifacts (e.g., components).

Based on:

Ruiyin Li, Mohamed Soliman, Peng Liang, Paris Avgeriou, (2022) *"Symptoms of architecture erosion in code reviews: A study of two OpenStack projects," in: Proceedings of the 19th IEEE International Conference on Software Architecture (ICSA)*, Honolulu, Hawaii, USA, 2022, pp. 24-35: IEEE.
DOI:10.1109/ICSA53651.2022.00011

Chapter 4

Symptoms of architecture erosion in code reviews: A study of two OpenStack projects

Abstract

The phenomenon of architecture erosion can negatively impact the maintenance and evolution of software systems, and manifest in a variety of symptoms during software development. While erosion is often considered rather late, its symptoms can act as early warnings to software developers, if detected in time. In addition to static source code analysis, code reviews can be a source of detecting erosion symptoms and subsequently taking action. In this study, we investigate the erosion symptoms discussed in code reviews, as well as their trends, and the actions taken by developers. Specifically, we conducted an empirical study with the two most active Open Source Software (OSS) projects in the OpenStack community (i.e., Nova and Neutron). We manually checked 21,274 code review comments retrieved by keyword search and random selection, and identified 502 code review comments (from 472 discussion threads) that discuss erosion. Our findings show that (1) the proportion of erosion symptoms is rather low, yet notable in code reviews and the most frequently identified erosion symptoms are architectural violation, duplicate functionality, and cyclic dependency; (2) the declining trend of the identified erosion symptoms in the two OSS projects indicates that the architecture tends to stabilize over time; and (3) most code reviews that identify erosion symptoms have a positive impact on removing erosion symptoms, but a few symptoms still remain and are ignored by developers. The results suggest that (1) code review provides a practical way to reduce erosion symptoms; and (2) analyzing the trend of erosion symptoms can help get an insight about the erosion status of software systems, and subsequently avoid the potential risk of architecture erosion.

4.1 Introduction

Along evolution, software systems tend to exhibit a (growing) gap between the intended and the implemented architecture. This phenomenon is commonly referred to as *architecture erosion* (Perry and Wolf, 1992) and is often described using diverse terms (e.g., architectural decay and degradation) (Li, Liang, Soliman and Avgeriou, 2021b). Architecture erosion has a negative impact on the maintenance and evolution of a software system, by causing architectural inconsistencies (e.g., defective interfaces that hinder interaction with outside components) and degradation of software quality (e.g., components that are highly resistant to change) (Li, Liang, Soliman and Avgeriou, 2021b). However, architecture erosion is typically considered rather late in the process, and especially after it has severely impacted the system (Garcia et al., 2022); this makes it hard or sometimes even impossible to handle erosion. Therefore, it is wise to detect and repair architectural erosion as early as possible. One way to achieve that is by looking for *symptoms* of erosion in the early phases.

Researchers have identified a number of *architecture erosion symptoms*, which indicate the presence of the aforementioned gap between intended and implemented architecture (see Section 4.2.2). Such symptoms include common violations of design decisions and principles (e.g., violations of inter-module communication rules in a layered architecture (De Silva and Balasubramaniam, 2012)) and structural issues (e.g., architectural smells (Le et al., 2018; Fontana et al., 2016; Macia, Arcoverde, Garcia, Chavez and von Staa, 2012), rigidity and brittleness of systems (Martin, 2000; Ayyaz et al., 2015)). The occurrence of such symptoms can act as early warnings for software engineers to tackle architecture erosion (e.g., by refactoring) (Li, Liang, Soliman and Avgeriou, 2021b; Ali et al., 2018).

Previous studies have investigated approaches, mostly using static source code analysis (Fontana et al., 2016; Le et al., 2018), for identifying erosion symptoms (Li, Liang, Soliman and Avgeriou, 2021b; Fontana et al., 2016; Le et al., 2018; Macia, Arcoverde, Garcia, Chavez and von Staa, 2012). However, existing code analysis tools may not be sufficient to accurately identify the wide range of erosion symptoms (Lenhard et al., 2017; Azadi et al., 2019). For instance, existing tools (e.g., Sonargraph, Arcan) cannot identify certain types of architectural smells (e.g., ambiguous interface, multipath dependency) (Azadi et al., 2019). Moreover, several architectural smells manifest at levels that are out of scope of existing techniques and tools, such as package-level and service-level (including microservices) smells (Mumtaz et al., 2021).

While certain erosion symptoms are hard to detect using source code analysis tools, there are other sources that could be tapped for this purpose, such as code

reviews. Code review is a widely used practice to manually find defects in code and improve code quality (Bacchelli and Bird, 2013). Besides, code reviews can improve the quality attributes of the software (Morales et al., 2015) and also result in improvements at the architecture level (Paixao et al., 2019). Identifying erosion symptoms from code reviews can be complementary to using source code analysis tools; this would enhance the accuracy of detecting erosion symptoms. To the best of our knowledge, this data source for detecting erosion symptoms has not been investigated before. In fact, little is known about whether erosion symptoms are widely discussed (if at all) during code reviews and how developers deal with them.

In this study we aim at empirically investigating architecture erosion symptoms that are discussed during the code review process, as well as their evolution trend, and the actions taken by developers on dealing with the erosion symptoms. Identifying erosion symptoms through code review could support detecting architecture erosion early in software systems, that would otherwise not be captured by static analysis of source code. Regarding the scope, we focus on violation and structural symptoms of architecture erosion (see Section 4.2.2), as these two types are the most frequently discussed in the literature according to our recent study (Li, Liang, Soliman and Avgeriou, 2022) (see Section 4.8.2).

To achieve our goal, we manually identified 502 code review comments (contained in 472 discussion threads) related to the symptoms of architecture erosion. The key contributions of this work are summarized as follows:

- We constructed a benchmark dataset containing violation and structural symptoms of architecture erosion for further use by researchers (see the replication package (Li, Soliman, Liang and Avgeriou, 2021a)).

- We manually identified the most frequently discussed architecture erosion symptoms from code reviews and provided a taxonomy of the erosion symptoms.

- We further analyzed the trends of the erosion symptoms and the actions taken by developers in dealing with the symptoms.

The remainder of this chapter is organized as follows. Section 4.2 describes the background of this work. Section 4.3 elaborates on the research questions and the study design. Section 4.4 presents the results, which are subsequently discussed in Section 4.5. Section 4.6 details the implications of the study results for researchers and practitioners and Section 4.7 discusses the threats to the validity of the study. Section 4.8 introduces related work and Section 4.9 concludes with future directions.

4.2 Background

4.2.1 Code Review Process

Code review is the process of analyzing code submitted by developers to judge whether it is suitable to be integrated into the code base. It is meant as a lightweight practice of code inspection (Fagan, 1976) and is applied in both open source and industrial projects. Code review can be conducted in different ways; for example, OSS communities use not only informal code review processes through online communication (e.g., mailing lists or issue trackers), but also employ formal code review processes supported by tools (e.g., Gerrit[1]) (Beller et al., 2014).

Code review tools (e.g., Gerrit) are increasingly being adopted in both industrial and OSS projects (Sadowski et al., 2018; Bosu et al., 2015; Morales et al., 2015). Figure 4.1 shows an overview of the code review process that can be conducted iteratively. A developer creates code review requests and submits the code changes (e.g., patches, bug fixes) to the code review tools where static source code analysis can be automatically conducted for checking errors in code (e.g., compilation errors). After passing the automated analysis, the review requests are assigned to reviewers, which subsequently submit feedback to approve or reject the integration of the code changes into the code base. If the reviewers find defects in the submitted code, they will reject the code changes along with their feedback to the developers. The review process iterates until either the reviewers approve (status "*MERGED*") or reject (status "*ABANDONED*") the code changes.

4.2.2 Architecture Erosion Symptoms

Architecture erosion reflects the deviation of the implemented architecture from the intended architecture over time (Perry and Wolf, 1992), and manifests in a variety of symptoms during software development. A symptom is a partial sign or indicator of the emergence of architecture erosion. Structural and violation symptoms are the most widely discussed erosion symptom types in the literature (Li, Liang, Soliman and Avgeriou, 2022) (see Section 4.8.2). In this work, we focus on these two types of symptoms and collectively refer to them as **architecture erosion symptoms**.

Structural symptoms denote various structural problems in software architecture. For instance, Le *et al*. (Le et al., 2016) found that architectural smells can be regarded as structural symptoms affecting the sustainability of software systems. Herold *et al*. (Herold et al., 2015) reported that certain structural anti-patterns are

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

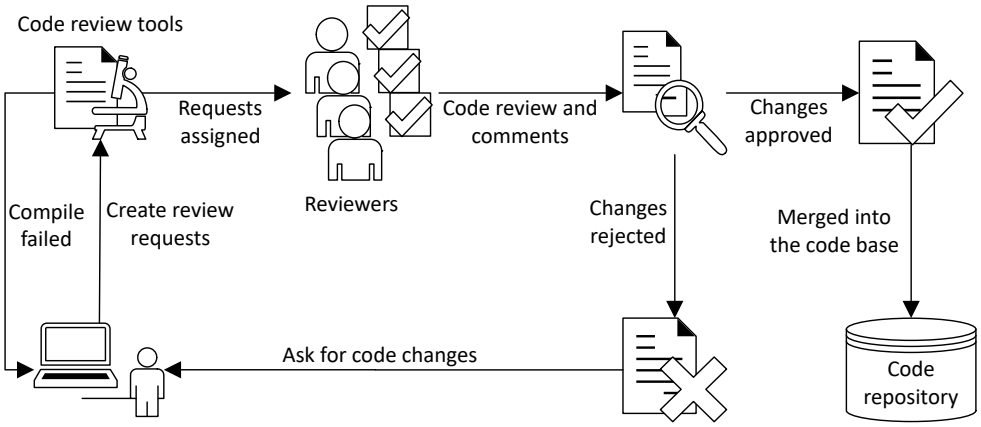

Figure 4.1: An overview of the code review process

symptoms of architecture erosion.

Violation symptoms refer to architectural violations during software development, such as violations of prescribed design decisions (e.g., layered pattern), principles (e.g., encapsulation), or constraints (e.g., uniform interface). For example, prior studies (Fontana et al., 2016; Macia, Arcoverde, Garcia, Chavez and von Staa, 2012; Mair et al., 2014) revealed that architectural violations (e.g., violations of intended design decisions), can be used to indicate the presence of architecture erosion.

4.3 Study Design

4.3.1 Research Questions

We aim at investigating the symptoms of architecture erosion that were identified in code reviews, and analyze the distribution and trend of the erosion symptoms, as well as how developers deal with those symptoms. To achieve this goal, we defined three Research Questions (RQs):

RQ1: Which symptoms of architecture erosion are frequently identified in code reviews?

Rationale: This RQ aims at identifying which types of erosion symptoms are frequently discussed during the code review process. Answering this RQ can help to understand erosion symptoms that occur in practice but might be ignored by code

analysis tools. The results can also help researchers and practitioners to devise tools and propose approaches for the automated identification of these frequently discussed symptoms.

RQ2: How do architecture erosion symptoms identified in code reviews evolve along time?

Rationale: This RQ aims at investigating how the discussion of architecture erosion symptoms changes along software evolution. That can help to understand to what extent architecture erosion accelerates or slows down. Answering this RQ can provide insights into the evolution of erosion symptoms throughout the code review process, and consequently contribute to understanding the sustainability and stability of architecture.

RQ3: Do the architecture erosion symptoms identified in code reviews get fixed in subsequent code changes?

Rationale: This RQ aims at investigating what actions developers might take after erosion symptoms are identified during the code review process, such as fixing the identified issues or ignoring them. Answering this RQ can help to reveal whether the identified erosion symptoms have an impact on code changes, and shed light on to what degree code reviews can result in removing the identified symptoms of architecture erosion and subsequently improving software quality.

4.3.2 Data Collection and Analysis

Figure 4.2 shows an overview of the data collection and analysis process. First, we employed a keyword-based approach to mine relevant comments that could potentially discuss erosion symptoms. Of course, the retrieved code review comments that contain the keywords, may not actually discuss erosion symptoms. Thus, we manually checked the retrieved code review comments to identify the discussions that contain erosion symptoms. In addition, we conducted a random selection of the review comments that do not contain any keywords as a supplement to the keyword-based approach. The whole process contains four steps as explained below. All the data and scripts are available at a replication package (Li, Soliman, Liang and Avgeriou, 2021a).

Step 1: Data Collection

The first step is to select software projects and collect code review data from

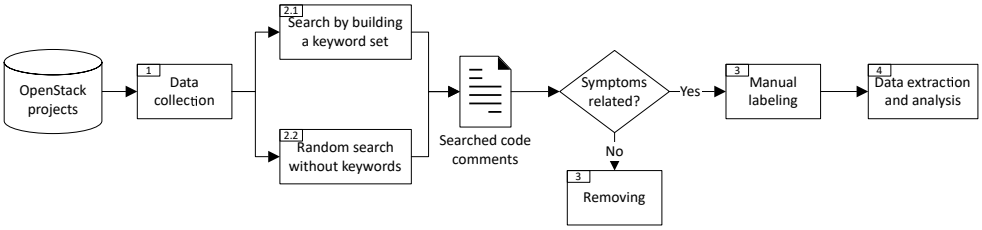

Figure 4.2: An overview of the data collection and analysis process

publicly available OSS repositories. OpenStack[2] is a widely-used open source cloud software platform, with which many organizations (e.g., IBM) collaboratively develop applications for cloud computing. The code review process in OpenStack is managed by the review tool Gerrit, and the code review data can be accessed through REST API[3]. We selected two of the projects of the OpenStack platform: Nova (a controller for providing cloud virtual servers) and Neutron (providing networking as a service for interface devices). These are the largest projects within OpenStack, and they have been actively developed over the last seven years with rich code review data. Moreover, the two projects have been used for understanding the detection of code-related quality issues (e.g., code smells (Han et al., 2021)). Thus, they might also contain the discussions on architecture erosion symptoms. We employed Python scripts to automatically mine code review data from the two projects between Jan 2014 and Dec 2018 (see Table 4.1). We organized the data in a structured way and stored them in MongoDB.

Table 4.1: An overview of the subject projects

Project	Domain	Language	#Code Changes	#Comments
Nova	Virtual server management	Python	22,762	156,882
Neutron	Network connectivity	Python	15,256	152,429

Step 2.1: Retrieve by Building a Keyword Set

While the selected projects in Table 4.1 have thousands of comments, a large number of review comments might not be related to architectural issues (Paixao et al., 2019) and it is inefficient to manually check each review comment. Therefore, we decided to search for the discussions on erosion symptoms through associated keywords. We first needed to determine relevant keywords that commonly occur in the discussions on erosion symptoms. In a recent empirical study on architecture

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

[3] https://gerrit-review.googlesource.com/Documentation/rest-api.html

erosion (Li, Liang, Soliman and Avgeriou, 2021b), Li *et al*. found that developers prefer to use certain common terms (e.g., erosion, decay, degradation) to describe the architecture erosion phenomenon in Q&A websites. Therefore, we conducted a trial search by using the terms identified in (Li, Liang, Soliman and Avgeriou, 2021b) (e.g., decay, erode, erosion, degrade, degradation, deteriorate, deterioration). However, we found that these description terms of architecture erosion were rarely used in code reviews. A possible explanation is that reviewers, during the code review process: a) focus more on specific code changes rather than general architectural concepts, and b) use terms related to specific architectural issues (e.g., a specific constraint violation).

To effectively locate the specific types of erosion symptoms in code reviews, we needed to establish a keyword set related to violation and structural symptoms of architecture erosion (see Section 4.2.2). In Table 4.2, we selected common violation and structural symptoms according to previous studies (Azadi et al., 2019; Le et al., 2016; Oizumi et al., 2019; Ganesh et al., 2013), and formulated an initial keyword set. For this set, we preferred to be inclusive rather than exclusive and thus included all terms mentioned in the aforementioned studies.

Considering that the effectiveness of keyword-based text mining techniques highly relies on the set of keywords, we worked towards refining this initial set of keywords. Specifically, we followed the iterative approach proposed by Bosu *et al*. (Bosu et al., 2014); similarly to our goal, Han *et al*. had established a set of keywords in order to mine code smell discussions in code reviews (Han et al., 2021). In this study, we mined code reviews and change logs of code changes (that contain status information "*MERGED*" and "*ABANDONED*") with the purpose of identifying relevant discussions on architecture erosion symptoms. The approach includes the following steps:

1. We prepared an initial set of keywords as mentioned above.

2. We searched using the initial keyword set in the collected review comments and built a corpus by collecting the relevant comments that encompass at least one keyword from the initial set of keywords (e.g., "violation", "inconsistent").

3. We processed the retrieved code review comments which contain at least one keyword in our keyword set and removed English stopwords, punctuation, and numbers. That results in a list of tokens.

4. We conducted a stemming process (using SnowballStemmer from the NLTK toolkit (Bird et al., 2010)) to obtain the stem of each token (e.g., "architecture" and "architectural" have the same token "architectur").

5. We built a document-term matrix (Tan et al., 2016) from the corpus, and found the additional words that co-occur frequently with each of our initial keywords (co-occurrence probability of 0.05 in the same document).

6. We manually checked the list of frequently co-occurring additional words to determine whether the newly discovered words should be added into the keyword set.

After performing the aforementioned steps, we found that no additional keywords co-occurred with the initial keywords based on the co-occurrence probability of 0.05 in the same document; this is similar to previous studies (Han et al., 2021) and (Bosu et al., 2014). Thus, we believe that the initial keyword set is sufficient to mine the violation and structural symptoms of architecture erosion from code review comments. In total, 20,211 code review comments were retrieved from the two projects.

Table 4.2: Symptoms of architecture erosion used in this study

Violation symptom	**Keywords**
Architecture violation	architecture, architectural, layer, design, violate, violation, deviate, deviation
Architecture inconsistency	inconsistency, inconsistent, consistent, mismatch, diverge, divergence, divergent
Constraint violation	rule, constraint, violate, violation
Structural symptom	**Keywords**
General terms of architectural smells	architecture, architectural, structure, structural, smell, antipattern, anti pattern, anti-pattern, defect
Cyclic dependency	cycle, cyclic, circular, dependence, dependency
Unnecessary dependency	unnecessary, dependency
Obsolete functionality	obsolete, unused
Ambiguous interface	ambiguous, interface
Unused interface and unused brick	unused, interface, brick
Sloppy delegation	sloppy, delegation
Brick functionality overload	brick, overload
Duplicate functionality	duplicated, clone, copypasted, redundant, copied, overload
Scattered functionality	scattered

Step 2.2: Random Selection of Review Comments without Keywords

It is possible that reviewers do not use the keywords in Table 4.2 when they described erosion symptoms during the code review process. Thus, we conducted a supplementary search by randomly selecting review comments that do not contain

any keywords in Table 4.2 from the rest of the review comments of the two OpenStack projects. We randomly selected 1,063 review comments out of the remaining 289,100 review comments (i.e., confidence level of 95% and margin of error of 3% (Israel, 1992)). Then, we manually checked the code review comments and retrieved 7 review comments related to erosion symptoms.

Step 3: Data Filtering and Labeling

The search process retrieved a large number of code review comments through keywords, but these still may largely contain irrelevant results. In other words, there are review comments that are not related to discussions on erosion symptoms but contain keywords, such as code snippets, links, and variable names with keywords (e.g., "`attr_interface`"). The irrelevant results were removed manually along with preprocessing by scripts in the replication package (Li, Soliman, Liang and Avgeriou, 2021a). To ensure we have the same understanding of the erosion symptoms when performing the manual removal, we conducted a pilot data filtering and labeling; for that, we used 50 comments randomly-selected from the retrieved comments, which were checked by two researchers, independently. To measure the inter-rater agreement between the two researchers, we calculated the Cohen's Kappa coefficient (Cohen, 1960) of the pilot and got an agreement of 0.898. During the formal data filtering and labeling process, the author conducted the data filtering and labeling, and another researcher reviewed and checked the results. Any disagreements on the comments were discussed between the two researchers until a consensus was reached. In total, we collected 502 code review comments (from 472 discussion threads) that contain discussions on erosion symptoms from the two OpenStack projects. Note that, a discussion thread can contain more than one code review comments that pertain to the same symptom.

Step 4: Data Extraction and Analysis

The data items in Table 4.3 are used to extract relevant data from the two OpenStack projects for answering the RQs. The author extracted the data which was

Table 4.3: Mapping between the extracted data items and RQs

#	Data item	Description	RQ
D1	Comment	The comments from reviewers on source code	RQ1
D2	Code review URL	The URL link of the code review comments and changes	RQ1
D3	Change ID	The unique id of each code change	RQ1
D4	Comment timestamp	The timestamp of the code review comments	RQ2
D5	Code change status	The status of each code change in the change logs	RQ3

subsequently reviewed by another researcher. To mitigate personal bias, any disagreements were discussed between two researchers to reach a consensus. The author then rechecked the extraction results of all the code reviews to ensure the correctness of the extracted data. Regarding the data analysis, we used descriptive statistics to analyze the extracted comments and refined the existing classifications in the literature.

4.4 Results

4.4.1 Results of RQ1

In total, we identified 502 code review comments (contained in 472 discussion threads) related to architecture erosion symptoms. As mentioned in Section 4.3.2, each discussion thread reflects one symptom but may contain more than one comment. Figure 4.3 shows the distribution of the identified erosion symptoms in the code reviews from Nova and Neutron. *Architectural violation* is the most frequently identified symptom with 75 (15.9%) discussion threads; this symptom entails that the reviewers perceived the implemented architecture to be violating the intended architecture during the development process. *Duplicate functionality* (Le et al., 2016) is a common architectural smell that comes second with 71 (15.5%) discussion threads, followed by *cyclic dependency* with 56 (11.9%) discussion threads. In addition, 39 (8.3%) discussion threads belong to *obsolete functionality*.

Considering the two categories of symptoms, i.e., violation symptoms and structural symptoms, the former amounts to 18.0% and comprises two categories: *architectural violation* (75 discussion threads) and *architectural inconsistency* (10 discussion threads). *Architectural violation* entails the violation of architectural constraints (e.g., layered pattern) or design rules (e.g., encapsulation). For example, one reviewer mentioned that "*this feels like a layer violation to be re-using the privsep stuff from os-brick here*"[4]. *Architectural inconsistencies* reflect a mismatch between the implemented and the intended architecture (e.g., inconsistent dependency). For example, one developer responded: "*ok I will remove that, there is inconsistent between agent implementations*"[5].

Compared to violation symptoms, structural symptoms (82.0%) are much more frequently discussed in code reviews. In the following, we refined the existing classification (e.g., Azadi et al. (2019); Le et al. (2016)) and provide a taxonomy of the most frequently discussed structural symptoms. Due to space limitations, we list

[4] https://review.opendev.org/c/openstack/nova/+/312488
[5] https://review.opendev.org/c/openstack/neutron/+/195439

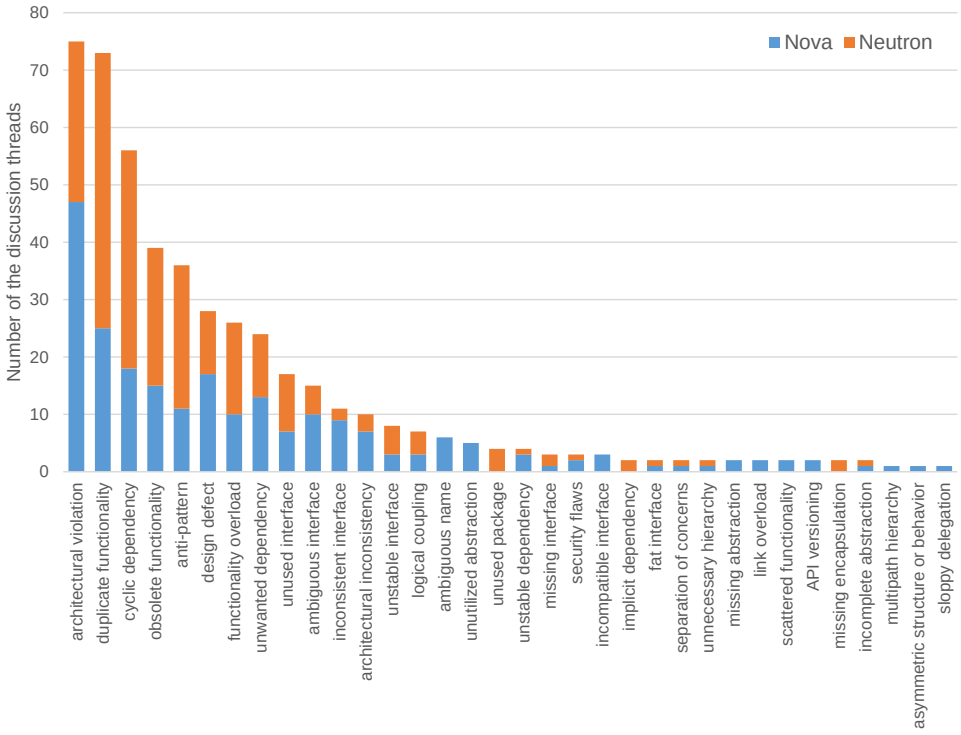

Figure 4.3: Number of the discussion threads on architecture erosion symptoms in Nova and Neutron

the major structural symptoms in each category, while the rest can be found in the replication package (Li, Soliman, Liang and Avgeriou, 2021a).

1. **Functionality-related smells**: Four types of structural symptoms contained in 140 discussion threads are related to functionality, namely *duplicate functionality*, *obsolete functionality*, *functionality overload*, and *scattered functionality*.

    - *Duplicate functionality* (73 discussion threads) concerns the replication of the same functionality among components. Ignoring the other components when altering the functionality of one component might cause architectural problems. For example, one reviewer commented: *"this is duplicated in the other module - we could pull this into the common module as a helper method"*[6].

[6] https://review.opendev.org/c/openstack/nova/+/229964

- *Obsolete functionality* (39 discussion threads) refers to unused and invalid functionality that can increase system complexity. For example, one reviewer stated "*oh, this module is not necessary at all now because we have already removed legacy v2 API code*"[7].
- *Functionality overload* (26 discussion threads) occurs when a component undertakes an excessive amount of functionality. For example, as described by one reviewer "*since this module is overloaded by tons of code, it makes sense for me to get it rid of all responsibilities that we can do*"[8].

2. **Dependency-related smells**: We identified five types of structural symptoms contained in 91 discussion threads related to dependency issues, namely *cyclic dependency*, *unwanted dependency*, *unstable dependency*, *implicit dependency*, *link overload*, *unnecessary hierarchy*, and *multipath hierarchy*.

   - *Cyclic dependency* (56 discussion threads) occurs when two or more components directly or indirectly interact with each other to form a circular chain. For example, as one reviewer pointed out "*the removal here is a good example how the restructuring resolves the cyclic dependency issue*"[9].
   - *Unwanted dependency* (24 discussion threads) denotes the dependencies that are obsolete, duplicate, or unnecessary between components. As one reviewer stated "*I think it's a wrong dependency, [...]*"[10].

3. **Interface-related smells**: We identified nine types of structural symptoms contained in 69 discussion threads related to interfaces, namely *unused interface*, *ambiguous interface*, *inconsistent interface*, *unstable interface*, *logical coupling*, *missing interface*, *incompatible interface*, *fat interface*, *API versioning*, and *sloppy delegation*.

   - *Unused interface* (17 discussion threads) occurs when an interface is not utilized. For example, one reviewer suggested "*this is a weird interface. MonitorBase doesn't seem to be used very much yet, is this a good opportunity to remove this abstract method and replace it with get_metrics?*"[11].
   - *Ambiguous interface* (15 discussion threads) appears when an interface has an unclear definition or provides a single or general entry-point (e.g., single parameter) to other components or connectors. For example, one re-

[7] https://review.opendev.org/c/openstack/nova/+/292473
[8] https://review.opendev.org/c/openstack/nova/+/282580
[9] https://review.opendev.org/c/openstack/nova/+/250907
[10] https://review.opendev.org/c/openstack/neutron/+/87841
[11] https://review.opendev.org/c/openstack/nova/+/219153

viewer stated "*I won't hold up on it, but this is a bit of a confusing interface to me, so I think documenting it thoroughly is important*"[12].

4. **General symptoms** refer to general descriptions of the structural symptoms, including *anti-patterns* (36 discussion threads) and *design defects* (28 discussion threads). For example, one reviewer stated "*This is an antipattern in concurrent programming and we should not do this.*"[13].

Finding 1: The proportion of architectural erosion symptoms is rather low in code reviews. The most frequently identified symptoms of architectural erosion are *architectural violation*, *duplicate functionality*, and *cyclic dependency*.

4.4.2 Results of RQ2

To answer RQ2 and gain a first insight into the architectural erosion trends of the two OpenStack projects (i.e., Nova and Neutron), we plotted line charts of the discussion threads on erosion symptoms in the two projects between 2014 and 2018.

As we can see from Figure 4.4, the numbers of discussion threads on erosion symptoms show a decreasing trend for both Nova and Neutron, as the two projects evolve. For Nova, we can see that the number of erosion symptoms shows a steady downward trend from 2014 to 2018. As for Neutron, the number of erosion symptoms fluctuates and reaches a peak in 2015, and then it follows a decreasing trend in the next three years. One possible reason for the drop in the numbers of erosion symptoms in both projects could be a potential decrease in the numbers of review comments. To check this, we looked at the percentage of erosion symptoms in review comments (see Figure 4.5). Although the numbers of review comments fluctuate over time, we found that the percentages of erosion symptoms have a declining tendency through a linear regression analysis. Thus, it is reasonable to argue that the architecture of the two OpenStack projects becomes stable over time, as they exhibit fewer erosion symptoms.

To perform a more detailed analysis, we analyzed the distribution of the numbers of discussion threads on erosion symptoms in the two projects over a period of 60 months between 2014 and 2018 (see Figure 4.6). We can see a fluctuating but downward trend for both Nova and Neutron. During the manual check process, we found several reviews that discussed issues concerning both Nova and Neutron, such as connections through interfaces. For example, one reviewer mentioned: "*I'm still concerned that Nova is not interacting with Neutron through a well defined interface*

[12] https://review.opendev.org/c/openstack/nova/+/427902
[13] https://review.opendev.org/c/openstack/nova/+/77995

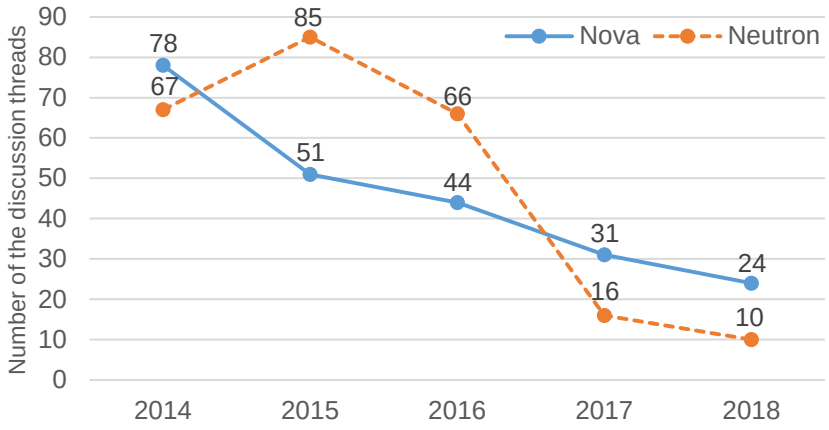

Figure 4.4: Trends of the discussion threads on architecture erosion symptoms in Nova and Neutron

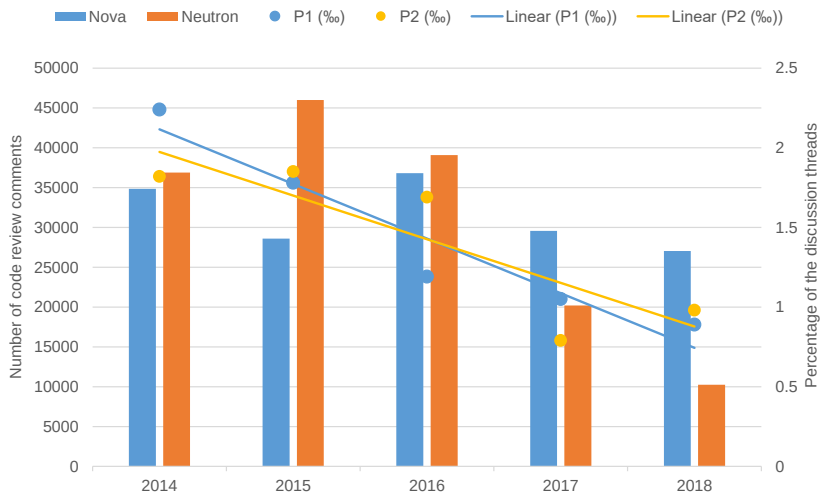

Figure 4.5: Percentages of the discussion threads on architecture erosion symptoms in Nova (i.e., P1) and Neutron (i.e., P2)

here"[14]. Interestingly, the two projects have largely similar fluctuating trends of the number of erosion symptoms (see Figure 4.6) along time, since both projects belong to the OpenStack platform, and their erosion symptoms to some extent follow a similar trend.

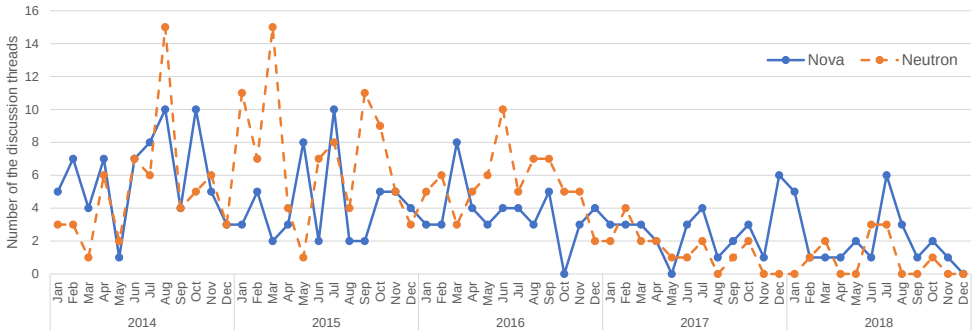

Figure 4.6: Monthly distribution of the discussion threads on architecture erosion symptoms per month in Nova and Neutron

To statistically verify this trend, we conducted a correlation analysis, using as null hypothesis (H_0) that *there is no linear relationship between the number of architecture erosion symptoms in Nova and Neutron*. We chose a p-value 0.05 as a statistical significance threshold, which is commonly used in empirical studies (Shull et al., 2007). We used Spearman's rank correlation (Zar, 1972) to assess the correlation between the monthly distribution of erosion symptoms in Nova and Neutron. The reason is that Spearman's rank correlation does not have any requirements on the normality of data distribution (Pagano, 2012). We adopted the classification of correlation coefficient by Marcus and Poshyvanyk (Marcus and Poshyvanyk, 2005), where a value belonging to [0.3-0.5] denotes a moderate correlation. We calculated a Spearman's rank correlation value of 0.412 (p-value is 0.0007), which indicates a statistically significant positive correlation (p-value $\ll$ 0.01); the correlation of 0.412 is moderate strong. Therefore, we rejected the null hypothesis (H_0) and accepted the alternative hypothesis (H_1), namely, *there is a linear relationship between the number of architecture erosion symptoms in Nova and Neutron*.

[14] `https://review.opendev.org/c/openstack/nova/+/275073`

Finding 2: Both the numbers and percentages of the architecture erosion symptoms in Nova and Neutron show declining tendencies, which suggests that the architecture of the two projects tends to become stable over time. To some extent, the architecture erosion symptoms in Nova and Neutron projects are correlated to each other.

4.4.3 Results of RQ3

During code review, the review comments can provide suggestions on how to remove architecture erosion symptoms. For example, Figure 4.7 shows a code snippet with a cyclic dependency before the review (left side) and after the dependency was removed (right side). Although the final versions of the submitted code changes can be detected by the status of code changes (status "*MERGED*" or "*ABANDONED*"), whether the erosion symptoms get fixed through code changes is not clear. Thus, to gain more insights from the code changes along with their discussions and status, we further investigated and presented the status of code changes in Table 4.4, including 361 "*MERGED*" and 111 "*ABANDONED*" code changes.

Figure 4.7: An example of cyclic dependency remove operation

Table 4.4: Status of the code changes

Status	Count	Sum
Merged (fixed)	309	
Merged (no response)	49	361
Merged (no response but fixed)	3	
Abandoned (verification failed)	79	
Abandoned (verification passed but voted to reject)	30	111
Abandoned (verification passed but no vote)	2	

"*MERGED*" status denotes that code changes pass the verification (i.e., automated testing) and receive the approval from the reviewers to be integrated into the main repository. The results show that most of the "*MERGED*" code changes (85.6%,

309 out of 361) reached an agreement (fixed) after discussions between the reviewers and developers. For example, a reviewer mentioned "*This is a layering violation. This should be:* `old_flavor = instance.get_flavor('old')`", which was responded by a developer "*Done*"[15]. This example shows that a reviewer pointed out a violation symptom (i.e., layering violation) that needed to be fixed. In addition, we noted that 13.6% (49 out of 361) of the "*MERGED*" code changes did not get any response from the developers or reviewers. In other words, while the identified erosion symptoms can be regarded as potential problems, they still remain in the code and are ignored by both developers and reviewers.

Regarding the code changes marked as "*ABANDONED*", they either failed their verification, or were voted to be rejected by the reviewers, or both. The results show that most of the "*ABANDONED*" code changes (71.2%, 79 out of 111) failed their verification, while 27.0% of the code changes were rejected by the reviewers after passing the automated testing. Such rejections are mostly due to code changes having bad quality or logical errors; for instance, one reviewer mentioned an issue "*no, it is not cleaner, it is a design mistake [...]*", and the developer agreed to resubmit a code change "*ok, I agree. I will make an update*"[16]. Another common reason for rejecting code changes that passed their verification, is a difference of opinion between developers and reviewers, which leads to reaching an agreement (upon discussion) to re-submit a code change. Moreover, we found that nearly half (45.0%, 50 out of 111) of the "*ABANDONED*" code changes did not receive any response from the developers and no actions were taken to cope with the identified erosion symptoms.

In summary, the results show that code reviews can have a significant and positive impact on removing erosion symptoms: they were either *fixed and merged* or *abandoned* (thus also removing the erosion symptoms). Most (89.6%, 423 out of 472) of the erosion symptoms were addressed and only 10.4% (49 out of 472) of the discussion threads that contain erosion symptoms were ignored by the developers and reviewers.

Finding 3: Most code reviews that identify erosion symptoms have a positive impact on removing the architecture erosion symptoms, but a few erosion symptoms remain and are ignored by the developers and reviewers (i.e., they persisted in the "*MERGED*" code changes). The main reason behind "*ABANDONED*" code changes is failed verification (i.e., failed automated test).

[15] https://review.opendev.org/c/121409
[16] https://review.opendev.org/c/openstack/nova/+/120309

4.5 Discussion

4.5.1 RQ1: Frequently Identified Erosion Symptoms

Our results indicate that the proportion of the review discussions on architecture erosion symptoms is rather low in code reviews (i.e., 2.4%, 502 out of 21,274 review comments). But they do exist and can be very useful to mine, as they provide an indication of the system's architectural erosion overall. Compared to previous studies, particularly (Azadi et al., 2019; Le et al., 2016) that only focus on architectural smells and their classification, we also focused on another type of erosion symptoms, that is, architectural violations. Besides, we refined the classification and provided a taxonomy that is comparatively more comprehensive and covers two common types of erosion symptoms (see Section 4.4.1). According to the results of RQ1, *architectural violation*, *duplicate functionality*, and *cyclic dependency* are the most frequently identified erosion symptoms. Regarding the structural symptoms of architecture erosion, functionality-related smells (e.g., duplicate functionality) attract more attention from code reviewers, followed by dependency-related and interface-related smells.

We observed that a large number of code review comments concern code-level problems instead of architecture-level problems. In other words, compared to the discussions on code-level issues (e.g., code smells (Han et al., 2021)), the percentage of the discussions on architecture erosion symptoms is lower. One obvious reason is that code reviewers usually focus on the submitted code snippets and seldom discuss architectural problems, since they may not really be familiar with the system architecture or they may lack architectural awareness (Paixao et al., 2019). Interestingly, many of the discussion threads that contain erosion symptoms are described as bug fixing during the code review process; this may be because code reviewers are not aware that the "bugs" may undermine the system architecture and make the architectural deviate from the intended architecture. This conjecture corroborates recent findings by Paixao *et al*. (Paixao et al., 2019) that the majority of "fixed bugs" affecting the system's behaviour is more than simple bugs throwing an exception. In other words, the code reviews related to architectural changes may be described as code-related changes (e.g., bugs) due to the lack of architectural awareness.

4.5.2 RQ2: Trend of Identified Erosion Symptoms

The number of architecture erosion symptoms in the two OpenStack projects (i.e., Nova and Neutron) shows a declining tendency. This indicates that fewer erosion

symptoms were discussed during the code review process; as the two projects get mature and stable over time, so does their architecture. This is similar to the findings of Bi *et al*. (Bi et al., 2021), who established that architectural discussions and changes decrease after several stable releases. One possible reason is that the architecture of the two projects tends to become stable after several major versions were released. As mentioned in Lehman's Law of software evolution, "*the quality of a system will appear to be declining during its evolution, unless proactive measures are taken*" (Lehman and Ramil, 2002). The declining trend of the erosion symptoms in the two projects implies that positive architectural evolution might take place to improve the structural quality (Uchôa et al., 2020) and make the architecture more stable over time.

We also observed that the number of the review comments on erosion symptoms in the two OpenStack projects shows a similar changing trend and are correlated to each other. The two projects (i.e., Nova and Neutron) are important components of the OpenStack cloud computing platform; this entails that there may be many interactions (e.g., couplings of structure and data) between the projects in the same community (e.g., OpenStack in this study), as reflected in the (almost) common evolution of their erosion symptoms. Moreover, the finding indicates that when developers change certain components, they should be aware of the potential couplings among the projects in the same platform.

4.5.3 RQ3: Impact of Identified Erosion Symptoms

The results of RQ3 show that most of the identified erosion symptoms (89.6%) were addressed through either being *fixed and merged* or *abandoned* after review votes. This indicates that code reviewers can effectively find the possible architectural issues in code and provide feasible refactoring recommendations to help developers to repair or remove the code snippets with erosion symptoms. It also indicates that the code review process has a positive impact on architectural improvements and sustainability by removing erosion symptoms in the two OpenStack projects.

However, a few of the identified erosion symptoms (10.4%) still remained and were ignored by developers and reviewers. One potential reason is that either the identified erosion symptoms did not attract the attention from the developers or different opinions existed between the developers and reviewers about the severity of the symptoms. The remaining erosion symptoms may increase the risk of architecture erosion, ultimately hindering the maintenance and evolution activities in the future.

4.6 Implications

4.6.1 Implications for Researchers

Establishing classification models to automatically identify architecture erosion symptoms. Whether architecture erosion symptoms are accurately identified plays a significant role in preventing architecture erosion and further extending the longevity of systems and their architecture. In this work, the identification process was manual. Researchers can focus on how to precisely and *automatically* identify erosion symptoms from various software artifacts. Furthermore, researchers can attempt to construct prediction models for recommending to *remove* erosion symptoms (Garcia et al., 2022). Erosion symptoms can be an early warning of architecture erosion, and the recommended repair activities (e.g., refactorings) can help to reduce the risk of architecture erosion. In this sense, establishing shared and labeled datasets of erosion symptoms based on diverse artifacts (e.g., source code, design documents) is the prerequisite for building models (e.g., classifiers based on deep learning techniques), which can be applied to form a more reliable basis for the identification of erosion symptoms.

Exploring the evolution of erosion symptoms. The aforementioned findings suggest that researchers can explore the evolution of architecture erosion symptoms for the purpose of preventing architecture erosion. For instance, researchers can measure the density of erosion symptoms in different modules over time and intervene if certain thresholds are reached (e.g., by refactoring). In addition, researchers can investigate the erosion symptoms that are not addressed in order to analyze whether and how the ignored erosion symptoms can have a negative impact on the system over time, as well as the root reasons why developers and reviewers ignored the symptoms.

4.6.2 Implications for Practitioners

Providing support for managing erosion symptoms. The findings of this work can provide guidance for developers in conducting refactoring and maintenance activities to deal with architecture erosion. For example, practitioners can pay more attention to those erosion symptoms that have the highest frequency (e.g., duplicate functionality). Moreover, to remove existing erosion symptoms and avoid introducing new erosion symptoms, practitioners can use the information on identified erosion symptoms in different textual artifacts (e.g., code reviews, commits, issues); raising their erosion awareness can help them be more cautious when performing

code changes (e.g., adding new features).

Keeping an eye on the remaining and ignored erosion symptoms. At present, the reasons for developers to accept a solution that will damage the architecture (e.g., sub-optimal design decisions leading to erosion symptoms) or refuse to address architecture erosion symptoms are still a research area under investigation (Paixao et al., 2019). To avoid the proliferation of erosion symptoms and mitigate the risk of architecture erosion, we encourage practitioners to systematically assess the severity of the ignored erosion symptoms, and continuously measure the impact of the remaining erosion symptoms (e.g., measure architecture smell size and density (Sharma et al., 2020)).

4.7 Threats to Validity

We discuss the threats to the validity by following the guidelines proposed by Wohlin *et al*. (Wohlin et al., 2012). Internal validity is not discussed, since we did not study causality.

Construct Validity pertains to the connection between the research questions and the objects of our study. One potential threat concerns the selection of the keyword set, namely, whether the keyword set was incomplete. To reduce the risk of missing keywords, we summarized the frequently-used keywords reported in previous studies, and identified the keywords by following the approach proposed by Bosu *et al*. (Bosu et al., 2014). Additionally, to further mitigate this threat, we conducted a random selection of review comments that did not contain any keywords.

External validity concerns the generalizability of the study findings. Our study used two OSS projects (i.e., Nova and Neutron) from the same community (i.e., OpenStack) and we have limited the code review data obtained from Gerrit between 2014 and 2018. These factors might restrict the generalizability of our findings in other settings, such as industrial systems, other OSS systems and different time periods. However, considering the popularity and size of the selected projects, we believe that our findings can provide researchers and practitioners an understanding of the common types and trends of architecture erosion symptoms identified in code reviews of large OSS systems, as well as the actions against the erosion symptoms.

Reliability refers to whether the study would yield the same results when other researchers conducted it. To reduce this threat, we ran a pilot data filtering and labeling to eliminate the misinterpretation of the results. The formal data filtering and labeling was conducted by the author, and reviewed and checked by another researcher; we obtained a Cohen's Kappa value (Cohen, 1960) of 0.898. During data

extraction and analysis, the data was extracted by the author and reviewed by another researcher. Any disagreements were discussed and addressed during the labeling and manual analysis of code review data. Besides, we specified the process of our study in Section 4.3 and provided a replication package online (Li, Soliman, Liang and Avgeriou, 2021a), which partially mitigate threats to the reliability of the study.

4.8 Related Work

4.8.1 Code Review

Code review is performed in a variety of ways for different purposes, for example, lightweight tool-based reviews (Bacchelli and Bird, 2013), checklist-based reviews (Gonçalves et al., 2020), and search-based reviews (Ouni et al., 2016a; Bosu et al., 2014; Han et al., 2021). Bacchelli and Bird (Bacchelli and Bird, 2013) investigated the tool-based code review process across different teams at Microsoft. They found that code review can facilitate knowledge transfer among team members and increase the team awareness, while available tools for code review do not always meet developers' expectations. Han *et al*. (Han et al., 2021) conducted an empirical study of code smell detection via code review, using a keyword-based approach to mine code review discussions of two OpenStack projects. They investigated the frequently identified code smells and the corresponding actions taken by developers. Ouni *et al*. (Ouni et al., 2016a) proposed the RevRec approach that formulates the peer code reviewers recommendation problem as a combinatorial search-based optimization problem, which provides decision-making support for code change submitters and reviewers to identify the most appropriate reviewers for code changes. Considering the benefits of code review on software development, we chose code review comments as our data source to empirically investigate the discussions on architecture erosion symptoms among developers during the code review process.

4.8.2 Identification of Architecture Erosion Symptoms

Macia *et al*. (Macia, Arcoverde, Garcia, Chavez and von Staa, 2012) explored the impact of code anomalies on architecture erosion, and their results revealed that certain kinds of early architectural smells (e.g., ambiguous interface, module concern overload) can be regarded as key architecture erosion symptoms and accelerate the erosion of architecture. Uchôa *et al*. (Uchôa et al., 2020) analyzed the impact of code review on design degradation evolution. They analyzed various erosion symptoms

to investigate the relationships between the density and diversity of symptoms and design degradation. Oizumi *et al*. (Oizumi et al., 2019) conducted an exploratory study to investigate whether symptoms of structural degradation (e.g., broken modularization, cyclic hierarchy, unutilized abstraction) with higher density and diversity in classes can be used as indicators of the need for root canal refactorings (a refactoring tactic). Their results indicated that certain symptoms might be indeed strong indicators of structural degradation, despite not being removed by refactoring. Mair *et al*. (Mair et al., 2014) considered architecture violations as a type of symptom of architecture erosion and investigated how software engineers repaired eroded software systems. Le *et al*. (Le et al., 2018, 2016) mentioned that architectural smells can be regarded as symptoms of architecture erosion and they proposed algorithms and metrics to detect instances of architecture erosion by analyzing the detected smells. Compared to the aforementioned studies that focus on the erosion symptoms by source code analysis (e.g., density of code smells (Uchôa et al., 2020), architectural smells (Le et al., 2018, 2016)), our work investigated the *discussions* on architecture erosion symptoms in code reviews, including the frequently discussed erosion symptoms and their trends, as well as the actions (i.e., *code changes*) taken by developers.

4.9 Conclusions and Future Work

To some extent, the trend of architecture erosion symptoms can reflect the trend of system sustainability and stability during evolution (Le et al., 2016). In this work, to study architecture erosion symptoms in code reviews, we performed an empirical study using the discussions of architecture erosion symptoms in code reviews by collecting and analyzing review comments from the two largest OpenStack projects (i.e., Nova and Neutron). Our findings show that *architectural violation*, *duplicate functionality*, and *cyclic dependency* are the most frequently identified erosion symptoms. The numbers and percentages of review comments on identified erosion symptoms manifest a declining trend in the two OpenStack projects, which indicates that their architecture becomes stable over time. Most of the identified erosion symptoms (89.6%) were addressed through either being *fixed and merged* or *abandoned* after review votes. This implies that, to some extent, code reviews might have a positive impact on removing erosion symptoms and extending the longevity of systems and their architecture.

Our findings suggest that researchers should establish classification models to support the identification of erosion symptoms and pay more attention to the evolution of erosion symptoms. Practitioners should manage the ignored erosion

symptoms with their evolution and continuously measure the impact of the erosion symptoms during the development life cycle. Besides, practitioners should be vigilant about the potential risk of architecture erosion to avoid erosion symptoms transferring to technical debt in a long run. To summarize, code review as a code inspection activity can help to find out and remove potential architecture erosion symptoms and to some extent prevent architecture erosion.

As a next step, we plan to empirically evaluate tools that can identify architecture erosion from source code and compare the results with associated artifacts (e.g., commits, issues) containing architecture erosion symptoms, as well as to investigate which symptoms cannot be identified by the existing tools.

Based on:
Ruiyin Li, Peng Liang, Paris Avgeriou, (2023) *"Warnings: Violation Symptoms Indicating Architecture Erosion," Information and Software Technology*, vol 164, p. 107319. DOI:10.1016/j.infsof.2023.107319

Chapter 5

Warnings: Violation Symptoms Indicating Architecture Erosion

Abstract

Context: As a software system evolves, its architecture tends to degrade, and gradually impedes software maintenance and evolution activities and negatively impacts the quality attributes of the system. The main root cause behind architecture erosion phenomenon derives from violation symptoms (i.e., various architecturally-relevant violations, such as violations of architecture pattern). Previous studies focus on detecting violations in software systems using architecture conformance checking approaches. However, code review comments are also rich sources that may contain extensive discussions regarding architecture violations, while there is a limited understanding of violation symptoms from the viewpoint of developers.
Objective: In this work, we investigated the characteristics of architecture violation symptoms in code review comments from the developers' perspective.
Method: We employed a set of keywords related to violation symptoms to collect 606 (out of 21,583) code review comments from four popular OSS projects in the OpenStack and Qt communities. We manually analyzed the collected 606 review comments to provide the categories and linguistic patterns of violation symptoms, as well as the reactions how developers addressed them.
Results: Our findings show that: (1) three main categories of violation symptoms are discussed by developers during the code review process; (2) The frequently-used terms of expressing violation symptoms are "*inconsistent*" and "*violate*", and the most common linguistic pattern is *Problem Discovery*; (3) Refactoring and removing code are the major measures (90%) to tackle violation symptoms, while a few violation symptoms were ignored by developers.
Conclusions: Our findings suggest that the investigation of violation symptoms can help researchers better understand the characteristics of architecture erosion and facilitate the development and maintenance activities, and developers should explicitly manage violation symptoms, not only for addressing the existing architecture violations but also preventing future violations.

5.1 Introduction

During software evolution, the implemented architecture tends to increasingly diverge from the intended architecture. The resulting gap between the intended and implemented architectures is defined as *architecture erosion* (Li, Liang, Soliman and Avgeriou, 2022; Perry and Wolf, 1992), and has been described using different terms in the literature and practice, such as architectural decay, degradation, and deterioration (Li, Liang, Soliman and Avgeriou, 2021b, 2022). Architecture erosion can negatively affect quality attributes of software systems, such as maintainability, performance, and modularity (Li, Liang, Soliman and Avgeriou, 2021b; Mendoza et al., 2021).

Architecture erosion can manifest in a variety of symptoms during the software life cycle; such symptoms indicate that the implemented architecture is moving away from the intended one. In our recent systematic mapping study (Li, Liang, Soliman and Avgeriou, 2022), four categories of architecture erosion symptoms were reported: *structural symptoms* (e.g., cyclic dependencies), *violation symptoms* (e.g., violation of the layered pattern), *quality symptoms* (e.g., high defect rate), and *evolution symptoms* (e.g., rigidity and brittleness of the software system). From these four types of symptoms, violation symptoms are deemed as the most critical symptoms that practitioners should address because the accumulation of violation symptoms can render the architecture completely untenable (De Silva and Balasubramaniam, 2012). Violation symptoms of architecture erosion include various types of architecturally-relevant violations in software systems, such as violations of design principles, architecture patterns, decisions, requirements, modularity, etc. (Li, Liang, Soliman and Avgeriou, 2022). For the sake of brevity, we refer to violation symptoms of architecture erosion as *violation symptoms* in the rest of this chapter.

While a handful of temporary violation symptoms might be innocuous regarding the software system, the accumulation of architecture violations can lead to architecture erosion (Perry and Wolf, 1992; De Silva and Balasubramaniam, 2012; Mendoza et al., 2021), severely impacting run-time and design-time qualities. Therefore, identifying and monitoring violation symptoms is crucial to reveal inconsistencies between the implementation and the intended architecture; eventually this can help to, at least partially, repair architecture erosion (Li, Liang, Soliman and Avgeriou, 2022; De Silva and Balasubramaniam, 2012).

Prior studies focusing on erosion symptoms through inspecting source code might ignore implicit semantic information, while our work dives into violation symptoms by analyzing textual artifacts such as code review comments from the developers' perspective. In contrast to violation symptoms identification from source code using predefined abstract models (Miranda et al., 2016; Pruijt and Brinkkem-

per, 2014) and rules (Terra and Valente, 2009; Rocha et al., 2017; Caracciolo et al., 2015; Juarez Filho et al., 2017), violation symptoms can also be detected by analyzing textual artifacts that contain information related to the system architecture and its design. Violation symptoms can occur at different stages of development and be self-admitted or pointed out by developers (Li, Soliman, Liang and Avgeriou, 2022). In our previous study (Li, Soliman, Liang and Avgeriou, 2022), we investigated two types of symptoms of architecture erosion in code reviews (i.e., structural and violations symptoms), and found that violation symptoms are the most frequently-discussed symptoms in architecturally-relevant issues during code review. Therefore, in this work, we focus on violation symptoms and attempt to categorize them and understand how developers express and address them.

Code review comments include rich textual information about the architecture changes that developers identified and discussed during development (Paixao et al., 2019). Code reviews are usually used to inspect defects in source code and help improve the code quality (Bacchelli and Bird, 2013). Compared to pull requests that might not provide specific advice for development practice (Li, Qi, Yu, Liang, Mo and Yang, 2021), code review comments provide a finer granularity of information for investigating architecture changes and violations from the developers' perspective.

Although valuable information on architecture violations is discussed during code review, there are no studies regarding the categories of violation symptoms that are discussed and admitted by developers, how developers express violation symptoms, and whether and how these symptoms are addressed during the development. To this end, we aim at understanding how developers discuss violation symptoms and providing an in-depth investigation to the categories of violation symptoms, as well as practical measures used to address them. We identified 606 (out of 21,583) code review comments related to architecture violations from four OSS projects in the OpenStack and Qt communities. The main contributions of this work are the following:

- We created a dataset containing violation symptoms of architecture erosion from code review comments, which can be used by the research community for the study of architecture erosion.

- We identified the 606 violation symptoms and classified them into three categories with ten subcategories, as well as the ways that developers addressed these categories.

- This is the first study that investigated violation symptoms in textual artifacts (specifically, code review comments) from the perspective of practitioners.

- We identified the linguistic patterns of expressing architecture violation symptoms from code review comments.

This chapter is organized as follows: Section 5.2 introduces the background of this study. Section 5.3 elaborates on the study design. The results of the research questions are presented in Section 5.4, while their implications are further discussed in Section 5.5. Section 5.6 elaborates on the threats to validity. Section 5.7 reviews the research work of this study. Finally, Section 5.8 summarizes this work and outlines the directions for future research.

5.2 Background

In this section, we overview the background of our study regarding code review and architecture erosion with the corresponding erosion symptoms.

5.2.1 Code Review

Code review is the process of analyzing assigned code for inspecting code and identifying defects. A methodical code review process can continuously improve the quality of software systems, share development knowledge, and prevent from releasing products with unstable and defective code. Currently, code review practices have become a crucial development activity that has been broadly adopted and converged to code review supported by tools. Moreover, tool-based code review has been widely used in both industry and open source communities. In recent years, many code review tools have been provided, such as Meta's Phabricator[1], VMware's Review-Board[2], and Gerrit[3].

Gerrit is a popular code review platform designed for code review workflows and is used in our selected projects (see Section 5.3.2). Once a developer submits new code changes (e.g., patches) and their description to Gerrit, the tool will create a page to record all the changes, and meanwhile the developer should write a message to describe the code changes, namely, a "*commit message*". Gerrit conducts a sanity check to verify the patch is compliant and to make sure that the code has no obvious compilation errors. After the submitted patch passes the sanity check, code reviewers will manually examine the patch and provide their feedback to correct any potential errors, and then give a voting score. Note that, code reviewers cannot only comment on source code but also on code commits. The review and vote

[1] https://www.phacility.com/
[2] https://www.reviewboard.org/
[3] https://www.gerritcodereview.com/

process will iterate with the purpose of improving the patch. Finally, the submitted patch will be merged into the code repository after passing the integration tests (i.e., without any issues and conflicts).

5.2.2 Architecture Erosion

The sustainability of architecture depends on architectural design to ensure the long-term use, efficient maintenance, and appropriate evolution of architecture in a dynamically changing environment (Venters et al., 2018). However, architecture erosion and drift are two essential phenomena threatening architecture sustainability. Architecture erosion happens due to the direct violations of the intended architecture, whereas architecture drift occurs due to extensive modifications that are not direct violations but introduce design decisions not included in the intended architecture (Perry and Wolf, 1992; Venters et al., 2018).

The architecture erosion phenomenon has been extensively discussed in the past decades and has been described by various terms (Li, Liang, Soliman and Avgeriou, 2022, 2021b), such as architecture decay (Hassaine et al., 2012; Le et al., 2018), degradation (Lenhard et al., 2019), and degeneration (Hochstein and Lindvall, 2005). Architecture erosion manifests in a variety of symptoms during development and maintenance. A symptom is a (partial) sign or indicator of the emergence of architecture erosion. According to our recent systematic mapping study (Li, Liang, Soliman and Avgeriou, 2022), the erosion symptoms can be classified into four categories: *structural symptoms* (e.g., cyclic dependencies), *violation symptoms* (e.g., layering violation), *quality symptoms* (e.g., high defect rate), and *evolution symptoms* (e.g., rigidity and brittleness of systems). Previous studies have investigated different symptoms of architecture erosion. Mair *et al*. (Mair et al., 2014) proposed a formalization method regarding the process of repairing eroded architecture through detecting violation symptoms and recommending optimal repair sequences. Le *et al*. (Le et al., 2018, 2016) regarded architectural smells as structural symptoms and provided metrics to detect instances of architecture erosion by analyzing the detected smells. Bhattacharya *et al*. (Bhattacharya and Perry, 2007) developed a model for tracking software evolution by measuring the loss of functionality (as evolution symptoms). Regarding the scope of our work, we focus on the nature of architecture erosion (i.e., violation symptoms) through code review comments in this work, which paves the way towards shedding light on architecture violations from the developers' perspective.

5.3 Methodology

The goal of this study is formulated by following the Goal-Question-Metric approach (Basili et al., 1994): **analyze** *code review comments* **for the purpose of** *identification and analysis* **with respect to** *violation symptoms of architecture erosion* **from the point of view of** *software developers* **in the context of** *open source software development*.

5.3.1 Research Questions

To achieve our goal, we define three Research Questions (RQs):

RQ1: What categories of violation symptoms do developers discuss?

Rationale: This RQ aims at investigating the categories of violation symptoms that frequently occur during the development process; an example of such a category is violations of architecture patterns. The proposed categories of violation symptoms in textual artifacts from code review comments can be used by practitioners as guidelines to avoid such violations in practice. For example, certain categories of violation symptoms may be associated with high erosion risks (Li, Liang, Soliman and Avgeriou, 2022) and be regarded as important to provide warnings to developers.

RQ2: How do developers express violation symptoms?

Rationale: Violation symptoms in code review comments are described in natural language, but there is a lack of evidence regarding how developers describe these violation symptoms. Specifically, we are interested in the terms and linguistic patterns[4] that developers use to denote violation symptoms. Establishing a list of the terms and linguistic patterns used by practitioners can subsequently provide a basis for the automatic identification of violation symptoms through natural language processing techniques.

RQ3: What practices are used by developers to deal with violation symptoms?

Rationale: We aim at exploring what developers do when they encounter violation symptoms during the development process; this includes whether developers address the violation symptoms and how they do that. The answers to this RQ can

[4]Grammatical rules that allow their users to speak properly in a common language (Da Silva, 2017)

help uncover best practices to cope with violation symptoms, and facilitate the development of methods and tools that promote such practices.

5.3.2 Project Selection

To understand violation symptoms that developers face in practice, we selected four OSS projects from two communities, namely OpenStack and Qt; these projects have been commonly used in previous studies (e.g., Kashiwa et al. (2022a)) due to their long development history and rich textual artifacts. OpenStack[5] is a widely-used open source cloud software platform, on which many organizations (e.g., IBM and Cisco) collaboratively develop applications for cloud computing. Qt[6] is a toolkit and a cross-platform framework for developing GUIs, and is used by around one million developers to develop world-class products for desktop, embedded, and mobile operating systems.

Both OpenStack and Qt contain a large number of sub-projects, thus we selected two sub-projects from each community: Neutron and Nova from the OpenStack community, and Qt Base and Qt Creator from the Qt community (see Table 5.1). Neutron (providing networking as a service for interface devices) and Nova (a controller for providing cloud virtual servers) are mainly written in Python; Qt Base (offering the core UI functionality) and Qt Creator (the Qt IDE) are mainly developed in C++. The selected four projects are the most active projects in the OpenStack and Qt communities, respectively, and they are widely known for a plethora of code review data recorded in the Gerrit code review system (Thongtanunam et al., 2017; Hirao et al., 2022).

5.3.3 Data Collection

An overview of our data collection, labelling, and analysis is shown in Figure 5.1. Starting with the process of data collection (the top part of Figure 5.1), we first employed Python scripts to mine code review comments (concerning source code and commits) of the four projects through the REST API[7] supported by the Gerrit tool. Then, we organized and stored the collected data in MongoDB. Our goal, as stated in the beginning of this section, is to analyze the violation symptoms that exist in code review comments from developers. Therefore, we removed the review comments that were generated by bots in the Qt community; we noticed that there were no code review comments generated by bots in the OpenStack community. However,

[5] https://www.openstack.org/
[6] https://www.qt.io/
[7] https://gerrit-review.googlesource.com/Documentation/rest-api.html

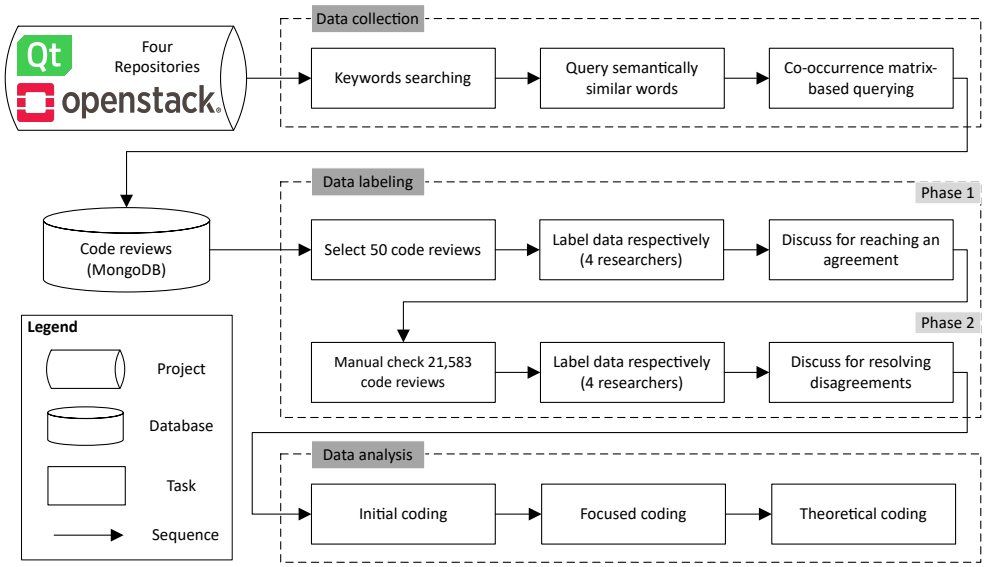

Figure 5.1: An overview of the data collection and analysis process

manually analyzing the entire history of code review comments of the four projects is prohibitive in terms of both effort and time. Thus, we decided to collect the code reviews of the four projects in seven years between Jan 2014 and Dec 2020 to guarantee sufficient revisions for long-lived software systems. Finally, we obtained 518,743 code review comments concerning code and 48,113 review comments concerning commit messages from the four projects in the past seven years. Each item in our dataset contains review ID and patch information, including `change_id`, `patch`, `file_url`, `line`, and `message`; the message variable includes code review comments concerning source code and commits. All the scripts and the dataset of this work have been made available in the replication package (Li, Liang and Avgeriou, 2022).

The collected review comments contain a large number of entries, such as "*Done*" and "*Ditto*", which are not related to the discussion on violation symptoms. To effectively collect and locate the associated code review comments on violation symptoms of architecture erosion, we decided to employ a keyword-based search approach. We employed the keywords presented in our previous work (Li, Soliman, Liang and Avgeriou, 2022) (see the coarse-grained keywords in Table 5.2) and improved the keyword set (see the fine-grained keywords in Table 5.2) as described below. Specifically, to derive possible and associated synonyms of the keywords in software engineering practices, we adopted the pre-trained word2vec model pro-

Table 5.1: An overview of the selected projects

Project	Domain	Repository	Language	# 1	# 2
Nova	Virtual server management	`https://opendev.org/openstack/nova`	Python	152,107	15,164
Neutron	Network connectivity	`https://opendev.org/openstack/neutron`	Python	181,839	16,719
Qt Base	Providing UI functionality	`https://code.qt.io/cgit/qt/qtbase.git/`	C++	123,546	13,369
Qt Creator	A cross-platform IDE	`https://code.qt.io/cgit/qt-creator/qt-creator.git`	C++	61,251	2,861
		Total		**518,743**	**48,113**

[1] *Review comments of code*
[2] *Review comments of commits*

posed by Efstathiou *et al*. (Efstathiou et al., 2018a) for querying semantically similar terms. The authors of (Efstathiou et al., 2018a) trained this model with over 15GB of textual data from Stack Overflow posts, which contain a plethora of textual expressions and words in the software engineering domain. We utilized this pre-trained word embedding model to query similar terms of the coarse-grained keyword set. For example, we got "*discrepancy*" and "*deviation*" which are similar terms to "*divergence*", and then two researchers discussed together to manually check and remove unrelated and duplicate words, such as "*oo*" which is related to programming languages rather than architecture violations. The keywords set used to search code review comments includes both the coarse-grained keywords and the fine-grained keywords listed in Table 5.2.

In addition, given that the effectiveness of the keyword-based approach highly depends on the set of keywords, we chose the iterative approach proposed by Bosu *et al*. (Bosu et al., 2014) to further improve the keyword set by adding possible keywords that are related to the keywords in Table 5.2. This approach has already been employed in previous studies (e.g., Li, Soliman, Liang and Avgeriou (2022); Han et al. (2021)) that used keyword-based search in code review data. We implemented this approach in the following steps:

1. Search in the collected review comments using the keyword set in Table 5.2, and then establish a corpus by collecting the relevant comments that encompass at least one keyword from the keyword set (e.g., "*violation*").

2. Process the retrieved code review comments that contain at least one keyword in our keyword set and remove English stopwords, punctuation, code snippets, and numbers. That results in a list of tokens.

3. Conduct a stemming process (using SnowballStemmer from the NLTK toolkit (Bird et al., 2010)) to obtain the stem of each token (e.g., "*architecture*" and "*architectural*" have the same token "*architectur*").

4. Build a document-term matrix from the corpus, and find additional words that co-occur frequently with each of our keywords (co-occurrence probability of 0.05 in the same document).

5. Manually check and discuss the list of frequently co-occurring additional words to determine whether the newly discovered words should be added to the keyword set.

After executing this approach, we have not found any keywords that co-occur with the keywords based on a co-occurrence probability of 0.05 in the same document. Therefore, we believe that we have minimized the possibility of missing potentially co-occurred and associated words, and the keyword set could be relatively adequate and comprehensive for the search in this study. The keywords used in this work are presented in Table 5.2. In total, we collected 21,583 code review comments from the four OSS projects that contain at least one keyword.

Table 5.2: Keywords related to violation symptoms of architecture erosion

Coarse-grained keywords
architecture, architectural, structure, structural, layer, design, violate, violation, deviate, deviation, inconsistency, inconsistent, consistent, mismatch, diverge, divergence, divergent, deviate, deviation
Fine-grained keywords
layering, layered, designed, violates, violating, violated, diverges, designing, diverged, diverging, deviates, deviated, deviating, inconsistencies, non-consistent, discrepancy, deviations, modular, module, modularity, encapsulation, encapsulate, encapsulating, encapsulated, intend, intends, intended, intent, intents, implemented, implement, implementation, as-planned, as-implemented, blueprint, blueprints, mis-match, mismatched, mismatches, mismatching

5.3.4 Data Labeling and Analysis

We filtered out a large number of irrelevant code review comments in Section 5.3.3. Still, the retrieved code review comments that contain at least one keyword might

be unrelated to violation symptoms. Thus, we needed to manually check and further remove these semantically unrelated review comments. We conducted **data labeling** in two phases, as illustrated in Figure 5.1.

Phase 1. We decided to conduct a pilot data labeling to reach a consensus and to ensure that we have the same understanding of violation symptoms. Four researchers (the author and three master students) had an online meeting to discuss the characteristics of violation symptoms. We randomly selected 50 review comments from the collected data. The four researchers independently labeled the violation symptoms from the review comments via MS Excel sheets, and provided reasons for their labeling results. Then the four researchers had another meeting to check the similarities and differences between their labeling results for reaching an agreement, and any disagreements were discussed with another researcher to reach a consensus. Note that, during the data labeling process, the researchers not only read the text content of the code review comments per se, but also read their corresponding code snippets, documentation, and commit messages. This helped us further mitigate the threat of wrong labels, such as simple violations at the code level (e.g., pep8 coding style violation[8]). In the end, to measure the inter-rater agreement between the researchers, we calculated the Cohen's Kappa coefficient value (Cohen, 1960) of the pilot data labeling and got an agreement value of 0.857, which demonstrates a substantial agreement between them.

Phase 2. After the pilot data labeling, the four researchers started the formal data labeling by dividing the retrieved 21,583 code review comments into four parts; each researcher manually labeled one fourth of this dataset (almost 5,400 review comments). The author created the MS Excel sheets and shared them with the other three researchers. The four researchers were asked to label the textual information associated with violation symptoms. After the formal data labeling, the author checked the data labeling results from the other three researchers to make sure that there were no false positive labeling results. To mitigate potential bias, we discussed all the conflicts in the labeling results until we reached an agreement. In other words, the data labeling results were checked by at least two researchers. The researchers followed the same process as in Phase 1 to conduct data labeling.

Finally, for **data analysis**, we employed Constant Comparison (Charmaz, 2014; Stol et al., 2016) to analyze and categorize the identified textual information. Constant Comparison (Charmaz, 2014; Stol et al., 2016) can be used to yield concepts, categories, and theories through a systematic analysis of qualitative data. The Constant Comparison process according to Charmaz *et al*. (Charmaz, 2014) includes three steps. The first step is *initial coding* executed by the four researchers, who examined the review comments by identifying violation symptoms from the re-

[8] `https://peps.python.org/pep-0008/`

trieved textual information. Second, we applied *focused coding* executed by the author and reviewed by another researcher, by selecting categories from the most frequent codes and using them to categorize the data. For example, "*feels like this DB work violates the level of abstraction we are expecting here*" was initially coded as *violation of abstraction*, and we merged this code into *violation of design principles*, since we considered that the violation of "*the expected level of abstraction*" belongs to *violation of design principles*. Third, we applied *theoretical coding* to specify the relationship between codes. We checked the disagreements on the coding results by the four researchers to reduce the personal bias, and discussed the disagreements with another researcher to get a consensus. The whole manual labeling and analysis process took the researchers around one and a half months.

During the data analysis process, if the violation symptoms were specifically stated, we assigned them to specific groups. Conversely, when the symptoms were defined more broadly or lacked specificity, we classified them into general categories. We relied solely on the explicit textual content of the comments themselves during data analysis, without subjective interpretation. Besides, we note that we did not find multiple violation symptoms were discussed in a single review comment. In other words, each identified review comment has one single label. In addition, unlike our previous study (Li, Soliman, Liang and Avgeriou, 2022) which focused on both structural and violation symptoms, we conducted the data collection and labeling processes in this study from scratch by following the aforementioned steps; consequently we have established a more comprehensive and larger dataset on violation symptoms (Li, Liang and Avgeriou, 2022).

5.4 Results

5.4.1 Overview

Before delving into the findings of the three RQs, we briefly report an overview of the descriptive statistics about the identified violation symptoms from the selected four projects.

Figure 5.2 shows the retrieved review comments containing the violation symptom keywords in Table 5.2 and the identified review comments related to violation symptoms from the four projects. We observed that (1) the proportion of retrieved review comments across the four projects aligns closely with the results presented in Table 5.1. Specifically, the percentages are as follows: Nova at 2.94%, Neutron at 2.42%, Qt Base at 2.22%, and Qt Creator at 1.15%; (2) the identified review comments account for a similar percentage of the retrieved review comments across the four

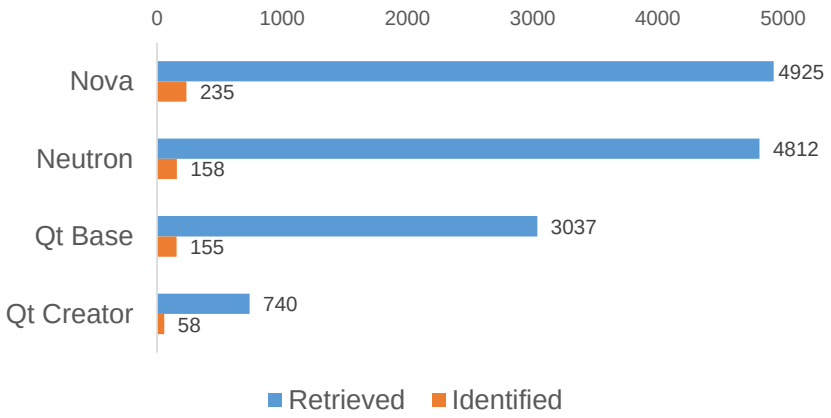

Figure 5.2: Overview of the retrieved review comments containing the violation symptom keywords and the identified review comments related to violation symptoms in the four projects

projects; the respective percentages are: Nova at 4.77%, Neutron at 3.28%, Qt Base at 5.10%, and Qt Creator at 7.84%.

In terms of the identified code review comments related to violation symptoms in our dataset, only a small portion of these comments (59 out of 606, 9.7%) pertained to the content in commit messages. In contrast, the vast majority of the identified review comments related to violation symptoms (547 out of 606, 90.3%) were associated with source code. Considering that the number of review comments on source code is 10 times higher than the number of review comments on commit messages as shown in Table 5.1, the proportion (10:1) is similar to the proportion of the identified review comments related to violation symptoms from the two sources (547:59).

5.4.2 RQ1 - Categories of Violations Symptoms

To answer RQ1, we identified 606 (out of 21,583) code review comments from the four projects that contain a discussion of violation symptoms of architecture erosion. As shown in Table 5.3, the collected code review comments can be classified into three categories of violation symptoms, with ten subcategories as follows.

- Design-related violation: six types of violation symptoms pertained to design are identified, including: structural inconsistencies, violation of design decisions, violation of design principles, violation of rules, violation of architecture patterns, and violation of database design.

- Specification-related violation: two types of violation symptoms related to specifications are identified, including: violation of documentation and violation of API specification.
- Requirement-related violation: two types of violation symptoms relevant to requirements are identified, including: violation of architecture requirements and violation of constraints.

In the following subsections, we present the detailed descriptions of the ten subcategories, accompanied by a range of real-world examples. The ten subcategories of violation symptoms are presented according to their frequencies in Table 5.3 within the dataset.

(1) Structural inconsistencies

Structural inconsistencies occur due to various reasons, for example, it might be associated with a change of software specifications, and related components that implement the specifications cannot be automatically updated to keep consistency (Grundy et al., 1998). Structural inconsistencies can delay system development, increase the cost of development process, and further jeopardise the properties of system quality (e.g., reliability and compatibility). For example, one developer mentioned how inconsistency leads to an increased number of collaborating modules (i.e., module dependencies), and consequently the complexity of the system:

"*Consistency is a nice thing, especially as in this case by not having consistency we are increasing the number of collaborating modules this module has.*"

Another example is related to inconsistent implementation between extension classes:

"*I think it could, but that would make it inconsistent with every other extension class's implementation of this method (I gripped). I think its best to keep them consistent. Feel free to file a low-hanging-fruit bug to fix this across all extensions.*"

In addition, architectural mismatch is another typical inconsistency issue in this category, which denotes the inability to successfully integrate component-based systems (Garlan et al., 1995). Architectural mismatch happens when there are conflicting assumptions among architecture elements (e.g., connectors and components) during development (Garlan et al., 1995, 2009). One reviewer mentioned an architectural mismatch between the server and agent side due to port design issues:

"*The port security extension adds functionality to disable port security, which is on by default. I don't think we should be changing the default behavior when port security is not present. User can explicitly disable port security if needed with port security extension. Enabling it here also makes a mismatch between server and agent side code.*"

Table 5.3: Categories of violation symptoms in code review comments

Category	Subcategory	Description	Count
Design-related violation	Structural inconsistencies	Violations of consistencies of structural design in architecture (e.g., architectural mismatch) that exist in various architectural elements (e.g., components, ports, modules, and interfaces).	205
	Violation of design decisions	Violations of selected design decisions, including design rationale, intents, or goals that may cause implementation errors and give rise to ever-increasing maintenance costs.	92
	Violation of design principles	Violations of the common design principles (e.g., the SOLID principles) or divergences from object-oriented development guidelines.	91
	Violation of rules	Violations of the predefined architecture rules or policies when the implementation does not actually follow the rules.	46
	Violation of architecture patterns	Violations of architecture patterns (e.g., violations of layered pattern) when the architecture pattern implementations do not conform to their definitions.	33
	Violation of database design	Violations of the principles or constraints in designing the database of systems.	25
Specification-related violation	Violation of documentation	Violations of the specification in development documents that hinder architecture improvements and modifications.	56
	Violation of API specification	Violations or inconsistencies of the API claims or specification.	36
Requirement-related violation	Violation of architecture requirements	Violations of the intended requirements (e.g., user requirements) during the development and maintenance process.	11
	Violation of constraints	Violations of specific constraints imposed by the intended architecture may have an impact on architecture design.	11

(2) Violation of design decisions

This category contains violations of design decisions that have been made, especially the violation of design rationale. Such violations may lead to implementation errors and consequently aggravate maintenance costs. For example, one developer stated:

💬"*I chose olso because the intent is for this file to eventually not use anything in nova, so adding a nova import seemed like a step in the wrong direction.*"

In addition, another reviewer pointed out a violation of design intent:

💬"*But if _get_provider_traits is being updated in this otherwise aggregate-specific change, it seems like we're splitting in two different directions which defeats the purpose of trying to be consistent.*"

Such violations imperil the actual implementation and lead to high error-proneness, and make software systems harder to implement, comprehend, maintain, and evolve. Here is an implementation error due to violating certain design decision as one developer mentioned:

💬"*Originally, QMT_CHECK was Q_ASSERT (before I added this code to QtCreator). It is a serious implementation error ... If you are not happy with a crash you can add a check against 0. This will avoid the crash here but I am pretty sure that it will crash sooner or later on a different location.*"

(3) Violation of design principles

The identified violation symptoms in this category denote violations of common design principles in object-oriented design and development, such as encapsulation, abstraction, and hierarchy (Martin and Martin, 2006). Common examples of object-oriented design principles include the SOLID principles proposed by Martin (Martin, 2003), i.e., *Single Responsibility Principle*, *Open Closed Principle*, *Liscov Substitution Principle*, *Interface Segregation Principle*, and *Dependency Inversion Principle*. As an example, one reviewer pointed out a violation of the Interface Segregation Principle:

💬"*But here don't we have to make a upcall from compute to api db, which will violate api/cell isolation rules. Is there any workaround in this case?*"

Another two examples are violations of abstraction and encapsulation:

💬"*Having said all of that: I get that I'm violating an abstraction layer in LinkLocalAddressPair and that this is surprising (and therefore bad).*"

💬"*It would seem to violate encapsulation to have to know to set the default value for an attribute outside of the object.*"

(4) Violation of documentation

The identified violation symptoms in this category encompass violations of instructions in documentation, e.g., not following the instructions on how to implement an interface. Such violations of documentation can hinder the subsequent architecture improvements and modifications (Macia, Arcoverde, Garcia, Chavez and von Staa, 2012). Two examples regarding violations of documentation are presented below:

💬*"The case of physical or service VM routers managed by L3 plugins doesn't appear to be supported here when the gateway IP is not or cannot be set to the router LLA. By supporting those cases in this way though, we're breaking the reference implementation as documented."*

💬*"The reimplementation in QStandardItemModel does *not* match the documentation as it replaces the entire set of roles with the new ones. It also violates the documentation with the silly EditRole->DisplayRole thing (touching the value for a role not passed in input; although one could say that QSIM 'aliases' EditRole with the DisplayRole in all cases)."*

(5) Violation of rules

The violations of architecture rules occur when the implementation does not actually follow the predefined rules by architects. Different systems have their own defined architecture rules or policies. During the development, rules provide the way to specify the allowed and specific relationships between architecture elements (e.g., components and modules). As mentioned by a developer:

💬*"... this implies that that caller is able to put constraints on the driver which may violate the rules built into the driver."*

Another violation of predefined rules pointed out by a reviewer is:

💬*"We would now be limiting new spawns to only be allowed to the host that was the cause of the violation, thus causing the violation to be made worse. But maybe this is okay, since there isn't much we can do if the policy has been violated prior to this."*

(6) Violation of API specification

All API-related violations and inconsistencies belong to this category. API documentation describes the explicit specification of interfaces and dependencies. Due to the changing business requirements and continuous demands to upgrade functionalities, API evolution is inevitable. Violations of API specification encompass improper API usages and inconsistent API calls. Improper applications of APIs can give rise to unexpected system behaviors and eventually cause architectural inconsistencies. For example, developers do not adhere to the contract or specification to use the required APIs. As one reviewer stated:

💬*"the point is that with neither a category backend nor completely disabled output, the macro should be unusable (even if it would print something, the category would be missing,*

i.e., the implementation would violate the api)."

Besides, inconsistent API calls can also cause inconsistencies, such as getting different responses from different versions of APIs (e.g., distinct parameters and returns). As one developer mentioned:

💬"*On v2 API, there is a lot of API inconsistencies and we need to fix them on v3 API. So we can change API parameters on v3 API.*"

(7) Violation of architecture patterns

Architecture patterns provide general and reusable solutions for particular problems, such as the layered pattern and client-server pattern. Violating architectural patterns undermines the sustainability and reliability of software systems and increases the risk of architecture erosion. For example, we found that violations of the layered pattern are one of the most common types of this category. Modern software systems often contain millions of lines of code across many modules. Therefore, employing hierarchical layers is a common practice to organize the relationships between modules. Violations of layered systems can negatively impact the quality attributes (e.g., reusability, maintainability, and portability), eventually leading to architecture erosion. For instance, a developer commented on a violation of the layered pattern:

💬"*That all said, I'm definitely -2 (even if not core ;-)) on that patch, because I think it's a layer isolation violation to just make the call here. It should be fixed at the Compute API level rather IMHO.*"

Another example regarding the layered pattern violation is:

💬"*We could get some race conditions when starting the scheduler where it would not know the allocation ratios and would have to call the computes, which is a layer isolation violation to me.*"

(8) Violation of database design

Databases are one of the key architectural elements (Bass et al., 2021), and can negatively impact the system quality attributes when their design is violated. This category includes the problems caused by code changes that violate the constraints of database design, such as primary and foreign key constraints. For example, as mentioned by one reviewer, the network port will no longer exist when a foreign key violation is generated:

💬"*If subnet was fetched in reference IPAM driver, port got deleted and foreign key violation was generated on ipallocation insert (because port no longer exists).*"

Another example is about unique key violation:

💬"*In the bug report, the randomly generated index happens to be 2, which violates the*

already existing (router, 1) unique key constraints."

(9) Violation of constraints

Constraints are pre-determined special design decisions with zero degrees of freedom (Bass et al., 2012), which can be regarded as special requirements that cannot be negotiable and impact certain aspects of architecture implementation. Such violations often denote the concrete statements, expressions, and declarations in source code that do not comply with the constraints imposed by the intended architecture (Terra et al., 2015), such as inter-module communication constraints. For example, one reviewer mentioned a constraint violation in the system:

💬*"If a specific subnet is passed in, then the IPAM system would try to give that subnet, but if it's already use or it violates the constraints in the IPAM system, it would refuse."*

Another example concerning constraint violation is:

💬*"This should probably be HTTPBadRequest: the provided allocation has a form that violates Inventory constraints, so if the allocation (the request body) changes, it could work."*

(10) Violation of architecture requirements

Requirements and especially quality attribute requirements are closely related to the architecture of a system. Architecturally significant requirements drive the architecture (Bass et al., 2021), but are, unfortunately, commonly violated (Li, Liang, Soliman and Avgeriou, 2022). Moreover, architecturally significant requirements specify the major features and functionalities that a particular product should include, and convey the expectations of the stakeholders for the software product. As an example, a developer mentioned that the existing code had violated the requirements:

💬*"We don't have this in our requirements ... I guess this was already violated by existing code so I don't really want to block on it, we can handle it separately."*

Another example of violation of requirements is:

💬*"This is global/module level data so it will get loaded once and stay loaded for the life time of the wsgi process even if the application is restrated in the interpreter. we are the ortinial code violated the requirement in pont3 of ..."*

5.4.3 RQ2 - Expression of Violation Symptoms

For answering RQ2, we inspected the content of the identified violation symptoms in code review comments to identify the frequently-used terms and associated linguistic patterns. Figure 5.3 presents the distribution of the most frequently-used terms related to the discussion of violation symptoms in review comments. We use

typical terms to represent the words which have the same meaning. For example, "*inconsistent*" contains all the terms that have the same meaning, such as *not consistent*, *inconsistent*, *inconsistency*, and *inconsistencies*. The most frequently-used term is "*inconsistent*" (37%, 225 out of 606) and is related to the notion of "*consistency*". "*Violate*" comes second with 140 (23%, out of 606) comments, followed by "*design*" (9%, 52 out of 606) and "*layer*" (6%, 35 out of 606). We put the less-frequently used terms into "*other terms*", such as "*module*" and "*architecture*".

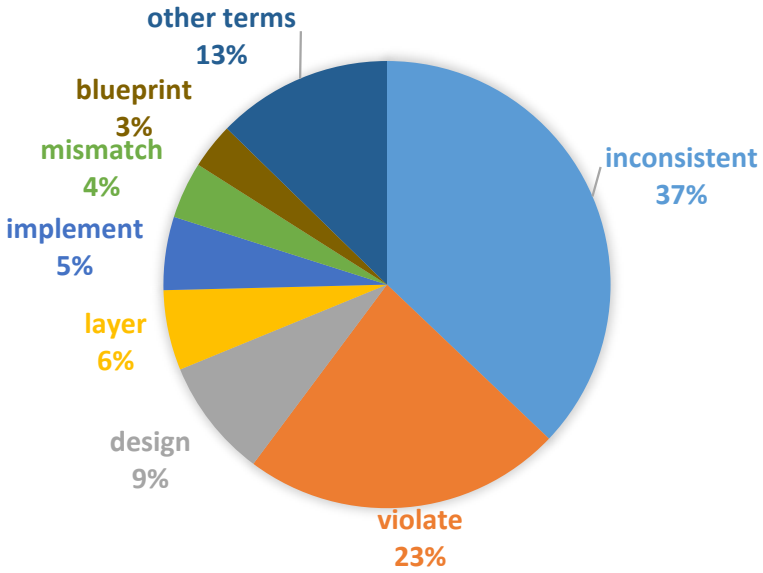

Figure 5.3: Distribution of the frequently used terms related to description of violation symptoms

In addition, to further analyze the characteristics of the comments related to violation symptoms, we summarized and categorized the linguistic patterns of expressing violation symptoms. We discovered the linguistic patterns by reviewing words or phrases that either frequently appear or are relatively unique to a particular category. Subsequently, we manually checked and categorized the comments into six categories based on the linguistic patterns proposed by Di Sorbo *et al*. (Di Sorbo, Panichella, Visaggio, Di Penta, Canfora and Gall, 2016), namely, *Feature Request*, *Opinion Asking*, *Problem Discovery*, *Solution Proposal*, *Information Seeking*, and *Information Giving*, which have been employed to identify the linguistic patterns used in various textual artifacts (e.g., app reviews (Di Sorbo, Panichella, Alexandru, Shimagaki, Visaggio, Canfora and Gall, 2016; Di Sorbo et al., 2017) and issue re-

Table 5.4: Categories and percentages of linguistic patterns used to express violation symptoms in code review comments

Linguistic Patterns	Description	Example	Percentage
Problem Discovery	Linguistic patterns related to unexpected or unintended behaviors	*"Well, so this patch actually makes the API worse: It is as unclear as before when a temporary file is created in the temporary directory and when not, *and* it deviates from the behavior of QTemporaryFile as well, so anyone knowing that behavior will get unexpected results here."*	90.4%
Solution Proposal	Linguistic patterns related to describe possible solutions for founded problems	*"I'm not convinced that it is wise to deviate from that with this one, other than to use QPlainTestLogger::outputMessage(), which has side-effects on WinCE, Windows and Android. ... Please consider inheriting QAbstractTestLogger instead and calling QAbstractTestLogger::outputString() instead of QPlainTestLogger::outputMessage()."*	10.4%
Opinion Asking	Linguistic patterns used for inquiring someone about his/her viewpoints and thoughts	*"Why diverge from the pattern established for toFoo_helper below?"*	6.8%
Information Giving	Linguistic patterns for informing someone about something	*"I didn't want to optimize in this way, because the code is wrong: The parent implementation should only be called if the sub-class does not handle the line itself. Several implementations got this wrong, and it is indeed not self-explanatory. I will fix all of those as part of moving away from the chaining approach."*	4.8%
Feature Request	Linguistic patterns for providing suggestions/recommendations/ideas	*"Recently there were some changes in API, virtual getters were replaced with protected setter. Please be consistent with it and provide protected setter and public getter, both non-virtual. ... I saw those changes came from them. And I strongly agree with it."*	2.0%
Information Seeking	Linguistic patterns related to ask help or information from others	*"I'm a bit confused as to how this violates the open/closed principle. ... I'm a bit unclear as to what your are suggestion as an alternative. Are you suggesting having a separate config option for each traffic type as an alternative?"*	1.8%

ports (Huang et al., 2018)). Table 5.4 presents the statistical results of the review comments related to violation symptoms, including the categories, descriptions, examples, and percentages. A few code review comments contain more than one linguistic pattern, and overall they add up to more than 100%. Our results show that most (90.4%) of the comments related to violation symptoms are about *Problem Discovery*, followed by *Solution Proposal* (10.4%) and *Opinion Asking* (6.8%). Moreover, we list the linguistic patterns (frequency ≥ 3) used to express violation symptoms in Table 5.5.

5.4.4 RQ3 - Dealing with Violation Symptoms

To answer RQ3 and gain a better understanding of how developers deal with violation symptoms, we plot a tree map (see Figure 5.4) of the distribution of the status (i.e., "`Merged`", "`Abandoned`", and "`Deferred`") of the patches containing violation symptoms. We further analyzed the developers' reactions (including *refactored*, *removed*, and *ignored*) in response to violation symptoms from code review comments.

We found that most (76.1%, 461 out of 606) of the violation symptoms are in "`Merged`" status, which means that developers agreed to merge the submitted code into the code repository. 23.1% (140 out of 606) of the patches are in "`Abandoned`" status, which means that the submitted code was rejected to be integrated into the code repository. Only a few patches (0.8%, 5 out of 606) stayed in "`Deferred`" status, which denotes a pending status and only exists in the code review of Qt (the reviewers and developers consider that the raised issues are not of very high priority and can be fixed in the following releases).

For the patches that contain violation symptoms, 77.7% (358 out of 461) of the merged patches and 82.9% (116 out of 140) of the abandoned patches, were addressed by refactoring. Moreover, 12.8% (59 out of 461) and 9.5% (44 out of 461) of violation symptoms were removed (i.e., deleted the code) and ignored (i.e., no changes), respectively, in the merged patches. Similar percentage of violation symptoms were removed (7.1%) and ignored (10.0%) in the abandoned patches. In the five deferred patches, developers refactored four submitted code snippets to cope with the violation symptoms and ignored one of them, while the remaining issues would be addressed in future releases.

Table 5.5: Linguistic patterns (frequency ≥ 3) used to express violation symptoms in code review comments

#	**Problem Discovery**
1	This is breaking [something]
2	This seems to violate [something]
3	It seems like a (layer) violation of [something]
4	It seems inconsistent/not consistent with [something]
5	[someone] probably have [an issue]
6	This violates/breaks [something]
7	[someone] is/are violating the rules
8	This is a violation of [something]
9	This is not consistent with [something]
10	[something] is/are inconsistent with [something]
11	There will be inconsistencies in [something]
12	There are inconsistencies [between something]/[of/in something]
13	[something] leads to inconsistency
14	[something] is a (poor/awful/terrible) design mistake/flaw/choice
15	[something] diverges/deviates from [something]
#	**Solution Proposal**
1	I think [someone] should/need to [verb + subject]
2	[someone] should modify/preserve/revise [something]
3	We should remove [something]
4	I think [do something] would better if [something]
5	[someone] should/need to stay/keep consistent with [something]
6	To fix it, we need to [do something]
7	I think what we should do is [something]
8	Did you consider the approach of [something]
#	**Opinion Asking**
1	Why are you diverging from [something]?
2	Would it be possible to [do something]?
3	Maybe we should [do something], what do you think about it?
4	Should/would we [do something]?
#	**Feature Request**
1	We should/need to keep consistent with [something]
2	It is better to [do something]
3	I wonder if we can [do something]
#	**Information Giving**
1	I will modify/fix [something] to address/keep consistent with [something]
2	That is why I [doing something]
#	**Information Seeking**
1	Is there a different [something]?
2	Are you suggesting [something]?
3	Are we planning to [do something]?

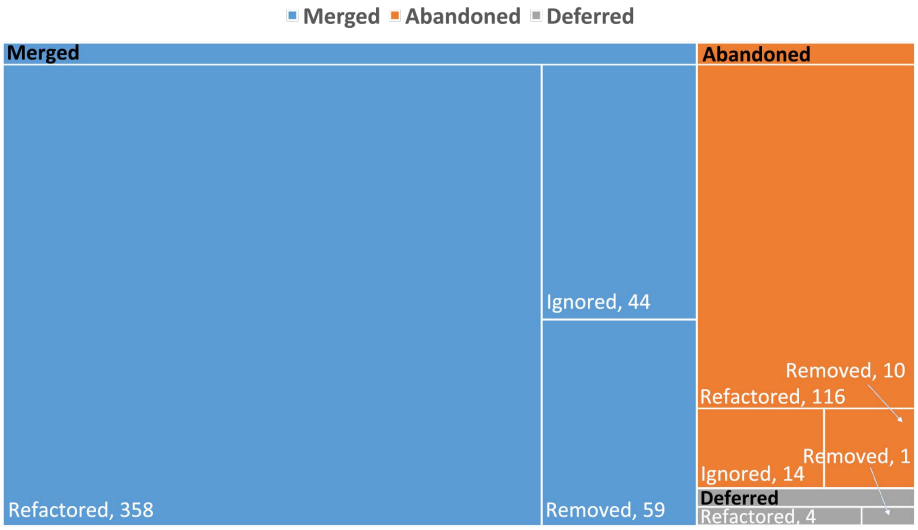

Figure 5.4: Distribution of the developers' reactions in response to violation symptoms in code review comments

5.5 Discussion

In this section, we first interpret the study results, and we then discuss the implications of the results for practitioners and researchers.

5.5.1 Interpretation of Results

The percentage of the identified violation symptoms from code review comments is rather low (2.8%, 606 out of 21,583) in the selected four OSS projects (i.e., Nova, Neutron, Qt Base, and Qt Creator). Prior studies (Li, Soliman, Liang and Avgeriou, 2022; Paixao et al., 2019) show that the percentage of architecturally-relevant information is comparatively lower than code-level issues (e.g., code smells) in code reviews, and our results comply with the findings from previous studies. Although the low percentage of architecturally-relevant code review comments, the identified architectural issues (especially architecture violation symptoms) can have a seriously negative impact on software maintenance and evolution.

RQ1: Categories of violation symptoms. We classified the collected violation symptoms into three categories of violation symptoms with ten subcategories that developers often discuss during development. Design-related violations are the

main category of violation symptoms. Specifically, we observed that structural inconsistency is the most common subcategory of violation symptoms. Structural inconsistency might be triggered by classes with many methods that make systems tend to be complex, overloaded, and contain architectural smells, and such classes have a greater possibility of being reused and becoming the source of architectural inconsistencies (Lenhard et al., 2019). Moreover, structural inconsistencies might be hard to detect with tools. For instance, *architecture mismatch* is a kind of structural inconsistency that is relatively difficult to detect by off-the-shelf tools due to various reasons (e.g., standard architectural description languages are used to document architecture, but they generally do not support tool-assisted detection of architecture mismatches) (Garlan et al., 2009).

Additionally, our results show that certain design-related factors are also common sources of violation symptoms that we cannot ignore. For instance, violations of layered pattern (i.e., *violation of architecture patterns* in Section 5.4.2) undermine the sustainability and reliability of systems and may gradually lead to architecture erosion due to their accumulation. In many cases, such violations usually require considerable effort to repair, to the extent that such a repair may not be financially feasible (Li, Liang, Soliman and Avgeriou, 2022; Sarkar et al., 2009). Besides, our results show that certain design-related violations (e.g., *violations of design decisions*, *design principles*, and *rules*) are common during development, and one possible reason is that the missing architectural knowledge leading to these violations might exist in certain small groups such as architects or team leaders. Therefore, our results suggest that disseminating and sharing the architectural knowledge related to violation symptoms across the development team is necessary.

RQ2: Linguistic patterns expressing violation symptoms. The results of RQ2 show that most (60%) of violation symptoms contain the terms about "*inconsistent*" (e.g., not consistent and inconsistency) and "*violate*" (e.g., violation and violating) (see Figure 5.3). One possible explanation is that such terms are more in line with the idiomatic expressions used by developers and reviewers, and they are commonly used to discuss issues related to violation symptoms in system design.

Regarding the linguistic patterns, the results show that the linguistic patterns of expressing violation symptoms from code review comments can be mapped to the categories of linguistic patterns identified in development emails (Di Sorbo, Panichella, Visaggio, Di Penta, Canfora and Gall, 2016). The six linguistic patterns can also be used to analyze textual artifacts in other sources (e.g., app reviews (Di Sorbo, Panichella, Alexandru, Shimagaki, Visaggio, Canfora and Gall, 2016; Di Sorbo et al., 2017) and issue reports (Huang et al., 2018)), and our findings indicate that code review discussions also encompass these categories of linguistic patterns. Besides, the major type of linguistic patterns of expressing violation

symptoms is *Problem Discovery* (see Table 5.4). We conjecture that developers incline to use the linguistic patterns regarding *Problem Discovery* (see Table 5.5) to specify violation symptoms, as one of the aims of code review is to identify issues during development.

RQ3: Reactions to violation symptoms from developers. We found that most of the identified violation symptoms were merged into the code repositories after refactoring or removing the smelly code. This observation indicates that code review is necessary, and to a large extent these review comments can help to mitigate the risk of architecture erosion caused by violation symptoms. In addition, developers' reactions (i.e., 77.7% patches are merged into the code base) also indicate that the review comments raising violation symptoms are crucial, especially for large-scale and long-term projects (Bosu et al., 2016a), as these review comments help reduce violation symptoms and improve the quality of software systems.

In some sense, *remove* the code that contains violation symptoms can also be considered as code improvement, for example, removing duplicated code or redundant dependencies decreases code complexity and increases system maintainability. Therefore, the percentage of *improvement* (*refactored* + *removed*) regarding addressing violation symptoms accounts for around 90% no matter whether it is merged into the code repositories or not. Only a small percentage (9.6%) of the identified violation symptoms were ignored and remained in the systems, and one possible reason is that developers have different opinions about how to address the remaining violation symptoms without reaching an agreement. Another possible reason is that the submitted code containing violation symptoms is not quite urgent to be fixed and has a lower priority, as the priority of issues depends on the severity and degree of the impact on different quality attributes (Li, Liang, Soliman and Avgeriou, 2022). In general, the study results show that developers incline to repair the issues related to violation symptoms when they are pointed out or discussed during code review.

5.5.2 Implications

(1) Implications for researchers

We identified three categories of violation symptoms from code review comments. Researchers are encouraged to investigate violation symptoms from other artifacts (such as pull requests, issues, and developer mailing lists) to provide more comprehensive empirical evidence, in order to further validate and consolidate the observations in this study. For example, the categories of violation symptoms can be further explored because only four OSS projects (written in Python and C++) from

two communities (OpenStack and Qt) were used in our study. It would be worth exploring violation symptoms in both industrial and OSS projects written in other programming languages (e.g., Java) and communities (e.g., Apache). Besides, researchers can further conduct a comparison between the identified violation symptoms by utilizing certain off-the-shelf architecture conformance checking techniques (e.g., reflexion models (Murphy et al., 1995)) and our dataset (i.e., manually collected violation symptoms). More specifically, they can try to perform quantitative comparisons regarding the identified violation symptoms between code and textual artifacts (e.g., code review comments and commit messages) with the purpose of evaluating the performance of the techniques. In addition, researchers can also try to map the violation symptoms from textual artifacts to code in order to improve and complement the existing architecture conformance checking techniques.

Moreover, we have created a dataset (Li, Liang and Avgeriou, 2022) containing violation symptoms of architecture erosion from code review comments. This dataset can act as a foundation for future study on architecture erosion (Li, Liang, Soliman and Avgeriou, 2022), especially architecture violation symptoms. For example, researchers can further explore the possibility of automatic identification of violation symptoms from textual artifacts through employing natural language processing techniques based on machine learning and deep learning algorithms. Automatically identifying violation symptoms would be of great value to developers, as manual identification can be extremely tedious, effort-intensive, and error-prone. Specifically, based on the models trained by machine learning and deep learning algorithms, researchers can devise auxiliary plugins to existing code review tools for providing warnings of violation symptoms to developers during development and maintenance.

(2) Implications for practitioners

The results in Section 5.4 and their explanations in Section 5.5.1 can be used by practitioners to guide their refactoring and maintenance activities. For example, having the categories of violation symptoms might help the practitioners to be aware of the possible violations in system architecture and then consider avoiding or repairing such issues in their daily development work. Moreover, the frequently-used terms and linguistic patterns related to descriptions of violation symptoms from the viewpoint of developers, can help developers pay more attention to violation symptoms during maintenance and evolution. For example, such terms and linguistic patterns could be a clear signal that developers should be wary of architecture erosion risks and avoid the appearance of violation symptoms.

Furthermore, we encourage practitioners to manage violation symptoms with

the purpose of facilitating refactoring and repairing architecture violations. As reported by Schultis *et al*. in Siemens, architecture violations must be explicitly managed, which includes addressing the existing architecture violations and preventing future violations (Schultis et al., 2016). Therefore, architecture teams (especially architects) should take the responsibility for collecting and monitoring violation symptoms, and then equip developers with the knowledge to repair or minimize architecture violations during development. Thus, practitioners can work with researchers and put effort to developing dedicated tools for managing violation symptoms and improving the productivity of maintenance activity.

5.6 Threats to Validity

The threats to the validity of this study are discussed by following the guidelines proposed by Wohlin *et al*. (Wohlin et al., 2012). Internal validity is not considered, since this study does not address any causal relationships between variables.

Construct validity pertains to whether the theoretical and conceptual constructs are correctly interpreted and measured. In this work, one potential threat is about the construction of the keyword set. To mitigate this threat, we first built the keyword set based on previous studies, and then we employed a pre-trained word embedding model to query and select similar keywords in the software domain. Besides, we constructed and used the co-occurrence matrix approach proposed by Bosu *et al*. (Bosu et al., 2014) to check the possible missing co-occurring words. In this way, the potential threat can be, at least partly, mitigated.

External validity concerns the extent to which we can generalize the findings to other studies. First, a potential threat to external validity is whether the selected projects are representative enough. Our work chose the two largest and most popular OSS projects (i.e., Nova and Neutron) from the OpenStack community, both written in Python language; we further selected another two major OSS projects (i.e., Qt Base and Qt creator) from the Qt community, which are written in C++. Second, another threat is that only Python and C++ OSS projects were selected, which may reduce the generalizability of the study results. Our findings may not generalize or represent all open source and closed source projects. It would be interesting to select more projects from different sources and programming languages to increase the external validity of the study results. Besides, it is worth exploring the generalizability of the findings regarding the frequent terms and linguistic patterns related to the identified violation symptoms in this work, for example, to investigate whether these findings are also applicable with other artifacts (e.g., pull requests and issues).

Reliability refers to the replicability of a study regarding yielding the same or

similar results when other researchers reproduce this study. The potential threat is mainly from the activities of data collection and data analysis. For data collection, we presented the detailed data collection steps in Section 5.3.3 and provided a replication package (Li, Liang and Avgeriou, 2022) for reproducing the data collection and filtering process, which can help to enhance the reliability of the results. Regarding data labeling, our observations show that developers generally discussed individual violation symptoms within one single review comment, as such, the threat of multiple symptoms discussed in one review comment with multiple labels is not present in this work. As for the data analysis, to mitigate personal bias, we conducted a pilot labeling and classification (see Phase I in Section 5.3.4) before the formal data labeling and classification process, and we got a Cohen's Kappa value of 0.857, which indicates a substantial inter-rater agreement. Likewise, we executed a similar process (see Phase II in Section 5.3.4) when we conducted the formal data labeling and analysis. Any disagreements were discussed between the four researchers to reach an agreement and at least two researchers participated in the data labeling and classification process.

5.7 Related Work

In this section, we discuss the work related to our study, which involves architecture violations and their detection approaches (i.e., architecture conformance checking), as well as the data sources (i.e., code review comments) used in this study.

5.7.1 Architecture Violations

Over the past decades, there have been extensive investigation on architecture violations. Brunet *et al*. (Brunet et al., 2012) performed a longitudinal study to explore the evolution of architecture violations in 19 bi-weekly versions of four open source systems. They investigated the life cycle and location of architecture violations over time by comparing the intended and recovered architectures of a system. They found that architecture violations tend to intensify as software evolves and a few design entities are responsible for the majority of violations. More interestingly, some violations seem to be recurring after being eliminated. Mendoza *et al*. (Mendoza et al., 2021) proposed a tool ArchVID based on model-driven engineering techniques for identifying architecture violations, and the tool supports recovering and visualizing the implemented architecture.

Moreover, Terra *et al*. (Terra et al., 2015) reported their experience in fixing architecture violations. They proposed a recommendation system that provides refac-

toring guidelines for developers and maintainers to repair architecture violations in the module architecture view of object-oriented systems. The results show that their approach can trigger correct recommendations for 79% architecture violations, which were accepted by architects. Maffort *et al*. (Maffort et al., 2016) proposed an approach to check architecture conformance for detecting architecture violations based on defined heuristics. They claimed that their approach relies on the defined heuristic rules and can rapidly raise architectural violation warnings. Different from the abovementioned studies focusing on detecting architecture violations in source code, our work investigates the architectural violation symptoms in code review comments from the perspective of developers, including the categories and linguistic patterns of expressing violation symptoms, as well as the reactions developers take to deal with violation symptoms.

5.7.2 Architecture Conformance Checking

Architecture conformance checking techniques are the most commonly-used approaches to detect architecture violations (Li, Liang, Soliman and Avgeriou, 2022). They can be checked statically or dynamically, and they are usually performed to compare the structure of the intended architecture (provided by the architects) with the extracted architecture information from source code that implements the architecture. For example, Pruijt *et al*. (Pruijt and Brinkkemper, 2014) proposed a metamodel for extensive support of semantically rich modular architectures in the context of architecture conformance checking. Miranda *et al*. (Miranda et al., 2016) presented an architectural conformance and visualization approach based on static code analysis techniques and a lightweight type propagation heuristic. They evaluated their approach in three real-world systems and 28 OSS systems to identify architecture violations.

Besides, rule-based conformance checking approaches are also employed to identify architecture violations. For example, previous studies detected architecture violations by checking the explicitly defined architectural rules (Mendoza et al., 2021; Schröder and Riebisch, 2017; Terra and Valente, 2009). Moreover, it is viable to check architecture conformance and identify architecture violations by defining and describing the systems through Architecture Description Languages (ADLs) (Terra and Valente, 2009; Rocha et al., 2017), or Domain-Specific Languages (DSLs) (Caracciolo et al., 2015; Juarez Filho et al., 2017). However, the aforementioned approaches have obvious limitations; for example, much effort is required to address the challenges of understanding the architecture design (e.g., concepts and relations), defining architectural rules (or description languages) in advance, and establishing a mapping between architectural elements and source code. Moreover, other limi-

tations, such as lack of generalizability, visualization of architecture views, and insufficient tooling support, hinder the above approaches from being widely used in practice. Additionally, to the best of our knowledge, prior studies regarding architecture violations focus on checking architecture conformance with source code using predefined abstract models and rules, and there is no evidence-based knowledge on identifying architecture violations from textual artifacts, such as code review comments.

5.7.3 Code Review Comments

Code review comments contain massive knowledge related to software development, and a variety of studies analyzed software defects and evolution through mining review comments and commit records. Zhou and Sharma (Zhou and Sharma, 2017) designed an automated vulnerability identification system based on a large number of commits and bug reports (containing rich contextual information for security research), and their approach can identify a wide range of vulnerabilities and significantly reduce false positives by more than 90% compared to manual effort.

Besides, Uchôa *et al*. (Uchôa et al., 2020) investigated the impact of code review on the evolution of design degradation through mining and analyzing a plethora of code reviews from seven OSS projects. They found that there is a wide fluctuation of design degradation during the revisions of certain code reviews. Paixão *et al*. (Paixão et al., 2020) explored how developers perform refactorings in code review, and they found that refactoring operations are most often used in code reviews that implement new features. Besides, they observed that the refactoring operations were rarely refined or undone along the code review, and such refactorings often contribute to new code smells and bugs. Given that previous studies discussed above investigated various aspects of code review regarding development and maintenance (e.g., decisions and design degradation), there are no studies that investigate architecture violations through code review comments; we decided to explore the violation-related issues (i.e., violation symptoms) from code review comments.

5.8 Conclusions

As software systems evolve, the changes in the systems could lead to cascading violations, and consequently the architecture will exhibit an eroding tendency. In this work, we conducted an empirical study to investigate the discussions on violation symptoms of architecture erosion from code review comments. We collected a large number of code review comments from four popular OSS projects in the OpenStack

(i.e., Nova and Neutron) and Qt (i.e., Qt Base and Qt Creator) communities. Our results show that ten subcategories of violation symptoms in three main categories are discussed by developers during the code review process. Besides, we found that the most frequently-used terms related to the description of violation symptoms concern *structural inconsistencies*, *design-related violations*, and *implementation-related violations*, such as *violation of design decisions*, *design principles*, and *architecture patterns*; the most common linguistic pattern (90.4%) used to express violation symptoms is *Problem Discovery*. *Refactoring* is the major measure that developers used to address violation symptoms, no matter whether the smelly code is integrated (i.e., 77.7% refactorings happened in the merged patches) or not (i.e., 82.9% refactorings happened in the abandoned patches). The finding indicates that code review can help reduce violation symptoms and increase system quality.

Our findings encourage researchers to investigate violation symptoms from various artifacts (e.g., pull requests, issues, and developer mailing lists) in order to provide more comprehensive evidence for validating and consolidating the findings. The most frequently-used terms and linguistic patterns used to express violation symptoms can help researchers and practitioners better understand and be aware of the natural language on describing violation symptoms of architecture erosion commonly used by developers. Besides, explicitly managing violation symptoms can to some extent help reduce the occurrence of architecture violations and prevent future violations during development and maintenance.

Developers usually discuss and address design-related issues in artifacts such as commits, issues, and pull requests (Brunet et al., 2014). In this context, we plan to construct classification models based on textual artifacts with machine learning and deep learning techniques for the purpose of automatically notifying developers about the potential violation symptoms of architecture erosion during development; for example, as a plugin to the Gerrit tool during the code review process. We also plan to invite practitioners to evaluate the effectiveness and efficiency of the proposed classification models and the tool on assisting developers in detecting violation symptoms.

Based on:

Ruiyin Li, Peng Liang, Paris Avgeriou, *"Towards Automatic Identification of Violation Symptoms of Architecture Erosion," (under review)* in a scientific journal.

Chapter 6

Towards Automatic Identification of Violation Symptoms of Architecture Erosion

Abstract

Architecture erosion has a detrimental effect on maintenance and evolution, as the implementation drifts away from the intended architecture. To prevent this, development teams need to understand early enough the symptoms of erosion, and particularly violations of the intended architecture. One way to achieve this, is through the automatic identification of architecture violations from textual artifacts, and particularly code reviews. In this chapter, we developed 15 machine learning-based and 4 deep learning-based classifiers with three pre-trained word embeddings to identify violation symptoms of architecture erosion from developer discussions in code reviews. Specifically, we looked at code review comments from four large open-source projects from the OpenStack (Nova and Neutron) and Qt (Qt Base and Qt Creator) communities. We then conducted a survey to acquire feedback from the involved participants who discussed architecture violations in code reviews, to validate the usefulness of our trained classifiers. The results show that the SVM classifier based on *word2vec* pre-trained word embedding performs the best with an F1-score of 0.779. In most cases, classifiers with the *fastText* pre-trained word embedding model can achieve relatively good performance. Furthermore, 200-dimensional pre-trained word embedding models outperform classifiers that use 100 and 300-dimensional models. In addition, an ensemble classifier based on the majority voting strategy can further enhance the classifier and outperforms the individual classifiers. Finally, an online survey of the involved developers reveals that the violation symptoms identified by our approaches have practical value and can provide early warnings for impending architecture erosion.

6.1 Introduction

Software architecture is regarded as critical in developing large and complex systems, as it reflects the system structure and behavior and acts as a bridge between business goals and the implemented system (Bass et al., 2021). However, the phenomenon of *architecture erosion* has become one of the major challenges of architecting, impeding software maintenance and undermining software sustainability (Li, Liang, Soliman and Avgeriou, 2022; Le et al., 2016; Venters et al., 2018). Architecture erosion refers to the increasing and accumulating divergences between implemented and intended architecture (Li, Liang, Soliman and Avgeriou, 2022; Perry and Wolf, 1992). It is often described by various terms in the literature (Li, Liang, Soliman and Avgeriou, 2022) and practice (Li, Liang, Soliman and Avgeriou, 2021b), such as architectural decay, degradation, and deterioration.

As evidenced in previous studies (Li, Soliman, Liang and Avgeriou, 2022; Li, Liang, Soliman and Avgeriou, 2022; Herold et al., 2015; Macia, Garcia, Popescu, Garcia, Medvidovic and von Staa, 2012; Fontana et al., 2016), various symptoms or signs can be observed when architecture erosion occurs during a software system's life cycle. Our recent systematic mapping study (Li, Liang, Soliman and Avgeriou, 2022) reported four categories of such architecture erosion symptoms: *structural symptoms* (e.g., cyclic dependencies), *violation symptoms* (e.g., layering violation), *quality symptoms* (e.g., high defect rate), and *evolution symptoms* (e.g., extensive ripple effects of changes).

In this study, we specifically concentrate on violation symptoms, as those are the most immediate symptoms of architecture erosion and they require the most attention from practitioners (Li, Soliman, Liang and Avgeriou, 2022; Li et al., 2023d). This symptom type stems from the definition of architecture erosion (Li, Liang, Soliman and Avgeriou, 2022), as a violation causes the implemented architecture to diverge from the intended one. Architecturally-relevant violations in software systems include violations of design principles, architecture patterns, decisions, and requirements (Li, Liang, Soliman and Avgeriou, 2022). The accumulation of violation symptoms can render the architecture completely untenable (De Silva and Balasubramaniam, 2012). For brevity, we refer to *violation symptoms of architecture erosion* as *violation symptoms* throughout the remainder of this chapter.

While a few temporary violation symptoms might be innocuous and might not result in software errors, the accumulation of architecture violations can affect architecture erosion (Perry and Wolf, 1992; De Silva and Balasubramaniam, 2012; Mendoza et al., 2021); consequently, erosion negatively affects quality attributes, including maintainability and performance (Li, Liang, Soliman and Avgeriou, 2021b; Mendoza et al., 2021). Thus, it is essential to identify and monitor violation symptoms

to facilitate the maintenance of the system architecture and further reveal inconsistencies between the implemented and the intended architecture. This is the first step towards, eventually, repairing or mitigating architecture erosion (De Silva and Balasubramaniam, 2012; Li, Liang, Soliman and Avgeriou, 2022).

Violation symptoms can be identified through source code analysis, but that has certain limitations. As pointed out in previous studies (Sharma and Spinellis, 2018; Li, Liang, Soliman and Avgeriou, 2022; Li et al., 2023d), various factors (e.g., limited types of architecture violations detected and supported programming languages) hinder the effective utilization of static code analysis tools in precisely identifying violation symptoms through source code. Alternatively, violations symptoms can be manually identified by analyzing textual artifacts that contain information related to system architecture and design, such as code comments, code reviews, and issue trackers (Li, Soliman, Liang and Avgeriou, 2022). In this context, recent work (Li, Soliman, Liang and Avgeriou, 2022; Li et al., 2023d) has introduced a taxonomy on violation symptoms, that can be used as a checklist to detect such symptoms during code review. However, it could be tedious, time-consuming, and potentially error-prone to manually identify potential architecture violations (Li et al., 2023d). Automated techniques could address this problem, but to the best of our knowledge, no prior study has investigated the automatic identification of violation symptoms in textual artifacts, such as code review comments.

To address this gap, we conduct an exploratory study to assess the feasibility of automatically identifying violation symptoms of architecture erosion from code review comments. The automated identification of violation symptoms could act in a complementary way to modern code review: the latter addresses code changes in general through knowledge sharing (Badampudi et al., 2023), while the former can offer an extra layer of scrutiny. For example, such an automated approach can issue warnings for potential or ignored violation symptoms, preventing the integration of code changes with architectural violations into the code base, and ensuring a more robust and reliable development process. Moreover, automatically identifying violation symptoms from code review comments could offer valuable insights at the architecture level and facilitate the corresponding knowledge sharing between developers and reviewers. The ultimate purpose of our approach is to complement the modern code review process, through warning of and guarding against architectural violation issues, eventually maintaining architectural integrity.

In this work, we developed 15 classifiers based on Machine Learning (ML) and 4 classifiers based on Deep Learning (DL), to identify violation symptoms from developer discussions in code reviews of four large open-source software projects from the OpenStack and Qt communities. Our results show that the SVM classifier based on *word2vec* performed the best with a Precision of 0.789, a Recall of 0.828, an F1-

score of 0.808, and an Accuracy of 0.803. We also found out that the *fastText* pre-trained word embedding can achieve relatively good results and help to improve the classifiers' performance. Furthermore, 200-dimensional pre-trained word embedding models outperform classifiers that use 100 and 300-dimensional models. Finally, an ensemble classifier based on the majority voting strategy can further enhance the classifier and outperform the individual classifiers.

We then conducted an online survey to validate the effectiveness of automatic identification of violation symptoms in practice. We received responses from 21 participants who were involved in the discussion of the collected review comments concerning architecture violations. The survey results show that the practitioners' perception of the classifiers' results confirms the promising applications of automatic identification of violation symptoms in practice. Especially, such classifiers can enable and inspire the participants to find, prioritize, and handle architecturally-related violation issues.

The main difference from previous studies that focused on utilizing tools to identify architectural issues by source code analysis, is that our work focuses on exploring the identification of violation symptoms through textual artifacts. The **main contributions** of this work are the following:

- This is the first work to explore the application of ML and DL-based classifiers regarding the automatic identification of violation symptoms in code review comments.
- We compared the performance of several trained classifiers using popular ML and DL-based algorithms, as well as the impact of three pre-trained word embeddings on classification performance.
- We gained insights from participants regarding the usefulness of ML and DL-based classifiers in identifying architecture violations during code review.

The remainder of this chapter is organized as follows. Section 6.2 introduces background information. Section 6.3 describes the research questions, study design, and experimental setup. The answers to the research questions are presented in Section 6.4. We discuss the implications of the study results in Section 6.5 and the threats to validity in Section 6.6. Section 6.7 reviews related work, and Section 6.8 concludes this work with future directions.

6.2 Background

In this section, we introduce a) the background of architecture erosion and its symptoms; b) the modern code review process in Gerrit, as this is used in our dataset.

6.2.1 Architecture Erosion and Related Symptoms

In the past decades, a wide variety of studies are concerned with the architecture erosion phenomenon, which has been extensively discussed and described with various terms (Li, Liang, Soliman and Avgeriou, 2021b, 2022), such as architecture decay (Hassaine et al., 2012; Le et al., 2018), degradation (Lenhard et al., 2019; Herold, 2020), degeneration (Hochstein and Lindvall, 2005).

Architecture erosion manifests in various symptoms during development and maintenance. A symptom is a partial sign or indicator of the emergence of architecture erosion (Le et al., 2016; Li, Liang, Soliman and Avgeriou, 2022; Li, Soliman, Liang and Avgeriou, 2022). According to a recent mapping study (Li, Liang, Soliman and Avgeriou, 2022), erosion symptoms can be classified into four categories (i.e., structural symptoms, violation symptoms, quality symptoms, and evolution symptoms).

Previous studies investigated different symptoms of architecture erosion. Mair *et al*. (Mair and Herold, 2013) proposed a formalization method regarding the process of repairing eroded architecture by finding violation symptoms and recommending optimal repair sequences. Le *et al*. (Le et al., 2016, 2018) regarded architectural smells as structural symptoms and they provided metrics to detect instances of architecture erosion by analyzing the detected smells. Martin (Martin, 2000) deemed that the evolution symptoms of architecture erosion include rigidity (a tendency to resist changes), fragility (a tendency to break frequently when modifications occur), immobility (inability to be reused), and viscosity (reduced effectiveness and efficiency due to design or environment issues).

As mentioned in the Introduction section, we focus on the most direct erosion symptoms of architecture erosion (i.e., violation symptoms). Table 6.1 presents several examples of violation symptoms of architecture erosion from the used dataset (see Section 6.3); a detailed comparison of this work and related work is given in Section 6.7.

6.2.2 Code Review in Gerrit

Code review is a crucial software development activity that involves the systematic examination of assigned code to identify defects and improve software quality (Bacchelli and Bird, 2013). A methodical code review process not only enhances the quality of software systems, but also facilitates the sharing of development knowledge and prevents the release of unstable and defective products. Over time, code review practices have become increasingly important, have been widely adopted in modern software development, and have received extensive tool support. In fact,

Table 6.1: Examples from our dataset that includes data from Nova, Neutron, Qt Base, and Qt Creator

Project	Example of violation symptoms of architecture erosion
Nova	*"But here don't we have to make a upcall from compute to api db, which will violate api/cell isolation rules. Is there any workaround in this case?"*
	"We could get some race conditions when starting the scheduler where it would not know the allocation ratios and would have to call the computes, which is a layer isolation violation to me."
Neutron	*"Having said all of that: I get that I'm violating an abstraction layer in LinkLocalAddressPair and that this is surprising (and therefore bad)."*
	"As you pointed out, since this patch is about the router availability zone it seems like a layer violation doing it in the mech driver."
Qt Base	*"This one seems to be a layering violation: why does QNetworkRequest understand QHttpNetworkConnection?"*
	This breaks the abstraction layer of publicQtConfig and is annoyingly verbose. instead, publicQtConfig should have an optional parameter 'name' to override the built-in 'feature' member.
Qt Creator	*"The unit tests are designed to not depend on any qt creator library to make the dependency breaking easier. Sometimes it works but it is not designed that way."*
	"This looks like we break the design. Why do we can not hold a pointer to the plugin?"

tool-based code review has become the norm in both industry and open-source communities, with a variety of available code review tools, such as Me's Phabricator[1], VMware's Review-Board[2], and Gerrit[3].

The two communities (i.e., OpenStack and Qt), from which the four open-source software (OSS) projects were selected in this work, use Gerrit as the code review tool. Figure 6.1 illustrates the modern code review process in Gerrit, which is a well-known code review tool and widely used in modern code review (Davila and Nunes, 2021). Gerrit allows for efficient collaboration on code changes before they are merged into the main code base. Once a developer commits new code changes (e.g., software patches) and their descriptions to Gerrit, the system creates a page to record all changes and relevant descriptions (e.g., commit messages). Then, the

[1] https://www.phacility.com/
[2] https://www.reviewboard.org/
[3] https://www.gerritcodereview.com/

system conducts a sanity check to verify whether the patch is compliant and to ensure that the code does not contain obvious compilation errors. After the submitted patch passes the sanity check, code reviewers manually examine the patch and provide comments to correct any potential errors, followed by giving a voting score. The comments are subsequently used by the developers to improve the patch and submit it for another review. Finally, the submitted patch is merged into the code repository after passing integration tests, which confirms that there are no issues or conflicts. Alternatively, the patch can be abandoned if the voting is negative.

The above process is used, among others, in revisions that enable the analysis of potentially significant architectural changes. Code review comments provide software developers a *finer granularity* for investigating architectural changes and violations in their daily development routines (Li, Qi, Yu, Liang, Mo and Yang, 2021). Meanwhile, developers provide reasoning and rationale for the changes they make in code review comments, which enables code review data to be a valuable source of knowledge for explaining changes (Paixao et al., 2019), offering insights into developers' concerns, suggestions, and potential violations.

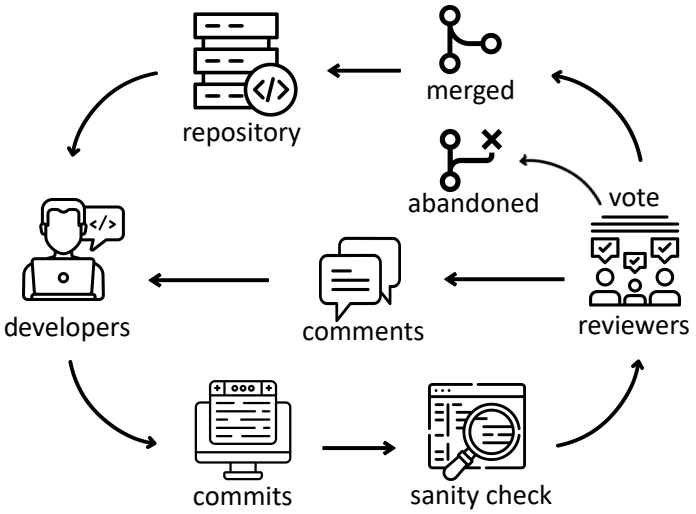

Figure 6.1: Code review process in Gerrit

6.3 Study Design

This section describes the research questions that motivated this research, followed by a detailed explanation of the data collection and the ML and DL models to auto-

matically identify architecture violations from textual artifacts. Finally, we provide details of the validation survey conducted to assess the effectiveness of automatic identification of violation symptoms in practice.

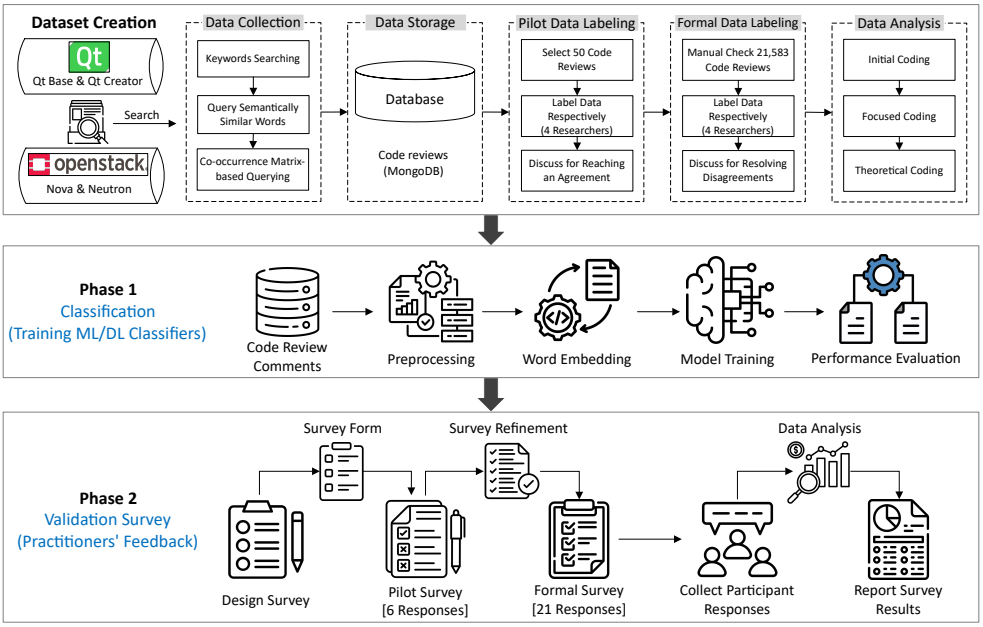

Figure 6.2: An overview of the research process

Figure 6.2 provides an overview of the whole process. As illustrated in this figure (see the top box, **Dataset Creation**), we collected data and established a dataset on architecture violations in our previous work (Li et al., 2023d). During **Phase 1**, we explored the possibility of using this dataset to train ML/DL-based classifiers to automatically identify violation symptoms in code review comments. During **Phase 2**, we sent the survey questionnaire to 200 involved participants and received 21 responses that were used to validate the usefulness of automatic identification of violation symptoms in practice.

6.3.1 Research Questions

In this work, we aim to explore the feasibility of automatically identifying violation symptoms of architecture erosion from code review comments. Specifically, we formulated the goal of this study by following the Goal-Question-Metric method (Basili et al., 1994): **analyze** *code review comments* **for the purpose of** *exploring the feasibility*

of automatically identifying violation symptoms of architecture erosion **with respect to** *the performance on architecture erosion identification* **from the point of view of** *developers* **in the context of** *open source software development*. To achieve our goal, we formulate three Research Questions (RQs):

RQ1: What is the performance of the classifiers in identifying violation symptoms from code review comments?

- **RQ1.1: Which classifier performs best in identifying the violation symptoms?**
- **RQ1.2: Which word embedding models can help to improve the performance of classifiers?**
- **RQ1.3: To what extent, does the dimensionality of word embedding models impact the performance of classifiers?**

Rationale: This RQ focuses on training feasible classifiers to automatically identify violation symptoms from textual artifacts. As mentioned in Section 6.1, the percentage of violation symptoms (belonging to architecture-level issues) is relatively lower than code-level issues. We thus expect that the description of violation symptoms is scattered in textual artifacts during development. RQ1.1 focuses on finding out the classifier with the best performance to distinguish violation symptoms from natural language by comparing several popular ML and DL algorithms. RQ1.2 aims at employing several well-known word embedding models to generate word vectors as features of labeled data, in order to choose the word embedding models that can help to improve the accuracy of classifiers. The word embedding models are a common ground for both ML and DL-based classifiers in our study. Finally, with RQ1.3, we seek to compare the differences in dimension size among different word embedding models, as it is one of the most important hyperparameters affecting the model performance (Hutter et al., 2019). Specifically, RQ1.3 endeavors to shed light on the extent to which the selection of dimensionality impacts the performance of the classifiers in automatically identifying violation symptoms.

RQ2: To what extent, a voting strategy can help to improve the performance of identifying violation symptoms from code review comments?

Rationale: After training individual classifiers based on ML and DL algorithms, we want to know whether, and to what extent, an ensemble classifier composed of trained classifiers can help to improve the performance of identifying violation

symptoms from code review comments. To answer this RQ, we compared the performance of individual classifiers with that of a combined classifier. More specifically, we employed a voting strategy to integrate the prediction values of each classifier and generated results after voting among individual classifiers. Answering this RQ can help to explore the possibility of further improving the performance of automatic identification of violation symptoms.

RQ3: Do practitioners find automatic identification of violation symptoms useful in practice?

Rationale: If the results generated by the classifiers based on ML and DL algorithms are acceptable regarding the effectiveness of identified violation symptoms, automatically identifying violation symptoms from textual artifacts would be helpful in practice. Such automated techniques could facilitate tasks such as locating architecture violations and inspiring further investigation and remediation of similar architectural issues. Through this RQ, we aim to conduct a survey to investigate how practitioners (both developers and reviewers) assess the effectiveness and practicability of the trained classifiers.

6.3.2 Data Collection

We relied on the dataset that we mined and manually identified in our previous study (Li et al., 2023d). As shown in Figure 6.2 (see **Dataset Creation**), we followed five steps to establish a specific dataset on architecture violations mined from code review comments. More specifically, we mined code review comments of four OSS projects in the OpenStack (i.e., Nova and Neutron) and Qt (i.e., Qt Base and Qt Creator) communities between 2014 and 2020. We mined the data through the REST API[4] supported by Gerrit. Then, through a series of tasks, including keyword search, manual identification, and pilot and formal data labeling of review comments related to architecture violations, we manually identified and labeled 606 code review comments related to architecture violations; code review comments were created for patches, and comments were made by code reviewers and developers. In terms of the data labeling process, four researchers first conducted a pilot data labeling by randomly selecting 50 review comments, to reach a consensus (the inter-rater agreement between the researchers measured in the Cohen's Kappa coefficient value (Cohen, 1960) was 0.857) and ensure that we have the same understanding of violation symptoms. Any disagreements were discussed between the

[4] https://gerrit-review.googlesource.com/Documentation/rest-api.html

four researchers to reach a consensus. After that, the four researchers started the formal data labeling by dividing the retrieved 21,583 code review comments into four parts (each researcher manually labeled around 5,400 comments). After the formal data labeling, the author checked the data labeling results from the other three researchers to make sure that there were no false positives. To mitigate potential bias, the four researchers discussed all the conflicts in the labeling results until we reached an agreement. To summarize, all data labeling results were checked by at least two researchers and conflicts were discussed among four researchers. Subsequently, all the collected code review comments regarding architecture violations from the four projects were combined into an integrated dataset used in this work. The dataset has been made available in a replication package (Li et al., 2023c).

6.3.3 Phase 1 - Automatic Classification of Violation Symptoms

In this section, we describe the experimental setup that we followed to evaluate the performance of the trained classifiers (see Figure 6.3). The steps in this setup correspond to the steps of Phase 1 in Figure 6.2. First, we pre-processed the collected code review comments to generate structured word sets, and then utilized word embedding techniques to encode the words and generate vectors for presenting the words in review comments. Subsequently, the generated vectors acted as the input to train the classifiers based on ML/DL models, in order to learn to classify review comments as violations and non-violations. The experimental environment is a computer equipped with Intel(R) Core(TM) i7-10510U CPU and 16GB RAM, running Windows 11 (64-bit). The next subsections elaborate on these steps for both the ML and DL models. More details of the experiments (such as hyperparameters and experimental environment) can be found in the replication package (Li et al., 2023c).

6.3.3.1 Machine Learning Models

Before training ML models, we need to pre-process our collected data, which involves transforming raw data into a structural format for model training. The **pre-processing** includes five steps listed as follows:

Step 1. Tokenization: The process of tokenization is to break a stream of text into words, punctuation, and other meaningful elements called *tokens*. In this work, we tokenized the review comments by splitting the text into its constituent set of words.

Step 2. Noise Removal: Developers usually use plain text when they discuss violation symptoms during code review, which contain various noise data such as punctuation, numbers, and special characters (e.g., "\", "*"). Noise data usually

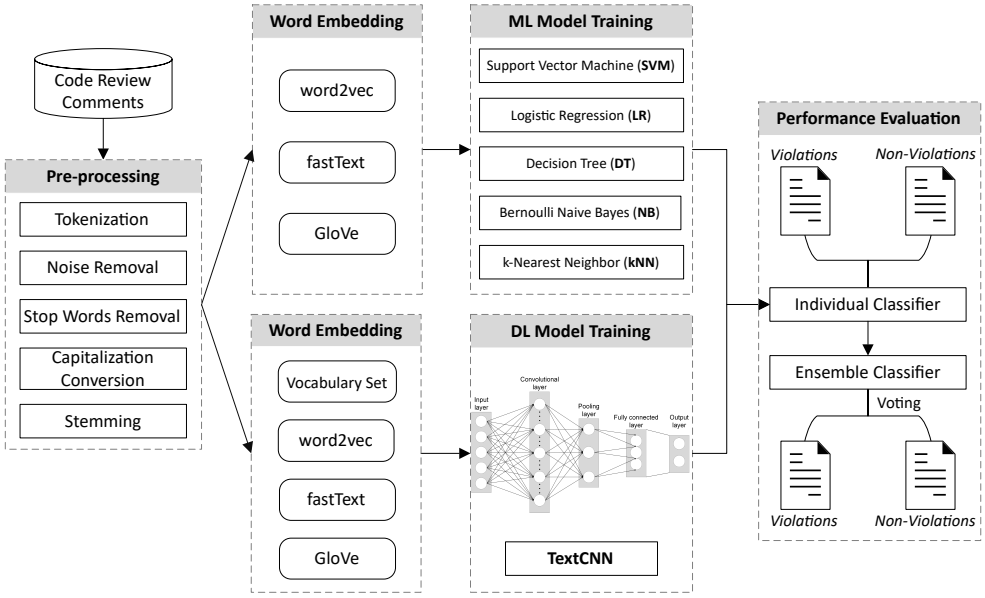

Figure 6.3: The framework of the experimental setup for classifying violation symptoms of architecture erosion from code review comments

does not contain valuable semantic information, and we therefore removed it.

Step 3. Stop words Removal: Stop words are frequently observed in text, for the purpose of distinguishing between texts, such as "the", "are", and "is", but they are typically devoid of valuable semantic content. Thus, we removed stop words in our dataset by using the Natural Language Toolkit (NLTK) (Bird et al., 2010).

Step 4. Capitalization Conversion: In order to ensure the consistency of word form and to avoid redundant counts of words, all textual data was converted to lowercase.

Step 5. Stemming: Stemming can be used to convert the words to their base forms. For instance, "architecture" and "architectural" have the same stem (i.e., token) "architectu". We conducted a stemming process (using Snowball Stemmer from the NLTK toolkit (Bird et al., 2010)) to obtain the stem of each token, since it is a more comprehensive and aggressive version of the Porter Stemmer[5].

Word Embedding. Feature extraction techniques are widely utilized to transform unstructured text sequences into structured feature spaces (e.g., matrices, vectors, or encodings). In this work, we employ word embedding techniques to extract textual features for generating interpretable input for machine learning algorithms.

[5]https://www.geeksforgeeks.org/snowball-stemmer-nlp/

Word embedding is a prominent method employed in natural language processing to represent words through dense vectors. In this work, we utilized the three prevalent and popular techniques (i.e., word2vec, fastText, and GloVe) to generate word embeddings. These techniques have gained widespread use in machine learning and deep learning-based tasks, owing to their efficacy in capturing the semantic and syntactic relationships among words.

- *word2vec*: word2vec was developed in 2013 (Mikolov et al., 2013) and it takes words as input and provides a word embedding matrix that contains high-dimensional unique vectors to represent each distinct word in given training corpora. word2vec can generate word embeddings through skip-gram and Continuous Bag-of-Words (CBOW) models. In this study, we adopted the pre-trained word2vec model proposed by Efstathiou *et al*. (Efstathiou et al., 2018a). This model was trained based on the corpus comprising over 15 GB of textual data collected from Stack Overflow posts, which contains a plethora of textual information in the software engineering domain.

- *fastText*: fastText is an open-source and lightweight library proposed by Meta AI Research lab in 2017 (Bojanowski et al., 2017) and it is capable of learning text representations and training text classifiers. Based on the fastText model, Meta published pre-trained word vectors with 300 dimensions for 157 languages constructed on Common Crawl and Wikipedia[6]. In this study, we employed the pre-trained fastText model, which can be transformed into 100 and 200-dimensional models, in order to control variables and avoid the impact of dimensionality on classifiers' performance.

- *GloVe*: Global Vectors for Word Representation (GloVe) is another common word embedding technique (Pennington et al., 2014). GolVe model is inspired by aggregated word co-occurrence statistics that may encode global information for words. Compared with word2vec, GloVe avoids the weakness regarding using local word co-occurrence information. In this study, we adopted the GloVe pre-trained word vectors on Twitter data (two billion tweets) with 200 dimensions.

Model Training. Given the binary classification task in this study, we selected five common ML algorithms that are widely used in classification tasks (Han, Pei and Tong, 2022), including Support Vector Machine (SVM), Logistic Regression (LR), Decision Tree (DT), Bernoulli Naive Bayes (NB), and k-Nearest Neighbor (kNN). These classification algorithms are also commonly applied in many areas of software engineering studies (Yang et al., 2022).

[6] `https://fasttext.cc/docs/en/crawl-vectors.html`

For the ML classifiers, we performed a *grid search* to tune the hyperparameters of each model and picked the best-performing ML models with specific hyperparameters on the validation set. Grid search, an exhaustive search technique, explores every possible combination within a predefined search space to identify the optimal configuration. More specifically, this method is widely employed in the realms of machine learning and optimization and it involves a systematic evaluation that examines all of the combinations of a set of candidate settings to find the best hyperparameter combination (Hutter et al., 2019).

6.3.3.2 Deep Learning Models

Similarly to the ML models, for the DL models we pre-processed the collected data before training DL classifiers, using the same **pre-processing** steps described in Section 6.3.3.1. For the pre-processing of DL classifiers, we needed to ensure the equal size of each paragraph length; therefore, we utilized the zero-padding strategy before dimension transformation to equalize all input text (i.e., the maximum length is set to 2000 words).

Model Training. In recent years, DL-based models are increasingly used in various text classification tasks and Convolutional Neural Network (CNN) has become one of the most popular model architectures for text classification (Minaee et al., 2021). We selected the TextCNN model to train the DL-based classifiers, since TextCNN, as an adapted CNN-based model, is reported to improve the state of the art on text classification (Minaee et al., 2021). TextCNN is a CNN-based architecture designed for processing textual data, and was introduced by Yoon Kim in 2014 (Kim, 2014). It can be used for a variety of Natural Language Processing (NLP) tasks, such as text classification and sentiment analysis. Compared with traditional ML models regarding text processing, the key advantage of TextCNN is its ability to automatically learn features from raw textual data, without relying on manual feature engineering.

We built the TextCNN for identifying violations and non-violations in code review comments through several processing layers, including an input layer, a convolutional layer, a pooling layer, a fully connected layer, and an output layer. Specifically, the TextCNN model applies one layer of convolutions over n-dimensional word embeddings to capture n-gram features at different scales. The convolutional layer learns to detect local patterns in the input text. The resulting feature maps are then fed through one max-pooling layer, which extracts the most salient features from each map and concatenates them to produce a fixed-length vector representation of the input text. This vector is then passed through one or more fully connected layers to produce the final output, i.e., the predicted labels for the input review com-

ments.

TextCNN is a convolutional neural network model that is commonly used for text classification tasks. This model takes a sentence as input and applies multiple convolutional filters to extract important features from the text. A TextCNN classifier based on the vocabulary set of a dataset uses a one-hot encoding scheme to represent words and generate the word embedding, while a TextCNN classifier can also utilize pre-trained word embedding models to capture semantic information of words. Additionally, the TextCNN model requires a proper setting to construct neural network models, such as hyperparameters and word embedding parameters. In this work, we set the embedding dimension to 200, the learning rate to 0.001, and the dropout to 0.25 to handle the overfitting, while the filter region size is (3, 4, 5). Each DL model's batch size is set to 16, and the network is trained for 100 epochs with the early stopping strategy with the purpose of avoiding overfitting. We configured the patience parameter to 8, indicating that the training process would terminate after 8 consecutive epochs with no observed improvement in model performance. These approaches are common practices in machine learning, aimed at preventing overfitting and optimizing the model's convergence (Watson et al., 2022).

6.3.3.3 Performance Evaluation

For the performance evaluation of ML/DL classifiers, we adopted four commonly used metrics, namely, Precision, Recall, F1-score (i.e., the harmonic mean of Precision and Recall), and Accuracy.

Precision presents the proportion of correctly classified violation symptom comments to all comments classified as violation symptoms. **Recall** represents the proportion of all violation symptom comments that are correctly identified. **F1-score** is the harmonic mean between precision and recall, assessing if an increase in one compensates for a decrease in the other. **Accuracy** is the percentage of correct classifications by a trained learning model. The above metrics are defined and calculated by the following equations.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{6.1}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{6.2}$$

$$\text{F}_1 - \text{score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Presion} + \text{Recall}} \tag{6.3}$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{6.4}$$

Here, we need to use four statistics to calculate the metrics, namely, False Positives (FP) represent the number of non-violation comments that are classified as violation comments; False Negatives (FN) represent the number of violation comments that are classified as non-violation comments; True Positives (TP) represent the number of violation comment that are classified as violation comments; True Negatives (TN) represent the number of non-violation comments that are classified as non-violation comments.

To avoid the class imbalance problem (He and Garcia, 2009), a data distribution problem that significantly impacts classifiers' performance, we constructed our dataset with a balanced class distribution (i.e., the same number of review comments labeled as "*violations*" and "*non-violations*"), which avoids the poor generalization ability of classifiers. With respect to the model training and validation strategy, we employed 10-fold cross-validation when we conducted a performance evaluation of the classifiers. We divided the collected dataset into three parts, i.e., we randomly set aside 80% of the data for the training set (60%) and validation set (20%), and 20% for the test set. Besides, we set a fixed seed for reproducibility during experiments.

6.3.4 Ensemble Classifier

To identify architecture violations from textual artifacts (i.e., code review comments), we trained several ML and DL-based classifiers to predict the labels of new review comments. However, the trained classifiers have varying performances in predicting the labels of target artifacts. In RQ2, we asked whether composing the individual classifiers into an ensemble classifier can help improve the overall performance. This composition can be done by utilizing the ensemble learning technique, i,e., majority voting strategy (Dietterich, 2000). Specifically, a combined classifier of several individual classifiers is generated with the purpose of building an improved classifier, which predicts an output (class) based on the highest probability of chosen class as the output (Dietterich, 2000). This aggregating criterion is like an election process and can produce combined voting results from each classifier's output.

The voting criteria include *Hard Voting* (voting is calculated on the predicted output class) and *Soft Voting* (voting is calculated on the predicted probability of the output class). To reduce the training time, we employed the *Hard Voting* (a simple majority voting strategy) strategy in this work. In other words, the final predicted results depend on the majority output (i.e., labels) of the individual classifiers.

6.3.5 Phase 2 - Validation Survey

Survey Design. To answer RQ3, we designed and executed a survey by following the guidelines proposed by Kitchenham and Pfleeger's personal opinion surveys (Kitchenham and Pfleeger, 2008); we used an anonymous survey to increase the response rate.

Surveys can be conducted in different ways, such as online questionnaires, phone surveys, or interviews (Lethbridge et al., 2005). Our survey is an online questionnaire of the *cross-sectional* type, which allows us to collect data and compare differences between participants at a single point in time. We adopted self-administered questionnaires to collect responses, since (1) respondents can answer the survey at their convenience (reducing our time investment and increasing the response rate); (2) questionnaires can reduce research bias since the researcher does not have any contact with the respondents when they fill out the survey; and (3) respondents enjoy better privacy than interviews.

We invited participants through customized emails (see the replication package (Li et al., 2023c)) attaching the links to their code review comments that contain violation symptoms and are identified by our approach, as well as the survey form (also see the replication package (Li et al., 2023c)). Our survey was constructed based on four parts: *Problem Statement*, *Our approach*, *Survey Goal*, and *Survey Questions*. *Problem Statement* introduces the research background regarding what is architecture erosion and its violation symptoms, as well as why it is important to investigate violation symptoms. *Our approach* describes the approach that we developed to automatically identify violation symptoms with high performance from a large number of code reviews as well as two examples of violation symptoms of architecture erosion detected by our approach from code review comments. In *Survey Goal*, we explain the purpose of this survey. In *Survey Questions*, we ask five general questions and present six statements concerning the usefulness of our approach. Note that, before the formal survey, we did a pilot survey (receiving 6 responses) with the purpose of revising and improving our survey questions and examples.

The five general survey questions include demographic information (e.g., their countries and work experience). The six statements are assessed using a five-point Likert scale (Strong Agree, Agree, Neutral, Disagree, and Strongly Disagree) with one option ("I don't know"). This scale enables participants to rate the extent that they agree or disagree with the statements. Finally, at the end of the survey, we ask an open-ended question to let the respondents freely provide comments and suggestions about our survey. To formulate the statements, we took inspiration from the survey questions of two survey studies (Nasab et al., 2021; Khalajzadeh et al., 2022) that evaluate the usefulness of automatic classifiers based on ML and DL techniques

in the software engineering community. We chose to use declarative statements in our validation survey since declarative statements are deemed appropriate for exploratory studies to convey simple and clear judgements; in contrast, descriptive statements are more suitable in explanatory studies to provide the details of the judgements in various aspects. The statements are as follows:

- Statement 1. "*Violation symptoms in code review comments identified by the approach, represent potential architectural violations*".
- Statement 2. "*Violation symptoms in code review comments identified by the approach can be used to improve the code quality*".
- Statement 3. "*I (as a practitioner) can find potentially constructive and useful architectural information from violation symptoms in code review comments identified by the approach*".
- Statement 4. "*Violation symptoms in code review comments identified by the approach can help us identify violation-related issues faster than if we do this manually*".
- Statement 5. "*Violation symptoms in code review comments identified by the approach might help us locate, identify, and prioritize potential architectural violations in our systems*".
- Statement 6. "*Violation symptoms in code review comments identified by the approach can provide us with input to find other violations of similar nature or otherwise*".

Participants. Our collected data from Gerrit contains the developers' information (i.e., names and email addresses); this information is not included in the replication package for privacy reasons. We invited participants to fill out our survey by sending emails to the developers whose comments are identified by our classifiers. During the pilot survey, we got responses from 6 developers. The pilot survey helped us to evaluate and refine our survey design. For example, we added more precise examples of violation symptoms from code review comments in our survey in order to enhance practitioners' understanding, and we refined the content of the customized emails based on the received feedback from the pilot survey participants. Then, we invited the remaining developers involved in the discussions on violation symptoms to fill in the survey.

Sample and Population. The target population is limited to developers whose code review comments are identified by our approach. We used a Non-Probabilistic Sampling method in this survey, namely, Convenience Sampling. As a type of non-probability sampling method, Convenience Sampling can help us to conveniently

reach a suitable number of developers who are willing to participate in our survey. The reason we chose this sampling method is that we could not intentionally choose a sample due to the limitations of geographical constraints and scheduling conflicts.

Data Analysis. We applied descriptive statistics to analyze the responses to the demographic and Likert scale questions. For the optional open question, we used the open coding and constant comparison method (Adolph et al., 2011) to analyze and categorize the qualitative response.

6.4 Results

6.4.1 RQ1: Identifying Violation Symptoms

To answer RQ1, we trained five ML classifiers and one DL classifier based on three word embedding techniques (i.e., word2vec, fastText, and GloVe). Moreover, we evaluated the performance of these classifiers and compared the difference between their performance.

RQ1.1: Performance of Classifiers

For ML-based classifiers, as shown in Table 6.2, we trained the five types of ML classifiers with three word embedding techniques (i.e., word2vec, fastText, and GloVe), leading to 15 classifiers (i.e., the five ML models times three word embeddings). We calculated the average values of the classifiers over the three pre-trained word embeddings. For each classifier based on three word embeddings, we underlined the best result of each metric (horizontal comparison); for each metric of the classifiers (precision, recall, F1, and accuracy), we marked the best metric result of each classifier on three word embeddings in bold (vertical comparison). From the results, we can see that the kNN classifiers have better scores of Precision than other classifiers with an average Precision of 0.816. As for the remaining metrics, the SVM classifiers achieve relatively better performance on nearly all metrics than other classifiers, with an average Recall of 0.801, F1-score of 0.779, and Accuracy of 0.773. Specifically, the SVM classifier based on *word2vec* performed the best with a Precision of 0.789, a Recall of 0.828, an F1-score of 0.808, and an Accuracy of 0.803.

For DL-based classifiers, as shown in Table 6.6, we trained one type of classifier, i.e., TextCNN, based on the vocabulary set from our dataset and three word embedding techniques (i.e. in total four classifiers) as mentioned in Section 6.3.3.1. We can see that TextCNN_FT performs the best compared with the other three DL models.

The ML-based classifiers demonstrated superior performance, achieving, on average, a Precision of 0.753, a Recall of 0.667, an F1-score of 0.700, and an Accuracy

Table 6.2: Performance comparison among ML-based classifiers

Classifier	Precision				Recall				F1-score				Accuracy			
	w2v	FT	GloVe	**Average**	w2v	FT	GloVe	**Average**	w2v	FT	GloVe	**Average**	w2v	FT	GloVe	**Average**
SVM	0.789	0.728	0.759	0.759	**0.828**	**0.746**	**0.828**	**0.801**	**0.808**	0.737	**0.792**	**0.779**	**0.803**	0.734	**0.783**	**0.773**
LR	0.832	0.777	**0.798**	0.802	0.730	0.713	0.713	0.719	0.777	**0.744**	0.753	0.758	0.791	**0.754**	0.766	0.770
NB	0.728	0.743	0.717	0.729	0.615	0.615	0.623	0.618	0.667	0.673	0.667	0.669	0.693	0.701	0.689	0.694
DT	0.667	0.657	0.643	0.656	0.689	0.721	0.664	0.691	0.677	0.688	0.653	0.673	0.672	0.672	0.648	0.664
kNN	**0.863**	**0.788**	0.797	**0.816**	0.516	0.549	0.451	0.505	0.646	0.647	0.576	0.623	0.717	0.701	0.668	0.695
Average	0.776	0.739	0.743	0.753	0.676	0.669	0.656	0.667	0.715	0.698	0.688	0.700	0.735	0.712	0.711	0.719

w2v: word2vec, FT: fastText

Table 6.3: Performance comparison of the classifiers based on the fastText model with different dimension size

Classifier	Precision			Recall			F1-score			Accuracy		
	100-dim	200-dim	300-dim	100-dim	200-dim	300-dim	100-dim	200-dim	300-dim	100-dim	200-dim	300-dim
SVM	0.745	0.728	0.657	0.672	0.746	0.754	0.707	0.737	0.702	0.721	0.734	0.680
LR	0.704	0.777	0.745	0.664	0.713	0.648	0.684	0.744	0.693	0.693	0.754	0.713
NB	0.745	0.743	0.712	0.623	0.615	0.607	0.679	0.673	0.655	0.705	0.701	0.680
DT	0.641	0.657	0.612	0.615	0.721	0.648	0.628	0.688	0.629	0.635	0.672	0.619
kNN	0.831	0.788	0.831	0.525	0.549	0.443	0.643	0.647	0.578	0.709	0.701	0.676
TextCNN	0.577	0.728	0.494	0.648	0.549	0.730	0.585	0.667	0.461	0.586	0.672	0.492

Table 6.4: Performance comparison of ensemble classifiers

Classifier	Precision					Recall					F1-score					Accuracy				
	Mean	Best	Voting	Imp_M	Imp_B	Mean	Best	Voting	Imp_M	Imp_B	Mean	Best	Voting	Imp_M	Imp_B	Mean	Best	Voting	Imp_M	Imp_B
SVM	0.759	0.789	0.795	4.74%	0.76%	0.801	0.828	0.828	3.37%	0.00%	0.779	0.808	0.811	4.11%	0.37%	0.773	0.803	0.807	4.40%	0.50%
LR	0.802	0.832	0.833	3.87%	0.12%	0.719	0.730	0.738	2.64%	1.10%	0.758	0.777	0.783	3.30%	0.77%	0.770	0.791	0.795	3.25%	0.51%
NB	0.729	0.743	0.738	1.23%	-0.67%	0.618	0.615	0.623	0.81%	1.30%	0.669	0.673	0.676	1.05%	0.45%	0.694	0.701	0.701	1.01%	0.00%
DT	0.656	0.657	0.689	5.03%	4.87%	0.691	0.721	0.746	7.96%	3.47%	0.673	0.688	0.717	6.54%	4.22%	0.664	0.672	0.705	6.17%	4.91%
kNN	0.816	0.863	0.836	2.45%	-3.13%	0.505	0.516	0.500	-0.99%	-3.10%	0.623	0.646	0.626	0.48%	-3.10%	0.695	0.717	0.701	0.86%	-2.23%
TextCNN	0.574	0.728	0.537	-6.45%	-26.24%	0.664	0.549	0.713	7.38%	29.87%	0.542	0.667	0.613	13.10%	-8.10%	0.563	0.672	0.549	-2.49%	-18.30%

Table 6.5: Performance comparison among ensemble ML-based classifiers with three word embeddings

Classifier	Precision					Recall					F1-score					Accuracy				
	Mean	Best	Voting	Imp_M	Imp_B	Mean	Best	Voting	Imp_M	Imp_B	Mean	Best	Voting	Imp_M	Imp_B	Mean	Best	Voting	Imp_M	Imp_B
ML_word2vec	0.776	0.789	0.810	4.38%	2.66%	0.676	0.828	0.697	3.11%	-15.82%	0.715	0.808	0.749	4.76%	-7.30%	0.735	0.803	0.766	4.22%	-4.61%
ML_fastText	0.739	0.777	0.798	7.98%	2.70%	0.669	0.713	0.713	6.58%	0.00%	0.698	0.744	0.753	7.88%	1.21%	0.712	0.754	0.766	7.58%	1.59%
ML_GloVe	0.743	0.759	0.768	3.36%	1.19%	0.656	0.828	0.705	7.47%	-14.86%	0.688	0.792	0.735	6.83%	-7.20%	0.711	0.783	0.746	4.92%	-4.73%

of 0.719. In contrast, the DL-based classifiers exhibited lower performance, with an average Precision of 0.574, Recall of 0.664, F1-score of 0.542, and Accuracy of 0.563. These findings suggest that the ML-based classifiers are more effective in accurately classifying the target variable than the DL-based classifiers. Overall, among these classifiers, the SVM classifiers outperform other classifiers, exhibiting the best average results on all metrics, except for the Precision metric.

Table 6.6: Performance comparison among DL-based classifiers

Classifier	Precision	Recall	F1-score	Accuracy
TextCNN_voc	0.475	0.541	0.469	0.471
TextCNN_w2v	0.584	0.680	0.596	0.598
TextCNN_FT	**0.728**	0.549	**0.667**	**0.672**
TextCNN_GloVe	0.507	**0.885**	0.434	0.512
Average	0.574	0.664	0.542	0.563

voc: vocabulary set, w2v: word2vec, FT: fastText

RQ1.2: Comparison of Embedding Models

From the results in Tables 6.2 and 6.6, we can also observe the differences between the classifiers based on three pre-trained word embedding models. As mentioned before, for each classifier based on three word embeddings, we underlined the best result of each classifier on each metric (horizontal comparison). When using the *word2vec* pre-trained word embedding, most of the ML classifiers have relatively better performance on Precision and Accuracy. Most of the ML classifiers with the *fastText* model have fairly better F1-scores. Besides, we found that the TextCNN classifier based on the *fastText* pre-trained word embedding achieves the best performance on all metrics except for Recall, with a Precision of 0.728, a Recall of 0.549, an F1-score of 0.667, and an Accuracy of 0.672.

In addition, we can see in Table 6.6 that the overall performance of the three TextCNN classifiers with pre-trained word embeddings outperforms the TextCNN classifier based on the vocabulary set from our dataset. This is because pre-trained word embedding models contain more features due to the large textual data used for generating word vectors, and they have been proven to be invaluable for improving the performance in NLP tasks (Qi et al., 2018; Radford et al., 2018).

In general, the three pre-trained word embedding models have an impact on the

performance of the DL-based classifiers. As shown in Table 6.6, when we compared their classification performance, in most cases, the classifiers with the *fastText* model can achieve relatively good performance.

RQ1.3: Comparison of Word Embedding Model Dimensionality

According to the results of RQ1.2, the *fastText* model can help to generate better results than the other two pre-trained word embedding models. Therefore, to further explore the impact of dimension sizes of the pre-trained word embedding models, we trained the classifiers based on the *fastText* model and we chose three common dimension sizes: 100, 200, and 300. In Table 6.3, similarly as before, we underlined the best result of each classifier on each metric (horizontal comparison). We can see that five of the six ML/DL classifiers can generate better F1-scores based on the *fastText* model with 200-dimensional word embedding, compared with 100 and 300-dimensional models.

RQ1 Summary. The SVM classifier trained on the pre-trained *word2vec* word embedding can achieve the best performance on identifying violation symptoms, with an F1-score of 0.779. In most cases, the classifiers with the *fastText* pre-trained word embedding model can achieve relatively good performance. Furthermore, 200 as the dimension size of the pre-trained word embedding models can generate better performance than 100 and 300-dimensional models.

6.4.2 RQ2: Improving Performance with Voting

To answer RQ2, we employed an ensemble learning technique, i.e., majority voting strategy, to build an ensemble classifier. We compared the performance of the individual classifiers with the ensemble classifiers. More specifically, we explored to what extent combining one type of classifier based on three word embedding techniques can achieve better performance, as shown in Table 6.4 and Table 6.5.

In Table 6.4, the "*Mean*" column presents the **Average** values of each metric of each classifier on three word embeddings from Table 6.2 and Table 6.6. The "*Best*" column denotes the corresponding classifier with the best performance from Table 6.2 and Table 6.6. The "*Voting*" column refers to the performance of the ensemble classifier (based on classifiers with three word embeddings) on each metric after utilizing a majority voting strategy. For example, the *Voting* values of SVM classifiers in Table 6.4 refer to the voting results among the three individual classifiers SVM_w2v, SVM_FT, and SVM_GloVe. "*Impro_M*" and "*Impro_B*" represent the improved results

of the corresponding ensemble classifiers compared with the results in "*Mean*" and "*Best*" columns, respectively.

According to Table 6.4, when we utilized the voting strategy for each ML classifier based on the three word embeddings (i.e., word2vec, fastText, GloVe), we can see that all ensemble classifiers perform better than the mean values of individual classifiers, with F1-score improvement from 0.48% to 13.10%. Even compared with the best-performing classifiers, most of the ensemble classifiers outperform the individual ML/DL classifiers, except for kNN. Moreover, the ensemble TextCNN of the three TextCNN classifiers based on pre-trained word embeddings (see Table 6.6) does not perform as well as TextCNN_FT (i.e., the best one) on Precision, F1-score, and Accuracy. This means that TextCNN_FT with the *fastText* pre-trained word embedding performs best compared with the other three TextCNN classifiers, and the voting strategy does not have an obvious improvement for the trained DL classifiers.

In addition, as shown in Table 6.5, we utilized the voting strategy for the five ML classifiers based on each of the three word embeddings. For example, ML_word2vec is the ensemble classifier of the five ML classifiers based on the *word2vec* word embedding (i.e., SVM_w2v, LR_w2v, NB_w2v, DT_w2v, and kNN_w2v). Similarly as before, the "*Mean*" column refers to the average metric results of ML classifiers (see the last row in Table 6.2); the "*Best*" column presents the best-performing classifiers in Table 6.2; the "*Voting*" column denotes the performance results of the ensemble classifiers after voting. It is clear that the three ensemble classifiers outperform the average performance results of the individual classifiers, with improvements from 3.11% to 7.98%. Even compared with the best-performing individual classifiers (i.e., the maximum values of the voting samples), the ensemble classifier ML_fastText can still demonstrate performance improvement on all metrics.

Moreover, we can observe the performance difference of ensemble classifiers with three word embeddings. In general, for the selected five ML models, the *fastText* pre-trained word embedding model can achieve relatively better performance improvements on almost all the metrics compared with the models using *word2vec* and *GloVe*. This finding affirms a similar observation in the result of RQ1.2.

RQ2 Summary. In comparison to individual ML classifiers, the ensemble classifiers achieve better performances after utilizing the majority voting strategy. However, no significant improvement was observed in the performance of the DL classifiers on our dataset when implementing the ensemble technique.

6.4.3 RQ3: Validation in Practice

As elaborated in Section 6.3.5, we designed and conducted a survey to solicit the perceptions from the practitioners on the usefulness of our trained models to automatically identify violation symptoms from code review comments.

Overall, we planned to send customized survey invitation emails to 200 code reviewers who were involved in the discussion of the collected review comments concerning architecture violations. However, we could only derive valid email addresses for 169 of them. We finally obtained 21 valid responses to our survey after sending 169 survey invitation emails and reminder emails; the response rate of 12.4%, is a reasonable response rate considering that the general response rate is around 5% in empirical software engineering research (Singer et al., 2008). Table 6.7 shows the demographic information of the 21 respondents. Nineteen of the respondents have more than 10 years of experience in software development, while two of them have 6-10 years of experience. Among the respondents, 52.4% (11) are software engineers, 33.3% (7) are developers, and 14.3% (3) are architects. Note that, *software engineers* usually have a broader work range than *developers*; for example, the former may work on full-stack development while the latter can deal only with the back end. Regarding the size of their companies, eight of the respondents reported that their companies have more than 1000 employees, five respondents worked in companies with employees between 501 and 1000, five respondents in companies with employees between 101 and 500, and three respondents in companies with employees between 21 and 100. Besides, the respondents worked in a variety of application domains.

Figure 6.4 presents the feedback of the 21 respondents on the usefulness of the trained models, i.e., the automatic identification of violation symptoms from the textual content of code review comments. According to the results, most of the respondents (61.9%) agreed that the violation symptoms identified by our models represent potential architectural violations (Statement 1). 38.1% respondents agreed that the identified violation symptoms can be used to improve the code quality (Statement 2), while the same percentage of respondents had a neutral attitude about this statement. 47.6% of the respondents agreed that the identified violation symptoms contain potentially constructive and useful architectural information (Statement 3).

Statements 3 to 6 investigate whether the trained classifiers can play an auxiliary role in development and code reviews. Approximately 42.9% respondents strongly agreed (14.3%) or agreed (28.6%) that the identified violation symptoms can help identify violation-related issues faster than manual identification (Statement 4); this indicates that the classifiers may help to speed up code review. Moreover, the majority (57.2%) of the respondents strongly agreed (4.8%) or agreed (52.4%) that au-

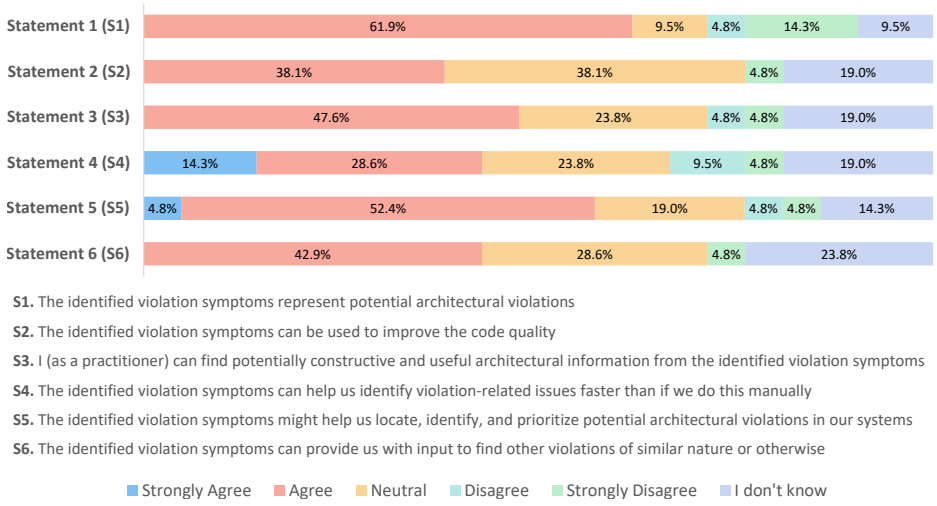

Figure 6.4: Practitioners' responses to statements regarding the usefulness of the trained models

tomatic identification can help locate and prioritize architectural violation issues (Statement 5). Furthermore, 42.9% of the respondents agreed that the identified violation symptoms can inspire them to find out other (or similar) violation symptoms (Statement 6). Notably, one respondent strongly disagreed with all six statements, indicating that he/she might not believe that ML and DL-based classifiers can be useful for identifying architectural violations in practice; we did not treat this response as an outlier. Overall, Figure 6.4 shows that positive feedback (strongly agree + agree) surpasses negative (disagree + strongly disagree) and neural feedback in most of the statements, except for Statement 2.

As for the feedback on the open-ended question, three participants expressed their concerns about false positives. For example, one participant pointed out: "*How useful the approach will depend a lot on the false-positive/false-negative rates, ..., an effective approach might be to make 'check for architecture violations' an explicit and mandatory part of the code review process*".

RQ3 Summary. Practitioners' responses indicate the usefulness of the automatic identification of violation symptoms discussed in code review comments in practice. Besides, the identified violation symptoms can enable and inspire practitioners to find architecturally-related violations and prioritize violation-related issues.

6.5 Discussion

6.5.1 Interpretation of Results

RQ1: Identifying Violation Symptoms

Our trained ML/DL classifiers can identify violation symptoms in code review comments. The results of RQ1.1 indicate that certain factors such as the type of classifier, the pre-trained word embeddings, and hyperparameters influence the performance of classifiers. Specifically, we found that the SVM classifier with *word2vec* outperforms other ML and DL-based classifiers. We also found that DL classifiers do not always have better performance than ML classifiers; we note that training DL classifiers takes much longer time than training ML classifiers, as DL classifiers have much more complex network architecture and more parameters. In recent years, some studies that leveraged DL-based techniques for software engineering tasks have achieved better performance that all other state-of-the-art techniques (e.g., ML-based techniques) (Yang et al., 2022). However, the achieved results imply that ML classifiers might be enough for certain text classification tasks with *small-size datasets*, especially as the trained DL classifiers do not show performance improvement in such a scenario. This finding is also observed in previous studies (Liu et al., 2018; Fu and Menzies, 2017): for certain software engineering tasks, simple ML-based approaches are capable of achieving equal or even better performance than DL-based approaches within less time. Thus, we advise researchers to not blindly apply DL-based techniques and consider simple approaches first.

Finding 1: *ML-based approaches have sufficient performance for many software engineering tasks, such as identification of architecture violations in code reviews.*

From the results of RQ1.2, we can see that the TextCNN classifier based on the *fastText* pre-trained word embedding can perform better than other DL models. We thus emphasize that the *fastText* pre-trained technique has a better capacity for training well-performed DL models; this also aligns with the finding in the work of Sesari *et al*. (Sesari et al., 2022), that is, the *fastText* pre-trained word embedding model has less bias compared to *word2vec* and *GloVe*.

Besides, the results of RQ1.3 show that 200-dimensional pre-trained word embedding models outperform 100 and 300-dimensional models. Our results are slightly different from previous studies (e.g., Ren et al. (2019); Li, Soliman and Avgeriou (2022)) where the 300-dimensional word embedding model has better performance. One possible reason is that the previous studies only measured the average

Table 6.7: Demographic information of the survey respondents

Country	Experience	Role	Company size	Company domain
Ireland	More than 10 years	Software Engineer	employees $\geq$ 1000	HCM services
Germany	More than 10 years	Developer	21 $\leq$ employees $\leq$ 100	Industrial electronics
Finland	More than 10 years	Software Engineer	21 $\leq$ employees $\leq$ 100	Medical software
Norway	More than 10 years	Architect	501 $\leq$ employees $\leq$ 1000	Industrial DataOps
Israel	More than 10 years	Software Engineer	employees $\geq$ 1000	Network
Germany	6 - 10 years	Developer	101 $\leq$ employees $\leq$ 500	Development tools
USA	More than 10 years	Developer	501 $\leq$ employees $\leq$ 1000	Iaas
Norway	More than 10 years	Developer	501 $\leq$ employees $\leq$ 1000	Cross-platform libraries for application development
Norway	More than 10 years	Developer	101 $\leq$ employees $\leq$ 500	SW development frameworks
USA	More than 10 years	Software Engineer	employees $\geq$ 1000	Software development
Germany	More than 10 years	Developer	101 $\leq$ employees $\leq$ 500	Middleware
USA	More than 10 years	Software Engineer	employees $\geq$ 1000	Research
Germany	More than 10 years	Software Engineer	employees $\geq$ 1000	Game development
Norway	More than 10 years	Software Engineer	501 $\leq$ employees $\leq$ 1000	Development framework
Finland	More than 10 years	Architect	21 $\leq$ employees $\leq$ 100	Medical devices
Germany	More than 10 years	Developer	501 $\leq$ employees $\leq$ 1000	Tools and frameworks for software development
Germany	More than 10 years	Software Engineer	101 $\leq$ employees $\leq$ 500	Defence development
Denmark	6 - 10 years	Software Engineer	101 $\leq$ employees $\leq$ 500	Network
India	More than 10 years	Architect	employees $\geq$ 1000	Development framework
USA	More than 10 years	Software Engineer	employees $\geq$ 1000	Tools for software development
Canada	More than 10 years	Software Engineer	employees $\geq$ 1000	Framework tool

results of classifiers with different dimensional word embedding models, and the average results might be influenced by the individual results. Our results present and compare the classification performance of each ML/DL classifier, which could be more accurate.

Finding 2: *The 200-dimensional fastText pre-trained word embedding model can be used to train classifiers with better performance when conducting binary classification of violation symptoms from code review comments*.

In general, due to a large number of hyperparameters of DL classifiers, we did not specifically focus on the best combination of hyperparameters of the TextCNN classifier, but certain key hyperparameters such as the dimension size of word embedding model. Thus, future work can consider performing hyperparameter tuning (e.g., utilizing the grid search technique) to explore other possible combinations of the hyperparameters to further improve performance.

RQ2: Improving Performance with Voting

Evaluating the performance of an ensemble classifier offers insight into the performance improvement compared with individual classifiers. For voting-based approaches, an accurate output is attained only when a majority of the constituent models generate the correct output (Hansen and Salamon, 1990). Our results provide further empirical evidence for the efficacy of the majority voting strategy regarding improving the performance of text classification. This technique effectively **combines the predictions of multiple classifiers to arrive at a final decision, resulting in improved overall classification performance**. Nevertheless, we also observed that such performance improvement is not obvious in TextCNN in our case. One possible reason is that we only get an ensemble DL classifier based on three individual TextCNN models, and that causes the voting results to be particularly susceptible to the influence of single results. We argue that the voting strategy remains valuable to minimize potential prediction errors if there are more outputs for voting.

Finding 3: *Compared with individual classifiers for identifying architecture violations from code review comments, ensemble learning is an effective way for improving classification performance. In most cases, ensemble classifiers that utilize the voting strategy can enhance all classification performance metrics*.

RQ3: Validation in Practice

The existence of architectural violations can be challenging, as they can impede practitioners' understanding of architecture during the development process. This particularly influences novice developers who lack practical familiarity with architecture. The results of RQ3 indicate a predominantly positive feedback, confirming the usefulness of our trained ML/DL classifiers in potentially helping developers during maintenance and evolution. Through our analysis of the received responses to the statements (e.g., Statements 3 and 5), we found that developers expressed a significant interest in utilizing automated checking approaches, tools, or plug-ins to identify violation symptoms during code review, recognizing their potential to enhance the productivity of dealing with architectural violations through code review. The survey results indicate that developers hold positive attitudes towards such tools. Moreover, such automated techniques regarding identifying potential architecture violations could be supplementary means to minimize architecture violations and improve software quality during code review. For example, such classifiers could provide warnings to reviewers based on the discussions of reviewed code, nudging them to prioritize (e.g., offering warnings to refactor or remove) potential violation symptoms that they might have ignored before based on various factors (e.g., the importance of involved components), and prompt reviewers to address violation symptoms.

Moreover, according to the feedback from respondents, certain practitioners might hold a conservative attitude toward ML and DL-based classifiers for identifying violation symptoms (see the "*Neutral*" results in Figure 6.4). One possible reason is that they might be skeptical about the performance of the classifiers (false positives or negatives), casting doubt about the reliability of identified architectural violations. Nevertheless, no classifier can ever achieve 100 % accuracy in classification tasks, and there is always a trade-off between precision and recall. This disclaimer should be made clear when showing classification tasks to practitioners. Another potential reason could be that certain identified review comments may be part of the architectural design *discussions* during the development process, rather than the final architectural decisions. Such concerns are beyond the target of our research, as the trained classifiers can only provide judgments based on the collected violations, instead of exploring the evolution of architecture violations.

Finding 4: *Although there are concerns about the correctness of the identified architecture violations, the positive responses from participants suggest that practitioners are receptive to the use of automated techniques for identifying architecture violations during code review.*

6.5.2 Implications

Establish and enrich datasets. Many studies utilized or proposed automated approaches to identify or predict knowledge-related information from textual artifacts. However, no studies focused on the automatic identification of violation symptoms from code review comments. This suggests that there is a need for more empirical reports and follow-up tools regarding such issues. Our study provides a starting point for researchers to automatically identify architecture violations in textual artifacts such as review comments. On top of this, we encourage researchers and practitioners to **establish and share datasets** related to architecture violations to lay a solid foundation for future research.

Moreover, our study is **scalable** in practice, as practitioners can retrain superior classifiers by adding more code review comments of violation symptoms from different projects, either manually or by using our classifiers. Besides, mining architecture violations from multiple textual artifacts (e.g., issues, source code comments, and pull requests) can provide a more comprehensive examination of architectural violations and also **enrich the dataset** used for identifying such violations. Furthermore, our findings regarding the experiments in this study might benefit or be adapted by researchers and practitioners to identify other issues, such as security and vulnerability discussions.

> **Suggestion 1**: *Researchers and practitioners are encouraged to establish and share datasets related to architecture violations. Besides, our study is scalable to retrain superior classifiers by adding more data from different sources, and can also be adapted to identify other issues*.

Choose appropriate classifiers. An ensemble classifier is a collection of classifiers that work together to classify instances by combining their individual outputs through voting. Researchers can consider exploring different selections of classifiers to further enhance the performance of ensemble classifiers. For example, to some extent, assigning varying weights to individual classifiers might partially mitigate the impact of biased classifiers on voting results.

> **Suggestion 2**: *It is recommended for researchers to explore various combinations of classifiers and assign different weights to individual classifiers to further enhance the classification performance of ensemble classifiers for identifying architecture violations*.

Take time investment into consideration. Even though our study did not specifically investigate the time efficiency of classifiers' training, we observed that the TextCNN classifiers exhibited a long training time due to a large number of parameters. In comparison to traditional ML classifiers, DL classifiers require a longer

training time, yet the results did not demonstrate a significant performance improvement. Therefore, when considering the practical implementation of classifiers, it is important to take into account the cost-benefit ratio of "Performance/Time". In situations where there is a limited dataset, traditional machine learning techniques may suffice in achieving the desired goals. Further research may explore additional methods for balancing the trade-off between performance and training time for classifiers.

Suggestion 3: *For certain software engineering tasks, constructing simple classifiers (e.g., individual or ensemble classifiers) can yield a superior return on investment compared to training complex DL-based classifiers, particularly when time investment needs to be taken into account.*

Optimize code review practice. In the future, researchers and practitioners can explore the development of plug-ins for code review platforms (e.g., Gerrit) that provide warnings of violations to developers or even suggestions to fix those violations. Such plug-ins can be developed utilizing historical code review data or harnessing the existing large language models through fine-tuning. These plug-ins have the potential to assist developers and maintainers in conducting comprehensive examinations of preexisting architectural violations, and may even serve as reminders of their presence. Additionally, implementing the trained classifiers (e.g., as an extension) into the existing tools (or toolsets) is also worthy of consideration in practice. For example, code review platforms can integrate such classifiers to empower the automated code review process. Such tools may provide more insights and help novices or maintainers troubleshoot architecturally-relevant violation issues.

Suggestion 4: *Researchers and practitioners can consider developing plug-ins for code review platforms (e.g., Gerrit) to assist developers in identifying potential architecture violations, or integrating the trained classifiers into existing tools to optimize the automated code review process.*

6.6 Threats to Validity

In this study, we discuss the potential threats to the validity related to our study and the adopted measures mitigating these threats by following the guidelines proposed by Wohlin *et al*. (Wohlin et al., 2012).

6.6.1 Construct validity

Construct validity concerns the connection between operational measures and the studied subjects. The first potential threat to construct validity is the suitability of our selected ML/DL algorithms, feature extraction techniques, and metrics. As mentioned by Peter *et al*. (Peters et al., 2017), it is not possible to use an exhaustive approach to test all learning models. We selected five common ML algorithms, one DL algorithm, and three popular pre-trained word embedding models, as they are all widely used to conduct text classification. We admit that using different algorithms and pre-trained word embedding models may generate varying results, despite their popularity. We cannot completely mitigate this threat. In addition, the four metrics (recall, precision, F1-score, and accuracy) are widely used to evaluate the performance of various automatic software engineering techniques (see Section 6.7.3). Therefore, the metrics pose little threat to the construct validity.

Another potential threat comes from the convenience sampling method of the survey, as individuals with negative views towards the research might be less inclined to participate in our survey. This is a compromise inherent to the convenience sampling method, and it implies that this threat cannot be completely mitigated. Nevertheless, to improve the response rate and collect as many opinions as possible from practitioners, we sent two rounds of reminder emails to the involved practitioners. It is also possible that some survey respondents did not understand the statements well. To reduce this threat to construct validity, we conducted a pilot survey for refining the survey design. Besides, we provided an "*I Don't Know*" option in our survey. Moreover, the survey respondents might have extra opinions besides the six defined statements in the survey. To mitigate this threat, we offered an optional open-ended question that allows respondents to freely share their thoughts with us.

6.6.2 External validity

External validity pertains to the generalizability of our experiment results and findings. We trained the ML and DL-based classifiers with our collected dataset regarding architecture violations. Selecting different datasets and experimental settings might influence the generalizability of (part of) our findings. We acknowledge that the chosen open-source projects are not representative of all projects from the OpenStack and Qt platforms, and our findings may not be generalized to diverse projects, particularly those from different domains such as closed-source projects or projects on other platforms like Apache. Furthermore, the survey participants are contributors to the selected projects, therefore, the findings of RQ3 might not

be generalized to other projects. To support further generalizability, we have made our dataset (Li et al., 2023c) available to enable other researchers and practitioners to replicate, reuse, and extend our study.

6.6.3 Reliability

Reliability reflects the replicability of a study regarding yielding the same or similar results when other researchers reproduce this study. One potential threat to reliability comes from the experimental settings (e.g., input data and hyperparameters for training classifiers) used in this work. We used the dataset from our previous study (Li et al., 2023d) that had publicly shared the labeled data and corresponding scripts for collecting data. Besides, we also shared the source code in our replication package (Li et al., 2023c) to facilitate reproduction and validation. Another potential threat might be from the survey design. To mitigate the threat, we provided the templates of our survey and customized emails in our replication package (Li et al., 2023c) to encourage further research in this area.

6.7 Related Work

In this section, we briefly discuss the current state of research in three associated areas of our study. We first describe why we focused on code review comments in Section 6.7.1; we then introduce previous studies on architecture violations in Section 6.7.2; we finally present the benefits of utilizing automated techniques in identifying artifacts in software development in Section 6.7.3.

6.7.1 Code Review Comments

Code review comments contain massive knowledge related to software development, and a variety of studies analyze software defects and evolution through mining review comments and commit records. Han *et al*. (Han, Tahir, Liang, Counsell, Blincoe, Li and Luo, 2022) conducted an empirical study by manually checking 25,415 code review comments to investigate code smells identified in modern code review from four OSS projects. Their results show that most reviewers provide constructive suggestions to help developers fix code, and developers are willing to repair the smells through suggested refactoring operations. El Asri *et al*. (El Asri et al., 2019) empirically investigated the impact of expressed sentiments on the code review duration and its outcome. They found that reviews with negative comments on average took longer time to complete than the reviews with positive/neutral

comments. Kashiwa *et al*. (Kashiwa et al., 2022b) reported Self-Admitted Technical Debt (SATD) during code review by checking 156,372 review records from OpenStack and Qt. They found that 28-48% SATD are introduced during code review, and 20% SATD comments are created because of reviewers' requests.

Besides, Uchôa *et al*. (Uchôa et al., 2020) investigated the impact of code review on the evolution of design degradation through mining and analyzing a plethora of code reviews from seven OSS projects. They found that there is a wide fluctuation of design degradation during the revisions of certain code reviews. Paixão *et al*. (Paixão et al., 2020) explored how developers perform refactorings in code review, and they found that refactoring operations are most often used in code reviews that implement new features. Besides, they observed that the refactoring operations were rarely refined or undone along the code review, and such refactorings often contribute to new code smells and bugs.

While the aforementioned studies investigated various aspects of code review regarding development and maintenance (e.g., decisions, design degradation, and SATD), there are no studies that investigated the automatic identification of violation symptoms in code review comments; we decided to fill this gap and explore the application of ML/DL techniques to specifically identify violation-related issues (e.g., violation symptoms) from code review comments.

6.7.2 Architecture Violations

Over the past decades, there have been various investigations on architecture violations. Brunet *et al*. (Brunet et al., 2012) performed a longitudinal study to explore the evolution of architectural violations in 19 bi-weekly versions of four open source systems. They investigated the life cycle and location of architecture violations over time by comparing the intended and recovered architectures of a system. They found that architectural violations tend to intensify as software evolves and a few design entities are responsible for the majority of violations. More interestingly, some violations seem to be recurring after being eliminated. Mendoza *et al*. (Mendoza et al., 2021) proposed a tool called ArchVID based on model-driven engineering techniques for identifying architecture violations from source code; this tool supports recovering and visualizing the implemented architecture.

Besides, Terra *et al*. (Terra et al., 2015) reported their experience in fixing architectural violations. They proposed a recommendation system ArchFix that provides refactoring guidelines for developers and maintainers to repair architectural violations in the module architecture view of object-oriented systems. Their results show that their approach can provide correct recommendations for 75% and 79% of the detected architecture violations on two selected systems, respectively. Maffort *et*

al. (Maffort et al., 2016) proposed an approach to check architecture conformance for detecting architecture violations based on defined heuristics. They claimed that their approach relies on the defined heuristic rules and can rapidly raise architecture violation warnings.

Different from the abovementioned studies focusing on detecting architecture violations in source code, our work explores the possibility of automatic identification of the violation symptoms from code review comments.

6.7.3 Analyzing Software Repositories with Machine Learning and Deep Learning

The application of ML and DL techniques to extract informative knowledge from various artifacts during software development is gaining popularity in the software engineering community. In recent years, a plethora of empirical studies have been conducted to investigate automated techniques for supporting software engineering tasks. Khalajzadeh *et al*. (Khalajzadeh et al., 2022) conducted an empirical study by extracting and manually analyzing app reviews from 12 OSS projects and developed ML and DL models to automatically detect and classify human-centric issues from app reviews and discussions. Their results show that automated techniques can help developers to recognize and appreciate human-centric issues of end-users more easily. Nasab *et al*. (Nasab et al., 2021) developed 15 ML and DL models to automatically identify security discussions, and then they collected practitioners' feedback through a validation survey and confirmed the promising applications of the models in practice.

AlOmar *et al*. (AlOmar et al., 2021) proposed an approach to automatically identify and classify self-admitted refactoring in commit messages. They compared their approach with pattern-based and simple random baselines, and the results show that a good performance and a relatively small training dataset is sufficient to classify self-admitted refactoring commits. Hey *et al*. (Hey et al., 2020) proposed a novel approach, named NoRBERT, for requirements classification by using the transfer learning capabilities of BERT. They applied their approach to different tasks in the domain of requirements classification, and the results show that their approach improves requirements classification and can be applied to unseen projects with convincing results.

In this work, word embeddings were utilized as word representations and as the input for training ML and DL-based classifiers, as prior studies have demonstrated the effectiveness of word embeddings in capturing rich semantic and syntactic features of words (Ren et al., 2019). We employed ML/DL-related techniques to promote the automatic identification of violation symptoms. Although much work has

been done on automatic text identification, there is currently no automatic way to identify architecture violations in code review comments. Our approach is not restricted to specific textual information, so future studies can replicate and extend our work with other artifacts.

6.8 Conclusions and Future Work

During code review, reviewers typically spend a substantial amount of effort in comprehending code changes, as significant information (e.g., architecturally relevant information) for inspecting code changes may be dispersed across several files that the reviewers are not acquainted with. Automatic identification of architecture violations from the discussions between reviewers and developers can save valuable time and effort to locate and check potential issues.

In this work, we attempted to address this issue of automatic identification of violation symptoms in code review comments. To this end, we first performed a series of experiments on a dataset of architecture violation discussions. Subsequently, to validate the usefulness of our trained ML and DL-based classifiers, we conducted a survey that acquired the feedback from the involved developers who discussed architecture violations in code reviews.

Specifically, we developed 15 ML-based and 4 DL-based classifiers to identify violation symptoms from developer discussions in code reviews (i.e., code review comments from four OSS projects in Gerrit). The results show that the SVM classifier based on *word2vec* pre-trained word embedding performs the best with an F1-score of 0.779. In most cases, classifiers with the *fastText* pre-trained word embedding model can achieve relatively good performance. Furthermore, 200-dimensional pre-trained word embedding models outperform classifiers that use 100 and 300-dimensional models. In addition, an ensemble classifier based on the majority voting strategy can further enhance the classifier and outperforms the individual classifiers. Moreover, practitioners' perception of the usefulness of the classifiers' results validates the promising applications of automatic identification of violation symptoms in practice. Especially, such classifiers can enable and inspire practitioners to find architecturally-related violation issues, prioritize, and eventually handle them.

In the future, we plan to conduct a more in-depth investigation of how practitioners deal with architecture violations in code review, what kind of practices and tools they use in dealing with architecture violations, and how using our approach in their daily job can support the identification of architecture violations. We also plan to further enhance the classifiers by identifying fine-grained types of architecture violations based on a larger dataset of more projects from both open-source and

industry projects. In particular, a bigger picture is that we plan to delve into a more in-depth investigation and comparison of the performance of trained classifiers as plug-ins or tools with fine-tuned large language models (such as Llama 2) using real-world scenarios. The goal of such plug-ins or tools for code review platforms is to provide warnings of potential architecture violations to practitioners (e.g., developers and reviewers) for them to re-check the related code changes during code review.

Based on:

Ruiyin Li, Peng Liang, Paris Avgeriou, (2023) *"Code Reviewer Recommendation for Architecture Violations: An Exploratory Study," in: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, Oulu, Finland, 2023, pp. 42–51: ACM. DOI:10.1145/3593434.3593450

Chapter 7

Code Reviewer Recommendation for Architecture Violations: An Exploratory Study

Abstract

Code review is a common practice in software development and often conducted before code changes are merged into the code repository. A number of approaches for automatically recommending appropriate reviewers have been proposed to match such code changes to pertinent reviewers. However, such approaches are generic, i.e., they do not focus on specific types of issues during code reviews. In this chapter, we propose an approach that focuses on architecture violations, one of the most critical type of issues identified during code review. Specifically, we aim at automating the recommendation of code reviewers, who are potentially qualified to review architecture violations, based on reviews of code changes. To this end, we selected three common similarity detection methods to measure the file path similarity of code commits and the semantic similarity of review comments. We conducted a series of experiments on finding the appropriate reviewers through evaluating and comparing these similarity detection methods in separate and combined ways with the baseline reviewer recommendation approach, RevFinder. The results show that the common similarity detection methods can produce acceptable performance scores and achieve a better performance than RevFinder. The sampling techniques used in recommending code reviewers can impact the performance of reviewer recommendation approaches. We also discuss the potential implications of our findings for both researchers and practitioners.

7.1 Introduction

Code review is widely employed in modern software development and is recognized as a valuable and effective practice at all stages of the development life cycle (Bacchelli and Bird, 2013). Active participation of developers in code review

decreases defects, improves the software quality, and facilitates knowledge sharing through rich communication among reviewers (Bacchelli and Bird, 2013; Ruangwan et al., 2019). Over the last decade, several tools have been widely used in both industry and open-source communities to make the code review process more effective, such as Phabricator[1], Review-Board[2], and Gerrit[3]. Although such tools provide automated techniques to support the code review process, there is still a significant amount of human factors that can influence code review activities, such as unqualified reviewers, response delays, and overloaded review workload (Chouchen et al., 2021; Balachandran, 2013; Ruangwan et al., 2019).

At the heart of the human-related issues lies the process of matching code to reviewers: authors who submit new code patches to a code review system, need to invite (or the system can assign) reviewers to manually check the uploaded code fragments based on the reviewers' expertise and past experience with reviews; this may be a labor-intensive and time-consuming task, especially for large projects (Çetin et al., 2021). Previous studies (Bosu et al., 2016b; Chouchen et al., 2021; Ruangwan et al., 2019) found that effective code review requires a significant amount of effort from reviewers who thoroughly understand the submitted code. However, inappropriate code reviewers might hinder the review process, delay the incorporation of a code change into a code base, and slow down the development process. Such problems arise from misunderstanding or simply lacking knowledge of the intention or effect of code changes (Dogan et al., 2019). A proper recommendation of code reviewers can help reduce delays and speed up development by finding appropriate reviewers who are more familiar with and spend less time reviewing the submitted code fragments (Thongtanunam et al., 2015; Balachandran, 2013).

There exist a number of code reviewer recommendation approaches in the literature (see Section 7.2.2). While these approaches can be effective, they are all generic in terms of the issues that reviewers focus. In this work, we focus on a particular type of issues: architecture violations. While architecture violations are one of the most frequently identified types of architecture issues during code review, they are not effectively covered by existing techniques and tools (Li, Soliman, Liang and Avgeriou, 2022). If code fragments with architecture violations are merged into the code base, it will increase the risk of architecture erosion (Li, Liang, Soliman and Avgeriou, 2022; Li, Soliman, Liang and Avgeriou, 2022) and gradually degrade architecture sustainability and stability (Venters et al., 2018).

The **goal** of this work is to offer an automated recommendation of code reviewers, who are potentially qualified to review architecture violations. More specifi-

[1] https://www.phacility.com/
[2] https://www.reviewboard.org/
[3] https://www.gerritcodereview.com/

cally, we aim at recommending potential code reviewers who have knowledge on architecture violations, through analyzing textual content of the review comments and the file paths of the reviewed code changes. Consequently, our approach is not limited to specific programming languages. This can act in a complementary way to a regular code review: a final check by reviewers who are knowledgeable in architecture violations can act as a quality gate to avoid code changes with architecture violations merged into the code base.

Our proposed approach is novel in terms of mining semantic information in review comments from code reviewers, based on common similarity detection methods. To validate our approach, we conducted a series of experiments on 547 code review comments related to architecture violations from four Open-Source Software (OSS) projects (i.e., Nova, Neutron, Qt Base, and Qt Creator). The results show that the employed similarity detection methods can produce acceptable performance scores (i.e., values of top-*k* accuracy and mean reciprocal rank metrics) and achieve a better performance than the baseline approach, RevFinder (Thongtanunam et al., 2015). We managed to further explore the performance of the proposed approach on our dataset, by using fixed sampling instead of incremental sampling. The main **contributions** of our work are:

- We explored the possibility of common similarity detection methods on recommending code reviewers who have awareness of architecture violations.
- We conducted experiments to evaluate and compare the performance of three similarity detection methods with the baseline approach RevFinder on four OSS projects.
- We shared the source code and dataset of our work (Li et al., 2023b) to encourage further research on code reviewer recommendation for architecture issues.

The remainder of this chapter is structured as follows: Section 7.2 describes the background regarding performing code review in Gerrit, code reviewer recommendation, and architecture violations. Section 7.3 elaborates on the research questions and study design. Section 7.4 presents the results of the research questions and discusses their implications. Section 7.5 clarifies the threats to validity and limitations of this study. Section 7.6 reviews the related work and Section 7.7 concludes this study with future directions.

7.2 Background

7.2.1 Code Review Process in Gerrit

Code review refers to the process of inspecting source code, which is a critical activity during development and can help to improve software quality (Bacchelli and Bird, 2013). The workflow of code review varies slightly between different platforms, e.g., the pull-request workflow in GitHub is different than code review in Gerrit. Gerrit is a commonly used platform for coordinating code review activities and facilitating traceable code reviews for git-based software development. In this work, we collected code review data from four OSS projects of two large communities OpenStack and Qt (see Section 7.3.2), both of which use Gerrit to conduct the code review process. We thus briefly elaborate on the code review process with Gerrit (see Figure 7.1).

A developer can submit new code patches or modify the original code fragments fetched from the repository through revisions; both take the form of commits. Gerrit then creates a page to record the submitted commits after conducting automated tasks, like a sanity check. Other developers of the project will be invited as reviewers to inspect the submitted commits and offer feedback (i.e., code review comments on the commits) to the developer. Such a review cycle will stop until the reviewers either approve the submitted code (status "*Merged*") or reject it (status "*Abandoned*"). In this process, we argue that one of the code reviewers should have awareness of architecture violations and provide a review on what they are and how to fix them. Such a code review that is focused on architecture violations would be complementary to regular code reviews.

7.2.2 Code Reviewer Recommendation

Expert recommendation is a common area in software engineering (Sülün et al., 2021), and code reviewer recommendation is a typical application of expert recommendation. Over the recent years, many approaches have been proposed for recommending code reviewers in the literature (Thongtanunam et al., 2015; Zanjani et al., 2015; Yu et al., 2016; Jiang et al., 2015; Rebai et al., 2020); these are briefly introduced below based on the categories in previous studies (Çetin et al., 2021; Lipcak and Rossi, 2018).

Heuristic-based approaches include problem-solving and practical methods. For example, the heuristic approaches, such as ReviewBot (Balachandran, 2013), cHRev (Zanjani et al., 2015), RevFinder (Thongtanunam et al., 2015), calculate

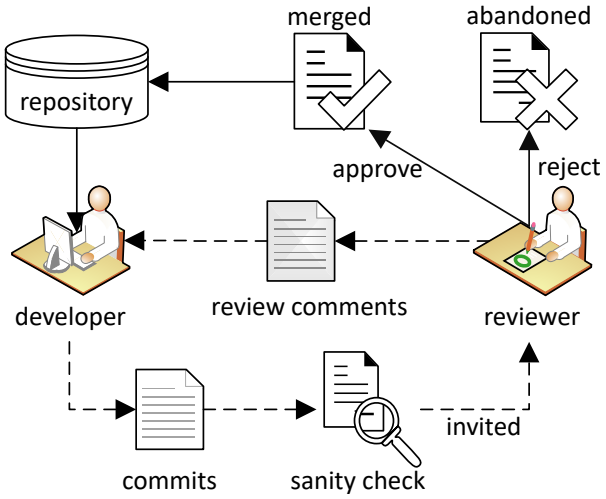

Figure 7.1: An overview of code review process in Gerrit

heuristic scores through building expertise models to measure candidate reviewers' expertise.

Machine learning-based approaches usually utilize data-driven machine learning techniques (e.g., Support Vector Machine (SVM) (Jiang et al., 2015)) and genetic algorithms (e.g., Indicator-Based Evolutionary Algorithm (IBEA) (Chouchen et al., 2021), NSGA-II (Rebai et al., 2020)) to recommend reviewers. Such approaches are based on a series of features, such as patches, bug report information.

Hybrid approaches combine different approaches (e.g., machine learning (Jiang et al., 2015), graph structure (Yu et al., 2016), genetic algorithm (Chouchen et al., 2021; Rebai et al., 2020)) for recommending reviewers. For example, Xia *et al*. (Xia et al., 2015) developed a hybrid incremental approach TIE (a Text mIning and filE location-based approach) to recommend reviewers through measuring textual content (i.e., multinomial Naive Bayes, a text mining technique) and file path similarity (i.e., a VSM-based approach).

Different approaches utilize different types of artifacts to recommend code reviewers. According to the recent literature review by Çetin *et al*. (Çetin et al., 2021), most of the studies use pull request history (e.g., changes lines, paths of changed files and titles), and some studies also use code review history (including comments made by pull requests and reviews). Our approach can be regarded as a heuristic-based approach, and is based on both the review comments and file paths of reviewed code changes. It differs from existing approaches as it focuses specifically on architecture violations.

7.2.3 Architecture Violations

During software evolution, architecture erosion can degrade the stability and sustainability of system architecture due to increasing changes and accumulated architecture violations (Li, Liang, Soliman and Avgeriou, 2022, 2021b; Venters et al., 2018). Architecture violations are the most common and prominent type of architecture erosion symptoms; various architecture violations have been investigated in the literature (Li, Liang, Soliman and Avgeriou, 2022). Architecture violations manifest in various ways: structural inconsistencies, violations of design decisions, violations of design principles, violations of architecture patterns, violations of API specification, etc. Previous studies on architecture violations have focused on analyzing history versions of source code. For example, Brunet *et al*. (Brunet et al., 2012) carried out a longitudinal study to analyze the evolution of 19 bi-weekly versions of four OSS projects, by examining the life cycle and location of architecture violations and comparing them to the intended architecture. Maffort *et al*. (Maffort et al., 2016) proposed an approach based on defined heuristics that can rapidly raise architecture violation warnings. In contrast to the studies focusing on detecting architecture violations in source code, we aim at finding reviewers who can review architecture violations during code review, regardless of the type of architecture (e.g., microservices, layered architecture).

7.3 Research Methodology

7.3.1 Research Questions

RQ1: Can common similarity detection methods be effectively used in recommending code reviewers for architecture violations?

Rationale: This study aims at proposing an approach for the automated recommendation of code reviewers who are knowledgeable on architecture violations. To this end, we propose to use similarity measurement, as this technique is commonly used to process textual artifacts like code reviews (Chouchen et al., 2021; Fejzer et al., 2018; Ouni et al., 2016b). With this RQ, we want to investigate whether common similarity measurement techniques (i.e., Jaccard coefficient, adapted Hamming distance, cosine similarity) can indeed be useful for recommending code reviewers based on the review comments and file paths of the reviewed code changes related to architecture violations (see Section 7.3.3). Specifically, we plan to evaluate

the performance of similarity detection methods using metrics widely adopted by the recommendation system community (Çetin et al., 2021), i.e., Top-*k* Accuracy and Mean Reciprocal Rank.

RQ2: How does the performance of the proposed similarity detection methods compare against existing code reviewer recommendation approaches?

Rationale: With this RQ, we want to compare the similarity detection methods with an existing approach using the artifacts that we collected in our study. Specifically, there are a number of code reviewer recommendation approaches (Lipcak and Rossi, 2018; Çetin et al., 2021; Sülün et al., 2021), which can be compared against the proposed approach. To be able to make the comparison, the source code of these approaches must be publicly available in order to reproduce them.

RQ3: Do the sampling techniques affect the performance of the proposed code reviewer recommendation approach?

Rationale: As mentioned in a recent literature review (Çetin et al., 2021), various sampling methods were used in code review recommendation. However, there are no studies that investigate whether sampling methods (see Section 7.3.3) can impact the performance of reviewer recommendation approaches, and which sampling techniques can achieve relatively better performance. By answering this RQ, we aim at providing empirical evidence about the influence of sampling techniques on reviewer recommendation performance.

7.3.2 Data Collection

Projects and Code Review Comments. The original dataset used in this study is from our previous work (Li et al., 2023d). Through a series of tasks (e.g., keywords search, manual identification and labeling of architecture violation related review comments), we collected 606 review comments on code changes and commit messages related to architecture violations. In our work, we focused on recommending reviewers who have awareness of architecture violations regarding code changes, since one of the purposes of reviewer recommendation is to help selecting reviewers for code changes. Therefore, we further extracted 547 review comments from the original dataset (Li et al., 2023d) that are only related to code changes on architecture violations (Li et al., 2023b). The dataset contains the code review comments

from four OSS projects, including Nova and Neutron from the OpenStack community[4], as well as Qt Base and Qt Creator from the Qt community[5]. As shown in Table 7.1, our dataset from the four OSS projects includes code review comments regarding architecture violations in eight years from June 2012 to December 2020. The review comments are related to various architecture violations (e.g., violations of design decisions, design principles, and architecture patterns), and were made by more than 200 reviewers. The items in the dataset contain review ID and patch information, including `change_id`, `patch`, `file_url`, `line`, and `comment`. The scripts and dataset of this work are available in (Li et al., 2023b).

Table 7.1: Details of the selected projects used in our work

Project	Time Period	Files[1]	Comments[2]	Reviewers[3]
Neutron	2013/11 - 2020/08	111	149	64
Nova	2013/01 - 2020/08	126	206	67
Qt Base	2012/12 - 2020/12	124	139	48
Qt Creator	2012/06 - 2020/11	49	53	25

[1] *Files: Code change files*
[2] *Comments: Code review comments on architecture violations*
[3] *Reviewers: Code reviewers of the code change files*

7.3.3 Recommendation Approach

Problem Statement. Since the artifacts we collected are from the OSS projects that use Gerrit as the code review tool, we take such projects as examples to formulate our approach. A software project S contains a set of m developers $D = \{d_1, \ldots, d_m\}$ and n code reviewers $R = \{r_1, \ldots, r_n\}$, and includes a set of j source code files $F = \{f_1, \ldots, f_j\}$ and a set of k code review comments $C = \{c_1, \ldots, c_k\}$. In general, R is used to represent a set of candidate code reviewers for code changes. Each reviewer r_i has their own expertise on certain source code file f_i, and has a review comment c_i on the corresponding commit.

In such a project, each new commit (i.e., code changes that are not yet merged into the code base) could be reviewed by a number of invited (or assigned) code reviewers. Our proposal is to generate a list of recommended reviewers, in which each prospective reviewer has a matched score representing their *expertise*. The higher the expertise score of a reviewer, the greater the probability for this reviewer to be

[4] `https://www.openstack.org/`
[5] `https://www.qt.io/`

recommended to review the commit. As mentioned in Section 7.2.2, our reviewer recommendation approach is based on the reviewer's *expertise*, which is extracted and calculated from historical commits (e.g., review comments and file paths) and commonly used in previous studies (Thongtanunam et al., 2015; Chouchen et al., 2021; Chen et al., 2022; Kong et al., 2022; Fejzer et al., 2018). The input of our recommendation approach is the past commit files, including file paths, review comments regarding architecture violations, and the corresponding reviewers.

Similarity Calculation. To answer RQ1 and present the *expertise* of reviewers, we chose cosine similarity, Jaccard coefficient, and adapted Hamming distance to measure the similarity of file paths and the semantic similarity of review comments on architecture violations.

In terms of the *similarity of file paths*, Jaccard coefficient and adapted Hamming distance are two common methods used to measure the similarity between file paths of code changes; they are considered efficient similarity measures and widely adopted in previous studies (e.g., Chouchen et al. (2021); Fejzer et al. (2018); Ouni et al. (2016b)). Jaccard coefficient is calculated to measure similarity, as shown in Equation (7.1):

$$\text{Jac\_Similarity}(X, Y) = \frac{|X \cap Y|}{|X \cup Y|} \tag{7.1}$$

where X and Y represent two entities whose similarity needs to be measured. Here, to measure the file path similarity, X and Y represent two file paths (i.e., the sets of tokens of file paths), and the more common tokens between the file paths X and Y, the higher similarity of the two file paths.

In addition, the similarity between file paths can be also calculated by the adapted Hamming distance (i.e., *similarity score = Hamming distance for the same length strings + difference in length of the two strings*). If two file paths have the same paths, then the similarity score returns 1, otherwise it returns the reciprocal score of the adapted Hamming distance of the two file paths.

In terms of the *semantic similarity of code review comments*, cosine similarity and Jaccard coefficient are used to measure the semantic similarity between code review comments. The two methods are often used in previous studies (e.g., Yu et al. (2016); Rahman et al. (2017)) to measure the semantic similarity of textual artifacts. Cosine similarity can be utilized to determine lexical similarity between two entities represented by two vectors of words. In our case, cosine similarity is used to measure the semantic similarity of review comments regarding architecture violations, as shown in Equation (7.2):

$$\text{Cos\_Similarity}(C_i, C_j) = \frac{v_i \cdot v_j}{|v_i|\,|v_j|} \tag{7.2}$$

where C_i and C_j represent two code review comments, and v_i and v_j denote their corresponding vectors. To generate vectors, we adopted a pre-trained Word2vec model, which was trained based on over 15 GB of textual data from Stack Overflow posts that contain a plethora of textual expressions and words in software engineering domain (Efstathiou et al., 2018b). The higher the similarity score, the closer the two vectors that represent the two review comments.

Regarding the Jaccard coefficient, as shown in Equation (7.1), X and Y represent two review comments in a set of tokens (i.e., tokenized words). The more common tokens between X and Y, the higher similarity score of the two review comments. Note that, before we calculated the semantic similarity by cosine similarity and Jaccard coefficient, we applied the following four pre-processing steps:

1. *Tokenization*. The process of tokenization is to break a stream of text into words, punctuation, and other meaningful elements called tokens.

2. *Noise removal*. Noise data usually does not contain valuable semantic information, and we therefore removed punctuation, numbers, and special characters (e.g., "\", "*");

3. *Stop words removal*. Stop words occur commonly but do not add valuable information to differentiate different text, such as "*the*", "*are*", and "*is*", which can be removed.

4. *Capitalization conversion*. We converted all the text to lower case, which can help to maintain the consistency of word form and avoid recounting the words.

Reviewer Recommendation. For a new code change that has been commented by reviewers but has not been merged into the code base, we aim at recommending code reviewers who are potentially aware of architecture violations through measuring the similarity of the file paths and review comments of the reviewed code changes.

We ranked the candidate code reviewers through calculating the reviewer scores using the file path similarity and the semantic similarity of historical commits (i.e., file paths and review comments of the reviewed code changes). This includes File Path similarity by Jaccard Coefficient (FP_JC), File Path similarity by adapted Hamming Distance (FP_HD), Review Comment semantic similarity by Cosine Similarity (RC_CS), and Review Comment semantic similarity by Jaccard Coefficient (RC_JC).

Given a new code change file f, we extracted its file path f_{new} and review comment c_{new}, and then calculated the above-mentioned similarity scores between the current code change and each past code change, including the past file path f_{past} and review comment c_{past}. For example, $FP\_JS(f_{new}, f_{past})$ calculates the file path similarity score between f_{new} and the file path f_{past} of a past code change by Jaccard coefficient. Similarly, $RC\_CS(c_{new}, c_{past})$ calculates the semantic similarity score between c_{new} and the review comment c_{past} of a past review comment by cosine similarity. Then, the scores are assigned to the associated reviewers, respectively. In other words, each reviewer has four candidate similarity scores by using the four similarity detection methods. By calculating the similarity scores in separate and combined ways, a reviewer recommendation list can be generated based on the sorted reviewers along with their scores.

Sampling and Validation. Sampling refers to the sampling techniques for constructing the *expertise* model, and validation denotes the process of testing the performance and effectiveness of certain sampling techniques. Unfortunately, most code reviewer recommendation studies did not provide detailed information and empirical validation on the sampling techniques for constructing their expertise models (Çetin et al., 2021), and it is nontrivial to explore the *effectiveness of the sampling and evaluation techniques* with the purpose of providing such empirical validation. Therefore, to answer RQ3, we planned to investigate whether and to what extent the sampling techniques can impact the performance of the proposed code reviewer recommendation approach. According to a recent literature review on code reviewer recommendation (Çetin et al., 2021), incremental sampling and fixed sampling are the two most popular sampling and validation techniques (see Figure 7.2), and have been commonly used in code reviewer recommendation studies (Çetin et al., 2021).

Since the code review data is temporal data, all prior studies organized their dataset chronologically (Çetin et al., 2021). Thus, we also sorted our dataset in a chronological order. The incremental sampling technique takes the historical review data as the input of the expertise model through increasing the sample number in each iteration (i.e., each step in Figure 7.2). The final performance of the recommendation approach is the average performance value of all the steps. In terms of the incremental sampling, we set four steps in this work, and we took 10% of the new sample as the validation set in each step. In terms of the fixed sampling, we employed a fixed percentage of the test set by randomly sampling with 10% in previous studies (e.g., Al-Zubaidi et al. (2020)) of the dataset of the four projects.

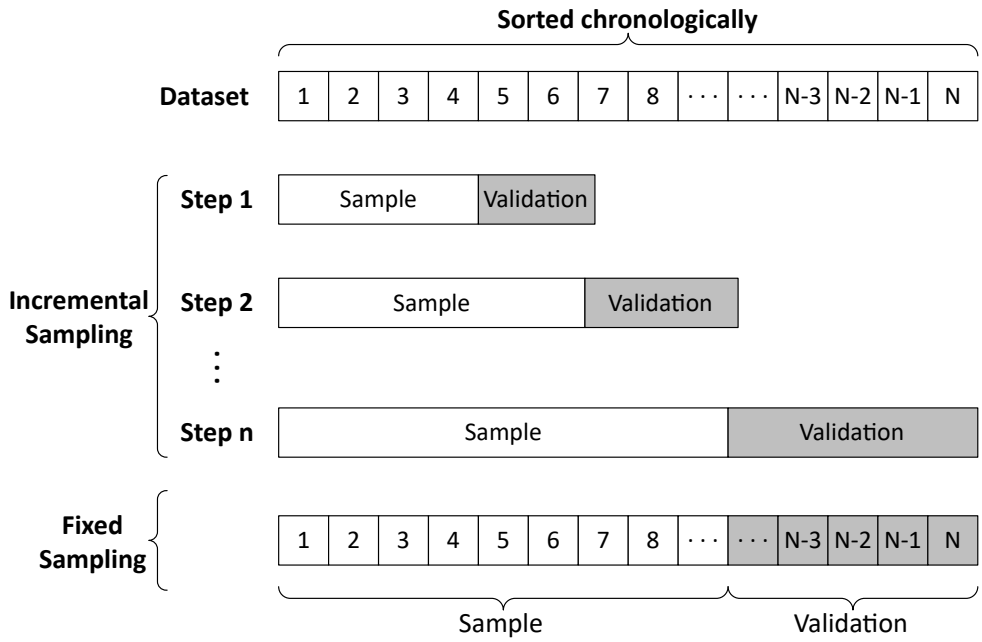

Figure 7.2: Overview of incremental sampling and fixed sampling

7.3.4 Baseline Approach

To answer RQ2, we needed to reproduce existing expertise-based approaches, such as (Yu et al., 2016; Kong et al., 2022; Thongtanunam et al., 2015; Chouchen et al., 2021), in order to compare them against our approach. However, we were only able to do that for one approach, namely RevFinder (Thongtanunam et al., 2015). The rest of the approaches had two main issues: (1) they require additional information that is not readily available (e.g., review workload); and (2) they do not make their source code or datasets available; some approaches did share parts of the source code, but still they can not be reproduced. This makes it difficult or even impossible to compare and evaluate our approach with these approaches.

Therefore, we reproduced one baseline approach, *RevFinder*, on our collected dataset. RevFinder is a file path-based approach, which is a specific expertise-based approach and supports recommending reviewers by measuring the file path similarity of commits. Specifically, when there is a new commit, developers who have reviewed or engaged in similar revisions (i.e., with similar file paths) are likely to be recommended. Previous studies (e.g., Kong et al. (2022); Chouchen et al. (2021);

Lipcak and Rossi (2018); Zanjani et al. (2015); Rebai et al. (2020)) using various artifacts (e.g., pull-requests, historical issues) usually compared their approaches with RevFinder (Thongtanunam et al., 2015), since file path is a common feature of various artifacts related to code review.

7.3.5 Evaluation Metrics

To evaluate the similarity detection methods and the baseline approach, we adopted two of the most prevalent metrics used in previous studies (Çetin et al., 2021): Top-*k* Accuracy and Mean Reciprocal Rank. We denote a code reviewer as *r* and a code reviewer set as *R*.

Top-*k* Accuracy measures the percentage of code reviews for which an approach can properly recommend the true code reviewers within the top-*k* positions in a ranked list of recommended code reviewers. In other words, this accuracy is regarding the ratio of the number of correctly recommended reviewer *r* (i.e., *isCorrect(r, Top-k)*) in the total number of reviewers of a ranked list of recommended reviewers. *isCorrect(r, Top-k)* returns 1 if there is at least one top-*k* reviewer *r* who actually reviewed the code, otherwise, *isCorrect(r, Top-k)* returns 0 which means a wrong recommendation. The higher the top-*k* accuracy value, the better the recommendation performance. By following the previous studies in Section 7.2.2, we set the *k* values of 1, 3, 5, and 10.

$$\text{Top-}k\text{ Accuracy} = \frac{1}{|\mathrm{R}|}\sum_{\mathrm{r}\in\mathrm{R}}\text{isCorrect}(\mathrm{r}, \text{Top-}k) \tag{7.3}$$

Mean Reciprocal Rank (MRR) calculates an average of reciprocal ranks of correct code reviewers in a recommendation list. Given a set of reviewers *R*, MRR can be calculated by Equation (7.4). *rank(r)* returns the value of the rank of the first correct reviewer in the recommendation list for reviewer *r*. The value of $\frac{1}{rank(r)}$ returns 0 if there is no one who actually reviewed the code in the recommendation list. Ideally, an approach that can provide a perfect ranking should achieve an MRR value of 1. Generally, the higher the MRR value, the better the recommendation approach is.

$$\mathrm{MRR}(\mathrm{R}) = \frac{1}{|\mathrm{R}|}\sum_{\mathrm{r}\in\mathrm{R}}\frac{1}{\mathrm{rank}(\mathrm{r})} \tag{7.4}$$

7.4 Results and Discussion

7.4.1 RQ1: Effectiveness of Our Approach

To answer RQ1, we evaluated the performance of the similarity detection methods for recommending code reviewers. We used two similarity detection methods to measure the similarity of file paths, as well as two similarity detection methods to measure the similarity of code review comments, that is, FP_JC, FP_HD, RC_CS, and RC_JC (see Section 7.3.3). Table 7.2 presents the performance of the similarity detection methods and their combinations.

Firstly, we evaluated the performance of the individual similarity detection methods, as shown in the four top rows per project in Table 7.2. The grey cells indicate the best performance metrics. The results show that the similarity detection methods yield varying results on different projects. For example, the performance of FP_JC and FP_HD on Neutron and Nova are better (with a higher top-*k* accuracy and MRR) than on Qt Base and Qt Creator. Secondly, we evaluated the performance of the combinations of similarity detection methods on the four projects (rows 6-17 per project in Table 7.2). For the combinations of two similarity detection methods, the mixed similarity detection method of FP_JC and FP_HD achieves the best performance on Neutron and Nova projects with 0.33 accuracy at top-10 recommendation, and the mixed similarity detection method of FP_JC and RC_JC achieves the best top-5 (i.e., 0.33, 0.19, 0.29, and 0.40) and top-10 accuracy (i.e., 0.33, 0.33, 0.43, and 0.40) on the four projects. For the combinations of three similarity detection methods, the mixed similarity detection method of FP_JC, FP_HD, and RC_JC gets the best accuracy and MRR on the four projects. In addition, we find that the performance of the combination of four similarity detection methods does not improve significantly when compared to the mixed approaches of three similarity detection methods.

Considering the **average** results of the similarity detection methods, we find that mixing three similarity detection methods can achieve a slightly higher top-*k* accuracy and MRR on Neutron, Nova, and Qt Base, and a significantly better performance on Qt Creator when compared to mixing two methods. However, combining four similarity detection methods has no obvious performance improvement. In general, the results show that combining three similarity detection methods can relatively get the best performance of top-*k* accuracy and MRR on the four projects.

Discussion of RQ1: The experiment results in Table 7.2 indicate that the selected similarity detection methods can produce acceptable performance scores (MRR values between 0.06 and 0.36) on code reviewer recommendation for architecture vi-

Table 7.2: Top-*k* (1, 3, 5, 10) accuracy and MRR results of the selected similarity detection methods on four OSS projects

Project	**Neutron**								**Nova**							
Similarity detection method	**Top-1**	**MRR**	**Top-3**	**MRR**	**Top-5**	**MRR**	**Top-10**	**MRR**	**Top-1**	**MRR**	**Top-3**	**MRR**	**Top-5**	**MRR**	**Top-10**	**MRR**
FP_JC	0.20	0.20	0.33	0.27	0.33	0.27	0.33	0.27	0.10	0.10	0.14	0.12	0.14	0.12	0.24	0.13
FP_HD	0.20	0.20	0.33	0.26	0.33	0.26	0.33	0.26	0.10	0.10	0.14	0.12	0.14	0.12	0.29	0.14
RC_CS	0.07	0.07	0.20	0.12	0.30	0.12	0.27	0.13	0.00	0.00	0.00	0.00	0.05	0.01	0.19	0.03
RC_JC	0.20	0.20	0.20	0.20	0.27	0.21	0.33	0.22	0.05	0.05	0.14	0.10	0.14	0.10	0.24	0.11
Average	0.17	0.17	0.27	0.21	0.31	0.22	0.32	0.22	0.06	0.06	0.11	0.09	0.12	0.09	0.24	0.10
FP_JC + FP_HD	0.27	0.27	0.33	0.29	0.33	0.30	0.33	0.30	0.10	0.10	0.14	0.12	0.14	0.12	0.29	0.14
FP_JC + RC_CS	0.20	0.20	0.20	0.20	0.27	0.22	0.27	0.22	0.00	0.00	0.00	0.00	0.00	0.00	0.19	0.02
FP_JC + RC_JC	0.27	0.27	0.33	0.26	0.33	0.29	0.33	0.29	0.05	0.05	0.05	0.05	0.19	0.08	0.33	0.09
FP_HD + RC_CS	0.13	0.13	0.13	0.13	0.27	0.16	0.27	0.16	0.00	0.00	0.00	0.00	0.05	0.01	0.19	0.03
FP_HD + RC_JC	0.00	0.00	0.07	0.03	0.07	0.03	0.20	0.05	0.00	0.00	0.05	0.02	0.19	0.06	0.24	0.06
RC_CS + RC_JC	0.00	0.00	0.07	0.03	0.07	0.03	0.20	0.05	0.00	0.00	0.05	0.02	0.10	0.03	0.19	0.04
Average	0.15	0.15	0.19	0.16	0.22	0.17	0.27	0.18	0.03	0.03	0.05	0.04	0.11	0.05	0.24	0.06
FP_HD + RC_CS + RC_JC	0.20	0.20	0.27	0.23	0.27	0.23	0.27	0.23	0.00	0.00	0.10	0.03	0.10	0.03	0.19	0.04
FP_JC + RC_CS + RC_JC	0.20	0.20	0.20	0.20	0.27	0.22	0.27	0.22	0.00	0.00	0.05	0.02	0.10	0.03	0.19	0.03
FP_JC + FP_HD + RC_JC	0.27	0.27	0.33	0.29	0.33	0.29	0.33	0.28	0.05	0.05	0.14	0.10	0.19	0.10	0.33	0.12
FP_JC + FP_HD + RC_CS	0.20	0.20	0.20	0.20	0.27	0.21	0.27	0.21	0.00	0.00	0.00	0.00	0.05	0.01	0.19	0.03
Average	0.22	0.22	0.25	0.23	0.29	0.24	0.29	0.24	0.01	0.01	0.07	0.04	0.11	0.04	0.23	0.06
FP_JC + FP_HD + RC_CS + RC_JC	0.20	0.20	0.20	0.20	0.27	0.22	0.27	0.22	0.00	0.00	0.10	0.03	0.10	0.03	0.24	0.05

Project	Qt Base								Qt Creator							
Similarity detection method	**Top-1**	**MRR**	**Top-3**	**MRR**	**Top-5**	**MRR**	**Top-10**	**MRR**	**Top-1**	**MRR**	**Top-3**	**MRR**	**Top-5**	**MRR**	**Top-10**	**MRR**
FP_JC	0.00	0.00	0.14	0.05	0.14	0.05	0.36	0.08	0.00	0.00	0.20	0.07	0.40	0.11	0.40	0.11
FP_HD	0.00	0.00	0.00	0.00	0.21	0.05	0.29	0.06	0.20	0.20	0.20	0.20	0.20	0.20	0.40	0.23
RC_CS	0.00	0.00	0.00	0.00	0.21	0.05	0.29	0.06	0.00	0.00	0.20	0.10	0.60	0.19	0.80	0.21
RC_JC	0.07	0.07	0.21	0.14	0.21	0.14	0.21	0.14	0.20	0.20	0.40	0.27	0.60	0.32	0.60	0.32
Average	0.02	0.02	0.09	0.05	0.19	0.07	0.29	0.09	0.10	0.10	0.25	0.16	0.45	0.21	0.55	0.22
FP_JC + FP_HD	0.00	0.00	0.14	0.06	0.14	0.05	0.43	0.15	0.20	0.20	0.20	0.20	0.40	0.25	0.40	0.25
FP_JC + RC_CS	0.14	0.14	0.14	0.14	0.21	0.16	0.29	0.17	0.20	0.20	0.40	0.30	0.40	0.30	0.80	0.36
FP_JC + RC_JC	0.00	0.00	0.07	0.04	0.29	0.10	0.43	0.09	0.40	0.40	0.40	0.40	0.40	0.40	0.40	0.40
FP_HD + RC_CS	0.00	0.00	0.14	0.05	0.21	0.07	0.29	0.08	0.20	0.20	0.40	0.30	0.40	0.30	0.80	0.35
FP_HD + RC_JC	0.07	0.07	0.21	0.13	0.29	0.15	0.36	0.16	0.00	0.00	0.20	0.07	0.20	0.07	0.40	0.10
RC_CS + RC_JC	0.07	0.07	0.21	0.13	0.29	0.15	0.36	0.16	0.00	0.00	0.20	0.07	0.20	0.07	0.40	0.10
Average	0.05	0.05	0.15	0.09	0.24	0.11	0.36	0.14	0.17	0.17	0.30	0.22	0.33	0.23	0.53	0.26
FP_HD + RC_CS + RC_JC	0.07	0.07	0.14	0.11	0.29	0.14	0.29	0.14	0.20	0.20	0.40	0.30	0.40	0.30	0.80	0.36
FP_JC + RC_CS + RC_JC	0.07	0.07	0.14	0.11	0.21	0.13	0.29	0.14	0.40	0.40	0.40	0.40	0.40	0.40	0.80	0.46
FP_JC + FP_HD + RC_JC	0.07	0.07	0.14	0.10	0.21	0.11	0.43	0.14	0.40	0.40	0.40	0.40	0.40	0.40	0.40	0.40
FP_JC + FP_HD + RC_CS	0.14	0.14	0.21	0.17	0.21	0.17	0.29	0.18	0.40	0.40	0.40	0.40	0.40	0.40	0.80	0.45
Average	0.09	0.09	0.16	0.12	0.23	0.14	0.33	0.15	0.35	0.35	0.40	0.38	0.40	0.38	0.70	0.42
FP_JC + FP_HD + RC_CS + RC_JC	0.07	0.07	0.21	0.14	0.21	0.14	0.29	0.15	0.40	0.40	0.40	0.40	0.40	0.40	0.80	0.22

olations on the four projects, compared to the results (MRR values between 0.14 and 0.59) of related studies on generic reviewer recommendation (e.g., Chouchen et al. (2021); Hu et al. (2020)) with more reviewer candidates (which means potentially better performance due to the larger datasets). Besides, we observed that the similarity detection methods can achieve varying performances on different OSS projects. One possible reason is that the effectiveness of code reviewer recommendation approach can be influenced by project characteristics (e.g., size and type of project datasets), which aligns with the findings by Chen *et al*. (Chen et al., 2022). In addition, the results show that combining three similarity detection methods based on file paths and semantic information can achieve the best performance of code reviewer recommendation. We cannot observe significant improvement when combining four similarity methods. We conjecture that this is because certain similarity detection methods might affect the final performance to varying degrees on our dataset and need to be assigned with appropriate weights to their similarity values; this requires further investigation, e.g., optimizing the performance by algorithms.

The results indicate that it is still challenging to recommend code reviewers when specific issues (e.g., architecture violations) are involved, and the performance cannot be always significantly improved by combining more similarity detection methods.

Finding 1: *The performance of the similarity detection methods and their combinations can produce acceptable performance scores, and achieve varying results on different projects. Combining three similarity methods can achieve the best performance of reviewer recommendation.*

7.4.2 RQ2: Comparison of Recommendation Approaches

To answer RQ2, we measured the performance of the similarity detection methods through a comparison with the baseline reviewer recommendation approach, RevFinder (Thongtanunam et al., 2015). As mentioned in Section 7.3.4, this was the only reviewer recommendation approach that we could reproduce. Specifically, we extracted the individual methods and the best-performing combinations of the similarity detection methods from Table 7.2, and we compared them with RevFinder on the four OSS projects. As shown in Figure 7.3, this includes the results of the four individual similarity detection methods, the combinations of two similarity detection methods (i.e., FP_JC + FP_HD and FP_JC + RC_JC), one combination of three similarity detection methods (i.e., FP_JC + FP_HD + RC_JC), and one combination of four similarity detection methods (i.e., FP_JC + FP_HD + RC_CS + RC_JC). Note that Fig-

ure 7.3 shows different scale for Qt Creator, to observe and compare the differences of the similarity detection methods. In terms of top-*k* accuracy, the four individual similarity detection methods and their combinations outperform RevFinder approximately 4 times on Neutron; the combination of FP_JC and RC_JC and the combination of FP_JC, FP_HD, and RC_JC achieve a better top-*k* accuracy than RevFinder on Nova; RevFinder achieves a relatively better top-*k* accuracy than the four individual similarity detection methods on Qt Base. Nearly all the similarity detection methods and their combinations outperform RevFinder on Qt Creator.

Table 7.3 presents the average Mean Reciprocal Rank (MRR) of the aforementioned similarity detection methods, their combinations, and RevFinder. The results show that the individual RC_JC method can achieve a better MRR on the four projects than RevFinder. All the mixed similarity detection methods can achieve a higher MRR than RevFinder on three of the four projects (except for Nova with mixing two and three similarity detection methods). Overall, the similarity detection methods and their combinations outperform RevFinder in the majority of the cases.

Table 7.3: Average MRR results by the selected similarity detection methods compared with RevFinder

Project	Neutron	Nova	Qt Base	Qt Creator
RevFinder	0.03	0.04	0.13	0.06
FP_JC	0.25	0.12	0.05	0.07
FP_HD	0.24	0.12	0.03	0.21
RC_CS	0.11	0.01	0.03	0.13
RC_JC	0.21	0.09	0.12	0.28
FP_JC + FP_HD	0.29	0.12	0.07	0.23
FP_JC + RC_JC	0.28	0.07	0.06	0.4
FP_JC + FP_HD + RC_JC	0.28	0.09	0.11	0.4
FP_JC + FP_HD + RC_CS + RC_JC	0.21	0.03	0.13	0.36

Discussion of RQ2: According to the results in Figure 7.3 and Table 7.3, we find that RevFinder does not perform as good as the claimed results in the original work (Thongtanunam et al., 2015) when it runs on our dataset related to specific issues (i.e., architecture violations). One possible reason could be that RevFinder recommends reviewers only by comparing the file path similarity without considering the semantic similarity of related textual artifacts. Another potential reason is that the specific dataset, specifically the size and type of the dataset (review comments on architecture violations), in our work may impact the performance of reviewer

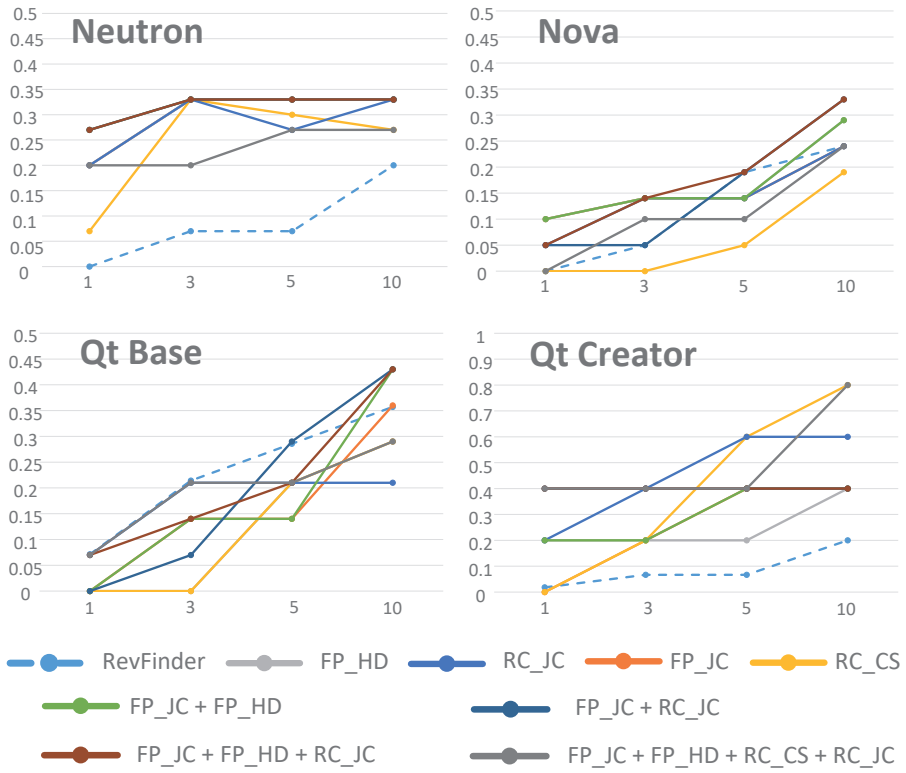

Figure 7.3: Performances of Top-*k* accuracy of mixed similarity detection methods compared to RevFinder

recommendation. Besides, the performance of RevFinder on the four projects also partially confirms the finding of RQ1, that is, project characteristics can impact the effectiveness of reviewer recommendation approaches.

> **Finding 2**: *The selected similarity detection methods and their combinations achieve a better performance than RevFinder in the majority of the cases.*

7.4.3 RQ3: Comparison of Sampling Methods

To answer RQ3, we used the incremental sampling technique to construct the *expertise* model and evaluated the performance of the selected similarity detection meth-

ods and their combinations on our dataset, as described in Section 7.3.3. We used the same performance metrics and the baseline approach mentioned in Section 7.3.4 and Section 7.3.5. Due to space limitations, we only present the top-*k* accuracy of the best-performing similarity detection methods and their combinations in Table 7.4.

According to the results in Table 7.4, when using the fixed sampling technique, almost all the top-*k* accuracy of the similarity detection methods and their combinations have better performance scores than when using the incremental sampling technique. For example, in terms of Neutron, the combination of FP_JC, FP_HD, and RC_JC achieves top-*k* accuracy of around 0.267, 0.333, 0.333, 0.333 for k = 1, 3, 5, 10, respectively. In comparison, this combination method achieves top-*k* accuracy of 0.008, 0.012, 0.012, 0.016 for k = 1, 3, 5, 10 when using incremental sampling. Moreover, the baseline approach, RevFinder, also has such performance differences when using fixed and incremental sampling techniques. Similar observations are also valid for the MRR results of the rest of the recommendation approaches listed in Table 7.4. In general, compared to using incremental sampling, the similarity detection methods and their combinations can achieve a better performance when using fixed sampling across all metrics.

Discussion of RQ3: The results of RQ3 show that all the approaches are sensitive to sampling techniques; it is clearly observed that all approaches can achieve a significantly higher recommendation performance when using the fixed sampling technique. One possible reason could be that the size of the dataset used to construct expertise models can influence the accuracy of reviewer recommendation. When small samples are taken as the historical review data, the first several iterations of the incremental sampling process may have relatively low performance and then decrease the final average top-*k* accuracy. This conjecture corroborates the recent findings by Hu *et al*. (Hu et al., 2020) that the investigated code reviewer approaches are sensitive to training data on evaluation metrics.

Finding 3: *Sampling techniques can impact the performance of code recommendation approaches. In our work, using the fixed sampling technique to construct an expertise model can achieve a significantly better performance compared to the incremental sampling technique.*

7.4.4 Implications

(1) Implications for researchers

Establish explicit standards for research on code reviewer recommendation. When a large number of submitted code changes happen, it is necessary to automate

Table 7.4: Top-*k* accuracy by the selected similarity detection methods compared with RevFinder

Project	**Neutron**								**Nova**							
Sampling	**Fixed Sampling**				**Incremental Sampling**				**Fixed Sampling**				**Incremental Sampling**			
k	**1**	**3**	**5**	**10**	**1**	**3**	**5**	**10**	**1**	**3**	**5**	**10**	**1**	**3**	**5**	**10**
RevFinder	0.000	0.067	0.067	0.200	0.000	0.000	0.002	0.014	0.000	0.048	0.190	0.238	0.001	0.008	0.014	0.018
FP_JC	0.200	0.333	0.333	0.333	0.003	0.012	0.013	0.014	0.095	0.143	0.143	0.238	0.001	0.016	0.021	0.026
FP_HD	0.200	0.333	0.333	0.033	0.003	0.008	0.013	0.024	0.095	0.143	0.143	0.286	0.001	0.015	0.018	0.022
RC_CS	0.067	0.200	0.300	0.267	0.003	0.005	0.006	0.016	0.000	0.000	0.048	0.190	0.001	0.011	0.013	0.022
RC_JC	0.200	0.200	0.267	0.333	0.005	0.018	0.024	0.024	0.048	0.143	0.143	0.238	0.001	0.010	0.015	0.016
FP_JC + FP_HD	0.267	0.333	0.333	0.333	0.003	0.008	0.015	0.016	0.095	0.143	0.143	0.286	0.001	0.015	0.018	0.023
FP_JC + RC_JC	0.267	0.333	0.333	0.333	0.008	0.012	0.013	0.014	0.048	0.048	0.190	0.333	0.005	0.016	0.019	0.024
FP_JC + FP_HD + RC_JC	0.267	0.333	0.333	0.333	0.008	0.012	0.012	0.016	0.048	0.143	0.190	0.333	0.003	0.018	0.019	0.024
FP_JC + FP_HD + RC_CS + RC_JC	0.200	0.200	0.267	0.267	0.005	0.008	0.017	0.017	0.000	0.095	0.095	0.238	0.004	0.015	0.015	0.024

Project	**Qt Base**								**Qt Creator**							
Sampling	**Fixed Sampling**				**Incremental Sampling**				**Fixed Sampling**				**Incremental Sampling**			
k	**1**	**3**	**5**	**10**	**1**	**3**	**5**	**10**	**1**	**3**	**5**	**10**	**1**	**3**	**5**	**10**
RevFinder	0.071	0.214	0.286	0.357	0.004	0.022	0.025	0.038	0.000	0.200	0.400	0.400	0.003	0.006	0.011	0.033
FP_JC	0.048	0.143	0.143	0.238	0.019	0.028	0.030	0.032	0.000	0.200	0.400	0.400	0.007	0.017	0.021	0.039
FP_HD	0.000	0.000	0.214	0.286	0.013	0.013	0.013	0.026	0.200	0.200	0.200	0.400	0.012	0.012	0.012	0.029
RC_CS	0.000	0.000	0.214	0.286	0.003	0.005	0.021	0.036	0.000	0.200	0.600	0.800	0.003	0.003	0.009	0.009
RC_JC	0.071	0.214	0.214	0.214	0.010	0.016	0.023	0.033	0.200	0.400	0.600	0.600	0.000	0.000	0.000	0.013
FP_JC + FP_HD	0.000	0.071	0.286	0.429	0.015	0.015	0.015	0.024	0.200	0.200	0.400	0.400	0.004	0.009	0.019	0.033
FP_JC + RC_JC	0.400	0.400	0.400	0.400	0.014	0.016	0.020	0.025	0.400	0.400	0.400	0.400	0.004	0.004	0.015	0.033
FP_JC + FP_HD + RC_JC	0.071	0.143	0.214	0.429	0.015	0.016	0.018	0.024	0.400	0.400	0.400	0.400	0.004	0.012	0.018	0.033
FP_JC + FP_HD + RC_CS + RC_JC	0.071	0.214	0.214	0.286	0.015	0.017	0.022	0.040	0.400	0.400	0.400	0.800	0.009	0.009	0.013	0.013

reviewer recommendation to speed up development iterations and ensure quality code reviews. In this work, we encountered certain issues that may hinder further research on code reviewer recommendation. For example, few existing studies chose to share their artifacts (e.g., datasets and source code). Moreover, most of the studies on reviewer recommendation are purely academic (Çetin et al., 2021) and lack support and validation from the industry. Besides, prior studies often use different metrics (e.g., precision, recall, and F-measure) and datasets (e.g., issues and pull-requests) in their experiments, which makes it harder to compare the performance among different approaches. Thus, it is necessary to establish explicit standards, such as promoting open science, validation in industry, and standardized metrics and datasets.

Refine the existing approaches and focus on specific issues during code review. The existing approaches for code reviewer recommendation should be refined through more empirical studies. For example, developer turnover is quite common during OSS development, but is rarely considered in current studies. In addition, employing hybrid methods to recommend reviewers by combining different approaches (see Section 7.2.2) may be promising and worth exploring in the future. Moreover, compared to general code reviewer recommendation, recommending reviewers who have awareness of specific types of issues, such as architecture violations, is important to detect and solve these issues during code review. In this work, we conducted an exploratory study that attempts to find appropriate reviewers who have knowledge of architecture violations based on historical commits related to architecture violations. It is worth investigating other types of issues (e.g., code smells, cyclic dependencies), architectural or otherwise, in recommending reviewers with pertinent knowledge.

(2) Implications for practitioners

Apply and validate in industry projects. Considering the characteristics of the open-source communities, the code reviewer recommendation approaches might have different performance on industrial projects; this is also pointed out in the study by Chen *et al*. (Chen et al., 2022). Practitioners can employ reviewer recommendation approaches in their projects by taking the associated project characteristics into consideration (e.g., constructing project-specific models). More empirical validation of these approaches from industrial projects is encouraged to consolidate the findings on their performance. Moreover, there has been little research in code reviewer recommendation (Çetin et al., 2021), and there is still a lack of industrial tools for recommending code reviewers. More collaborations between academia and industry are indispensable to devise dedicated tools (e.g., plug-ins or bots) for

existing code review systems like Gerrit.

Optimize code reviewer recommendation approaches. In this work, our reviewer recommendation approach is based on the similarity of file paths and the semantic similarity of review comments. However, there are certain realistic factors that should be considered for practical software development, such as workload, availability, and developer turnover. Therefore, practitioners should pay more attention to the above-mentioned factors when optimizing the existing approaches for code reviewer recommendation. For example, they could periodically generate an updated list of candidate reviewers and add weights to the reviewers' availability.

7.5 Threats to Validity

In this section, we discuss the threats to the validity of this study, which may affect the results of our study.

Construct Validity: The main threat to construct validity in this study concerns the performance metrics (i.e., Top-*k* accuracy and MRR) used in our work. This threat is partially mitigated in our study, as the chosen metrics are widely adopted in existing code reviewer recommendation studies (Çetin et al., 2021). Besides, we shared our dataset and source code (Li et al., 2023b) to facilitate the replication of our study and future research.

Reliability: The threats to reliability stem from how researchers potentially influence the study implementation. Possible threats in this study might come from the experimental settings (e.g., sampling percentage and iteration steps) and the reliability of measures (e.g., results of the similarity detection methods). As described in Section 7.3.3, we used a consistent and reproducible process to conduct sampling and validation on our dataset. Besides, to compare the effectiveness of our approach, we used the baseline approach (i.e., RevFinder (Thongtanunam et al., 2015)), which is a common baseline used in related studies.

External Validity: The threats to external validity pertain to the generalizability of our results. In this study, the experiment results were produced based on the code review data of four OSS projects in Gerrit. Therefore, our results may not be generalized to commercial projects or open-sourced projects on other platforms (e.g., GitHub). Our future work will try to explore commercial and GitHub projects with rich code review data to better generalize the results of our approach.

7.6 Related Work

Code reviewer recommendation has been gaining increasing attention in software engineering research in recent years, but there is still a lack of tools for recommending code reviewers (Çetin et al., 2021). Balachandran (Balachandran, 2013) proposed automated reviewer recommendation through the ReviewBot tool, which aims at improving the review quality in an industrial context through automated static code analysis. Thongtanunam *et al*. (Thongtanunam et al., 2015) proposed an expertise-based approach RevFinder based on file path similarity; their assumption is that files with similar paths have close functionality and the associated reviewers are likely to have related experience. Zanjani *et al*. (Zanjani et al., 2015) developed the cHRev approach that considers the review history including review number and review time. The cHRev approach can build an expertise model based on historical code changes and then recommend relevant peer reviewers. Yu *et al*. (Yu et al., 2016) provided a reviewer recommendation approach by building a social network named Comment Networks, which can capture common interests in social activities between contributors and reviewers, and then rank reviewers based on historical comments and the generated comment networks. Similarly, Kong *et al*. (Kong et al., 2022) proposed the Camp approach based on collaboration networks along with reviewers' expertise from pull requests and file paths.

Compared with the previous studies, our study specifically focuses on semantic information in review comments on architecture violations. We aim at recommending code reviewers who have awareness of architecture violations and can have a final check on the pending code changes (i.e., not yet being merged into the code base) that may potentially lead to architecture violations; this complements other reviewer recommendation approaches that can be used in combination with our approach to find pertinent (generic) code reviewers.

7.7 Conclusions

When a large number of code changes are submitted to a code review system like Gerrit, it is more efficient to find suitable code reviewers through automated reviewer recommendation compared to manually assigning reviewers. In this chapter, we conducted an exploratory study to recommend qualified reviewers who have awareness of architecture violations, as a promising and feasible way to detect and prevent architecture erosion through code review.

Our study is the first attempt to explore the possibility of using similarity detection methods to recommend code reviewers on architecture violations. We evalu-

ated the selected similarity detection methods and compared them with the baseline approach, RevFinder. The results show that the similarity detection methods and their combinations can produce acceptable performance, and the combined similarity detection methods outperform the baseline approach across most performance metrics on our dataset. Besides, we found that different sampling techniques used to build expertise models can impact the performance of code reviewer recommendation approaches, and the fixed sampling technique outperforms the incremental sampling technique on our dataset.

In the future, we plan to further optimize our reviewer recommendation approach (e.g., improve the performance through hybrid approaches discussed in Section 7.2.2) on larger datasets concerning architecture issues from diverse OSS projects and commercial systems (e.g., explore the possibility in a cross-project scenario).

Chapter 8
Conclusions and Future Work

This chapter concludes the dissertation by summarizing the contributions to this Ph.D. project, and discussing future work. Section 8.1 revisits the Research Questions (RQs), their answers, and the corresponding contributions. Section 8.2 discusses some promising directions for future work.

8.1 Research Questions and Contributions

For the sake of clarity and convenience, we reiterate the problem statement as presented in Chapter 1 as follows: *Current research on architecture erosion is still incomplete and lacks a comprehensive understanding of the nature of architecture erosion. Due to the limitations of existing tools, it is essential and complementary to identify architecture erosion from textual artifacts beyond source code, to gain insights into the practical erosion symptoms and their countermeasures. One textual artifact that is particularly worth exploring for the identification of architectural violations, is code reviews*. To handle this problem, we decomposed it into six RQs by leveraging the design science framework (Wieringa, 2014) (see Figure 1.2 in Chapter 1), including four knowledge questions and two design problems). We answered the six RQs in Chapters 2 to 7 respectively. In the following paragraphs, we revisit the RQs and provide a concise summary of the findings for the six RQs.

RQ1: What is the current state of the art of architecture erosion?

Establishing a landscape of the architecture erosion phenomenon and obtaining a comprehensive understanding of this phenomenon is a prerequisite of handling and addressing architecture erosion. To this end, we conducted a systematic mapping study that covers the literature spanning from January 2006 to May 2019. The key results are summarized as follows: (1) Among the different terms used to describe the architecture erosion phenomenon, "*architecture erosion*" is the most frequently-used term followed by "*architecture decay*". Architecture erosion manifests not only through architectural violations and structural issues, but also through causing problems in software quality and during the

evolution of software systems. These four perspectives (i.e., violation, structure, quality, and evolution) regarding the definition of architecture erosion are worthy of investigation in both research and practice. (2) Four types of architecture erosion symptoms are identified aligned to the four perspectives of architecture erosion, while structural and violation symptoms are the most common symptoms. (3) Non-technical reasons, such as management and organization issues, contribute to architecture erosion alongside technical reasons. (4) Architecture erosion negatively impacts software system quality attributes like maintainability and evolvability. It is essential for practitioners to raise their awareness of the consequences of architecture erosion (e.g., increased cost, failure of software projects). Practitioners can then advocate for management intervention to prioritize addressing architecture erosion and prevent potential system failures. (5) Approaches and tools for detecting and addressing architecture erosion are categorized into 19 and 35 categories respectively, and practitioners can adapt these tools and methods according to their specific needs.

The **contributions** compared to the state of the art are as follows: This is the first systematic mapping study to provide a comprehensive understanding regarding the definition of architecture erosion and existing support for identifying and handling architecture erosion. This study also lays a solid theoretical foundation on the nature of architecture erosion for follow-up research.

RQ2: What is the current state of the practice regarding architecture erosion from the developers' perspective?

In addition to analyzing the literature regarding architecture erosion, it is critical to investigate the state of the practice of architecture erosion and the relevant countermeasures employed in the industry. To this end, we conducted an empirical study to explore how developers perceive and discuss the phenomenon of architecture erosion by collecting relevant information on architecture erosion from the perspective of practitioners using three data sources (i.e., developers' online communities, surveys, and interviews). The findings unveiled that developers commonly describe architecture erosion using terms like "*erode*", "*decay*", and "*degrade*". They often consider structural issues and their impact on run-time qualities, maintenance, and evolution. Non-technical factors also contribute to architecture erosion. While there is a lack of dedicated tools for detecting architecture erosion, developers can use associated practices and tools to identify the symptoms of architecture erosion. Identifying and analyzing these symptoms can help understand the architectural structure and detect architecture erosion tendencies. The identified measures can be utilized during architecture implementation to effectively address architecture erosion.

The **contributions** are as follows: This is the first study to investigate the understanding of architecture erosion from the developers' perspective in practice. We employed triangulation (Runeson and Höst, 2009) (i.e., developers' online communities, surveys, and interviews) to collect data with the purpose of revealing the commonalities and differences between academic research and industrial practice.

RQ3: How are architecture erosion symptoms discussed in code review comments?

Identifying architecture erosion is the prerequisite for handling it. One promising and feasible way is to identify architecture erosion through *architecture erosion symptoms*, as the occurrence of the symptoms can act as early warnings for software engineers to tackle architecture erosion (e.g., by refactoring) (Li, Liang, Soliman and Avgeriou, 2021b; Ali et al., 2018). Existing code analysis tools may not be sufficient to accurately identify the wide range of erosion symptoms (Lenhard et al., 2017; Azadi et al., 2019). To this end, we focused on architecture erosion symptoms in code reviews and analyzed discussions from the Nova and Neutron projects in OpenStack. The results show that the most frequently identified erosion symptoms are architectural violations, duplicate functionality, and cyclic dependency. The number of comments on erosion symptoms decreased over time, indicating increased stability in the architecture. Most erosion symptoms were addressed by fixing or abandoning them after review votes. Code reviews can help identify and mitigate erosion symptoms, thus extending system longevity.

The **contributions** compared to the state of the art are as follows: While previous studies have examined architecture erosion symptoms through source code, this is the first study that analyzed such symptoms through textual artifacts in software development. Additionally, it investigated commonly discussed erosion symptoms and their evolution trends during code review, along with the corresponding measures implemented to address them in practice.

RQ4: Which violation symptoms are discussed during code review?

After finding out the common erosion symptoms discussed by developers during software development, we conducted a case study to further investigate the most frequent violation symptoms in depth. We collected and analyzed 606 code review comments from four popular open-source projects to study violation symptoms. The key findings include: (1) Developers discuss 10 categories of violation symptoms during code review, with the most common symptoms regarding structural inconsistencies, design-related violations, and implementation-related violations. (2) The commonly used terms to express

violation symptoms are "*inconsistent*" and "*violate*", and the most frequent linguistic pattern is "*Problem Discovery*". (3) Refactoring and removing code are the primary measures (90%) employed to address violation symptoms, while some symptoms are ignored by developers.

The **contributions** are as follows: This study provides insights on frequent violation symptoms in practice during code review. The findings highlight the importance of investigating violation symptoms for researchers to gain a deeper understanding of the characteristics of architecture violations and to facilitate development and maintenance activities.

RQ5: Design classifiers to automatically identify violation symptoms of architecture erosion.

This RQ focuses on proposing feasible means to automatically identify architecture violations from code reviews. Specifically, we developed 15 machine learning-based and 4 deep learning-based classifiers using three pre-trained word embeddings to identify violation symptoms of architecture erosion from developer discussions in code reviews. We used the dataset collected from our previous study pertaining to RQ4 (i.e., Chapter 5). The analysis focused on code review comments from four prominent open-source projects: Nova and Neutron from the OpenStack community, and Qt Base and Qt Creator from the Qt community. To validate the effectiveness of our trained classifiers, we conducted a survey to gather feedback from participants involved in discussions about architecture violations during code reviews. The results indicate that (1) the SVM classifier, utilizing the *word2vec* pre-trained word embedding, performed the best with an F1-score of 0.779; (2) classifiers employing the *fastText* pre-trained word embedding model achieved favorable performance; (3) classifiers using 200-dimensional pre-trained word embeddings outperformed those using 100 or 300-dimensional models; (4) by employing a majority voting strategy, an ensemble classifier can enhance the performance and surpass individual classifiers; (5) practitioners perceive the classifiers' results as valuable, confirming the practical potential of automated identification of violation symptoms.

The **contributions** compared to the state of the art are as follows: This is the first study to explore the possibility of automatically identifying architecture violations from code review comments through machine learning techniques. Besides, this study elaborates on our experience regarding classifiers' training, while the identified architecture violations are validated by the involved developers in practice.

RQ6: Recommend qualified code reviewers to handle architecture violations.

After proposing automated approaches to identify violation symptoms of architecture erosion, our next step focused on streamlining the code review process to both address architecture violations and mitigate human factors. To this end, we aimed at automating the recommendation of code reviewers who are qualified to review architecture violations based on reviews of code changes. We employed three common similarity detection methods to measure the file path similarity and the semantic similarity of review comments. Through a series of experiments, we evaluated these methods separately and compared them with the baseline approach, RevFinder (Thongtanunam et al., 2015), for recommending appropriate reviewers. The results demonstrate that the common similarity detection methods achieve acceptable performance scores and outperform RevFinder (Thongtanunam et al., 2015) in recommending code reviewers for architecture violations. The sampling techniques used in recommending code reviewers have an impact on the performance of reviewer recommendation approaches.

The **contributions** are as follows: This study is the first attempt to explore the possibility of using similarity detection methods to recommend code reviewers who have awareness of architecture violations, as a promising and feasible way to detect and prevent architecture erosion through code review.

8.2 Future Work

The research presented in this Ph.D. thesis marks an initial step toward the prevention of architecture erosion. Future work can further improve and broaden our existing work in multiple directions. To encourage future research on architecture erosion, we discuss some prospective directions.

(1) Documenting and managing architecture erosion symptoms

When we analyzed the actions taken by developers in response to erosion symptoms identified during code reviews, we found that developers occasionally chose to overlook certain symptoms that had been pointed out during code reviews. These ignored symptoms of architecture erosion can be transformed into potential technical debt (Fu et al., 2022), and over time, the accumulation of technical debt can exacerbate the challenges associated with architecture maintenance. Therefore, we suggest that future research should concentrate on devising effective strategies for documenting and managing erosion symptoms; such strategies include establishing traceable links to these erosion symptoms, and incorporating periodic architectural evaluations for handling those symptoms into the software development process.

(2) Recommending architecturally-aware developers to check architecture-related code

Software development encompasses a series of knowledge-intensive activities that heavily depend on the expertise of developers. Various textual artifacts generated during development contain developers' comprehension and awareness of software architecture. Future research can consider employing techniques, such as AI-based code analysis and natural language processing, to analyze textual artifacts and extract relevant architecture information, and ultimately recommend architecturally-aware developers to review and refactor architecturally-relevant code snippets. For example, developing algorithms for recommending developers based on factors like historical data (e.g., commits), availability, workload, and so forth, is a feasible way to reduce code maintenance effort and facilitate timely product iterations.

(3) Designing dedicated tools for identifying architecture erosion symptoms and issuing warnings

The results of the systematic mapping study (see Chapter 2) and the investigation involving industrial developers (see Chapter 3) shed light on a significant truth: despite the existence of certain tools that aid in identifying architecture erosion, the absence of dedicated tools specifically designed for this purpose remains an urgent problem. In Chapter 6, we took an exploratory step forward by training classifiers to automatically identify violation symptoms from textual artifacts (i.e., code review comments). Nevertheless, dedicated tools (as IDE plug-ins or standalone applications) could play a vital role in effectively identifying symptoms of architecture erosion and alerting developers during the development or code review process. The development of dedicated tools warrants attention from both researchers and industry practitioners as it holds immense potential for improving software development practices, especially during maintenance and evolution.

Appendix A
Selected Studies for Chapter 2

[S1] Z. Li and J. Long, A case study of measuring degeneration of software architectures from a defect perspective, in: Proceedings of the 18th Asia-Pacific Software Engineering Conference (APSEC), Ho Chi Minh, Vietnam, 2011, pp. 242-249: IEEE.

[S2] H. Baumeister, F. Hacklinger, R. Hennicker, A. Knapp, and M. Wirsing, A component model for architectural programming, Electronic Notes in Theoretical Computer Science, vol. 160, pp. 75-96, 2006.

[S3] M. Zhao and J. Yang, A DCA-based method for software prognostics and health management, in: Proceedings of the IEEE Prognostics and System Health Management Conference (PHM), Beijing, China, 2012, pp. 1-5: IEEE.

[S4] R. Terra and M. T. Valente, A dependency constraint language to manage object-oriented software architectures, Software: Practice and Experience, vol. 39, no. 12, pp. 1073-1094, 2009.

[S5] G. M. Rama, A desiderata for refactoring-based software modularity improvement, in: Proceedings of the 3rd Annual India Software Engineering Conference (ISEC), Mysore, India, 2010, pp. 93-102: ACM.

[S6] A. Mokni, C. Urtado, S. Vauttier, M. Huchard, and H. Y. Zhang, A formal approach for managing component-based architecture evolution, Science of Computer Programming, vol. 127, pp. 24-49, 2016.

[S7] S. Ayyaz, S. Rehman, and U. Qamar, A four method framework for fighting software architecture erosion, International Journal of Computer, Control, Quantum and Information Engineering, vol. 9, no. 1, pp. 133-139, 2015.

[S8] E. Bouwers and A. van Deursen, A lightweight sanity check for implemented architectures, IEEE Software, vol. 27, no. 4, pp. 44-50, 2010.

[S9] C. Izurieta and J. M. Bieman, A multiple case study of design pattern decay, grime, and rot in evolving software systems, Software Quality Journal, vol. 21, no. 2, pp. 289-323, 2013.

[S10] M. Mirakhorli and J. Cleland-Huang, A pattern system for tracing architectural concerns, in: Proceedings of the 18th Conference on Pattern Languages of Programs (PLoP), Portland, Oregon, USA, 2011, pp. 1-10: ACM.

[S11] A. Sejfia, A pilot study on architecture and vulnerabilities: Lessons learned, in: Proceedings of the 2nd IEEE/ACM International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE), Montreal, Quebec, Canada, 2019, pp. 42-47: IEEE.

[S12] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha, A recommendation system for repairing violations detected by static architecture conformance checking, Software: Practice and Experience, vol. 45, no. 3, pp. 315-342, 2015.

[S13] S. Herold and A. Rausch, A rule-based approach to architecture conformance checking as a quality management measure, in Relating System Quality and Software Architecture: Morgan Kaufmann, 2014, pp. 181-207.

[S14] A. Caracciolo, M. F. Lungu, and O. Nierstrasz, A unified approach to architecture conformance checking, in: Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA), Montreal, QC, Canada, 2015, pp. 41-50: IEEE.

[S15] S. Hassaine, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, ADvISE: Architectural decay in software evolution, in: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR), Szeged, Hungary, 2012, pp. 267-276: IEEE.

[S16] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, An empirical study of architectural change in open-source software systems, in: Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories (MSR),Florence, Italy, 2015, pp. 235-245: IEEE.

[S17] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, An empirical study of architectural decay in open-source software, in: Proceedings of the 15th IEEE International Conference on Software Architecture (ICSA), Seattle, WA, USA, 2018, pp. 176-185: IEEE.

[S18] F. A. Fontana, R. Roveda, M. Zanoni, C. Raibulet, and R. Capilla, An experience report on detecting and repairing software architecture erosion, in: Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), Venice, Italy, 2016, pp. 21-30: IEEE.

[S19] M. Mirakhorli, A. Fakhry, A. Grechko, M. Wieloch, and J. Cleland-Huang, Archie: A tool for detecting, monitoring, and preserving architecturally significant code, in: Proceedings of the 22nd ACM/SIGSOFT International Symposium on Foundations of Software Engineering (FSE), Hong Kong, China, 2014, pp. 739-742: ACM.

[S20] T. Greifenberg, K. Müller, and B. Rumpe, Architectural consistency checking in plugin-based software systems, in: Proceedings of the 2nd Workshop on Software Architecture Erosion and Architectural Consistency (SAEroCon), Dubrovnik/Cavtat, Croatia, 2015, pp. 1-7: ACM.

[S21] M. Riaz, M. Sulayman, and H. Naqvi, Architectural decay during continuous software evolution and impact of 'design for change' on software architecture, in: Proceedings of the International Conference on Advanced Software Engineering and Its Applications (ASEA), Jeju Island, Korea, 2009, pp. 119-126: Springer.

[S22] S. Bhattacharya and D. E. Perry, Architecture assessment model for system evolution, in: Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA), Mumbai, Maharashtra, India, 2007, pp. 44-53: IEEE.

[S23] L. Pruijt, C. Köppe, and S. Brinkkemper, Architecture compliance checking of semantically rich modular architectures: A comparative study of tool support, in: Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM), Eindhoven, The Netherlands, 2013, pp. 220-229: IEEE.

[S24] S. Miranda, E. Rodrigues Jr, M. T. Valente, and R. Terra, Architecture conformance checking in dynamically typed languages, Journal of Object Technology, vol. 15, no. 3, pp. 1-34, 2016.

[S25] S. Schröder, M. Soliman, and M. Riebisch, Architecture enforcement concerns and activities - An expert study, Journal of Systems and Software, vol. 145, pp. 79-97, 2018.

[S26] E. Guimarães, A. Garcia, and Y. Cai, Architecture-sensitive heuristics for prioritizing critical code anomalies, in: Proceedings of the 14th International Conference on Modularity (MODULARITY), Fort Collins, CO, USA, 2015, pp. 68-80: ACM.

[S27] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems, in: Proceedings of the 11th

Annual International Conference on Aspect-oriented Software Development (AOSD), Potsdam, Germany, 2012, pp. 167-178: ACM.

[S28] A. Bandi, E. B. Allen, and B. J. Williams, Assessing code decay: A data-driven approach, in: Proceedings of the 24th International Conference on Software Engineering and Data Engineering (SEDE), San Diego, CA, USA, 2015, pp. 1-8: ISCA.

[S29] V. V. G. Neto, W. Manzano, L. Garcés, M. Guessi, B. Oliveira, T. Volpato, and E. Y. Nakagawa, Back-SoS: Towards a model-based approach to address architectural drift in systems-of-systems, in: Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC), Pau, France, 2018, pp. 1461-1463: ACM.

[S30] D. Reimanis and C. Izurieta, Behavioral evolution of design patterns: Understanding software reuse through the evolution of pattern behavior, in: Proceedings of the 18th International Conference on Software and Systems Reuse (ICSR), Cincinnati, OH, USA, 2019, pp. 77-93: Springer.

[S31] A. Gurgel, I. Macia, A. Garcia, A. von Staa, M. Mezini, M. Eichberg, and R. Mitschke, Blending and reusing rules for architectural degradation prevention, in: Proceedings of the 13th International Conference on Modularity (MODULARITY), Lugano, Switzerland, 2014, pp. 61-72: ACM.

[S32] M. Langhammer, Co-evolution of component-based architecture-model and object-oriented source code, in: Proceedings of the 18th International Doctoral Symposium on Components and Architecture (WCOP), Vancouver, BC, Canada, 2013, pp. 37-42: ACM.

[S33] S. Herold and A. Rausch, Complementing model-driven development for the detection of software architecture erosion, in: Proceedings of the 5th International Workshop on Modeling in Software Engineering (MiSE), San Francisco, CA, USA, 2013, pp. 24-30: IEEE.

[S34] L. De Silva and D. Balasubramaniam, Controlling software architecture erosion: A survey, Journal of Systems and Software, vol. 85, no. 1, pp. 132-151, 2012.

[S35] H. Rocha, R. S. Durelli, R. Terra, S. Bessa, and M. T. Valente, DCL 2.0: modular and reusable specification of architectural constraints, Journal of the Brazilian Computer Society, vol. 23, no. 1, pp. 1-25, 2017.

[S36] I. M. Bertran, Detecting architecturally-relevant code smells in evolving software systems, in: Proceedings of the 33rd International Conference on Software Engineering (ICSE), Waikiki, Honolulu, HI, USA, 2011, pp. 1090-1093: ACM.

[S37] L. Zhang, Y. Sun, H. Song, F. Chauvel, and H. Mei, Detecting architecture erosion by design decision of architectural pattern, in: Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE), Miami Beach, FL, USA, 2011, pp. 758-763: KSI.

[S38] S. Wong, Y. Cai, M. Kim, and M. Dalton, Detecting software modularity violations, in: Proceedings of the 33rd International Conference on Software Engineering (ICSE), Waikiki, Honolulu, HI, USA, 2011, pp. 411-420: ACM.

[S39] M. Mirakhorli and J. Cleland-Huang, Detecting, tracing, and monitoring architectural tactics in code, IEEE Transactions on Software Engineering, vol. 42, no. 3, pp. 206-221, 2016.

[S40] S. Herold, M. English, J. Buckley, S. Counsell, and M. Ó. Cinnéide, Detection of violation causes in reflexion models, in: Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), Montreal, QC, Canada, 2015, pp. 565-569: IEEE.

[S41] R. Pérez-Castillo, I. G. R. de Guzmán, and M. Piattini, Diagnosis of software erosion through fuzzy logic, in: Proceedings of the IEEE Symposium on Computational Intelligence in Dynamic and Uncertain Environments (CIDUE), Paris, France, 2011, pp. 49-56: IEEE.

[S42] D. Nam, Y. K. Lee, and N. Medvidovic, Eva: A tool for visualizing software architectural evolution, in: Proceedings of the 40th International Conference on Software Engineering (ICSE) Companion, Gothenburg, Sweden, 2018, pp. 53-56: ACM.

[S43] M. Altinisik, E. Ersoy, and H. Sözer, Evaluating software architecture erosion for PL/SQL programs, in: Proceedings of the 11th European Conference on Software Architecture (ECSA) Companion, Canterbury, United Kingdom, 2017, pp. 159-165: ACM.

[S44] M. Lindvall, M. Becker, V. Tenev, S. Duszynski, and M. Hinchey, Good change and bad change: An analysis perspective on software evolution, Transactions on Foundations for Mastering Change I, vol. 9960, pp. 90-112, 2016.

[S45] V. Bandara and I. Perera, Identifying software architecture erosion through code comments, in: Proceedings of the 18th International Conference on Advances in ICT for Emerging Regions (ICTer), Colombo, Sri Lanka, 2018, pp. 62-69: IEEE.

[S46] J. K. Chhabra, Improving package structure of object-oriented software using multi-objective optimization and weighted class connections, Journal of King Saud University-Computer and Information Sciences, vol. 29, no. 3, pp. 349-364, 2017.

[S47] M. de Oliveira Barros, F. de Almeida Farzat, and G. H. Travassos, Learning from optimization: A case study with Apache Ant, Information and Software Technology, vol. 57, pp. 684-704, 2015.

[S48] C. Dimech and D. Balasubramaniam, Maintaining architectural conformance during software development: A practical approach, in: Proceedings of the 7th European Conference on Software Architecture (ECSA), Montpellier, France, 2013, pp. 208-223: Springer.

[S49] R. Mo, J. Garcia, Y. Cai, and N. Medvidovic, Mapping architectural decay instances to dependency models, in: Proceedings of the 4th International Workshop on Managing Technical Debt (MTD), San Francisco, CA, USA, 2013, pp. 39-46: IEEE.

[S50] A. Strasser, B. Cool, C. Gernert, C. Knieke, M. Körner, D. Niebuhr, H. Peters, A. Rausch, O. Brox, and S. Jauns-Seyfried, Mastering erosion of software architecture in automotive software product lines, in: Proceedings of the 40th International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM), Smokovec, Slovakia, 2014, pp. 491-502: Springer.

[S51] H. Koziolek, D. Domis, T. Goldschmidt, and P. Vorst, Measuring architecture sustainability, IEEE Software, vol. 30, no. 6, pp. 54-62, 2013.

[S52] H. Koziolek, D. Domis, T. Goldschmidt, P. Vorst, and R. J. Weiss, MORPHOSIS: A lightweight method facilitating sustainable software architectures, in: Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA), Helsinki, Finland, 2012, pp. 253-257: IEEE.

[S53] T. Olsson, M. Ericsson, and A. Wingkvist, Motivation and impact of modeling erosion using static architecture conformance checking, in: Proceedings of the 1st IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, 2017, pp. 204-209: IEEE.

[S54] F. Schmidt, S. MacDonell, and A. M. Connor, Multi-objective reconstruction of software architecture, International Journal of Software Engineering and Knowledge Engineering, vol. 28, no. 6, pp. 869-892, 2018.

[S55] J. Brunet, R. A. Bittencourt, D. Serey, and J. Figueiredo, On the evolutionary nature of architectural violations, in: Proceedings of the 19th Working Conference on Reverse Engineering (WCRE), Kingston, ON, Canada, 2012, pp. 257-266: IEEE.

[S56] F. Jaafar, S. Hassaine, Y.-G. Guéhéneuc, S. Hamel, and B. Adams, On the relationship between program evolution and fault-proneness: An empirical study, in: Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR), Genova, Italy, 2013, pp. 15-24: IEEE.

[S57] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, On the relevance of code anomalies for identifying architecture degradation symptoms, in: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR), Szeged, Hungary, 2012, pp. 277-286: IEEE.

[S58] L. Juarez Filho, L. Rocha, R. Andrade, and R. Britto, Preventing erosion in exception handling design using static-architecture conformance checking, in: Proceedings of the 11th European Conference on Software Architecture (ECSA), Canterbury, UK, 2017, pp. 67-83: Springer.

[S59] T. Bakota, P. Hegedüs, I. Siket, G. Ladányi, and R. Ferenc, QualityGate SourceAudit: A tool for assessing the technical quality of software, in: Proceedings of the IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), Antwerp, Belgium, 2014, pp. 440-445: IEEE.

[S60] S. Herold and M. Mair, Recommending refactorings to re-establish architectural consistency, in: Proceedings of the 8th European Conference on Software Architecture (ECSA), Vienna, Austria, 2014, pp. 390-397: Springer.

[S61] T. Haitzer, E. Navarro, and U. Zdun, Reconciling software architecture and source code in support of software evolution, Journal of Systems and Software, vol. 123, pp. 119-144, 2017.

[S62] M. Stal, Refactoring software architectures, in Agile Software Architecture, Chapter 3: Elsevier, 2014, pp. 63-82.

[S63] J. Adersberger and M. Philippsen, ReflexML: UML-based architecture-to-code traceability and consistency checking, in: Proceedings of the 5th European

Conference on Software Architecture (ECSA), Essen, Germany, 2011, pp. 344-359: Springer.

[S64] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic, Relating architectural decay and sustainability of software systems, in: Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), Venice, Italy, 2016, pp. 178-181: IEEE.

[S65] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. Mendonça, Static architecture-conformance checking: An illustrative overview, IEEE Software, vol. 27, no. 5, pp. 82-89, 2010.

[S66] B. Merkle, Stop the software architecture erosion: Building better software systems, in: Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (SPLASH/OOPSLA) Companion, Reno/Tahoe, Nevada, USA, 2010, pp. 129-138: ACM.

[S67] A. Nicolaescu, H. Lichter, A. Göringer, P. Alexander, and D. Le, The ARAMIS workbench for monitoring, analysis and visualization of architectures based on run-time interactions, in: Proceedings of the 2nd Workshop on Software Architecture Erosion and Architectural Consistency (SAEroCon), Dubrovnik/-Cavtat, Croatia, 2015, pp. 1-7: ACM.

[S68] M. Feilkas, D. Ratiu, and E. Jurgens, The loss of architectural knowledge during system evolution: An industrial case study, in: Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC), Vancouver, BC, Canada, 2009, pp. 188-197: IEEE.

[S69] M. Mair, S. Herold, and A. Rausch, Towards flexible automated software architecture erosion diagnosis and treatment, in: Proceedings of the 11th Working IEEE/IFIP Conference on Software Architecture (WICSA) Companion, Sydney, NSW, Australia, 2014, pp. 1-6: ACM.

[S70] S. Gerdes, S. Jasser, M. Riebisch, S. Schröder, M. Soliman, and T. Stehle, Towards the essentials of architecture documentation for avoiding architecture erosion, in: Proceedings of the Workshop on Sustainable Architecture: Global Collaboration, Requirements, Analysis (SAGRA), Copenhagen, Denmark, 2016, pp. 1-4: ACM.

[S71] N. Medvidovic and V. Jakobac, Using software evolution to focus architectural recovery, Automated Software Engineering, vol. 13, no. 2, pp. 225-256, 2006.

[S72] D. Baum, J. Dietrich, C. Anslow, and R. Müller, Visualizing design erosion: How big balls of mud are made, in: Proceedings of the 6th IEEE Working Conference on Software Visualization (VISSOFT), Madrid, Spain, 2018, pp. 122-126: IEEE.

[S73] M. Dalgarno, When good architecture goes bad, Methods and Tools, vol. 17, no. 1, pp. 27-34, 2009.

Bibliography

Adersberger, J. and Philippsen, M.: 2011, Reflexml: Uml-based architecture-to-code traceability and consistency checking, *in: Proceedings of the 5th European Conference on Software Architecture (ECSA)*, Springer, Essen, Germany, pp. 344–359.

Adolph, S., Hall, W. and Kruchten, P.: 2011, Using grounded theory to study the experience of software development, *Empirical Software Engineering* **16**(4), 487–513.

Al-Zubaidi, W. H. A., Thongtanunam, P., Dam, H. K., Tantithamthavorn, C. and Ghose, A.: 2020, Workload-aware reviewer recommendation using a multi-objective search-based approach, *in: Proceedings of the 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, ACM, Virtual, USA, pp. 21–30.

Ali, N., Baker, S., O'Crowley, R., Herold, S. and Buckley, J.: 2018, Architecture consistency: State of the practice, challenges and requirements, *Empirical Software Engineering* **23**(1), 224–258.

AlOmar, E. A., Mkaouer, M. W. and Ouni, A.: 2021, Toward the automatic classification of self-affirmed refactoring, *Journal of Systems and Software* **171**, 110821.

Avgeriou, P., Kruchten, P., Ozkaya, I. and Seaman, C.: 2016, Managing technical debt in software engineering (dagstuhl seminar 16162), *Dagstuhl Reports*, Vol. 6, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, pp. 110–138.

Ayyaz, S., Rehman, S. and Qamar, U.: 2015, A four method framework for fighting software architecture erosion, *International Journal of Computer, Control, Quantum and Information Engineering* **9**(1), 133–139.

Azadi, U., Fontana, F. A. and Taibi, D.: 2019, Architectural smells detected by tools: A catalogue proposal, *in: Proceedings of the 2nd IEEE/ACM International Conference on Technical Debt (TechDebt)*, IEEE, Montreal, QC, Canada, pp. 88–97.

Baabad, A., Zulzalil, H. B. and Baharom, S. B.: 2020, Software architecture degradation in open source software: A systematic literature review, *IEEE Access* **8**, 173681–173709.

Babar, M. A., Dingsøyr, T., Lago, P. and Van Vliet, H.: 2009, *Software Architecture Knowledge Management*, Springer.

Bacchelli, A. and Bird, C.: 2013, Expectations, outcomes, and challenges of modern code review, *in: Proceedings of the 35th International Conference on Software Engineering (ICSE)*, IEEE, San Francisco, CA, USA, pp. 712–721.

Badampudi, D., Unterkalmsteiner, M. and Britto, R.: 2023, Modern code reviews—survey of literature and practice, *ACM Transactions on Software Engineering and Methodology* **32**(4), 1–61.

Balachandran, V.: 2013, Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation, *in: Proceedings of the 35th International Conference on Software Engineering (ICSE)*, IEEE, San Francisco, CA, USA, pp. 931–940.

Balalaie, A., Heydarnoori, A. and Jamshidi, P.: 2016, Microservices architecture enables DevOps: Migration to a cloud-native architecture, *IEEE Software* **33**(3), 42–52.

Bandi, A., Allen, E. B. and Williams, B. J.: 2015, Assessing code decay: A data-driven approach, *in: Proceedings of the 24th International Conference on Software Engineering and Data Engineering (SEDE)*, ISCA, San Diego, CA, USA, pp. 1–8.

Bandi, A., Williams, B. J. and Allen, E. B.: 2013, Empirical evidence of code decay: A systematic mapping study, *in: Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, IEEE, Koblenz, Germany, pp. 341–350.

Barney, S., Petersen, K., Svahnberg, M., Aurum, A. and Barney, H.: 2012, Software quality trade-offs: A systematic map, *Information and Software Technology* **54**(7), 651–662.

Basili, V. R., Caldiera, G. and Rombach, H. D.: 1994, The goal question metric approach, *Encyclopedia of Software Engineering* pp. 528–532.

Bass, L., Clements, P. and Kazman, R.: 2012, *Software Architecture in Practice (3rd Edition)*, 3rd edn, Addison-Wesley Professional.

Bass, L., Clements, P. and Kazman, R.: 2021, *Software Architecture in Practice (4th Edition)*, 4th edn, Addison-Wesley Professional.

Baum, D., Dietrich, J., Anslow, C. and Müller, R.: 2018, Visualizing design erosion: How big balls of mud are made, *in: Proceedings of the 6th IEEE Working Conference on Software Visualization (VISSOFT)*, IEEE, Madrid, Spain, pp. 122–126.

Behnamghader, P., Le, D. M., Garcia, J., Link, D., Shahbazian, A. and Medvidovic, N.: 2017, A large-scale study of architectural evolution in open-source software systems, *Empirical Software Engineering* **22**(3), 1146–1193.

Beller, M., Bacchelli, A., Zaidman, A. and Juergens, E.: 2014, Modern code reviews in open-source projects: Which problems do they fix?, *in: Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, ACM, Hyderabad, India, pp. 202–211.

Besker, T., Martini, A. and Bosch, J.: 2018, Managing architectural technical debt: A unified model and systematic literature review, *Journal of Systems and Software* **135**, 1–16.

Bhattacharya, S. and Perry, D. E.: 2007, Architecture assessment model for system evolution, *in: Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, IEEE, Mumbai, Maharashtra, India, pp. 44–53.

Bi, T., Ding, W., Liang, P. and Tang, A.: 2021, Architecture information communication in two OSS projects: The why, who, when, and what, *Journal of Systems and Software* **181**, 111035.

Bi, T., Liang, P. and Tang, A.: 2018, Architecture patterns, quality attributes, and design contexts: How developers design with them, *in: Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, pp. 49–58.

Bi, T., Liang, P., Tang, A. and Yang, C.: 2018, A systematic mapping study on text analysis techniques in software architecture, *Journal of Systems and Software* **144**, 533–558.

Bianchi, A., Caivano, D., Lanubile, F. and Visaggio, G.: 2001, Evaluating software degradation through entropy, *in: Proceedings of the 7th International Software Metrics Symposium (METRICS)*, IEEE, pp. 210–219.

Bird, S., Klein, E. and Loper, E.: 2010, Natural language processing with Python: Analyzing text with the natural language toolkit, *Language Resources and Evaluation* **44**, 421–424.

Bojanowski, P., Grave, E., Joulin, A. and Mikolov, T.: 2017, Enriching word vectors with subword information, *Transactions of the Association for Computational Linguistics* **5**, 135–146.

Booch, G.: 2005, *The unified modeling language user guide*, Pearson Education India.

Bosu, A., Carver, J. C., Bird, C., Orbeck, J. and Chockley, C.: 2016a, Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft, *IEEE Transactions on Software Engineering* **43**(1), 56–75.

Bosu, A., Carver, J. C., Bird, C., Orbeck, J. and Chockley, C.: 2016b, Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft, *IEEE Transactions on Software Engineering* **43**(1), 56–75.

Bosu, A., Carver, J. C., Hafiz, M., Hilley, P. and Janni, D.: 2014, Identifying the characteristics of vulnerable code changes: An empirical study, *in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, ACM, Hong Kong, China, pp. 257–268.

Bosu, A., Greiler, M. and Bird, C.: 2015, Characteristics of useful code reviews: An empirical study at Microsoft, *in: Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories (MSR)*, IEEE, Florence, Italy, pp. 146–156.

Breivold, H. P. and Crnkovic, I.: 2010, A systematic review on architecting for software evolvability, *in: Proceedings of the 21st Australian Software Engineering Conference (ASWEC)*, IEEE, Auckland, New Zealand, pp. 13–22.

Brosig, F., Meier, P., Becker, S., Koziolek, A., Koziolek, H. and Kounev, S.: 2014, Quantitative evaluation of model-driven performance analysis and simulation of component-based architectures, *IEEE Transactions on Software Engineering* **41**(2), 157–175.

Brunet, J., Bittencourt, R. A., Serey, D. and Figueiredo, J.: 2012, On the evolutionary nature of architectural violations, *in: Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, IEEE, pp. 257–266.

Brunet, J., Murphy, G. C., Terra, R., Figueiredo, J. and Serey, D.: 2014, Do developers discuss design?, *in: Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, IEEE, Hyderabad, India, pp. 340–343.

Caracciolo, A., Lungu, M. F. and Nierstrasz, O.: 2015, A unified approach to architecture conformance checking, *in: Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, IEEE, Montreal, QC, Canada, pp. 41–50.

Çetin, H. A., Doğan, E. and Tüzün, E.: 2021, A review of code reviewer recommendation studies: Challenges and future directions, *Science of Computer Programming* **208**, 102652.

Chaniotaki, A.-M. and Sharma, T.: 2021, Architecture smells and pareto principle: A preliminary empirical exploration, *in: Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, IEEE, Madrid, Spain, pp. 190–194.

Charmaz, K.: 2014, *Constructing Grounded Theory*, sage.

Chen, L.: 2018, Microservices: Architecting for continuous delivery and DevOps, *in: Proceedings of the 15th IEEE International Conference on Software Architecture (ICSA)*, IEEE, pp. 39–397.

Chen, L., Babar, M. A. and Zhang, H.: 2010, Towards an evidence-based understanding of electronic data sources, *in: Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, BCS, Keele University, UK, pp. 1–4.

Chen, Q., Kong, D., Bao, L., Sun, C., Xia, X. and Li, S.: 2022, Code reviewer recommendation in tencent: Practice, challenge, and direction, *in: Proceedings of the 44nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, ACM, Pittsburgh, PA, USA, pp. 115–124.

Chouchen, M., Ouni, A., Mkaouer, M. W., Kula, R. G. and Inoue, K.: 2021, Whoreview: A multi-objective search-based approach for code reviewers recommendation in modern code review, *Applied Soft Computing* **100**, 106908.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R. and Stafford, J.: 2010, *Documenting Software Architectures: Views and Beyond (2nd Edition)*, Pearson Education.

Cohen, J.: 1960, A coefficient of agreement for nominal scales, *Educational and Psychological Measurement* **20**(1), 37–46.

Conway, M. E.: 1968, How do committees invent, *Datamation* **14**(4), 28–31.

Da Silva, A. R.: 2017, Linguistic patterns and linguistic styles for requirements specification (i) an application case with the rigorous rsl/business-level language, *in: Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPLoP)*, ACM, Irsee, Germany, pp. 1–27.

Dalgarno, M.: 2009, When good architecture goes bad, *Methods and Tools* **17**(1), 27–34.

D'Ambros, M., Gall, H., Lanza, M. and Pinzger, M.: 2008, Analysing software repositories to understand software evolution, *Software Evolution*, Springer, chapter 3, pp. 37–67.

Davila, N. and Nunes, I.: 2021, A systematic literature review and taxonomy of modern code review, *Journal of Systems and Software* **177**, 110951.

de Oliveira Barros, M., de Almeida Farzat, F. and Travassos, G. H.: 2015, Learning from optimization: A case study with Apache Ant, *Information and Software Technology* **57**, 684–704.

De Silva, L. and Balasubramaniam, D.: 2012, Controlling software architecture erosion: A survey, *Journal of Systems and Software* **85**(1), 132–151.

Di Sorbo, A., Panichella, S., Alexandru, C. V., Shimagaki, J., Visaggio, C. A., Canfora, G. and Gall, H. C.: 2016, What would users change in my app? summarizing app reviews for recommending software changes, *in: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, ACM, Seattle, WA, USA, pp. 499–510.

Di Sorbo, A., Panichella, S., Alexandru, C. V., Visaggio, C. A. and Canfora, G.: 2017, Surf: Summarizer of user reviews feedback, *in: Proceedings of the 39th IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*, IEEE, Buenos Aires, Argentina, pp. 55–58.

Di Sorbo, A., Panichella, S., Visaggio, C. A., Di Penta, M., Canfora, G. and Gall, H.: 2016, Deca: development emails content analyzer, *in: Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE) Companion*, ACM, Austin, TX, USA, pp. 641–644.

Dietterich, T. G.: 2000, Ensemble methods in machine learning, *in: Proceedings of the 1st International Workshop of Multiple Classifier Systems (MCS) 2000*, Springer, Cagliari, Italy, pp. 1–15.

Dogan, E., Tüzün, E., Tecimer, K. A. and Güvenir, H. A.: 2019, Investigating the validity of ground truth in code reviewer recommendation studies, *in: Proceedings of the 13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, IEEE, Gothenburg, Sweden, pp. 1–6.

Efstathiou, V., Chatzilenas, C. and Spinellis, D.: 2018a, Word embeddings for the software engineering domain, *in: Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, ACM, Gothenburg, Sweden, pp. 38–41.

Efstathiou, V., Chatzilenas, C. and Spinellis, D.: 2018b, Word embeddings for the software engineering domain, *in: Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, ACM, Gothenburg, Sweden, pp. 38–41.

El Asri, I., Kerzazi, N., Uddin, G., Khomh, F. and Idrissi, M. J.: 2019, An empirical study of sentiments in code reviews, *Information and Software Technology* **114**, 37–54.

Fagan, M.: 1976, Design and code inspections to reduce errors in program development, *IBM Systems Journal* **15**(3), 182–211.

Feilkas, M., Ratiu, D. and Jurgens, E.: 2009, The loss of architectural knowledge during system evolution: An industrial case study, *in: Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC)*, IEEE, Vancouver, BC, Canada, pp. 188–197.

Fejzer, M., Przymus, P. and Stencel, K.: 2018, Profile based recommendation of code reviewers, *Journal of Intelligent Information Systems* **50**(3), 597–619.

Fellah, A. and Bandi, A.: 2019, On architectural decay prediction in real-time software systems, *in: Proceedings of the 28th International Conference on Software Engineering and Data Engineering (SEDE)*, Vol. 64, ISCA, San Diego, CA, USA, pp. 98–108.

Fontana, F. A., Roveda, R., Zanoni, M., Raibulet, C. and Capilla, R.: 2016, An experience report on detecting and repairing software architecture erosion, *in: Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, IEEE, Venice, Italy, pp. 21–30.

Foote, B. and Yoder, J.: 1997, Big ball of mud, *Pattern Languages of Program Design* **4**, 654–692.

Fowler Jr, F. J.: 2013, *Survey Research Methods (5th edition)*, Sage publications.

Fu, L., Liang, P., Rasheed, Z., Li, Z., Tahir, A. and Xiaofeng, H.: 2022, Potential technical debt and its resolution in code reviews: An exploratory study of the OpenStack and Qt communities, *in: Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ACM, Helsinki, Finland, pp. 216–226.

Fu, W. and Menzies, T.: 2017, Easy over hard: A case study on deep learning, *in: Proceedings of the 11th joint meeting on Foundations of Software Engineering (FSE)*, ACM, Paderborn, Germany, pp. 49–60.

Ganesh, S., Sharma, T. and Suryanarayana, G.: 2013, Towards a principle-based classification of structural design smells, *Journal of Object Technology* **12**(2), 1–29.

Garcia, J., Kouroshfar, E., Ghorbani, N. and Malek, S.: 2022, Forecasting architectural decay from evolutionary history, *IEEE Transactions on Software Engineering* **48**(7), 2439–2454.

Garcia, J., Popescu, D., Edwards, G. and Medvidovic, N.: 2009, Identifying architectural bad smells, *in: Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE, Kaiserslautern, Germany, pp. 255–258.

Garlan, D., Allen, R. and Ockerbloom, J.: 1995, Architectural mismatch: Why reuse is so hard, *IEEE Software* **12**(6), 17–26.

Garlan, D., Allen, R. and Ockerbloom, J.: 2009, Architectural mismatch: Why reuse is still so hard, *IEEE Software* **26**(4), 66–69.

Gerdes, S., Jasser, S., Riebisch, M., Schröder, S., Soliman, M. and Stehle, T.: 2016, Towards the essentials of architecture documentation for avoiding architecture erosion, *in: Proceedings of the 10th European Conference on Software Architecture Workshops (ECSA)*, ACM, pp. 1–4.

Godfrey, M. W. and Lee, E. H.: 2000, Secrets from the monster: Extracting Mozilla's software architecture, *in: Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET)*, Citeseer, pp. 1–9.

Gonçalves, P. W., Fregnan, E., Baum, T., Schneider, K. and Bacchelli, A.: 2020, Do explicit review strategies improve code review performance?, *in: Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, ACM, Seoul, South Korea, pp. 606–610.

Gopal, M. K.: 2016, Design quality metrics on the package maintainability and reliability of open source software, *International Journal of Intelligent Engineering and Systems* **9**(4), 195–204.

Grottke, M., Matias, R. and Trivedi, K. S.: 2008, The fundamentals of software aging, *in: Proceedings of the 19th IEEE International Conference on Software Reliability Engineering Workshops (ISSREW)*, IEEE, Seattle, WA, USA, pp. 1–6.

Grundy, J., Hosking, J. and Mugridge, W. B.: 1998, Inconsistency management for multiple-view software development environments, *IEEE Transactions on Software Engineering* **24**(11), 960–981.

Gurgel, A., Macia, I., Garcia, A., von Staa, A., Mezini, M., Eichberg, M. and Mitschke, R.: 2014, Blending and reusing rules for architectural degradation prevention, *in: Proceedings of the 13th International Conference on Modularity (MODULARITY)*, ACM, Lugano, Switzerland, pp. 61–72.

Han, J., Pei, J. and Tong, H.: 2022, *Data Mining: Concepts and Techniques*, Morgan kaufmann.

Han, X., Tahir, A., Liang, P., Counsell, S., Blincoe, K., Li, B. and Luo, Y.: 2022, Code smells detection via modern code review: A study of the openstack and qt communities, *Empirical Software Engineering* **27**(6), 127.

Han, X., Tahir, A., Liang, P., Counsell, S. and Luo, Y.: 2021, Understanding code smell detection via code review: A study of the OpenStack community, *in: Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC)*, IEEE, Madrid, Spain (Virtual), pp. 323–334.

Hansen, L. K. and Salamon, P.: 1990, Neural network ensembles, *IEEE transactions on pattern analysis and machine intelligence* **12**(10), 993–1001.

Hassaine, S., Guéhéneuc, Y.-G., Hamel, S. and Antoniol, G.: 2012, ADvISE: Architectural decay in software evolution, *in: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE, Szeged, Hungary, pp. 267–276.

He, H. and Garcia, E. A.: 2009, Learning from imbalanced data, *IEEE Transactions on Knowledge and Data Engineering* **21**(9), 1263–1284.

Herold, S.: 2020, An initial study on the association between architectural smells and degradation, *in: Proceedings of the 14th European Conference on Software Architecture (ECSA)*, Springer, L'Aquila, Italy, pp. 193–201.

Herold, S., Blom, M. and Buckley, J.: 2016, Evidence in architecture degradation and consistency checking research: Preliminary results from a literature review, *in: Proceedings of the 10th European Conference on Software Architecture Workshops (ECSAW)*, ACM, pp. 1–7.

Herold, S., English, M., Buckley, J., Counsell, S. and Cinnéide, M. Ó.: 2015, Detection of violation causes in reflexion models, *in: Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, Montreal, QC, Canada, pp. 565–569.

Hey, T., Keim, J., Koziolek, A. and Tichy, W. F.: 2020, Norbert: Transfer learning for requirements classification, *in: Proceedings of the 28th IEEE International Requirements Engineering Conference (RE)*, IEEE, Zurich, Switzerland, pp. 169–179.

Hirao, T., McIntosh, S., Ihara, A. and Matsumoto, K.: 2022, Code reviews with divergent review scores: An empirical study of the openstack and qt communities, *IEEE Transactions on Software Engineering* **48**(2), 69–81.

Hochstein, L. and Lindvall, M.: 2005, Combating architectural degeneration: A survey, *Information and Software Technology* **47**(10), 643–656.

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

Hove, S. E. and Anda, B.: 2005, Experiences from conducting semi-structured interviews in empirical software engineering research, *in: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS)*, IEEE, Como, Italy, pp. 10–23.

Hu, Y., Wang, J., Hou, J., Li, S. and Wang, Q.: 2020, Is there a "golden" rule for code reviewer recommendation?: An experimental evaluation, *in: Proceedings of the 20th IEEE International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, Macau, China, pp. 497–508.

Huang, Q., Xia, X., Lo, D. and Murphy, G. C.: 2018, Automating intention mining, *IEEE Transactions on Software Engineering* **46**(10), 1098–1119.

Hutter, F., Kotthoff, L. and Vanschoren, J.: 2019, *Automated machine learning: methods, systems, challenges*, Springer Nature.

ISO/IEC25010: 2011, *ISO/IEC 25010: 2011 Systems and software engineering-Systems and software Quality Requirements and Evaluation (SQuaRE)-System and software quality models*.

Israel, G. D.: 1992, Determining sample size, *Technical report*, Florida Cooperative Extension Service, Institute of Food and Agricultural Sciences, University of Florida, Florida, U.S.A.

Izurieta, C. and Bieman, J. M.: 2007, How software designs decay: A pilot study of pattern evolution, *in: Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM)*, IEEE, pp. 449–451.

Izurieta, C. and Bieman, J. M.: 2013, A multiple case study of design pattern decay, grime, and rot in evolving software systems, *Software Quality Journal* **21**(2), 289–323.

Jaktman, C. B., Leaney, J. and Liu, M.: 1999, Structural analysis of the software architecture: A maintenance assessment case study, *Working Conference on Software Architecture*, Springer, pp. 455–470.

Jiang, J., He, J.-H. and Chen, X.-Y.: 2015, Coredevrec: Automatic core member recommendation for contribution evaluation, *Journal of Computer Science and Technology* **30**(5), 998–1016.

Juarez Filho, L., Rocha, L., Andrade, R. and Britto, R.: 2017, Preventing erosion in exception handling design using static-architecture conformance checking, *in: Proceedings of the 11th European Conference on Software Architecture (ECSA)*, Springer, Canterbury, UK, pp. 67–83.

Kashiwa, Y., Nishikawa, R., Kamei, Y., Kondo, M., Shihab, E., Sato, R. and Ubayashi, N.: 2022a, An empirical study on self-admitted technical debt in modern code review, *Information and Software Technology* **146**, 106855.

Kashiwa, Y., Nishikawa, R., Kamei, Y., Kondo, M., Shihab, E., Sato, R. and Ubayashi, N.: 2022b, An empirical study on self-admitted technical debt in modern code review, *Information and Software Technology* **146**, 106855.

Khalajzadeh, H., Shahin, M., Obie, H. O., Agrawal, P. and Grundy, J.: 2022, Supporting developers in addressing human-centric issues in mobile apps, *IEEE Transactions on Software Engineering* **49**(4), 2149–2168.

Kim, Y.: 2014, Convolutional neural network for sentence classification, *in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, ACL, Doha, Qatar, pp. 1746–1751.

Kiremire, A. R.: 2011, The application of the pareto principle in software engineering, *Consulted January* **13**, 1–12.

Kitchenham, B. A. and Pfleeger, S. L.: 2008, Personal opinion surveys, *Guide to Advanced Empirical Software Engineering*, Springer, chapter 3, pp. 63–92.

Kitchenham, B. and Charters, S.: 2007, Guidelines for performing systematic literature reviews in software engineering, *Report*, Technical Report, Version 2.3 EBSE-2007-01, Keele University & University of Durham.

Kong, D., Chen, Q., Bao, L., Sun, C., Xia, X. and Li, S.: 2022, Recommending code reviewers for proprietary software projects: A large scale study, *in: Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, Honolulu, HI, USA, pp. 630–640.

Koziolek, H., Domis, D., Goldschmidt, T. and Vorst, P.: 2013, Measuring architecture sustainability, *IEEE Software* **30**(6), 54–62.

Kumar, N.: 2016, Software architecture validation methods, tools support and case studies, *Emerging Research in Computing, Information, Communication and Applications*, Springer, chapter 32, pp. 335–345.

Land, R.: 2002, Software deterioration and maintainability-a model proposal, *in: Proceedings of the 2nd Conference on Software Engineering Research and Practice in Sweden (SERPS)*, Citeseer.

Le, D. M., Behnamghader, P., Garcia, J., Link, D., Shahbazian, A. and Medvidovic, N.: 2015, An empirical study of architectural change in open-source software systems, *in: Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories (MSR)*, IEEE, pp. 235–245.

Le, D. M., Carrillo, C., Capilla, R. and Medvidovic, N.: 2016, Relating architectural decay and sustainability of software systems, *in: Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, IEEE, Venice, Italy, pp. 178–181.

Le, D. M., Link, D., Shahbazian, A. and Medvidovic, N.: 2018, An empirical study of architectural decay in open-source software, *in: Proceedings of the 15th IEEE International Conference on Software Architecture (ICSA)*, IEEE, Seattle, WA, USA, pp. 176–185.

Lehman, M. M. and Ramil, J. F.: 2002, Software evolution and software evolution processes, *Annals of Software Engineering* **14**(1-4), 275–309.

Lenhard, J., Blom, M. and Herold, S.: 2019, Exploring the suitability of source code metrics for indicating architectural inconsistencies, *Software Quality Journal* **27**(1), 241–274.

Lenhard, J., Hassan, M. M., Blom, M. and Herold, S.: 2017, Are code smell detection tools suitable for detecting architecture degradation?, *in: Proceedings of the 11th European Conference on Software Architecture (ECSA) Companion*, ACM, Canterbury, United Kingdom, pp. 138–144.

Lethbridge, T. C., Sim, S. E. and Singer, J.: 2005, Studying software engineers: Data collection techniques for software field studies, *Empirical Software Engineering* **10**(3), 311–341.

Li, R., Liang, P. and Avgeriou, P.: 2022, Replication Package for the Paper: Warnings: Violation Symptoms Indicating Architecture Erosion, https://doi.org/10.5281/zenodo.7054370.

Li, R., Liang, P. and Avgeriou, P.: 2023a, Code reviewer recommendation for architecture violations: An exploratory study, *in: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, IEEE, Oulu, Finland, pp. 42–51.

Li, R., Liang, P. and Avgeriou, P.: 2023b, Replication Package for the Paper: Code Reviewer Recommendation for Architecture Violations: An Exploratory Study, https://doi.org/10.5281/zenodo.7292880.

Li, R., Liang, P. and Avgeriou, P.: 2023c, Replication Package for the Paper: Towards Automatic Identification of Violation Symptoms of Architecture Erosion, https://zenodo.org/doi/10.5281/zenodo.10038355.

Li, R., Liang, P. and Avgeriou, P.: 2023d, Warnings: Violation symptoms indicating architecture erosion, *Information and Software Technology* **164**, 107319.

Li, R., Liang, P., Soliman, M. and Avgeriou, P.: 2021a, Li R, Liang P, Soliman M, Avgeriou P. Replication package for the paper: Understanding software architecture erosion: A systematic mapping study, https://doi.org/10.5281/zenodo.5562418.

Li, R., Liang, P., Soliman, M. and Avgeriou, P.: 2021b, Understanding architecture erosion: The practitioners' perceptive, *in: Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC)*, IEEE, Madrid, Spain, pp. 311–322.

Li, R., Liang, P., Soliman, M. and Avgeriou, P.: 2022, Understanding software architecture erosion: A systematic mapping study, *Journal of Software: Evolution and Process* **34**(3), e2423.

Li, R., Soliman, M., Liang, P. and Avgeriou, P.: 2021a, Replication Package for the Paper: Symptoms of Architecture Erosion in Code Reviews: A Study of Two OpenStack Projects, https://doi.org/10.5281/zenodo.5676037.

Li, R., Soliman, M., Liang, P. and Avgeriou, P.: 2021b, Replication package for the paper: Understanding architecture erosion: The practitioners' perceptive, https://doi.org/10.5281/zenodo.4481564.

Li, R., Soliman, M., Liang, P. and Avgeriou, P.: 2022, Symptoms of architecture erosion in code reviews: A study of two openstack projects, *in: Proceedings of the 19th IEEE International Conference on Software Architecture (ICSA)*, IEEE, Honolulu, Hawaii, USA, pp. 24–35.

Li, Y., Soliman, M. and Avgeriou, P.: 2022, Identifying self-admitted technical debt in issue tracking systems using machine learning, *Empirical Software Engineering* **27**(6), 131.

Li, Z., Avgeriou, P. and Liang, P.: 2015, A systematic mapping study on technical debt and its management, *Journal of Systems and Software* **101**, 193–220.

Li, Z. and Long, J.: 2011, A case study of measuring degeneration of software architectures from a defect perspective, *in: Proceedings of the 18th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, Ho Chi Minh, Vietnam, pp. 242–249.

Li, Z., Qi, X., Yu, Q., Liang, P., Mo, R. and Yang, C.: 2021, Multi-programming-language commits in oss: An empirical study on apache projects, *in: Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC)*, IEEE, Madrid, Spain, pp. 219–229.

Lipcak, J. and Rossi, B.: 2018, A large-scale study on source code reviewer recommendation, *in: Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, IEEE, Prague, Czech, pp. 378–387.

Liu, Z., Xia, X., Hassan, A. E., Lo, D., Xing, Z. and Wang, X.: 2018, Neural-machine-translation-based commit message generation: How far are we?, *in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, ACM, Montpellier, France, pp. 373–384.

Ma, Z., Li, R., Li, T., Zhu, R., Jiang, R., Yang, J., Tang, M. and Zheng, M.: 2020, A data-driven risk measurement model of software developer turnover, *Soft Computing* **24**(2), 825–842.

Macia, I., Arcoverde, R., Garcia, A., Chavez, C. and von Staa, A.: 2012, On the relevance of code anomalies for identifying architecture degradation symptoms, *in: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE, Szeged, Hungary, pp. 277–286.

Macia, I., Garcia, J., Popescu, D., Garcia, A., Medvidovic, N. and von Staa, A.: 2012, Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems, *in: Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development (AOSD)*, ACM, pp. 167–178.

Maffort, C., Valente, M. T., Terra, R., Bigonha, M., Anquetil, N. and Hora, A.: 2016, Mining architectural violations from version history, *Empirical Software Engineering* **21**(3), 854–895.

Mair, M. and Herold, S.: 2013, Towards extensive software architecture erosion repairs, *in: Proceedings of the 7th European Conference on Software Architecture (ECSA)*, Springer, Montpellier, France,, pp. 299–306.

Mair, M., Herold, S. and Rausch, A.: 2014, Towards flexible automated software architecture erosion diagnosis and treatment, *in: Proceedings of the 11th Working IEEE/IFIP Conference on Software Architecture (WICSA) Companion*, ACM, Sydney, NSW, Australia, pp. 1–6.

March, S. T. and Smith, G. F.: 1995, Design and natural science research on information technology, *Decision support systems* **15**(4), 251–266.

Marcus, A. and Poshyvanyk, D.: 2005, The conceptual cohesion of classes, *in: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, IEEE, pp. 133–142.

Martin, R. C.: 2000, Design principles and design patterns, *Object Mentor* **1**(34), 597.

Martin, R. C.: 2003, *Agile software development: principles, patterns, and practices*, Prentice Hall PTR.

Martin, R. C. and Martin, M.: 2006, *Agile Principles, Patterns, and Practices in C#, 1st Edition*, Pearson.

Mendoza, C., Bocanegra, J., Garcés, K. and Casallas, R.: 2021, Architecture violations detection and visualization in the continuous integration pipeline, *Software: Practice and Experience* **51**(8), 1822–1845.

Merkle, B.: 2010, Stop the software architecture erosion: Building better software systems, *in: Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (SPLASH/OOPSLA) Companion*, ACM, Reno/Tahoe, Nevada, USA, pp. 129–138.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S. and Dean, J.: 2013, Distributed representations of words and phrases and their compositionality, *in: Proceedings of the 27th Annual Conference on Neural Information Processing Systems (NeurIPS)*, Curran Associates, Inc., Lake Tahoe, Nevada, United States, pp. 3111–3119.

Minaee, S., Kalchbrenner, N., Cambria, E., Nikzad, N., Chenaghlu, M. and Gao, J.: 2021, Deep learning-based text classification: A comprehensive review, *ACM Computing Surveys* **54**(3), 1–40.

Miranda, S., Rodrigues Jr, E., Valente, M. T. and Terra, R.: 2016, Architecture conformance checking in dynamically typed languages, *Journal of Object Technology* **15**(3), 1–34.

Mo, R., Garcia, J., Cai, Y. and Medvidovic, N.: 2013, Mapping architectural decay instances to dependency models, *in: Proceedings of the 4th International Workshop on Managing Technical Debt (MTD)*, IEEE, San Francisco, CA, USA, pp. 39–46.

Mokni, A., Huchard, M., Urtado, C., Vauttier, S. and Zhang, Y.: 2015, An evolution management model for multi-level component-based software architectures, *in: Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, KSI, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, pp. 674–679.

Mokni, A., Urtado, C., Vauttier, S., Huchard, M. and Zhang, H. Y.: 2016, A formal approach for managing component-based architecture evolution, *Science of Computer Programming* **127**, 24–49.

Morales, R., McIntosh, S. and Khomh, F.: 2015, Do code review practices impact design quality? A case study of the Qt, VTK, and ITK projects, *in: Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, Montreal, QC, Canada, pp. 171–180.

Mumtaz, H., Singh, P. and Blincoe, K.: 2021, A systematic mapping study on architectural smells detection, *Journal of Systems and Software* **173**, 110885.

Murphy, G. C., Notkin, D. and Sullivan, K.: 1995, Software reflexion models: Bridging the gap between source and high-level models, *in: Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, ACM, Washington, DC, USA, pp. 18–28.

Murphy, G. C., Notkin, D. and Sullivan, K. J.: 2001, Software reflexion models: Bridging the gap between design and implementation, *IEEE Transactions on Software Engineering* **27**(4), 364–380.

Nasab, A. R., Shahin, M., Liang, P., Basiri, M., Raviz, S. H., Khalajzadeh, H., Waseem, M. and Naseri, A.: 2021, Automated identification of security discussions in microservices systems: Industrial surveys and experiments, *Journal of Systems and Software* p. 111046.

Neri, D., Soldani, J., Zimmermann, O. and Brogi, A.: 2020, Design principles, architectural smells and refactorings for microservices: A multivocal review, *SICS Software-Intensive Cyber-Physical Systems* **35**(1), 3–15.

Neto, V. V. G., Manzano, W., Garcés, L., Guessi, M., Oliveira, B., Volpato, T. and Nakagawa, E. Y.: 2018, Back-sos: Towards a model-based approach to address architectural drift in systems-of-systems, *in: Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC)*, ACM, Pau, France, pp. 1461–1463.

Nord, R. L., Ozkaya, I., Kruchten, P. and Gonzalez-Rojas, M.: 2012, In search of a metric for managing architectural technical debt, *in: Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA)*, IEEE, Helsinki, Finland, pp. 91–100.

Oizumi, W., Garcia, A., Sousa, L. D. S., Cafeo, B. and Zhao, Y.: 2016, Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems, *in: Proceedings of the 38th International Conference on Software Engineering (ICSE)*, ACM, pp. 440–451.

Oizumi, W. N., Garcia, A. F., Colanzi, T. E., Ferreira, M. and Staa, A. V.: 2015, On the relationship of code-anomaly agglomerations and architectural problems, *Journal of Software Engineering Research and Development* **3**(1), 1–22.

Oizumi, W., Sousa, L., Oliveira, A., Carvalho, L., Garcia, A., Colanzi, T. and Oliveira, R.: 2019, On the density and diversity of degradation symptoms in refactored classes: A multicase study, *in: Proceedings of the 30th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, Berlin, Germany, pp. 346–357.

Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimhigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S. and Wolf, A. L.: 1999, An architecture-based approach to self-adaptive software, *IEEE Intelligent Systems and Their Applications* **14**(3), 54–62.

Ouni, A., Kula, R. G. and Inoue, K.: 2016a, Search-based peer reviewers recommendation in modern code review, *in: Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, Raleigh, NC, USA, pp. 367–377.

Ouni, A., Kula, R. G. and Inoue, K.: 2016b, Search-based peer reviewers recommendation in modern code review, *in: Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, Raleigh, NC, USA, pp. 367–377.

Pagano, R. R.: 2012, *Understanding Statistics in the Behavioral Sciences*, Cengage Learning.

Paixao, M., Krinke, J., Han, D., Ragkhitwetsagul, C. and Harman, M.: 2019, The impact of code review on architectural changes, *IEEE Transactions on Software Engineering* pp. 1–19.

Paixão, M., Uchôa, A., Bibiano, A. C., Oliveira, D., Garcia, A., Krinke, J. and Arvonio, E.: 2020, Behind the intents: An in-depth empirical study on software refactoring in modern code review, *in: Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, ACM, Seoul, South Korea, pp. 125–136.

Pal, A., Chang, S. and Konstan, J. A.: 2012, Evolution of experts in question answering communities, *in: Proceedings of the 6th International Conference on Weblogs and Social Media (ICWSM)*, AAAI, pp. 274–281.

Parnas, D. L.: 1994, Software aging, *in: Proceedings of the 16th International Conference on Software Engineering (ICSE)*, IEEE, Sorrento, Italy, pp. 279–287.

Pennington, J., Socher, R. and Manning, C. D.: 2014, Glove: Global vectors for word representation, *in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, ACL, Doha, Qatar, pp. 1532–1543.

Perry, D. E. and Wolf, A. L.: 1992, Foundations for the study of software architecture, *ACM SIGSOFT Software Engineering Notes* **17**(4), 40–52.

Peters, F., Tun, T. T., Yu, Y. and Nuseibeh, B.: 2017, Text filtering and ranking for security bug report prediction, *IEEE Transactions on Software Engineering* **45**(6), 615–631.

Petersen, K., Vakkalanka, S. and Kuzniarz, L.: 2015, Guidelines for conducting systematic mapping studies in software engineering: An update, *Information and Software Technology* **64**, 1–18.

Pruijt, L. and Brinkkemper, S.: 2014, A metamodel for the support of semantically rich modular architectures in the context of static architecture compliance checking, *in: Proceedings of the 11th Working IEEE/IFIP Conference on Software Architecture (WICSA) Companion*, ACM, Sydney, NSW, Australia, pp. 1–8.

Qi, Y., Sachan, D., Felix, M., Padmanabhan, S. and Neubig, G.: 2018, When and why are pre-trained word embeddings useful for neural machine translation?, *in: Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, ACL, New Orleans, Louisiana, USA, pp. 529–535.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I. et al.: 2018, Improving language understanding by generative pre-training, *In: Preprint* pp. 1–12.

Rahman, M. M., Roy, C. K. and Kula, R. G.: 2017, Predicting usefulness of code review comments using textual features and developer experience, *in: Proceedings of the 14th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, IEEE, Buenos Aires, Argentina, pp. 215–226.

Rama, G. M.: 2010, A desiderata for refactoring-based software modularity improvement, *in: Proceedings of the 3rd Annual India Software Engineering Conference (ISEC)*, ACM, Mysore, India, pp. 93–102.

Rebai, S., Amich, A., Molaei, S., Kessentini, M. and Kazman, R.: 2020, Multi-objective code reviewer recommendations: Balancing expertise, availability and collaborations, *Automated Software Engineering* **27**(3), 301–328.

Reimanis, D. and Izurieta, C.: 2019, Behavioral evolution of design patterns: Understanding software reuse through the evolution of pattern behavior, *in: Proceedings of the 18th International Conference on Software and Systems Reuse (ICSR)*, Springer, Cincinnati, OH, USA, pp. 77–93.

Ren, X., Xing, Z., Xia, X., Lo, D., Wang, X. and Grundy, J.: 2019, Neural network-based detection of self-admitted technical debt: From performance to explainability, *ACM Transactions on Software Engineering and Methodology* **28**(3), 1–45.

Rocha, H., Durelli, R. S., Terra, R., Bessa, S. and Valente, M. T.: 2017, Dcl 2.0: modular and reusable specification of architectural constraints, *Journal of the Brazilian Computer Society* **23**(1), 1–25.

Ruangwan, S., Thongtanunam, P., Ihara, A. and Matsumoto, K.: 2019, The impact of human factors on the participation decision of reviewers in modern code review, *Empirical Software Engineering* **24**(2), 973–1016.

Runeson, P. and Höst, M.: 2009, Guidelines for conducting and reporting case study research in software engineering, *Empirical Software Engineering* **14**(2), 131–164.

Sadowski, C., Söderberg, E., Church, L., Sipko, M. and Bacchelli, A.: 2018, Modern code review: A case study at Google, *in: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, ACM, Gothenburg, Sweden, pp. 181–190.

Sangal, N., Jordan, E., Sinha, V. and Jackson, D.: 2005, Using dependency models to manage complex software architecture, *in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM, pp. 167–176.

Sangwan, R. S., Vercellone-Smith, P. and Laplante, P. A.: 2008, Structural epochs in the complexity of software over time, *IEEE Software* **25**(4), 66–73.

Sarkar, S., Ramachandran, S., Kumar, G. S., Iyengar, M. K., Rangarajan, K. and Sivagnanam, S.: 2009, Modularization of a large-scale business application: A case study, *IEEE Software* **26**(2), 28–35.

Schmitt Laser, M., Medvidovic, N., Le, D. M. and Garcia, J.: 2020, Arcade: an extensible workbench for architecture recovery, change, and decay evaluation, *in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ACM, Sacramento, CA, USA, pp. 1546–1550.

Schröder, S. and Riebisch, M.: 2017, Architecture conformance checking with description logics, *in: Proceedings of the 11th European Conference on Software Architecture (ECSA) Companion*, ACM, Canterbury, United Kingdom, pp. 166–172.

Schultis, K.-B., Elsner, C. and Lohmann, D.: 2016, Architecture-violation management for internal software ecosystems, *in: Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, IEEE, Venice, Italy, pp. 241–246.

Sesari, E., Hort, M. and Sarro, F.: 2022, An empirical study on the fairness of pre-trained word embeddings, *in: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*, ACL, Dublin, Ireland.

Shahin, M., Liang, P. and Babar, M. A.: 2014a, A systematic review of software architecture visualization techniques, *Journal of Systems and Software* **94**, 161–185.

Shahin, M., Liang, P. and Babar, M. A.: 2014b, A systematic review of software architecture visualization techniques, *Journal of Systems and Software* **94**, 161–185.

Sharma, T., Singh, P. and Spinellis, D.: 2020, An empirical investigation on the relationship between design and architecture smells, *Empirical Software Engineering* **25**(5), 4020–4068.

Sharma, T. and Spinellis, D.: 2018, A survey on software smells, *Journal of Systems and Software* **138**, 158–173.

Shaw, M. and Clements, P.: 2006, The golden age of software architecture, *IEEE Software* **23**(2), 31–39.

Shull, F., Singer, J. and Sjøberg, D. I.: 2007, *Guide to Advanced Empirical Software Engineering*, Springer.

Singer, J., Sim, S. E. and Lethbridge, T. C.: 2008, Software engineering data collection for field studies, *Guide to Advanced Empirical Software Engineering*, Springer, chapter 1, pp. 9–34.

Sinha, V. S., Mani, S. and Gupta, M.: 2013, Exploring activeness of users in QA forums, *in: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, IEEE, pp. 77–80.

Soliman, M., Galster, M., Salama, A. R. and Riebisch, M.: 2016, Architectural knowledge for technology decisions in developer communities: An exploratory study with stackoverflow, *in: Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, IEEE, Venice, Italy, pp. 128–133.

Soliman, M., Salama, A. R., Galster, M., Zimmermann, O. and Riebisch, M.: 2018, Improving the search for architecture knowledge in online developer communities, *in: Proceedings of the 15th IEEE International Conference on Software Architecture (ICSA)*, IEEE, Seattle, WA, USA, pp. 186–195.

Stal, M.: 2014, Refactoring software architectures, *Agile Software Architecture*, Elsevier, chapter 3, pp. 63–82.

Stol, K.-J., Ralph, P. and Fitzgerald, B.: 2016, Grounded theory in software engineering research: A critical review and guidelines, *in: Proceedings of the 38th International Conference on Software Engineering (ICSE)*, ACM, Austin, TX, USA, pp. 120–131.

Strasser, A., Cool, B., Gernert, C., Knieke, C., Körner, M., Niebuhr, D., Peters, H., Rausch, A., Brox, O. and Jauns-Seyfried, S.: 2014, Mastering erosion of software architecture in automotive software product lines, *in: Proceedings of the 40th International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*, Springer, Smokovec, Slovakia, pp. 491–502.

Sturtevant, D.: 2017, Modular architectures make you agile in the long run, *IEEE Software* **35**(1), 104–108.

Sülün, E., Tüzün, E. and Doğrusöz, U.: 2021, Rstrace+: Reviewer suggestion using software artifact traceability graphs, *Information and Software Technology* **130**, 106455.

Tahir, A., Yamashita, A., Licorish, S., Dietrich, J. and Counsell, S.: 2018, Can you tell me if it smells?: A study on how developers discuss code smells and anti-patterns in stack overflow, *in: Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering (EASE)*, ACM, pp. 68–78.

Tan, P.-N., Steinbach, M. and Kumar, V.: 2016, *Introduction to Data Mining*, Pearson Education India.

Terra, R. and Valente, M. T.: 2009, A dependency constraint language to manage object-oriented software architectures, *Software: Practice and Experience* **39**(12), 1073–1094.

Terra, R., Valente, M. T., Czarnecki, K. and Bigonha, R. S.: 2015, A recommendation system for repairing violations detected by static architecture conformance checking, *Software: Practice and Experience* **45**(3), 315–342.

Thongtanunam, P., McIntosh, S., Hassan, A. E. and Iida, H.: 2017, Review participation in modern code review: An empirical study of the android, qt, and openstack projects, *Empirical Software Engineering* **22**(2), 768–817.

Thongtanunam, P., Tantithamthavorn, C., Kula, R. G., Yoshida, N., Iida, H. and Matsumoto, K.-i.: 2015, Who should review my code? A file location-based code-reviewer recommendation approach for modern code review, *in: Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, Montreal, QC, Canada, pp. 141–150.

Tian, F., Liang, P. and Babar, M. A.: 2019, How developers discuss architecture smells? an exploratory study on stack overflow, *in: Proceedings of the 16th IEEE International Conference on Software Architecture (ICSA)*, IEEE, pp. 91–100.

Tran, J. B. and Holt, R. C.: 1999, Forward and reverse repair of software architecture, *in: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, IBM Press, Mississauga, Ontario, Canada, pp. 1–9.

Tsay, J., Dabbish, L. and Herbsleb, J.: 2014, Influence of social and technical factors for evaluating contribution in github, *in: Proceedings of the 36th International Conference on Software Engineering (ICSE)*, ACM, pp. 356–366.

Uchôa, A., Barbosa, C., Oizumi, W., Blenilio, P., Lima, R., Garcia, A. and Bezerra, C.: 2020, How does modern code review impact software design degradation? An in-depth empirical study, *in: Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, Adelaide, Australia, pp. 511–522.

van Gurp, J. and Bosch, J.: 2002, Design erosion: Problems and causes, *Journal of Systems and Software* **61**(2), 105–119.

Vassileva, J.: 2008, Toward social learning environments, *IEEE Transactions on Learning Technologies* **1**(4), 199–214.

Venters, C. C., Capilla, R., Betz, S., Penzenstadler, B., Crick, T., Crouch, S., Nakagawa, E. Y., Becker, C. and Carrillo, C.: 2018, Software sustainability: Research and practice from a software architecture viewpoint, *Journal of Systems and Software* **138**, 174–188.

Wang, S., Lo, D. and Jiang, L.: 2013, An empirical study on developer interactions in stackoverflow, *in: Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*, ACM, pp. 1019–1024.

Wang, T., Wang, D. and Li, B.: 2019, A multilevel analysis method for architecture erosion, *in: Proceedings of the 31st International Conference on Software Engineering and Knowledge Engineering (SEKE)*, KSI, Hotel Tivoli, Lisbon, Portugal, pp. 443–566.

Watson, C., Cooper, N., Palacio, D. N., Moran, K. and Poshyvanyk, D.: 2022, A systematic literature review on the use of deep learning in software engineering research, *ACM Transactions on Software Engineering and Methodology* **31**(2), 1–58.

Wieringa, R. J.: 2014, *Design Science Methodology for Information Systems and Software Engineering*, Springer.

Wohlin, C.: 2016, Second-generation systematic literature studies using snowballing, *in: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, ACM, Limerick, Ireland, pp. 1–6.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B. and Wesslén, A.: 2012, *Experimentation in Software Engineering*, Springer Science & Business Media.

Xia, X., Lo, D., Wang, X. and Yang, X.: 2015, Who should review this change?: Putting text and file location analyses together for more accurate recommendations, *in: Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, Bremen, Germany, pp. 261–270.

Yang, Y., Xia, X., Lo, D., Bi, T., Grundy, J. and Yang, X.: 2022, Predictive models in software engineering: Challenges and opportunities, *ACM Transactions on Software Engineering and Methodology* **31**(3), 1–72.

Yin, R. K.: 2009, *Case Study Research: Design and Methods*, Vol. 5, sage.

Yu, Y., Wang, H., Yin, G. and Wang, T.: 2016, Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?, *Information and Software Technology* **74**, 204–218.

Zanjani, M. B., Kagdi, H. and Bird, C.: 2015, Automatically recommending peer reviewers in modern code review, *IEEE Transactions on Software Engineering* **42**(6), 530–543.

Zar, J. H.: 1972, Significance testing of the Spearman rank correlation coefficient, *Journal of the American Statistical Association* **67**(339), 578–580.

Zhang, L., Sun, Y., Song, H., Chauvel, F. and Mei, H.: 2011, Detecting architecture erosion by design decision of architectural pattern, *in: Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE)*, KSI, Miami Beach, FL, USA, pp. 758–763.

Zhou, X., Jin, Y., Zhang, H., Li, S. and Huang, X.: 2016, A map of threats to validity of systematic literature reviews in software engineering, *in: Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, pp. 153–160.

Zhou, Y. and Sharma, A.: 2017, Automated identification of security issues from commit messages and bug reports, *in: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE)*, ACM, Paderborn, Germany, pp. 914–919.