UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ARMANDO SOARES SOUSA

A SYSTEMATIC AND EFFICIENT APPROACH FOR IDENTIFYING ARCHITECTURAL TECHNICAL DEBT

FORTALEZA
2024

ARMANDO SOARES SOUSA

A SYSTEMATIC AND EFFICIENT APPROACH FOR IDENTIFYING ARCHITECTURAL TECHNICAL DEBT

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de doutor em Ciência da Computação. Área de Concentração: Engenharia de Software

Orientador: Prof. Dr. Lincoln Souza Rocha

Coorientador: Dr. Ricardo de Sousa Britto

FORTALEZA

2024

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S696s Sousa, Armando Soares.
A systematic and efficient approach for identifying architectural technical debt / Armando Soares Sousa. – 2024.
173 f. : il. color.

Tese (doutorado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação , Fortaleza, 2024.
Orientação: Prof. Dr. Lincoln Souza Rocha.
Coorientação: Prof. Dr. Ricardo de Sousa Britto.

1. Architectural Technical Debt. 2. Mining Software Repositories. 3. Code Change Analysis. I. Título.
CDD 005

ARMANDO SOARES SOUSA

A SYSTEMATIC AND EFFICIENT APPROACH FOR IDENTIFYING ARCHITECTURAL TECHNICAL DEBT

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de doutor em Ciência da Computação. Área de Concentração: Engenharia de Software

Aprovada em: 24/05/2024.

To my beloved family, whose unwavering belief
in me has been the cornerstone of my journey.

ACKNOWLEDGEMENTS

To Prof. Dr. Lincon Rocha, for his unwavering belief in me from the outset, for his exceptional guidance, and for mentoring me throughout my journey as a researcher and computer scientist.

To my co-supervisor, Dr. Ricardo Britto, for sharing his extensive experience in applied industry research and for his invaluable support during the experiments conducted at Ericsson.

To FUNCAP (Cearense Foundation for Support to Scientific and Technological Development), for their generous financial support, which enabled the continuation of my research grant.

To the professors of the Master's and Doctorate Program in Computer Science at UFC, for their insightful teachings, valuable collaborations, and constructive suggestions.

To my doctoral classmates, for their thought-provoking discussions, constructive criticisms, and helpful suggestions.

To the professors from the UFPI computing department, for their support during my absence for qualification.

"If I have seen further, it is by standing on the shoulders of giants." (Isaac Newton)

RESUMO

Dívida Técnica Arquitetural (DTA) refere-se aos custos acumulados e compensações que surgem de decisões arquiteturais e compensações técnicas feitas durante o processo de desenvolvimento de software. Resulta dos compromissos assumidos para cumprir metas e prazos de curto prazo, muitas vezes levando a consequências de longo prazo em termos de qualidade, manutenção e evolução do sistema. É uma das principais Dívidas Técnicas (DT) que mais impactam a manutenção de sistemas de software complexos. Às vezes, devido à falta de informações, os engenheiros de software dependem principalmente de artefatos de código-fonte como fonte de informações para gerenciar a DTA, o que é uma tarefa desafiadora. Para isso, é necessário identificar quais artefatos do código-fonte estão relacionados a problemas arquiteturais e decidir se esses artefatos estão gerando um esforço de manutenção recorrente e crescente ao longo do tempo. Esta tese tem como objetivo propor uma abordagem automatizada para identificar dívidas técnicas arquiteturais e seu impacto em arquivos de código-fonte usando *Architectural Smells*, métricas de código, dados históricos e informações de repositórios Git. A abordagem emprega uma variedade de técnicas de pesquisa, incluindo revisão de literatura, estudos de caso, entrevistas com profissionais e avaliação de generalização usando ChatGPT. Com base no método *Design Science*, apresentamos uma solução que pode ser usada por pesquisadores e profissionais da indústria para identificar artefatos de código relacionados a DTA em repositórios de código sob gerência de configuração. O método proposto permite identificar artefatos de código-fonte que auxiliam na refatoração da tomada de decisão para resolução de DTA sem a necessidade de avaliação por especialistas em arquitetura de software. Nossa análise revelou que os arquivos de código-fonte associados a *Architectural Smells*, que são frequentemente modificados e apresentam tamanho e complexidade crescentes ao longo do tempo, têm maior probabilidade de estar associados a DTA. Portanto, podemos concluir que é viável identificar sistematicamente a presença de DTA utilizando apenas informações de artefatos de código-fonte usando um Sistema de Controle de Versão. Essa abordagem automatizada oferece benefícios potenciais para os desenvolvedores, fornecendo *insights* sobre questões architeturais e reduz o espaço de pesquisa para efeitos de DTA nos artefatos do código-fonte do projeto.

Keywords: dívida técnica arquitetural; mineração de repositórios de software; análise de mudança de código.

ABSTRACT

Architectural Technical Debt (ATD) refers to the accumulated costs and trade-offs that arise from architectural decisions and technical trade-offs made during the software development process. It results from the compromises made to meet short-term goals and deadlines, often leading to long-term consequences in terms of system quality, maintainability, and evolution. It is one of the leading Technical Debts (TD) that most impact maintaining complex software systems. Sometimes, due to a lack of information, software engineers rely mainly on source code artifacts as a source of information to manage ATD, which is a challenging task. For this, it is necessary to identify which source code artifacts are related to architectural problems and decide whether these artifacts are leading to a recurring and increasing maintenance effort over time. This thesis aims to propose an automated approach to identifying architectural technical debt and its impact on source code files using Architectural Smells, code metrics, historical data, and information from Git repositories. The approach employs a range of research techniques, including literature review, case studies, interviews with practitioners, and generalization assessment using ChatGPT. Based on the Design Science Method, we present a solution that can be used by researchers and industry practitioners to identify ATD-related code artifacts in code repositories under configuration management. The proposed method allows us to identify source code artifacts that help refactor decision-making for ATD resolution without requiring evaluation by experts in software architecture. Our analysis revealed that source code files associated with Architectural Smells, which are frequently modified and exhibit increasing size and complexity over time, are more likely to be associated with ATD. Therefore, we can conclude that it is feasible to systematically identify the presence of ATD by solely using information from source code artifacts within a Version Control System. This automated approach offers potential benefits for developers by providing insights into architectural issues and reducing the search space for ATD effects on the project's source code artifacts.

Keywords: architectural technical debt; mining software repositories; code change analysis.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

This chapter introduces this thesis that proposes to develop an automated approach to identifying Architectural Technical Debt (ATD) and its impact on source code files. It starts in Section 1.1, outlining the problem this research addresses. Section 1.2 provides the research context. Section 1.3 presents the motivation, explaining what we are studying and why it is important. Section 1.4 identifies research gaps in the area under study. Section 1.5 presents the goal and research questions that guide this work. Section 1.6 describes the research design and methods used. Section 1.7 outlines the main contributions of this research and summarizes the studies conducted. Next, in Section 1.8, the proposed approach is summarized. The publications achieved through this research are presented in Section 1.9. Finally, Section 1.10 outlines the organization of the thesis's remaining chapters.

1.1 Problem Outline

When best software architecture practices are not prioritized, there is often a natural pressure for faster deliveries, leading to low-quality architectural design. This, in turn, can result in poorly designed components, such as those with low modularity, high coupling, and low cohesion (BROWN *et al.*, 2010; LI *et al.*, 2014; MARTINI *et al.*, 2015; TOLEDO *et al.*, 2021a). This phenomenon is commonly known as ATD (BESKER *et al.*, 2018). The occurrence and accumulation of ATD in complex software systems can increase maintenance efforts and make system evolution more challenging in the long term, ultimately hindering future development activities (KRUCHTEN *et al.*, 2019).

Many software projects do not prioritize the identification and monitoring of ATD items. These items can be challenging to identify, as they often permeate various stages and artifacts of the software development cycle (BESKER *et al.*, 2017a). The accumulation of ATD can lead to more significant maintenance efforts, such as higher bug-fix costs, more substantial efforts to add new features, and increased efforts to maintain existing features (TOLEDO *et al.*, 2021b; XIAO *et al.*, 2021). Over time, this can result in excessive expenditures on software maintenance efforts and even lead to software bankruptcy (ALFAYEZ *et al.*, 2018).

This thesis focuses on Architectural Technical Debt (ATD) identification by examining the recurring effort required to maintain source code artifacts over time in software repositories under version control. For example, to address the escalating challenge of ATD, integrating our automated ATD detection method into CI/CD pipelines is feasible. This integration can enable proactive mitigation and ensure long-term software quality. Moreover, it fosters a continuous ATD management strategy that identifies and resolves architectural issues earlier, preventing their accumulation and adverse impact on software maintainability.

1.2 Research Context

This thesis aims to present an automated approach to identify ATD and show the findings of studies applied in large-scale software projects[1]. Our primary focus will be on ATD, the type of TD that impacts the software architecture. Examples of ATD include architectural violations (e.g., the implemented architecture deviates from a set of predefined architectural rules), suboptimal application of established architectural patterns, early architectural decisions with unforeseen trade-offs, and architectural smells. Identifying ATD in large-scale projects is more difficult than in other TD types. This is because large-scale projects are more complex, distributed, and involve multiple people. In addition, software architecture documentation is not always properly updated, which further complicates the task.

Specifically, the thesis proposes a method for automatically identifying source code artifacts affected by ATD (when it is present) and assists software architects and developers in making decisions regarding the payment of Technical Debt (TD) generated by these source code files in the context of large-scale software projects.

1.3 Motivation

The motivation for our research stems from the challenges faced by developers and architects at Ericsson in handling ATD. One of the advisors identified the problem through interactions with multiple developers and architects at regular Kaizen events at Ericsson. Although the company has processes in place to manage code Technical Debt (SOUSA *et al.*, 2021), we identified that their ATD management could be improved, starting with ATD identification. Because it is important to identify ATD items to manage the ATD in the best way, we investigated the following points: (i) solving challenges and implications of ATD; (ii) identifying methods and effective strategies for identifying ATD items; (iii) addressing issues related to their ATD items; and (iv) identifying effective strategies for evaluating the impact of ATD as from source-code

[1] Dikert *et al.* (2016a) define as large-scale software projects that involve at least 50 human resources – not necessarily only developers, but also other staff collaborating in software development – or at least six teams.

repository under version control.

In our research, we conducted a systematic mapping study (SMS) (PETERSEN *et al.*, 2008) on ATD and created a method to assist software architects and developers in making decisions regarding the payment of TD. To demonstrate the feasibility of our approach, we applied the Design Science Method (OFFERMANN *et al.*, 2009) in the Cassandra database, an Apache project widely used by large companies like Ericsson. Additionally, the method's generalizability was evaluated in other large-scale projects from the open-source ecosystem.

Finally, many companies that produce large-scale software need help to identify and manage ATD items. Thus, the solutions that are already in place and the solutions proposed in the context of this thesis have the potential to be effective and applicable in companies that want to manage and pay their TD, mainly regarding ATD that causes a major impact on the maintenance and evolution of their software systems.

1.4 Research Gaps

While there have been significant research efforts in the identification of ATD (LI *et al.*, 2015; CARPIO, 2016; VERDECCHIA *et al.*, 2018; MARTINI *et al.*, 2018b; PÉREZ, 2020; VERDECCHIA *et al.*, 2021), there are still some gaps and limitations in the existing literature. Kruchten *et al.* (2019) observed some gaps and limitations in the existing literature about the identification of ATD:

- **Lack of Consensus**: There is no consensus on the definition of ATD and its various types, which has led to different approaches and techniques for identifying ATD. This lack of consensus can make it difficult to compare and validate the effectiveness of different identification techniques.
- **Limited Empirical Evaluation**: Many of the techniques and tools proposed for identifying ATD have been evaluated on a limited number of case studies or in controlled laboratory settings, which may not reflect the complexities of real-world software systems. More empirical evaluation of a wide range of systems is needed to establish the effectiveness of these techniques.
- **Lack of Integration**: There is a lack of integration between different techniques for identifying ATD, which can limit their effectiveness. For example, while metrics-based approaches can identify code complexity, they may not capture other design issues that are indicators of ATD.

– **Inadequate Tool Support**: Some of the proposed techniques for identifying ATD rely on manual inspection or analysis, which can be time-consuming and error-prone. More effective and automated tools are needed to improve the accuracy and efficiency of ATD identification.
– **Limited Coverage**: Many of the existing techniques and approaches for identifying ATD are focused on code-level issues, such as architectural smells or code complexity. However, ATD can also arise from higher-level architectural decisions, such as system-level trade-offs or technology choices.

Also, according to Besker *et al.* (2018) there are four main gaps highlighted in the ATD process:

(i) lack of guides on how to successfully manage ATD in practice;
(ii) no consensus regarding the identification of ATD;
(iii) there is a lack of tools and methods to evaluate and monitor ATD;
(iv) there is a lack of systematic methods to evaluate the impact of ATD items.

To the best of our knowledge, no study has a method that identifies ATD items without using expert analysis in an automated way using only data extracted from the code repository. So, one of the most challenging aspects of ATD is its identification, which can be seen as the first step to establishing ways for ATD measurement and management.

This thesis fills the existing gaps by employing quantitative (repository mining in Apache Cassandra project) and qualitative analysis (interviews with Ericsson developers) to identify the source code artifacts from the code repository under version control. Furthermore, we proposed a method focused on extracting data from a repository in an automated way without needing expert analysis. Finally, a method and respective tool to support software architects and developers were proposed, facilitating decisions to pay ATD from source code artifacts of large-scale software systems.

1.5 Research Goal and Research Questions

In the context of Technical Debt and Software Architecture, we define Architectural Technical Debt (ATD) as the accumulation of artifacts that significantly impact the system's architecture and require frequent changes in conjunction with other architectural elements, generating recurring maintenance effort.

This kind of Technical Debt arises, for example, when design decisions are made without considering the long-term impact, when the architectural documentation is outdated, or when critical system components are not properly decoupled.

This thesis aims to propose an automated approach to identifying ATD and its impact on source code files. It focuses on leveraging data solely from software repositories under version control systems within large-scale software projects.

To achieve this overarching goal, two primary research questions have been formulated:

RQ1 *What are the main challenges that large-scale software projects face with Architectural Technical Debt?*

RQ2 *How to identify ATD in a systematic and reliable way?*

RQ1 was mainly addressed in the systematic review (Chapter 4), where we detailed a systematic mapping study about ATD. It is important because it provides a structured and comprehensive overview of the state of the art in this area. It helps to identify the key research questions and gaps in the literature, as well as to categorize and analyze the existing studies systematically. Also, we conducted an investigation in a case study reported in Chapter 5, an industrial case study at Ericsson, to investigate the factors related to TD accumulation in large-scale Global Software Engineering (GSE) projects to understand the process of Technical Debt Management (TDM) in a real-world industrial scenario and the main factors of TD accumulation. While **RQ2** was addressed as a combination of method and tool proposed in Chapter 7, we performed various exploratory tests in various Git repositories to discover the leading factories related to ATD in source code. We proposed a method and a tool to extract information history and data from Git repositories. Besides, we conducted another study in Ericsson reported in Chapter 8 to identify source-code artifacts that indicate the presence of ATD using a systematic method. Finally, in Chapter 9, we conducted a qualitative validation process of the proposed method using Self-Admitted Technical Debt (SATD) in issues related to architectural problems. Thus, both quantitative and qualitative methods were employed to validate the proposed approach.

1.6 Research Design and Methods

The research problem and associated research questions of this thesis were addressed using the Design Science process (PEFFERS *et al.*, 2007; OFFERMANN *et al.*, 2009). Design Science is a research approach commonly used in computer science that involves creating innovative solutions to real-world problems through the development and evaluation of artifacts such as models, prototypes, and algorithms (WIERINGA, 2014). The goal of Design Science is to develop new knowledge and theories by applying existing knowledge and theories to the creation of new artifacts. It is often used in applied research where the focus is on designing solutions that can be implemented in practice. Besides, we employed a literature review using a systematic mapping study (PETERSEN *et al.*, 2008) and investigated a real-world case using a Case Study Research (RUNESON *et al.*, 2012). Thus, rather than focusing on proposing new theories, this thesis focused on understanding challenges with the defined research problem. Furthermore, it also focused on providing an empirical solution to propose a method to solve the studied problem.

This research employed an iterative and evolutionary approach, encompassing literature review, case studies, data analysis, interviews with software industry professionals, and experiments in open-source projects to identify artifacts impacted by ATD. By developing an automated method leveraging code repository analysis and mining tools, the research established a means to identify ATD-impacted code artifacts without requiring the expertise of a software architect.

As depicted in Figure 1, this thesis followed a Design Science approach. Informed by a literature review on ATD and a case study on TD at Ericsson, it proposed a method for identifying ATD. This method was subsequently evaluated through a study at Ericsson and an experiment with Apache open-source projects. Figure 2 illustrates the research papers (P1, P2, P3, P4, and P5) included in this thesis, each with distinct objectives, research questions, and analytical perspectives. However, the individual contributions of each paper were combined to address the research questions of this thesis and support the identification of ATD.

1.7 Contributions and Summary of Studies

An overview of the research reported in this thesis is depicted in Figure 2.

The individual research papers (P1, P2, P3, P4, and P5) included in this thesis have their contributions. However, this thesis has the following general novel contributions: (i) a holistic view of ATD challenges regarding the identification and monitoring process. This contribution resulted from paper P1; and (ii) a method and a tool to identify ATD items using only data extracted from repository code under version control. This contribution resulted from the combination of the solutions put forward in papers P2, P3, P4, and P5.

In the remainder of this section, we present a summary of the included research

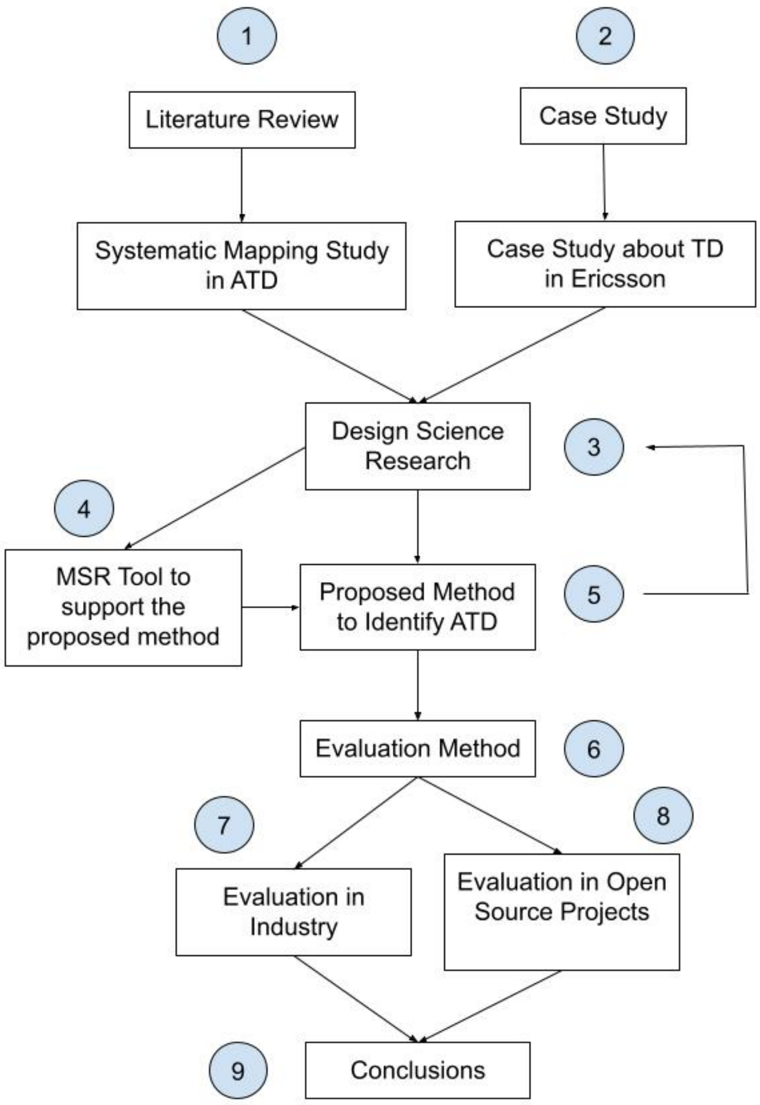

1
2
Literature Review
Case Study
Systematic Mapping Study in ATD
Case Study about TD in Ericsson
Design Science Research
3
4
MSR Tool to support the proposed method
Proposed Method to Identify ATD
5
Evaluation Method
6
7
8
Evaluation in Industry
Evaluation in Open Source Projects
9
Conclusions

Figure 1 – Research Design Overview

papers and their respective main contributions.

The research paper P1 (Systematic Mapping Study in Architectural Technical Debt) offers a comprehensive review and guidance for researchers and practitioners seeking insights into the identification, measurement, monitoring, tools, methods, and calculation of ATD. The main contributions of this study are (i) a mapping of the last ten years of the main works and research on Architectural Technical Debt; and (ii) analysis and organization of existing literature to classify and identify trends on how to identify, measure, and monitor ATD and tools and methods related to ATD, thus allowing good directions for researchers and practitioners seeking a better understanding within the context of ATD.

The research paper P2 (Technical Debt in Large-Scale Distributed Projects An Industrial Case Study) is related to an industrial case study at Ericsson to investigate the factors

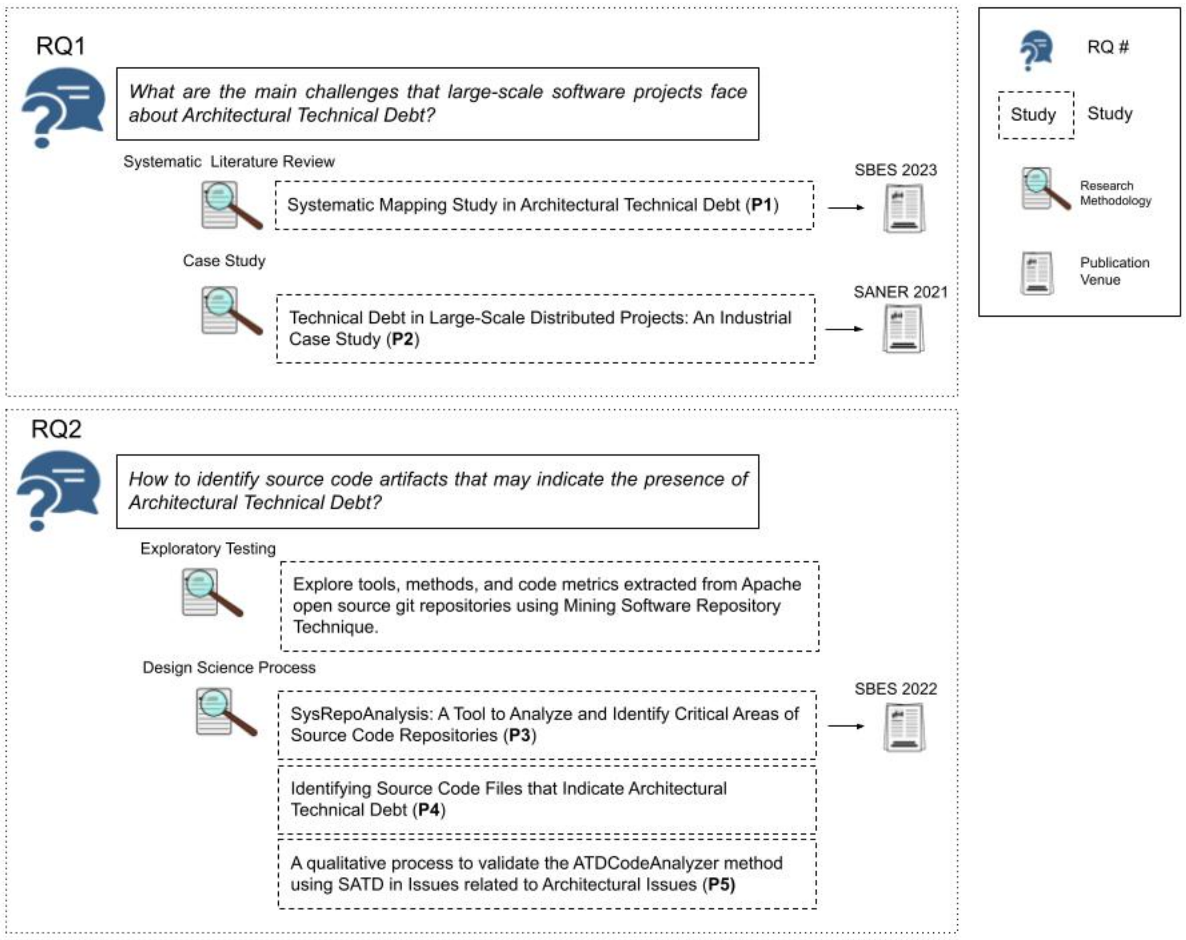

RQ1
What are the main challenges that large-scale software projects face about Architectural Technical Debt?
Systematic Literature Review
Systematic Mapping Study in Architectural Technical Debt (P1)
SBES 2023
Case Study
Technical Debt in Large-Scale Distributed Projects: An Industrial Case Study (P2)
SANER 2021
RQ2
How to identify source code artifacts that may indicate the presence of Architectural Technical Debt?
Exploratory Testing
Explore tools, methods, and code metrics extracted from Apache open source git repositories using Mining Software Repository Technique.
Design Science Process
SysRepoAnalysis: A Tool to Analyze and Identify Critical Areas of Source Code Repositories (P3)
SBES 2022
Identifying Source Code Files that Indicate Architectural Technical Debt (P4)
A qualitative process to validate the ATDCodeAnalyzer method using SATD in Issues related to Architectural Issues (P5)
RQ #
Study
Study
Research Methodology
Publication Venue

Figure 2 – Thesis overview: Research questions, methodologies and publications venues

related to TD accumulation in large-scale Global Software Engineering (GSE) projects. The main contributions of this study are (i) the main factors related to GSE that can contribute to Technical Debt accumulation; and (ii) the Technical Debt Management process documentation that can be used as a model for other companies that want to start a TDM process.

The research paper P3 (SysRepoAnalysis: A tool to analyze and identify critical areas of source code repositories) is a tool to automate the process of extracting information from commits and modified files in the code repository, generating metrics that aid in identifying critical source code files. The main contributions of this study are (i) a tool to automate the process of collecting data from commits and modified files in a Git repository over time; (ii) an automatic way to calculate code metrics related to cyclomatic complexity, file frequency in commits, and LOC modification based on historical analysis; and (iii) a software visualization for code repository to show areas related to the maintainability effort over time.

The research paper P4 (Identifying source code files that indicate Architectural Technical Debt) aims to identify source code artifacts that indicate the presence of ATD using the proposed method in this thesis in a real-world industrial case. The main contributions of this study are (i) an automatic extraction approach using only the source code repository to get information about ATD items; and (ii) an empirical evaluation of the approach, through its use in the Cassandra project and also with feedback from developers involved with the project.

Finally, the research paper P5 (A qualitative process to evaluate the ATDCodeAnalyzer method using SATD in Issues related to Architectural Issues) investigated the effectiveness of the ATDCodeAnalyzer method for identifying Architectural Technical Debt (ATD). The study used four real-world Apache project repositories to analyze how the method prioritizes qualitative insights over purely quantitative metrics. We employed Self-Admitted Technical Debt (SATD), to extract explicit developer comments about code issues, to validate ATDCodeAnalyzer's accuracy. The process involved linking ATD-impacted files with SATD comments and pinpointing codebase areas with architectural problems. This was further refined by triangulating data with issues from the project's issue tracker.

1.8 The Proposed Method

We proposed an automated method (ATDCodeAnalyzer) for identifying source code files impacted by ATD using information and source code files from version control systems. This method involves five phases described in the following paragraph.

In **Phase 1**, historical data is extracted from commits and modified files in the Git repository. Next, in **Phase 2**, source code files with Architectural Smells (AS) are selected, and specific metrics are calculated from those files. After that, in **Phase 3**, quartiles are calculated, and critical files related to ATD are selected. Additionally, in **Phase 4**, critical source code files and their dependent files are analyzed using co-change. Finally, in **Phase 5**, possible source code files with ATD are reported.

This proposed method (fully described in Chapter 8) allows developers to systematically identify source code artifacts impacted by ATD, even in the absence of architectural documentation. We used the Design Science process to propose the method, and the method has been applied in a study analyzing data from the Apache Cassandra repository, and the identified critical files were confirmed by experienced developers to be related to ATD in the project.

1.9 Publications

At the moment, we have three research papers (P1, P2, and P3) published in prestigious software engineering venues:

– Sousa, A., Rocha, L., Britto, R., Gong, Z. and Lyu, F., 2021, March. Technical Debt in Large-Scale Distributed Projects: An Industrial Case Study. In 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 590-594). IEEE.

– Sousa, A., Ribeiro, G., Avelino, G., Rocha, L. and Britto, R., 2022, October. SysRepo-Analysis: A tool to analyze and identify critical areas of source code repositories. In Proceedings of the XXXVI Brazilian Symposium on Software Engineering (SBES) (pp. 376-381).

– Sousa, A., Rocha, L. and Britto, R., 2023, September. Architectural Technical Debt-A Systematic Mapping Study. In Proceedings of the XXXVII Brazilian Symposium on Software Engineering (SBES) (pp. 196-205).

In addition, the research paper P4 (Investigating source code files as an indicator of Architectural Technical Debt), was submitted to PROFES 2024 (International Conference on Product-Focused Software Process Improvement) and is under review. Also, we have another paper P5 (A qualitative process to validate the ATDCodeAnalyzer method using SATD in issues related to Architectural Issues) that is planned to be submitted to a scientific journal.

1.10 Thesis Structure

The rest of this document is structured as follows:

Chapter 2: is the background chapter lays the foundational concepts necessary for understanding the core elements presented in this work. **Chapter 3:** the related works section encompasses the primary tools, frameworks, and case studies associated with ATD. **Chapter 4** is a literature review that discusses works related to ATD. This chapter summarizes the main findings from previous studies on this topic. **Chapter 5:** presents an industrial case study that aims to understand the factors related to TD in large-scale software projects. The study was conducted in a real-world setting, and the results are discussed in this chapter. **Chapter 6:** describes the proposed method for identifying ATD, including the artifacts and activities involved in the process. **Chapter 7:** introduces a tool called SysRepoAnalysis, developed to facilitate data extraction and metrics calculation for the proposed method. **Chapter 8:** describes how the proposed method was performed, in a real-world industrial case, to identify source code files affected by ATD. **Chapter 9:** describes the evaluation of the proposed method using qualitative techniques in issues related to architectural problems with a focus on Self-Admitted Technical Debt (SATD). **Chapter 10:** concludes the thesis by summarizing the key points discussed in the previous chapters, discusses the research questions, the implications of the findings and outlining the next steps in the research.

2 BACKGROUND

The research conducted in this work focuses on several key areas. In Section 2.1, we examined Technical Debt, covering the main concepts and aspects such as the types of TD and effective management techniques. In Section 2.2, we explored Software Architecture, defining its importance and outlining guidelines for best practices. In Section 2.3, we investigated Architectural Technical Debt, discussing its definition, the importance of identifying and managing ATD, and the associated challenges and gaps. In Section 2.4, we highlight the concept of Self-Admitted Technical Debt (SATD), its main types, and the significance of tracking SATD, which aids in identifying technical debt within the source code directly from code repositories. In Section 2.5, we studied Mining Software Repositories (MSR), defining their purpose, leading techniques, and methods, and explaining their essential role. In Section 2.6, we examined Software Smells, providing an overview of how to use smells to identify software problems and discussing potential gaps and opportunities in this area. Finally, in Section 2.7, we examined Architectural Smells, defining their main types and classifications, and emphasizing the importance of discovering architectural issues.

2.1 Technical Debt

Technical Debt (TD) is a metaphor reflecting technical compromises that sacrifice a software product's long-term health to achieve short-term benefits. It is crucial to manage TD to avoid software degradation.

Li *et al.* (2015a) define TD as follows: "*In software-intensive systems, Technical Debt is a set of design or implementation constructs that are expedient in the short term, but that can be configured in a context that can impact future changes at a high cost. Technical Debt presents a contingent debt where the impact is limited to the system's internal quality, primarily in the maintenance and evolution of the system*".

According to Ampatzoglou *et al.* (2015), there are three important concepts in TD: (i) debt, the amount of money (in financial terms) owed by one party to another party, where the obligation of the debtor to repay in the future; (ii) interest, the additional effort that is needed to be spent on maintaining the software due to technical debt not paid; and (iii) the principal, defined in terms of TD context as: "*The effort that is required to address the difference between the current and the optimal level of design-time quality, in an immature software artifact or the*

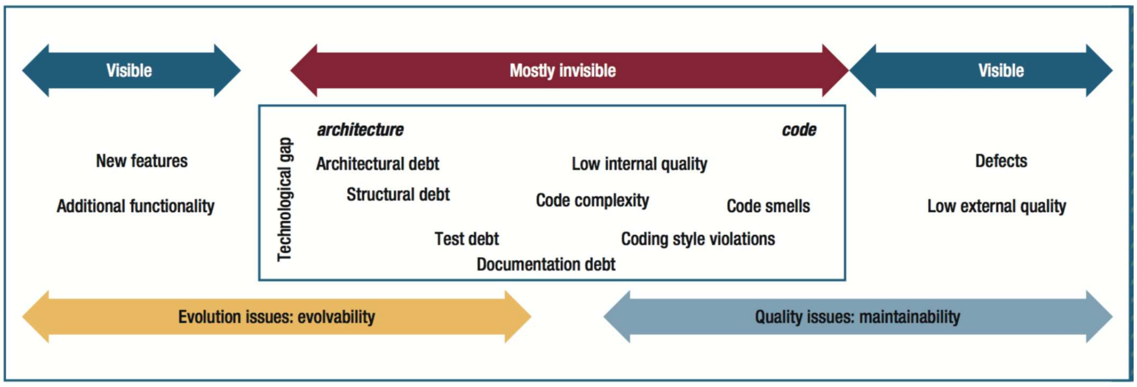


Figure 3 – Technical Debt Landscape (KRUCHTEN *et al.*, 2012)

complete software system".

Kruchten *et al.* (2012) further characterize the TD by organizing the overview landscape, as depicted in Figure 3. In particular, the proposed landscape organization describes the possible software improvements of a system from a given state. This organization distinguishes between visible (e.g., new features and low external quality) and invisible elements (e.g., architectural debt, documentation debt, and code complexity).

Li *et al.* (2015a) proposed a classification of 10 TD types: Requirement TD, Architect TD, Design TD, Code TD, Test TD, Build TD, Documentation TD, Infrastructure TD, Version TD, and Defect TD. They also identified the following activities as the most important for managing technical debt: TD identification, TD measurement, TD prioritization, TD monitoring, TD prevention, TD payment, TD documentation, and TD communication. Hence, it is necessary to standardize the definition of TD types and define a basic set of activities that manage and control TD in the software development life-cycle process.

2.2 *Software Architecture*

According to Perry e Wolf (1992), the software architecture consists of the formula (2.1) and the explanation of its terms. According to this definition, software architecture is a set of architectural elements that have some organization. The elements and their organization are defined by decisions taken to satisfy objectives and constraints.

Three types of architectural elements are highlighted: (i) processing elements - which use or transform information; (ii) data elements - which contain the information to be used and transformed; and (iii) connection elements - which connect elements. The organization dictates the relationships between the architectural elements. These relationships restrict the interaction of the elements in order to achieve the objective of the system.

$$Architecture = (Elements, Organizations, Decisions) \tag{2.1}$$

For Bass *et al.* (2003), "*the architecture of a program or computer systems is the structure or structures of the system, which is composed of software elements, the externally visible properties of these elements, and the relationships between them*". Although different from the definition of Perry e Wolf (1992), this definition makes explicit the role of abstraction in architecture (when it mentions externally visible properties) and also the role of multiple architectural views (system structures). It is essential to note the use of the term "software elements" as the fundamental pieces of architecture.

Another definition of software architecture is given by Taylor *et al.* (2009), where it is said that software architecture is the set formed by the main design decisions made concerning the software under development or in evolution. Design decisions represent aspects of software development or evolution related to the structure, functional behavior, interaction, non-functional properties, and software implementation. They are called principals, those design decisions relevant from the software architecture point of view. These decisions are also referred to as architectural decisions.

The architectural design brings numerous benefits to software development, among which we highlight the ones mentioned by Taylor *et al.* (2009): (i) improved communication - architecture can be used as a central point for discussion between stakeholders in the software; (ii) it allows previous analysis - with the existence of an architectural project, it is possible to evaluate the software against its objectives even before it is built; (iii) favors large-scale reuse - a family of software can reuse a well-planned software architecture; and (iv) documentation improvement - the architectural design documentation serves as a guide for the implementation, verification, validation, and evolution of the software.

The software architecture establishes principles and guidelines that must be followed during the software's evolution and maintenance, where the non-continuity or respect for each one of these principles that were initially established can result in the degradation of the software architecture.

2.3 *Architectural Technical Debt*

In large-scale software development, the rules and decisions of architectural design assume a central role in the management of the Technical Debt of software, more precisely referring to the Architectural Technical Debt (BESKER *et al.*, 2018). Generally, TD occurs due to a lack of knowledge of the team, failure to communicate the reference artifacts produced, the omission of activities due to lack of resources (time, money, or people), among other factors. Thus, to prevent technical debts from increasing and putting the software project at risk, it is necessary to manage them (i.e., identify, estimate, prioritize, resolve, and monitor them).

ATD commonly refers to violations of good practice, architectural consistency, integrity, or naive implementation of architectural techniques (MARTINI; BOSCH, 2015a). All these violations can compromise modularity, reusability, analyzability, modifiability, stability and evolution during the software architecture process (LI *et al.*, 2015a). It is difficult to measure because it is a cross-cutting interest in the software development cycle (NORD *et al.*, 2012). With this, it only becomes visible when there are complications of maintenance, evolution, or operation of the software (LI *et al.*, 2015a).

Besker *et al.* (2018) performed a systematic literature review on ATD, noting the need for software companies to support continuous and rapid delivery of software with added value to the customer, both in the short and long term. However, it is possible to observe the impediment of such deliveries during the evolution and maintenance of existing systems hampered by what was recently called TD. More specifically, the ATD, since this term has received great attention in recent years due to its significant impact on the success of the system, and, if it is not addressed, it can cause costly repercussions. Within this context, a systematic study was carried out to understand the underlying factors of ATD. In this study, a descriptive model was made to illustrate and explain different questions about ATD to synthesize and compile research efforts developed in the area. Also, they showed that it is challenging to manage ATD, but it is necessary to repay this kind of debt to avoid ATD accumulation and suggested that there are at least five activities to aid ATD management: (i) ATD identification: the ATD items are detected and described. (ii) ATD measurement: it is calculated the cost to fix the ATD items. The ATD items are analyzed and estimated. (iii) ATD prioritization: the ATD items are sorted according to defined criteria (e.g. importance). (iv) ATD repayment: the software architect makes a decision to repay the ATD item. (v) ATD monitoring: the ATD items are monitored over time regarding their costs and benefits.

2.4 Self-Admitted Tecnical Debt

Self-admitted technical debt (SATD) is the deliberate recognition by developers within the source code, usually through comments or annotations. It represents a conscious decision to postpone an optimal solution in favor of a quicker or simpler fix. For instance, developers promptly identify technical debt by inserting comments like "FIXME" or "TODO," indicating areas needing immediate attention. These comments directly address the code segments responsible for the introduced technical debt. Such decisions are often influenced by time constraints, limited resources, or expertise boundaries encountered during development (MALDONADO; SHIHAB, 2015). Brief examples of these comments are: *"TODO: - This method is too complex, lets break it up"* from ArgoUml, and *"Hack to allow entire URL to be provided in host field"* from JMeter (MALDONADO; SHIHAB, 2015), (SHNEIDERMAN; WATTENBERG, 2001), (WEHAIBI *et al.*, 2016).

Potdar e Shihab (2014) introduced Self-Admitted Technical Debt (SATD) as a distinct subset of Technical Debt (TD), where developers annotate the appearance of TD, often through code comments. Rantala *et al.* (2020) refined this concept, introducing Keyword-Labeled SATD (KL-SATD), focusing on specific keywords like "TODO," "FIXME," or "HACK." Wehaibi *et al.* (2016) explored the association between defects and SATD, revealing that files with SATD undergo a higher frequency of defect-fixing activities post-SATD introduction. Despite this, the overall defect rate induced by SATD changes was lower than changes without SATD. They also highlighted a correlation between SATD introduction and increased software complexity.

In a comprehensive empirical study, Bavota e Russo (2016) systematically classified a substantial sample of SATD comments into various TD categories. They observed that SATD comments primarily indicated Code Debt, followed by comparable instances of Defect Debt and Requirement Debt, trailed by Design Debt, Documentation Debt, and Test Debt. Their classification relied on analyzing SATD comments. Moreover, they found no significant correlations between code file quality and SATD instances, yet noted a trend of SATD persisting in systems for extended periods, with an increasing frequency of SATD introductions over project lifetimes.

Iammarino *et al.* (2021) explored the relationship between refactoring actions and SATD removals, revealing that SATD removals often co-occur with refactoring actions. However, a minor proportion of refactoring actions directly targeted SATD removal, with most occurring incidentally or as a result of the SATD removal process itself.

Tracking Self-Admitted Technical Debt throughout the software development life cycle is crucial for several reasons. In their survey, (SIERRA *et al.*, 2019a) present a concise summary of SATD research as follows:

Awareness and Visibility: Identifying SATD helps teams comprehend the trade-offs made during development, shedding light on areas where short-term gains led to compromises. This understanding allows teams to assess impacts on code quality and potential risks.

Prioritization: By tracking SATD, teams can prioritize attention to areas within the codebase that require immediate focus. Understanding the whereabouts of technical debt aids in resource allocation and planning for necessary improvements or refactoring.

Risk Management: Accumulated SATD can result in increased system complexity, decreased maintainability, and potential system failures. Proactive tracking allows teams to mitigate these risks and prevent their escalation.

Impact on Development Velocity: Unaddressed SATD can impede development progress over time. Strategic tracking enables teams to address technical debt systematically, potentially enhancing overall development speed and efficiency in the long term.

Quality Assurance: Understanding SATD helps in identifying potential defect-prone areas in software. This insight enables quality assurance teams to concentrate testing efforts, reducing the likelihood of issues in critical parts of the software.

Decision-Making Support: Awareness of existing SATD aids in making informed decisions while planning new features or enhancements. It allows teams to weigh the implications of adding more technical debt against the benefits of proposed changes.

Continuous Improvement: Tracking SATD facilitates a feedback loop for continuous improvement. Teams can learn from past instances, refining development practices and strategies to minimize the introduction of new technical debt.

In essence, monitoring SATD throughout the software development life cycle is essential for maintaining code health, ensuring product quality, managing risks, and fostering sustainable development practices.

2.5 *Mining Software Repository*

The Mining Software Repositories (MSR) help discover important information about software projects, allowing you to extract and analyze data available in different software repositories (HASSAN, 2008).

Software repositories contain various software artifacts that cover important aspects like software project management and maintenance (project management documents, mailing lists, and public communication tools). It is usually stored in unstructured repositories where information is dispersed and represented in non-standard texts. It can also contain implementation artifacts (version control system, issue tracker, and bug tracker) that generally have a standardized structure that makes it easier to find information (ROBLES, 2010).

A software repository records activities and meta-information about the manipulated artifacts. A software repository analysis can aid in the task of defect prediction (ZHANG *et al.*, 2014), effort prediction, text mining, and discovering trends about the data recorded (STEIDL *et al.*, 2014; GIL *et al.*, 2012).

Software repositories provide a large amount of data containing software changes throughout its evolution. Those repositories can be effectively used to extract and analyze pertinent information and derive conclusions related to the software history or its current snapshot (HEMMATI *et al.*, 2013).

2.6 *Software Smells*

Smells in software systems impair software quality and make them hard to maintain and evolve. The software engineering community has been widely studying this phenomenon for the last few years. In this context, Sharma e Spinellis (2018) presented a comprehensive overview of the current knowledge and practices related to software smells. The authors conducted a survey of 445 primary studies and analyzed the information and observations about software smell. They explored the definitions, causes, effects, and detection mechanisms of smells presented in the literature. The study also identified the challenges and opportunities in software smell detection and prevention practices.

The first time that the term “code smell” was cited by Kent Beck, it caused a strong impact on the software engineering community. Hence, since then it is widely used by the community (FOWLER, 2018). Also, it is defined informally as “certain structures in the code that suggest the possibility of refactoring”. Later, various researchers gave diverse definitions of software smells.

Sharma e Spinellis (2018) explored and identified the following dimensions of software smells in the literature:

– **Indicator**: Authors define smells as an indicator to or a symptom of a deeper design problem;

- **Poor solution**: The literature describes smells as a suboptimal or poor solution;
- **Violates best practices**: According to authors such as Suryanarayana *et al.* (2014) and Sharma *et al.* (2016), smells violate recommended best practices of the domain;
- **Impacts quality**: Smells make it difficult for a software system to evolve and maintain (YAMASHITA, 2014; KHOMH *et al.*, 2011). It is commonly agreed that smells impact the quality of the system;
- **Recurrence**: Many authors define smells as recurring problems.

Sharma e Spinellis (2018) also identified and cataloged a wide range of smells (close to 200 examples) made available online[1]. They also classified the existing smells into four main categories:

- **Effect-based**: Mantyla *et al.* (2003) classified smells based on their effects on software development activities. The categories provided by the classification include bloaters, couplers, and change preventers;
- **Principle-based**: Ganesh *et al.* (2013) and Suryanarayana *et al.* (2014) classified design smells based on the primary object-oriented design principle that the smells violate. The principle-based classification divided the smells into four categories, namely: abstraction, modularization, encapsulation, and hierarchy smells;
- **Artifact characteristic-based**: proposed a smell classification based on characteristics of the types. Categories such as data, interfaces, responsibility, and unnecessary complexity include in his classification. Similarly, Karwin (2010) classified SQL anti-patterns in the following categories — logical database design, physical database design, query, and application development anti-patterns;
- **Granularity-based**: Moha *et al.* (2009) classified smells using two-level classification. At first, a smell is classified into an either inter-class or intra-class category. The second level of classification assigns non-orthogonal categories i.e., structural, lexical, and measurable to the smells. Similarly, Brown *et al.* (1998) discussed anti-patterns classified into three major categories — software development, software architecture, and software project management anti-patterns.

Sharma e Spinellis (2018) also have explored factors that introduce smells in software systems, for example, lack of skill or awareness, frequently changing requirements, language,

[1] https://www.tusharma.in/smells

platform, or technology constraints, knowledge gap, the process or lack of process, schedule pressure, priority to features over quality, politics, team culture, poor human resource planning. Finding that the main impacts on software products are maintainability, effort/cost, reliability, change proneness, testability, and performance.

Sharma e Spinellis (2018) categorized existing smell detection methods into five groups:

- **Metrics-based**: A typical metrics-based smell detection method takes source code as the input, prepares a source code model (such as an AST - Abstract Syntax Tree) typically by using a third-party library, detects a set of source code metrics that capture the characteristics of a set of smells, and detects smells by applying a suitable threshold (MARINESCU, 2005);
- **Rules/heuristic-based**: Smell detection methods that define rules or heuristics (MOHA *et al.*, 2009) typically takes source code model and sometimes additional software metrics as inputs;
- **History-based**: Some authors have detected smells by using source code evolution information (PALOMBA *et al.*, 2014). Such methods extract structural information of the code and how it has changed over a period of time. This information is used by a detection model to infer smells in the code. For example, by applying association rule mining on a set of methods that have been changed and committed often to the version control system together, divergent change smell can be detected (PALOMBA *et al.*, 2014);
- **Machine learning-based**: Various machine learning methods such as Support Vector Machines (MAIGA *et al.*, 2012) and Bayesian Belief Networks (KHOMH *et al.*, 2009) have been used to detect smells. A typical machine learning method starts with a mathematical model representing the smell detection problem. Existing examples and source code model could be used to instantiate a concrete populated model. The method results in a set of detected smells by applying a chosen machine learning algorithm on the populated model;
- **Optimization-based**: Approaches in this category apply optimization algorithms such as genetic algorithms (OUNI *et al.*, 2015) to detect smells. Such methods apply an opimization algorithm on computed software metrics and, in some cases, existing examples of smells to detect new smells in the source code.

Finally, Sharma e Spinellis (2018) identified the following gaps and research opportunities in the present set of tools and techniques:

- Existing literature does not differentiate between a smell (as an indicator) and a definite quality problem;
- The community believes that the existing smell detection methods suffer from high false-positive rates. Also, existing methods cannot define, specify, and capture the context of a smell;
- The currently available tools can detect only a very small number of smells. Further, most of the tools largely only support the Java programming language;
- Existing literature has produced inconsistent smell definitions. Similarly, smell detection methods and the corresponding produced results are highly inconsistent;
- The current literature does not establish an explicit connection between smells and their impact on the productivity of a software development team.

2.7 Architectural Smells

Architectural Smells (AS) can be seen as the code smells metaphor at the architecture level. AS represent the violation of design principles or decisions that impact internal software qualities with significant adverse effects on maintenance and evolution costs (AZADI *et al.*, 2019).

Azadi *et al.* (2019) proposed a catalog of twelve AS (Cyclic Dependency, Hub-like Dependency, Unstable Dependency, Cyclic Hierarchy, Scattered Functionality, God Component, Abstraction without Decoupling, Multipath Hierarchy, Ambiguous Interface, Unutilized Abstraction, Implicit Cross-module Dependency, and Architecture Violation) organized in three classifications:

- **Modularity**: Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.
- **Hierarchy**: Hierarchy is a ranking or ordering of abstractions, where an abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provides crisply defined conceptual boundaries, relative to the perspective of the viewer.
- **Health Dependency Structure**: The dependency structure of a (sub)system is considered unhealthy when it promotes a chain of changes in the system each time it is modified.

Besides, Azadi *et al.* (2019) provides a catalog of AS based on "violated principles" and found eleven AS that are outstanding overall. From this catalog, we selected the cycle

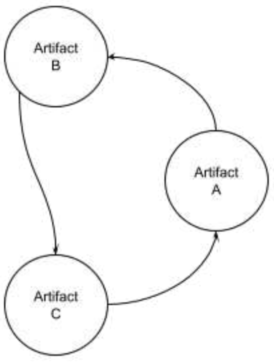


Figure 4 – Example of Cycle dependency (Adapted from (SAS *et al.*, 2022b))

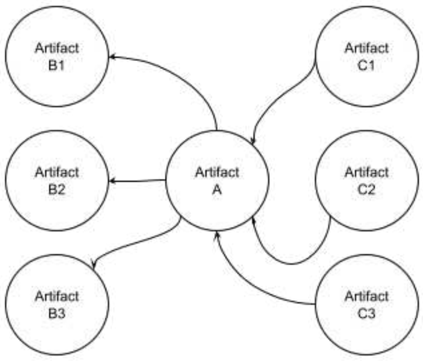


Figure 5 – Example of Hub-like dependency (Adapted from (SAS *et al.*, 2022b))

dependency and hub-like dependency because these two are more important because of the impact on modularity and health dependency structure. Where Cyclic Dependency (CD) is this smell that arises when two or more architectural components strut depend on each other directly or indirectly. We can see in Figure 4 an example of Cycle Dependency among artifacts A, B, and C. Hub-Like Dependency (HLD) is the smell that occurs when an abstraction or a concrete class has (outgoing and ingoing) dependencies with a large number of other abstractions or concrete classes. We can see in Figure 5 an example of Hub-like Dependency affecting component A, with the afferent (incoming) dependencies on the right and (outgoing) efferent dependencies on the left.

3 RELATED WORK

In this chapter, we discuss studies that either report approaches for detecting ATD or share conceptual similarities with our investigation.

3.1 Overview

Identifying and monitoring ATD items has been on the agenda of Technical Debt researchers in recent years. As software systems become more complex, it is essential to identify the software artifacts that accumulate ATD (LI *et al.*, 2014),(LI *et al.*, 2015), (MARTINI *et al.*, 2015),(VERDECCHIA *et al.*, 2018), (MARTINI *et al.*, 2018b), (VERDECCHIA *et al.*, 2021). To do so, reduce the maintenance effort and evolution effort of software systems. To provide information to professionals responsible for the maintenance and evolution of complex software systems, it is necessary to investigate how the identification of software artifacts that indicate ATD is done. We discuss studies that either report approaches for detecting ATD or share conceptual similarities with our research.

3.2 Tools

Fontana *et al.* (2016b) provided an experience report using static code analysis tools (Sonargraph[1], SonarQube[2], and InFusion[3]) describing experimentation using these tools to evaluate ATD via Architectural Smells (AS) comparing quality index, software metrics, and code smells. They found that the quality index still needs to be completed to evaluate the software projects about ATD. However, the metrics and index found can be useful to improve tools and methods that want to summarize the state of tools in identifying ATD.

Ludwig *et al.* (2017) proposed a tool to measure and visualize architectural complexity based on the propagation cost to identify sources of TD via static software code metrics to describe, measure, and visualize architectural complexity metrics not currently found in other tools. They developed a quality model that focused on architectural complexity and relies on only a small set of essential software metrics that address the primary sources of TD.

Zitzewitz (2019) proposed the tool Sonargraph to check Arquictural rules via DSL (Domain Specific Language) and source code analysis to identify ATD via software metrics

[1] https://www.sonarqube.org
[2] https://www.hello2morrow.com/products/sonargraph
[3] https://www.intooitus.com/products/infusion

like *cyclomatic complexity*, *propagation cost*, and *cyclicity*. Among them are also some metrics that were specifically developed to measure maintainability, like “Maintainability Level” and “Structural Debt Index”. However, this tool does not provide history change analysis over the source code versions.

Sharma *et al.* (2020) studied Architectural Smells (AS) characteristics to investigate correlation and causation relationships between architecture and design smells. The authors implemented the tool *Designite* to detect AS using mining software repository techniques in open-source repositories to investigate seven architecture, and 19 design smells. The authors found a high correlation between AS and design smells, showing the causality analysis reveals that the design smells cause AS. Also, the authors found that there is a negative impact of AS on maintenance efforts in terms of the increased number of implementation issues and code commits.

3.3 Frameworks

Martini *et al.* (2018b) proposed a framework to identify and estimate ATD using a case study in a large software company about components modularization and how new applications use these components. They created a measurement system that identified the need for refactoring according to the stakeholders’ goals. Then, the authors developed a formula quantifying the benefits of refactoring in terms of development months related to indicators to repaying ATD about modularity.

Roveda *et al.* (2018) proposed a framework to calculate an ATD Index related to the evaluation of architectural violations. The index is based on the detection of AS, their criticality, their history and dependents metrics. This index can be used to identify and prioritize the most critical classes or packages in the projects; in this way, the developers can easily identify and focus their attention on them also this index provides a comparison to estimate the severity of an AS.

Verdecchia *et al.* (2020) proposed a comprehensive method to calculate the technical debt principal index of a system, employing statistical analysis. Their approach aimed to be language and tool independent, allowing for composability of tools at various levels of analysis, such as class, package, or module. They formulated the problem mathematically by treating the output of any tool as a set of architectural rules applied to each artifact in the system. Their mathematical model incorporated granularity levels and clusters of architectural rules known as architectural dimensions. While offering several advantages, this approach had some drawbacks, including its dependency on a benchmark of software projects, necessitating continuous updates to calculate certain statistics used in the index calculation. The authors validated their approach in a subsequent study using questionnaires.

3.4 Case studies

Martini *et al.* (2018a) performed a multiple case study on several AS detected in four industrial projects to evaluate the impact of ATD. They used questionnaires, interviews and thorough inspection of the code with the practitioners. The authors created correlation variables related to impact factors, overall negative impact, side effects, effort, and refactoring priority to evaluate the negative impact of TD related to AS. They found that *cyclic dependency* was the AS with the worst impact and the most expensive to refactor, and *hub-like dependency* also has a similarly strong negative impact.

Toledo *et al.* (2021a) performed an exploratory multi-case study to identify ATD via interviews with practitioners working with microservices. The authors showed that poor business logic among services, poorly designed shared databases, lack of data traceability mechanisms, poorly designed APIs, and shared libraries are the leading root causes of ATD items in micro-services and impact interest in database, dependencies components, API complexity, coupling among services, and dependencies among teams. Then, the authors proposed a guide to developing microservices systems to manage ATD, helping identify the consequences and payment of ATD items.

Sas *et al.* (2022a) performed a case study on 31 open-source Java Systems to check the relation between AS and source code changes. They studied the frequency and size of changes to check the correlation between the presence of the selected set of AS. They found that 87% of the analyzed commits have more change and increase with the number of smells increases over time. Also, the introduction of AS increases the change frequency of the source code affected by the AS, and the size of changes is significantly higher in smelly artifacts than in non-smelly ones.

3.5 Conclusions

Tables 1 and 2 provide a summary of the distinctions between existing work on ATD tools, frameworks, case studies, and our research. Our work advances the existing state-of-the-art in two significant ways. Firstly, we introduce an automated approach, a departure from the conventional reliance on manually-set thresholds or expert analysis. Secondly, we evaluate our approach through interviews with software developers engaged in an industrial project. Additionally, we extend this evaluation to three other prominent real-world open-source projects within the distributed systems domain, conducting extensive measurements and comparisons. Furthermore, our approach offers a publicly available replication kit, enabling the implementation of our methodology.

Previous works have used various methods and tools to analyze the source code, and documents of software systems and generate ATD indicators, such as customized formulas and expert analysis based on questionnaires and interviews. However, there is a gap in the existing literature as there is a lack of automated approaches that can extract source code artifacts affected by ATD over time without intervention or expert analysis. This study aims to address this gap by proposing an automated approach that extracts source code impacted by ATD through historical analysis, code metrics, and architectural smells. This study uses both quantitative data and qualitative data through interviews to evaluate the proposed approach.

Table 1 – Comparing Tools, Frameworks and this work

Work	Focus	Metrics Used	Capabilities	Limitations
Fontana *et al.* (2016b)	Evaluation of ATD via Architectural Smells.	Quality index, software metrics, code smells.	Evaluation, code assessment, tool comparison.	Need for improved quality index for ATD evaluation.
Ludwig *et al.* (2017)	Measure and visualize architectural complexity.	Propagation cost, unique architectural complexity metrics.	Architecture visualization, TD identification.	Limited software metrics, focus on architectural complexity.
Zitzewitz (2019)	Check architectural rules via DSL.	Cyclomatic complexity, propagation cost, cyclicity.	Architecture validation, maintainability assessment.	Lack of historical change analysis over code versions.
Sharma *et al.* (2020)	Detection of AS using mining software repository.	Architecture and design smells.	Causation analysis, correlation study.	Emphasis on correlation between AS and design smells.
Martini *et al.* (2018b)	Identify and estimate ATD.	Components modularization, refactoring indicators and development months	Refactoring need identification, Quantifying refactoring benefits.	Dependency on stakeholder goals, Specific to modularization and applications.
Roveda *et al.* (2018)	Calculate ATD Index related to architectural violations.	Detection of Architectural Smells (AS), criticality, history, and metrics.	AS identification, Prioritization of critical classes/packages.	Focuses on severity of AS, Relies on AS detection and historical data.
Verdecchia *et al.* (2020)	Calculate technical debt principal index of a system via statistical analysis.	Mathematical formulation, architectural rules, granularity levels.	Language/tool independence, Multiple levels of analysis, Composability of tools.	Dependency on software project benchmark for certain statistics used in the index.
This work	On extracting critical classes using only code analysis from the repository as well as its historical analysis.	Architectural Smells, Amount modified of LOC, Frequency of Files in Commits and Cyclomatic Complexity.	We extract the critical classes affected by ATD automatically without necessity of specialist analysis.	We need a large amount of commits to be more precisely.

Table 2 – Comparing Case Studies about ATD and this work

Case Study	Approach	Methodology	Findings
Martini *et al.* (2018a)	Multiple case study on AS detected in industrial projects.	Questionnaires, interviews, code inspection.	Identifies impact factors, worst AS impact (cyclic dependency), expensive to refactor, hub-like dependency also has strong negative impact.
Toledo *et al.* (2021a)	Exploratory multi-case study on ATD in microservices.	Interviews with practitioners, root causes identification.	Identifies root causes of ATD in microservices, impacts on database, dependencies, APIs, proposes guide for managing ATD in microservices.
Sas *et al.* (2022a)	Case study on relation between AS and source code changes.	Analysis of frequency and size of changes.	Correlation between presence of AS and increased change frequency, larger changes in smelly artifacts compared to non-smelly ones.
This work	Using Design Science process, we conducted a case study employing Apache Cassandra and validated it with Ericsson Developers, who maintain this project. Additionally, we performed experiments on four real-world Apache projects, to evaluate the proposed method.	Using automatic process without expert intervention, we extracted critical classes from commits and analyzed the metrics' behavior over time.	We observed an increase in the analyzed metrics over time. Furthermore, we identified a relationship among LOC modifications in commits, modified files in commits, and the time resolution of these issues with architectural impact.

4 SYSTEMATIC MAPPING STUDY IN ARCHITECTURAL TECHNICAL DEBT

This chapter presents some of the works identified using a systematic mapping of the literature to identify the leading research carried out on Architectural Technical Debt. The methodology and process of the systematic mapping of the literature on ATD are summarized in the first section, while the second section focuses on works that identify types of ATD. The other sections present studies on how to measure and monitor ATD, as well as works related to tools and methods used to identify and analyze ATD. In addition, there are works that seek to calculate the cost of paying the technical debt of ATD items. The chapter concludes with a summary in the final section.

4.1 Systematic Mapping Study in ATD

We report our Systematic Mapping Study (SMS) findings that address the following research question: *How to identify and monitor ATD items in complex software systems?* Answering this question can help find what the community has been studying about identification and monitoring ATD and help to handle the related problems of this subject.

The first version of the systematic mapping study on ATD was conducted during the second half of 2020. The second version was generated to update the research up until the first quarter of 2022. After that, we updated the SMS in mid-2024 to gain an overview of recent years' research on ATD. In addition, we sought to identify trends, challenges, and open questions in this field of research.

4.2 Related Work

TD and ATD have been on the research agenda in recent years. We choose four other secondary studies in these areas. We will discuss the goals and results of each study below.

In their SMS, Li et al. (LI *et al.*, 2015a) collected 94 studies on TD and TD management and proposed a classification to understand the main aspects of TD and provide an overview of TD management in its current state. They proposed a classification of 10 TD types: Requirement TD, Architect TD, Design TD, Code TD, Test TD, Build TD, Documentation TD, Infrastructure TD, Version TD, and Defect TD. They also identified the following 8 activities as the most important for managing technical debt: TD identification, TD measurement, TD prioritization, TD monitoring, TD prevention, TD payment, TD documentation, and TD communication. The authors observed that Code TD is the most studied subject, and the main three TDM activities are identification, measurement, and repayment. They also noted that the term "debt" is interpreted differently regarding software development life-cycle and business factors, making it difficult to interpret TD. The authors found that code analysis, calculation model, and refactoring are the main techniques used to manage and pay off TD. Finally, they identified a need for more empirical studies in the industry and more specific tools to understand the TDM process better.

Alves et al. (ALVES *et al.*, 2016) conducted an SMS on TD and its management, highlighting the types of TD and indicators helpful in identifying TD. The authors observed essential aspects of identifying the types of TD. For example, the TD types of indicators existing in the software projects and the strategies that have been developed for the management of this debt. For types of TD, the authors found: Design debt, Architecture debt, Documentation debt, Test debt, Code debt, Defect debt, Requirements debt, Infrastructure debt, People debt, Test automation debt, Process debt, Build debt, Service debt, Usability debt, and Versioning debt. The authors observed that papers were highly concentrated on architecture, design, documentation debt, and some papers on code and test debt. Besides, strategies are used to identify TD to discover TD items when analyzing the different artifacts created during the development of a software project. The following indicators are most cited: Code Smell, highlighted God class, Code Complexity, and Duplicated code. The following artifacts are used more: Source Code, Documentation, Test Report, Bug Report, System Architecture Specification, Backlogs, Commits, and Change Report.

Besker et al. (BESKER *et al.*, 2018) presented a Systematic Literature Review (SLR) to elaborate a scheme about the current knowledge in ATD. More specifically about the debt, interest, principal, challenges, and solutions to ATD management. Creating new knowledge to help researchers develop a guide with new directions about ATD and help practitioners create a unified model to identify, evaluate problems, consequences, and challenges of ATD. Their SLR observed that the importance of ATD in software development is that the architectural rules are crucial in developing large-scale software to improve development, testing, and quality activities. They observed five main categories of ATD: Architectural Dependence, Standards and "Policy" Deformity, Lack of Test and Quality, Subsystem Integration Management Deformity, and QA (Quality assurance) Synergy Conflict. Besides, some adverse effects like the accumulation of technical debt can hinder QA, hinder maintenance, and maintain the software's organization and evolution in new features.

In their SMS, Verdecchia et al. (VERDECCHIA *et al.*, 2018) identified, classified, and evaluated the current state of ATD identification. They found that source code packages, components and connectors, source code classes, and source code files are the main characteristics of ATD identification techniques used in architectural abstraction levels. They also observed that certain ATD identification definitions, such as dependency violation, modularity, compliance violation, change proneness estimation, and customization analysis, are more commonly used. The authors noted that architectural anti-patterns, architectural smells, modularity analysis, evolution analysis, dependency analysis, cost analysis, human knowledge-based analysis, compliance checking, self-admitted analysis, and manual classification are the more common analysis of ATD. Furthermore, they identified that the main inputs for ATD identification analysis are source code, evolutionary data, architectural documents, issue tracker, and human knowledge. The authors also observed a lack of temporal dimension control regarding the evolution of software systems and that ATD identification is a recent problem that researchers and practitioners are addressing.

One limitation of Li et al. (LI *et al.*, 2015a) is that it does not specifically address the identification or management of ATD. Similarly, Alves et al. (ALVES *et al.*, 2016) may not cover all types of ATD, which could limit its usefulness for practitioners. Additionally, while Besker et al (BESKER *et al.*, 2018) and Verdecchia et al. (VERDECCHIA *et al.*, 2018) offer valuable insights into ATD characteristics, they may not provide comprehensive management techniques for ADT in particular. Despite covering some ATD issues, the reviewed papers do not focus on studying the identification and monitoring of ATD items. We, therefore, conduct this research considering the gaps mentioned above by studying, via SMS, important points like types, measurements, monitoring, tools, and methods used to identify and monitor ATD items on software projects. In addition, we proposed a roadmap of ATD that can be useful in guiding the ATD management process.

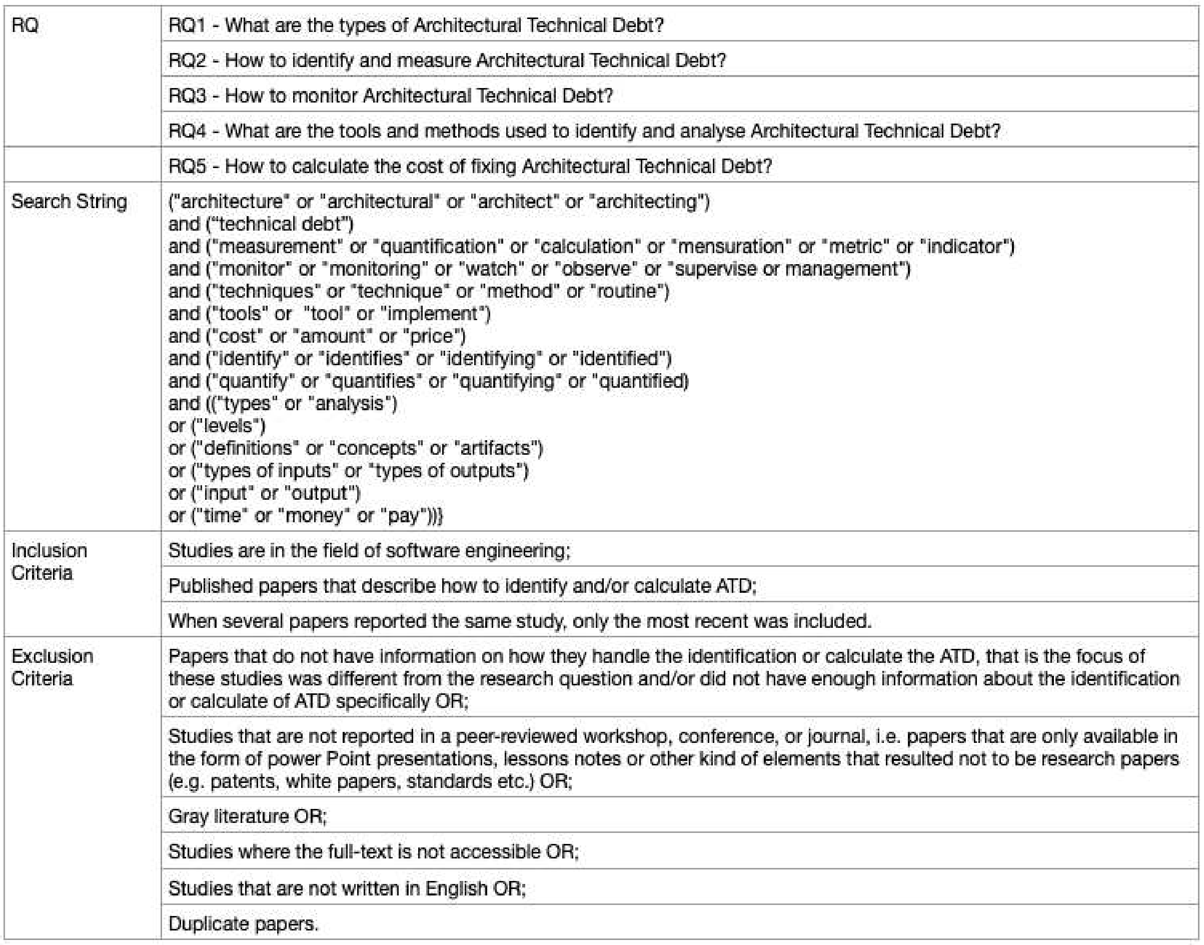
4.3 Research Design

We organized the SMS process in two phases: planning and execution, as seen in the following subsections.

4.3.1 Planning

Previous secondary studies (see Section 2) investigated TD and ATD, albeit with different objectives than our study. They identified many important primary sources of information. We used the SMS strategy (PETERSEN *et al.*, 2008) to identify and analyze the state of the art of ATD, specifically on identifying and monitoring ATD items. In this way, we search the main digital libraries (Scopus, Web of Science, IEEE, and ACM). In addition, we performed searches on Google Scholar and Snowballing technique to converge to the most cited papers. Furthermore, for the remaining mapping phases, we followed practices from software engineering literature studies as suggested in (KITCHENHAM; CHARTERS, 2007) for replicability and audit results. We include defining appropriate research questions, a base search string, inclusion and exclusion criteria, data collection, dataset cleaning, and study selection. All research protocols, data, graphs, tables, and selected studies are available in the replication kit[1]. The Research Questions (RQs), details, and the main aspects of the investigation are available in [Table 3].

Table 3 – Summary of the literature study



RQ	RQ1 - What are the types of Architectural Technical Debt?
	RQ2 - How to identify and measure Architectural Technical Debt?
	RQ3 - How to monitor Architectural Technical Debt?
	RQ4 - What are the tools and methods used to identify and analyse Architectural Technical Debt?
	RQ5 - How to calculate the cost of fixing Architectural Technical Debt?
Search String	("architecture" or "architectural" or "architect" or "architecting") and ("technical debt") and ("measurement" or "quantification" or "calculation" or "mensuration" or "metric" or "indicator") and ("monitor" or "monitoring" or "watch" or "observe" or "supervise or management") and ("techniques" or "technique" or "method" or "routine") and ("tools" or "tool" or "implement") and ("cost" or "amount" or "price") and ("identify" or "identifies" or "identifying" or "identified") and ("quantify" or "quantifies" or "quantifying" or "quantified) and (("types" or "analysis") or ("levels") or ("definitions" or "concepts" or "artifacts") or ("types of inputs" or "types of outputs") or ("input" or "output") or ("time" or "money" or "pay"))}
Inclusion Criteria	Studies are in the field of software engineering;
	Published papers that describe how to identify and/or calculate ATD;
	When several papers reported the same study, only the most recent was included.
Exclusion Criteria	Papers that do not have information on how they handle the identification or calculate the ATD, that is the focus of these studies was different from the research question and/or did not have enough information about the identification or calculate of ATD specifically OR;
	Studies that are not reported in a peer-reviewed workshop, conference, or journal, i.e. papers that are only available in the form of power Point presentations, lessons notes or other kind of elements that resulted not to be research papers (e.g. patents, white papers, standards etc.) OR;
	Gray literature OR;
	Studies where the full-text is not accessible OR;
	Studies that are not written in English OR;
	Duplicate papers.

[1] https://github.com/Technical-Debt-Large-Scale/smsatd2/blob/main/python/analyses/extractionatd.ipynb

4.3.2 Execution

The search was performed in previous digital libraries on April 21st, 2022. It resulted in 719 suggestions of articles, from which we selected 168 because they have detailed context in TD and, more specifically to ATD. The selection of primary studies was conducted using a two-stage screening procedure. In the first stage, only the studies' abstracts and titles were considered by the three authors of this paper, resulting in 134 papers. In the second stage, the full texts were read, resulting in 57 papers. For the first stage (level-1 screening), the number of found studies was equally divided between the three authors. As a result, a new number of studies were judged and updated as potentially relevant. To evaluate the reliability of the inter-rate agreement between the authors, we calculated Cohen's kappa coefficient (FLEISS *et al.*, 2013). The second stage (level-2 screening), performed by the first and the second author, consisted of applying the selection criteria to the full text of the studies selected during the level-1 screening. The total number of found studies in the first stage was equally divided between the two authors. As a result, a new number of studies were judged as relevant. Finally, the first author searched Google Scholar to find relevant papers (applying inclusion/exclusion criteria), applied Snowballing technique, and found more 13 papers. In the second phase conducted in 2022, we identified 70 relevant papers. Finally, in the third phase performed on June 11th, 2024, we searched selected digital libraries for papers related to ATD published between 2022 and 2024. This yielded 15 new papers. These were combined with the 70 papers identified in a previous search conducted on April 21st, 2022, resulting in a total of 85 papers relevant to our research questions (RQs). These selected papers are available on the replication kit[2].

4.4 Results

In this section, we describe the results of the mapping study reported herein, which are based on the data extracted from selected papers.

4.4.1 General Results

Figure 6 shows the number of primary studies appearing each year. The primary studies span from 2012 to 2024. As we can see, there has been an increasing publication through the years, demonstrating the recent interest of researchers and practitioners in the subject. The

[2] https://github.com/Technical-Debt-Large-Scale/smsatd2/blob/main/md/selectedpapers.md

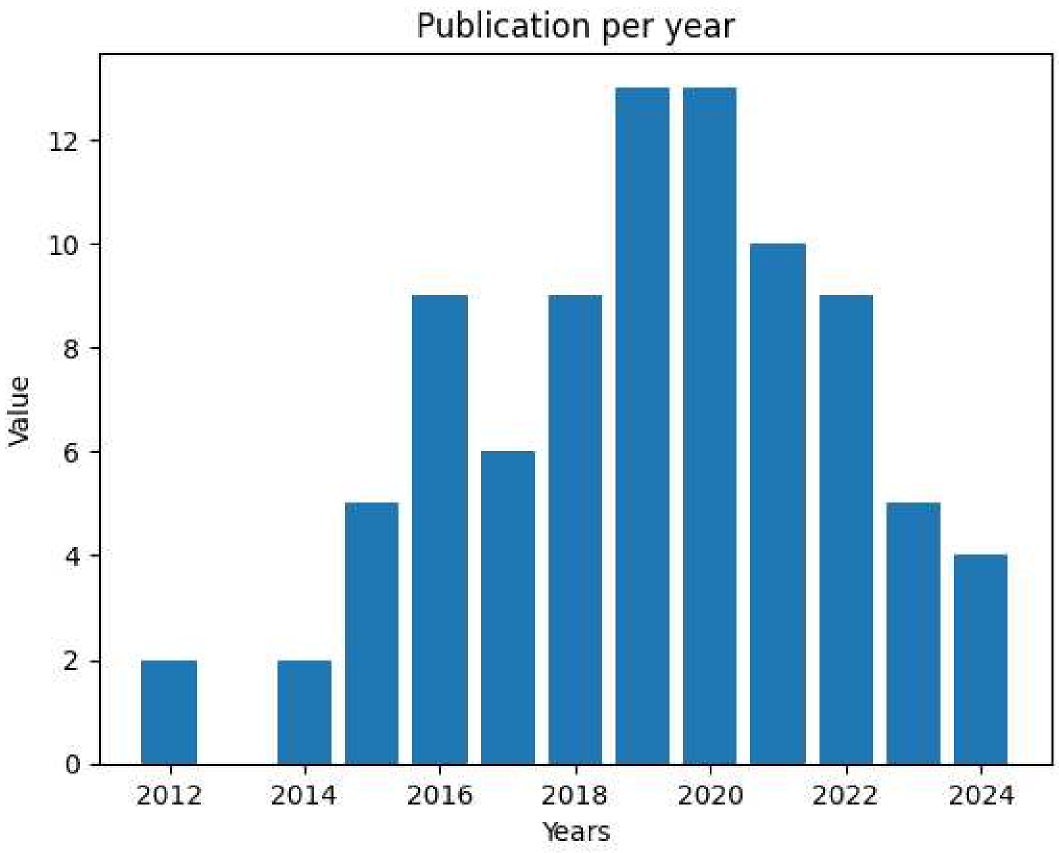


Figure 6 – Publication per year

amount in 2024 is lower because we rerunned the SMS in June 2024.

4.4.2 Types of Architectural Technical Debt - RQ1

Martini and Bosch (MARTINI; BOSCH, 2016a) suggested a classification based on a set of causes and effects of ATD to aid the generation of indicators about interest and principal regarding ATD items. However, the paper does not show details about how to classify types of ATD items. Besker et al. (BESKER *et al.*, 2018) proposed a unified model classification for ATD management describing a model to aid the ATD phenomenon. The paper suggested an ATD Identification as follows: Architectural Dependence, Standards and "Policy" Deformity, Lack of Test and Quality, Subsystem Integration Management Deformity, and QA Synergy Conflict. Li et al. (LI *et al.*, 2015a) proposed a more generic classification and embraced a large number of ATD studies. This classification is defined based on the following characteristics: Architecture Smells, Architecture anti-patterns, Complex architectural behavioral dependencies, Violations of good architectural practices (Architecture Violation), Architectural compliance issues, and System-level structure quality issues. Due to the scope of Li et al. (LI *et al.*, 2015a) classification, we choose to use this classification to organize ATD types. Despite some similarities in these characteristics, we observed that some less-cited works need to fit these characteristics, as is the case of Social Debt, Model-driven ATD, and Self Admitted Debt. With that, to facilitate the analysis, we created one category for each of these less cited.

Forty-nine papers make a study that allows defining a type of ATD. However, the other 36 papers refer only to tools, techniques, or general descriptions. To group the identified papers, we organized them into eight categories: System-level structure quality issues (14), Architecture Smells (11), Architecture Compliance Issues (9), Complex Architectural Behavioral dependencies (6), Violation of good architectural practice (3), Social Debt (3), Model-driven Debt (2) and Self-admitted debt (1). The distribution of these papers is detailed in the correspondent replication kit.[3].

4.4.3 Measurement of ATD - RQ2

The measure of ATD is defined as how is the measuring of the amount of identified ATD items in a software system (BROWN *et al.*, 2010). We analyzed the papers about measuring ATD items in two ways: the first is regarding if the paper measures or does not measure the ATD item, and the second is related to how the measure is detailed in each paper. We found that 49 studies measure the ATD items, showing the importance of identifying and measuring the ATD. However, 36 studies did no measure ATD items. We used a keywording process in the SMS to get recurrent terms that measure the ATD item. We observed that there are three more critical kinds of measurement: the first is the amount of architecture smells (12), the second is modularity metrics (7), followed by the amount of software architecture rules violated (6), and the complexity measures of files (5). The other methods have less than four studies, each one: architectural root and hotspots. The distribution of these papers is detailed in the correspondent replication kit.[4].

4.4.4 Monitoring of ATD - RQ3

We analyzed the papers about the monitoring of ATD items in two ways: the first is whether the paper monitors or does not monitor the ATD item. The second is related to how the monitoring is performed in each paper. The monitoring of ATD is identified if there are at least

[3] https://github.com/Technical-Debt-Large-Scale/smsatd2/blob/main/md/sprq1.md
[4] https://github.com/Technical-Debt-Large-Scale/smsatd2/blob/main/md/sprq2.md

the following activities: identify, measure, and check if the ATD item persists or does not persist in the next analyzed version (BROWN *et al.*, 2010).

We observed in the SMS that 28 studies monitoring the ATD items showed the importance of tracking the ATD item. However, 57 studies do not monitor ATD items. So, not all studies show the importance of checking whether the ATD items increase or decrease during the software evolution. About the studies that monitor ATD, we can see that the most significant kind of monitoring is analysis for each release, corresponding to 24 studies, followed by the version analyzed (4). The distribution of these papers is detailed in the correspondent replication kit.[5].

We can see that some studies follow the activities proposed by Li et al. (LI *et al.*, 2015a) on the ATD monitoring process. However, there is a lack of studies on a more general process of monitoring ATD effectively. This is also demonstrated by the lack of specific tools and methods to identify and analyze ATD. Most tools and methods borrow identification from analysis based on code metrics, component dependency analysis, and analogies to identify and analyze ATD.

4.4.5 Tools and Methods to Identify and Analyze ATD - RQ4

We found no generic tool enough to cover all kinds of ATD. Also, there are some tools that highlight to identify or help to identify and analyze the ATD items, like Sonarqube (8), Understand (4), Arcan (4), Titan (3), cast (3), Sonargraph (3), DV8 (2), and Fusion (2). The rest of the tools are reached only one time per each selected paper. Finally, nine studies implement a custom tool to identify and analyze the ATD items. The distribution of these papers is detailed in the correspondent replication kit[6] and detailed tools[7].

We performed a keywording process during the reading and analysis of the selected papers to get recurrent terms used in the papers to identify or analyze architectural debts. The ATD method identification process is not trivial, as no generic method still meets the identification or analysis of ATD. Figure 7 shows the most common methods to identify or analyze ATD items. You can find more details about each method and papers distribution in replication kit[8].

We found no generic method covering all kinds of ATD analysis. However, two

[5] https://github.com/Technical-Debt-Large-Scale/smsatd2/blob/main/md/sprq3.md
[6] https://github.com/Technical-Debt-Large-Scale/smsatd2/blob/main/md/toolsandotherdistribution.md
[7] https://github.com/Technical-Debt-Large-Scale/smsatd2/blob/main/md/toolsdetailed.md
[8] https://github.com/Technical-Debt-Large-Scale/smsatd2/blob/main/md/sprq4.md

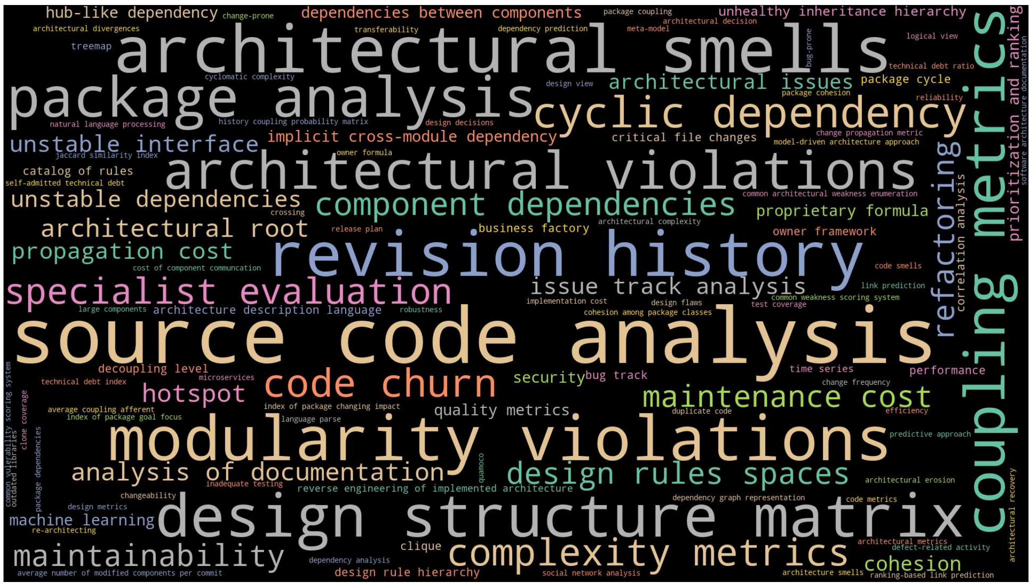


Figure 7 – Methods most commons to identify and analyze ATD items

methods are highlighted: the first is source code analysis, and the second is architectural smells to analyze the ATD items. Other methods are used, like design structure matrix, modularity violations, code churn, coupling and cohesion metrics, complexity metrics, and analysis of architectural documentation. We also observed a mix of the selected papers' methods to identify and analyze the ATD items.

4.4.6 *Calculate the Cost of the ATD item - RQ5*

To calculate the cost of an ATD item is necessary to check the effort to pay the ATD item due to financial and technical impact to reduce or mitigate the ATD impact on the system over time(LI *et al.*, 2014). There are two kinds of costs: the cost of principal regarding the effort to pay all ATD items and the cost of interest to maintain the ATD items (CURTIS *et al.*, 2012), (TOM *et al.*, 2013). We analyzed the cost of ATD items in two ways: the first is regarding if the paper calculates the cost or does not calculate the cost of the ATD item, and the second is related to how to calculate the cost in detail for each paper. We observed that 25 papers calculated ATD anyway, and 60 papers did not calculate the cost of fixing ATD.

We can observe that many papers (25) show that the top way to calculate the ATD item's cost is by using a custom formula. So, the main effort is calculated based on days (10) or based on hours (6) in terms of the main effort to fix the ATD item. Another two studies calculate the effort using DWM (Developer Work Months). Another two papers show a numeric scale to calculate the effort to fix the ATD items. Finally, five studies show effort per release to fix the

ATD items. Besides, all these studies showed an expert evaluation to analyze and fill the formula to calculate the cost of fixing the ATD item. You can find the distribution of these papers in the replication kit[9].

Finally, we created a summary of Identification and Monitoring for ATD items described in Figure 8.

4.5 Discussion

4.5.1 Types of Architectural Technical Debt - RQ1

As we can see, the three main types of ATD are *System-level structure quality issues*, *Architecture Smells*, and *Architectural compliance issues*. These indicate that quality issues, code source organization, and compliance issues are more used characteristics to classify ATD.

We found that some selected papers did not fit to Li et al. (LI *et al.*, 2015a) classification. Therefore, we proposed three new categories to fit these papers as follows: i) *Architectural Model-driven Debt* - the authors propose a set of documents that model the archiving of system components based on a set of techniques that evaluate existing documents (for example, NLP (Natural Language Process), Machine Learning, and Model Checking) to describe the system architecture. Example: papers SP1(PEREZ *et al.*, 2019) and SP10(VERDECCHIA, 2018). ii) *Architectural Social Debt* - the authors evaluate social behaviors derived from anti-patterns, individual or group behaviors that generate ATD. Michael Golden (GOLDEN, 2010) observed cultural implications that can minimize or maximize the unintentional TD caused by human factors. Example: papers SP12(MARTINI; BOSCH, 2017), SP45(ZALEWSKI, 2017), and SP46(TAMBURRI, 2019). iii) *Architectural Self Admitted Debt* - records are made, usually via comments in source code that identify ATD. Potdar et al. (POTDAR; SHIHAB, 2014) found Self-admitted TD as one of the most TD identification approaches used in repository mining techniques where developers comment directly in code or message commits about sub-option solutions in software architecture. Example: paper SP43(SIERRA *et al.*, 2019b).

Papers SP13 (FONTANA *et al.*, 2016b), SP23 (CAI; KAZMAN, 2019), and SP26 (FONTANA *et al.*, 2016a) do not fit into types of ATD because they report assessment tools to assist the process of identifying and assessing ATDs. The SP47 paper was removed from this classification because it is a concept paper on ATD, showing the causes and consequences

[9] https://github.com/Technical-Debt-Large-Scale/smsatd2/blob/main/md/sprq5.md

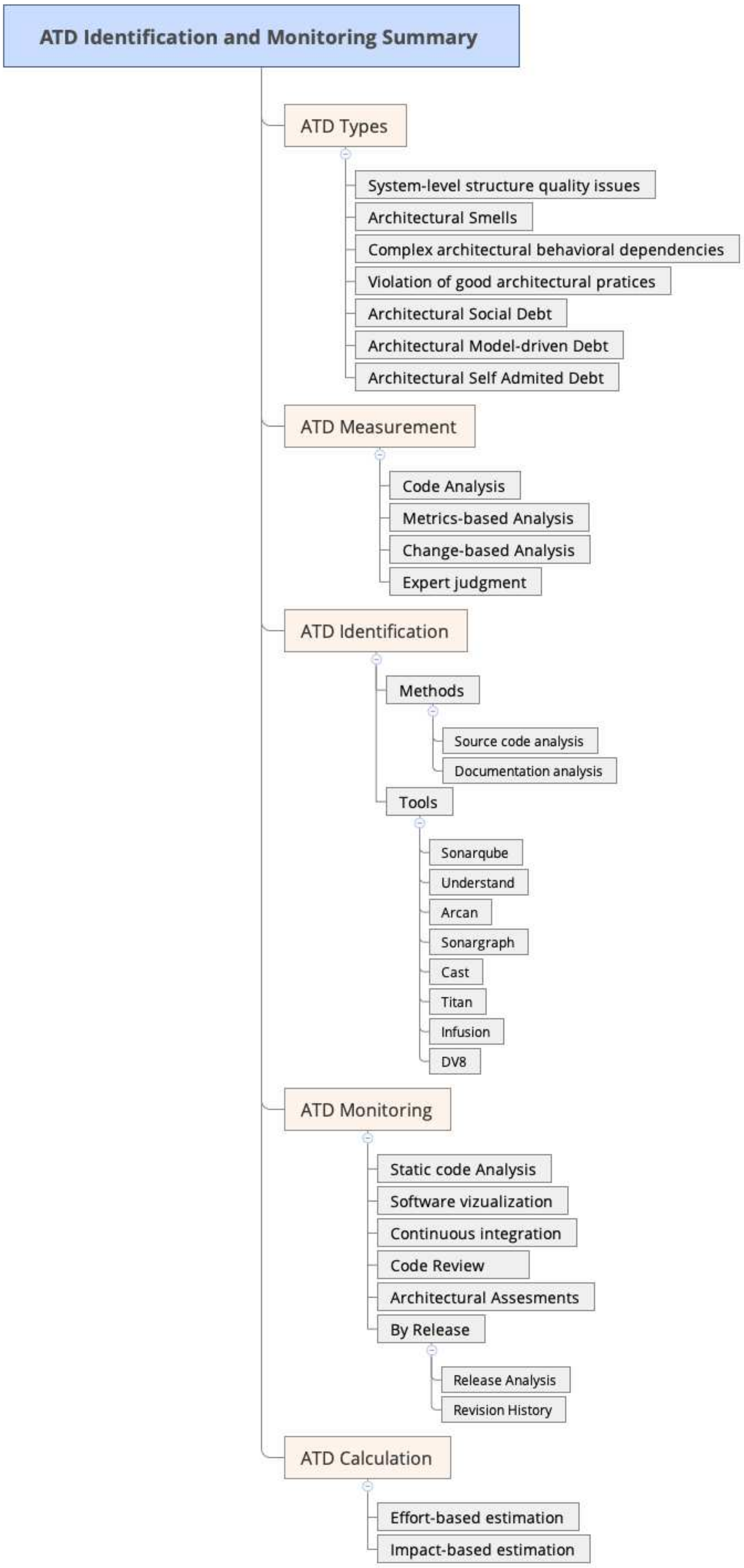


Figure 8 – ATD Identification and Monitoring Summary

of ATD. Papers SP32 (BESKER *et al.*, 2017b), SP34 (MARTINI *et al.*, 2016) present a report and address conceptual aspects and perceptions of ATD in large software companies. Finally, paper SP36 (MARTINI *et al.*, 2015) proposes its classification based on causes that generate the accumulation of ATD.

We found that researchers and practitioners tend to use the term "Architectural Debt" without considering a clear and precise meaning to it. It is difficult because the architectural level depends on the phase and the actors that identify the ATD during the software development cycle (ELIASSON *et al.*, 2015).

4.5.2 Measurement of ATD - RQ2

As we can see, the top ways of measures of ATD are *architecture smell*, *software architecture rules violated*, *complexity metrics*, and *modularity metrics*. These indicate that architectural features related to code and compliance checking are more used characteristics to measure ATD (SHARMA *et al.*, 2020), (KNODEL *et al.*, 2006).

We found that some selected papers use a mix of methods to define a unit of ATD. For example, in SP17(MARTINI *et al.*, 2018a), the authors used architectural smells to identify and prioritize ATDs. In SP19(KAZMAN *et al.*, 2015a), the authors used source code analysis, issue tracker, file dependencies, and Design Structure Matrix to define the architecture root. In SP2(MARTINI *et al.*, 2018b), the authors used source code analysis to calculate cohesion, coupling, and complexity metrics to identify modularity among components to define a unit of complexity measures of files. In the SP7(LI *et al.*, 2014), the authors consider modularity metrics to check architectural compliance issues, and in the SP14(LI *et al.*, 2015), the authors used architectural decisions during project design based on change scenarios.

Li et al. (LI *et al.*, 2015a) show measurement of Technical Debt in terms of quantifying the benefit and cost of known TD in a software system through estimation techniques or estimates of the overall TD level in a system. However, our work proposed to measure ATD items using units of items identified to facilitate the organization of the ATD item in a list of ATD or ATD backlog issues to control the amount of ATD items classified by chosen unit. Further details about ATD cost estimation are presented in the RQ5, which covers calculating the effort to fix the ATD Items in terms of principal.

There are some implications for researchers and practitioners. For researchers, measuring ATD items should be further investigated because there is no consensus on defining a unit of ATD or how to calculate the ATD items. For practitioners, only a few cases (SP6(FENG *et al.*, 2019), SP19(KAZMAN *et al.*, 2015a), and SP40(CURTIS *et al.*, 2012)) present details on how to measure the ATD item in the industry. Hence more industrial cases are needed to measure and classify different types of ATD in practice.

4.5.3 Monitoring of ATD - RQ3

As we can see, the top ways of ATD monitoring are *release analysis*, *release plan*, and *version analysis*. These indicate that way of monitoring ATD regarding changes in project documentation and source code changes over time. However, there is a lack of software architecture documentation, and, commonly, this documentation is outdated. Hence, causing a challenge to ATD monitoring.

We found that some selected papers use a mix of artifacts and methods to monitor ATD. For example, in SP1(PEREZ *et al.*, 2019), the authors work on a release plan using several different artifacts on the architectural level without considering source code. In SP2(MARTINI *et al.*, 2018b), the authors analyzed each release version calculating the modularity among the files changed in the release. In SP6(FENG *et al.*, 2019), the authors monitored the evolution of "hostspot" during a study comparing the revision history and issue tracker to detect problematic files.

Li et al. (LI *et al.*, 2015a) observed five classifications of TD monitoring: Threshold-based approach (regarding monitoring quality metrics), TD propagation tracking (the impact of TD in other parts of the system), Planned check (systematic way to identify and track TD), based on a quality attribute (monitoring presence of quality attributes), and TD Plot (monitoring a set of aggregation of TD measures using a dashboard). Besker et al. (BESKER *et al.*, 2018) in their SLR, observed the primary goal of a systematic monitoring process of ATD is to capture and track the presence of ATD items in a system and provide early evaluation of impacts in the system to detect costs and risks in architectural degradation.

We can see that some studies follow the activities proposed by Li et al. (LI *et al.*, 2015a) on the ATD monitoring process. However, there is a lack of studies on a more general process of monitoring ATD effectively. This is also demonstrated by the lack of specific tools and methods to identify and analyze ATD. Most tools and methods borrow identification from analysis based on code metrics, component dependency analysis, and analogies to identify and analyze ATD.

4.5.4 Tools and Methods to Identify and Analyze ATD - RQ4

As we can see, the main tools cited by selected papers are SonarQube, Arcan, Understand, Sonargraph, Cast, and Titan. We observed that the aspects of the architectural debt could not be detected by tools that use only source code analysis. It is hard to automatically extract information about project design documentation or the model proposed at the start of the project. Another part of the system depends on technical aspects like the database, operational systems, APIs, or outdated technology. Even some conventions created in the initial of the project were broken during the implementation of components (KRUCHTEN *et al.*, 2019).

As we can see, the primary methods cited by selected papers are source code analysis, architectural smells, modularity violations, and architectural violations. Source code analysis is the most used method found in our SMS. However, it is necessary to use other methods to compose a way to extract information about ATD items or analysis of ATD. For example, architectural smells are needed to use together with source code analysis to identify cycle packages, hub-like dependencies, or unstable dependencies. Architectural Smell is the second method most used, and modularity metrics are the third more used method. However, is necessary to develop more precise methods to cover other parts of software architecture analysis to extract information about software architecture documentation, and models proposed at the start of the project, and extract information about other technical aspects. It is also necessary to consider methods that cover business changes and business decisions that impact architectural decision-making that can intentionally create architectural debts.

4.5.5 Calculate the Cost of ATD item - RQ5

As we can see, the top ways of the calculus of cost to fix ATD are the use of *formulas to calculate effort based on time, for example, man-hours* (hours, days, or months), and the use of *formulas to calculate effort in a release*. These indicate that specific formulas and expert evaluation are the main methods to do it.

We found that some selected papers use effort based on estimated money to fix the ATD. For example, in SP40(CURTIS *et al.*, 2012), the authors used a repository of projects that register the software's complexity and register a set of architectural rules based on specific languages and technologies to calculate the effort to fix the ATD based on an aggregated formula that sums the effort involved in technical debts. The final results are defined in terms of hours and multiple by U$75 dollars per hour to find the total effort to pay the technical debt items. In SP8(MARTINI; BOSCH, 2016a), the authors used a particular set of formulas created by the AnaConDebt framework related to principal and interest to calculate the effort to fix the ATD Items identified.

Besker et al. (BESKER *et al.*, 2018) in their SLR, observed that the main aspects of the calculation of effort to fix ATD are integrating the resource involved to refactor the ATD items in terms of time and evaluating the cost-benefits to repay the ATD items. (BROOKS, 1974) observed that if there is a complex problem to solve for complex software systems, not always adding more workforce solves the problem. However, it can cause the opposite effect; that is, it can delay the problem's resolution even more. (KRUCHTEN *et al.*, 2019) observed that it is necessary to calculate the cost of doing whatever you need to do with technical debts at some point in a software product's life. This involves computing or estimating the cost to carry and eliminate the technical debt items.

We also observed that there is no consensus on how to estimate the effort to pay the ATD. Since in many cases, the calculations of effort estimation are done in an ad-hoc way or even by creating specific formulas based on the organization context. Also, using specialists' analysis to fill out such formulas and/or make estimates based on the experience of those involved in the software project or product. As a result of the selected studies' analysis, we can see that the aspects related to ATD are essential for the software industry. The non-payment of ATD items can cause many negative structural impacts on the software development life cycle.

The findings of RQ5 have significant implications for both researchers and practitioners. For researchers, the insights provided regarding the cost implications and effort required to address ATD shed light on the practical aspects of managing architectural debt. The study explores the challenges associated with estimating the necessary resources to mitigate ATD, considering factors such as time, expertise, and the potential impact on business goals and software quality attributes. This understanding of cost analysis and effort calculation empowers practitioners to assess trade-offs, prioritize debt resolution activities, allocate resources effectively, and make informed decisions about ATD management.

4.6 ATD Roadmap

Based on the research questions and results found in this study, we have proposed a classification and a roadmap for Architectural Technical Debt. This framework can be useful for researchers and practitioners looking to organize and implement a process to identify and monitor ATD.

4.6.1 Classification of ATD

The classification we have proposed, based on the types of ATD, measurement of ATD, monitoring of ATD, methods to identify and analyze ATD, and calculation of ATD items, can serve as a valuable framework for organizing activities related to the identification and monitoring of ATD in a software development process. This classification provides a structured approach to understanding and addressing various aspects of ATD, enabling practitioners to effectively manage and mitigate the impact of ATD on software projects.

4.6.1.1 Types of ATD

– *System-level structure quality issues*: These are issues related to the overall structure and organization of the system, such as high coupling, low cohesion, and poor modularity. They can result in difficulties in understanding, testing, and maintaining the system.
– *Architectural smells*: These are indicators of potential design problems that can lead to ATD.
– *Complex architectural behavior dependencies*: These are issues related to the interactions between components and modules in the system, such as cyclic dependencies, unexpected dependencies, and tight coupling. They can result in reduced flexibility and increased complexity.
– *Violation of good architectural practices*: These are issues related to the adherence to established architectural principles and guidelines, such as separation of concerns, layering, and encapsulation. Violations of these practices can lead to decreased system quality and maintainability.
– *Architectural social debt*: These are issues related to the communication and collaboration between stakeholders involved in the architecture, such as misalignment of goals and priorities, lack of documentation, and poor communication.
– *Architectural model-driven debt*: These are issues related to the use of architectural models and notations, such as inconsistencies between models and code, incorrect or incomplete models, and lack of traceability between models and code.
– *Architectural self-admitted debt*: These are issues that are explicitly acknowledged by

developers or architects as ATD, either through comments in the code or documentation.

4.6.1.2 Measurement of ATD

– *Code analysis*: analyzing the source code of the system to identify architectural smells, architectural violations, and other indicators of ATD.
– *Metrics-based analysis*: using software metrics such as coupling, cohesion, and complexity to identify potential ATD.
– *Change-based analysis*: analyzing changes to the system over time to identify areas that are more prone to ATD accumulation.
– *Expert judgment*: using expert opinions from architects and developers to identify potential ATD and their impact on the system.

4.6.1.3 Monitoring of ATD

– *Static code analysis tools*: using tools to measure code quality metrics, such as architectural smells, complexity, and coupling, and track these metrics over time related to software structure.
– *Software visualization tools*: using tools to provide insights into the architecture and identify potential ATD hotspots.
– *Continuous integration and delivery practices*: integrating automated tests and quality checks into the development pipeline to detect and fix ATD early in the development process.
– *Code reviews and architectural assessments*: reviewing the architecture and design documents, tracking and analyzing defect reports, user feedback, and performance metrics to identify potential ATD issues.

4.6.1.4 Methods to identify and analyze ATD

– *Code analysis*: analyzing the source code of the system to identify architectural smells, design smells, architectural violations, and other indicators of ATD.
– *Architecture and design review*: review the system's architecture and design documents to identify issues such as coupling, cohesion, and modularity.
– *Metrics-based analysis*: using software metrics such as coupling, cohesion, and complexity

to identify potential ATD.

- *Change-based analysis*: analyzing changes to the system over time to identify areas that are more prone to ATD accumulation.
- *Expert judgment*: using expert opinions from architects and developers to identify potential ATD and their impact on the system.

4.6.1.5 Calculation of ATD items

- *Effort-based estimation*: estimating the effort required to resolve ATD items, which can be broken down into different components, such as design, coding, testing, deployment, and maintenance.
- *Impact-based estimation*: estimating the impact of the ATD on system quality attributes, such as performance, scalability, and maintainability, and translating them into monetary values by estimating the impact on business goals, such as lost revenue, increased maintenance costs, and decreased customer satisfaction.

4.6.2 Roadmap of ATD

Based on the topics discussed aforementioned, we proposed a possible roadmap for dealing with Architectural Technical Debt:

1. Identify and classify different types of Architectural Technical Debt that exist in your software project related to architectural issues.

2. Measure the amount of Architectural Technical Debt in your software project by using metrics such as code complexity, design smells, and architectural smells.

3. Set up a monitoring system to continuously track the accumulation of Architectural Technical Debt over time and to identify new debt items as they are introduced. It is important the visibility of the technical debt in the project documenting and recording the Technical Debt via issue tracker for example.

4. Use a variety of tools and methods to identify and analyze Architectural Technical Debt, such as code reviews, static analysis, and architectural visualization.

5. Prioritize the Architectural Technical Debt items based on their severity and impact on the software project's functionality and maintainability.

6. Estimate the cost of each Architectural Technical Debt item in terms of the effort required to address it, the risk it poses to the project, and the benefits of fixing it.

7. Create a plan for addressing the most critical Architectural Technical Debt items first, balancing the cost and benefits of each item against the available resources. It is important to define a cycle like a revision plan for each release plan.

8. Continuously monitor and measure the progress of the ATD management plan and adjust it as necessary to ensure that the project is moving towards a more maintainable and scalable architecture.

By following this roadmap, practitioners can identify, manage, and mitigate ATD in their software projects, leading to improved software quality and long-term sustainability.

4.7 Validity Threats

The validity threats are discussed using the categories construct validity, internal, and reliability validity described by Runeson and Höst (RUNESON *et al.*, 2012). Regarding **Construct validity** in our SMS, it is connected to the potentially subjective analysis of the selected studies. According to Kitchenham and Charters (KITCHENHAM; CHARTERS, 2007), the data extraction should be performed independently by two or more researchers, and in case of inconsistencies, a third author was involved in the discussion to clear up any disagreement. Related to **internal validation** about the interpretation bias of researchers, one researcher determined the articles, and two others reviewed them in two stages to generate the final set. Finally, regarding **reliability**, we use a replication kit available in the repository[10] that contains the dataset, SMS protocol, scripts regarding data analysis, and scripts regarding generating results in terms of figures and tables.

4.8 Conclusions and Future Work

This research reported a systematic mapping study about ATD identification and monitoring from 85 studies selected from 2012 to 2024. We observed that the concern about the ATD subject is increasing during this time span.

The overall conclusion is that: (i) the three main types of ATD are *System-level structure quality issues*, *Architecture Smells*, and *Architectural compliance issues*; (ii) the top ways of ATD measures are *architecture smell*, *software architecture rules violated*, *complexity metrics*, and *modularity metrics*; (iii) the top ways of ATD monitoring are *release analysis*,

[10] https://github.com/Technical-Debt-Large-Scale/smsatd2

release plan, and *version analysis*; (iv) regarding tools, the most cited are *SonarQube*, *Arcan*, *Understand*, and there are many tools developed by owners. Besides, the primary methods cited are *source code analysis*, *architectural smells*, *modularity violations*, and *architectural violations*; (v) the top ways of the calculus of cost to fix ATD are specific *formulas to calculate effort based on time* (hour, days, or months) associated with specialist evaluation and *customized formulas to calculate effort in a release*.

Our investigation has some implications for both *researchers* and *practitioners*: (i) the perception of ATD depends on the architectural level, the phase, and the actors involved in the software development cycle; (ii) about measuring ATD items, there is no consensus on defining a unit of ATD or calculating the ATD items because it depends on the studied context, and there are few cases that present details on how to measure the ATD item in the industry; (iii) about ATD monitoring, there is no consensus on defining a cycle of monitoring of ATD, and generally, this process is based on a manual process performed by specialists. For practitioners, only a few cases present details on how to monitor the ATD item in the industry, and there is a lack of tools to aid, automate, and visualize this process; (iv) more studies are necessary about tools and methods to aid ATD, covering specific aspects of software architecture and architectural debt; only a few cases present details on using the available tools to identify and analyze the ATD item in the industry; (v) about how to calculate the effort to fix ATD items, there is no consensus on defining a general method to calculate the principal or interest of ATD Items. For practitioners, only a few cases present details on how to calculate the ATD item in the industry, and there is a lack of tools to aid and automate this process.

We identified that there is no consensus regarding the detection of architectural technical debt because the software architecture permeates several phases of the software development cycle, for example, aspects of business direction, non-functional requirements neglected, lack of standardization and updating of the architectural project during the software's maintenance and evolution, due to the lack of standardization in the management of the software system's architectural complexity. Hence, we can see that it is necessary to do more studies on these aspects.

We proposed a roadmap to assist in the identification and monitoring of ATD items to ensure that software projects move towards a more maintainable and scalable architecture, thereby preventing the accumulation of ATD throughout the software development life cycle.

Finally, we plan to investigate the gaps identified in this mapping study. Another research approach will be to conduct an overall evaluation of the researchers and software engineering practitioners' proposed classification of the ATD identification process and the calculation process to fix ATD.

Artefact Availability

We use a replication kit available in the repository[11] that contains the dataset, SMS protocol, scripts regarding data analysis, and scripts regarding generating results in terms of figures and tables.

[11] https://github.com/Technical-Debt-Large-Scale/smsatd2

5 TECHNICAL DEBT IN LARGE-SCALE DISTRIBUTED PROJECTS: AN INDUSTRIAL CASE STUDY

This chapter presents a study focused on understanding the relationship between technical debt accumulation and various factors in large-scale distributed projects. We conducted a case study at Ericsson, a European Telecom Company, to identify this relationship. The study used data from 33 projects and conducted regression analysis and interviews with senior developers to interpret the results. The study found that task complexity has a strong relationship with technical debt accumulation, while global distance was mentioned by the interviewees as a relevant factor. Based on these findings, practitioners should consider avoiding complex or big tasks and breaking down big tasks into smaller ones if possible.

5.1 Introduction

Organizations around the world develop software in a globally distributed way (Global Software Engineering – GSE) to achieve benefits such as reduced time-to-market and access to skilled people all over the world (HERBSLEB; MOITRA, 2001; CONCHúIR *et al.*, 2009; RAMASUBBU *et al.*, 2011). However, geographical, temporal, and cultural distances amplify the difficulties associated with coordination and communication in GSE projects (HERBSLEB; MOITRA, 2001).

It is often the case that GSE projects involve a large number of people (large-scale projects[1]). The combination of scale and global distribution may lead to problems, such as more software defects (ESPINOSA *et al.*, 2007), schedule and budget overruns (HERBSLEB; MOCKUS, 2003), and make it challenging to manage Technical Debt (TD) (BAVANI, 2012).

TD reflects technical compromises to achieve short-term benefit at the cost of hurting a software product's long-term health, which puts future development and maintenance at high potential risk (CUNNINGHAM, 1992). TD refers to any incomplete, immature, or inadequate artifact in the software development life cycle affecting subsequent development and maintenance activities, which is treated as a type of debt that the developers owe the system (SEAMAN; GUO, 2011).

To keep TD accumulation under control, Technical Debt Management (TDM) is required throughout the development process. Part of TDM includes activities preventing

[1] Dikert *et al.* (2016b) define as large-scale software projects that involve at least 50 human resources – not necessarily only developers, but also other staff collaborating in software development – or at least six teams.

potential TD from being incurred. Meanwhile, TDM also includes activities dealing with the accumulated TD to make it visible and controllable and balance the software project's cost and value. TDM in large-scale GSE projects can be more complex. For example, it may be more challenging to ensure a common understanding of TD and TDM across multiple sites (BAVANI, 2012). Moreover, factors such as distance (CARMEL; AGARWAL, 2001; TAMBURRI *et al.*, 2013) are known to be associated with TD accumulation. Also, TD-related decisions are often not systematically used. There are no generic approaches used in the industry that facilitate systematic TDM (e.g., TD decisions are often not even explicitly captured) (HOLVITIE *et al.*, 2014).

To the best of our knowledge, there is no investigation on the TD accumulation in large-scale globally distributed software projects. Given the relevance of the topic for both research and industry, we have made an attempt to fill the existing gap through conducting an industrial case study in Ericsson, a company that develop hardware and software telecommunication solutions.

In this chapter, we report the findings of our investigation, which address the following research question: *What factors are related to the TD accumulation in large-scale GSE projects*?

The remainder of this chapter is organized as follows: The second section describes the background and related work. The third section presents the research design. The fourth section presents the results and discussions. Validity threats are discussed in the fifth section. Finally, we provide our conclusions and view on future work in last section.

5.2 Background and Related Work

In the GSE-related literature, the following topics stand out: the global distance (GD) between the teams, the form of communication between the project participants, and the developers' level of maturity.

GD measures the overhead of cooperation and coordination in communication between several sites. Effective communication between distributed sites is crucial for a successful distributed project (YAO *et al.*, 2010). However, distance negatively affects communication, which in turn reduces coordination effectiveness (CASEY; RICHARDSON, 2006). Thus, increasing the risk of incurring TD. Kazman et al. (KAZMAN *et al.*, 2015b) used a model approach to analyze the software architecture as a set of design rules spaces. Heikkilä et al. (HEIKKILA *et al.*, 2017) explored how hard communication is for the practitioners in large-scale globally distributed software projects.

Team Maturity (TM) implies an increased capability of controlling and managing TD, with important historical data available for development teams to quantitatively manage and control key projects as well as organizational processes (FALESSI *et al.*, 2013). (HARTER *et al.*, 2000) shows that high maturity can reduce cycle time and development effort suggests a lower TD.

Another aspect that makes software development more difficult is task complexity (TC). Alzaghoul et al. (ALZAGHOUL; BAHSOON, 2014) found that higher complexity may indicate higher rework costs. As a result, increased complexity might lead to an increase in TD.

When a developer identifies a debt, documenting the debt helps to manage TD systematically. Formal documentation can make the TD traceable and increase the effectiveness of TDM (DAS *et al.*, 2007). Guo *et al.* (2016) have proposed an approach to TDM based on systematic monitoring for each incremental release of a software product.

TD monitoring and TD repayment are two of the most important TDM activities, which helps managers to see the changes in the cost and benefit of unresolved TD over time (LI *et al.*, 2015b). Seaman e Guo (2011) suggest that through monitoring, development teams could find a proper guide using a TD list as the center of monitoring the status of TD. TD repayment has a strong relationship with TD measurement and monitoring because to repay the debt, the team should check how urgent the debt is and decide when to repay the debt, as mentioned. The perception of TD through monitoring and repayment was studied by Besker *et al.* (2017c) that explored the perception of TD in the software development cycle. They did a survey to estimate the time lost caused by the Technical Debt accumulation during the software life cycle.

Digkas *et al.* (2018a) conducted a case study on 57 open-source Java projects from the Apache ecosystem to investigate how developers fix issues and payback TD over time. They found that a small portion of the issue types is responsible for the largest amount of TD repayment.

To gain a better understanding of TD in industrial setups, Rios *et al.* (2018) conducted a tertiary study to look into the state of practice in several companies to understand the cost of managing TD, how maturity is managed in TD, what tools are used to track TD, and how a TD tracking process is deployed in practice.

Despite covering several TD issues, the reviewed papers do not focus on studying the TD accumulation in large-scale GSE projects. This paper fills the gaps mentioned above by employing an exploratory case study in a real and large-scale GSE project.

5.3 Research Design

To address our research question, we have conducted an exploratory longitudinal case study (RUNESON *et al.*, 2012).

5.3.1 The Case and Unit of Analysis

The **case and unit of analysis** is a telecommunication software product developed by Ericsson. This software has been evolving for more than 24 years with several technological changes, such as the inclusion of additional programming languages (Java in addition to C++) and a change in development methodology from plan-driven to agile practices.

The product is developed in a geographically distributed fashion, and includes (or has included) sites located in the USA, Sweden, Italy, and India, as you can see in Figure 9. It involves cross-functional teams that have from 4 to 7 developers and use agile practices. Project managers use a mix of agile and plan-driven practices to manage and coordinate teams across sites. The teams are responsible for tasks such as product customization (PC), bug fixing, and product refactoring. PCs are carried as independent projects that may take from 1 to 6 months.

The data collected and used in our investigation comprises the period from January 2013 to August 2016. It includes only PC tasks because they have the most significant impact and value for the company's customers who use the product.

The TDM process detected in the case study can be viewed in Figure 10, and details about each activity are described in Table 4.

5.3.2 Variables

In this section, the following **variables** were used for analysis: Technical Debt (TD), Task Complexity (TC), Lead Time (LT), Global Distance (GD), Total Developers (DV), Task Scaling (TS), and Team Maturity (TM). We selected these variables due to their relevance in GSE contexts and also due to the possibility to measure them in the investigated case (BRITTO *et al.*, 2016a; BRITTO *et al.*, 2016b). Table 5 presents a description of the investigated variables.

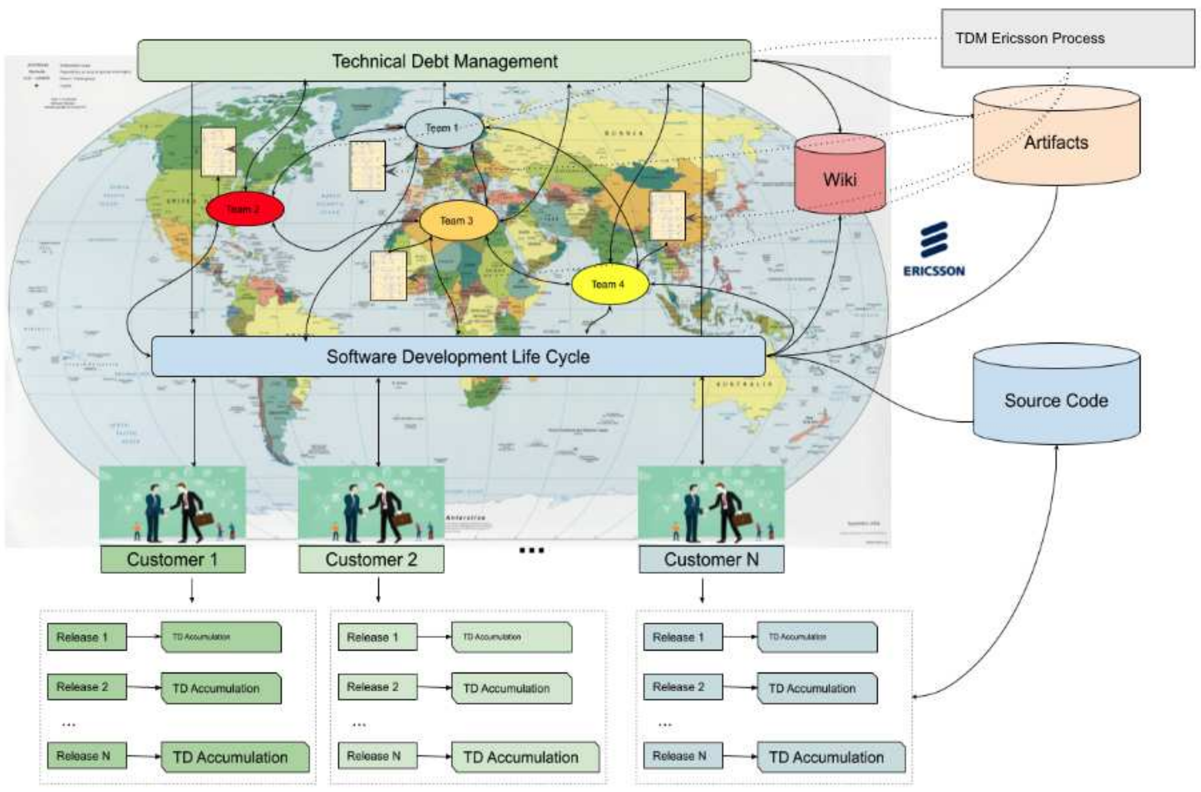


Figure 9 – Ericsson TD Management

5.3.3 Data Collection and Data Analysis

To collect the data associated with the investigated variables, we employed three **data collection** methods: archival research, semi-structured interviews, and repository mining.

We employed **archival research** to measure LT, TS, from the 33 investigated PCs. TD was measured through **repository mining**, while TC and TM were measured through interviews in a previous investigation conducted by the second author of this paper (BRITTO *et al.*, 2016b).

To analyze this data, we employed the **data analysis** method hierarchical multiple regression analysis, aiming at understanding the relationship between the selected factors and TD accumulation. More details can be viewed in the replication kit available in the data repository[2] of this study.

To support the interpretation of the regression analysis results, we conducted two **semi-structured interviews**: we first interviewed a Software Architect (SA1) in June 2017. In a second moment, we conducted a group semi-structured interview with two other software architects and another a semi-structured interview with 2 Software Architects (SA2 and SA3) in

2 https://github.com/Technical-Debt-Large-Scale/tdmls

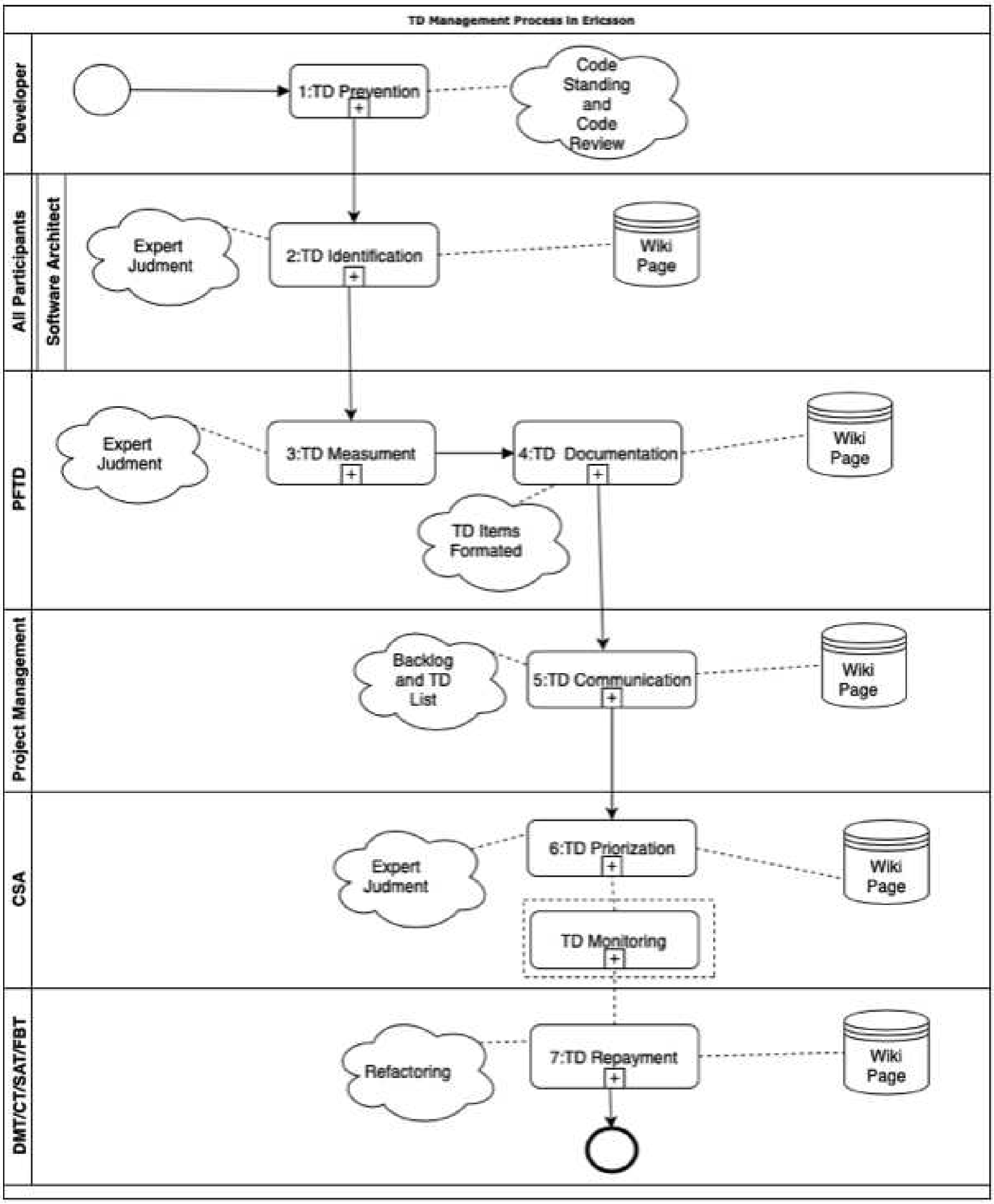


Figure 10 – Ericsson TDM Process

January 2018. Each meeting took approximately one hour. All interviewees had more than ten years of large-scale GSE experience. The questions can be viewed in the data repository of this study.

The semi-structured interview results were analyzed using content analysis since it is a systematic and rule-guided technique used for analyzing all sorts of textual data. It provides a brief and broad description of the phenomenon and allows researchers to enhance the understanding of raw data (MAYRING, 2014).

Table 4 – Ericsson TDM Activity

Seq	TDM Activity	Description	Benefits	Challenges
1	Prevention	The process starts when Developers and Software Architects defining code standards and code review to Prevent TD	Adopt of code standard and code review	-
2	Identification	The Software Architect check the code by Expert judgment and share the good practices in wiki page system shared by all participants of project	There is a tool that aid the process using SonarQube	-
3	Measurement	The assigned System Manager analysis more important fidings to measure and record in wiki page.	Facilitate the maintainability. Generally, it is measured in man-days and the cost is just a gues based on experience or intuition.	It is difficult to qualify the cost and benefits of fixing TD
4	Documentation	The PFTD (Person who identified the TD) format the TD items and record in wiki page session about the TD documentation	to garantee complete overview and track of TD process	-
5	Communication	The Project Manager creates a Backlog and TD list to share with all participants of project.	The product community (Software Architects, Program Managers and Line Managers) share the updates and information among the sites.	-
6	Prioritization	The CSA (Chief Architect), by expert judment scale the TD prioritization in spreadsheets and record in Wiki page of the project.	A TD list was generated after the discussion within the product cummunity. The prioritization if performed by Chief Architect	The conflit between TD and other tasks like new features.
7	Repayment	The prioritized issues are repayments by refactoring techquinque	TD was repaid through refactoring the codebae ty teams with free space. Code refactoring to minimize the problem	-

Table 5 – Study variables

ID	Name	Type	Data Collection	Description
TD	Technical Debt	Dependent	Repository Mining	Is the amount of dollars calculated by using SonarQube needed to fix all problems (duplication, violations, comments, coverage, complexity, bugs, bad design) in the code base.
TC	Task Complexity	Independent	Interviews	Is the parameter used to describe how complex the task. Each PC was estimated by a positive integer (complexity points) (BRITTO *et al.*, 2016b).
LT	Lead Time	Independent	Archival Research	Is the total time needed to deliver a task, counted by days.
TS	Task Scaling	Independent	Archival Research	Is the capacity of a task resizes according to the increase in demand for this task.
GD	Global Distance	Independent	Archival Research	Is the metric that measures the complexity of communication between sites, which represents the overhead of cooperation and coordination when more than one site is involved (AVRITZER *et al.*, 2015).
DV	Total Developers	Independent	Archival Research	Is the number of developers involved in the development of each task.
TM	Team Maturity	Independent	Interviews	Is the parameter to describe the level of how a team can deliver the product independently (BRITTO *et al.*, 2016b).

5.4 Results and Discussion

This section presents and discuss the results of the conducted regression analysis. We first present the results of checking the assumptions of the employed method, which is followed by the actual results of the analysis and discussion.

5.4.1 *Regression Analysis Assumptions*

We created a box-plot (Figure 11) to analyze the TD values among all involved sites. As a result we identified two outliers. Only one was removed since the other was deemed as relevant for the analysis.

Table 6 – Factors Correlated to TD

Factors	Impact	Spearman's ρ	p-value	Correlated
LT	Positive	0.486	5.00×10^{-3}	YES
TC	Positive	0.650	5.69×10^{-5}	YES
DV	Positive	0.505	3.00×10^{-3}	YES
TS	Negative	-0.439	1.20×10^{-2}	YES
TM	N/A	-0.135	4.62×10^{-1}	NO
GD	N/A	0.034	8.55×10^{-1}	NO

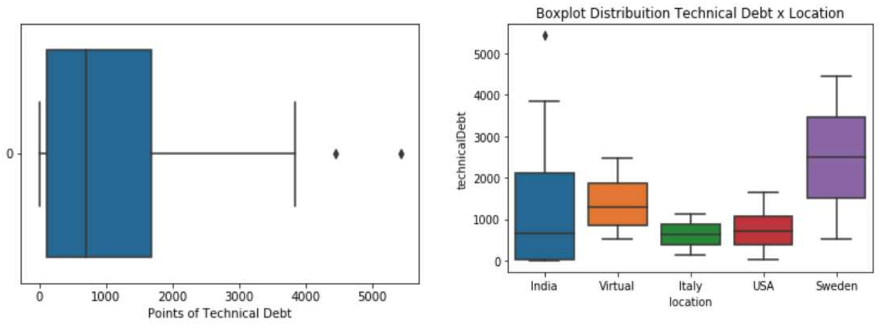


Figure 11 – Boxplot points of TD and distribution of TD x Location

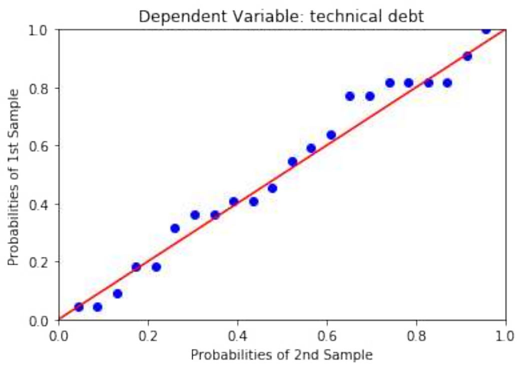


Figure 12 – PPplot

First, to check correlation among the selected features and TD (and identify linear relationships), we used Spearman's rank coefficient (Table 6). As a result, we identified that four features (LT, TC, DV and TS) correlated with TD (p-value < than 0.05).

Second, to further investigate the nature of the relationship between TD and the factors with significant correlation, we used partial regression plots (FOX, 2015). As a result, the plots confirmed that there is some level of linearity between TD and LT, TC, DV and TS.

Third, we tested for auto-correlation using the Durbin-Watson test. If the Durbin-Watson test's value is between 1.5 and 2.5, there is no linear auto-correlation in the data. The Durbin-Watson values in our tests are the following: lead time = 1.614, task complexity = 2.155, total developers=1.230, task scaling = 1.727, and technical debt=2.041, which were acceptable. So, the residuals are independent in our data.

Table 7 – Testing Multicollinearity - (VIF, Tolerance)

Model	LT	TC	DV	TS
model1	(1,1)	-	-	-
model2	(1.12, 0.89)	(1.12, 0.89)	-	-
model3	(1.60, 0.62)	(1.14, 0.88)	(1.57, 0.64)	-
model4	(1.76, 0.57)	(1.50, 0.67)	(1.82, 0.55)	(1.51, 0.66)

$$\begin{bmatrix} x1 \\ x2 \\ x3 \\ x4 \end{bmatrix} = \begin{bmatrix} LeadTime \\ TaskComplexity \\ TotalDevelopers \\ TaskScaling \end{bmatrix} = \begin{bmatrix} 311.52 \\ 3234.82 \\ 1241.58 \\ -1495.39 \end{bmatrix}$$

Figure 13 – Regression coefficients

Fourth, we tested the normality of the residuals. To do so, we used P-P plots. The points on the plot remain close to the diagonal line, which means residuals are normally distributed. So, we do not violate the assumption of normality. (Figure 12).

Fifth, we tested the assumption of homoscedasticity. To do so, we used the Breusch-Pagan test. The Lagrange multiplier statistic was 2.326 and p-value was 0.676, i.e., the assumption of homoscedasticity was met.

Finally, to verify the presence or absence of multicollinearity, we used Tolerance/VIF (Variance Inflation Factor). The tolerance of independent variables should be greater than 0.1 and VIF less than 10. Table 7 shows that the tolerance values in our study are all greater than 0.1 and the VIF values all less than 10.

As a result, the following regression model was presented where TD is the dependent variable and LT, TC , DV, and TS are the predictors. Equation (5.1) presents the resulting model used in our analysis:

$$TD = 1048.31 + 311.52 * LT + 3234.82 * TC + 1241.58 * DV - 1495.39 * TS \qquad (5.1)$$

The statistical regression analysis caused the GD and TM variables to be discarded from the regression models, as mentioned above.

Figure 13 represents the variables and their respective regression coefficients in regression model.

5.4.2 Results and Discussions of Factors related to TD

According to the interviewees, all four factors (TC, LT, DV, and TS), relate to TD accumulation. Although the interviewees mentioned that TM and GD are also somehow related to TD, we could not confirm this in the conducted regression analysis.

Architect SA1 said that task complexity (TC) has a strong relation to TD since complicated tasks tend to have more debt. However, it was hard for the architect to judge exactly what can be seen as a complicated task. The SA2 confirmed this:

SA2: *what is a complex task is hard to say, when a task contains a lot of lines of code, but from the functional perspective, it is very easy to build and will not create any debts at all, do we still think it has low complexity?*

Alzaghoul and Bahsoon (ALZAGHOUL; BAHSOON, 2014) found that increase in a software's complexity leads to an increase in TD. If the complexity increases due to changes in a software's structure, the dependencies between different parts of the software may become more complex as well, which may cause potential extra work to maintain the software.

We also learned that the longer it takes to complete a product customization developemnt cycle, the higher the TD. Besker et al. (BESKER *et al.*, 2019) identified that the shorter lead times (LT) can help to maintain costs under control, through using good planning between the moment of the product customer's order until the delivery can offer many advantages like cost reduction.

Regarding task scaling (TS), we observed that as the size of TS increases, the amount of TD tends to decrease. This looks counter-intuitive at first. For example, Guo et al.(GUO *et al.*, 2016) identified that for large systems developed in a collocated manner, it is easy to lose track of delayed tasks or to misunderstand their impact. In our case, which does not go in the same direction of Guo et al., we believe that the observed relationship may relate to the fact that tasks with high TS often involved senior developers to support newcomers, which might have lead to lower TD in those cases.

In the case of total developers (DV), the total number of developers in a software project is critical factor in GSE projects, due to the difficulty communicate when there is a large amount of people (CARMEL; AGARWAL, 2001), (HERBSLEB; MOITRA, 2001).

Regarding global distance (GD), although we did not identify a statistical significant relationship between GD and TD accumulation, software architect SA2 mentioned:

SA2: *The worst case is that people working with the same functionality are sitting in different places and doing different phases of the work.*

Architect SA2 also mentioned team maturity (TM), although it was not statistically significantly related to TD in our results. Although not significant, we identified that maturity tends to relate to TD accumulation (the higher the maturity, the higher the TD). After investigating our dataset, we identified that the largest and most complex tasks tend to be attributed to the mature teams in the investigated case. This means that the observed correlation between TM and TD is likely affected by the complexity and the size of the PCs.

5.5 Validity Threats

The validity threats associated with our investigation are discussed using the categories internal and external validity described by Runeson *et al.* (2012).

In relation to **internal validity**, one limitation is that we were able to investigate a subset of factors that potentially relate to TD accumulation. Other factors can still be studied, such as social, cultural, and other technical factors not evaluated in this study.

Regarding **external validity**, since we employed the case study method, our findings are strongly bound by our research context. In addition, the investigated case involved only one product in one company. To mitigate this threat, we described the context of our study in as much detail as possible so that the readers can identify if the context of our investigation is similar to theirs and reuse our findings whenever applicable.

5.6 Conclusions and Future Work

This chapter reports the results of a case study conducted in Ericsson that aimed at investigating the accumulation of TD in a large-scale globally distributed software project.

The overall conclusion is that TD accumulation strongly correlates with specifics factors of GSE projects. We believe that the process of TDM becomes more complex in globally distributed projects with different sites and different teams. Thus, a suitable TDM process must consider the GSE factors that correlate with TD accumulation, which we plan to investigate in a future study.

Our investigation has some implications for both researchers and practitioners. Regarding researchers, we believe that it is still necessary to conduct similar research in other companies to learn more about the accumulation of TD in large-scale globally distributed software projects.

We identified that task complexity is the factor most related to TD accumulation. Thus, practitioners should be aware of this and try to avoid complex projects and subdivide them into less complex projects as much as possible to prevent TD accumulation.

Finally, we plan to continue investigating other cases in this company to strengthen the empirical evidence reported herein.

6 ATDCODEANALYZER - THE PROPOSED APPROACH

In this chapter, we present the ATDCodeAnalyzer, an automatic method that can be applied to a repository under Git version control. By following the steps outlined, it is possible to identify and assess code artifacts affected by ATD. Section 6.1 presents the activities carried out to define the proposed method. This is followed by a detailed description of the method in Section 6.2, the objectives and research questions in Section 6.3, and the hypotheses defined to test the method in Section 6.4. Section 6.5 outlines the detailed steps of the proposed method. Finally, Section 6.6 describes the relationship between the proposed method and the SysRepoAnalysis tool (Chapter 7), as well as the study conducted in partnership with Ericsson to apply the proposed method in a real-world case (Chapter 8).

6.1 Introduction

In the systematic mapping presented in chapter 4, we observed that there is still no consensus regarding the identification of Architectural Technical Debt items, mainly when referring to the definition of a widely established method to identify source code artifacts from software repositories. In addition, there is an intrinsic challenge to the recent area of ATD that becomes more complex by permeating several stages and artifacts in the software development cycle. As we saw in chapter 5 (case study in Ericsson TDM process), this problem becomes more difficult and complex in large-scale systems.

Defining a method that generates indicators of architectural technical debt through the historical analysis of changes that have occurred in the software code over time, can be useful to identify critical areas of the code repository that help the Software Architect and developers to make decisions about the effort to maintain code artifacts that cause architectural problems.

Proposing a systematic method that enables the automatic identification of ATD items can benefit both researchers and practitioners, as there is still a lack of methods that are independent of specialist intervention or analysis. For industry practitioners, an automatic method can aid in the identification and monitoring of ATD items, streamlining the process of identifying, monitoring, and paying for ATDs using only historical analysis of source code under version control, without the need for manual analysis by specialists. In addition, updating architectural project documents typically requires manual analysis of other artifacts by software architects or developers, which can be time-consuming and difficult.

In order to design the first version of our method, we performed several tests in several GitHub Apache repositories so that we could collect data, organize them for analysis, and finally select relevant features to our hypothesis.

In the data collection stage, we focused on testing data extraction from code repositories hosted on GitHub to clone such repositories locally and extract information from commits and modified files in the analyzed versions. Once the data was identified, this data was worked on to ensure that it was ready for analysis so that we could extract metrics related to the number of commits, number of files, number of LOCs, the occurrence of files in commits, number of lines modified in commits and number of lines modified in files over time. We did an exploratory data analysis through a thorough examination to understand which features are determining factors in identifying recurrent maintenance in the analyzed code artifacts. After completing the Exploratory Data Analysis, we realized which features are most relevant to the maintenance effort based on the change of LOCs and identifying architectural issues. In addition to identifying architecture problems using Architectural Smells, we also seek to verify how the variables relate using Spearman's correlation.

6.2 Research Method

We conducted a DSR process (PEFFERS *et al.*, 2007; OFFERMANN *et al.*, 2009) described in Figure 14 to proposed the approach to identify ATD items. We performed a SMS about ATD, and we observed that the main challenge of ATD is the identification process. The **problem identification** is explained in Chapter 4 following the literature review. Besides, the Chapter 5 that explain the case study about Ericsson TDM process was one of motivations to created a automatic process to identify ATD items in large scale software projects. The **Objective of the Solution** is explained in this Chapter 6 by explaining the proposed method and what is required by the practitioners to find artifacts related to ATD. In **Design and Development** phase, we created a method based on previous work on ATD, summarized in this Chapter 6 and in Chapter 7, we created a Tool to aid extract data from git repositories and implement some steps of the proposed method. Based on previous works and some exploratory tests in Apache git repositories, we defined a proposition and refined the **hypothesis** to validate our method. The **Demonstration** were explained in Chapter 8, which describes the tests performed and semi-structured interviews. Also, we performed the **Evaluation** in Chapter 9. Finally, in Chapters 8 and 9, we showed the **Summary Results** of the study performed in this research.

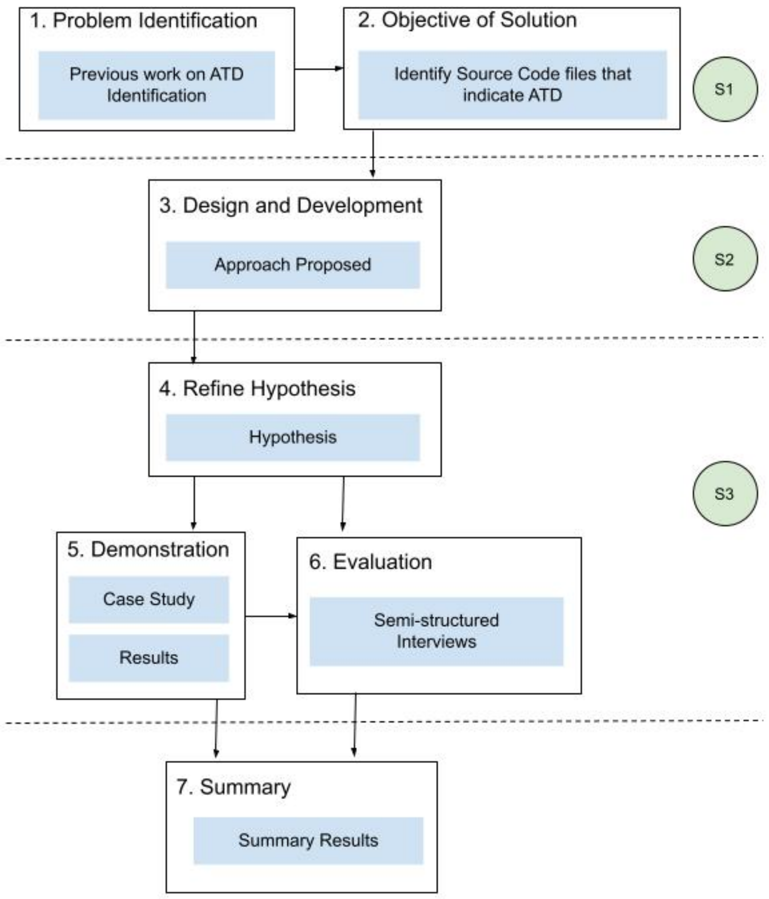

1. Problem Identification
Previous work on ATD Identification
2. Objective of Solution
Identify Source Code files that indicate ATD
S1
3. Design and Development
Approach Proposed
S2
4. Refine Hypothesis
Hypothesis
S3
5. Demonstration
Case Study
Results
6. Evaluation
Semi-structured Interviews
7. Summary
Summary Results

Figure 14 – Research Process adapted from a proposal by Offermann *et al.* (2009). Stage 1 (S1) - problem identification, Stage 2 (S2) - solution design and Stage 3 (S3) - Evaluation

6.3 Goal and Research Questions

There is still no consensus on how to identify ATD. Currently, the available literature has not yet been able to precisely identify and monitor software source code items that indicate ATD in large-scale software. In addition, there are still few ATD management methods or guides. The tools and methods available are still not sufficient to assess the impact of ATD items within a software system over time. These gaps demonstrate the value of conducting this study.

Characterizing ATD empirically is one of the great current challenges in ATD, mainly identifying ATD indicators in software repositories. Once the artifacts that generate ATD are identified, it will be possible to analyze, monitor, and make decisions to pay them or not.

As the ATD theme is not matured, we decided to investigate if source code files that have Architectural Smells that change a lot over time impact other files generating a recurring maintenance effort.

This study aims to identify automatically which source code files from a software code repository indicate the presence of ATD. To achieve this goal, we want to define an approach that generates indicators of ATD through the historical analysis of source code changes in the software code over time. Then, identifying critical areas of the code repository can be helpful to software architect, and developers in making decisions about the effort to maintain code artifacts.

The proposed solution can help answer specific research questions.

RQ1 - How to identify code artifacts affected by ATD? the proposed solution can identify code artifacts affected by ATD through historical analysis of changes that have occurred in the software code over time. By collecting data from code repositories and extracting information from commits and modified files, the proposed solution can extract metrics related to the number of commits, number of files, number of LOCs, the occurrence of files in commits, number of lines modified in commits, and number of lines modified in files over time. Also we can use Architectural Smell to select source code files that indicate the presence of ATD. By answering this question, we can check which characteristics generate a lot of maintenance effort and many recurring changes over time in source code artifacts using AS and software code metrics. It will be possible to identify the project's "hotspots" and investigate what characteristics these files have in common that make the development team spend a lot of energy on their maintenance. In addition, very recurrent changes in the same place on the system may indicate instability or immaturity of the file(s) that undergo a lot of maintenance over time.

RQ2 - How effective and useful is the proposed solution to identify the code artifacts affected by ATD? the proposed solution can be evaluated by comparing its results to those obtained by manual analysis by specialists. The solution's effectiveness can be measured by how well it identifies code artifacts affected by ATD, while its usefulness can be measured by how much time and effort it saves compared to manual analysis by specialists. The proposed solution can also be tested on different code repositories to evaluate its generalizability and applicability in different contexts. Besides, files with many recurring changes and causing changes to propagate to other files that depend on it can indicate the presence of unstable or immature files. Then, by answering this question, it will be possible to know if the magnitude of the maintenance effort spent on this set of files over time is substantial concerning the effort of maintaining the project as a whole or if it is a "minimal" effort, that is, a diluted effort in the project and that does not cause a significant impact.

6.4 Proposition and Hypothesis

The ATD phenomenon is complex because it permeates almost all areas of Software Development Life Cycle (SDLC). From the observations made in several works on ATD, we found a particular pattern related to artifacts that are changed together over time and produce a recurring maintenance effort. Hence, we elaborate a hypothesis about that. So far, we are experimenting with the following ATD concept: there are several definitions of Architectural Technical Debt (ADT) (LI *et al.*, 2014; MARTINI; BOSCH, 2015b; MARTINI *et al.*, 2016; LI *et al.*, 2016; MARTINI; BOSCH, 2017; BESKER *et al.*, 2017a; VERDECCHIA, 2018; VERDECCHIA *et al.*, 2020; VERDECCHIA *et al.*, 2021; XIAO *et al.*, 2021; TOLEDO *et al.*, 2021a; VERDECCHIA *et al.*, 2022). However, these definitions converge to two main aspects: (i) an ATD item must impact the software at the architectural level, and (ii) an ATD item must be a technical debt itself (i.e., its occurrence implies an extra and recurrent maintenance effort throughout the software lifecycle). In this sense, it is possible to observe that the precise identification of an ATD must be made by considering two dimensions. The first one looks at identifying if the phenomenon affects the system at the architectural level. The second one looks at identifying if the phenomenon, mapped into implementation artifacts, leads to an extra and recurrent maintenance effort. From the proposition mentioned above, we have the following hypothesis: **Source code files that indicate the presence of ATD are files that have the following characteristics: i) impact on the software architecture of the system; ii) constantly changed together with other source code files over time; iii) generate a recurring effort to be maintained; iv) propagate/induce recurring changes in other source code files.**

For example, as we can see in Figure 15, given a software system S that has the set of source code files A1, A2, A3, A4, A5, A6, A7, and A8, created and changed in releases R0, R1, R2, R3, R4, and R5. The files A5 and A6 make a set of files with recurrent modifications over the releases. We want to investigate if this set of files can indicate the presence of ATD. We want to investigate if files A5 and A6 have an internal design that can impact modularity, communication among components, or evolvability. Besides, wrong architecture in software can cause slower and more expensive to add new capabilities in the future or make it difficult to fix bugs, for example. Also, we want to check if files A5 and A6 constantly change together over time, then it can imply a high dependency between them. Besides, we want to check if A5 and A6 have recurring efforts to maintain the same set of files over time may indicate that such files need to be refactored to become more independent and less coupled. Finally, we want to check if

A5 and A6 have recurring change propagation in other files; then it can be an indication of poor software design.

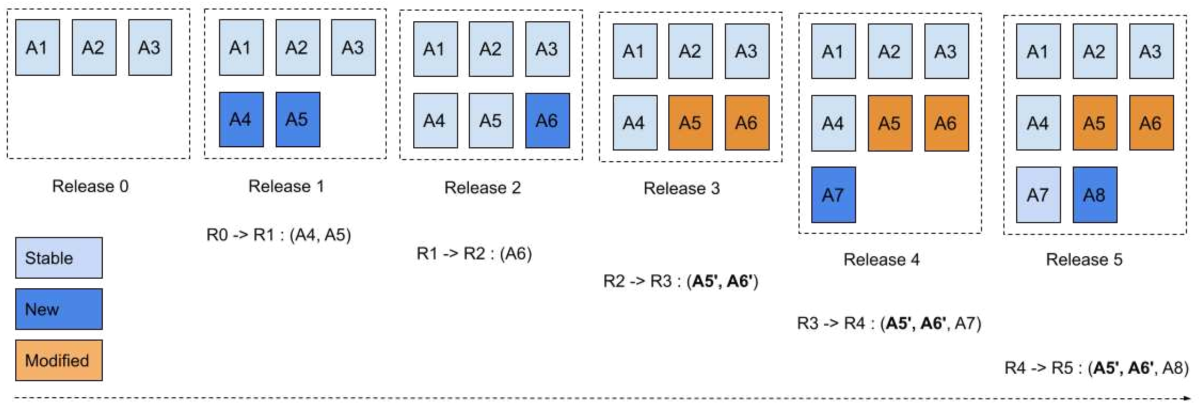


Figure 15 – Set of files with recurrent modifications over the time

6.5 Proposed Approach

As you can see in Figure 16, we defined an approach to indicate source code files with ATD, extracting historical data from the Git repository, and we applied it in Apache Cassandra Repository (Chapter 8) to get data and analyze the results according to RQs. The method was divided into 5 phases described below:

1. **Phase 1** (p1). Extract historical data from commits and modified files from the Git repository.
2. **Phase 2** (p2). Select source code files with AS and calculate specific metrics from those source code files.
3. **Phase 3** (p3). Calculating Quartiles and Selecting Critical Files related to ATD.
4. **Phase 4** (p4). Analyzes critical source code files and their dependent files with co-change.
5. **Phase 5** (p5). Report possible source code files with ATD.

A replication kit[1] containing the steps of the approach are available online.

6.5.1 Variables

In this study, the following **variables** were used for analysis: **accumulated modified LOCs (AMLOC)**, **cyclomatic complexity (CC)** of each source code file, and **file occurrence in commits (FOC)**. We selected these variables due to their relevance source code analysis

[1] https://github.com/mining-software-repositories/cassandra

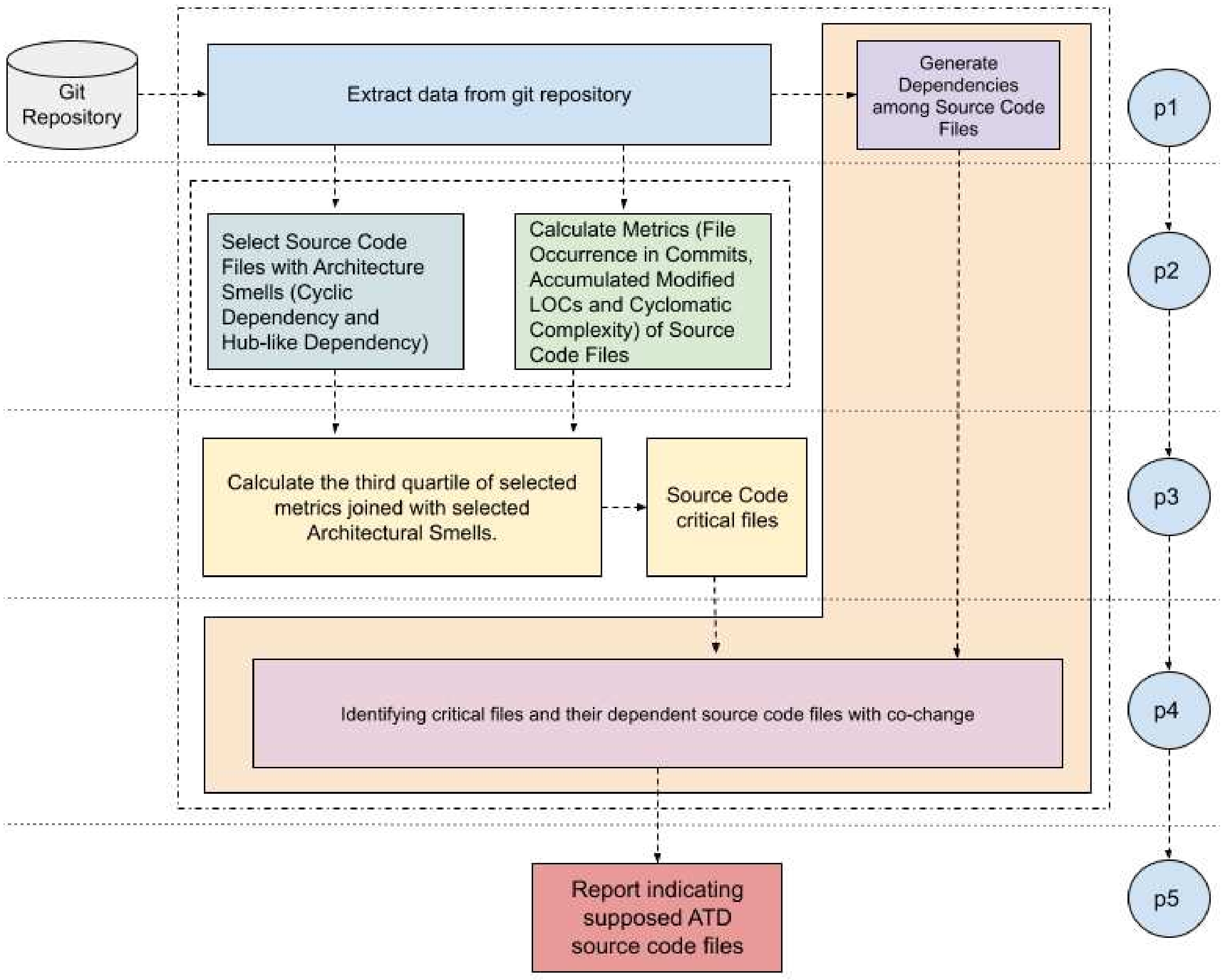

Git Repository
Extract data from git repository
Generate Dependencies among Source Code Files
Select Source Code Files with Architecture Smells (Cyclic Dependency and Hub-like Dependency)
Calculate Metrics (File Occurrence in Commits, Accumulated Modified LOCs and Cyclomatic Complexity) of Source Code Files
Calculate the third quartile of selected metrics joined with selected Architectural Smells.
Source Code critical files
Identifying critical files and their dependent source code files with co-change
Report indicating supposed ATD source code files
p1
p2
p3
p4
p5

Figure 16 – Summary of approach to indicate source code files with ATD

Table 8 – Study Variables

ID	Name	Description
FOC	File Occurrence in Commits	It is the amount of file occurrence in commits during the range analyzed.
AMLOC	Accumulated Modified LOCs	It is related to the accumulated modified lines over time for each selected file in the range analyzed.
CC	Cyclomatic Complexity	It is related to the Cyclomatic Complexity of each file selected in the range analyzed.

provides a lot of critical quantitative measures to analyze the software structure or modules (KRUCHTEN *et al.*, 2019). The Table 8 presents a description of the investigated variables. Also, we considered files with **AS (Cyclic Dependency and Hub-like Dependency)** to select files that can indicate architectural issues.

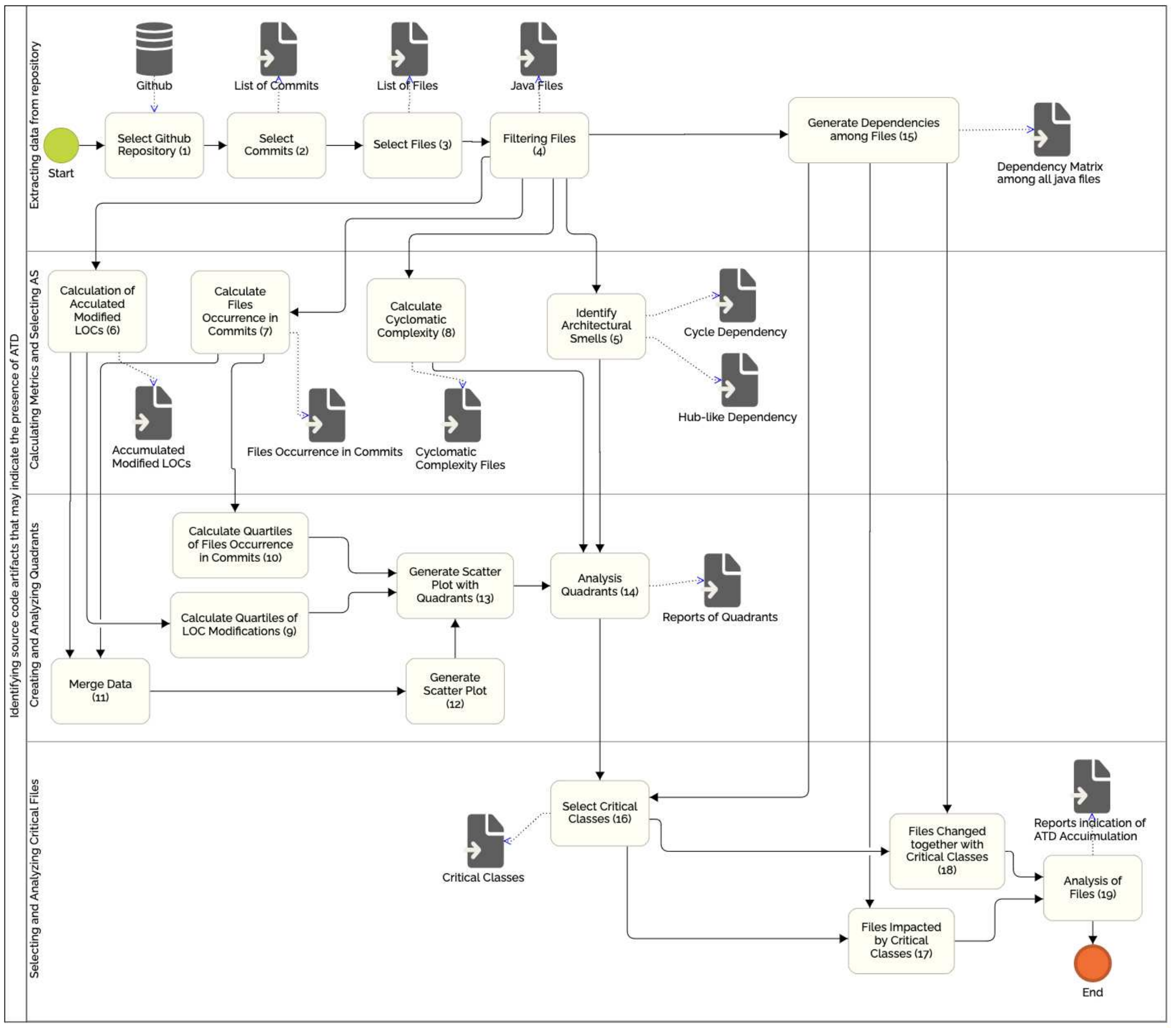

Identifying source code artifacts that may indicate the presence of ATD
Extracting data from repository
Calculating Metrics and Selecting AS
Creating and Analyzing Quadrants
Selecting and Analyzing Critical Files
Github
List of Commits
List of Files
Java Files
Start
Select Github Repository (1)
Select Commits (2)
Select Files (3)
Filtering Files (4)
Generate Dependencies among Files (15)
Dependency Matrix among all java files
Calculation of Acculated Modified LOCs (6)
Calculate Files Occurrence in Commits (7)
Calculate Cyclomatic Complexity (8)
Identify Architectural Smells (5)
Cycle Dependency
Hub-like Dependency
Accumulated Modified LOCs
Files Occurrence in Commits
Cyclomatic Complexity Files
Calculate Quartiles of Files Occurrence in Commits (10)
Calculate Quartiles of LOC Modifications (9)
Generate Scatter Plot with Quadrants (13)
Analysis Quadrants (14)
Reports of Quadrants
Merge Data (11)
Generate Scatter Plot (12)
Select Critical Classes (16)
Critical Classes
Files Changed together with Critical Classes (18)
Reports indication of ATD Accuimulation
Analysis of Files (19)
Files Impacted by Critical Classes (17)
End
HEFLO

Figure 17 – Process overview

6.5.2 *Process Detailed*

In this section, we describe the process based on phases depicted in Figure 16 breaking down into steps detailed in Figure 17.

Phase 1 - Extracting data from the repository

To collect the data associated with the investigated variables from Git repository selected, we followed the steps described in the process depicted in the Figure 17.

According to steps described in Figure 17, we followed the steps below to get data for analysis:

Step 1. The Github repository was selected to be cloned and analyzed.

Step 2. A set of commits related to range from selected versions was selected to be analyzed. To perform an analysis of commits and modified files, for each commit, we used

Pydriller[2] as a tool to extract information from Git repositories.(SPADINI *et al.*, 2018)

Step 3. All modified files from the set of commits were selected to be analyzed. Changes considered in the analyzed files over time. For a given file, as its commits are saved, the changes in lines of code and the Cyclomatic Complexity of the file in each commit are recorded. Only the .java files from the Github code repository. We consider the analysis of a period of time as the analysis performed considered the modifications of the file sets from the range of commit related to the interval between releases selected versions.

Step 4. We applied filters to select only .java files related to the main system implementation.

Phase 2 - Selecting AS and Calculating Metrics

Step 5. We use the tool Arcan (FONTANA *et al.*, 2017) to identify the files that have Architectural Smells. The set of files with ASs can be found in replication kit [3]. We opted for this tool because it was empirically validated and represented state-of-the-art for extracting Architectural Smells from java source code. We selected only classes with **Cycle Dependency** and **Hub-like Dependency** because these kinds of Architectural Smells strongly relate to modularity and the impact of structural dependencies (MARTINI *et al.*, 2018a), (SAS *et al.*, 2019), (SHARMA *et al.*, 2020)

We have to consider two critical sets in this research to understand better the steps of calculating the metrics used in the proposed method. Let C be the set of all analyzed commits, where C = [c1, c2, c3, . . . , c*k*] and *k* represents the total amount of analyzed commits. Let F be the set of all unique files that occur in set C, where F = [f1, f2, f3, . . . , f*i*] and *i* represents the total number of unique files.

Step 6. We calculated the number of accumulated lines modified for each file in the analyzed range. To do this, we computed all LOCs added, and all LOCs removed from each file in the range of analyzed commits.

MLOC is the number of lines of code modified in a file f in a commit c. MLOC is given by formula 1 below:

$$MLOC(f,c) = Added\ lines(f,c) + Deleted\ lines(f,c) \tag{6.1}$$

[2] https://github.com/ishepard/pydriller
[3] https://github.com/mining-software-repositories/cassandra

AMLOC(f,C) is the accumulation of all modified lines in a file f in the set C of all analyzed commits. Where AMLOC(f,C) is given by formula 2 below:

$$AMLOC(f,C) = \sum_{k=1}^{n} MLOC(f,ck) \tag{6.2}$$

where *n* is the total number of commits analyzed.

Step 7. We calculated the occurrence of each file in the set of commits analyzed. To do this, we add up how many times the file appeared in the range of analyzed commits.

The occurrence (OC) of a file f in a commit c is given by formula 3:

$$OC(f,c) = \begin{cases} 1, & \text{if f has changed in c} \\ 0, & \text{otherwise} \end{cases} \tag{6.3}$$

The total occurrences (FOC) of a file f in the set C of analyzed commits is given by formula 4:

$$FOC(f,C) = \sum_{k=1}^{n} OC(f,ck) \tag{6.4}$$

where *n* is the total number of commits analyzed.

Step 8. We calculated the Cyclomatic Complexity of each file in the set of commits analyzed. To do this, we used the Pydriller tool to extract Cyclomatic Complexity from each file in the range of analyzed commits. The file complexity was calculated using the standard McCabe metrics (MCCABE, 1976), well established in research.

Phase 3 - Calculating Quartiles and Selecting Critical Files

Step 9. We calculated the maintenance effort based on line of code changes. Canfora, Cerulo, and Luigi (CANFORA *et al.*, 2007) proposed a software maintenance effort estimation approach based on the observation of changes in artifacts' lines of code over time. In this context, we defined in our work the effort to maintain code artifacts as the cumulative amount of changes in lines of code that a file has undergone over a period between a range of versions (initial commit to final commit). For example: if we consider a range of 10 versions (commits) an A1 file had 1.000 lines changed in this range, A2 file had 100 lines changed in this range, and an A3 file had no lines changed, in that same range, the maintenance effort of A1 is greater than A2 and for A3 means that there was no maintenance effort for this file. We calculate the distribution

Table 9 – Distribution of Maintenace Effort

Scale	Interval
Small maintenance effort	[1, q1]
Average maintenance effort	[q1, q2]
Large maintenance effort	[q2, q3]
Maintenance effort too great	[q3, q4]

of maintenance effort on the change lines of each file analyzed. Aniche *et al.* (2018) proposed in their work an impact scale of analyzed code smells based on a quartile distribution using the size (LOC) of the analyzed files. Then, we based on this approach to make a distribution of quartiles (q1, q2, q3, and q4), calculating the number of lines of code changed, both for commits and files.

Step 10. We can analyze the maintenance effort per commit within the analyzed period (initial commit to final commit). For example: in a c1 commit, a total of 1.000 lines were changed in files A1, A2, and A3; in a c2 commit, a total of 15.000 lines were changed in files A4, A5, and A6. As a result, this last commit had an effort on the order of 15 times greater than the commit c1. That is, c2 had a very great maintenance effort.

Let's consider the following distribution of effort: within a scale S (small), M (medium), L (large) and XL (extra large), where S (low effort), M (medium effort), L (large effort) and XL (exertion too great). Thus, given this scale, the distribution of efforts will follow the following references with the quartiles' intervals described in Table 9. Hence, it is possible to compare the maintenance effort between project commits.

In our context, we use the following meaning for "recurring changes" of a file. It means how often a file was "committed" over the analyzed range of commits.

Based on the same approach as in the previous item, it is possible to make a distribution of quartiles (q1, q2, q3, and q4) by calculating the number of times a file appeared in commits within the analyzed period (initial commit to final commit). In the same way as the previous item, we consider a scale S, M, L, and XL, where S (the file appears in a few commits), M (the file appears in some commits), L (the file appears in many commits) and XL (the file appears in a large number of commits). With that, let's adopt the following scale to represent recurring changes of a file that can be described in Table 10

Step 11. We merged the data gated from steps 6 and 7 to create tuples containing (file, amount of Modified LOCs, amount of File Occurrence in Commits) for each analyzed file to select highlighted points.

Step 12. We generated a scatter plot to show the relationship between the two

Table 10 – Distribution of File Occurrence in Commits

Scale	Interval
low frequency of changes	[1, q1]
the average frequency of changes	[q1, q2]
high frequency of changes	[q2, q3]
very high frequency of changes	[q3, q4]

analyzed variables (amount of Modified LOCs, amount of File Occurrence in Commits) for the analyzed files.

Step 13. We created a second scatter plot plotting two new axes corresponding to the quadrants (Q1, Q2, Q3, Q4), where the quadrants follow the relationships (LOCs Modification, Files Occurrence in Commits) like (high, high), (low, high), (low, low), and (high, low) respectively. Plotting these quadrants could help to analyze and select outstanding points.

Phase 4 - Identifying Critical Files with Co-Changes

Step 14. We analyzed each quadrant looking for the values that stood out the most in relation to the highest values for the Modified Loc amounts and Commit Frequency amounts.

Step 15. We generated a Dependency Matrix among all classes (.java files) to find the dependencies among analyzed classes (.java files). For instance, if a class (A.java file) A imports Class B (B.java file), then class A depends on Class B.

Step 16. We selected the most critical files (critical classes that can affect Architectural level) based on thresholds defined in steps 9 and 10 and the found ASs.

Step 17. For each critical file selected, the files that depend on it were identified, that is, the files that import the critical file. For this, we applied a filter under the dependency matrix created in step 15.

Step 18. The files that appear together with the critical classes in the same commits were identified. To do this, we selected the files that have co-change with critical classes.

Phase 5 - Reporting indicating Source code files with ATD

Step 19. We analyzed the intersection files among files impacted by critical classes and files that changed together with critical classes. So, in this step, we considered only the files that depend on critical classes and also have co-change with critical classes.

At the moment, the steps above can be performed in any git repository related to java projects to extract the ATD items according to the proposed method.

6.6 Conclusion

As we have seen in this chapter and before chapters, we have observed that the ATD identification process is one of the most critical issues in the ATD context. We observed the most common problems in the literature review of ATD, also carried out a case study in Ericsson Company regarding factors that indicate TD in large-scale projects. Besides, we performed some exploratory tests in Apache Git repositories to match common problems and characteristics of ATD issues in code repositories. After that, we defined an automatic method based on the propositions and hypotheses that guide the tests of our method. Finally, we propose doing one more study and interviews in a large-scale project to validate our proposed method, explained in chapter 8. Also, we performed the proposed method in other repositories and evaluated in chapter 9 using SATD techniques to identify issues related to archictural problems. The aim is for the method to be extensible for both researchers and practitioners to be used in their repositories to extract source code files affected by ATD. There is a prototype available online [4] that is detailed in a real case explained in chapter 8.

[4] https://github.com/mining-software-repositories/cassandra

7 SYSREPOANALYSIS: A TOOL TO ANALYZE AND IDENTIFY CRITICAL AREAS OF SOURCE CODE REPOSITORIES

In this chapter, we describe the tool that we created to aid the process of collecting data from git repositories. This tool implements important steps of the proposed method, for example, the automatic extraction of information of commits and modified files according to a range of commits, the metrics chosen in the proposed method are calculated automatically and the tool generates some visualizations that aid in analysis critical areas of code repositories that can indicate extra effort in maintenance and evolution.

7.1 Introduction

The maintainability of software is related to the ease with which its components can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment (COMMITTEE *et al.*, 1990). As software evolves, it undergoes improvements, modifications, and adaptation to new requirements. As a result, the source code often becomes more complex, undermining maintainability over time (SZŐKE *et al.*, 2017). In this sense, (TORNHILL, 2019) defines the highlighted points of software as its complex parts that have changed quickly. Identifying such points can help the development team as it directs its focus to the places where possible *bugs* and revision points are found.

Mining Software Repositories (MSR) techniques help software engineers understand particular phenomena within their repositories. An important phenomenon to be observed should be which areas of the software spend the most effort by the development team or even which directories/files are constantly changed throughout the development cycle. These areas are closely linked to code quality and are guides for improvements and refactorings (TORNHILL, 2019).

We created the SysRepoAnalysis tool to facilitate the process of extracting data, generating the studied metrics, plotting special graphs such as scatter plot and treemaps with heatmap based on the selected metrics. It is an open-source tool that analysis commits and modified files from the source code repository to identify highlighted files and highlighted areas based on historical analysis of software metrics. It implements an asynchronous analysis of source code repository via web application and message broker that intermediate the actions among producers and consumers that execute the actions registred in web application, allowing users to execute the approach in a fully automated fashion and allows extensions of new actions adding new producers and new consumers in a Message Broker to fire new actions over the repository analysis. The tool is available on GitHub[1].

The SysRepoAnalysis is language-independent, i.e., it is not bound to any specific source code program language. For this reason, the SysRepoAnalysis tool can analyze any kind of git source code repository. However, there is a special analysis of java projects that identifies highlighted files based on used metrics described in section 7.3.

The SysRepoAnalysis tool targets both researchers and practitioners as intended users. On the one hand, researchers can leverage the implementation of SysRepoAnalysis to independently conduct experimentation, replicate previous results, and extend or refine the source code repository analyzed. On the other hand, practitioners can use the tool in their current practice to gain an overview of highlighted areas or source code files that outstand in a repository, facilitating details analysis over time.

The remainder of this chapter is organized as follows: the second section presents the related work. The third section presents the Tool. The fourth section presents the usage scenarios. Finally, we provide our conclusions and view on future work about this tool in the last section.

7.2 Related Work

CodeCity generates an interactive 3D visualization tool for analyzing large software systems. Using a city metaphor, it depicts classes as buildings and packages as districts of a “software city” it represents systems as cities, where classes are depicted as buildings and packages as the districts of the city. The city provides an overview of the system, and by navigating around it, possibly investigate its structural organization using CK metrics (CHIDAMBER; KEMERER, 1994). It is necessary to install a plugin to allow the software project analysis for each version and generate the 3D visualization (WETTEL; LANZA, 2008). The drawback to this plugin is that it requires the developers to generate the graphs and look at them actively.

CodeScene provides code visualizations based on version-control data and machine learning algorithms that identify social patterns and hidden risks in code (TORNHILL, 2018). It detects hotspots—complex code that an organization has to work with frequently and prioritizes technical debt based on how the developers work with the code. However, as this tool uses a set of concentric circles to represent the structure of directories and files of the repository, it is not

[1] https://github.com/Technical-Debt-Large-Scale/sysrepoanalysis

appropriate to view complex software with thousands of directories and thousands of files.

Understand[2] is a powerful tool that supports multiple languages and which allows the code to be analyzed and visualized in different ways, including different metrics and the use of treemaps. However, it is a commercial tool with a private license to be used.

Umemura (2017) implements a tool for monitoring the entropy and metrics of software from Git repositories. Its purpose is to assist designers and developers in project management. In the Treemap generated by the tool, the area of each rectangle is measured based on the entropy value of each project file and package. However, this tool does not consider software metrics like modified LOCs and the frequency of files in commits. Data visualization tools are used in work to facilitate understanding of the behavior of software metrics over time.

Avancini (2021) a study is carried out in the area of Software Ecosystem (SECO) research, focusing on systems that provide functions and/or services to other systems (APIs). From this, a software visualization tool is developed to help analyze and evaluate the use of APIs based on metrics of its ecosystem. Once the steps of data extraction, analysis, and obtaining the metrics have been carried out, the visualization proposed in work consists of a Treemap combined with a Heatmap. However, this tool does not provide a historical analysis of the source code repository.

Despite covering several source code analysis issues, the reviewed tools do not focus on identifying automatically source code files that can take a high effort of maintenance over time analyzing the repository as a whole considering historical analysis. Besides, our tool uses Treemaps and Heatmaps as software repository visualization techniques. The literature points to Treemap as a visualization technique that optimizes the use of the visualization space. Therefore, it is suitable for producing a holistic view of complex systems, highlighting their characteristics and hierarchy. The Heatmap coloring method allows the determination of a feature and its display on a color importance scale. Areas with higher values are highlighted with “stronger” colors in the produced structure.

7.3 SysRepoAnalysis Tool Overview

7.3.1 Architectural Components

In this section we describe each component of the architecture depicted in figure 18

[2] https://scitools.com

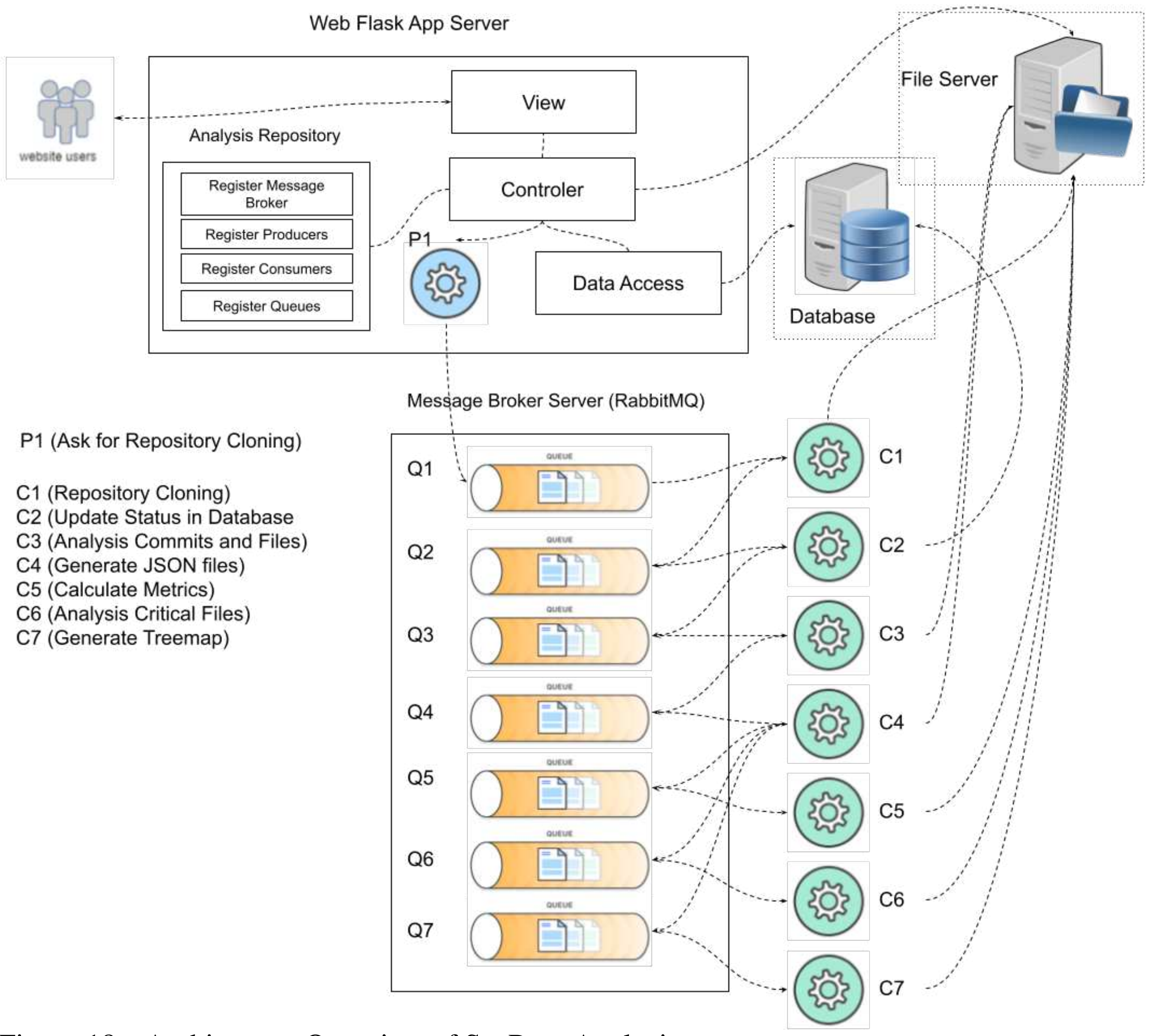


Figure 18 – Architecture Overview of SysRepoAnalysis

Web Flask App Server. The SysRepoAnalysis is hosted on Flask Server[3]. The Flask server allows the SysRepoAnalysis tool to interact with Github repositories, Database server, FileServer and Message Broker Server. There are four components in Flaks Server: *View components* to manage I/O forms, *Controller components* to control interactions among users interactions requests with application, fire actions over repositories under analysis, interaction with database via *DataAccess component* and *Analysis Repository component* that is responsible to registry the Producers, Consumers, actions and Message Broker Server.

Message Broker Server that is an intermediary module that translates a message from the messaging sender (producers) to the messaging receiver (consumers) using queues to control the actions requests over repositories. It is running over RabbitMQ[4] because it can be

[3] https://flask.palletsprojects.com/en/2.1.x/
[4] https://www.rabbitmq.com/

deployed in distributed configurations to meet high-scale, high-availability requirements and has easy integration with python applications.

There is a **Producer (P1)** that starts the process of cloning the user repository. This producer is implemented in Web Flask App Server to facilitate the user web integration. This producer (P1) sends a message to **queue (Q1)** that is responsible for controlling the cloning repository requests. The **Consumer C1** consumes the actions (messages) from the queue (Q1) and executes the cloning process of users repositories in **File Server**. The File Server is responsible for saving cloned repositories and file results generated from Web Flaks App Server and other consumers interacting with this web application. The **Consumer C2** receives messages from **queue (Q2)** that is responsible for controlling the status actions of repositories and updating the status on **Database**. **Consumer C3** receives messages from **queue (Q3)** that is responsible for controlling the requests about actions to analyze commits and modified files from repositories after that C3 saves file results analysis in File Server. **Consumer C4** receives messages from **queue (Q4)** that is responsible for controlling the requests about actions to generate structured JSON files from repositories analysis after that C4 saves file results analysis in File Server. **Consumer C5** receives messages from **queue (Q5)** that is responsible for controlling the requests about actions to calculate software metrics from repositories analyzed, after that C5 saves file metrics results in File Server. **Consumer C6** receives messages from **queue (Q6)** that is responsible for controlling the requests about actions to analyze highlighted files from repositories analyzed and generate scatter plots about that after that C6 saves reports results in File Server. **Consumer C7** receives messages from **queue (Q7)** that is responsible for controlling the requests about actions to generate treemap and heatmap based on calculated metrics from repositories analyzed, after that C6 saves reports results in File Server.

7.4 Usage Scenarios

SysRepoAnalysis supports four main usage scenarios, which are described in the following sections. These scenarios are illustrated by using Apache Kafka[5] as an example. We use for the analysis the main branch of Kafka cloned in HEAD (2022-06-04). It was analyzed in this repository more than 1.055 directories, 4.775 files, and more than 10.092 commits. This code repository was chosen because it is considered a large source code repository very used by

[5] https://github.com/apache/kafka

the developer's community and adopted in the real world by large companies[6].

7.4.1 Extract commits and files

The first usage scenario that is supported by our tool is the analysis of commits and modified files from the branch selected for analysis after the repository is cloned in the File Server. After the cloning process, it is fired a historical analysis of all commits and all modified files based on Pydriller[7] component extract information from the Git repository, such as commits, developers, modifications, diffs, source codes, and quickly export CSV files.

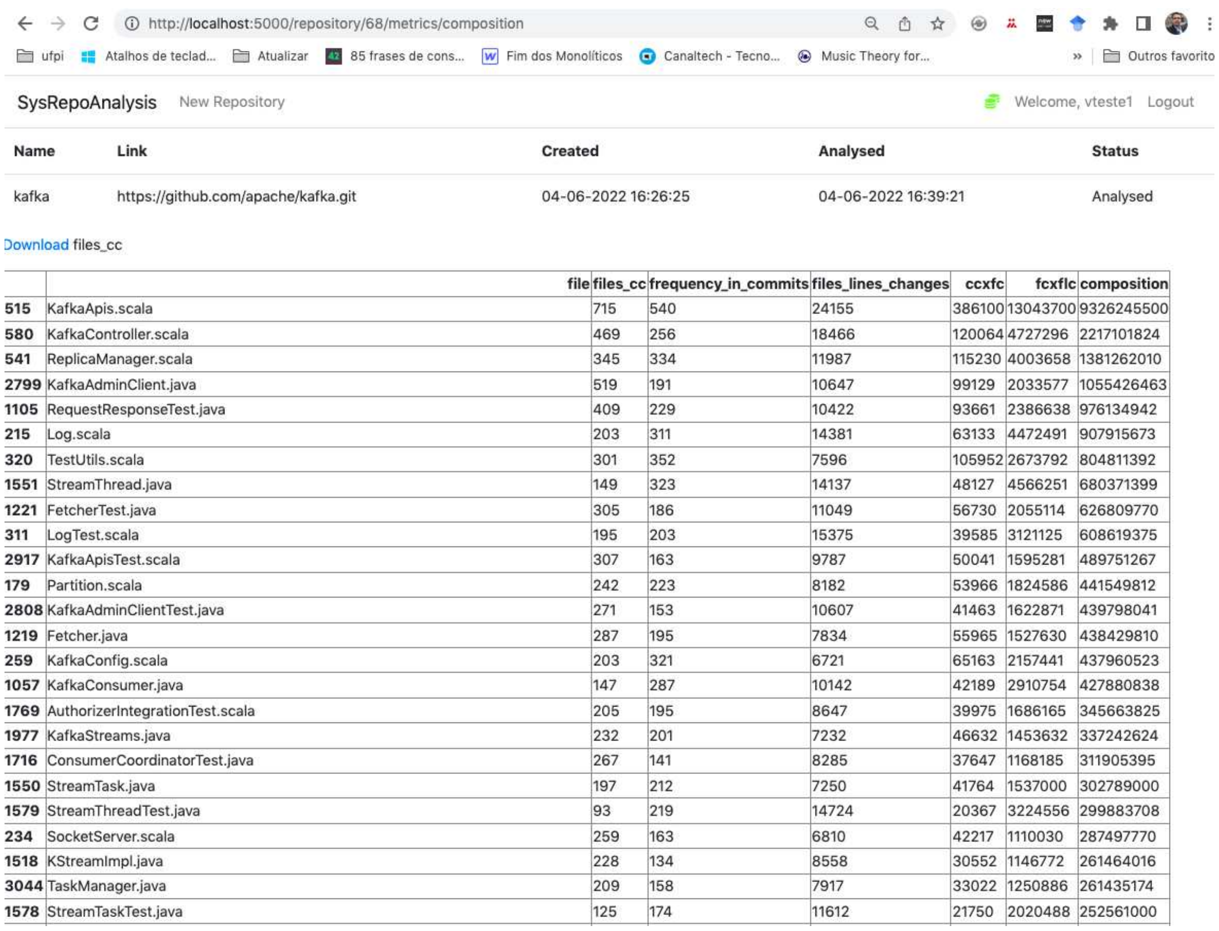
7.4.2 Calculate software metrics

The main metrics used are cyclomatic complexity (it is related to the Cyclomatic Complexity of each file selected in the range analyzed), accumulation of LOC modifications (it is related to the accumulated modified lines over time for each selected file in the range analyzed) (GRAYLIN *et al.*, 2009), and frequency of occurrence of files in commits over time (it is the amount of file occurrence in commits during the range analyzed). In addition, we use the composition (the product of these three metrics) to find very complex source code files that are frequently modified and have many LOCs changed over time. You can see the result of composition among three metrics related to all commits analyzed in the main branch of Kafka repository in figure 19

7.4.3 Analysis of highlighted Files

The tool basically uses the relationship between the metrics "accumulation of LOC modifications (AMLOC)" and "frequency of occurrence of files in commits (FOC)" to generate a scatter plot that relates these two metrics for the .java files in the kafka/core/src folder and selects the files that have the value of AMLOC greater than the 3rd. quartile of AMLOC distribution and that have a FOC value greater than the 3rd. quartile of FOC distribution. Anichie et al.(ANICHE *et al.*, 2018) proposed in their work an impact scale of analyzed code smells based on a quartile distribution using the size (LOC) of the analyzed files. With this, the SysRepoAnalysis tool shows the list of files that have files that have the highest frequency of commits and the highest number of lines modified over time. The tool comes pre-configured to show the top 20 .java files

[6] https://kafka.apache.org
[7] https://github.com/ishepard/pydriller



http://localhost:5000/repository/68/metrics/composition

SysRepoAnalysis New Repository Welcome, vteste1 Logout

Name	Link	Created	Analysed	Status
kafka	https://github.com/apache/kafka.git	04-06-2022 16:26:25	04-06-2022 16:39:21	Analysed

Download files_cc

	file	files_cc	frequency_in_commits	files_lines_changes	ccxfc	fcxflc	composition
515	KafkaApis.scala	715	540	24155	386100	13043700	9326245500
580	KafkaController.scala	469	256	18466	120064	4727296	2217101824
541	ReplicaManager.scala	345	334	11987	115230	4003658	1381262010
2799	KafkaAdminClient.java	519	191	10647	99129	2033577	1055426463
1105	RequestResponseTest.java	409	229	10422	93661	2386638	976134942
215	Log.scala	203	311	14381	63133	4472491	907915673
320	TestUtils.scala	301	352	7596	105952	2673792	804811392
1551	StreamThread.java	149	323	14137	48127	4566251	680371399
1221	FetcherTest.java	305	186	11049	56730	2055114	626809770
311	LogTest.scala	195	203	15375	39585	3121125	608619375
2917	KafkaApisTest.scala	307	163	9787	50041	1595281	489751267
179	Partition.scala	242	223	8182	53966	1824586	441549812
2808	KafkaAdminClientTest.java	271	153	10607	41463	1622871	439798041
1219	Fetcher.java	287	195	7834	55965	1527630	438429810
259	KafkaConfig.scala	203	321	6721	65163	2157441	437960523
1057	KafkaConsumer.java	147	287	10142	42189	2910754	427880838
1769	AuthorizerIntegrationTest.scala	205	195	8647	39975	1686165	345663825
1977	KafkaStreams.java	232	201	7232	46632	1453632	337242624
1716	ConsumerCoordinatorTest.java	267	141	8285	37647	1168185	311905395
1550	StreamTask.java	197	212	7250	41764	1537000	302789000
1579	StreamThreadTest.java	93	219	14724	20367	3224556	299883708
234	SocketServer.scala	259	163	6810	42217	1110030	287497770
1518	KStreamImpl.java	228	134	8558	30552	1146772	261464016
3044	TaskManager.java	209	158	7917	33022	1250886	261435174
1578	StreamTaskTest.java	125	174	11612	21750	2020488	252561000

Figure 19 – Composition of three metrics

that stand out the most according to these pre-defined criteria. You can see the Scatter plot that shows probable highlighted files depicted in figure 20. The top 20 highlighted files identified by the tool are depicted in figure 21. The tool also provides a report with other information about other files that do not appear in the top 20 highlighted files to allow developers to compare the software metrics among them.

7.4.4 Generate Treemaps with heatmaps

The tool allows the generation and display of a treemap(JOHNSON; SHNEIDERMAN, 1998) of the repository's directory and file structure and the rendering of a heatmap(LAWSON, 1956) based on the metric (cyclomatic complexity, accumulation of LOC modifications, frequency of occurrence of files in commits over time and composition of these three aforementioned metrics) chosen to be analyzed. For example, you can see the Kafka repository treemap using heatmap based on ciclomatic complexity of all files in Kafka repository in figure 22. All treemaps with calculated metrics are generated automatically after the repository is cloned, and the user can view and navigate the treemap generated just by selecting the appropriate option in the

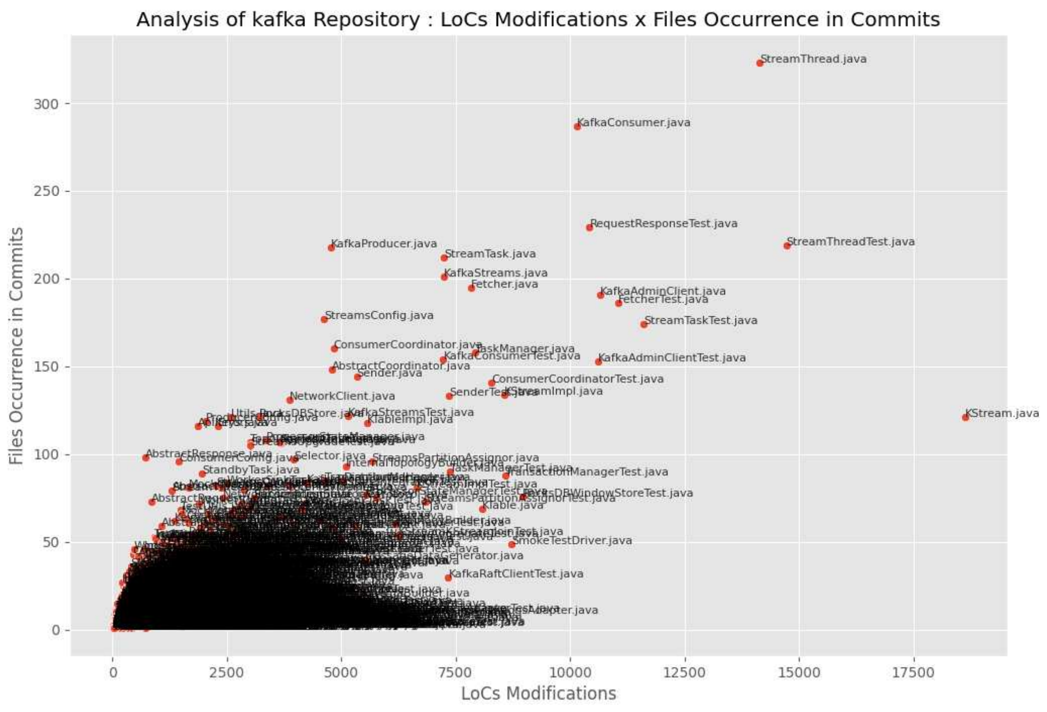
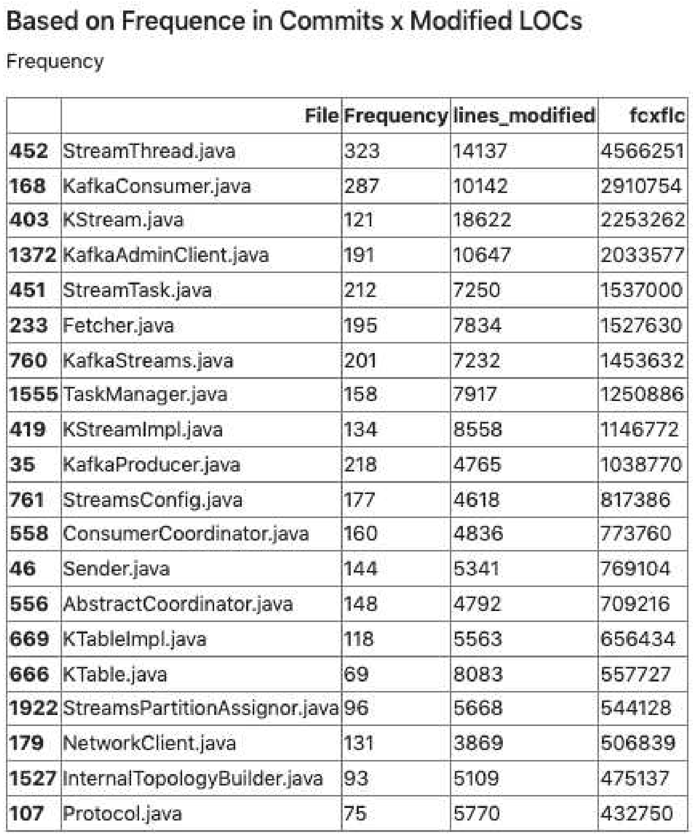


Figure 20 – Scatter plot of .java files



Based on Frequence in Commits x Modified LOCs

Frequency

	File	Frequency	lines_modified	fcxflc
452	StreamThread.java	323	14137	4566251
168	KafkaConsumer.java	287	10142	2910754
403	KStream.java	121	18622	2253262
1372	KafkaAdminClient.java	191	10647	2033577
451	StreamTask.java	212	7250	1537000
233	Fetcher.java	195	7834	1527630
760	KafkaStreams.java	201	7232	1453632
1555	TaskManager.java	158	7917	1250886
419	KStreamImpl.java	134	8558	1146772
35	KafkaProducer.java	218	4765	1038770
761	StreamsConfig.java	177	4618	817386
558	ConsumerCoordinator.java	160	4836	773760
46	Sender.java	144	5341	769104
556	AbstractCoordinator.java	148	4792	709216
669	KTableImpl.java	118	5563	656434
666	KTable.java	69	8083	557727
1922	StreamsPartitionAssignor.java	96	5668	544128
179	NetworkClient.java	131	3869	506839
1527	InternalTopologyBuilder.java	93	5109	475137
107	Protocol.java	75	5770	432750

Figure 21 – Top 20 Kafka highlighted files suggested by the tool

selected repository. A sample of treemap navigation is available at Kafka GitHub Page [8].

Among the most important features of the treemap we can highlight:

1. The treemap is initially loaded with the repository's base directory, displaying all directories, files, sub-directories, and a color scale related to the density of the selected metric.

2. The rectangle area of each file is related to the number of lines of code in the file. It is proportional to the total lines of code of all files in the repository, that is, the greater the LOC of the file, the greater the area of the rectangle that represents the file.

3. The color density of the rectangle represents the value of the selected metric. The higher the metric number, the darker the rectangle's color.

4. The user can navigate among the directories and files. When the user clicks on a directory, the treemap is redrawn, detailing all the files and sub-directories of the selected directory.

5. When the user clicks on a file, a hint is displayed showing information about the file name, weight (LOC), type (directory or file), and value of the selected metric.

7.5 Conclusions

In this chapter, we have presented the SysRepoAnalysis, an analysis and visualization tool that integrates metrics to help in the identification of highlighted source code files and highlighted areas using treemaps and heatmaps to represent the metrics analyzed in a source code repository under version control.

The chosen source code software repository (Apache Kafka) showed that the proposed tool produces an overview of the directory and file structure of the repository, allowing to obtain details, on-demand, of its components. The coloring technique adopted contributed to the identification of the most affected areas of software, based on the analysis metrics used: cyclomatic complexity, frequency of files in commits, and the number of lines changed over time. The software components that present higher values for the metrics discussed were easily identified since they were highlighted with darker colors.

As future work, we intend to start new empirical studies employing our tool in other large-scale software repositories. Also, we want to integrate with the Arcan tool to allow the extraction of Architectural Smells from the analyzed repository. In addition, we plan to extend the tool by implementing components plug-in style according to the specific needs of practitioners

[8] https://armandossrecife.github.io/kafka-treemap/

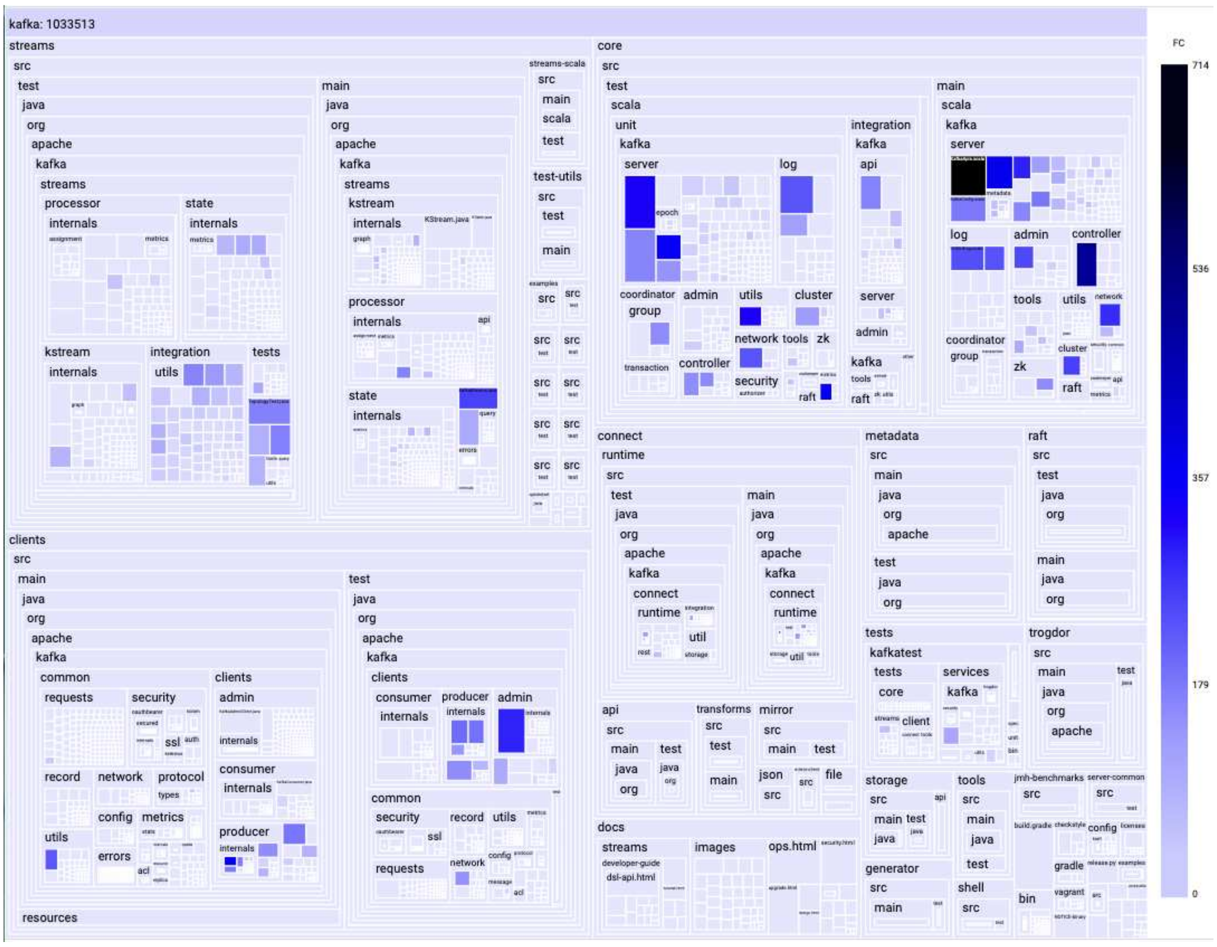


Figure 22 – Kafka Treemap with heatmap based on Ciclomatic Complexity of all files

in the context of large-scale industrial software development. Finally, we intend to extend the SysRepoAnalysis tool with a history mechanism, allowing users to conduct longitudinal analyses on the evolution of source code in their software systems over time.

8 IDENTIFYING SOURCE CODE FILES THAT INDICATE ARCHITECTURAL TECHNICAL DEBT

In this chapter, we present how the proposed method was applied in a real-world case. We describe a study conducted with the Apache Cassandra repository, which was validated by a group of developers from Ericsson. During the study, we followed the steps of the proposed method described in Chapter 6, extracted and analyzed data, and presented the results to the Ericsson developers. This was done to evaluate the effectiveness and usefulness of the proposed solution in identifying artifacts affected by ATD.

8.1 Introduction

We used the proposed method, described in chapter 6, on the Apache Cassandra repository[1] as the first step to demonstrate the feasibility of our approach. Next, we triangulate our findings with the results of a semi-structured interview with experienced developers from Ericsson (a European Telecom Company). They are also collaborators of the Apache Cassandra project.

To explore the maintenance and evolution of the Apache Cassandra repository, we identify source code files with AS (SURYANARAYANA *et al.*, 2014; AZADI *et al.*, 2019) to check if there is any relationship between recurrent maintenance efforts in a set of source code files and if this set of files can indicate ATD over the time. We performed several analyses in modified source code files from version 3.0.0 to 3.11.11 in the Apache Cassandra repository. We also selected three software metrics (CHIDAMBER; KEMERER, 1994; GRAYLIN *et al.*, 2009; ELISH; AL-KHIATY, 2013) for this study: accumulated modified LOCs, the Cyclomatic Complexity, and the files frequency in commits.

We looked for patterns that could be helpful to identify software artifacts (source code files) frequently changed over time and check if these artifacts generate any impact of changes in the other source code files of the project (e.g., change propagation (HASSAN; HOLT, 2004) related to addition or removal of LOCs in source code files).

The main analysis was done concerning the recurrent efforts found in files with Architectural Smells (Cycle Dependency and Hub-like Dependency). Such smells can indicate modularity violations, which is a critical aspect used to detect ATD (SURYANARAYANA *et al.*, 2014). During the semi-structured interview, we presented our findings to developers from

[1] https://github.com/apache/cassandra

Ericsson and asked them if our results make sense to them. They were quite receptive to the approach and the results. In fact, they agreed that the code files indicated by our method indeed are, somehow, involved with existing ATDs in the Apache Cassandra.

In this chapter, we report the findings of our investigation, which address the main research question: *How to identify code artifacts affected by ATD?*

8.2 Research Design

In our research, we employed the Design Science Methodology (DSM) and followed the process designed by Offerman (OFFERMANN *et al.*, 2009). The process includes the following stages: Problem Identification, Solution Design, and Evaluation. In the remainder of this section, we describe how we carried out each phase.

8.2.1 Problem Identification

Our research was motivated by the challenges faced by developers and architects at a target Company to handle ATD. The problem was identified by one of the authors in interactions with multiple developers and architects at regular Kaizen events. While this Company has some processes to manage code TD (SOUSA *et al.*, 2021), we identified that ATD management could be improved, starting with ATD identification. Further, to improve our understanding of the problem, we conducted a systematic mapping study (SOUSA *et al.*, 2023).

8.2.2 Solution Design

Our systematic mapping study also was the main basis for developing the solution present in the chapter 4. We learned that existing definitions of ATD (LI *et al.*, 2014; MARTINI; BOSCH, 2015b; MARTINI *et al.*, 2016; LI *et al.*, 2016; MARTINI; BOSCH, 2017; BESKER *et al.*, 2017a; VERDECCHIA, 2018; VERDECCHIA *et al.*, 2020; VERDECCHIA *et al.*, 2021; XIAO *et al.*, 2021; TOLEDO *et al.*, 2021a; VERDECCHIA *et al.*, 2022) converge into two main aspects: (i) an ATD item must impact the software at the architectural level, and (ii) an ATD item must be a TD itself, i.e., its occurrence implies an extra and recurrent maintenance effort throughout the software lifecycle. Our solution accounts for these two dimensions at the same time to identify code artifacts that indicate the presence of ATD in a given project.

8.2.3 Evaluation

To validate our solution, we formulated and verified the following hypothesis: *Source code files that indicate the presence of ATD are files that have the following characteristics: i) impact on the software architecture of the system; ii) constantly changed together with other source code files over time; iii) generate a recurring effort to be maintained; iv) propagate/induce recurring changes in other source code files.*

To verify our hypothesis, we focused on the Apache Cassandra project. The target Company uses Cassandra as a cornerstone of multiple products in its portfolio. It is a free and open-source, distributed, NoSQL DBMS designed to handle large amounts of data across many servers, which makes it a good fit for many telecom-related use cases. The target Company contributes to the Cassandra Project actively, having 15 developers dedicated to supporting the evolution of the software system. Finally, the Cassandra project has a significant scale and sufficient lifespan (from 2008) and is quite popular in the industry in general.

The evaluation included using data from the Cassandra repository[2] and semi-structured interviews. The data we collected includes commits from the releases 3.0.0 (2015) to 3.11.11 (2021). The releases family 3.x was selected because it is still the most used version in production by the community.

Interview Participant Selection. We used a convenience sampling strategy to select the subjects for the semi-structured group interview (WOHLIN *et al.*, 2012). To get a deep understanding of our research topic, the subjects should have experience related to Cassandra software development. In this research, it was convenient to select and access the participants from this Company developers because there are a lot of developers that collaborate on this Project.

Semi-Structured Interview. We conducted a semi-structured interview in May 2021 that is described and detailed in the following. We reviewed the main aspects of the proposed solution, like ATD concepts, revision history in source code repositories, and the metrics used. We started the interview by introducing the researchers and the purpose of the interview to four of the Company's experienced developers, who are also contributors to the Cassandra project, via video conference meeting. The interviewees had 2 to 5 years of experience in the investigated project. We explained the main concepts and technical details of the solution, including the process, data structures, and the final results related to the source code files that indicate ATD and

[2] https://github.com/apache/cassandra

the files that they impacted. Using the prototype available in the replication kit, we demonstrated the solution presenting the results and how the proposed solution can help developers and software architects find critical source code files related to ATD. Also, we reinforced the research results, asked the respondents questions, and got feedback on-the-fly. Two authors of this paper conducted the interview. The video conference meeting was recorded, and the authors helped with questions for the developers who participated in the video conference. The presentation and discussion lasted about 35 minutes. The video conference was transcribed into text for further analysis. The notes were discussed with the respective interviewees to ensure that they reflected what was discussed during the meeting. This documentation is available in the replication kit.

Data Analysis. We described our data analysis arranged by RQ. You can see more details about that in steps from phase 4, and in final step from phase 5, depicted in Figure 17.

RQ1. To answer this RQ, we identify the files that have AS, more specifically CD and HLD, to select the initial set of files that can impact the software architecture in the system under analysis, we also apply the accumulation of modified LOCs, file occurrence in commits, and the CC (Cyclomatic Complexity), described in the second step from phase 4 to get the list of critical classes.

In first step of phase 2 (identify AS), we found source code files that can impact the system's modularity, very complex files, and these files have AS. So, we can validate the first item from our hypothesis by finding these source code files.

In steps from phase 4 and phase 5, we found source code files that are frequently changed over versions, i.e., files that appear in many commits over time, and in addition, have other files with "co-change", so, we can validate the second item of our hypothesis.

RQ2. The focus was to check the validity of the approach using real project data available in a Github repository that is actively used by developers who maintain this project. The critical files found in R1 are files that can impact a lot of other files in the system under analysis because CD and HLD are AS with great change propagation (SHARMA *et al.*, 2020). Hence, if we identify the files that depend on critical files, we can find the files that can change together with critical files. To do so, we can select files that depend on critical files and the files that have co-change with critical files. Finally, we can observe if these sets of files have recurring changes over time and have recurring efforts to be maintained over time if the accumulation of modified LOCs, file occurrence in commits, and CC increases over time.

In this case, the effort is calculated using the amount of LOCs modified in a file over time. The more LOCs are modified (changed, removed, or added) in the same file in an increasing way, and this can indicate a recurring effort to maintain these files. It is also necessary to consider that such files must have a high frequency of commits. So, we can validate the third item of our hypothesis.

Also, to analyze the impact of ATD maintenance on the system under analysis, it was necessary to find the set of critical files, their dependent files, and the files that have co-changed with the critical files. Once these files were found, it was necessary to sum the total effort of their LOC modifications within the analyzed interval to obtain the total effort of maintaining files that indicate ATD in the system. After that, it was necessary to find the total effort of the accumulation of modified LOCs of the other core files of the system, to be able to compare the percentage of maintenance of the source code files that indicate ATD concerning the other core implementation files of the system. Also, in all these files, we can observe the "Ripple Effect" (AGRAWAL; SINGH, 2018; AGRAWAL; SINGH, 2020) among critical files and their dependencies files, so these critical files propagate changes in dependencies files. So, we can validate the fourth item of our hypothesis.

8.3 Proposed Solution

As you can see in Figure 16, we defined an approach to indicate source code files with ATD, extracting historical data from the Git repository, and we applied it in Apache Cassandra Repository to get data and analyze the results according to RQs. The method was divided into 5 phases described below:

1. **Phase 1** (p1). Extract historical data from commits and modified files from the Git repository.
2. **Phase 2** (p2). Select source code files with AS and calculate specific metrics from those source code files.
3. **Phase 3** (p3). Calculating Quartiles and Selecting Critical Files related to ATD.
4. **Phase 4** (p4). Analyzes critical source code files and their dependent files with co-change.
5. **Phase 5** (p5). Report possible source code files with ATD.

The following variables were used as parameter of the proposed solution: **Accumulated Modified LOCs (AMLOC)**, **Cyclomatic Complexity (CC)** (GRAYLIN *et al.*, 2009) of each source code file, and **File Occurrence in Commits (FOC)** (ELISH; AL-KHIATY, 2013). We selected these variables due to their relevance source code analysis provides a lot of critical quantitative measures to analyze the software structure, or modules (KRUCHTEN *et al.*, 2019). Also, we considered files with **Cycle Dependency (CD)** and **Hub-like Dependency (HLD)** to select files with Architectural Smells (AS) that can indicate architectural issues. Besides, we checked Spearman correlation among variables, and we found a correlation between FOC and AMLOC (*correlation=0.6042, pvalue=6.97e-141*), and a correlation between FOC and CC (*correlation=0.5267, pvalue=2.46e-101*).

A replication kit[3] containing the steps of the approach, the data, the scripts, and the prototype are available online. All steps are depicted in Figure 17

8.3.1 Phase 1 - Extracting data from the repository

Step 1. The Cassandra Github repository was selected to be cloned and analyzed.

Step 2. A set of commits related to range from v.3.0.0 to v.3.11.11 was selected to be analyzed. To perform an analysis of commits and modified files, for each commit, we used Pydriller[4] as a tool to extract information from Git repositories.(SPADINI *et al.*, 2018)

Step 3. All modified files from the set of commits were selected to be analyzed. Changes considered in the analyzed files over time. For a given file, as its commits are saved, the changes in lines of code and the Cyclomatic Complexity of the file in each commit are recorded. Only the .java files from the Github code repository in the scr/java/org/apache/cassandra/* directory were analyzed. We consider the analysis of a period of time as the analysis performed considered the modifications of the file sets from the range of commit related to the interval between releases v.3.0.0 and v.3.11.11.

Step 4. We applied filters to select only .java files related to the main system implementation. To filter only core Cassandra implementation files we point to the directory src/java/org/apache/cassandra. This folder was selected because it represents the main source code implementation in this repository.

8.3.2 Phase 2 - Selecting AS and Calculating Metrics

Step 5. We use the tool Arcan (FONTANA *et al.*, 2017) to identify the files that have Architectural Smells. The set of files with ASs can be found in replication kit. We opted for this tool because it was empirically validated and represented state-of-the-art for extracting

[3] https://github.com/mining-software-repositories/cassandra
[4] https://github.com/ishepard/pydriller

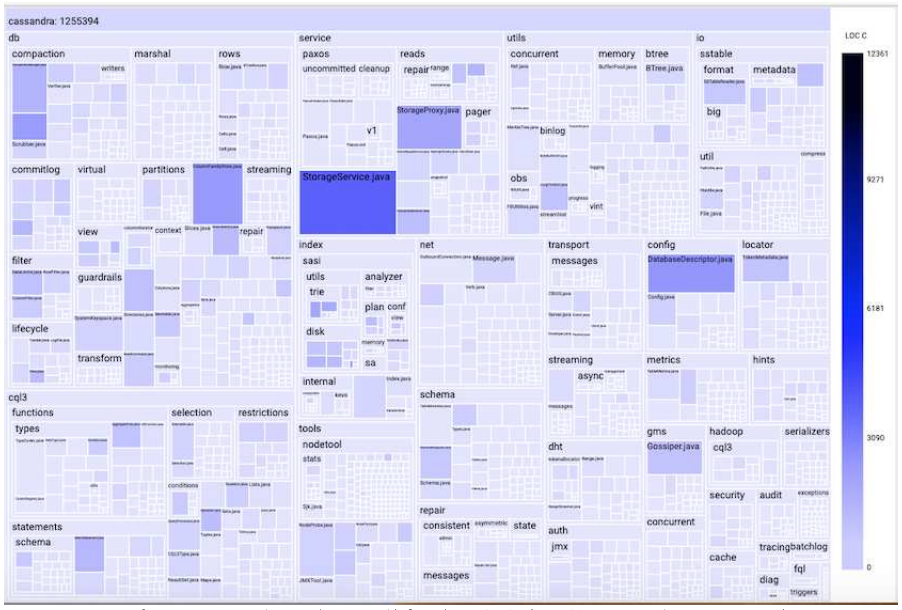


Figure 23 – Treemap of Accumulated Modified LOC in Cassandra Repository

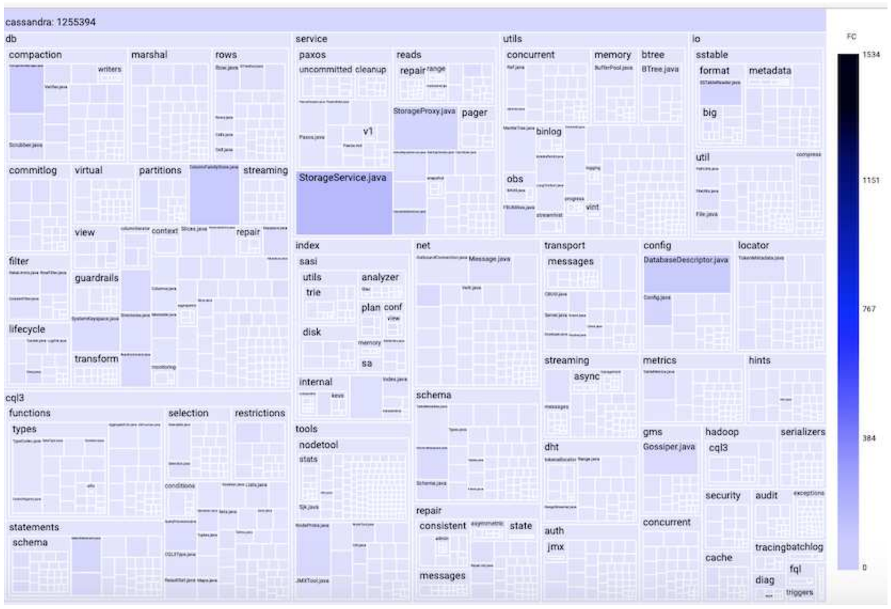


Figure 24 – Treemap of File Occurrence in Commits in Cassanadra Repository

Architectural Smells from java source code. We selected only classes with Cycle Dependency and Hub-like Dependency because these kinds of Architectural Smells strongly relate to modularity and the impact of structural dependencies. (SAS *et al.*, 2019), (SHARMA *et al.*, 2020)

Step 6. We calculated the number of accumulated lines modified for each file in the analyzed range. To do this, we computed all LOCs added, and all LOCs removed from each file in the range of analyzed commits. As you can see in the Figure 23

Step 7. We calculated the occurrence of each file in the set of commits analyzed. To do this, we add up how many times the file appeared in the range of analyzed commits. As you can see in the Figure 24

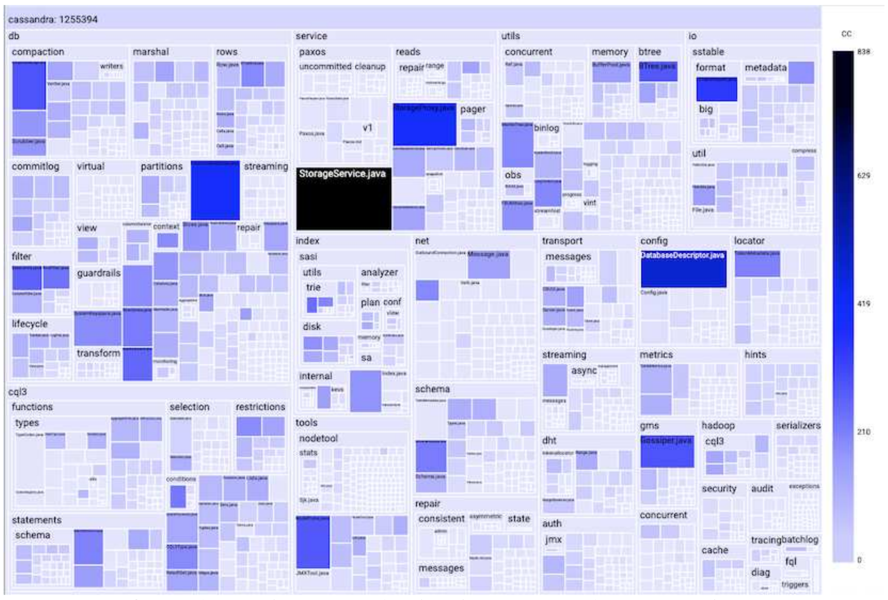


Figure 25 – Treemap of Cyclomatic Complexity in Cassandra Repository

Table 11 – Accumulated Loc Modifications in Commits Quartiles

category	%	quartile	N.Lines
None	0	-	1
Small	25	q1	161
Medium	50	q2	361
Large	75	q3	873
Extra Large	100	q4	40375

Step 8. We calculated the Cyclomatic Complexity of each file in the set of commits analyzed. As you can see in the Figure 25

8.3.3 Phase 3 - Calculating Quartiles and Selecting Critical Files

Step 9. We calculated the maintenance effort based on line of code changes.

Step 10. We can analyze the maintenance effort per commit within the analyzed period (initial commit to final commit).

The Table 11 shows a possible distribution of maintenance effort based on quartiles regarding the number of lines of code changed in a commit. As you can see in the Table 9 that shows the distribution of efforts based on the quartiles' intervals. Hence, it is possible to compare the maintenance effort between project commits.

In addition, we can also consider the maintenance effort for each file based on the number of lines of code changed. Hence, it is possible to compare the maintenance effort between the project files. The following Table 12 shows a possible distribution of quartiles regarding the number of lines of code changed in a file.

Table 12 – Accumulated Loc Modifications Quartiles

category	%	quartile	N.Lines
None	0	-	1
Small	25	q1	12
Medium	50	q2	50
Large	75	q3	150
Extra Large	100	q4	3537

Table 13 – Files Occurrence in Commits Quartiles

category	%	quartile	N.Commits
None	0	-	1
Small	25	q1	11
Medium	50	q2	15
Large	75	q3	22
Extra Large	100	q4	144

Based on the same approach as in the previous item, it is possible to make a distribution of quartiles (q1, q2, q3, and q4) by calculating the number of times a file appeared in commits within the analyzed period (initial commit to final commit). The Table 13 shows a possible distribution of quartiles regarding the number of file Occurrence in Commits.

Finally, you can see the boxplot related to file occurrence quartiles and LoCs Modifications quartiles in Figure 26. Where we choose the third quartile as the reference to extract highlighted points. The q3 $\geq$ 22 is related to very high file occurrence in commits, and q3 $\geq$ 150 is related to a very high number of accumulated modified lines in the analyzed range.

Step 11. We merged the data gated from steps 6 and 7 to create tuples containing (file, amount of Modified LOCs, amount of File Occurrence in Commits) for each analyzed file to select highlighted points.

Step 12. We generated a scatter plot to show the relationship between the two analyzed variables (amount of Modified LOCs, amount of File Occurrence in Commits) for the analyzed files.

Step 13. We created a second scatter plot plotting two new axes corresponding to the quadrants (Q1, Q2, Q3, Q4), where the quadrants follow the relationships (LOCs Modification, Files Occurrence in Commits) like (high, high), (low, high), (low, low), and (high, low) respectively. Plotting these quadrants could help to analyze and select outstanding points. As you can see in Figure 27

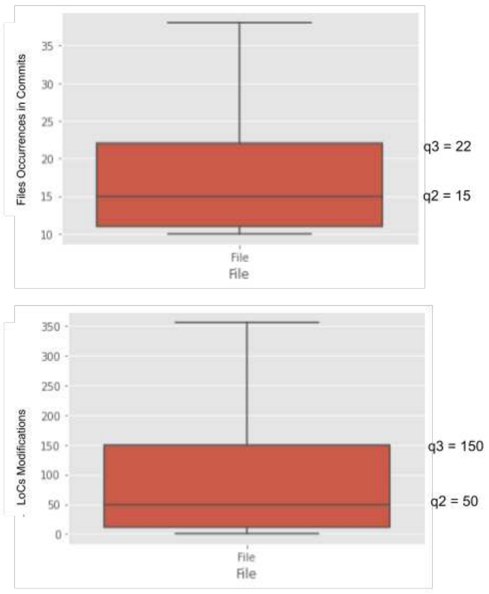


Figure 26 – Boxplot Files occurrence in commits and LoCs Modifications

8.3.4 Phase 4 - Identifying Critical Files with Co-Changes

Step 14. We analyzed each quadrant looking for the values that stood out the most in relation to the highest values for the Modified Loc amounts and Commit Frequency amounts.

Step 15. We generated a Dependency Matrix among all classes (.java files) to find the dependencies among analyzed classes (.java files). For instance, if a class (A.java file) A imports Class B (B.java file), then class A depends on Class B.

Step 16. We selected the most critical files (critical classes that can affect Architectural level) based on thresholds defined in steps 9 and 10 and the found ASs, that is, files from the Q1 quadrant (accumulation of modified LOCs $\geq$ 150 and file occurrence in commits $\geq$ 22), files with high cyclomatic complexity (cc $\geq$ 15) and files that have AS (cycle dependency and hub-like dependency). Even though different cyclomatic complexity thresholds are suggested in the literature (such as 10), we used 15, which is also widely recommended and was chosen by (ANTINYAN *et al.*, 2014) in a similar context to the one where this case was conducted, cc $\geq$ 15 are very complex files and hard to maintain. Hence, we selected 12 files as critical classes with all the above characteristics, as you can see in Table 14.

Step 17. For each critical file selected, the files that depend on it were identified, that is, the files that import the critical file. For this, we applied a filter under the dependency matrix created in step 15.

Step 18. The files that appear together with the critical classes in the same commits were identified. To do this, we selected the files that have co-change with critical classes.

8.3.5 Phase 5 - Reporting indicating Source code files with ATD

Step 19. We analyzed the intersection files among files impacted by critical classes and files that changed together with critical classes. So, in this step, we considered only the files that depend on critical classes and also have co-change with critical classes.

Finally, as you can see in the last step of the proposed approach, We selected critical classes and impacted files by them. Here, we have 12 critical classes (Table 14) that indicate ATD and 251 files impacted by them, as you can see in Figure 28.

8.4 Results

In this section, we present the results of our study arranged by RQ according to RQs described in the proposed method.

8.4.1 RQ1 - How to identify code artifacts affected by ATD?

We first show the findings related to identified critical files that indicate ATD. Second, we show to interviewed on how the observed results can help practitioners make better decisions about future changes in the software.

8.4.1.1 Critical files that indicate ATD

To identify the files that indicate the presence of ATD we validate a hypothesis with four items defined in subsection Evaluation. Our approach (Figure 16) contains a set of steps that, once followed, allow us to identify files that meet these four items of the proposed hypothesis. Once this set of files has been identified, we can conclude that they may indicate the presence of ATD. More specifically files that can compromise system-wide quality attributes, as maintainability and evolvability (LI *et al.*, 2014). For our analysed repository (Cassandra Project), we found 12 source code files (Table 14) that may indicate the presence of ATD within the analyzed commit range. These aforementioned critical source code files have CD, HLD, high file frequency in commits, a high number of accumulated modified LOCs, and a high number of CC.

All the 12 classes aforementioned have high Cyclomatic Complexity (CC $\geq$ 15),

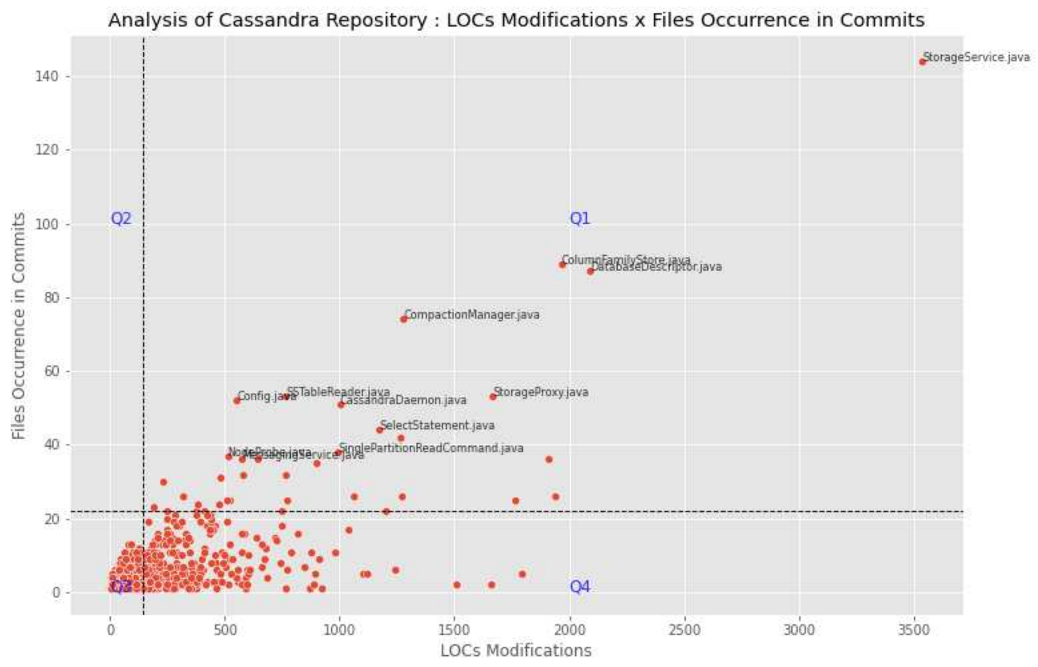


Figure 27 – LoCs Modifications and Files occurrence in commits of Cassandra Repository

showing that these classes can have any kind of structural code problems and should be investigated. Also, we can observe in Table 14 that the values of LOCs and CC increase over time, so it could indicate an increase of maintenance in the critical classes over time. We found that all these identified classes have AS (CD and HLD) with code metrics (AMLOC, CC, and FOC) increasing, generating a lot of maintenance effort, and generating a lot of recurring changes over time.

You can see the Scatter plot between AMLOC and FOC in Figure 27. There are four quadrants: Q1 (high, high), Q2 (low, high), Q3 (low, low), and Q4 (high, low). All the 12 classes aforementioned are in Q1. We can observe that the Q1 contains 31 source code files with a high number of AMLOC and a high frequency of commits in an analyzed range of commits. However, only 12 classes have all properties to be selected as critical files that can indicate ATD.

8.4.1.2 *Developers feedback from RQ1 results*

We presented the importance of historical analysis of all modified files during the commits of releases analyzed. We chose this commit range because it contains a lot of changes, bug fixes, insertion of new features, and improvements in released versions. Hence, we can check a lot of different behaviors about dependency, complexity, and changes in source code files over time. A Developer (Dev1) confirmed this: "*...I think of is that you chose to analyze all commits in the range from 3.0 to 3.11, which is a very interesting choice because I would say that if you take only a patch version, there are no major changes, and refactorings occur so often, but in a broader range of commits it is possible to get things like new features being implemented, major refactorings and things like that.*"

We presented the critical files found by the proposed approach and we showed a list with 12 critical class (Table 14) to Developers and they realize that some classes has recurrent problems over time. A Developer (Dev1) confirmed this: "*...from this list I think Storageservice.java is something that I knew for a fact that it would be on this list before you even showed this mostly because almost everything goes through this class like all features, and the same it is interesting that Nodeprobe.java is in this list because this is basically a tool that is used in Cassandra that helps navigate and look at statuses and things like that so that's also very interesting observation that you saw a high number there.*"

Besides, the Devs identified some files that should be more investigated because these files should not have appeared in the list of critical files. According to Dev1 the Config.java should be removed, and Columnfamilystore.java should be more investigated.

Dev1: "*I'm a little bit surprised on a few of these files though, for example Config.java I'm not sure if this complexity number is high or low. I guess it's low in relation right to everything. I'm a little surprised on Columnfamilystore that this is maybe it's because it's so central in everything that Cassandra is because it's basically the structure of the tables.*"

When we asked if the presented list could help developers make better decisions about which artifacts affect the SA.

Question: "*Do you think that if you have a chance to analyze before creating a release, if you had a list like that, this list could help developers make good decisions, related to ATD?*"

Dev1: *"Yeah it's a interesting question because this like this graph it it tells me what I already know, for example Storageservice.java I already know it's like a god class, it probably you know there's lots of sonar warnings about this class because it's massive right like it's really big. Yes and it does it does look interesting because things like Columnfamilystore.java I wouldn't think is on this list so that's perhaps something that could be further explored right why is this on this list yeah that that would be interesting*" We observed that some classes appear as critical files that were not observed by other tools. "*It would be interesting to explore further*

Table 14 – Critical Classes LOC and Cyclomatic Complexity over time

	filename	LOC_s	LOC_f	CC_s	CC_f	AMLOC	FOC	AS	Impact
1	StorageService.java	3306	3743	717	838	3537	144	CD, HLD	37 classes
2	DatabaseDescriptor.java	1445	1799	374	471	2092	87	CD, HLD	102 classes
3	ColumnFamilyStore.java	1617	1864	314	373	1966	89	CD, HLD	53 classes
4	StorageProxy.java	1867	2143	333	377	1667	53	CD, HLD	10 classes
5	CompactionManager.java	1252	1654	194	265	1279	74	CD, HLD	9 classes
6	SSTableReader.java	1588	1600	322	324	767	53	CD, HLD	60 classes
7	SelectStatement.java	803	918	157	177	1174	44	CD, HLD	7 classes
8	CassandraDaemon.java	522	643	95	117	1001	51	CD, HLD	1 class
9	SinglePartitionReadCommand.java	350	884	63	157	994	38	CD, HLD	3 classes
10	NodeProbe.java	1215	1145	263	255	513	37	CD, HLD	52 classes
11	MessagingService.java	929	915	159	165	574	36	CD, HLD	20 classes
12	Config.java	234	310	4	32	553	52	CD, HLD	1 class

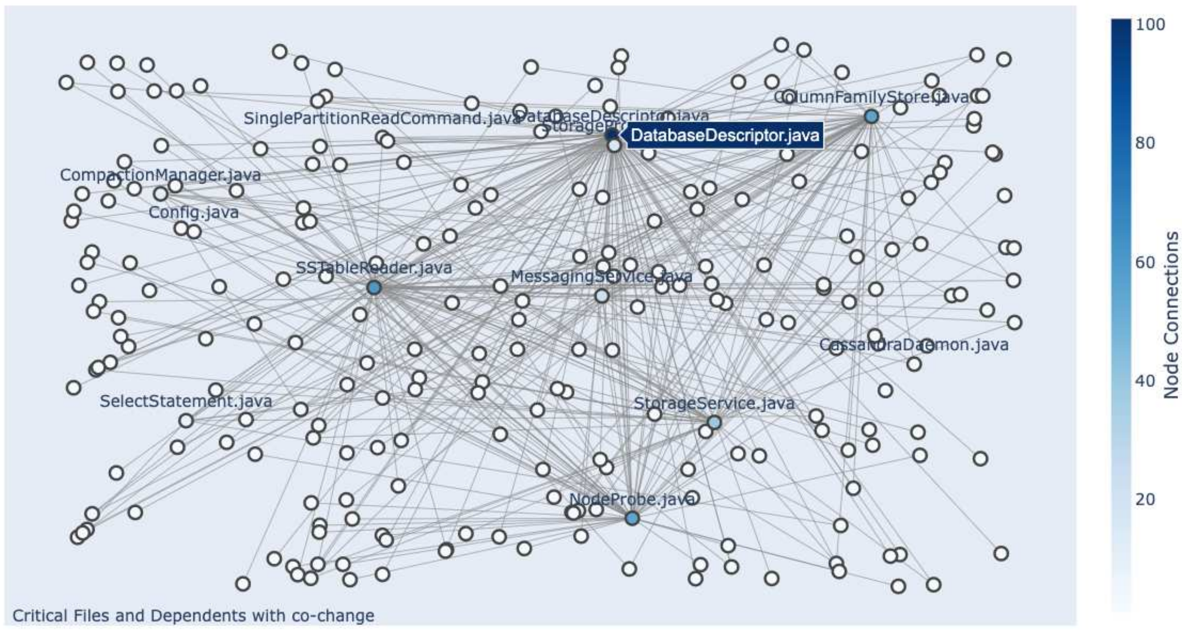


Figure 28 – Graph of critical classes and impacted files in Cassandra Repository

the Columnfamily.java file because of the metrics that we are using here. We can check that the approach is consistent. However, that class maybe has something different that should have more characteristics to be investigated to indicate ATD.". Our approach allows showing files with ATD characteristics that were not evidenced by other tools. A developer (Dev1) confirmed this: "*... precisely yeah, I think that's what I would take away from this tool, right that would be the thing that I believe is valuable, right, like it's showing me something that I didn't suspect, so to speak.*"

8.4.2 *RQ2 - How effective and useful is the proposed solution to identify the code artifacts that indicate the presence of ATD?*

We apply our automated process to a Cassandra Project code repository and identify a set of critical classes (source code files affected by ATD) and other files impacted by these critical classes that generate recurring maintenance efforts. Based on these results, we presented this set of critical files to a group of experienced developers maintaining the Cassandra project and asked them to compare them with the tools and methods used to identify source code indicating ATD. Then, we show to interviewed how the observed results can help practitioners select source code files that have more impact on change propagation related to architectural problems and recurrent efforts to maintain these source code files.

Source code files that depend on critical classes (found by our approach) may change if the critical classes are changed, as critical classes have a CD and HLD, and such architectural smells are strong indicators of change propagation. This is also an example of high cohesion showing strongly (or cyclically) connected components. Hence, if the critical classes change, the other dependent classes can be changed.

Source code files that change along joined the critical classes may have changes caused by the critical classes. In this case, source code files that simultaneously depend on critical classes and undergo changes from critical classes may indicate an ATD accumulation because this can compromise the architectural level of the system (LI *et al.*, 2014). As we can see in Table 14, the 12 critical classes impact 251 other files, i.e., there are 251 files that change together with the 12 critical files during the analyzed commit range shown in Figure 28.

We selected 1408 .java core implementation files (from scr/java/org/apache/cassandra/* directory - excluded test classes) within the range of analyzed commits from the Apache Cassandra repository. Twelve critical classes (.java files) impact another 251 classes (.java files). Thus, these critical classes can impact 17,8% of the implementation of system files under analysis. That is, 17,8% (251/1408) source code files indicate recurrent effort in this set of classes identified by our approach.

We also show the findings related to the impact of maintainability and effort (using code churn) to change critical files and dependent source code files that indicate ATD. Next, we show to interviewed how the observed results can help practitioners make better decisions about the set of candidate files to refactoring to pay ATD or even decrease ATD in the system, and we have positive feedback that the proposed approach can help to select files or sets of files for refactoring that can pay ATD or decrease ATD.

We presented these results to the developers, and we asked: "*Imagine we create a tool that will first show this slot of critical files and then the list of files that are in a key one area. Then we enable you to click in a critical file (for example, StorageService.java), and then it will* *show this one impacts hardly the other 37 files. Would this tool help you to debug or help you to think in a different way when you need to make a change to those files?*" This was confirmed by Developer (Dev1): "*I think that we can use a case when you want to do a major refactoring of the code base; then maybe you want to know what classes belong together like for example, every time you do x, y, and z, in critical classes, you have to make a change in these twelve classes right, and then you can kind of understand if we break these apart then maybe we can make the code a little easier so to speak.*". Hence, the tool could be helpful in help what files or sets of files should be a candidate for refactoring to decrease the ATD.

We also showed the analyzed change effort maintainability related to change LOC over time. In our target repository, the system has 1.926 classes (from scr/java/org/apache/cassandra/* directory - implementation classes and Tests classes) that have changed 310.431 modified LOCs within the analyzed commit range. When we consider only the core implementation files (removing test classes), we have 1.408 files that have changed 188.439 modified LOCs in the analyzed commit range.

Our approach selected 12 critical classes and their 251 impacted classes, so, there are 263 (source code artifacts that constantly change together over time and have a recurring effort to be maintained together) source code files that can indicate ATD and impact on change of these source code file is about of 60.885 modified LOCs in the system within the analyzed commit range.

The estimated effort spent by the 263 source code files (12 critical classes and their 251 impacted classes) indicated ATD, and that was 60.885 modified LOCs caused by these files. The total effort of all implementation classes was 310.431 LOCs modified in the system within the analyzed interval. Thus, the percentage of effort (ATD) spent on changing LOCs was 19,61% of the total LOCs changed within the analyzed range. Furthermore, if we consider only the main implementation files (removing the test files), we have a modification effort of 188.439 LOCs within the analyzed range. With this, we will have an increase in the LOC modification effort representing a percentage of 32,31% of LOCs modified caused by the 263 source code files that indicate ATD. Identifying these dependencies helps the development team and software architect manage the change's impact and simplify code maintenance.

Considering all Cassandra implementation source code files (.java) within the analyzed range of commits, we found that the effort spent on ATD represents about 19,61% of the effort spent modifying LOCs. Furthermore, if we consider only the system's core implementation files (.java), removing the test files, we find that the effort spent on ATD represents about 32,31% of the effort to modify LOCs. With this, we can observe a high effort of ATD within the range of commits analyzed.

Then, we asked if the developers know what files cause more change impact in other files and if the percentage of these impact in the maintenance of the system is high, then the developers could use this set of files to make decisions about refactoring that can help to pay or decrease ATD in the system. It was confirmed by Dev1: "*For large refactorings in major releases, it is important to know the magnitude of effort required for changes in source code in the release maintenance.*". Also, another developer realizes that a high impact of LOC changes in a set of files recurrently can cause a high impact on system maintainability over time. The Developer (Dev2) confirmed this: "*I think that it is important to know what source code files generate a recurring effort to be maintained because files that cause much impact from changes in other files; if these impacts are related to structural issues and software modularity, then it may require a greater maintenance effort.*"

8.5 Discussion

This section presents discussions for the results of our study. We try to extend our views by including findings from the literature and comparing them with the situation in the studied case.

8.5.1 Overall Discussion and Implications

For **RQ1**. Source code artifacts that indicate the presence of ATD thought AS seen to be widespread in this studied case. These findings complement the work about the relationship between AS and source code change (SAS *et al.*, 2022a) because we proposed a systematic approach to identify the source code artifacts related to ATD Items. Some tools show ATD indicators, as we saw in (FONTANA *et al.*, 2016b), (LUDWIG *et al.*, 2017) and (SHARMA *et al.*, 2020). However, such tools do not automatically extract source code files affected by ATD and other files affected by these artifacts. *Practitioners* should be interested in this outcome. In particular, they should expect an automatic way to identify source code ATD items using only source code analysis from repositories under VCS.

We found other works that use a mix of methods to identify ATD items. Martini *et* *al.* (2018a) used architectural smells to identify and prioritize ATDs. Kazman *et al.* (2015b) used source code analysis, issue tracker, file dependencies, and Design Structure Matrix to define the architecture root. Martini et al. (MARTINI *et al.*, 2018b) used source code analysis to calculate cohesion, coupling, and complexity metrics to identify modularity among components to define a unit of complexity measures of files. Li *et al.* (2014) considered modularity metrics to check architectural compliance issues, Li *et al.* (2015) used architectural decisions during project design based on change scenarios. Besides, Some frameworks were created to identify ATD, (MARTINI *et al.*, 2018b) and (ROVEDA *et al.*, 2018). However, they use custom formulas that depend on the context of the software system to find the artifacts affected by ATD. We also observed that other works ((MARTINI *et al.*, 2018a),(TOLEDO *et al.*, 2021a)) performed case studies focused on identifying ATD causes and effects of ATD in the software development process, however, they used qualitative methods such as surveys, questionaries, and interviews. For *researchers*, the identification of ATD items using only source code from software repository under VCS should be further investigated because there is not yet a consensus on it and there is a lack of experiments about that.

For **RQ2**. It is necessary to carry out a study or evaluation of existing solutions in the literature to identify the code artifacts affected by ATD. This may involve using solutions that need to analyze sample codebases and compare the results with manual code reviews or other methods of identifying technical debt. However, it is essential to establish an automatic process to monitor the source code change over time because when we find a source code artifact that indicates the presence of ATD is necessary to identify the other source code files that depend on it. An automatic process is important to help the developers and software architects to plan refactoring and the impact of the payment of ATD items. Our findings show that the critical source code files that indicate ATD are responsible for frequent changes in about 17,8% of the other source code files of the analyzed project. In addition, we can observe that the changes made (changes in LOCs throughout the analyzed range) in the set of critical files and their files impacted by ATD represent about 32,31% of the LOCs modification effort of all implementation files of the analyzed project. The findings from the semi-structured interview with experienced developers also showed that the proposed solution was well-received and considered useful in practice. These results suggest the proposed approach can provide valuable information for managing ATD in software systems.

We found that some works use a mix of artifacts and methods to monitor ATD items and their impacts. Perez *et al.* (2019) worked on release plan using several different artifacts at architectural-level without considering source code. Martini *et al.* (2018b) analyzed each release version calculating the modularity among the files changed in the release. Feng *et al.* (2019) monitored the evolution of hotspots during the period of a study by comparing the revision history and issue tracker to detect problematic files. Li *et al.* (2014) proposed to use software modularity metrics that can be directly calculated based on source code indicate ATD and suggested two modularity metrics that can be used as alternative ATD indicators of ANMCC (Average Number of Modified Components per Commit). As we can see, monitoring the impact of ATD items should be further investigated because there is a lack of studies that use only source code files from software repositories under VCS.

Besides, it is critical to know ways of the calculus of effort to fix ATD, and we found in literature that the main way to do this is the use of customized formulas to calculate effort based on time and the use of customized formulas to calculate effort in a release. These indicate that the use of formulas and expert evaluation are the main methods to do it. We found existing studies using estimated time or money as a proxy to indicate the effort needed to fix ATD problems (AMPATZOGLOU *et al.*, 2015), (DIGKAS *et al.*, 2018b), (OSPINA *et al.*, 2021). Curtis *et al.* (2012) used a repository of projects that register the software's complexity and register a set of architectural rules based on specific languages and technologies to calculate the effort to fix the ATD based on an aggregated formula that sums the effort involved in technical debts. The final results are defined in terms of hours and multiple by U$75 dollars per hour to find the total effort to pay the technical debt items. Martini e Bosch (2016b) used a particular set of formulas created by the AnaConDebt framework related to principal and interest to calculate the effort to fix the ATD Items identified.

For *researchers*, identifying and calculating the effort to fix ATD items should be further investigated because there needs to be a consensus on defining a general method to calculate ATD items using only source code artifacts from software repository under VCS. For *practitioners*, only a few cases present details on how to calculate the ATD item in the industry, and there is a lack of tools to aid and automate this process. More industrial cases are needed to calculate and automate the estimation of effort to fix ATD items in the SDLC.

8.6 Validity Threats

The validity threats are discussed using the categories construct validity, internal, external, and reliability validity described by Runeson *et al.* (2012).

Regarding **construct validity**, we aim to identify the AS in files and to measure code metrics over time to check the persistence of problems about the architectural level that can indicate ATD. We developed a detailed analysis repository process in a real-world using a well-known protocol template (RUNESON *et al.*, 2012) that was reviewed by the three authors in several iterations to ensure that the data to be collected would indeed be relevant to the research questions. In the analyzed commits range, even observing that the project was refactored periodically, we noticed that certain areas and files continued to increase their maintenance effort and continued impacting several other files at the architectural level. We also noticed that the AS persisted even after the refactoring which can reinforce the indication of ATD accumulation. Another threat is related to the choice of Arcan tool for the detection of the Architectural Smells considered in this project. This is partially mitigated as the Arcan tool has already been used and evaluated in several studies (FONTANA *et al.*, 2016), (BIAGGI *et al.*, 2018), (SAS *et al.*, 2021), (SAS *et al.*, 2022a).

In relation to **internal validity**, one limitation is that we were able to investigate only one software code repository to validate our hypothesis about ATD identification using only source code under version control. We have to investigate other software code repositories with similar characteristics to reproduce our proposed approach to check other software code repository results.

Regarding **external validity**, since we employed Design Science method, our findings are strongly bound by our research context. In addition, the investigated case involved only one software repository. To mitigate this threat, we described the context of our study in as much detail as possible so that the readers can identify if the context of our investigation is similar to theirs and reuse our findings whenever applicable.

Finally, about **reliability**, all data, scripts, and prototype of proposed solution are available on a replication kit available online. Also, the tools used in this study are free, allowing other researchers to assess the study's rigor or replicate the results using the same repository or even on different git repositories. The analysis was also performed using well-established techniques already used in previous works to analyze similar artifacts (extracting code smells and metrics via mining software repository techniques).

8.7 Conclusions

This chapter reports the results of a systematic approach applied in the Apache Cassandra repository, in collaboration with Ericcson (a European Telecom Company) Developers and Software Architects, carried out with quantitative data collected and analyzed in more than 5.000 commits (from 2015 to 2021)

To collect the quantitative data, we used a tool called Arcan to mine the Architectural Smells (AS). We selected the Cycle Dependency (CD) and Hub-like Dependency (HLD) as they are the most complex Architectural Smells and cause the most significant impact of changes in their dependent files. Also, we use the accumulation of modified lines (AMLOC), the frequency of commits in each file (FOC), and the cyclomatic complexity (CC) to select critical files in the system. We then use different techniques to study the impact of evolving and maintaining selected critical files and their dependent files as they evolve over time.

The overall conclusion is that source files with CD and HLD, which are frequently modified and show increasing growth in LOCs and CC over time, can indicate the presence of ATD. In addition, we have also found that these source files tend to affect many other dependent source files that have frequent changes over time. Thus, our findings suggest that it is possible to use only information from source code artifacts, under version control system, to identify the presence of ATD systematically. Also, we performed a semi-structured interview with software practitioners experienced from four Company developers that collaborated with Apache Cassandra Project, and they agreed with the results.

Our investigation has some implications for both researchers and practitioners. Our study is fully replicable by researchers. Raw data is also available online in the replication kit, allowing researchers to use such data in their approaches and facilitating the comparison of results. Regarding practitioners, we provide an automated approach based only on the source code repository analysis that can help evaluate critical files that impact software architecture that cause ATD. This approach can be reproducible, providing useful information about source code files that cause expensive effort maintenance related to the system's architecture.

Finally, we plan to continue investigating other software code repositories with similar characteristics to reproduce our proposed approach to check other software code repository results to strengthen the empirical evidence reported herein. Besides, we want to investigate the behavior of self-admitted technical debt saved directly in the source code files related to architectural problems.

9 EVALUATING THE ATDCODEANALYZER METHOD

In this chapter, we illustrate the evaluation process of the proposed method, verifying the efficacy of the ATDCodeAnalyzer method concerning Self-Admitted Technical Debt (SATD) and issues associated with architectural problems. To demonstrate and evaluate our approach, we conducted an empirical study involving four real-world applications. Additionally, we designed an Inspection Labeling Process, using a LLM (Large Language Models) based on ChatGPT, to categorize issues related to architectural problems, simplifying the identification of issues potentially causing architectural impacts.

9.1 Introduction

Due to the difficulty of recruiting developers with expertise in the evaluated systems, we opted for a qualitative analysis based on examining issue summary, descriptions and comments within the issue tracker of the evaluated repositories. Issue tracker systems contain valuable information about tasks related to new features, improvements, and bug fixes. Therefore, we analyzed issues that potentially impact the software architecture, followed by an examination of commits and critical classes identified by our ATD identification method.

The qualitative evaluation method employed here places a strong emphasis on quality, but we use quantitative metrics to check the performance of the evaluation method. In essence, its purpose is to assess the effectiveness and accuracy of the ATDCodeAnalyzer by evaluating the quality of its results of inspections concerning issues associated with critical files and those that exhibit architectural impacts.

The ATDCodeAnalyzer is a code analysis approach tailored to detect instances of Architectural Technical Debt (ATD) in a codebase. This entails the identification of code impacted by architectural problems. The utilization of SATD pertains to developers openly acknowledging the presence of technical debt within the code. SATD comments are commonly found in code or commit messages, providing insights into areas of the codebase requiring potential enhancements or refactoring (MALDONADO; SHIHAB, 2015). Lastly, issues related to architectural problems correspond to a evaluation process focused on pinpointing challenges associated with the architectural aspects of a software system, highlighting areas within the codebase where architectural issues exist.

This evaluation process describes a methodology for analyzing architectural issues in a Git repository, identifying critical classes impacted by ATD and correlating them with SATD in commits and issues from issue tracker related to architectural problems.

We want to answer the following research question: ***To what extent do issues associated with commits affecting critical classes demonstrate a relationship with (Self-Admitted) Technical Debt (TD/ATD)?*** Understanding the extent to which issues related to commits affecting critical classes correlate with (Self-Admitted) Technical Debt (TD/ATD) is crucial for assessing the effectiveness of the ATDCodeAnalyzer method. This research question is significant because it addresses the practical application of the ATDCodeAnalyzer and its relevance in identifying ATD in software projects.

To address this question, we consider the following steps:

Identifying Critical Classes: We utilize the ATDCodeAnalyzer to extract the classes that are impacted by architectural technical debt from the analyzed repository.

Analyzing Commit History: We examine the commit history of the software repository to identify commits affected by critical classes. This involves analyzing commit messages, code changes in modified files, and associated issues.

Assessing Technical Debt: We analyze the self-admitted technical debt associated with the software project, considering commit messages, code comments in commit diffs, and issues.

Identifying Correlations: We investigate any correlations between the issues associated with commits affecting critical classes and the presence of technical debt. This may involve statistical analysis related to issues with architectural impact and commits with SATD in messages and code comments.

9.2 Approach and Methodology

As previously outlined, the ATDCodeAnalyzer approach introduced in this thesis aims to automatically identify source code artifacts affected by ATD. This approach distinguishes itself from previous methods by operating autonomously, obviating the requirement for expert analysis in software architecture within the context of the analyzed repository. Therefore, following the application of the ATDCodeAnalyzer, our objective is to determine the extent to which the identified critical classes are genuinely influenced by architectural issues and technical debt. To achieve this, we have devised a qualitative evaluation method, based on inspection of issue tracker with architectural impact, to assess the efficacy of our proposed approach.

As can be seen in Figure 29, our process evaluation comprises eight phases.

A - *Software Artifacts Collection and ATDCodeAnalyzer*: The process begins by extracting information from the repository and executing the ATDCodeAnalyzer to identify Critical Classes based on our proposed approach.

B - *Commit Analysis*: The modified files are analyzed based on the range and select only the commits that have critical classes identified by ATDCodeAnalyzer.

C - *Issue Tracker Analysis*: This phase is related to the process of extracting information from all issues from the issue tracker, recording information about fields related to each issue in the analyzed repository.

D - *Correlation Metrics*: This phase calculates metrics based on the data collected in ATDCodeAnalyzer and Issue Tracker Analysis.

E - *SATD Analysis*: This phase is related to extracting SATD keywords from commits and the issue tracker in the repository. In this phase, it selects commits that have SATD keywords in messages and the diff of modified files, and issues that have SATD keywords in the summary, description, and comments.

F - *Select Issues for Inspection*: During this phase, issues are selected for manual inspection to check if these issues have architectural problems.

G - *Inspection Aided by ChatGPT*: In this phase, We have developed a semi-automatic inspection model based on ChatGPT, to aid in the analysis of each issue to determine if the issue has architectural problems and save the justification for each inspection.

H - *Issues with Architectural Problems*: Finally, in this last phase, we calculate the percentage of issues that do or do not have architectural problems to validate if the Critical Classes identified by ATDCodeAnalyzer have architectural issues.

More details about each phase are provided in the rest of this chapter.

9.2.1 Software Artifacts Collection and ATDCodeAnalyzer

The data for this study were obtained from four Apache Foundation's repositories popular open-source Java projects hosted on GitHub that are widely used in industrial distributed software systems. As the Apache Foundation's repositories follows best practices in the software development cycle, in addition, they follow issue registration standards, bug fix standards and release monitoring standards. We selected a representative dataset of Git repositories, encom-

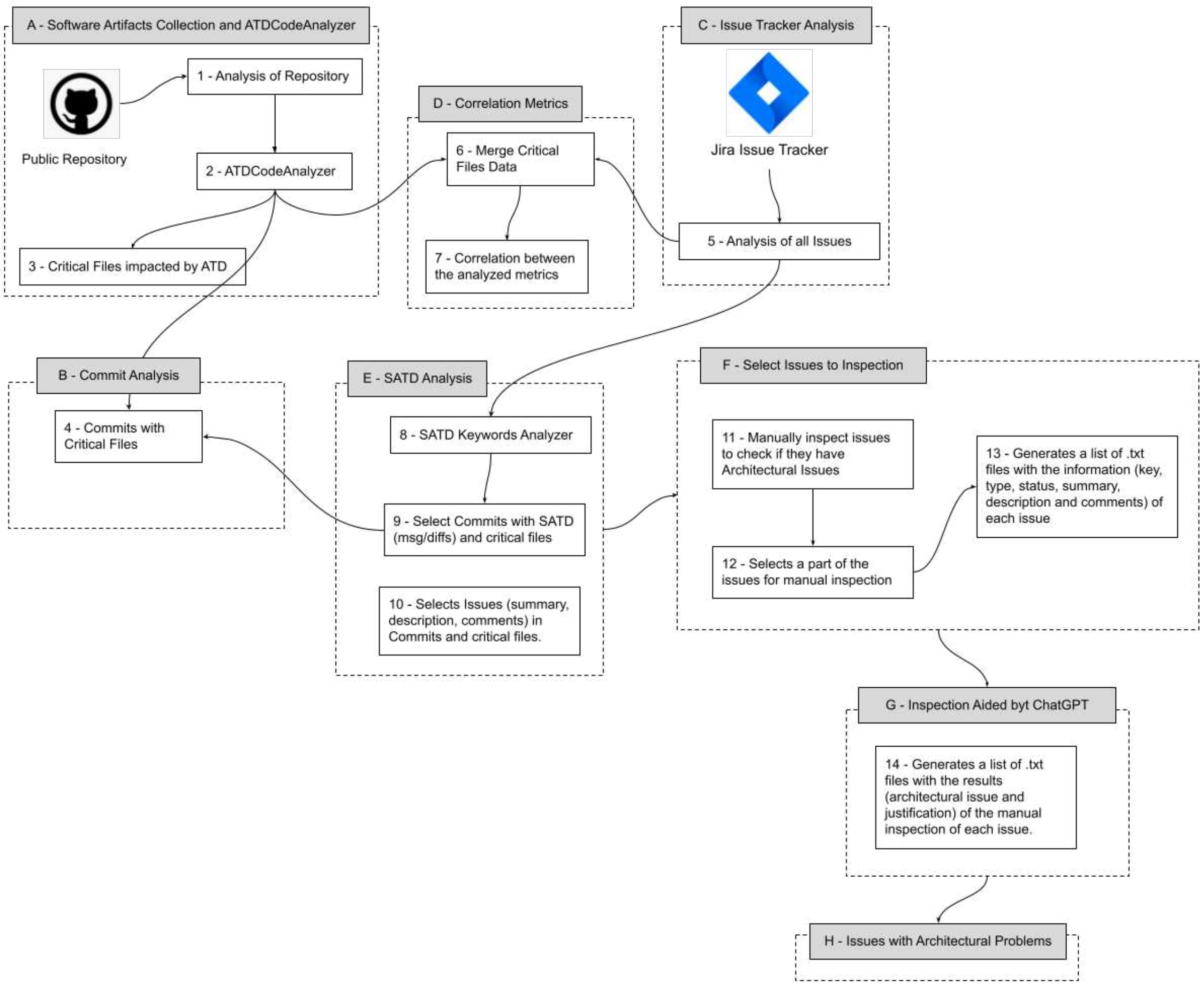

A - Software Artifacts Collection and ATDCodeAnalyzer
Public Repository
1 - Analysis of Repository
2 - ATDCodeAnalyzer
3 - Critical Files impacted by ATD
B - Commit Analysis
4 - Commits with Critical Files
C - Issue Tracker Analysis
Jira Issue Tracker
5 - Analysis of all Issues
D - Correlation Metrics
6 - Merge Critical Files Data
7 - Correlation between the analyzed metrics
E - SATD Analysis
8 - SATD Keywords Analyzer
9 - Select Commits with SATD (msg/diffs) and critical files
10 - Selects Issues (summary, description, comments) in Commits and critical files.
F - Select Issues to Inspection
11 - Manually inspect issues to check if they have Architectural Issues
12 - Selects a part of the issues for manual inspection
13 - Generates a list of .txt files with the information (key, type, status, summary, description and comments) of each issue
G - Inspection Aided byt ChatGPT
14 - Generates a list of .txt files with the results (architectural issue and justification) of the manual inspection of each issue.
H - Issues with Architectural Problems

Figure 29 – ATDCodeAnalyzer Evaluation

passing significant source code repositories such as Apache Cassandra[1], Apache Kafka[2], Apache Hadoop[3] and Apache ActiveMQ[4]. These software projects exhibit complex software architectures, boasting many attributes, including more than 10000 commits, over 100 collaborators, over 1000 Java files, over 100000 lines of code, and at least ten-year lifetime. Consequently, these repositories are characterized by substantial commit histories, many contributors, intricate codebases, and widespread adoption within the software industry. Each repository's numerical and metric attributes will be comprehensively detailed. We applied the ATDCodeAnalyzer method to these repositories to identify critical classes impacted by ATD. In the first moment, we choose the Apache Cassandra repository (RP1) to exemplify the process evaluation with more details.

[1] https://cassandra.apache.
[2] https://kafka.apache.org
[3] https://hadoop.apache.org
[4] https://activemq.apache.org

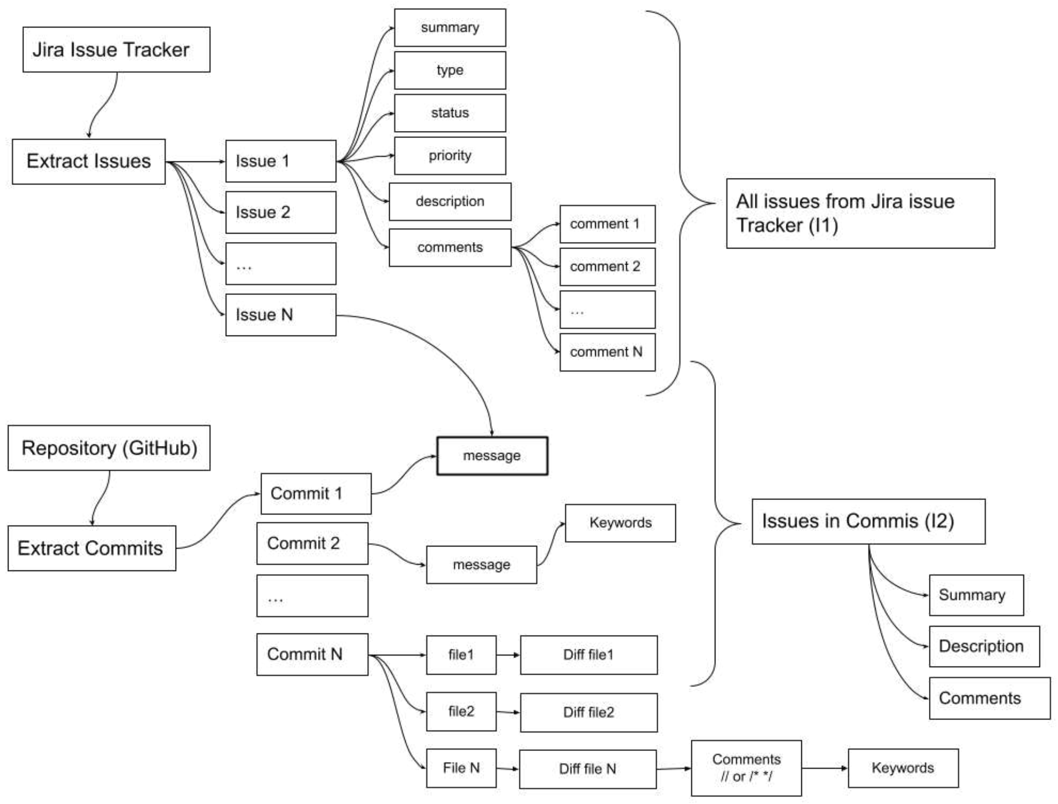


Figure 30 – Relationship between Commits and Issues

We analyzed the Github repository of Apache Cassandra [5], Apache Kafka[6], Apache ActiveMQ[7] and Apache Hadoop[8]. You can see more details about all repositories in Table 15. You can see more details about Apache Cassandra analysis in Figure 31. Here, we will conduct a thorough analysis of the Apache Cassandra repository to evaluate the effectiveness of our proposed method by analyzing relevant commits and issues. You can see the relationship between commits and issues in Figure 30. Furthermore, we conducted a comprehensive manual inspection process, complemented by the development of a semi-automatic inspection system aided by ChatGPT using prompt engineering. This system facilitated an issue labeling process focusing on architectural impact. The remaining repositories will be analyzed later.

To extract detailed information about each repository, we utilized the Cloc[9] application to extract essential information, including lines of code (LOC), file count, Java file count, and comment count in the last commit of 2023/10/04. Further details can be found in the replication kit.

We applied the ATDCodeAnalyzer method to extract the files (critical classes) that are

[5] http://github.com/apache/cassandra.git
[6] https://github.com/apache/kafka.git
[7] https://github.com/apache/activemq.git
[8] https://github.com/apache/hadoop.git
[9] https://github.com/AlDanial/cloc

Table 15 – Details of Analyzed Repositories.

id	Property	RP1	RP2	RP3	RP4
1	name	Cassandra	Kafka	ActiveMQ	Hadoop
2	files	4998	5648	5174	14970
3	LOC files	1055561	874543	466706	4323485
4	java files	4459	4350	4367	11811
5	LOC java files	680827	649612	417291	1889967
6	Java comments	166567	175655	150220	613888
7	QTY commits	29142	11810	11537	26945
8	QTY issues	18635	14326	5955	12247
9	QTY releases	297	65	85	374
10	Life span (years)	14.53	12	14	14
11	stars	8200	26100	2200	13900
12	forks	3500	13000	1400	8600
13	colaborators	426	1062	130	552
14	first commit	2009/03/02	2011/08/01	2009/01/01	2009/05/19
15	last commit	2023/10/04	2023/10/04	2023/10/04	2023/10/04

impacted by ATD in the Apache Cassandra repository, and we obtained the following critical files: *[“StorageService.java”, “ColumnFamilyStore.java”, “DatabaseDescriptor.java”, “CompactionManager.java”, “StorageProxy.java”, “SSTableReader.java”, “Config.java”, “CassandraDaemon.java”, “SelectStatement.java”, “SinglePartitionReadCommand.java”, “NodeProbe.java”, “MessagingService.java”]*

All steps and details about how to extract critical classes from Apache Cassandra using ATDCodeAnlyzer are available in the replication package[10].

9.2.2 Commit Analysis

In this step, we filtered only commits, from Apache Cassandra, that contain at least one critical class, resulting in 4522 commits, as you can see in the Figure 32.

9.2.3 Issue Tracker Analysis

We analyzed 18635 issues spanning nearly 14 years, from March 2, 2009, to October 4, 2023, extracted from the Jira issue tracker[11] of the Apache Cassandra project.

[10] https://github.com/mining-software-repositories/cassandra
[11] https://issues.apache.org/jira/projects/CASSANDRA

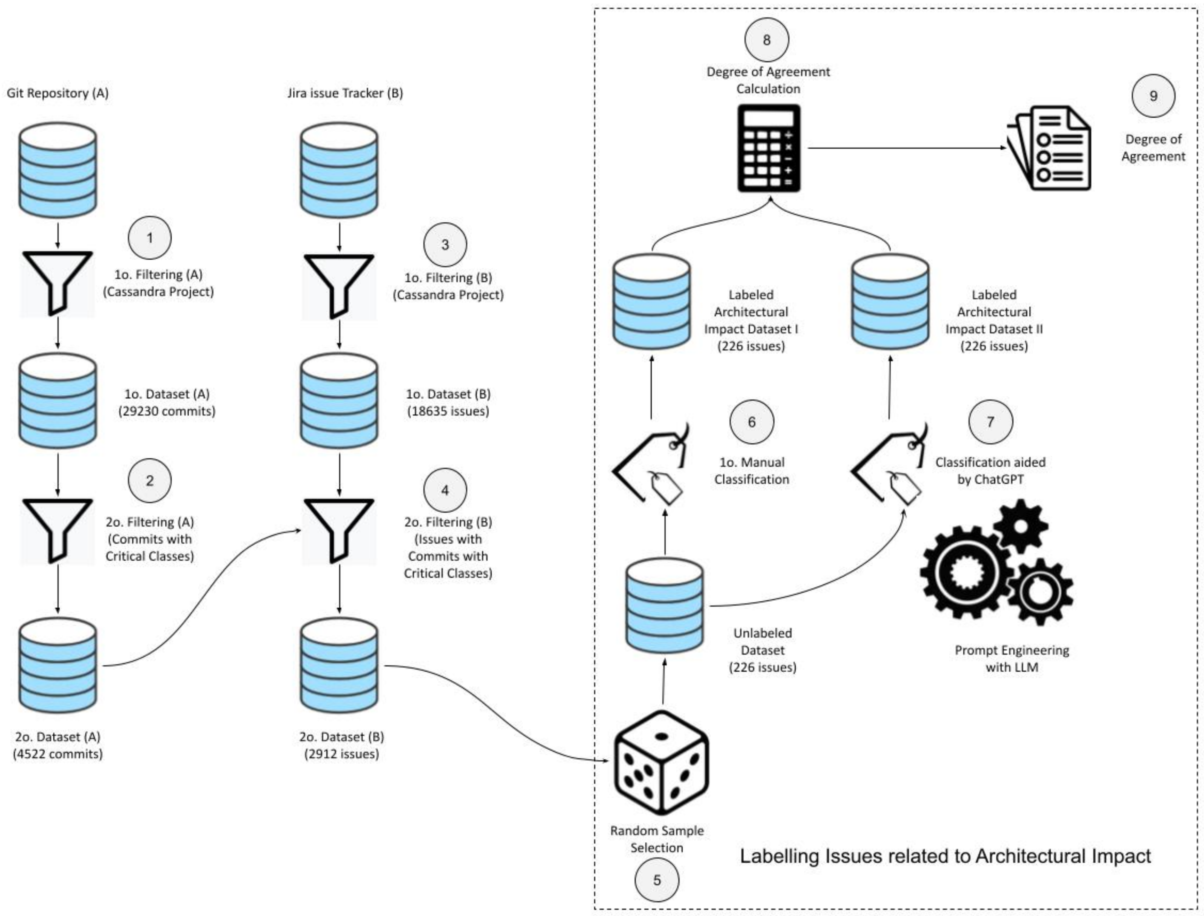

Git Repository (A)
Jira issue Tracker (B)
1
1o. Filtering (A) (Cassandra Project)
1o. Dataset (A) (29230 commits)
2
2o. Filtering (A) (Commits with Critical Classes)
2o. Dataset (A) (4522 commits)
3
1o. Filtering (B) (Cassandra Project)
1o. Dataset (B) (18635 issues)
4
2o. Filtering (B) (Issues with Commits with Critical Classes)
2o. Dataset (B) (2912 issues)
5
Random Sample Selection
Unlabeled Dataset (226 issues)
6
1o. Manual Classification
7
Classification aided by ChatGPT
Prompt Engineering with LLM
Labeled Architectural Impact Dataset I (226 issues)
Labeled Architectural Impact Dataset II (226 issues)
8
Degree of Agreement Calculation
9
Degree of Agreement
Labelling Issues related to Architectural Impact

Figure 31 – Evaluating the ATDCodeAnalyzer in Apache Cassandra

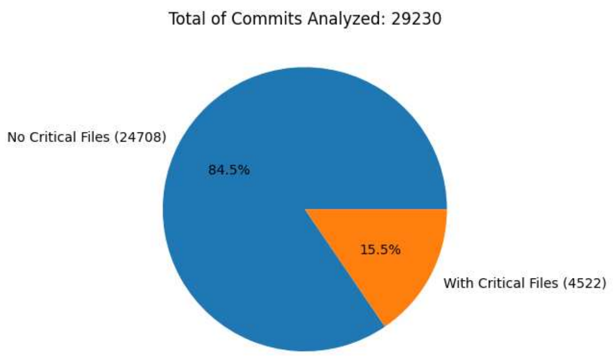


Figure 32 – Analyzed Commits from Apache Cassandra repository

Table 16 – Summary of AIC (Appears in Commits), AICWI (Appears in Commits with Issues), AII (Appears in Issues), AIB (Appears in Bugs), AIII (Appears in Improvement Issues), and AINFI (Appears in New Feature Issues) for Critical Java Files identified by ATDCodeAnalyzer

File	AIC	AICWI	AII	AIB	AIII	AINFI
StorageService.java	1317	1034	967.0	160.0	79	3
ColumnFamilyStore.java	1178	893	832.0	139.0	96	3
DatabaseDescriptor.java	854	697	659.0	90.0	62	4
StorageProxy.java	572	445	463.0	71.0	55	2
CompactionManager.java	668	506	406.0	72.0	47	5
Config.java	500	393	379.0	58.0	37	3
SSTableReader.java	457	399	352.0	47.0	48	1
MessagingService.java	328	241	288.0	50.0	31	1
NodeProbe.java	369	303	287.0	39.0	26	2
SelectStatement.java	93	85	245.0	39.0	25	0
CassandraDaemon.java	386	316	235.0	28.0	34	5
SinglePartitionReadCommand.java	419	309	83.0	5.0	5	1

9.2.4 Correlation Metrics

After merge the data metrics from commit analysis and issue analysis we get the Table 16 related to merge data metrics correlation to critical classes from Apache Cassandra.

The correlation of metrics for critical classes suggests that these files are more likely to appear frequently in commits associated with documented issues in commits (Spearman correlation=0.85), bug issues (Spearman correlation=0.84), and improvements issues (Spearman correlation=0.73). These correlation metrics are related to critical classes related to commits and critical classes to issues from Apache Cassandra, as you can see in the Figure 33.

9.2.5 SATD Analysis

We implemented a method to extract SATD keywords from commit messages and code comments from modified files in the commit. We applied this method to the repository to identify SATD-related commits and their associated keywords. The SATD analysis and the method are available in replication kit[12].

We extracted and processed the commit data, including commit messages and code diffs related to critical files selected by ATDCodeanalyzer. We identified which SATD-related commits involve changes to the critical files.

To select SATD keywords, we referred to works such as (RANTALA; MÄNTYLÄ,

[12] https://github.com/Technical-Debt-Large-Scale/my_validation

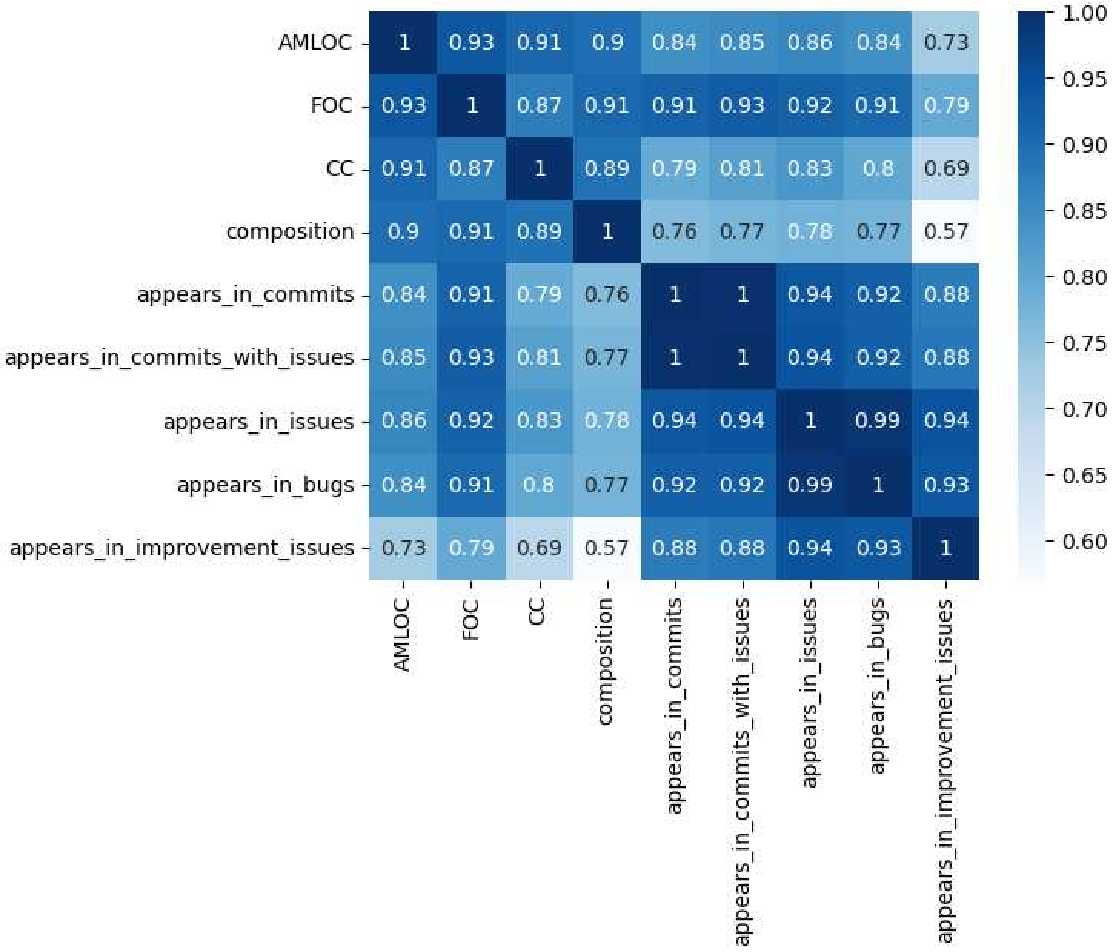


Figure 33 – Critical Files Metrics Correlation in Commits and Issues

2020), (LI *et al.*, 2023), and (POTDAR; SHIHAB, 2014). We compiled a set of keywords by unifying the SATD keywords identified in these studies.

The following 235 keywords were used to perform SATD keywords in issues and commits:

keywords = [‘API’, ‘FIXME’, ‘TODO’, ‘ability to evolve’, ‘ability to handle increased load’, ‘annotation’, ‘anti-pattern’, ‘any chance of a test’, ‘architectural debt’, ’architectural issue’, ’architectural problem’, ’architectural smell’, ’avoid calling it twice’, ’avoid extra seek’, ’bad practice’, ’brittle code’, ’buggy code’, ’by hard coding instead of’, ’cast’, ’checkstyle errors’, ’circular dependency’, ’clean’, ’clean up code’, ’cleanup’, ’code cleanup’, ’code complexity’, ’code debt’, ’code defect’, ’code dependencies’, ’code difficulty’, ’code duplication’, ’code entanglement’, ’code flaw’, ’code improvement’, ’code interdependencies’, ’code issue’, ’code problem’, ’code redundancy’, ’code restructuring’, ’code rot’, ’code simplification’, ’code smell’, ’cognitive complexity’, ’comment’, ’complex code’, ’complex code relationships’, ’complexity’, ’concrete code’, ’concurrency issue’, ’confusing’, ’constructor’, ’cross-module’, ’cyclic dependency’, ’cyclomatic complexity’, ’dead code’, ’debug’, ’delicate code’, ’dependability’, ’dependencies’, ’dependency’, ’deprecated code’, ’design’, ’design debt’, ’design defect’, ’design flaw’, ’design flaws’, ’design issue’, ’design problem’, ’design smell’, ’difficult to maintain code’, ’difficult to understand code’, ’disorganized code’, ’documentation’, ’documentation debt’, ’documentation does not mention’, "documentation doesn’t match", ’duplication’, ’ease of maintenance’, ’easy to break code’, ’encapsulation’, ’endpoints’, ’error message’, ’exception’, ’exposed internal state’, ’extension point’, ’fault tolerance’, ’files’, ’findbugs’, ’fix’, ’flaky’, ’flaky code’, ’formatting’, ’fragile code’, ’get rid of’, ’good to have coverage’, ’hack’, ’handling’, ’hard-coded strings’, ’hard-coded values’, ’header’, ’implementation’, ’implementation debt’, ’improvement’, ’inconsistency’, ’indirect dependency’, ’ineffective solution’, ’ineffective way’, ’inefficient solution’, ’inefficient way’, ’infinite loop’, ’inter-module’, ’interface’, "it’d be nice", "it’s not perfectly documented", ’javadoc’, ’lack of abstraction’, ’lack of code comments’, ’lack of cohesion’, ’lack of documentation’, ’lack of encapsulation’,

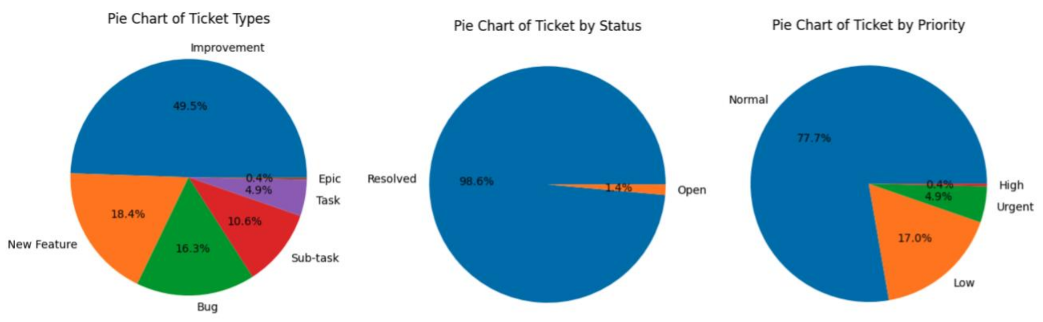


Figure 34 – Percentual of Issues that appear in commits with critical files

'lack of generalization', 'lack of information hiding', 'lack of modularity', 'lack of separation of concerns', 'lack of test cases', 'lack of testing', 'latency', 'lead to huge memory allocation', 'leak', 'less verbose', 'literal strings', 'literal values', 'logging', 'magic numbers', 'magic strings', 'maintainability', 'maintainability issue', 'make it less brittle', 'makes it much easier', 'makes it very hard', 'minor', 'misleading', 'modularity', 'module dependencies', 'module-to-module', 'monolithic code', 'more efficient', 'more readable', 'more robust', 'more tests', 'more tightly coupled than ideal', 'multithreading issue', 'naming', 'need to update documentation', 'no longer needed', 'not done yet', 'not implemented', 'not supported yet', 'not thread safe', 'not used', 'output', "patch doesn't apply cleanly", 'performance', 'please add a test', 'poor solution', 'poor way', 'poorly documented code', 'poorly structured code', 'poorly tested code', 'quick fix', 'race condition', 'reduce duplicate code', 'redundant', 'refact', 'refactor', 'refactoring', 'reliability', 'rename', 'repeated code', 'response time', 'robustness', 'scalability', 'scalability issue', 'short term solution', 'should be updated to reflect', 'should improve a bit by', 'simplify', "solution won't be really satisfactory", 'some holes in the doc', 'spaghetti code', 'speed', 'speed up', 'spurious error messages', 'suboptimal solution', 'suboptimal way', 'support for', 'synchronization issue', 'system dependencies', 'system design problem', 'takes a long time', 'technical debt', 'technical debt due to architectural issues', 'technical debt due to design issues', 'technical kludge', 'temporary solution', 'test', "test doesn't add much value", 'testing debt', 'there is no unit test', 'throughput', 'tidy up', 'tight coupling', 'too long', 'too much', 'trustworthiness', 'typo', 'ugly', 'undocumented code', 'undocumented strings', 'undocumented values', 'unnecessary', 'unreliable code', 'unstable', 'untested code', 'unused', 'unused code', 'unused import', 'update', 'violation', 'wastes a lot of space', 'work in progress', 'workaround', 'would significantly improve', 'wrong solution', 'wrong way']

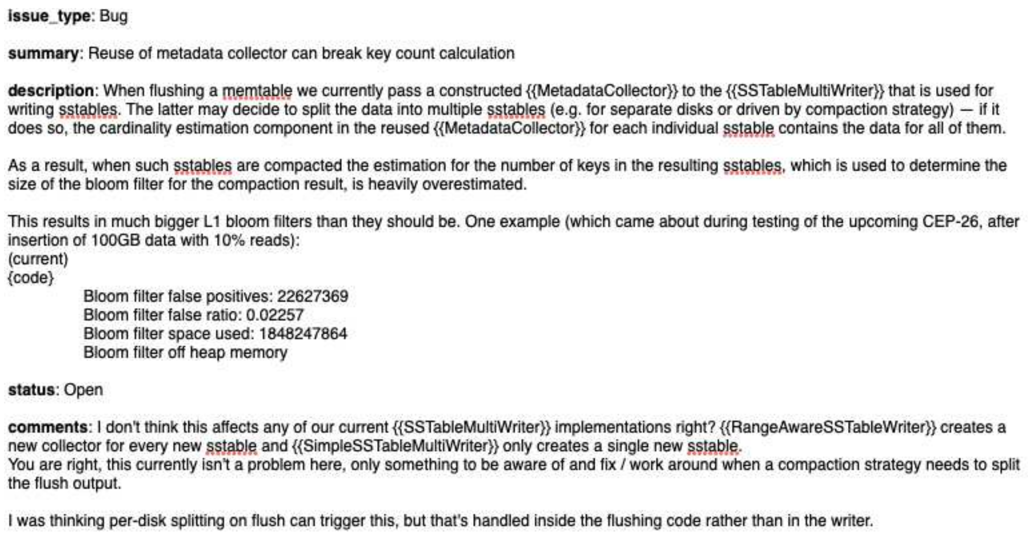
9.2.6 Select Issues for Inspection

For the commits involving critical classes, we extracted and analyzed the issues mentioned in the commit messages.

Calculating the Distribution of Issues: In this step, we delve into the data to calculate the percentage of issues based on their type, status, and priority as you can see in Figure 34. This statistical analysis provides a comprehensive view of the issue landscape, highlighting which categories are more prevalent and need closer examination. For example, 49.5% of issues that appear in commits with critical classes are improvement issues.

Manual Inspection for Architectural Issues: A critical step in our process involves a manual inspection of the selected issues. We developed a systematic way[13] using the Apache Cassandra Jira Issue Tracker to classify issues related to architectural impact. This method is designed to be extensible and reproducible for other Apache projects, allowing for identification of issues with architectural impact. This hands-on approach allows us to thoroughly assess

[13] https://docs.google.com/document/d/1umbEJMVsdxTzBVOr8VDRCscpwOK9-ePVJ-o862L5j08



issue_type: Bug

summary: Reuse of metadata collector can break key count calculation

description: When flushing a memtable we currently pass a constructed {{MetadataCollector}} to the {{SSTableMultiWriter}} that is used for writing sstables. The latter may decide to split the data into multiple sstables (e.g. for separate disks or driven by compaction strategy) — if it does so, the cardinality estimation component in the reused {{MetadataCollector}} for each individual sstable contains the data for all of them.

As a result, when such sstables are compacted the estimation for the number of keys in the resulting sstables, which is used to determine the size of the bloom filter for the compaction result, is heavily overestimated.

This results in much bigger L1 bloom filters than they should be. One example (which came about during testing of the upcoming CEP-26, after insertion of 100GB data with 10% reads):
(current)
{code}

```
        Bloom filter false positives: 22627369
        Bloom filter false ratio: 0.02257
        Bloom filter space used: 1848247864
        Bloom filter off heap memory
```

status: Open

comments: I don't think this affects any of our current {{SSTableMultiWriter}} implementations right? {{RangeAwareSSTableWriter}} creates a new collector for every new sstable and {{SimpleSSTableMultiWriter}} only creates a single new sstable.
You are right, this currently isn't a problem here, only something to be aware of and fix / work around when a compaction strategy needs to split the flush output.

I was thinking per-disk splitting on flush can trigger this, but that's handled inside the flushing code rather than in the writer.

Figure 35 – Detailed issue information

whether these issues exhibit architectural concerns. By scrutinizing each issue individually, we can identify elements that might affect the software's overall architectural quality.

Selective Issue Inspection: To manage the inspection process efficiently, we selectively choose a subset of issues for a closer look. We calculated sample size based on the method norm.ppf()[14] related to normal distribution random to take a percentage and returns a standard deviation multiplier for what value that percentage occurs (WITTE; WITTE, 2017).We used the following parameters: confidence level = 0.95, margin of error = 0.05, population proportion = 0.8 and population size = 2912 then resulting in 226 issues. This strategic selection ensures that we focus our resources where they matter most, optimizing the identification of architectural issues and reducing inspection time for less critical problems.

Generating Detailed Issue Files: As part of our meticulous analysis, we generate a set of 226 .txt files for each selected issue. These files contain key information, including issue type, status, summary, description, and comments. This structured approach streamlines further analysis and facilitates cross-referencing of data, enabling a more comprehensive understanding of the issues at hand. As you can see the Figure 35. You can find this set of issues in zip file my_issues in replication kit[15] related to Apache Cassandra.

9.2.7 Inspection Aided by ChatGPT

To expedite the issue inspection process, we explored utilizing an Artificial Intelligence (AI) powered approach. While we initially considered a generic large language model

[14] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html
[15] https://github.com/Technical-Debt-Large-Scale/my_validation/tree/main/cassandra

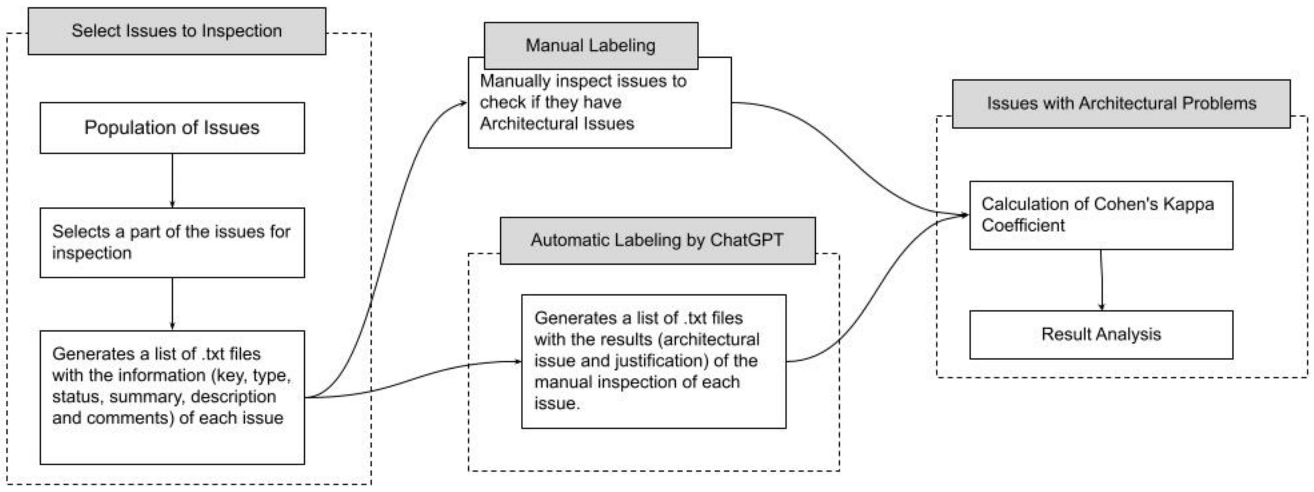

Select Issues to Inspection
Population of Issues
Selects a part of the issues for inspection
Generates a list of .txt files with the information (key, type, status, summary, description and comments) of each issue
Manual Labeling
Manually inspect issues to check if they have Architectural Issues
Automatic Labeling by ChatGPT
Generates a list of .txt files with the results (architectural issue and justification) of the manual inspection of each issue.
Issues with Architectural Problems
Calculation of Cohen's Kappa Coefficient
Result Analysis

Figure 36 – Issue labeling process

(LLM) like ChatGPT, we determined a more specialized model trained on our inspection for architectural issues would be more effective for identifying and classifying issues. We utilized the ChatGPT [16] tool version 3.5[17], implementing Few-shot and Chain-of-thought prompting strategies (BROWN *et al.*, 2020), (ZHOU *et al.*, 2022), to expedite the manual inspection of each issue and identify architectural problems. This decision to employ the tool was aimed at streamlining our inspection process.

We created a classifier of issues from the Apache Cassandra issue tracker. The classifier analyzes each issue and determines whether or not it has an architectural impact. The issue labeling process is related to check if the issue impacts or no impacts the software architecture of the analyzed repository.

We followed a structured approach to enhance the quality of issue labeling. In Stage 1, we focused on understanding Architectural Technical Debt (ATD), Self-Admitted Technical Debt (SATD), commits, and issues. Stage 2 involved showcasing sample issues and introducing the 'Architectural Impact Classifier.' Stage 3 included the selection of 226 issues associated with critical classes, followed by manual and automatic inspections in Stages 4 and 5. Lastly, Stage 6 concluded with the calculation of the Cohen's Kappa coefficient to assess the agreement between manual and automatic inspections, aiding in the analysis of critical issues' impact on software architecture. The issue labeling process involves several steps, as you can see in the Figure 36.

In the following subsections, we provide a detailed account of each stage. More

[16] https://chat.openai.com
[17] ChatGPT 3.5 was chosen because it is free and has no token limit during the immediate engineering process.

details about all steps of this inspection process is available in the replication kit[18].

9.2.7.1 Stage 1 - Introduction

We start with an introduction to Architectural Technical Debt (ATD) and Self-admitted Technical Debt (SATD), explaing the concepts, techniques, and types of ATD, providing examples. We discuss different issue types in Jira Cassandra and introduce Apache Cassandra. We also share examples of SATD keywords in commits, including their presence in messages and differences in modified files, as well as in Cassandra issue fields such as summary, description, and comments. You can see these prompts in a set of prompts S1P1..S1P4 available the inspection process in the evaluation replication kit.

9.2.7.2 Stage 2 - Example Showcase

We provide showcase a few sample issues (issues about task, bug-fix, improvement, and new feature) that have been converted into .txt files. These files contain issue-specific information, including issue ID, type, status, summary, description, and comments. We also mentioned that these issues have already undergone manual inspections without ChatGPT. You can see the following prompts: S2P1..S2P11 available the inspection process in the evaluation replication kit.

9.2.7.3 Stage 3 - Data Preparation

We generated individual txt files for each of the 226 issues for inspection. Each file should contain detailed issue information, including issue ID, type, status, summary, description, and comments. You can find this set of issues in zip file my_issues in replication kit[19] related to Apache Cassandra.

9.2.7.4 Stage 4 - Manual Inspection

The 226 issues are selected for manual inspection. We saved the results of these manual inspections in 226 separate .txt files to be evaluated in another stage.

[18] https://github.com/Technical-Debt-Large-Scale/my_validation/blob/main/inspection_process.md
[19] https://github.com/Technical-Debt-Large-Scale/my_validation/tree/main/cassandra

9.2.7.5 Stage 5 - ChatGTP Inspection

We utilized ChatGPT to inspect each of the 226 .txt files representing issues. The primary goal is to identify potential architectural impacts, architectural issues, and/or technical debt within the analyzed issues. Also, we requested justification for each inspection. Finally, we saved the results of these ChatGPT inspections in 226 separate .txt files to be evaluated in another stage. You can find this set of issues in the zip file my_results_inspection in the replication kit related to Apache Cassandra.

9.2.7.6 Stage 6 - Manual Review

Finally, in the last stage, we conducted a manual review of all 226 sets of inspection results (.txt files) both manual inspection and inspection by ChatGPT. Cohen's Kappa coefficient is calculated to compare the agreement between the results obtained in the manual inspection and inspection by ChatGPT. This helps assess the consistency between manual and automatic inspections, contributing to a precise analysis of the relationship between critical issues and their impact on the software's architecture.

9.2.8 Issues with Architectural Problems

We meticulously examined each issue to identify any potential architectural problems. Subsequently, we generated a comprehensive spreadsheet, indicating whether each analyzed issue exhibited architectural concerns with a simple "Yes" or "No".

We collect data on the number and severity of architectural issues found in the issues related to the critical files. We analyzed the data to determine if there is a statistically significant correlation between the presence of SATD-related commits and the presence of architectural issues in the critical files.

Finally, once we have developed an LLM-based model to assist in issue inspection, we can apply this model to accelerate the issue inspection process for other projects.

9.3 The Architectural Impact Issues Dataset Analysis

In this section, we will analyze our dataset to assess the relationship between architectural issues and commits with critical classes. We begin by establishing the analysis design, including its overall goal, research questions, analyzed projects, and methodology.

9.3.1 Goal and Research Questions

We applied the evaluation method to Apache ActiveMQ, Apache Kafka, and Apache Hadoop to assess the impact of critical files associated with architecturally-driven issues in these repositories. We examined if these issues indicate architectural problems within the software, the **goal** can be formulated as follows:

We analyzed the number of issues with architectural impact on the critical classes within the context of four open-source Java software systems.

Informed by "architectural impact issues" identified in commits involving critical classes across the selected software repositories, we formulated the following two **research questions**:

RQ1) What is the proportion of issues classified as impacting architectural design, as observed in each analyzed software repository?

This research question aims to understand what percentage of issues found in each analyzed software repository directly affect the software's overall architecture.

RQ2) Which classes are most commonly involved in issues that impact the overall architecture?

This research question investigates whether classes, in the analyzed projects, are more likely to have issues that affect the system's architecture.

9.3.2 Analysed Projects

In this experiment, we have been analysed four Git Repositories: Apache Cassandra, Apache ActiveMQ, Apache Kafka and Apache Hadoop. We evaluated 685 issues from all analyzed repositories, more precisely, Cassandra (226 issues), ActiveMQ (132 issues), Kafka (179 issues) and Hadoop (148 issues) using an inspection process aided by the LLM model to classify them as either Yes or No based on their architectural impact. More details about dataset collection of the analyzed projects can be seen in Table 17.

9.3.2.1 Evaluating the ATDCodeAnalyzer in Apache ActiveMQ

The process evaluation in Apache ActiveMQ is detailed in the Figure 37 as follows:

Table 17 – Dataset collection of evaluation method

Description	RP1	RP2	RP3	RP4	Total
Apache Project	Cassandra	Kafka	ActiveMQ	Hadoop	-
Data Analysis - start (s1s)	2009/02/02	2011/08/01	2009/08/01	2009/05/19	-
Data Analysis - final (s1f)	2023/10/04	2023/10/04	2023/10/04	2023/10/04	-
Extracted Commits (s2)	29230	11732	7941	26906	75809
Commits with Critical Classes (s3)	4522	1452	721	2776	9471
Extracted Issues (s4)	18635	14326	5955	12247	51163
Issues with critical classes (s5)	2912	939	480	239	4570
Issues inspected (s6)	226	179	132	148	685
Issues with Architectutural Impact (s7)	96	72	55	51	274
Issues without Architectutural Impact (s8)	130	107	77	97	411

First Filter (A) - ActiveMQ Project: In this step, we select and filter data related to the Apache ActiveMQ project from the GitHub repository. We focus on data that is relevant to architectural changes or issues.

Second Filter (A) - Commits with Critical Classes: After filtering the ActiveMQ project data, we identify commits that involve critical classes. We consider critical classes to be those that are related to architectural issues identified by ATDCodeAnalyzer.

First Filter (B) - ActiveMQ Project: In this step, we perform a similar filtering process on the Jira Issue Tracker. We specifically look for issues related to the ActiveMQ project.

Second Filter (B) - Issues with Commits with Critical Classes: In this step, we identify Jira issues that are associated with commits containing critical classes. This allows us to link code changes with corresponding issues.

In Apache ActiveMQ repository, and we obtained the following critical classes:

ActiveMQ critical classes=['DemandForwardingBridgeSupport.java', 'SubQueueSelectorCacheBroker.java', 'BrokerServiceAware.java', 'TransportConnector.java','BrokerService.java', 'TopicSubscription.java', 'QueueBrowserSubscription.java', 'Queue.java','DurableTopicSubscription.java', 'QueueSubscription.java', 'QueueDispatchSelector.java', 'PendingQueueMessageStoragePolicy.java', 'MirroredQueue.java','MessageQueue.java']

All steps and details about how to extract critical files from Apache ActiveMQ using ATDCodeAnlyzer are available in the replication package[20].

Semi-automatic classification aided by ChatGPT with Prompt Engineering: In this step, we use ChatGPT to assist us in the classification process. We provide prompts to ChatGPT to help automate or semi-automate some of the classification tasks.

Finally, we have a set of issues labeled with architectural impact and without architectural impact.

[20] https://github.com/Technical-Debt-Large-Scale/atdcodeanalyzer

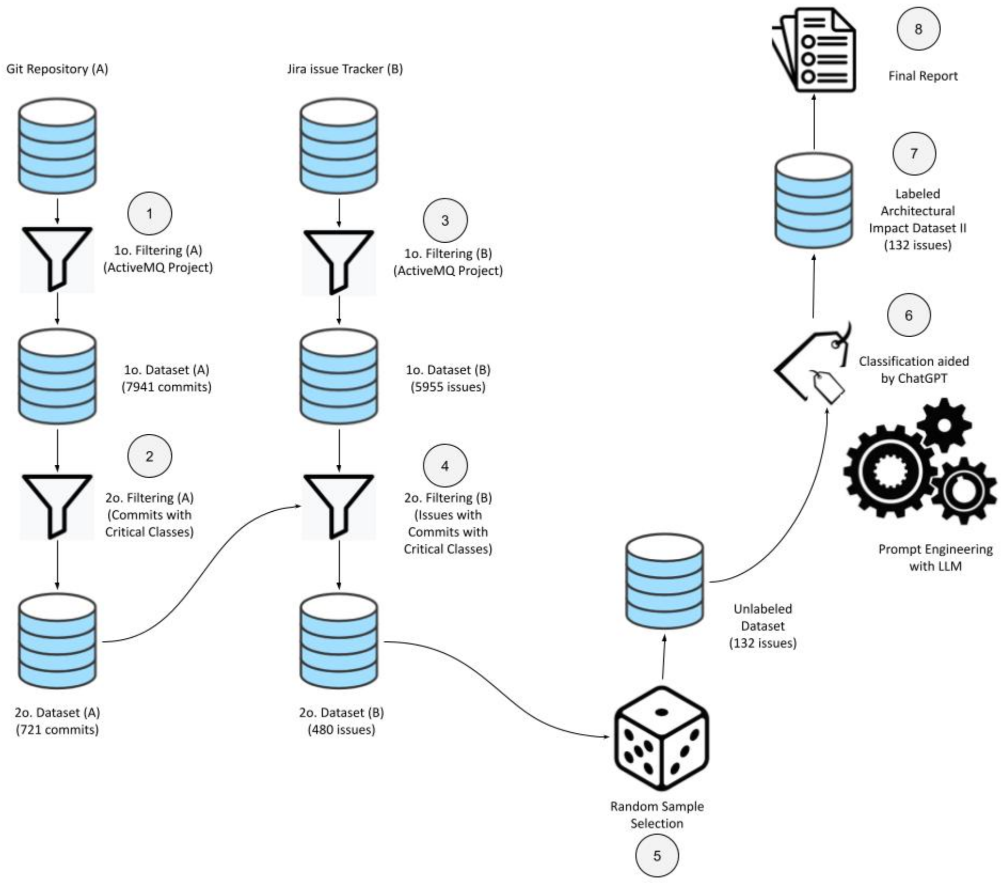

Git Repository (A)
Jira issue Tracker (B)
1
1o. Filtering (A) (ActiveMQ Project)
3
1o. Filtering (B) (ActiveMQ Project)
1o. Dataset (A) (7941 commits)
1o. Dataset (B) (5955 issues)
2
2o. Filtering (A) (Commits with Critical Classes)
4
2o. Filtering (B) (Issues with Commits with Critical Classes)
2o. Dataset (A) (721 commits)
2o. Dataset (B) (480 issues)
Random Sample Selection
5
Unlabeled Dataset (132 issues)
6
Classification aided by ChatGPT
Prompt Engineering with LLM
7
Labeled Architectural Impact Dataset II (132 issues)
8
Final Report

Figure 37 – Evaluating the ATDCodeAnalyzer in Apache ActiveMQ repository

9.3.2.2 Evaluating the ATDCodeAnalyzer in Apache Kafka

The process evaluation in Apache Kafka is detailed in the Figure 38.

First Filter (A) - Kafka Project: In this step, we select and filter data related to the Apache Kafka project from the GitHub repository. We focus on data that is relevant to architectural changes or issues.

Second Filter (A) - Commits with Critical Classes: After filtering the Kafka project data, we identify commits that involve critical classes. We consider critical classes to be those that are related to architectural issues identified by ATDCodeAnalyzer

First Filter (B) - Kafka Project: In this step, we perform a similar filtering process on the Jira Issue Tracker. We specifically look for issues related to the Kafka project.

Second Filter (B) - Issues with Commits with Critical Classes: In this step, we identify Jira issues that are associated with commits containing critical classes. This allows us to link code changes with corresponding issues.

In Apache Kafka repository, and we obtained the following critical files:

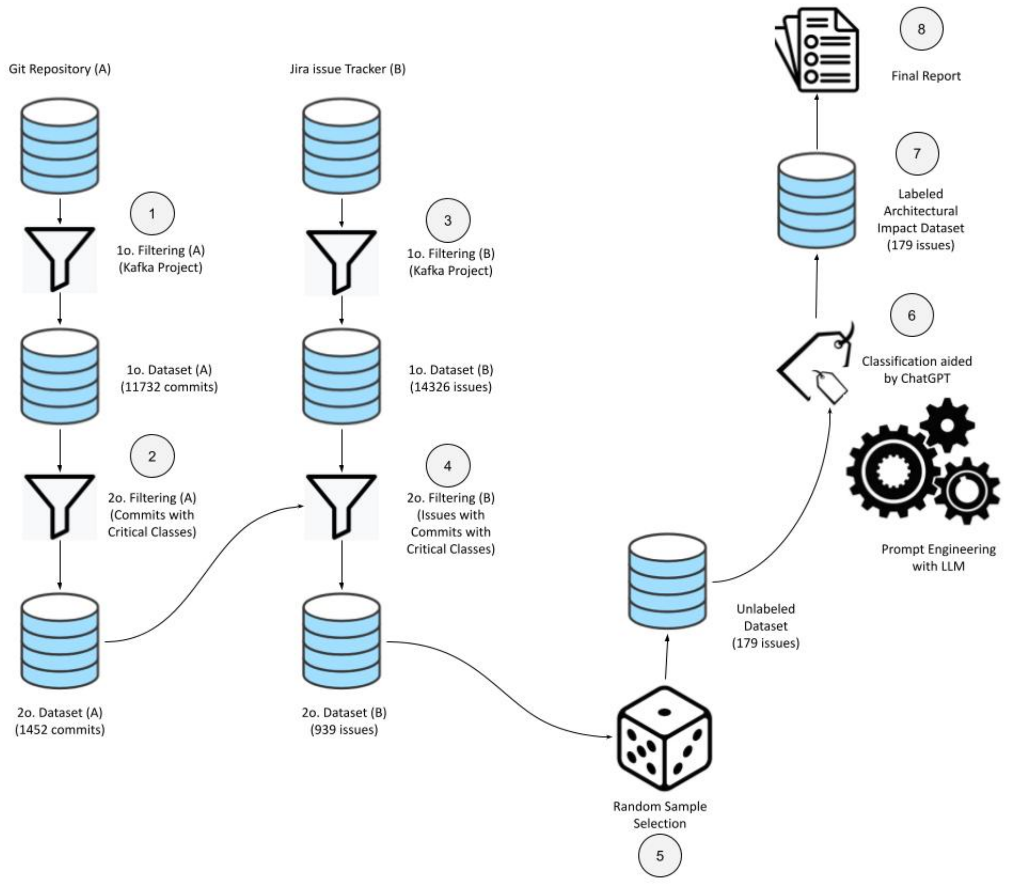

Git Repository (A)
Jira issue Tracker (B)
1
1o. Filtering (A) (Kafka Project)
1o. Dataset (A) (11732 commits)
2
2o. Filtering (A) (Commits with Critical Classes)
2o. Dataset (A) (1452 commits)
3
1o. Filtering (B) (Kafka Project)
1o. Dataset (B) (14326 issues)
4
2o. Filtering (B) (Issues with Commits with Critical Classes)
2o. Dataset (B) (939 issues)
Random Sample Selection
5
Unlabeled Dataset (179 issues)
Prompt Engineering with LLM
6
Classification aided by ChatGPT
7
Labeled Architectural Impact Dataset (179 issues)
8
Final Report

Figure 38 – Evaluating the ATDCodeAnlyzer in Apache Kafka repository

Kafka Critical Classes: ['StreamThread.java', 'KafkaConsumer.java', 'StreamTask.java', 'Fetcher.java', 'KafkaStreams.java','KStreamImpl.java', 'KafkaProducer.java','StreamsConfig.java', 'ConsumerCoordinator.java']

All steps and details about how to extract critical files from Apache Kafka using ATDCodeAnlyzer are available in the replication package[21].

Semi-automatic classification aided by ChatGPT with Prompt Engineering: In this step, we use ChatGPT to assist us in the classification process. We provide prompts to ChatGPT to help automate or semi-automate some of the classification tasks.

Finally, we have a set of issues labeled with architectural impact and without architectural impact.

9.3.2.3 Evaluating the ATDCodeAnalyzer in Apache Hadoop

The process evaluation in Apache Hadoop is detailed in the Figure 39.

First Filter (A) - Hadoop Project: In this step, we select and filter data related to

[21] https://github.com/Technical-Debt-Large-Scale/atdcodeanalyzer

the Apache Hadoop project from the GitHub repository. We focus on data that is relevant to architectural changes or issues.

Second Filter (A) - Commits with Critical Classes: After filtering the Hadoop project data, we identify commits that involve critical classes. We consider critical classes to be those that are related to architectural issues identified by ATDCodeAnalyzer

First Filter (B) - Hadoop Project: In this step, we perform a similar filtering process on the Jira Issue Tracker. We specifically look for issues related to the Hadoop project.

Second Filter (B) - Issues with Commits with Critical Classes: In this step, we identify Jira issues that are associated with commits containing critical classes. This allows us to link code changes with corresponding issues.

In Apache Hadoop repository, and we obtained the following critical files:

Hadoop Critical Classes: ['Configuration.java', 'Writable.java', 'StringUtils.java', 'FSDataOutputStream.java', 'BytesWritable.java', 'WritableComparable.java', 'DatanodeProtocol.java', 'ClientProtocol.java', 'FSNamesystem.java', 'DataNode.java','BlockManager.java', 'ResourceScheduler.java', 'ContainerManager.java', 'FairScheduler.java','CapacityScheduler.java', 'NodeManager.java', 'Job.java', 'Mapper.java', 'Reducer.java', 'InputFormat.java', 'OutputFormat.java']

All steps and details about how to extract critical files from Apache Hadoop using ATDCodeAnlyzer are available in the replication package[22].

Semi-automatic classification aided by ChatGPT with Prompt Engineering: In this step, we use ChatGPT to assist us in the classification process. We provide prompts to ChatGPT to help automate or semi-automate some of the classification tasks.

Finally, we have a set of issues labeled with architectural impact and without architectural impact.

9.4 Results

This section presents the results in two parts. The first part focuses on the Apache Cassandra results and includes details about the inspection process and the definition of the labeling inspection process. The second part presents the results of the analysis of the four repositories (Apache Cassandra, Apache Kafka, Apache ActiveMQ and Apache Hadoop) analyses related to issues with architectural impact in commits with critical classes.

[22] https://github.com/Technical-Debt-Large-Scale/atdcodeanalyzer

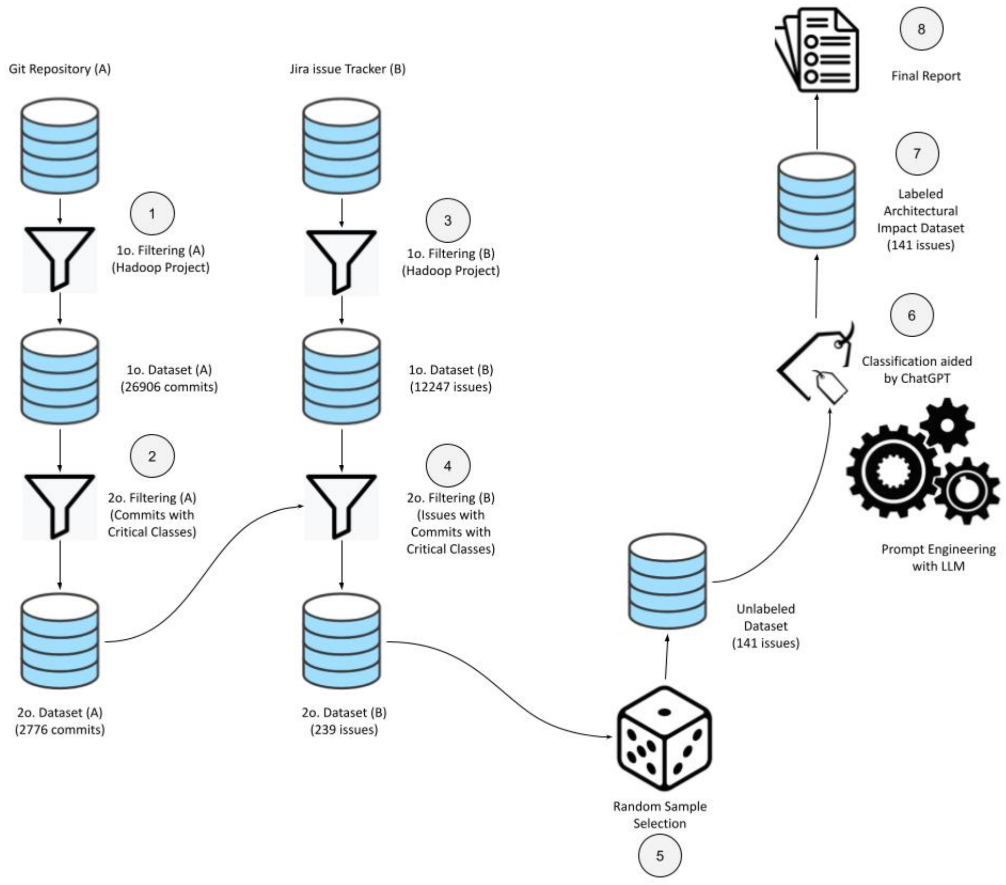

Git Repository (A)
Jira issue Tracker (B)
1
1o. Filtering (A) (Hadoop Project)
3
1o. Filtering (B) (Hadoop Project)
1o. Dataset (A) (26906 commits)
1o. Dataset (B) (12247 issues)
2
2o. Filtering (A) (Commits with Critical Classes)
4
2o. Filtering (B) (Issues with Commits with Critical Classes)
2o. Dataset (A) (2776 commits)
2o. Dataset (B) (239 issues)
Random Sample Selection
5
Unlabeled Dataset (141 issues)
6
Classification aided by ChatGPT
Prompt Engineering with LLM
7
Labeled Architectural Impact Dataset (141 issues)
8
Final Report

Figure 39 – Validating in Hadoop

9.4.1 Analysis of Apache Cassandra Inpection Process

In this section, we present the results of the evaluation conducted in the Apache Cassandra repository related to inspection process.

9.4.1.1 Commit data and Issue data

We provide data related to the analysis of Cassandra's commits and issues within the timeframe from 2009 to 2023.

9.4.1.2 Results of Issues Inspection

In this section, we present the outcomes of both manual inspections and inspections conducted with the assistance of ChatGPT. Additionally, we calculate Cohen's Kappa to evaluate the degree of agreement between manual inspections and those supported by ChatGPT. All data and scripts used to evaluate the inspection process is availabe in the replication kit[23].

[23] https://github.com/Technical-Debt-Large-Scale/my_validation/blob/main/evaluate_inspection_model.ipynb

Table 18 – Results of Manual Inspection

issue_key	issue_type	summary	description	ai
CASSANDRA-3237	Improvement	refactor x column implmentation...	x columns are annoying. composite columns...	Yes
CASSANDRA-10661	Improvement	Integrate SASI to Cassandra	We have recently released new secondary index ...	Yes
CASSANDRA-8707	Bug	Move SegmentedFile, IndexSummary ...	There are still a few bugs with resource manag...	Yes
CASSANDRA-10091	New Feature	Integrated JMX authn authz	It would be useful to authenticate with JMX th...	Yes
...	...	...	... ...	...
CASSANDRA-185	New Feature	user-defined column ordering	name ordering by x String order isnt gene...	No
CASSANDRA-13994	Improvement	Remove dead compact storage code bef...	4.0 comes without x (after [CASSANDRA-111...	No
CASSANDRA-13985	Improvement	Support restricting access to specif...	There are a few use cases where it makes sense...	No
CASSANDRA-17153	New Feature	Guardrails for collection items and ...	Add guardrails for the number of items and siz...	No

Table 19 – Results of ChaGTP Inspection

issue_key	issue_type	summary	description	ai
CASSANDRA-3237	Improvement	refactor x column implmentation...	x columns are annoying. composite columns...	Yes
CASSANDRA-10661	Improvement	Integrate SASI to Cassandra	We have recently released new secondary index ...	Yes
CASSANDRA-8707	Bug	Move SegmentedFile, IndexSummary ...	There are still a few bugs with resource manag...	Yes
CASSANDRA-10091	New Feature	Integrated JMX authn authz	It would be useful to authenticate with JMX th...	Yes
...	...	...	... ...	...
CASSANDRA-9425	Sub-task	Make node-local schema fully immutable	The way we handle schema changes currently is ...	Yes
CASSANDRA-9402	Task	Implement proper sandboxing for UDFs	We want to avoid a security exploit for our us...	Yes
CASSANDRA-185	New Feature	user-defined column ordering	name ordering by x String order isnt gene...	No
CASSANDRA-13994	Improvement	Remove dead compact storage code bef...	4.0 comes without x (after [CASSANDRA-111...	No

Results of Manual Inspection

We inspected 226 issues in total, and during the manual inspection, we found that 33.63% of them were classified as "Yes" (76), while the remaining 66.37% were categorized as "No" (150). You can see a fragment of results of manual inspection in Table 18.

Results of ChatGTP Inspection

We inspected 226 issues in total, and during the ChatGPT inspection, we found that 42,48% of them were classified as "Yes" (96), while the remaining 57,52% were categorized as "No" (130). You can see the fragment of results of ChatGPT inspection in Table 19

Results Cohen's Kappa between Manual and ChatGPT Inspection

In the preceding section, we discussed the percentages of 'Yes' and 'No' results obtained from both manual inspection and ChatGPT inspection. To quantify the agreement between these two inspection methods, we applied Cohen's Kappa analysis. The calculated Cohen's Kappa score was found to be 0.721, indicating a substantial level of agreement between manual inspection and ChatGPT inspection, as you can see the Table 20.

In our evaluation of the model used for inspecting issues, we obtained the following key performance metrics: Precision: 0.926, Recall: 0.833, Accuracy: 0.867 and F1-Score: 0.893. These metrics provide valuable insights into the model's effectiveness. A precision of 0.926 indicates that when the model predicts an issue as 'Yes,' it is correct 92,6% of the time. A recall of 0.833 means that the model correctly identifies all actual 'Yes' issues. The high accuracy of

Table 20 – Cohen's kappa score interpretation.

Kappa Statistic	Strength of Agreement
<0.00	Poor
0.00 - 0.20	Slight
0.21 - 0.40	Fair
0.41 - 0.60	Moderate
0.61 - 0.80	Substantial
0.81 - 1.00	Almost Perfect

0.867 highlights the model's ability to correctly classify issues in general. The F1-Score of 0.893 balances the trade-off between precision and recall.

These results demonstrate the model's good performance in identifying architectural issues and technical debt in software issues, emphasizing its potential as an effective inspection tool when compared to manual inspection. The model's high recall value is particularly significant, as it indicates that the model excels in identifying issues with 'Yes' classifications, which is crucial in identifying and addressing architectural problems and technical debt issues.

9.4.2 Analysing Issues with Architectural Impact

In this section we will do a comparison of issues in commits with critical classes among Apache Cassandra, Apache ActiveMQ, Apache Kafka and Apache Hadoop related to architectural impact. All data and scripts used to perform this comparison are available in the replication kit[24].

9.4.2.1 RQ1) What is the proportion of issues classified as impacting architectural design, as observed in each analyzed software repository?

As you can see in Figure 40, from all issues in commits with critical classes from all analyzed repositories have over 34.5% of issues with architectural impact. Also, As you can see in Figure 41, the time to resolve issues with architectural impact associated with critical classes is greater than for issues associated with no critical classes. This suggests that architectural technical debt in these critical classes may lead to longer resolution times for issues affecting them. Furthermore, we can see in figure 43 a further pattern can also be seen in figure 42, where commits with critical classes tend to decrease over time, showing the importance of resolving the issues associated with these kind of classes.

[24] https://github.com/Technical-Debt-Large-Scale/my_validation/blob/main/my_comparison.ipynb

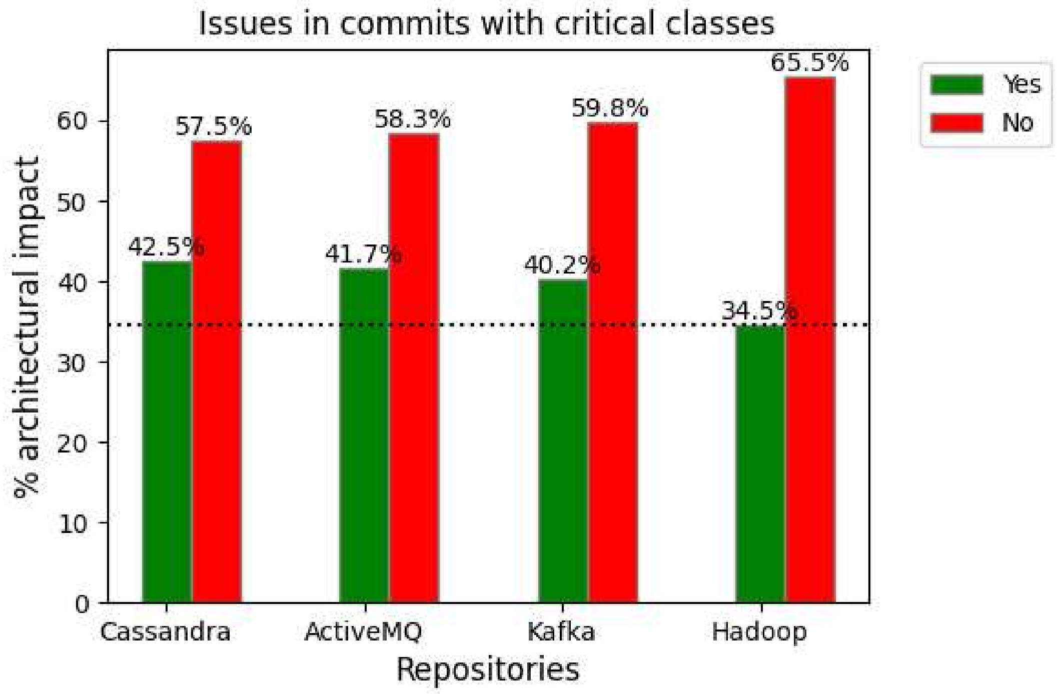
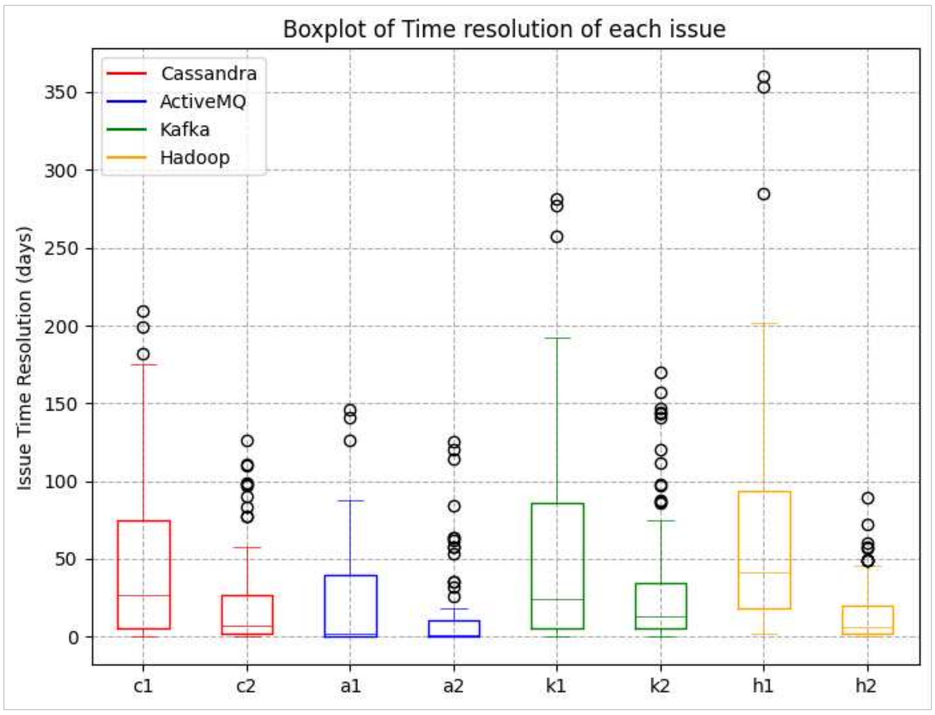


Figure 40 – Percentual of issues with architectural impact and without architectural impact



c1 (Cassandra) - issues with AI
c2 (Cassandra) - issues without AI
a1 (ActiveMQ) - issues with AI
a2 (ActiveMQ) - issues without AI
k1 (Kafka) - issues with AI
k2 (Kafka) - issues without AI
h1 (Hadoop) - issues with AI
h2 (Hadoop) - issues without AI

AI - Architectural Impact

Figure 41 – Boxplot time resolution from issues related to architectural impact

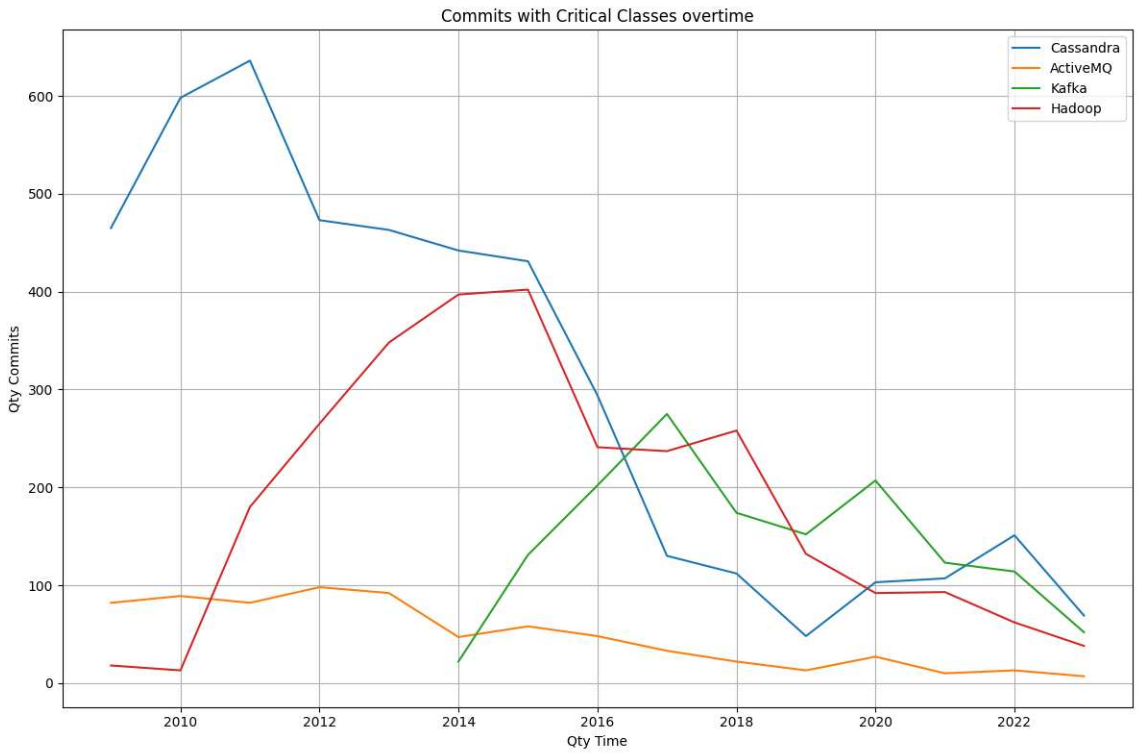


Figure 42 – Commits with critical classes overtime

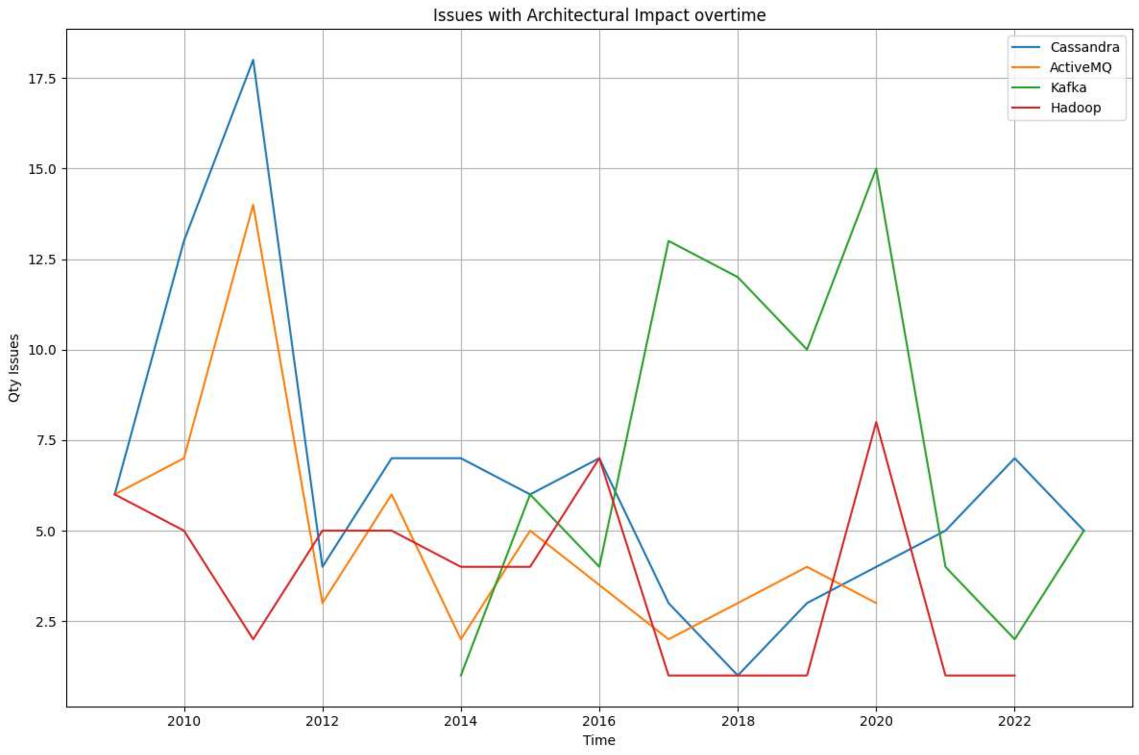


Figure 43 – Issues with architectural impact overtime

9.4.2.2 RQ2)Which classes are most commonly involved in issues that impact the overall architecture?

As you can see in Table 21, the critical classes found by ATDCodeAnalyzer are among the TOP 20 classes (Apache Cassandra) with the most issues with architectural impact. Also, the Table 22 shows that all critical classes from Apache Kakfa appears in TOP 20 classes with the most issues with architectural impact. Also, the Table 23 shows that there are 9 critical classes from Apache Hadoop appears in TOP 20 classes with the most issues with architectural impact. Finally, the Table 24 shows that there are 6 critical classes from Apache ActiveMQ that appears in TOP 20 classes with the most issues with architectural impact.

Furthermore, our analysis also reveals that some frequently changed classes identified as impacted by ATD are primarily metadata classes or those containing configurations or special project notes. For instance, in Table 23 the package-info.java file, which serves a dual purpose as a place for package-level documentation and a home for package-level annotations, is likely to be modified more often than other code due to its informational nature. Also, there are some Test classes that appears in issues with architectural problems because are special test classes created to validate the bug-fix or validate the improvement implemented in issue.

To visualize the distribution of issues with architectural impact between critical and non-critical classes, you can see the boxplots in Figures 44, 45, 46, and 47 show the results for each analyzed project. The critical classes tends to be more issues with architectural impact. We observe that critical classes consistently rank among the top 20 classes with the most architectural impact issues.

Based on these results, our analysis confirms a key hypothesis: critical classes identified by our method are more likely to be involved in issues that significantly impact the software architecture. This finding suggests that these critical classes are potential hotspots for architectural problems.

9.5 Conclusions

Therefore, changes affecting critical classes in a software project tend to also involve issues that have a significant impact on the software's architecture. Also, this kind of these issues tend to spend more time to be resolved. This means that fixing these issues might require not just patching a specific bug but also potentially restructuring or redesigning parts of the architecture.

Table 21 – Top 20 classes (Cassandra) with more issues with architectural impact

Id	Class	Qtd of issues with architectural impact	Critical Class
1	**ColumnFamilyStore.java**	20	Yes
2	**StorageService.java**	14	Yes
3	**DatabaseDescriptor.java**	11	Yes
4	**SSTableReader.java**	11	Yes
5	**CompactionManager.java**	10	Yes
6	Memtable.java	8	No
7	**Config.java**	8	Yes
8	CommitLog.java	6	No
9	SSTableWriter.java	6	No
10	**MessagingService.java**	6	Yes
11	AntiCompactionTest.java	5	No
12	SSTable.java	5	No
13	**StorageProxy.java**	5	Yes
14	Table.java	5	No
15	CompactionTask.java	5	No
16	StreamingTransferTest.java	4	No
17	**NodeProbe.java**	4	Yes
18	Util.java	4	No
19	CQLTester.java	4	No
20	CassandraServer.java	4	No

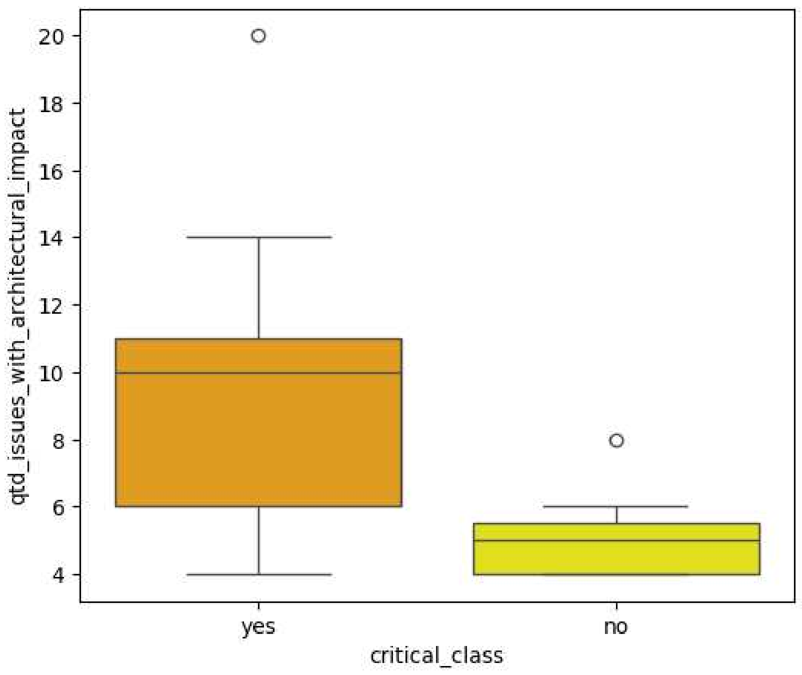


Figure 44 – Boxplot of TOP 20 classes (Apache Cassandra) with the most issues with architectural impact

Table 22 – Top 20 classes (Kafka) with more issues with architectural impact

Id	Class	Qtd of issues with architectural impact	Critical Class
1	**StreamThread.java**	14	Yes
2	StreamThreadTest.java	14	No
3	FetcherTest.java	13	No
4	**Fetcher.java**	11	Yes
5	**StreamTask.java**	10	Yes
6	**KStreamImpl.java**	10	Yes
7	**ConsumerCoordinator.java**	10	Yes
8	**KafkaConsumer.java**	9	Yes
9	**StreamsConfig.java**	8	Yes
10	ConsumerCoordinatorTest.java	8	No
11	StreamTaskTest.java	8	No
12	**KafkaProducer.java**	7	Yes
13	TaskManager.java	7	No
14	StreamThreadStateStoreProviderTest.java	6	No
14	KafkaConsumerTest.java	6	No
16	AbstractCoordinator.java	6	No
17	MockProcessorContext.java	5	No
18	TopologyTestDriver.java	5	No
19	**KafkaStreams.java**	5	Yes
20	MemoryRecords.java	5	No

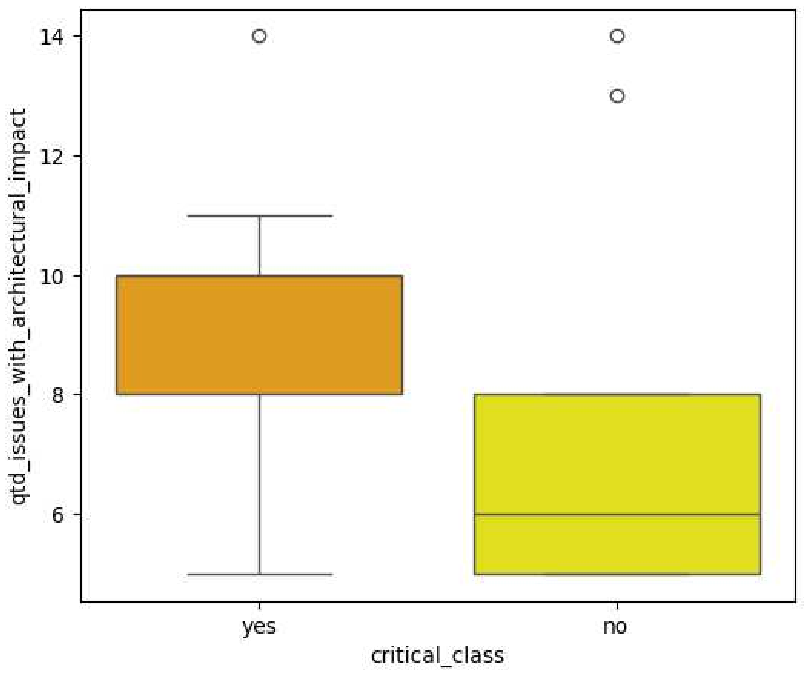


Figure 45 – Boxplot of TOP 20 classes (Apache Kafka) with the most issues with architectural impact

Table 23 – Top 20 classes (Hadoop) with more issues with architectural impact

Id	Class	Qtd of issues with architectural impact	Critical Class
1	package-info.java	38	No
2	**Configuration.java**	15	Yes
3	**Writable.java**	9	Yes
4	FileContext.java	8	No
5	**WritableComparable.java**	8	Yes
6	FileSystem.java	7	No
7	**BlockManager.java**	7	Yes
8	**ClientProtocol.java**	7	Yes
9	Utils.java	7	No
10	**ContainerManager.java**	6	Yes
11	EventCounter.java	6	No
12	RawLocalFileSystem.java	6	No
13	DatanodeProtocol.java	6	No
14	TestConfiguration.java	6	No
15	FSDataOutputStream.java	6	No
16	**NodeManager.java**	5	Yes
17	Options.java	5	No
18	Groups.java	5	No
19	**StringUtils.java**	5	Yes
20	**CapacityScheduler.java**	5	Yes

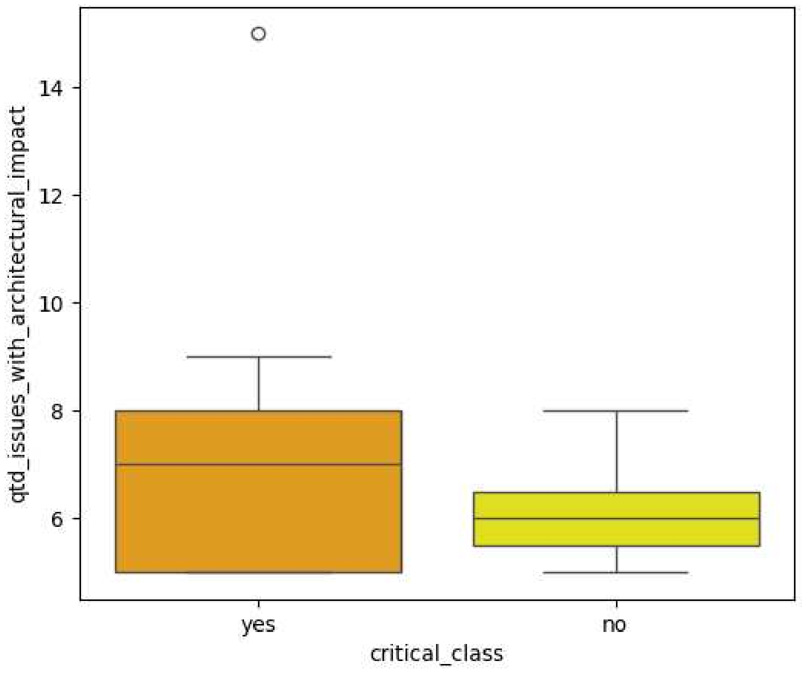


Figure 46 – Boxplot of TOP 20 classes (Apache Hadoop) with the most issues with architectural impact

Table 24 – Top 20 classes (ActiveMQ) with more issues with architectural impact

Id	Class	Qtd of issues with architectural impact	Critical Class
1	**Queue.java**	12	Yes
2	**QueueDispatchSelector.java**	5	Yes
3	**BrokerService.java**	5	Yes
4	Topic.java	4	No
5	PrefetchSubscription.java	4	No
6	AbstractStoreCursor.java	4	No
7	KahaReferenceStore.java	3	No
8	**QueueBrowserSubscription.java**	3	Yes
9	AMQ2149Test.java	3	No
10	BaseDestination.java	3	No
11	TransactionContext.java	3	No
12	ActiveMQConnection.java	3	No
13	QueueDuplicatesFromStoreTest.java	3	No
14	TestSupport.java	3	No
15	RegionBroker.java	3	No
16	StoreDurableSubscriberCursor.java	3	No
17	DemandForwardingBridgeSupport.java	3	No
18	AbstractSubscription.java	3	No
19	**DurableTopicSubscription.java**	3	Yes
20	**TransportConnector.java**	3	Yes

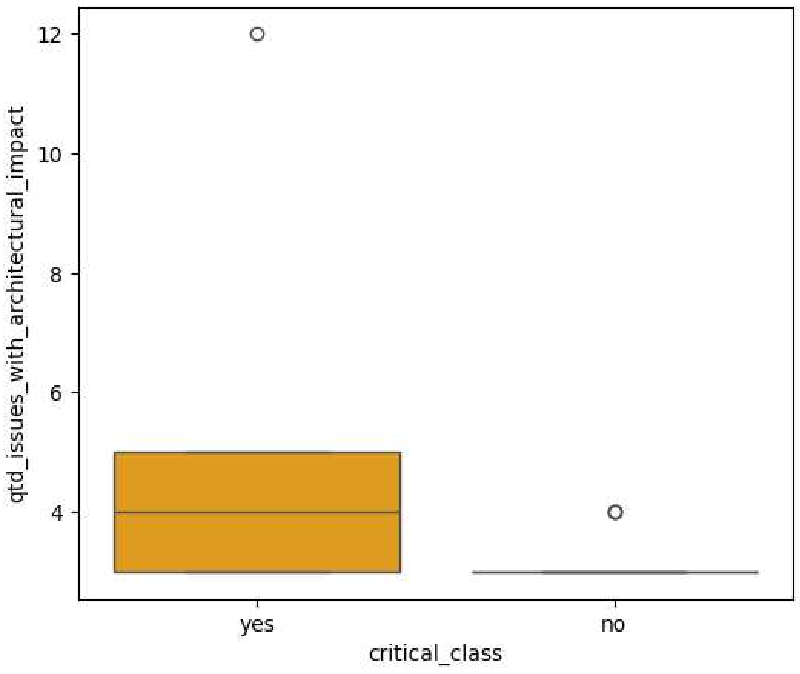


Figure 47 – Boxplot of TOP 20 classes (Apache ActiveMQ) with the most issues with architectural impact

We observed that the critical classes, identified by the ATDCodeAnalyzer method, are the classes that have a greater number of issues with architectural impact.

This experiment has yielded the following specific contributions:

– We developed a LLM-based approach to identify issues with architectural impact.
– We provided a dataset (related to Apache Cassandra, Apache ActiveMQ, Apache Kafka, and Apache Hadoop) containing texts with details about inspections of issues with architectural impact, alongside information on the corresponding commits.

9.6 Artefact Availability

We use a replication kit [25] that contains the dataset, evaluation protocol, scripts regarding data analysis, and scripts regarding generating results and utilized prompts.

To promote further research in this area, this replication kit facilitates easy study replication, addressing the common challenge of high cost and complexity in this type of mining process. The proposed method for semi-automatic issue inspection is extensible to other open-source projects, provided they adhere to good practices for recording issue tracker tickets linked to the corresponding source code repository.

[25] https://github.com/Technical-Debt-Large-Scale/my_validation

10 CONCLUSIONS AND FUTURE WORK

This chapter summarizes the primary findings of this research and concludes the thesis by answering the research questions. It also explores promising avenues for future research in the field of ATD identification and monitoring processes.

Our thesis aims to offer valuable insights into the identification, measurement, and monitoring of Architectural Technical Debt (ATD) in software development projects. To achieve this overarching goal, we undertook a multi-faceted approach, beginning with a systematic mapping study to review existing literature and research pertaining to ATD.

Following this, we conducted a case study within a large-scale distributed project. This case study provided us with a deeper understanding of the primary factors associated with Technical Debt (TD) in extensive software systems. Additionally, we performed a series of exploratory analyses across various Git repositories, allowing us to experiment and formulate a methodology based on data extracted from these repositories. Our method focuses on the selection of code files that are indicative of ATD.

In parallel, we developed a dedicated tool known as "SysRepoAnalysis" to streamline data extraction and metric calculations, further contributing to the ATD management process.

As a critical step in our research, we carried out a study titled "Identifying source code files that indicate Architectural Technical Debt." The primary objective of this study was to evaluate the proposed method in a real-world case. Our research aspires to broaden the comprehension of ATD and to provide practical tool and techniques for its effective ATD identification in software development projects.

Finally, we devised a evaluation method, employing Self-Admitted Technical Debt (SATD) keywords found in commits and issues. This method enables us to assess whether issues associated with architectural problems encompass the critical files identified by the ATDCode-Analyzer. Consequently, our thesis strives to create a systematic approach for identifying source code artifacts within software repositories that signify the presence of ATD.

10.1 General Discussion

This section we document (i) a revisitation of the research questions underlying this study and (ii) an overview of the implications of our findings.

10.1.1 Research Questions Revisited

In this subsection, we discuss how our thesis answers the research questions presented in Chapter 1.

RQ1 - What are the main challenges that large-scale software projects face about Architectural Technical Debt?

RQ1 was addressed through studies in different publications. The systematic mapping study in the Chapter 4 provided a comprehensive overview of the state of the art in Architectural Technical Debt (ATD) and helped identify research questions and gaps in the literature. The investigation conducted in an industrial case study at Ericsson in Chapter 5 it was important to understand the factors related to TD accumulation in large-scale Global Software Engineering (GSE) projects.

According to our SMS we observed a growing interest in the identification of Architectural Technical Debt (ATD), and this has led to a diverse range of research in the field. Studies related to ATD are presented at various venues, including well-established international conferences in the field of software engineering and architecture, such as the International Conference on Software Engineering (ICSE), the TechDebt Conference and the International Conference on Software Architecture (ICSA).

We also observed that ATD identification techniques are based on TD identification techniques at the source code level. The state-of-the-art of ATD identification techniques is diverse and includes various analysis types such as identification of architectural antipatterns, dependency analysis, change impact analysis, and manual classification of software artifacts. However, only a small portion of the literature addresses ATD resolution, and the history analysis is considered only by a small portion of state-of-the-art identification techniques. Although many tools for ATD identification are proposed in the literature, only a small portion of them is publicly available. In addition, as we can observe in Chapter 4 and Chapter 5, there are three main challenges related to ATD: i)Identification: ATD items can be challenging to identify and monitor, as they often permeate various stages and artifacts of the software development cycle. ii) Accumulation: The accumulation of ATD can lead to more significant maintenance efforts, such as higher bug-fix costs, more substantial efforts to add new features, and increased efforts to maintain existing features. iii) Hindering future development: Over time, accumulated ATD can make system evolution more challenging in the long term, ultimately hindering future development activities.This suggests promising research directions for the future.

RQ2-How to identify ATD in a systematic and reliable way?

RQ2 was addressed through a proposed method and tool in the Chapter 7, which extracted information from git repositories to discover the leading factors related to ATD in source code. Another study in Ericcson reported in Chapter 8 used a systematic method to identify source code artifacts that indicate the presence of ATD and employed quantitative and qualitative methods to validate the proposed method.

According to the discussion presented in this thesis, the way to identify source code artifacts that may indicate the presence of Architectural Technical Debt (ATD) is detailed in the proposed method, presented in Chapter 6 and evaluated in Chapter 8. It involves extracting information from git repositories to discover the leading factors related to ATD in source code. This method includes several steps, such as identifying and collecting data from the repositories, preprocessing the data, analyzing the data to identify ATD factors, and finally, identifying the source code artifacts that may indicate the presence of ATD.

In the study, presented in Chapter 8, we used a systematic method to identify source code artifacts that indicate the presence of ATD. The method employed both quantitative and qualitative methods to validate its effectiveness. The results of this study showed that it is possible to identify source code artifacts that indicate the presence of ATD using Architectural Smells and analyze various metrics and characteristics of the code, such as complexity, coupling, and modularity. Furthermore, in chapter 9, we applied a systematic method to four real-world projects to validate the effectiveness of ATDCodeAnalyzer. Our analysis observed that the critical classes, identified by our method, tend to be the project classes that most appear in issues with architectural impact, showing that such classes are involved in software architectural problems.

Hence, the proposed method involves a combination of Architectural Smell and analysis of various metrics and characteristics of the code, as well as considering the context and history of the code within the software development process. Using a systematic and well-defined method is essential to ensure the accuracy and effectiveness of the identification process.

10.1.2 Conclusion validity

One significant challenge to the validity of the conclusions drawn in this thesis is the potential incompleteness of our results. We acknowledge that architectural technical debt is a complex and multifaceted phenomenon with various dimensions and is influenced by numerous factors, as discussed in Chapter 4. While the research reported in this thesis aims to advance our understanding of the identification of source code files affected by architectural technical debt, our results cannot claim to be comprehensive or exhaustive concerning all kinds of git repositories because we performed testing and validation using java system projects. In this thesis, we focused on studies and tools that use source code analysis to identify, analyze, and evaluate architectural technical debt related to git repositories using well-established mining software repositories techniques. Therefore, it is crucial to interpret the results presented in this thesis in light of the specific research methodology used. We employed the Design Science Methodology to propose a solution that can be useful in real-world software projects that use git repositories to extract essential information for decision-making regarding software architecture and technical debt.

10.1.3 Research Implications

In this section, we give a quick overview of how software researchers and practitioners can use the results of our thesis to advance the field of software engineering.

In Chapter 4, the systematic mapping study on Architectural Technical Debt has several research implications. Firstly, the study reveals that there is no consensus on the detection and identification of ATD, indicating the need for further research in this area. Secondly, the study provides a comprehensive review and guidance for researchers and practitioners seeking insights into the identification, measurement, monitoring, tools, methods, and calculation of ATD. This implies that the study can serve as a reference for future research and practical applications related to ATD. Thirdly, the study identifies trends in ATD types, measurements, monitoring, tools, and methods, providing directions for future research and practice in this field. Fourthly, the study highlights the need for a combination of methods to extract ATD information beyond the source code and develop more precise methods to extract information from software architectural documents. Finally, the study suggests that using formulas and expert evaluation are the primary ways to determine the cost of fixing ATD, indicating a need for further research to develop more accurate and reliable cost calculation methods.

In Chapter 5, this study explored technical debt (TD) accumulation in large-scale Global Software Engineering (GSE) projects at Ericsson. The findings highlight the importance of managing TD in such projects to prevent software degradation, which can be more challenging in distributed projects. The study identified factors such as task complexity, lead time, total number of developers, and task scaling as being related to TD accumulation. Specifically, the study revealed that task complexity has a strong correlation with TD accumulation. Therefore, practitioners are advised to avoid complex or big tasks and instead break down large tasks into smaller ones whenever possible to prevent TD accumulation. Implementing this approach to managing TD may lead to better software quality and prevent software degradation in large-scale distributed projects. The study's implications emphasize the need for empirical studies to better understand and manage technical debt in software engineering projects. Researchers and practitioners can use the findings to inform TD management practices in similar projects. However, to further validate the findings, it is recommended to conduct similar research in other companies to gain more knowledge about TD accumulation in large-scale globally distributed software projects.

In Chapter 7, the SysRepoAnalysis tool has several research implications. First, it provides a way to automate the process of extracting information from code repositories, which can save time and increase efficiency. Second, it generates metrics that can help identify critical source code files, which can aid in software maintenance and improvement. Third, the tool offers visualization features that allow developers to easily identify and analyze critical areas of the software. Finally, by exporting the generated metrics to .csv files, the tool enables data analysis in other tools, facilitating further research and analysis. Then, this tool has the potential to enhance the software development process by providing developers with a more efficient and effective way to analyze and identify critical areas of their code repositories.

In Chapter 8, the research implications of the study on identifying source code files that indicate Architectural Technical Debt are several. First, the study provides a systematic method for identifying ATD in source code artifacts based on a combination of architectural smells, code change analysis, and code metrics. This method could be used by practitioners to identify and prioritize areas of the software that require architectural refactoring to reduce ATD. Second, the study highlights the importance of using information from source code repositories to identify ATD items. This approach is more efficient than relying on manual analysis of the code, which is time-consuming and error-prone. Third, the study demonstrates the effectiveness of combining different techniques such as architectural smells, code change analysis, and code metrics to identify ATD in source code artifacts. This could inspire future research to explore other combinations of techniques to identify ATD or other types of software debt. Therefore, the study provides a valuable contribution to the field of software engineering by providing a systematic method for identifying ATD in source code artifacts and highlighting the importance of using information from source code repositories to do so.

Finally, in Chapter 9, we introduce a systematic evaluation approach for our proposed method, conducted through experiments across four real-world repositories. Our objective was to assess the behavior of metrics associated with critical classes identified by our method that contains issues with architectural impact. Initially, we developed a semi-automatic inspection model to aid in scrutinizing issues sourced from the repositories' issue trackers. Subsequently, we employed ATDCodeAnalyzer on the selected repositories to pinpoint critical classes affected by Architecture Technical Debt (ATD). In a subsequent step, we extracted commits containing critical classes and correlated these with issue data. This allowed us to evaluate the behavior of issues indicating architectural impact.

10.1.4 Replicability

Throughout all the studies included in this thesis proposal, we made a concerted effort to be as transparent as possible about our research processes and results. Our aim was to enable independent scrutiny and replication of our work. To this end, we created a replication package for each study, which includes all of the source code we used, the data we considered, and all of our intermediate, additional, and final results.

10.2 Future Research Directions

The works described in this thesis serve as a starting point for the ATD identification and monitoring process, laying the foundation for further advancements and insights in managing Architectural Technical Debt in software development. The ATDCodeAnalyzer method, contributes in addressing the challenges of managing Architectural Technical Debt (ATD), paves the way for further advancements in software development. Several promising avenues for future research directions include:

1. Expanding ATDCodeAnalyzer applicability:

Integration into Development Workflows: Explore seamless integration of ATDCode-Analyzer into existing development workflows, empowering developers to proactively detect and manage ATD.

2. Impact Analysis of ATD on Software Quality Metrics:

Correlation Studies: Analyze the relationship between ATD presence and diverse software quality metrics, such as performance, security, maintainability, and reliability.

Identification of High-Impact ATD: Develop methodologies to identify ATD elements most likely to adversely affect software quality metrics, facilitating prioritized refactoring efforts.

3. Augmenting ATD Identification using Machine Learning:

Machine Learning Models: Investigate machine learning techniques to train models for ATD detection based on code patterns, commit history, and relevant data.

Enhancing Accuracy: Explore machine learning approaches to improve ATD identification accuracy, minimizing false positives and reducing manual intervention.

4. ATD identification from issue analysis:

New experiments to investigate other LLM models or techniques to identify issues with architectural impact

5. Prioritization and Refactoring of ATD Items:

Prioritization Framework: Develop frameworks for prioritizing ATD elements considering their impact on software quality, development efforts, and business priorities.

Automated Refactoring Plans: Design tools or frameworks capable of generating automated refactoring plans for prioritized ATD elements, streamlining the refactoring process.

6. Integration of ATD Management into Software Development Processes:

Comprehensive ATD Framework: Develop frameworks embedding ATD identification, tracking, and management into the software development lifecycle.

Continuous Monitoring: Implement mechanisms for incessant monitoring of ATD throughout the development process, enabling early detection and proactive remediation.

Deploy the ATDCodeAnalyzer method in a real environment.

BIBLIOGRAPHY

AGRAWAL, A.; SINGH, R. Ruffle: Extracting co-change information from software project repositories. In: IEEE. **2018 International Conference on Smart Systems and Inventive Technology (ICSSIT)**. [*S. l.*], 2018. p. 88–91.

AGRAWAL, A.; SINGH, R. Identification of co-change patterns in software evolution. In: IEEE. **2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)**. [*S. l.*], 2020. p. 781–785.

ALFAYEZ, R.; CHEN, C.; BEHNAMGHADER, P.; SRISOPHA, K.; BOEHM, B. An empirical study of technical debt in open-source software systems. In: SPRINGER. **Disciplinary Convergence in Systems Engineering Research**. [*S. l.*], 2018. p. 113–125.

ALVES, N. S.; MENDES, T. S.; MENDONÇA, M. G. de; SPÍNOLA, R. O.; SHULL, F.; SEAMAN, C. Identification and management of technical debt: A systematic mapping study. **Information and Software Technology**, Elsevier, v. 70, p. 100–121, 2016.

ALZAGHOUL, E.; BAHSOON, R. Evaluating technical debt in cloud-based architectures using real options. In: **2014 23rd Australian Software Engineering Conference**. [*S. l.*: *s. n.*], 2014. p. 1–10. ISSN 1530-0803.

AMPATZOGLOU, A.; AMPATZOGLOU, A.; CHATZIGEORGIOU, A.; AVGERIOU, P. The financial aspect of managing technical debt: A systematic literature review. **Information and Software Technology**, Elsevier, v. 64, p. 52–73, 2015.

ANICHE, M.; BAVOTA, G.; TREUDE, C.; GEROSA, M. A.; DEURSEN, A. van. Code smells for model-view-controller architectures. **Empirical Software Engineering**, Springer, v. 23, n. 4, p. 2121–2157, 2018.

ANTINYAN, V.; STARON, M.; MEDING, W.; ÖSTERSTRÖM, P.; WIKSTROM, E.; WRANKER, J.; HENRIKSSON, A.; HANSSON, J. Identifying risky areas of software code in agile/lean software development: An industrial experience report. In: IEEE. **2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)**. [*S. l.*], 2014. p. 154–163.

AVANCINI, R. Uma ferramenta de visualização de software para avaliação de uso de api no contexto de ecossistemas de software. Universidade Federal de São Paulo, 2021.

AVRITZER, A.; BEECHAM, S.; BRITTO, R.; KROLL, J.; MENASCHE, D. S.; NOLL, J.; PAASIVAARA, M. Extending survivability models for global software development with media synchronicity theory. In: IEEE. **Global Software Engineering (ICGSE), 2015 IEEE 10th International Conference on**. [*S. l.*], 2015. p. 23–32.

AZADI, U.; FONTANA, F. A.; TAIBI, D. Architectural smells detected by tools: a catalogue proposal. In: IEEE. **2019 IEEE/ACM International Conference on Technical Debt (TechDebt)**. [*S. l.*], 2019. p. 88–97.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software architecture in practice**. [*S. l.*]: Addison-Wesley Professional, 2003.

BAVANI, R. Distributed agile, agile testing, and technical debt. **IEEE Software**, v. 29, n. 6, p. 28–33, Nov 2012. ISSN 0740-7459.

BAVOTA, G.; RUSSO, B. A large-scale empirical study on self-admitted technical debt. In: **Proceedings of the 13th international conference on mining software repositories**. [*S. l.*: *s. n.*], 2016. p. 315–326.

BESKER, T.; MARTINI, A.; BOSCH, J. Impact of architectural technical debt on daily software development work—a survey of software practitioners. In: IEEE. **2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**. [*S. l.*], 2017. p. 278–287.

BESKER, T.; MARTINI, A.; BOSCH, J. The pricey bill of technical debt: When and by whom will it be paid? In: IEEE. **2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [*S. l.*], 2017. p. 13–23.

BESKER, T.; MARTINI, A.; BOSCH, J. The pricey bill of technical debt: When and by whom will it be paid? In: **2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [*S. l.*: *s. n.*], 2017. p. 13–23.

BESKER, T.; MARTINI, A.; BOSCH, J. Managing architectural technical debt: A unified model and systematic literature review. **Journal of Systems and Software**, Elsevier, v. 135, p. 1–16, 2018.

BESKER, T.; MARTINI, A.; BOSCH, J. Technical debt triage in backlog management. In: IEEE. **2019 IEEE/ACM International Conference on Technical Debt (TechDebt)**. [*S. l.*], 2019. p. 13–22.

BIAGGI, A.; FONTANA, F. A.; ROVEDA, R. An architectural smells detection tool for c and c++ projects. In: IEEE. **2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**. [*S. l.*], 2018. p. 417–420.

BRITTO, R.; ŠMITE, D.; DAMM, L.-O. Experiences from measuring learning and performance in large-scale distributed software development. In: ACM. **Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement**. [*S. l.*], 2016. p. 17.

BRITTO, R.; SMITE, D.; DAMM, L.-O. Software architects in large-scale distributed projects: An ericsson case study. **IEEE Software**, IEEE, v. 33, n. 6, p. 48–55, 2016.

BROOKS, F. P. The mythical man-month. **Datamation**, v. 20, n. 12, p. 44–52, 1974.

BROWN, N.; CAI, Y.; GUO, Y.; KAZMAN, R.; KIM, M.; KRUCHTEN, P.; LIM, E.; MACCORMACK, A.; NORD, R.; OZKAYA, I. *et al.* Managing technical debt in software-reliant systems. In: **Proceedings of the FSE/SDP workshop on Future of software engineering research**. [*S. l.*: *s. n.*], 2010. p. 47–52.

BROWN, T.; MANN, B.; RYDER, N.; SUBBIAH, M.; KAPLAN, J. D.; DHARIWAL, P.; NEELAKANTAN, A.; SHYAM, P.; SASTRY, G.; ASKELL, A. *et al.* Language models are few-shot learners. **Advances in neural information processing systems**, v. 33, p. 1877–1901, 2020.

BROWN, W. H.; MALVEAU, R. C.; MCCORMICK, H. W. S.; MOWBRAY, T. J. **AntiPatterns: refactoring software, architectures, and projects in crisis**. [*S. l.*]: John Wiley & Sons, Inc., 1998.

CAI, Y.; KAZMAN, R. Dv8: automated architecture analysis tool suites. In: IEEE. **2019 IEEE/ACM International Conference on Technical Debt (TechDebt)**. [*S. l.*], 2019. p. 53–54.

CANFORA, G.; CERULO, L.; PENTA, M. D. Identifying changed source code lines from version repositories. In: IEEE. **Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)**. [*S. l.*], 2007. p. 14–14.

CARMEL, E.; AGARWAL, R. Tactical approaches for alleviating distance in global software development. **IEEE Software**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 18, n. 2, p. 22–29, 2001.

CARPIO, P. M. del. Identification of architectural technical debt: An analysis based on naming patterns. In: IEEE. **2016 8th Euro American Conference on Telematics and Information Systems (EATIS)**. [*S. l.*], 2016. p. 1–8.

CASEY, V.; RICHARDSON, I. Uncovering the reality within virtual software teams. In: **Proceedings of the 2006 International Workshop on Global Software Development for the Practitioner**. New York, NY, USA: ACM, 2006. (GSD '06), p. 66–72. ISBN 1-59593-404-9.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. **IEEE Transactions on software engineering**, IEEE, v. 20, n. 6, p. 476–493, 1994.

COMMITTEE, S. . S. C. *et al.* Ieee standard glossary of software engineering terminology (ieee std 610.12-1990). **New York, NY: The Institute of Electrical and Electronics Engineers**, 1990.

CONCHúIR, E.; ÅGERFALK, P. J.; HOLMSTROM, H.; FITZGERALD, B. Global software development: Where are the benefits? **Communications of the ACM**, ACM, New York, NY, USA, v. 52, n. 8, p. 127–131, aug 2009.

CUNNINGHAM, W. The wycash portfolio management system. In: **Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)**. New York, NY, USA: ACM, 1992. (OOPSLA '92), p. 29–30. ISBN 0-89791-610-7.

CURTIS, B.; SAPPIDI, J.; SZYNKARSKI, A. Estimating the principal of an application's technical debt. **IEEE software**, IEEE, v. 29, n. 6, p. 34–42, 2012.

DAS, S.; LUTTERS, W. G.; SEAMAN, C. B. Understanding documentation value in software maintenance. In: **Proceedings of the 2007 Symposium on Computer Human Interaction for the Management of Information Technology**. New York, NY, USA: ACM, 2007. (CHIMIT '07). ISBN 978-1-59593-635-6.

DIGKAS, G.; LUNGU, M.; AVGERIOU, P.; CHATZIGEORGIOU, A.; AMPATZOGLOU, A. How do developers fix issues and pay back technical debt in the apache ecosystem? In: **2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [*S. l.*]: IEEE, 2018.

DIGKAS, G.; LUNGU, M.; AVGERIOU, P.; CHATZIGEORGIOU, A.; AMPATZOGLOU, A. How do developers fix issues and pay back technical debt in the apache ecosystem? In: IEEE. **2018 IEEE 25th International Conference on software analysis, evolution and reengineering (SANER)**. [*S. l.*], 2018. p. 153–163.

DIKERT, K.; PAASIVAARA, M.; LASSENIUS, C. Challenges and success factors for large-scale agile transformations: A systematic literature review. **Journal of Systems and Software**, Elsevier, v. 119, p. 87–108, 2016.

DIKERT, K.; PAASIVAARA, M.; LASSENIUS, C. Challenges and success factors for large-scale agile transformations: A systematic literature review. **Journal of Systems and Software**, Elsevier Inc., v. 119, p. 87–108, 2016.

ELIASSON, U.; MARTINI, A.; KAUFMANN, R.; ODEH, S. Identifying and visualizing architectural debt and its efficiency interest in the automotive domain: A case study. In: IEEE. **2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)**. [*S. l.*], 2015. p. 33–40.

ELISH, M. O.; AL-KHIATY, M. A.-R. A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software. **Journal of Software: Evolution and Process**, Wiley Online Library, v. 25, n. 5, p. 407–437, 2013.

ESPINOSA, J. A.; NAN, N.; CARMEL, E. Do gradations of time zone separation make a difference in performance? a first laboratory study. In: **Second IEEE International Conference on Global Software Engineering - ICGSE'07.** [*S. l.*: *s. n.*], 2007. p. 12–22.

FALESSI, D.; SHAW, M. A.; SHULL, F.; MULLEN, K.; STEIN, M. Practical considerations, challenges, and requirements of tool-support for managing technical debt. In: **Proceedings of the 4th International Workshop on Managing Technical Debt**. Piscataway, NJ, USA: IEEE Press, 2013. (MTD '13), p. 16–19. ISBN 978-1-4673-6443-0.

FENG, Q.; CAI, Y.; KAZMAN, R.; CUI, D.; LIU, T.; FANG, H. Active hotspot: An issue-oriented model to monitor software evolution and degradation. In: IEEE. **2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [*S. l.*], 2019. p. 986–997.

FLEISS, J. L.; LEVIN, B.; PAIK, M. C. **Statistical methods for rates and proportions**. [*S. l.*]: john wiley & sons, 2013.

FONTANA, F. A.; PIGAZZINI, I.; ROVEDA, R.; ZANONI, M. Automatic detection of instability architectural smells. In: IEEE. **2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [*S. l.*], 2016. p. 433–437.

FONTANA, F. A.; PIGAZZINI, I.; ROVEDA, R.; TAMBURRI, D.; ZANONI, M.; NITTO, E. D. Arcan: A tool for architectural smells detection. In: IEEE. **2017 IEEE International Conference on Software Architecture Workshops (ICSAW)**. [*S. l.*], 2017. p. 282–285.

FONTANA, F. A.; ROVEDA, R.; ZANONI, M. Technical debt indexes provided by tools: a preliminary discussion. In: IEEE. **2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)**. [*S. l.*], 2016. p. 28–31.

FONTANA, F. A.; ROVEDA, R.; ZANONI, M. Tool support for evaluating architectural debt of an existing system: An experience report. In: **Proceedings of the 31st Annual ACM Symposium on Applied Computing**. [*S. l.*: *s. n.*], 2016. p. 1347–1349.

FOWLER, M. **Refactoring: improving the design of existing code**. [*S. l.*]: Addison-Wesley Professional, 2018.

FOX, J. **Applied regression analysis and generalized linear models**. [*S. l.*]: Sage Publications, 2015.

GANESH, S.; SHARMA, T.; SURYANARAYANA, G. Towards a principle-based classification of structural design smells. **J. Object Technol.**, v. 12, n. 2, p. 1–1, 2013.

GIL, J.; GOLDSTEIN, M.; MOSHKOVICH, D. An empirical investigation of changes in some software properties over time. In: IEEE. **2012 9th IEEE Working Conference on Mining Software Repositories (MSR)**. [*S. l.*], 2012. p. 227–236.

GOLDEN, J. M. Transformation patterns for curing the human causes of technical debt. **Cutter IT Journal**, v. 23, n. 10, p. 30, 2010.

GRAYLIN, J.; HALE, J. E.; SMITH, R. K.; DAVID, H.; KRAFT, N. A.; CHARLES, W. *et al.* Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship. **Journal of Software Engineering and Applications**, Scientific Research Publishing, v. 2, n. 03, p. 137, 2009.

GUO, Y.; SPINOLA, R.; SEAMAN, C. Exploring the costs of technical debt management – a case study. **Empirical Software Engineering**, Springer New York LLC, v. 21, n. 1, p. 159–182, 2016. ISSN 13823256. Cited By 7.

HARTER, D. E.; KRISHNAN, M. S.; SLAUGHTER, S. A. Effects of process maturity on quality, cycle time, and effort in software product development. **Manage. Sci.**, INFORMS, Institute for Operations Research and the Management Sciences (INFORMS), Linthicum, Maryland, USA, v. 46, n. 4, p. 451–466, apr 2000. ISSN 0025-1909.

HASSAN, A. E. The road ahead for mining software repositories. In: IEEE. **2008 Frontiers of Software Maintenance**. [*S. l.*], 2008. p. 48–57.

HASSAN, A. E.; HOLT, R. C. Predicting change propagation in software systems. In: IEEE. **20th IEEE International Conference on Software Maintenance, 2004. Proceedings.** [*S. l.*], 2004. p. 284–293.

HEIKKILA, V.; PAASIVAARA, M.; LASSSENIUS, C.; DAMIAN, D.; ENGBLOM, C. Managing the requirements flow from strategy to release in large-scale agile development: a case study at ericsson. **Empirical Software Engineering**, Springer New York LLC, v. 22, n. 6, p. 2892–2936, 2017. ISSN 13823256. Cited By 1.

HEMMATI, H.; NADI, S.; BAYSAL, O.; KONONENKO, O.; WANG, W.; HOLMES, R.; GODFREY, M. W. The msr cookbook: Mining a decade of research. In: IEEE. **2013 10th Working Conference on Mining Software Repositories (MSR)**. [*S. l.*], 2013. p. 343–352.

HERBSLEB, J. D.; MOCKUS, A. An empirical study of speed and communication in globally distributed software development. **IEEE Transactions on Software Engineering**, v. 29, n. 6, p. 481–494, June 2003.

HERBSLEB, J. D.; MOITRA, D. Global software development. **IEEE software**, IEEE, v. 18, n. 2, p. 16–20, 2001.

HOLVITIE, J.; LEPPANEN, V.; HYRYNSALMI, S. Technical debt and the effect of agile software development practices on it-an industry practitioner survey. In: IEEE. **2014 Sixth International Workshop on Managing Technical Debt**. [*S. l.*], 2014. p. 35–42.

IAMMARINO, M.; ZAMPETTI, F.; AVERSANO, L.; PENTA, M. D. An empirical study on the co-occurrence between refactoring actions and self-admitted technical debt removal. **Journal of Systems and Software**, Elsevier, v. 178, p. 110976, 2021.

JOHNSON, B.; SHNEIDERMAN, B. **Tree-maps: A space filling approach to the visualization of hierarchical information structures**. [*S. l.*], 1998.

KARWIN, B. **SQL antipatterns: avoiding the pitfalls of database programming**. [*S. l.*]: Pragmatic Bookshelf, 2010.

KAZMAN, R.; CAI, Y.; MO, R.; FENG, Q.; XIAO, L.; HAZIYEV, S.; FEDAK, V.; SHAPOCHKA, A. A case study in locating the architectural roots of technical debt. In: IEEE. **2015 IEEE/ACM 37th IEEE International Conference on Software Engineering**. [*S. l.*], 2015. v. 2, p. 179–188.

KAZMAN, R.; CAI, Y.; MO, R.; FENG, Q.; XIAO, L.; HAZIYEV, S.; FEDAK, V.; SHAPOCHKA, A. A case study in locating the architectural roots of technical debt. In: **2015 IEEE/ACM 37th IEEE International Conference on Software Engineering**. [*S. l.*]: IEEE, 2015.

KHOMH, F.; PENTA, M. D.; GUEHENEUC, Y.-G. An exploratory study of the impact of code smells on software change-proneness. In: IEEE. **2009 16th Working Conference on Reverse Engineering**. [*S. l.*], 2009. p. 75–84.

KHOMH, F.; VAUCHER, S.; GUÉHÉNEUC, Y.-G.; SAHRAOUI, H. Bdtex: A gqm-based bayesian approach for the detection of antipatterns. **Journal of Systems and Software**, Elsevier, v. 84, n. 4, p. 559–572, 2011.

KITCHENHAM, B.; CHARTERS, S. Guidelines for performing systematic literature reviews in software engineering. Citeseer, 2007.

KNODEL, J.; LINDVALL, M.; MUTHIG, D.; NAAB, M. Static evaluation of software architectures. In: IEEE. **Conference on Software Maintenance and Reengineering (CSMR'06)**. [*S. l.*], 2006. p. 10–pp.

KRUCHTEN, P.; NORD, R.; OZKAYA, I. **Managing Technical Debt: Reducing Friction in Software Development**. [*S. l.*]: Addison-Wesley Professional, 2019.

KRUCHTEN, P.; NORD, R. L.; OZKAYA, I. Technical debt: From metaphor to theory and practice. **Ieee software**, IEEE, v. 29, n. 6, p. 18–21, 2012.

LAWSON, R. Implications of surface temperatures in the diagnosis of breast cancer. **Canadian Medical Association Journal**, Canadian Medical Association, v. 75, n. 4, p. 309, 1956.

LI, Y.; SOLIMAN, M.; AVGERIOU, P. Automatic identification of self-admitted technical debt from four different sources. **Empirical Software Engineering**, Springer, v. 28, n. 3, p. 1–38, 2023.

LI, Z.; AVGERIOU, P.; LIANG, P. A systematic mapping study on technical debt and its management. **Journal of Systems and Software**, Elsevier, v. 101, p. 193–220, 2015.

LI, Z.; AVGERIOU, P.; LIANG, P. A systematic mapping study on technical debt and its management. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 101, n. C, p. 193–220, mar 2015. ISSN 0164-1212.

LI, Z.; LIANG, P.; AVGERIOU, P. Architectural debt management in value-oriented architecting. In: **Economics-Driven Software Architecture**. [*S. l.*]: Elsevier, 2014. p. 183–204.

LI, Z.; LIANG, P.; AVGERIOU, P. Architectural technical debt identification based on architecture decisions and change scenarios. In: IEEE. **2015 12th Working IEEE/IFIP Conference on Software Architecture**. [*S. l.*], 2015. p. 65–74.

LI, Z.; LIANG, P.; AVGERIOU, P. Architecture viewpoints for documenting architectural technical debt. In: **Software Quality Assurance**. [*S. l.*]: Elsevier, 2016. p. 85–132.

LI, Z.; LIANG, P.; AVGERIOU, P.; GUELFI, N.; AMPATZOGLOU, A. An empirical investigation of modularity metrics for indicating architectural technical debt. In: **Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures**. [*S. l.*: *s. n.*], 2014. p. 119–128.

LUDWIG, J.; XU, S.; WEBBER, F. Compiling static software metrics for reliability and maintainability from github repositories. In: IEEE. **2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)**. [*S. l.*], 2017. p. 5–9.

MAIGA, A.; ALI, N.; BHATTACHARYA, N.; SABANÉ, A.; GUÉHÉNEUC, Y.-G.; ANTONIOL, G.; AÏMEUR, E. Support vector machines for anti-pattern detection. In: IEEE. **2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering**. [*S. l.*], 2012. p. 278–281.

MALDONADO, E. d. S.; SHIHAB, E. Detecting and quantifying different types of self-admitted technical debt. In: IEEE. **2015 IEEE 7Th international workshop on managing technical debt (MTD)**. [*S. l.*], 2015. p. 9–15.

MANTYLA, M.; VANHANEN, J.; LASSENIUS, C. A taxonomy and an initial empirical study of bad smells in code. In: IEEE. **International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.** [*S. l.*], 2003. p. 381–384.

MARINESCU, R. Measurement and quality in object-oriented design. In: IEEE. **21st IEEE International Conference on Software Maintenance (ICSM'05)**. [*S. l.*], 2005. p. 701–704.

MARTINI, A.; BESKER, T.; BOSCH, J. The introduction of technical debt tracking in large companies. In: IEEE. **2016 23rd Asia-Pacific Software Engineering Conference (APSEC)**. [*S. l.*], 2016. p. 161–168.

MARTINI, A.; BOSCH, J. The danger of architectural technical debt: Contagious debt and vicious circles. In: IEEE. **2015 12th Working IEEE/IFIP Conference on Software Architecture**. [*S. l.*], 2015. p. 1–10.

MARTINI, A.; BOSCH, J. Towards prioritizing architecture technical debt: information needs of architects and product owners. In: IEEE. **2015 41St euromicro conference on software engineering and advanced applications**. [*S. l.*], 2015. p. 422–429.

MARTINI, A.; BOSCH, J. An empirically developed method to aid decisions on architectural technical debt refactoring: Anacondebt. In: IEEE. **2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)**. [*S. l.*], 2016. p. 31–40.

MARTINI, A.; BOSCH, J. An empirically developed method to aid decisions on architectural technical debt refactoring: Anacondebt. In: IEEE. **2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)**. [*S. l.*], 2016. p. 31–40.

MARTINI, A.; BOSCH, J. On the interest of architectural technical debt: Uncovering the contagious debt phenomenon. **Journal of Software: Evolution and Process**, Wiley Online Library, v. 29, n. 10, p. e1877, 2017.

MARTINI, A.; BOSCH, J.; CHAUDRON, M. Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. **Information and Software Technology**, Elsevier, v. 67, p. 237–253, 2015.

MARTINI, A.; FONTANA, F. A.; BIAGGI, A.; ROVEDA, R. Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company. In: SPRINGER. **European Conference on Software Architecture**. [*S. l.*], 2018. p. 320–335.

MARTINI, A.; SIKANDER, E.; MADLANI, N. A semi-automated framework for the identification and estimation of architectural technical debt: A comparative case-study on the modularization of a software component. **Information and Software Technology**, Elsevier, v. 93, p. 264–279, 2018.

MARTINI, A.; SIKANDER, E.; MEDLANI, N. Estimating and quantifying the benefits of refactoring to improve a component modularity: a case study. In: IEEE. **2016 42th Euromicro conference on software engineering and advanced applications (SEAA)**. [*S. l.*], 2016. p. 92–99.

MAYRING, P. Qualitative content analysis: theoretical foundation, basic procedures and software solution. AUT, 2014.

MCCABE, T. J. A complexity measure. **IEEE Transactions on software Engineering**, IEEE, n. 4, p. 308–320, 1976.

MOHA, N.; GUÉHÉNEUC, Y.-G.; DUCHIEN, L.; MEUR, A.-F. L. Decor: A method for the specification and detection of code and design smells. **IEEE Transactions on Software Engineering**, IEEE, v. 36, n. 1, p. 20–36, 2009.

NORD, R. L.; OZKAYA, I.; KRUCHTEN, P.; GONZALEZ-ROJAS, M. In search of a metric for managing architectural technical debt. In: IEEE. **2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture**. [*S. l.*], 2012. p. 91–100.

OFFERMANN, P.; LEVINA, O.; SCHÖNHERR, M.; BUB, U. Outline of a design science research process. In: **Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology**. [*S. l.*: *s. n.*], 2009. p. 1–11.

OSPINA, S.; VERDECCHIA, R.; MALAVOLTA, I.; LAGO, P. Atdx: A tool for providing a data-driven overview of architectural technical debt in software-intensive systems. In: **European Conference on Software Architecture**. [*S. l.*: *s. n.*], 2021.

OUNI, A.; KULA, R. G.; KESSENTINI, M.; INOUE, K. Web service antipatterns detection using genetic programming. In: **Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation**. [*S. l.*: *s. n.*], 2015. p. 1351–1358.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; POSHYVANYK, D.; LUCIA, A. D. Mining version histories for detecting code smells. **IEEE Transactions on Software Engineering**, IEEE, v. 41, n. 5, p. 462–489, 2014.

PEFFERS, K.; TUUNANEN, T.; ROTHENBERGER, M. A.; CHATTERJEE, S. A design science research methodology for information systems research. **Journal of management information systems**, Taylor & Francis, v. 24, n. 3, p. 45–77, 2007.

PÉREZ, B. A semiautomatic approach to identify architectural technical debt from heterogeneous artifacts. In: SPRINGER. **Software Architecture: 14th European Conference, ECSA 2020 Tracks and Workshops, L'Aquila, Italy, September 14–18, 2020, Proceedings 14**. [*S. l.*], 2020. p. 5–16.

PEREZ, B.; CORREAL, D.; ASTUDILLO, H. A proposed model-driven approach to manage architectural technical debt life cycle. In: IEEE. **2019 IEEE/ACM International Conference on Technical Debt (TechDebt)**. [*S. l.*], 2019. p. 73–77.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. **ACM SIGSOFT Software engineering notes**, ACM New York, NY, USA, v. 17, n. 4, p. 40–52, 1992.

PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. Systematic mapping studies in software engineering. In: **12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12**. [*S. l.*: *s. n.*], 2008. p. 1–10.

POTDAR, A.; SHIHAB, E. An exploratory study on self-admitted technical debt. In: IEEE. **2014 IEEE International Conference on Software Maintenance and Evolution**. [*S. l.*], 2014. p. 91–100.

RAMASUBBU, N.; CATALDO, M.; BALAN, R. K.; HERBSLEB, J. D. Configuring global software teams: A multi-company analysis of project productivity, quality, and profits. In: **Proceedings of the 33rd International Conference on Software Engineering - ICSE'11**. [*S. l.*: *s. n.*], 2011. p. 261–270.

RANTALA, L.; MÄNTYLÄ, M. Predicting technical debt from commit contents: reproduction and extension with automated feature selection. **Software Quality Journal**, Springer, v. 28, p. 1551–1579, 2020.

RANTALA, L.; MÄNTYLÄ, M.; LO, D. Prevalence, contents and automatic detection of kl-satd. In: IEEE. **2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**. [*S. l.*], 2020. p. 385–388.

RIOS, N.; NETO, M. G. de M.; SPÍNOLA, R. O. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. **Information and Software Technology**, Elsevier BV, v. 102, p. 117–145, oct 2018.

ROBLES, G. Replicating msr: A study of the potential replicability of papers published in the mining software repositories proceedings. In: IEEE. **2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)**. [*S. l.*], 2010. p. 171–180.

ROVEDA, R.; FONTANA, F. A.; PIGAZZINI, I.; ZANONI, M. Towards an architectural debt index. In: IEEE. **2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**. [*S. l.*], 2018. p. 408–416.

RUNESON, P.; HOST, M.; RAINER, A.; REGNELL, B. **Case Study Research in Software Engineering: Guidelines and Examples**. [*S. l.*]: John Wiley Sons, 2012. ISBN 978-1118104354.

SAS, D.; AVGERIOU, P.; FONTANA, F. A. Investigating instability architectural smells evolution: an exploratory case study. In: IEEE. **2019 IEEE International Conference on software maintenance and evolution (ICSME)**. [*S. l.*], 2019. p. 557–567.

SAS, D.; AVGERIOU, P.; PIGAZZINI, I.; FONTANA, F. A. On the relation between architectural smells and source code changes. **Journal of Software: Evolution and Process**, Wiley Online Library, v. 34, n. 1, p. e2398, 2022.

SAS, D.; AVGERIOU, P.; UYUMAZ, U. On the evolution and impact of architectural smells—an industrial case study. **Empirical Software Engineering**, Springer, v. 27, n. 4, p. 1–45, 2022.

SAS, D.; PIGAZZINI, I.; AVGERIOU, P.; FONTANA, F. A. The perception of architectural smells in industrial practice. **Ieee software**, IEEE, v. 38, n. 6, p. 35–41, 2021.

SEAMAN, C.; GUO, Y. Measuring and monitoring technical debt. In: **Advances in Computers**. [*S. l.*]: Elsevier, 2011. v. 82, p. 25–46.

SHARMA, T.; FRAGKOULIS, M.; SPINELLIS, D. Does your configuration code smell? In: IEEE. **2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)**. [*S. l.*], 2016. p. 189–200.

SHARMA, T.; SINGH, P.; SPINELLIS, D. An empirical investigation on the relationship between design and architecture smells. **Empirical Software Engineering**, Springer, v. 25, n. 5, p. 4020–4068, 2020.

SHARMA, T.; SPINELLIS, D. A survey on software smells. **Journal of Systems and Software**, Elsevier, v. 138, p. 158–173, 2018.

SHNEIDERMAN, B.; WATTENBERG, M. Ordered treemap layouts. In: IEEE. **IEEE Symposium on Information Visualization, 2001. INFOVIS 2001.** [*S. l.*], 2001. p. 73–78.

SIERRA, G.; SHIHAB, E.; KAMEI, Y. A survey of self-admitted technical debt. **Journal of Systems and Software**, Elsevier, v. 152, p. 70–82, 2019.

SIERRA, G.; TAHMID, A.; SHIHAB, E.; TSANTALIS, N. Is self-admitted technical debt a good indicator of architectural divergences? In: IEEE. **2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [*S. l.*], 2019. p. 534–543.

SOUSA, A.; ROCHA, L.; BRITTO, R. Architectural technical debt-a systematic mapping study. In: **Proceedings of the XXXVII Brazilian Symposium on Software Engineering**. [*S. l.*: *s. n.*], 2023. p. 196–205.

SOUSA, A.; ROCHA, L.; BRITTO, R.; GONG, Z.; LYU, F. Technical debt in large-scale distributed projects: An industrial case study. In: **2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [*S. l.*: *s. n.*], 2021. p. 590–594.

SPADINI, D.; ANICHE, M.; BACCHELLI, A. Pydriller: Python framework for mining software repositories. In: **Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018**. New York, New York, USA: ACM Press, 2018. p. 908–911. ISBN 9781450355735. Disponível em: http://dl.acm.org/citation.cfm?doid=3236024.3264598.

STEIDL, D.; HUMMEL, B.; JUERGENS, E. Incremental origin analysis of source code files. In: **Proceedings of the 11th Working Conference on Mining Software Repositories**. [*S. l.*: *s. n.*], 2014. p. 42–51.

SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. **Refactoring for software design smells: managing technical debt**. [*S. l.*]: Morgan Kaufmann, 2014.

SZŐKE, G.; ANTAL, G.; NAGY, C.; FERENC, R.; GYIMÓTHY, T. Empirical study on refactoring large-scale industrial systems and its effects on maintainability. **Journal of Systems and Software**, Elsevier, v. 129, p. 107–126, 2017.

TAMBURRI, D. A. Software architecture social debt: managing the incommunicability factor. **IEEE Transactions on Computational Social Systems**, IEEE, v. 6, n. 1, p. 20–37, 2019.

TAMBURRI, D. A.; KRUCHTEN, P.; LAGO, P.; VLIET, H. van. What is social debt in software engineering? In: IEEE. **2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)**. [*S. l.*], 2013. p. 93–96.

TAYLOR, R. N.; MEDVIDOVIC, N.; DASHOFY, E. **Software architecture: foundations, theory, and practice**. [*S. l.*]: John Wiley & Sons, 2009.

TOLEDO, S. S. de; MARTINI, A.; SJOBERG, D. I. Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study. **Journal of Systems and Software**, Elsevier, v. 177, p. 110968, 2021.

TOLEDO, S. S. de; MARTINI, A.; SJOBERG, D. I.; PRZYBYSZEWSKA, A.; FRANDSEN, J. S. Reducing incidents in microservices by repaying architectural technical debt. In: IEEE. **2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**. [*S. l.*], 2021. p. 196–205.

TOM, E.; AURUM, A.; VIDGEN, R. An exploration of technical debt. **Journal of Systems and Software**, Elsevier, v. 86, n. 6, p. 1498–1516, 2013.

TORNHILL, A. Assessing technical debt in automated tests with codescene. In: IEEE. **2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [*S. l.*], 2018. p. 122–125.

TORNHILL, A. Your code as a crime scene. The Pragmatic Bookshelf,, 2019.

UMEMURA, T. K. **Uma ferramenta para monitoramento da entropia de mudança e sua relação com métricas de software**. Dissertação (B.S. thesis) – Universidade Tecnológica Federal do Paraná, 2017.

VERDECCHIA, R. Identifying architectural technical debt in android applications through automated compliance checking. In: IEEE. **2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)**. [*S. l.*], 2018. p. 35–36.

VERDECCHIA, R.; KRUCHTEN, P.; LAGO, P. Architectural technical debt: A grounded theory. In: SPRINGER. **European Conference on Software Architecture**. [*S. l.*], 2020. p. 202–219.

VERDECCHIA, R.; KRUCHTEN, P.; LAGO, P.; MALAVOLTA, I. Building and evaluating a theory of architectural technical debt in software-intensive systems. **Journal of Systems and Software**, Elsevier, v. 176, p. 110925, 2021.

VERDECCHIA, R.; LAGO, P.; MALAVOLTA, I.; OZKAYA, I. Atdx: Building an architectural technical debt index. In: **ENASE**. [*S. l.*: *s. n.*], 2020. p. 531–539.

VERDECCHIA, R.; MALAVOLTA, I.; LAGO, P. Architectural technical debt identification: The research landscape. In: IEEE. **2018 IEEE/ACM International Conference on Technical Debt (TechDebt)**. [*S. l.*], 2018. p. 11–20.

VERDECCHIA, R.; MALAVOLTA, I.; LAGO, P.; OZKAYA, I. Empirical evaluation of an architectural technical debt index in the context of the apache and onap ecosystems. **PeerJ Computer Science**, PeerJ Inc., v. 8, p. e833, 2022.

VERDECCHIA, R. *et al.* Architectural technical debt: Identification and management. Gran Sasso Science Institute, 2021.

WEHAIBI, S.; SHIHAB, E.; GUERROUJ, L. Examining the impact of self-admitted technical debt on software quality. In: IEEE. **2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)**. [*S. l.*], 2016. v. 1, p. 179–188.

WETTEL, R.; LANZA, M. Codecity: 3d visualization of large-scale software. In: **Companion of the 30th international conference on Software engineering**. [*S. l.*: *s. n.*], 2008. p. 921–922.

WIERINGA, R. J. **Design science methodology for information systems and software engineering**. [*S. l.*]: Springer, 2014.

WITTE, R. S.; WITTE, J. S. **Statistics**. [*S. l.*]: John Wiley & Sons, 2017.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**. [*S. l.*]: Springer Science & Business Media, 2012.

XIAO, L.; CAI, Y.; KAZMAN, R.; MO, R.; FENG, Q. Detecting the locations and predicting the costs of compound architectural debts. **IEEE Transactions on Software Engineering**, IEEE, 2021.

YAMASHITA, A. Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. **Empirical Software Engineering**, Springer, v. 19, n. 4, p. 1111–1143, 2014.

YAO, Y.; HUANG, S.; JIE, L.; LIU, X. ming. Structural characteristic of large-scale software development network. In: **2010 2nd International Conference on Computer Engineering and Technology**. [*S. l.*]: IEEE, 2010.

ZALEWSKI, A. Risk appetite in architectural decision-making. In: IEEE. **2017 IEEE International Conference on Software Architecture Workshops (ICSAW)**. [*S. l.*], 2017. p. 149–152.

ZHANG, F.; MOCKUS, A.; KEIVANLOO, I.; ZOU, Y. Towards building a universal defect prediction model. In: **Proceedings of the 11th Working Conference on Mining Software Repositories**. [*S. l.*: *s. n.*], 2014. p. 182–191.

ZHOU, Y.; MURESANU, A. I.; HAN, Z.; PASTER, K.; PITIS, S.; CHAN, H.; BA, J. Large language models are human-level prompt engineers. **arXiv preprint arXiv:2211.01910**, 2022.

ZITZEWITZ, A. von. Mitigating technical and architectural debt with sonargraph. In: IEEE. **2019 IEEE/ACM International Conference on Technical Debt (TechDebt)**. [*S. l.*], 2019. p. 66–67.

APPENDIX A – SELECTED PAPERS SET 1

We show the SMS selected papers which are available on table 25, and table 26.

Table 25 – Selected Papers Set 1

sp	Citation	Title
SP1	Perez et al. 2019	A Proposed Model-driven Approach to Manage Architectural Technical Debt Life Cycle
SP2	Martini et al 2017	A semi-automated framework for the identification and estimation of Architectural Technical Debt: A comparative case-study on the modularization of a software component
SP3	Roveda et al. 2018	Towards an Architectural Debt Index
SP4	Snipes et al. 2018	A Case Study of the Effects of Architecture Debt on Software Evolution Effort
SP5	Nayebi et al. 2019	A Longitudinal Study of Identifying and Paying Down architecture debt
SP6	Feng et al. 2019	Active Hotspot: An Issue-Oriented Model to Monitor Software Evolution and Degradation
SP7	Li et al. 2014	An Empirical Investigation of Modularity Metrics for Indicating architectural technical debt
SP8	Martini et al. 2016	An Empirically Developed Method to Aid Decisions on Architectural Technical Debt Refactoring: AnaConDebt
SP9	Xiao et al. 2016	Identifying and Quantifying Architectural Debt
SP10	Verdecchia 2018	Identifying Architectural Technical Debt in Android Applications through Automated Compliance Checking
SP11	Zitzweitz et al. 2019	Mitigating Technical and Architectural Debt with Sonargraph Using static analysis to enforce architectural constraints
SP12	Martini and Bosch 2017	On the interest of architectural technical debt: Uncovering the contagious debt phenomenon
SP13	Fontana et al. 2016	Tool support for evaluating architectural debt of an existing system: An experience report
SP14	Li et al. 2015	Architectural Technical Debt Identification based on Architecture Decisions and Change Scenarios
SP15	Martini et al. 2014	Architecture Technical Debt: Understanding Causes and a Qualitative Model
SP16	Martini et al. 2016	Estimating and Quantifying the Benefits of Refactoring to Improve a Component Modularity: a Case Study
SP17	Martini et al. 2018	Identifying and Prioritizing Architectural Debt Through Architectural Smells: A Case Study in a Large Software Company
SP18	Cai and Kazman 2016	Software Architecture Health Monitor
SP19	Kazman et al. 2015	A Case Study in Locating the Architectural Roots of Technical Debt
SP20	Martini and Bosch 2015	Towards Prioritizing Architecture Technical Debt: Information Needs of Architects and Product Owners
SP21	Tommasel 2019	Applying Social Network Analysis Techniques to Architectural Smell Prediction
SP22	Nord et al. 2012	In Search of a Metric for Managing Architectural Technical Debt
SP23	Cai and Kazman 2019	DV8: Automated Architecture Analysis Tool Suites
SP24	Skiada et al. 2018	Exploring the Relationship between Software Modularity and Technical Debt
SP25	Eliasson et al. 2015	Identifying and Visualizing Architectural Debt and Its Efficiency Interest in the Automotive Domain: A Case Study
SP26	Fontana et al. 2016	Technical Debt Indexes provided by tools: a preliminary discussion
SP27	MacComark et al. 2016	Technical debt and system architecture: The impact of coupling on defect-related activity
SP28	Sas et al. 2019	Investigating Instability Architectural Smells Evolution: An Exploratory Case Study
SP29	Fontana et al. 2019	PageRank and criticality of architectural smells
SP30	Spinellis et al. 2019	Evolution of the Unix System Architecture: An Exploratory Case Study
SP31	Izurieta et al. 2018	A position study to investigate technical debt associated with security weaknesses
SP32	Besker et al. 2017	The pricey bill of technical debt: When and by whom will it be paid?
SP33	Ampatzoglou et al. 2016	The Perception of Technical Debt in the Embedded Systems Domain: An Industrial Case Study
SP34	Martini et al. 2016	The Introduction of Technical Debt Tracking in Large Companies
SP35	Aaramaa et al 2017	Requirements volatility in software architecture design: An exploratory case study
SP36	Martini et al 2015	Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study
SP37	Cai el al 2018	Design Rule Spaces: A New Model for Representing and Analyzing Software Architecture
SP38	Bogner et al 2019	Assuring the Evolvability of Microservices: Insights into Industry Practices and Challenges
SP39	Ludwig et al. 2017	Compiling Static Software Metrics for Reliability and Maintainability from GitHub Repositories
SP40	Curtis et al 2012	Estimating the principal of an applications technical debt
SP41	Kumar et al 2018	Exploring multilateral cloud computing security architectural design debt in terms of technical debt
SP42	Hanssen et al 2019	Identifying Scalability Debt in Open Systems
SP43	Sierra et al 2019	Is Self-Admitted Technical Debt a Good Indicator of Architectural Divergences?
SP44	Bogner et al 2018	Limiting technical debt with maintainability assurance...
SP45	Zalewski 2017	Risk Appetite in Architectural Decision-Making

APPENDIX B – SELECTED PAPERS SET 2

Table 26 – Selected Papers Set 2

sp	Citation	Title
SP46	Tamburri 2019	Software Architecture Social Debt: Managing the Incommunicability Factor
SP47	Verdechia et al. 2020	Architectural Technical Debt: A Grounded Theory
SP48	Pujols et al. 2020	Skuld: a self-learning tool for impact-driven technical debt management
SP49	Molnar et al. 2020	Long-Term Evaluation of Technical Debt in Open-Source Software
SP50	Janes et al. 2020	Towards an Approach to Identify Obsolete Features based on Importance and Technical Debt
SP51	Maikantis et al. 2020	Software Architecture Reconstruction via a Genetic Algorithm: Applying the Move Class Refactoring
SP52	San Martín et al 2020	Characterizing Architectural Drifts of Adaptive Systems
SP53	de Toledo el al. 2020	Improving agility by managing shared libraries in microservices
SP54	Rademacher et al. 2020	A modeling method for systematic architecture reconstruction of microservice-based software systems
SP55	Larrucea et al. 2020	Managing security debt across PLC phases in a VSE context
SP56	Raibulet et al. 2020	A preliminary analysis of self-adaptive systems according to different issues
SP57	Sharma et al. 2020	An empirical investigation on the relationship between design and architecture smells
SP58	Verdecchia et al. 2020	ATDx: Building an Architectural Technical Debt ...
SP59	Perez et al. 2020	A semiautomatic approach to identify architectural ...
SP60	Sas et al. 2021	The perception of Architectural Smells in industrial ...
SP61	Belle et al. 2021	Systematically reviewing the layered architectural ...
SP62	de Toledo et al. 2021	Identifying architectural technical debt, ...
SP63	Xiao et al. 2021	Detecting the Locations and Predicting the Costs ...
SP64	Verdecchia et al. 2021	Building and evaluating a theory of ...
SP65	Ospina et al. 2021	ATDx: A tool for providing a data-driven ...
SP66	de Toledo et al. 2021	Reducing Incidents in Microservices by ...
SP67	Borowa et al. 2021	The influence of cognitive biases on ...
SP68	Sas et al. 2022	On the relation between architectural ...
SP69	De Toledo et al. 2022	Accumulation and Prioritization of ...
SP70	Verdecchia et al. 2022	Empirical evaluation of an archite...
SP71	Pigazzini et al. 2022	Exploiting dynamic analysis for architectural smell detection ...
SP72	Albuquerque et al. 2022	Comprehending the use of intelligent techniques to support ...
SP73	Bacchiega et al. 2022	Microservices smell detection through dynamic analysis ...
SP74	Damaceno et al. 2022	Towards an understanding of technical debt in reference ...
SP75	Amoroso 2022	Architectural Degradation and Technical Debt Dashboards ...
SP76	Snoeck and Wautelet 2022	Agile MERODE: a model-driven software engineering ...
SP77	Capilla et al. 2023	Detecting Architecture Debt in Micro-Service Open-Source ...
SP78	Moldt et al. 2023	Renew: Modularized architecture and new features ...
SP79	Santamaria et al. 2023	Integrating privacy debt and VSE's software ...
SP80	Sas and Paris 2023	An architectural technical debt index based on machine ...
SP81	Lefever et al. 2023	Towards the Assisted Decomposition of Large-Active Files ...
SP82	Funke et al. 2024	A Process for Monitoring the Impact of Architecture Principles ...
SP83	Esposito et al. 2024	On the Correlation between Architectural Smells and Static ...
SP84	Liu et al. 2024	Prevalence and severity of design anti-patterns in open source ...
SP85	Gnoyke et al. 2024	Evolution Patterns of Software-Architecture Smells: An ...