Mitigating Low Bus Factor Risks:A Proactive Approach to Address Single Points of Failure in Software Development Teams

O. V. De Silva
Index No: 20000367

T.P.M De Silva
Index No: 20000375

H.U. Samaliarachchi
Index No: 20001551

Supervisor: Dr. Ajantha Atukorale
Co-Supervisor: Dr. Thilina Halloluwa

University of Colombo, School of Computing
Colombo, Sri Lanka

Submitted in partial fulfillment of the requirements of the
B.Sc(Hons) in Software Engineering 4th Year Project (SCS4223)

April 2025

Declaration

I certify that this dissertation does not incorporate, without acknowledgment, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, be made available for photocopying and for inter-library loans, and for the title and abstract to be made available to outside organizations.

Member Name: O. V. De Silva

Signature of Member: [illegible] Date: 30.06.2025

Member Name: T.P.M. De Silva

Signature of Member: [illegible] Date: 30.06.2025

Member Name: H.U. Samaliarachchi

Signature of Member: [illegible] Date: 30.06.2025

This is to certify that this dissertation is based on the work of O. V. De Silva, T.P.M. De Silva and H.U. Samaliarachchi under my supervision. The thesis has been prepared according to the format stipulated and is of an acceptable standard.

Supervisor's Name: Dr. Ajantha Atukorale

Signature of Supervisor: [illegible] Date: 30.06.2025

Co-supervisor's Name: Dr. Thilina Halloluwa

Signature of Co-supervisor: [illegible] Date: 30.06.2025

Acknowledgement

We would like to express our heartfelt gratitude to our supervisor, Dr. Ajantha Atukorale, and co-supervisor, Dr. Thilina Halloluwa, for their invaluable guidance, encouragement, and unwavering support throughout this research. Despite their demanding schedules, they consistently made time to provide insightful feedback and direction, which significantly contributed to the success of our work.

We are also sincerely thankful to Dr. Randil Pushpananda and Ms. Amali Perera, the course coordinators of the SCS4223 Final Year Project in Software Engineering at the University of Colombo School of Computing, for their continuous support, coordination, and dedication in facilitating this project from start to finish.

Our deep appreciation extends to the industry experts who generously contributed their time and expertise to validate our results, providing us with practical insights and constructive suggestions that enriched the quality and impact of our research. We also thank our fellow colleagues who assisted us during the initial validation phases, offering helpful feedback and encouragement.

Last but certainly not least, we are profoundly grateful to our parents and colleagues for their unwavering support and encouragement throughout this year-long journey. Their belief in us has been a constant source of motivation and strength.

Abstract

This research addresses a common problem in software engineering where the knowledge is concentrated in a small subset of developers within a team which can lead to serious problems on their departure. The bus factor is a vital metric for assessing the resilience of software projects by estimating the risks associated with the loss of key contributors. Traditional approaches often rely on simplistic heuristics, failing to capture the complex collaboration dynamics and distributed knowledge within development teams. This thesis presents a graph-based method that models contributor interactions through social network analysis to identify critical knowledge holders. By incorporating multiple dimensions—such as lines of code contributions, file ownership diversity, interaction centrality, and knowledge decay—the proposed approach offers a more comprehensive and context-aware assessment of project risk. To enhance accuracy, the method integrates project context data from Jira issues, enabling the identification of essential contributors who may not write significant amounts of code but play key roles in the development process. Additionally, the tool supports efficient onboarding through automated documentation generation, facilitating smoother knowledge transfer. This study adopted a pragmatic research philosophy with an inductive, mixed-methods approach, leveraging Design Science Research Methodology (DSRM) to develop and evaluate a system for identifying critical knowledge holders in software projects. Through iterative development, the tool was enhanced across three stages: analyzing version control data to detect high-risk areas, integrating issue-tracking data for contextual insights, and generating detailed documentation to support knowledge transfer. The outcomes were validated through comparative analysis and stakeholder feedback, ensuring both technical accuracy and real-world applicability. The solution is delivered as an interactive, web-based application optimized for usability and visual analysis while evaluating its effectiveness using GitHub repositories, beginning with university projects and extending to 12 real-world industry projects from diverse organizations. The tool successfully identified all key contributors in these 12 projects with an overall accuracy of 85.56% , demonstrating its robustness across different team sizes, repository structures, and workflows. Compared to traditional techniques, this approach provides deeper insights for proactive risk mitigation and significantly strengthens team resilience.

Contents

List of Tables

List of Figures

Acronyms

API Application Programming Interface. 37, 39

BF Bus Factor. 18

DK Design Knowledge. 16

DOA Degree of Authorship. 8, 9

DSRM Design Science Research Methodology. 16

DSRP Design Science Research Process. 17

LLM Large Language Model. 40, 41

LOC Lines of Code. 42, 43

LOCC Lines of Code Changes. 10, 17

MVP Minimum Viable Product. 25

NPM Node Package Manager. 39

PoC Proof-of-Concept. 63

SDLC Software Development Life Cycle. 15, 25

VCS Version Control System. 5, 7, 9, 13, 17, 36

Chapter 1

Introduction

In the complex and rapidly evolving landscape of Software Development, the combined effort of a team leads to the success of the project. The average software team consists of approximately five to ten membersDikert et al. (2016), where the dynamic interplay of skills, individual talents, and experience within the team is pivotal. However, the continuous progress of such teams is frequently weakened by the uneven distribution of critical knowledge, which is often concentrated in the hands of a few key members. This imbalance can lead to significant vulnerabilities, particularly when these individuals are no longer available.

The critical role of collaborative team efforts in ensuring project success has long been recognized Constantino et al. (2020), yet knowledge concentration among a small subset of team members remains a significant risk. It's clear that about 16% of open-source projects suffer from the fact that the exit of all key developers leads to severe disruptionsAvelino et al. (2019). Only 41% of these projects managed to continue their development in the absence of their central figuresJabrayilzade et al. (2022). This phenomenon, often quantified using the "bus factor," measures the number of key team members whose loss would stall project progress Coplien and Harrison (2004). Traditional methods for calculating the bus factor rely heavily on static metrics, such as commit frequency or simplistic thresholds, which fail to account for the complex and dynamic nature of modern development environments.

The disadvantages of a low bus factor cause not only project delays but can also lead to a single point of failure, increase the on-boarding times for new team members, and increase the overall organizational dependency of employees. This research proposes a novel approach to bus factor analysis that leverages contribution pattern analysis and social network metrics providing a dynamic, interaction-focused perspective that adapts to modern workflows. It identifies critical contributors and highlights areas of concentrated knowledge, enabling proactive risk management through knowledge transfer and team-wide collaboration strategies.

1.1 Key Terms and Concepts

Bus factor

The concept of the bus factor, which is also known as the truck factor, was introduced in the book "Organizational Patterns of Agile Software DevelopmentCoplien and Harrison (2004)" as the minimum number of team members that have to suddenly be hit by a bus before the project stalls. In other words, The Bus Factor is a simple approach to express and comprehend a software project's reliance on key developers. It refers to the minimum number of developers required for project maintenance and evolution without halting the projectAvelino et al. (2019). A lower bus factor indicates a higher existential risk for projects. Conversely, a higher bus factor (relative to team size) implies a more evenly distributed knowledge base, minimizing the impact if a team member leaves the project suddenlyJabrayilzade et al. (2022).

Degree centrality

Degree centrality is a local centrality measure that indicates how well-connected a developer is within a network. It is calculated based on the number of edges, or direct connections, a developer (vertex) has with other developers. In other words, the degree centrality of a node v in a graph G is defined as $CD(v) = \deg(v)$, where $\deg(v)$ represents the total number of connections associated with that nodeBosu and Carver (2014). A higher degree centrality suggests that the developer is more actively engaged or influential within the network due to having more direct interactionsBock et al. (2023).

Lines of code (LOC)

Lines of Code (LOC) is a metric in software engineering used to quantify the size of a software project by counting the number of lines in its source code.

1.2 Background

Software projects are typically developed by teams rather than individuals. According to data from the ISBSG repository, the average software development team consists of about 7.9 members, with a median size of 5. In such collaborative environments, tracking how knowledge is distributed among team members can be a complex task. However, monitoring this distribution is essential, as studies have shown that developers with limited expertise in a particular component are more likely to introduce bugs into the project Jabrayilzade et al. (2022).

Inherently, software development projects are fraught with risks. Many projects face challenges like cost overruns, missed deadlines, and the development of incorrect functionalities, often due to underestimated risks such as personnel shortages. As software projects grow in complexity, the need for effective risk management strategies has become more pressing. To mitigate these risks, companies have increasingly adopted risk management solutions, which rely on indicators to evaluate project risks such as cost, time, and quality Cosentino et al. (2015). One critical risk factor is employee turnover, which can be assessed through the concentration of knowledge among developers. This is commonly referred to as the bus factor, which helps gauge the risk of key personnel leaving the project.

A low bus factor can arise from several factors: organizational, technical, and cultural Almarimi et al. (2021). Organizational factors include a lack of formal documentation, while technical challenges arise from maintaining complex or specialized systems. Cultural factors may involve environments that discourage knowledge sharing or promote isolated work. The consequences of a low bus factor are far-reaching—ranging from project delays and a single point of failure to increased onboarding times for new team members and greater organizational dependency on specific employees.

Numerous studies Avelino et al. (2016); Jabrayilzade et al. (2022); Klimov et al. (2023); Almarimi et al. (2021); Fritz et al. (2014); Rigby et al. (2016) have investigated the distribution of knowledge among software developers. However, these studies predominantly rely on algorithmic approaches that use hard-coded thresholds to measure the weight of contributions and knowledge distribution.

From a broader perspective, solving this problem has significant implications for the software industry. By ensuring a more equitable distribution of knowledge, companies can improve productivity, enhance project resilience, and increase overall employee satisfaction.

1.3 Research Problem

The uneven distribution of knowledge within software development teams is a common issue that often goes unaddressed. This knowledge imbalance can result in various problems, including unequal workloads, increased stress on key individuals, and potential project delays or failures. While previous research has developed algorithmic-based methods to identify the bus factor, as well as machine learning techniques related to detecting code smells, a significant gap still exists in the practical application of these methodologies at the organizational level. Therefore, developing a solution that can monitor and manage knowledge distribution, while identifying key dependencies to facilitate effective knowledge transfer, is both timely and crucial.

1.4 Research Aim

Design and develop a comprehensive application-based solution that can identify potential single points of failure (i.e., those contributing to a low bus factor) in software projects, highlight knowledge dependencies on individual developers and visualize their contribution areas in order to implement effective strategies for managing and transferring critical knowledge to reduce organizational bus factor risks.

1.5 Objectives

- To identify key personnel who are the potential single point of failure earlier in the project life cycle.
- To enhance the process of critical knowledge transfer within software development teams.
- To apply both the proposed tool and the benchmark algorithms to real-world project data and evaluate the practical relevance of identified key developers and bus factor scores.

1.6 Research Questions

1. How can we identify personnel who are potential single points of failure (i.e., those who cause a low bus factor) during the project life-cycle?
2. What methods can be implemented to help project members effectively manage critical knowledge transfer?
3. To what extent do the bus factor and key developer estimations generated by our tool align with those identified by existing state-of-the-art algorithms?

1.7 Scope

In-scope

- Identifying key personnel within software development teams

  Analyze the data residing within respective data sources (Ex: Version Control System (VCS), Jira issues) clearly identifying and indicating the key people within the team who're potential single point of failure.

- Visualize developer dependencies and key contributor effort distribution across project components.

  Developer dependencies are visualized using a network graph that highlights interactions between contributors, helping to identify critical knowledge links and potential single points of failure. Additionally, a treemap visualization illustrates the percentage of work done by key developers across different areas of the project, with each segment representing a component and the developer's relative contribution.

- Automatically generate project documentation to support on-boarding and knowledge transfer.

  Generate comprehensive documentation that highlights key components and workflows within the project, helping newly on-boarded developers quickly understand the system and facilitating smoother knowledge transfer.

Out-scope

- Results will only be valid for the organizations/software teams that maintain code-bases with proper version control in GitHub.

- The primary focus lies on identifying issues in the code-base rather than directly modifying the code-base itself. No automated tools for refactoring code to reduce the identified dependency will be offered from the system.

- The project will not perform comprehensive security audits or vulnerability assessments, and it will not include features specifically designed for identifying security issues in the code.

- The project will not involve making direct decisions about hiring or firing personnel, managing HR processes or policies, offering a comprehensive project management suite, or replacing existing tools like Jira, but will integrate with them and aid in making decisions to improve the knowledge distribution in certain high-risk areas.

1.8 Significance

This research addresses critical challenges faced by software development teams by supporting the organizations in proactively identifying key contributors whose absence could critically impact project continuity. Also, we propose a practical solution for visualizing developer dependencies and contribution distribution using interactive network graphs and treemaps powered by dynamic contribution metrics. These visualizations allow teams to pinpoint areas of knowledge concentration and take action to balance knowledge across the team.

We believe this approach will not only improve risk awareness within teams but also streamline the process of onboarding new developers and managing long-term maintainability of software projects. By leveraging a custom centrality score that incorporates lines of code, unique files modified, and degree centrality, our system offers a nuanced view of contribution beyond conventional commit counts.

The final outcome of this research is a web-based, open-source application that can be integrated into existing workflows. In contrast to existing work that primarily focuses on algorithmic estimation, we prioritize usability, real-world applicability, and transparency. Furthermore, the tool runs efficiently on standard hardware, without requiring high-performance computing environments—making it accessible to organizations of all sizes.

1.9 Overview of the dissertation

This dissertation is organized into six main chapters. Chapter 2 provides a comprehensive review of existing literature related to developer knowledge retention, bus factor estimation, and visualisation tools, highlighting the limitations of traditional methods. Chapter 3 details the research methodology, including the philosophical approach, data collection and analysis techniques, and the design process followed during development. Chapter 4 describes the implementation of the proposed system, from contribution analysis to the documentation generation. Chapter 5 presents the evaluation of the proposed methods through both qualitative and quantitative analysis, along with user feedback. Finally, Chapter 6 discusses the challenges encountered, summarizes the key contributions, identifies limitations, and outlines directions for future work.

Chapter 2

Literature Review

Several studies have been conducted to calculate the bus factor in recent years, most of which are algorithmic. Primarily, these studies calculate the bus factor using different ways of measuring code authorship, with some building upon and improving earlier research. This section provides an overview of the existing approaches for calculating the bus factor and its advancements over the years.

2.1 Early commit based approaches

Several bus factor estimation algorithms consider the knowledge distribution among project members using VCS data. When considering the methodology of capturing this knowledge distribution using VCS data it's important to examine the code understandability for a single developer. If the code can be easily understood by many members within a team the bus factor will be increased resulting in an even knowledge distribution. Early bus factor algorithms faced significant challenges in accurately capturing critical knowledge distribution across team members Avelino et al. (2016). As algorithms evolved, each introduced new methods to address previous limitations, focusing on refining how knowledge is assessed, reducing overestimations, and making the process scalable for larger teams.

Early bus factor estimation algorithms primarily relied on data from VCS, like Git, which record every modification made to a project's files. These algorithms, such as the one developed by Zazworka et al. (2010), operated on the assumption that any developer who made changes to a file was "knowledgeable" about that file, meaning they could maintain or extend it in the future Avelino et al. (2016). However, this assumption led to several issues:

- **Overestimating Knowledge**
  Treating any contributor as "knowledgeable" about a file doesn't consider the

depth or significance of their changes. For instance, if a developer made minor modifications, like updating documentation or fixing small bugs, the algorithm would still count them as knowledgeable about the file.

- **Struggling with Minor Contributions and Large Commits**
  Another problem arose with "clean-up" commits, where a developer might update multiple files without making substantial changes to the underlying code. These types of changes are often necessary but don't indicate deep familiarity with the code. Additionally, some commits are large but simple, such as reformatting code or updating dependencies. The initial algorithm could not differentiate these superficial edits from substantial contributions, so the bus factor estimation didn't always reflect actual developer expertise.

To address these issues, later algorithms Avelino et al. (2016); Cosentino et al. (2015); Rigby et al. (2016); Jabrayilzade et al. (2022) introduced more sophisticated approaches for evaluating developer knowledge.

2.2 Algorithmic advancements

Cosentino et al. (2015) proposed a six-step algorithm to create a tool for assessing and visualizing the bus factor of a software project. The developer knowledge calculation and identifying the key developer steps play an important role in this approach.
In the developer knowledge calculation, the Knowledge percentages are assigned to developers for each file based on their contributions. This knowledge is aggregated at different levels (directory, branch, project) and by file extension, using four metrics (M1-M4):

- M1: Assigns all knowledge of a file/line to the last contributor.
- M2: Knowledge is distributed based on modification frequency.
- M3: Similar to M2 but groups consecutive changes by the same developer.
- M4: Weighs more recent contributions more heavily.

During the key developer identification process, developers with sufficient knowledge of an artifact are identified as key developers. This step distinguishes between primary (modified a significant portion) and secondary developers (modified at least half of the threshold for primary developers).

Avelino et al. (2016) incorporated the Degree of Authorship (DOA) metric which was suggested by Fritz et al. (2014) where the contributions are weighted based on the extent and frequency of changes made by each developer. This meant:

- Weighted Contributions: Instead of counting every change equally, DOA assigns a higher weight to developers who contribute more significantly to a file over time. If a developer consistently makes large, meaningful contributions to a file, their DOA score increases, indicating they have a deeper knowledge of it.

- Discounting Insignificant Edits: Minor or infrequent changes contribute little to a developer's DOA score. This helps to avoid counting developers as "knowledgeable" based on minor edits, creating a more accurate picture of true expertise within the team.

DOA helps identify key developers who are genuinely knowledgeable about a file or set of files. This improved accuracy in bus factor estimates by differentiating between developers who only made occasional minor edits and those who were actively responsible for important portions of the codebase. In this approach Avelino et al. (2016), the authors of a file are calculated based on the changes performed in the file. These changes can be minor or major depending on the component and they haven't focused much on the impact of a change and its influence on the authorship of the specific file. The authors have mentioned that treating all the files as equal and not defining a clear line between core and non-core files were problems pointed out by the developers in a survey they performed. Therefore, the algorithm can further be improved by taking those limitations into concern.

An essential refinement in bus factor estimation has been the consideration of knowledge decay and time sensitivity. Early algorithms did not account for the fact that a developer's familiarity with code can decrease over time if they haven't actively engaged with it recently.

Jabrayilzade et al. (2022) introduced mechanisms to address this issue by a multimodal bus factor algorithm incorporating code reviews, VCS data, and meetings data. The algorithm introduced here also considers the DOA formula by Fritz et al. (2014), adjusting it by incorporating the contribution decay, code reviews, and meetings. This approach of calculating the bus factor is based on the algorithm introduced by Avelino et al. (2016).

- **Time-Sensitive Contribution Metrics**
  Jabrayilzade et al. (2022) began addressing this by incorporating time-sensitive elements into their calculations. They included a "knowledge decay" factor, which gradually reduces the weighting of a developer's contribution over time if they have not made recent updates.

- **Impact on Bus Factor Calculation**
  By factoring in knowledge decay, the bus factor becomes a dynamic, time-sensitive metric rather than a static one. This change improves risk assessments

by focusing on current knowledge holders, reducing reliance on outdated contributions that no longer add to the project's resiliency. It allows for a more responsive approach to identifying single points of failure, making the project better prepared to address critical team changes.

The algorithm developed by Rigby et al. (2016) introduced a blame-based approach to assess the susceptibility of software projects to developer turnover. Unlike previous commit-based methods, such as those proposed by Avelino et al. (2016), which rely on overall commit frequency, Rigby et al. (2016) approach leverages the `git-blame` feature to attribute authorship to each line of code. This focus on line-level authorship provides a granular view of developer knowledge and potential points of failure within the codebase.

In this approach, a line of code is considered abandoned if attributed to a developer who has since left the project, with an entire file being considered abandoned if at least 90% of its lines are abandoned. This high threshold is intended to exclude minor or trivial contributions and to provide a realistic measure of the project's vulnerability to turnover.

2.3 Metric-based techniques

Lisan and Norris (2024) examine two notable bus factor algorithms: one proposed by Cosentino et al. (2015) (CST) and another by Rigby et al. (2016) (RIG). The CST algorithm, which is publicly available, was enhanced in this study to incorporate additional metrics like Lines of Code Changes (LOCC) and cosine difference of lines of code (change-size-cos). On the other hand, the RIG algorithm, which employs a git-blame-based approach, was implemented from scratch in this study due to the unavailability of its code or tool. The authors' implementation and comparison of both algorithms on five open-source projects revealed key insights, which were further validated by feedback from the principal developers of these projects. However, the differentiation between core files and non-core files hasn't been properly addressed. The results indicated that LOCC and change-size-cos were more accurate than commit-based measurements, and their scalable implementation of the CST algorithm outperformed the existing tools.

2.4 ML based techniques

Despite all the algorithmic approaches, Almarimi et al. (2021) have developed the tool:csDetector by taking an ML-based approach for detecting the community smells by learning from known bad community development practices to identify similar issues, including the bus factor itself.

Each smell type is detected using its specific pre-trained model, and the results are displayed to the user via a command-line interface. The effectiveness of csDetector was evaluated using 143 open-source projects from GitHub, achieving an average F1 score of 84% in detecting ten common community smells. However, this study doesn't focus mainly on bus factor calculation and also it lacks a visualization component that would help developers track the health of their development community.

2.5 Visualisation tools

Visualization tools play a crucial role in bus factor analysis by presenting data in an intuitive and interactive manner. Only a limited number of bus factor analysis tools offer features for visualization. The tool by Cosentino et al. (2015) features a visualization tool that simplifies the bus factor analysis with an intuitive GUI and web interface. It displays project details like the bus factor, file counts, and developer knowledge distribution. Users can interact with clickable elements to explore relationships between files, directories, and developers. The tool also simulates developer turnover, recalculating the bus factor and highlighting affected artifacts to assess project vulnerabilities dynamically. Later, Klimov et al. (2023) introduced a web application designed to compute, export, and explore the bus factor metric named Bus Factor Explorer which's based on the study by Jabrayilzade et al. (2022). In this tool, the authors have excluded the data related to meetings and reviews considered in that algorithm to simplify the implementation. The application features treemap visualization, simulation mode, and a chart editor.

2.6 Network-Based Approaches for Identifying Core Developers in Software Projects

Identifying core developers is crucial for assessing the bus factor of a software project. While traditional approaches often rely on contribution metrics such as commit count or lines of code, these methods overlook the relational and collaborative aspects of software development. Network-based methods offer a more holistic view by capturing the interactions and structural positions of developers within social and technical networks.

Joblin et al. (2016) introduced several network-based models to distinguish core and peripheral developers, emphasizing the value of social network structures over simple contribution counts. By applying metrics such as degree centrality, eigenvector centrality, and hierarchy analysis, their work demonstrated how core developer roles emerge and evolve within communication and collaboration networks over time.

Oliva et al. (2015) examined the Apache Ant project from a socio-technical perspective. They identified key developers based on the artifacts they modified and analyzed their communication patterns using social network analysis (SNA). In addition, they constructed a coordination requirements network—based on task dependencies—to assess whether key developers occupied central positions in both social and technical contexts. Their study also evaluated socio-technical congruence, comparing expected and actual coordination, offering deeper insights into developer roles.

Zhang et al. (2011) evaluated the effectiveness of various network metrics—including degree centrality, PageRank, HITS (an algorithm to detect hubs and authorities), and betweenness centrality—in identifying core developers in the ArgoUML project. Using a ground truth based on mailing list posting privileges, they found all metrics performed similarly, with precision and recall around 60%. Their approach highlights the potential of network-based methods for developer classification in open-source environments.

Building on these foundations, the study by Bock et al. (2023) proposed an automated classification method using three widely accepted network metrics: degree centrality, a local metric measuring the number of developer connections; eigenvector centrality, a global metric that weighs a developer's influence by their neighbors' importance; and hierarchy centrality, which captures a developer's position relative to loosely connected clusters. Developers scoring in the top 20% quantile for each metric were considered core, based on the commonly used 80/20 threshold. This study validated the classification across various GitHub projects, demonstrating strong agreement with developer perception and improving generalizability through the use of diverse network structures.

Together, these studies underscore the value of network-based approaches in revealing the structural and social dynamics of software projects, providing a robust foundation for more accurate and context-aware bus factor estimation.

2.7 Limitations of the traditional techniques

Traditional bus factor estimation techniques, while foundational, face several significant limitations. Early methods often relied on oversimplified assumptions, treating any contributor to a file as equally knowledgeable without considering the depth or significance of their changes Zazworka et al. (2010); Avelino et al. (2016). These approaches also failed to account for the complexity or criticality of different files, treating all code artifacts uniformly despite their varying levels of importance Avelino et al. (2016). Additionally, knowledge decay over time was neglected, leading to inflated estimates based on outdated contributions Jabrayilzade et al. (2022). Scalability posed another challenge, as fine-grained analyses, like the git-blame-based approach, struggled to handle large-scale projects efficiently Rigby et al. (2016). Moreover, traditional algorithms often ignored non-code contributions, such as code reviews or documentation, which are vital to understanding a team's overall knowledge distribution Jabrayilzade et al. (2022); Almarimi et al. (2021). These limitations highlight the need for more comprehensive, dynamic, and multi-modal approaches to bus factor estimation.

When accessing the bus factor, the methods have progressed from simple VCS-based algorithms to multi-modal and machine learning approaches reflecting a broader trend towards integrating diverse data sources and advanced analytical techniques. When considering the past recent researches, the key turning points include the introduction of multi-metric and blame-based methods, the incorporation of visualization tools, and the shift towards comprehensive, multi-modal analyses.

Despite not directly calculating the bus factor, network-based methods provide a robust framework for identifying core developers in a software development project. These methods model developer interactions—such as communication patterns, coordination needs, and collaboration structures—using social network analysis (SNA). As highlighted in studies Bock et al. (2023),Joblin et al. (2016), Oliva et al. (2015), Zhang et al. (2011) network-based metrics such as degree centrality, eigenvector centrality, and hierarchy centrality offer richer insight into developers' roles and influence in a project. They can reveal hidden leaders, key communicators, and coordination bottlenecks that traditional contribution-based metrics may overlook.

In the studies mentioned previously, there is inadequate attention given to handling code with varying levels of complexity and understandability. Since the Bus factor is a metric to measure the knowledge spread within an organization it's important to treat files of differing code complexities distinctly to accurately reflect their impact on knowledge spread and maintainability. Balancing complexity and usability, the accuracy of different metrics, and the inclusion of non-code contributions when calculating the bus factor further need to be studied and addressed.

To bridge these gaps, our proposed method incorporates enhancements that directly address the limitations identified in existing literature. Table 2.7 summarizes these shortcomings alongside the corresponding features of our approach that mitigate each issue. By aligning the enhancements with specific weaknesses in prior techniques, the table provides a clear rationale for the methodological choices made in this research.

Limitations in Existing Techniques	**Enhancements in Proposed Approach**
Equal weighting of all contributors regardless of contribution depth	Introduces a composite centrality score combining LOC (Lines of Code), file diversity, and degree centrality to reflect the significance of contributions.
Neglect of temporal relevance and knowledge decay	Limits data analysis to contributions within the most recent 1.5 years, ensuring focus on current project knowledge holders.
Scalability issues in fine-grained analyses (e.g., git-blame)	Uses higher-level contribution metrics that scale better for large projects.
Exclusion of non-code activities such as documentation and code reviews	Incorporates Jira issue tracking data (e.g., reporting, commenting, assignments), capturing non-code contributions.
Limited insight into communication and collaboration structure	Builds a collaborative developer network using both Git and Jira data to uncover social-technical relationships.
Limited visualization for actionable insights	Provides interactive network graphs and treemap visualizations that highlight critical developers and their contribution areas, enabling stakeholders to easily detect knowledge silos and key dependencies within the project.

Table 2.1: Addressing Gaps in Existing Bus Factor Estimation Techniques

Chapter 3

Research Methodology

3.1 Chapter Overview

This chapter provides a comprehensive overview of the methods and processes used in the development and evaluation of the system. Starting with outlining the research philosophy and approach that guided the study, laying the foundation for how decisions were made throughout the design process. The selected Software Development Life Cycle (SDLC) model is then discussed, followed by a breakdown of the data collection methods and the steps taken to prepare the data for analysis. Furthermore, the hardware configurations, preliminary experiments and finally, the system architecture is presented through various diagrams, including use case, activity, and architectural diagrams, to visually represent the system's structure and behavior.

3.2 Research Philosophy

For research philosophy, pragmatic research philosophy is chosen as it focuses on practical outcomes and uses mixed methods for addressing the research questions, explicitly bridging positivism and interpretivism philosophies. With that it prioritizes the research problem over rigid philosophical commitments, allowing flexibility to combine empirical observations with subjective interpretations (Kirongo A. Chege (2020)). The positivist approach highlights evaluating the success of a study by looking over the observable evidence and results. This aligns well with our research goals, which include evaluating the accuracy of our results by comparing them with baseline datasets, analyzing the tool performance which require quantitative assessments, produce measurable and reproducible findings while providing a solid empirical foundation for our study. Complementing this, the interpretivist approach adds a layer of depth by acknowledging the subjective aspects of project dynamics. Through methods such as collecting feedback from project stakeholders and analyzing the contextual factors behind team interactions, we seek to embrace the variability of the results based on the subjective experience and perspective of the individuals.

3.3 Research Approach

This study adopts an inductive research approach, emphasizing on deriving general insights and patterns from specific observations rather than testing predefined theories. Unlike the deductive method which begins with a hypothesis and focuses on verifying it the inductive approach is suits best to the exploratory nature of this research since the main goal here is not to confirm or refute an existing theory, but to uncover and understand how critical knowledge holders can be identified in software projects through a novel method developed from real-world data and observed patterns.

The inductive reasoning with a mixed-methods strategy, provides both flexibility and depth while the integration of quantitative and qualitative data allows the system to evolve over time, enhancing both its accuracy and practical usefulness.

3.4 Research Strategy

Design Science Research Methodology (DSRM) was selected as the research strategy for this study as it well aligns with the goal of creating a new process model tailored to a specific problem domain. Rooted in engineering and the sciences of the artificial, the Design Science Research (DSR) paradigm is inherently problem-oriented and focuses on addressing practical challenges while expanding human knowledge through the creation of innovative artifacts and the development of Design Knowledge (DK) Brocke et al. (2020). A major advantage of this approach is its innovative potential as it boosts human and organizational capabilities by creating practical solutions like models, methods, and tools. Thanks to its structured, iterative process and focus on real-world impact, DSR is especially effective in scenarios like this. By applying DSRM, this research was able to develop a robust and impactful process model to tackle the issue of bus factor mitigation within organizations. The following subsections details how this methodology was applied throughout the study.

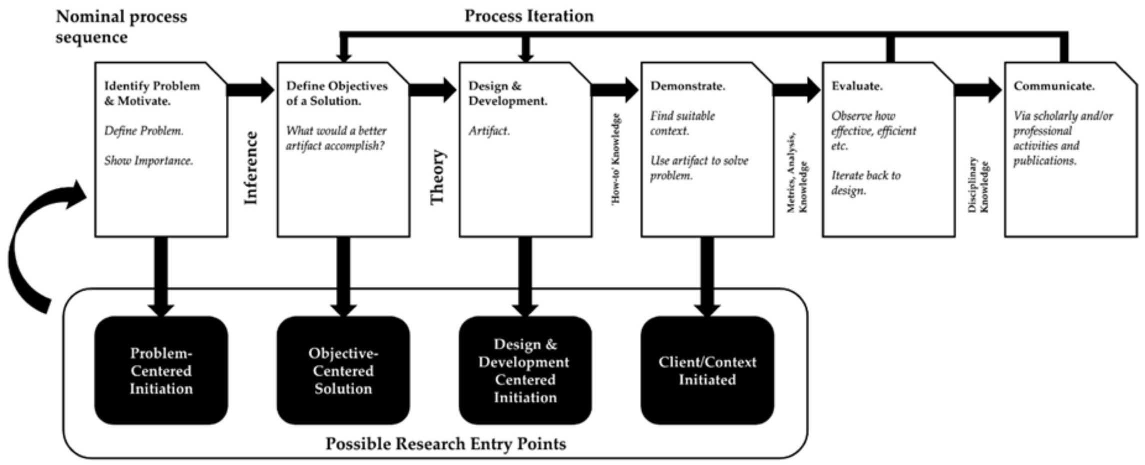

Nominal process sequence
Process Iteration
Identify Problem & Motivate.
Define Problem.
Show Importance.
Inference
Define Objectives of a Solution.
What would a better artifact accomplish?
Theory
Design & Development.
Artifact.
'How-to' Knowledge
Demonstrate.
Find suitable context.
Use artifact to solve problem.
Metrics, Analysis, Knowledge
Evaluate.
Observe how effective, efficient etc.
Iterate back to design.
Disciplinary Knowledge
Communicate.
Via scholarly and/or professional activities and publications.
Problem-Centered Initiation
Objective-Centered Solution
Design & Development Centered Initiation
Client/Context Initiated
Possible Research Entry Points

Figure 3.1: Design Science Research Methodology

3.4.1 Problem Identification and Motivation

Starting with the initial phase of the Design Science Research Process (DSRP), this research adopts a problem-centered approach, aiming to identify the key individuals influencing the bus factor and facilitate the transfer of their knowledge to minimize related risks. The bus factor reflects the threat associated with losing critical knowledge when essential team member(s) leave the particular organization they have been working on, making it a serious concern for project sustainability and the organization's capacity for growth and adaptation. This challenge prompted the motivation to undertake this study and additionally, a detailed description and explanation of the problem is presented in the introduction and literature review sections.

3.4.2 Iteration 1

Objectives

- Develop a system to identify key personnel contributing to the bus factor within an organization using only the metrics derived from VCS data.
- Determine areas of high risk associated with potential employee turnover.
- Providing early indications for relevant stakeholders about identified risks.

Design and Development

- Gather contributor data from VCS data (GitHub) which includes the contributor list and the list of files, LOCC which each contributor has contributed.

- Key individuals (Critical knowledge holders) were identified using a graph based method which incorporates a custom centrality metric calculated using the prior collected data.
- Assessed the risk level of different areas based on the identified key personnel and their contributions. For a given area in the code, if majority of the contributions made by the key individuals, such areas are considered as high-risk areas that require immediate attention.
- Visualize the code areas and the risk factor associated with it using the tree-map visualization.

Demonstration

The implemented tool calculated the BF for projects in the primary dataset and compared the results with ground truth values. Additionally, BF was computed for projects in the secondary dataset to enable a comparative analysis of the findings.

Evaluation

The accuracy of the results was calculated against ground truth data from the primary dataset. Additionally, variance between outputs from existing tools and the proposed artifact was measured to assess deviations in the results.

3.4.3 Iteration 2

Objectives

- Integrate project-context data (Jira issues) to identify critical knowledge holders who may not contribute significant code.
- Improve the precision of bus-factor estimates across varied collaboration patterns by blending code and process signals.
- Provide richer, actionable explanations of risk to managers by linking contributors to both code and issue-tracking artefacts.

Design and Development

- For acquiring the data, queried the Jira boards, using the Jira REST API for issues updated in the same 18-month window as the Git commit history.
- Applied fuzzy matching on names and emails to merge Jira users with GitHub contributors, producing a unified contributor registry.

- Added Jira co-participation edges (shared issues, comments, task assignments) to the file-co-edit graph; edge weights reflect both code and issue interactions.
- Extended the centrality formula to include a Jira activity term, weighted by tunable parameters (jira_weight, jira_only_weight).
- Treemap and network visuals now colour-encode nodes by the composite score, with tooltips disaggregating Git vs. Jira contributions.

Demonstration

The enhanced tool was executed on 15 projects with public or cooperative Jira boards. Comparative dashboards illustrated how previously overlooked coordinators (e.g., QA leads, product owners) emerged as key contributors. These visualizations were reviewed with project stakeholders for validation.

Evaluation

The accuracy of the results was calculated against ground truth data from the simulated dataset.

3.4.4 Iteration 3

Objectives

- Develop a comprehensive documentation generation mechanism which enables the users to generate and store in-detailed documentation of the functionalities offered by a given project. (Reason: In case if a key individual departs, the newcomers spend more time understanding what functionalities are offered from that particular piece of software, what is the flow of the method invocation and how they interact and collaborate with each other to provide the desired functionality, etc. Having detailed documentation which contains all of this information cuts down this latency and provides a smooth onboarding.)

Design and Development

- Integrated RepoMix to convert the source code repository into a single .xml file that includes structural metadata (classes, functions, files).
- Developed a prompt template for the DeepSeek API, instructing the model to analyze the XML and output a developer-friendly documentation structure in JSON.
- Enabled visualization of the generated documentation structure within the tool, seamlessly integrated with the key developer identification module to provide a comprehensive view of both system functionality and contributor responsibility.

Demonstration

Documentations were generated for 10 projects from the primary dataset used for calculating the bus factor, where the exact functionalities and workflows are already known.

Evaluation

It employed an empirical validation approach, where the tool's generated outputs is compared the manual, ground-truth understanding of how those systems operated. It allowed for iterative checking whether the component accurately traced method invocations, captured key workflows, and organized information logically.

3.4.5 Communication

- Dissemination of Results: Shared the research findings with the evaluation panels and have already submitted two research papers for publication which are under peer-review right now.
- Stakeholder Engagement: Presented the outcomes to the industry experts, and organizations/teams connected with this research, highlighting the practical implications and benefits of the developed system.

3.5 Time horizons

Although longitudinal studies are often preferred in Design Science Research (DSR) due to their ability to capture the iterative nature of development, assess long-term practical impact, and observe emerging behaviors over time, this study adopted a cross-sectional time horizon. The primary reason for this choice was the practical constraint of limited time available for the research. Despite this limitation, the cross-sectional approach proved effective by enabling targeted data collection and analysis at key stages of the study. This snapshot-style evaluation provided timely insights that directly informed the iterative development process, allowing for meaningful refinement of the artifact based on the results and feedback gathered during each evaluation phase.

3.6 Data Collection

This research adopted a multi-faceted approach to data collection, combining both qualitative and quantitative methods to gain a comprehensive understanding of the system's usability and potential for adoption. The different methods used for the data collection are listed below.

- **Initial Survey:** An initial survey was conducted to validate the core idea behind the research and to assess the perceived need for a tool capable of calculating the bus factor more accurately. Participants included software developers, engineers, and project managers who provided valuable input on the practicality of such a tool, including the types of features and insights they would expect from it. This method was effective in grounding the research in real-world needs and aligning the system design with user expectations.

- **Expert Opinions:** To deepen the understanding of team dynamics and the practical aspects of software project management, expert opinions were gathered from individuals with significant experience in the software industry. These insights helped contextualize how key developers are traditionally identified, how knowledge flows within teams, and the challenges in recognizing complex or emerging risks such as knowledge silos. Expert input was crucial in shaping the assumptions of the proposed approach and refining its practical relevance.

- **Primary Dataset:** A primary dataset was constructed specifically for this research, comprising a selected set of software projects where the actual bus factor and the list of critical contributors were already known. This dataset served as the ground truth for validating the accuracy and effectiveness of the proposed graph-based method. Using this controlled set of data allowed for reliable benchmarking and performance evaluation under known conditions.

- **Secondary Datasets:** In addition to the primary dataset, several secondary datasets were obtained from previously published research that attempted to address the bus factor calculation problem. These datasets, along with the results reported in those studies, were used to conduct experiments and compare the performance of the proposed approach against existing methods. This comparative analysis helped highlight the improvements and contributions made by the novel technique.

- **Synthetic Datasets:** Since the organizations reluctant to share project-specific or context-related data such as internal communications, project management data or meeting logs, synthetic datasets were also created to supplement the real-world data. These datasets were carefully designed to reflect realistic project scenarios by closely analyzing publicly available data from open-source projects and incorporating feedback from industry experts. This process helped mitigate inconsistencies and ensured a higher degree of reliability and representativeness across both real and synthetic datasets.

3.7 Data Preparation and Analysis

3.7.1 Data Processing

In preparing the primary dataset for this study, most of the data was derived from version control systems of selected software projects. A key step in the data processing phase involved filtering out bot accounts that are commonly used in modern development pipelines to automate tasks such as dependency updates, code formatting, etc. These automated contributors do not reflect actual knowledge ownership or decision-making within a team and thus were excluded to ensure that only real human contributors were considered in the analysis.

Moreover, it was important to address the scale of contributor involvement when processing secondary datasets. Most real-world industry projects typically involve small to medium-sized teams with around 20 members or fewer but majority of the projects listed in the available datasets featured significantly larger contributor bases. This disparity can skew analysis, especially when trying to generalize results to closed source or enterprise settings. Therefore, data points from open-source projects with unusually large contributor pools were selectively filtered out to retain only those that aligned more closely with realistic team sizes.

Another refinement involved excluding open-source projects that had aged significantly. In the early stages of any project, knowledge tends to be concentrated among a few core individuals. However, as open-source projects mature and grow in popularity, knowledge and responsibilities are naturally distributed among a wider contributor base. This evolutionary behavior does not align with the dynamics observed in most real-world, closed-source projects, where critical knowledge often remains centralized. Due to this reason, older projects were excluded from the secondary datasets to maintain relevance and realism in the evaluation process.

3.7.2 Data Analysis

The data analysis phase combined both quantitative and qualitative approaches to offer a more complete evaluation of the proposed method's effectiveness.

On the qualitative side, the initial user survey served to capture subjective feedback on the idea of a more accurate bus factor calculation tool, including user expectations and practical feature suggestions. Furthermore, feedback from individuals directly involved in the selected projects was collected during the final result evaluation phase. These participants were asked to assess whether the system accurately identified critical knowledge holders in their teams. Since only those who have worked closely within a project can truly understand the flow of knowledge, this feedback offered valuable insights.

For the quantitative evaluation, standard metrics such as accuracy, precision, and recall were used to assess how well the proposed method identified key contributors compared to known ground truth data, providing an objective measure of the system's performance. Apart from that, system resource usage was also monitored, including memory consumption and CPU usage, to evaluate the efficiency and scalability of the tool under varying workloads. This allowed research to not only assess the accuracy of the results but also the practicality of deploying the tool in real-world settings. Also, benchmarking the system's performance against existing algorithmic approaches and tools will provide a clearer and more comprehensive understanding of its effectiveness.

Accuracy

Accuracy measures the proportion of true key developers correctly identified by the tool relative to the total number of actual key developers. It answers the question, which *out of all developers who truly matter, how many did the tool correctly flag?*

$$\text{Accuracy} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives} + \text{False Positives}} \times 100\%$$

In the above formula;

- True Positives: Developers correctly identified by your tool
- False Negatives: Key developers not identified by your tool

Precision

Precision evaluates how reliable the tool's positive identifications are by answering *of all developers labeled as key by the tool, how many are actually key?*

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \times 100\%$$

Recall

Recall (or Sensitivity) quantifies how comprehensively the tool captures all actual key developers.(Did the tool identify most (or all) of the true key contributors?)

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \times 100\%$$

F1-score

The F1-Score harmonizes precision and recall into a single metric, balancing their trade-offs by showcasing *how well does the tool balance reliability (precision) and completeness (recall)?*

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

3.7.3 SUS Score

The System Usability Scale (SUS) yields a score from 0 to 100 that represents a percentile-based measure of overall usability. While the raw SUS value is technically a percentile, it is often mapped to adjective ratings, letter grades, and percentile ranks to aid interpretation. A commonly used benchmark (Bangor, Kortum & Miller, 2009) is shown in Table 3.7.3.

Score Range	**Adjective Rating**
0–25	Worst imaginable
26–39	Poor
40–52	OK (below average)
53–73	Good (average)
74–85	Excellent
86–100	Best imaginable

Table 3.1: SUS Score Interpretation (Bangor, Kortum & Miller, 2009)

In practice, a SUS score of 68 is considered the average benchmark:

- Scores below 68 indicate usability issues that may require improvement.
- Scores around 68 are "above average."
- Scores above 80.3 correspond approximately to an A grade and are deemed truly excellent.

In our study, the average SUS score of **86.7** falls in the "Best imaginable" range, underscoring an exceptionally high level of user satisfaction and ease of use.

3.8 Hardware Configuration

The testing and evaluation of the proposed system were conducted on a machine running Windows 11 64-bit, equipped with an 11th Gen Intel Core i7-1165G7 processor (2.80GHz) and 16 GB of RAM, providing sufficient computational power and memory to handle the data processing and experimental workloads efficiently.

3.9 Software Development Life-Cycle

As for the Software Development Life Cycle (SDLC), this research followed the iterative and incremental approach. This enables a cyclical and flexible method of development where the product evolves through repeated cycles (iterations) and is built in small, manageable increments.

The incremental aspect of the approach was beneficial as the system gets developed in multiple stages, and each stage introduces a certain set of new features or enhancements allowing the development team to build upon a stable foundation while testing each addition thoroughly. Since the research-based product involved evolving requirements and gradual integration of components, adapting the incremental model ensured that progress was tangible and measurable after each phase without disrupting previously developed functionality. Meanwhile, the iterative nature of the process enabled continuous refinement where each iteration began with a subset of requirements, aiming to produce a working prototype or Minimum Viable Product (MVP), which was then tested and reviewed. Feedback from stakeholders and observations during testing helped identify gaps, uncover new requirements, and improve system behavior.

3.10 Experiments and Preliminary Results

This section presents a detailed exploration of the factors influencing bus factor estimation, highlighting the limitations of traditional methodologies and the complexities of modern software development. Through a series of targeted experiments, we compare prominent tools and algorithms, such as the Bus Factor Explorer and Avelino et al. (2016) approach, to assess their accuracy and practical relevance. Additionally, we investigate correlations between bus factor values and key metrics like cognitive complexity, cyclomatic complexity, team size, and programming language trends. These analyses reveal critical gaps in existing methods, such as their inability to account for nuanced collaboration dynamics and knowledge decay. By identifying these shortcomings, the findings of this section establish a solid foundation for the development of a more dynamic and comprehensive approach to bus factor analysis.

3.10.1 Comparative analysis of Bus Factor estimations by Bus Factor Explorer and Avelino et al. algorithm

To better understand the variability in bus factor estimations, we conducted a comparative analysis of two prominent methodologies: the Bus Factor Explorer and the algorithm proposed by Avelino et al. (2016). These algorithms were chosen due to their significant impact in the domain—Bus Factor Explorer being regarded as an intuitive tool leveraging modern refinements, and Avelino et al. (2016) algorithm serving as the benchmark for bus factor studies. Notably, the Bus Factor Explorer incorporates a modified version of Avelino et al. (2016)'s algorithm, refined by Jabrayilzade et al. (2022), which introduces an exponential decay model. This model prioritizes recent contributions by halving the knowledge impact of older contributions every five months, addressing the temporal relevance of activity—a consideration absent in Avelino et al. (2016)'s approach.

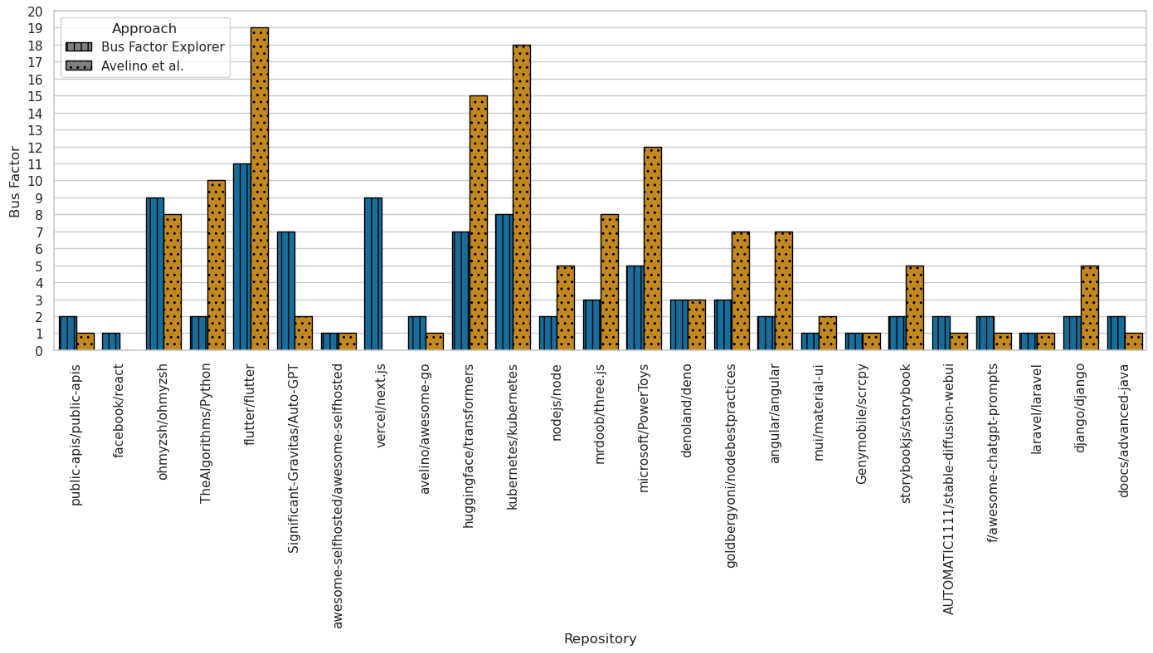


Figure 3.2: Bus Factor estimations for various repositories using two approaches: Bus Factor Explorer (solid hatching) and Avelino et al. algorithm (dotted hatching). The x-axis lists repositories, and the y-axis indicates the corresponding bus factor values

We analyzed bus factor estimations across 25 widely recognized GitHub repositories, encompassing projects with star counts ranging from 70,000 to 240,000. The results(Figure 3.2) highlighted notable discrepancies between the two algorithms. While repositories like flutter/flutter and kubernetes/kubernetes displayed significantly higher bus factor values under Avelino et al. (2016)'s method, others, such as public-apis/public-apis and ohmyzsh/ohmyzsh, showed relatively smaller variations. This underscores the sensitivity of bus factor estimates to the choice of algorithm and highlights the overall impact over the final bus factor value by the introduction of factors such as knowledge decay in assessing developer contributions.

3.10.2 Correlation analysis between bus factor and Cognitive/Cyclomatic Complexity

Cognitive complexity explains how difficult it is for a developer to understand the code, which directly affects the maintainability and debugging of the code. In contrast, Cyclomatic complexity measures the number of linearly independent paths through a program's source code Ebert and Cain (2016). We create the hypothesis that repositories with higher cognitive and cyclomatic complexities might have lower bus factor due to the high difficulty in understanding and managing the code. In other words, a higher bus factor could be associated with lower complexity, concluding that multiple contributors can easily continue the project in case some developers are unavailable.

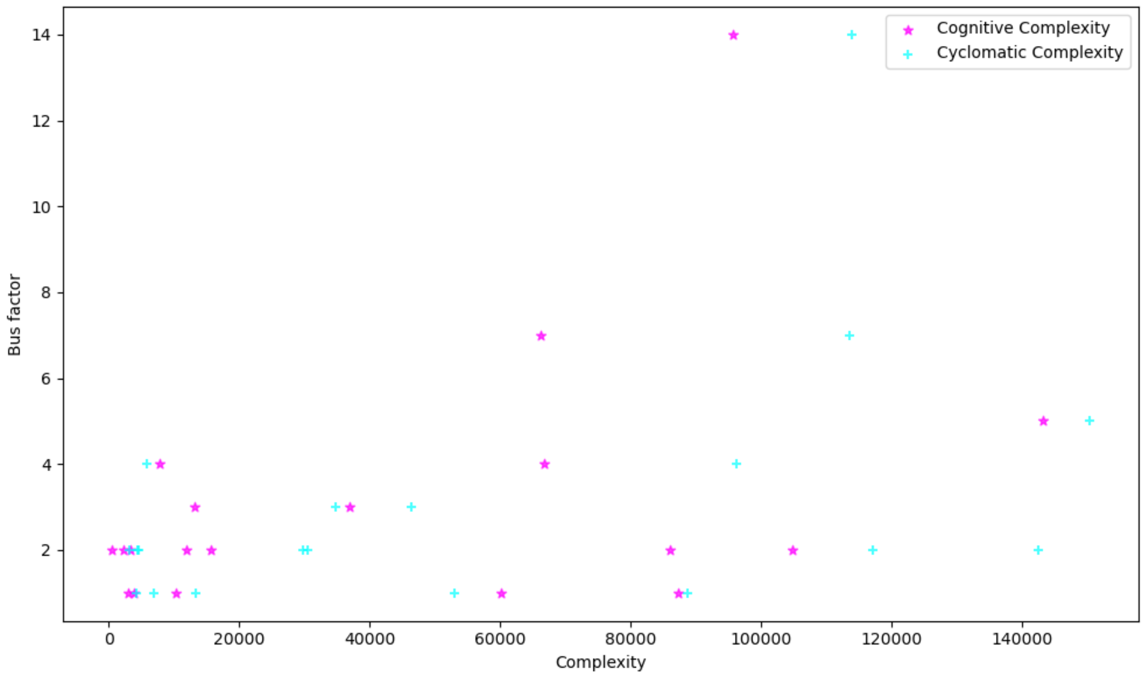


Figure 3.3: Scatter plot illustrating the relationship between Bus Factor and two software complexity metrics—Cognitive Complexity(magenta stars) and Cyclomatic Complexity(cyan plus markers) for selected repositories

To explore this correlation between bus factor and cognitive/cyclomatic complexity we took set of repositories and calculated the bus factor using bus factor explorer tool Klimov et al. (2023) and SonarCloud[1] which is an online automated software quality analysis platform delivered by SonarQube[2] to get the Cognitive Complexity

[1]https://sonarcloud.io/login

[2]https://www.sonarsource.com/products/sonarqube/

and Cyclomatic Complexity. The scatter plot in 3.3 suggests a weak correlation between bus factor and cognitive or cyclomatic complexity which also means that higher cognitive or cyclomatic complexity does not necessarily lead to a high bus factor.

3.10.3 Correlation analysis between the number of contributors and Bus Factor

This examines the relationship between team size and bus factor, motivated by the observation in prior studies that agile software development projects typically thrive with team sizes of 5–8 members, while exceeding 20 members often hinders collaboration and communication. Such challenges are critical, as reduced collaboration can result in a low bus factor, suggesting a potential negative correlation between these two factors. Large teams, despite offering diverse expertise, may inadvertently contribute to lower bus factor values due to issues such as increased specialization leading to knowledge silos, communication complexities causing misconceptions and knowledge gaps, and difficulties in assigning clear ownership, which can disengage contributors and restrict knowledge sharing.

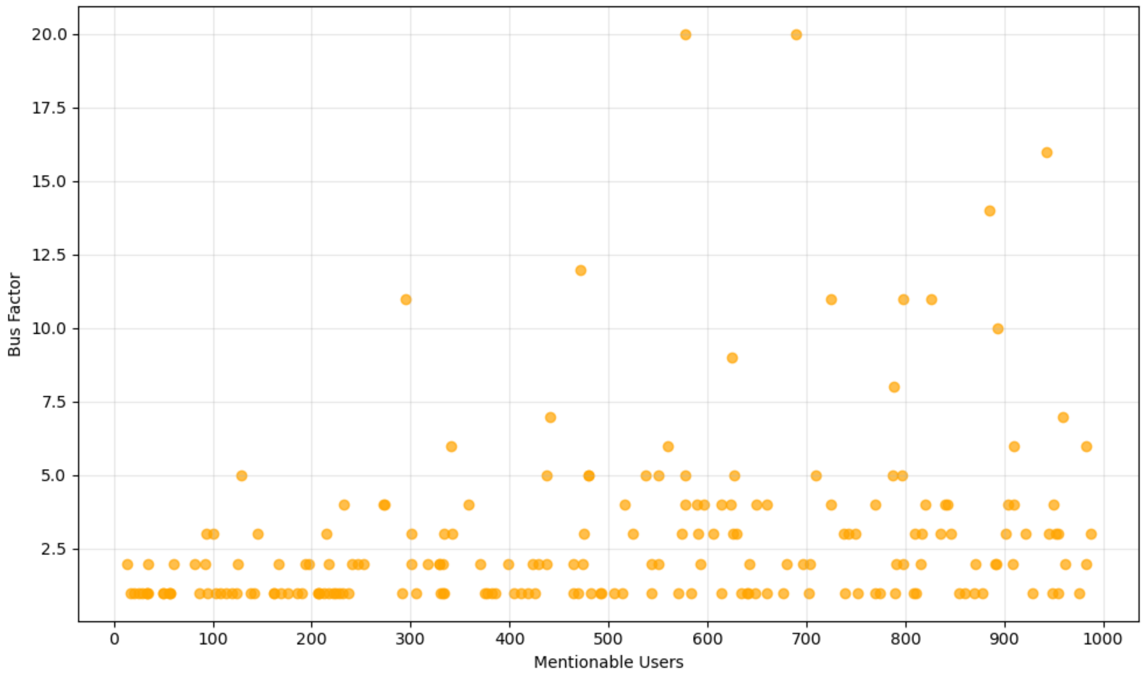


Figure 3.4: Scatter plot showing the relationship between Bus Factor and the number of contributors in software projects. Each point represents a project, with Bus Factor on the vertical axis and the number of contributors on the horizontal axis

To explore this, we analyzed 200 repositories extracted from the dataset used in the study of the Bus Factor Explorer tool. The bus factor and the number of mentionable users were plotted(Figure 3.4), and their correlation was calculated. While a slight positive correlation (0.28) was observed—indicating a weak tendency for bus factor to increase with larger teams—the trend was neither strong nor definitive. This finding underscores that team size alone does not strongly predict bus factor values, and other factors may play a more significant role in determining the resilience of software development teams.

3.10.4 Analyzing Bus factor in small teams (team size ≤10)

Conducted to focuses on analyzing the bus factor within small teams, defined as having 10 or fewer contributors, to understand the dynamics of key developer dependencies in such setups. Using 20 randomly selected GitHub repositories with star counts exceeding 1,000, bus factor values were calculated using the Avelino et al. algorithm and the Bus Factor Explorer tool(Table 3.2). A consistent trend was observed across the repositories: despite the presence of multiple contributors, a significant portion of the work was typically managed by a single individual.

Notably, discrepancies arose between the two tools. For example, in the go-size-analyzer repository, Avelino et al. (2016)'s tool listed the same user twice, likely due to a single individual using multiple GitHub accounts. In the qBitTorrent-fluent-theme repository, the Bus Factor Explorer identified a different primary contributor than Avelino's tool, emphasizing the latter's focus on recent contributions. While both methods generally reported a bus factor of 1, the variation in identifying the key contributor highlights differences in their algorithms—such as how each accounts for recency and activity patterns. This reinforces the nuanced understanding required when analyzing bus factor in small teams, where dependency on individual contributors is particularly pronounced.

3.10.5 Analyzing the trend of Bus Factor towards specific programming languages

To investigate whether the bus factor of software projects exhibits trends based on the programming language used, a hypothesis is considered along with several factors that potentially influencing the bus factor, including the language's learning curve, complexity, community size, resource availability, documentation culture, maturity, stability, and paradigm familiarity. To test this, an unbiased sample from same original dataset from the Bus Factor Explorer study was extracted and analyzed, grouping projects by their primary programming language. The average bus factor for each language was calculated and visualized using a bar plot sorted in ascending order.

The analysis(Figure 3.5) revealed that irrespective of the programming language, most projects exhibit a relatively low average bus factor, typically between 2 and 4 while the overall average bus factor across all languages was approximately 2.49

Table 3.2: Summary of repositories and their bus factor values (no. of contributors ≤ 10)

Repository	**Language**	**No. of contributors**	**No. of stars**	**Busfactor explorer**	**Algorithm by Avelino et al.**
murex	Go	10	1.5k	1	1
dum	Rust	10	1.5k	1	1
wat	Python	7	1.6k	1	1
so-vits-svc	Python	8	3.6k	1	1
bright	Typescript	7	1.4k	1	1
go-size-analyzer	Go	6	1.2k	1	2
JankyBorders	C	4	1.3k	1	1
circumflex	Go	6	1.3k	1	1
lucky-commit	Rust	9	1.3k	1	1
focus-rings	Typescript	10	1.1k	1	1
errorx	Go	6	1.1k	1	1
gotraceui	Go	6	1.1k	1	1
suture	Ruby	8	1.4k	1	1
pyapp	Rust	10	1.2k	1	1
shpool	Rust	6	1.2k	1	1
frourio	Typescript	7	1.2k	1	1
suspend-react	Typescript	8	1.4k	1	1
qBitTorrent-fluent-theme	Python	6	1.2k	1	2
OpenNJet	C	4	1.2k	1	1
httm	Rust	4	1.4k	1	2

(rounded to 3), underscoring that the risk of low bus factor persists regardless of language choice. This trend was consistent even among widely used languages like Java and Python, suggesting that the bus factor is more heavily influenced by the project's structure and dynamics rather than the characteristics of the language itself. While factors such as community size, documentation quality, and language maturity might enhance a project's overall development process, they do not significantly mitigate the risk of a low bus factor.

Also conducting these experiments clarified several other drawbacks of the existing Bus Factor Explorer algorithm:

- **Inaccuracy:** Despite claims of accuracy, the tool failed to produce correct results for a known set of repositories in almost all cases, indicating significant flaws in its calculations.

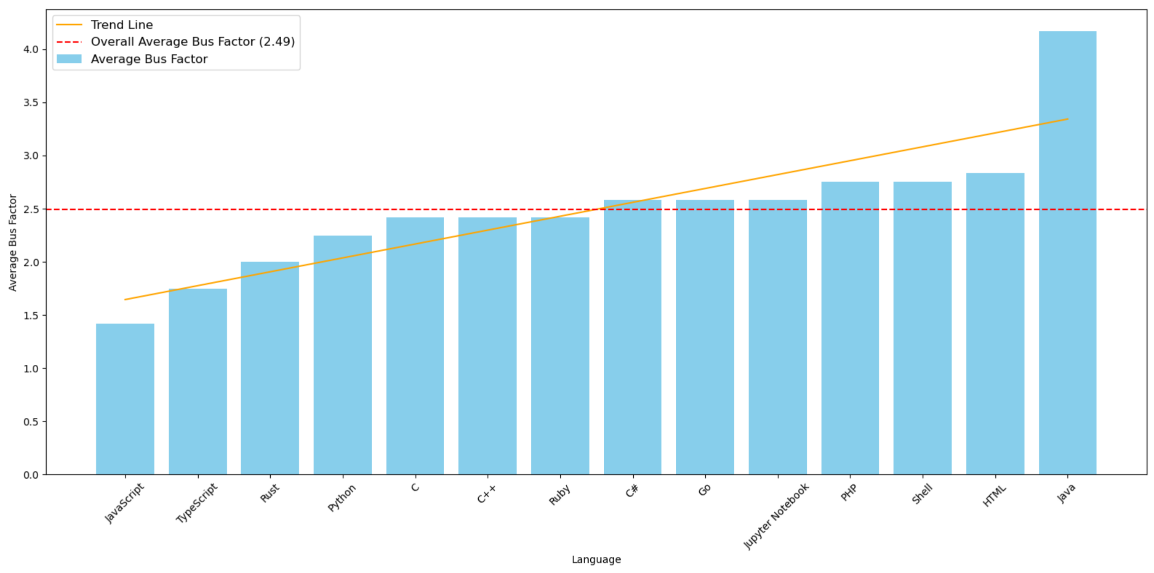


Figure 3.5: A bar plot depicting average Bus Factor for projects grouped by programming language. The bars(light blue) represent the average Bus Factor for each language, while the dashed line(red) indicates the overall average Bus Factor (2.49). A trend line (orange) highlights the increasing trend in the average Bus Factor across languages

- **Bot Filtering:** The algorithm does not properly filter out bot accounts, resulting in bots being incorrectly identified as key contributors in some instances.
- **User-Friendliness:** The tool returns users' email addresses instead of GitHub usernames, making the results less user-friendly and harder to interpret.

These limitations motivated us to explore alternative approaches for analyzing bus factor trends and developing customized methods tailored to our research goals.

3.11 System Design

3.11.1 High-Level Architecture Diagram

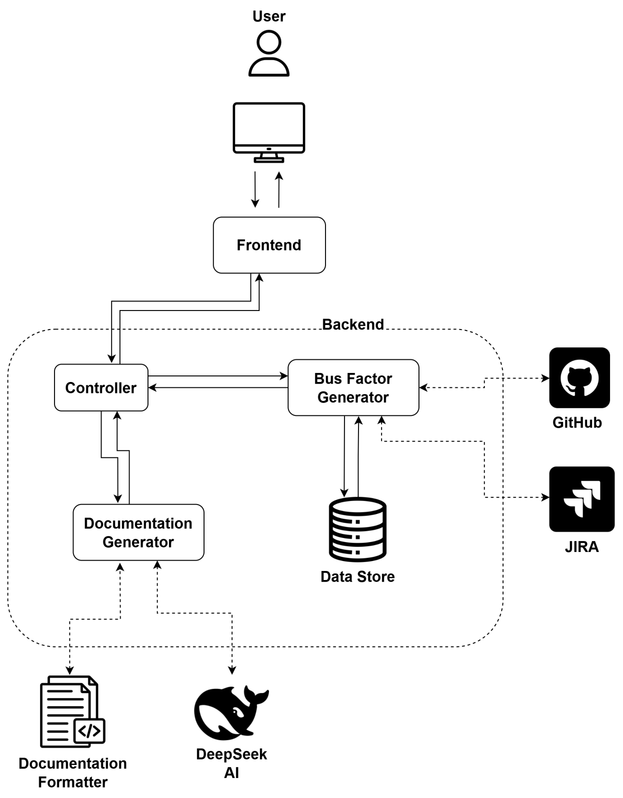


Figure 3.6: High-Level Architecture Diagram

3.11.2 Use Case Diagram

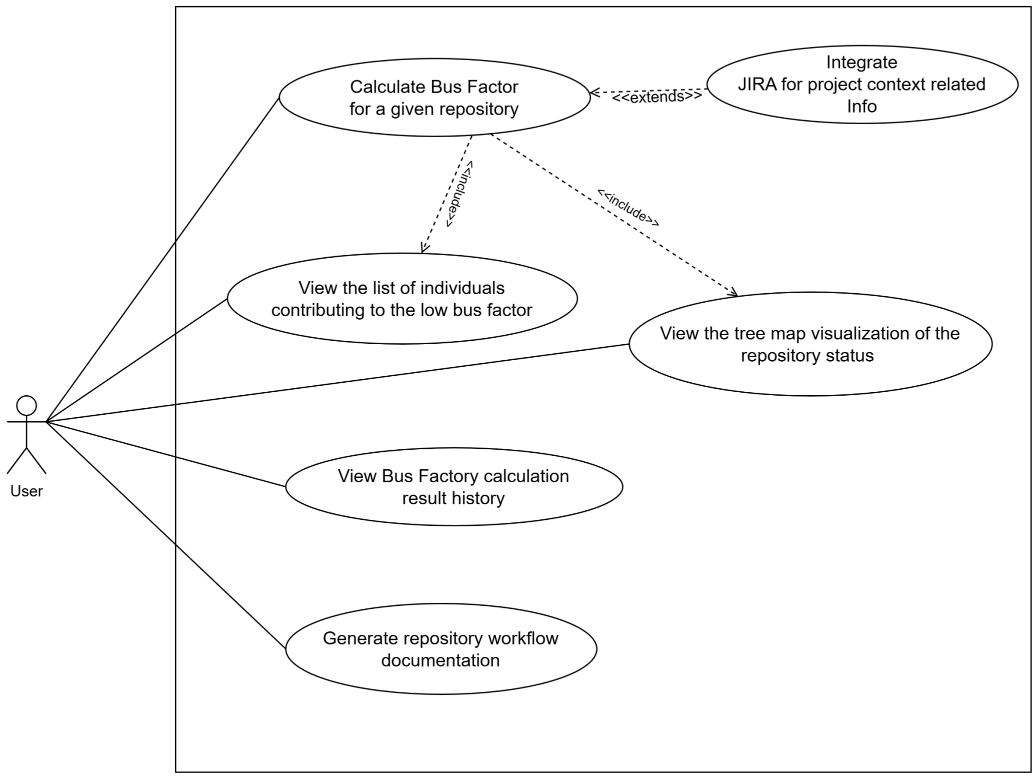

Calculate Bus Factor for a given repository
Integrate JIRA for project context related Info
<<extends>>
<<include>>
<<include>>
View the list of individuals contributing to the low bus factor
View the tree map visualization of the repository status
User
View Bus Factory calculation result history
Generate repository workflow documentation

Figure 3.7: Use Case Diagram

3.11.3 Activity Diagrams

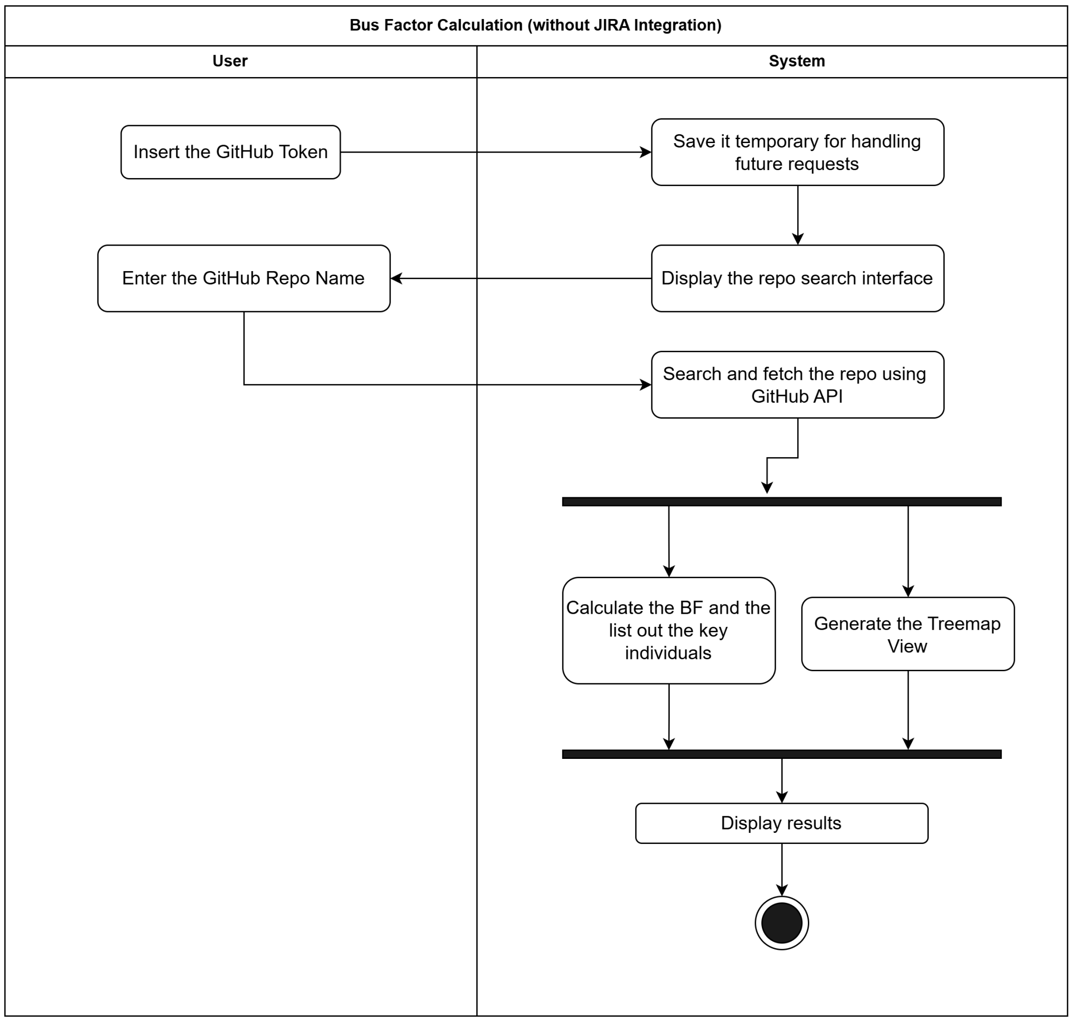

Bus Factor Calculation (without JIRA Integration)
User
System
Insert the GitHub Token
Save it temporary for handling future requests
Display the repo search interface
Enter the GitHub Repo Name
Search and fetch the repo using GitHub API
Calculate the BF and the list out the key individuals
Generate the Treemap View
Display results

Figure 3.8: Activity diagram - BF Calculation without JIRA Integration

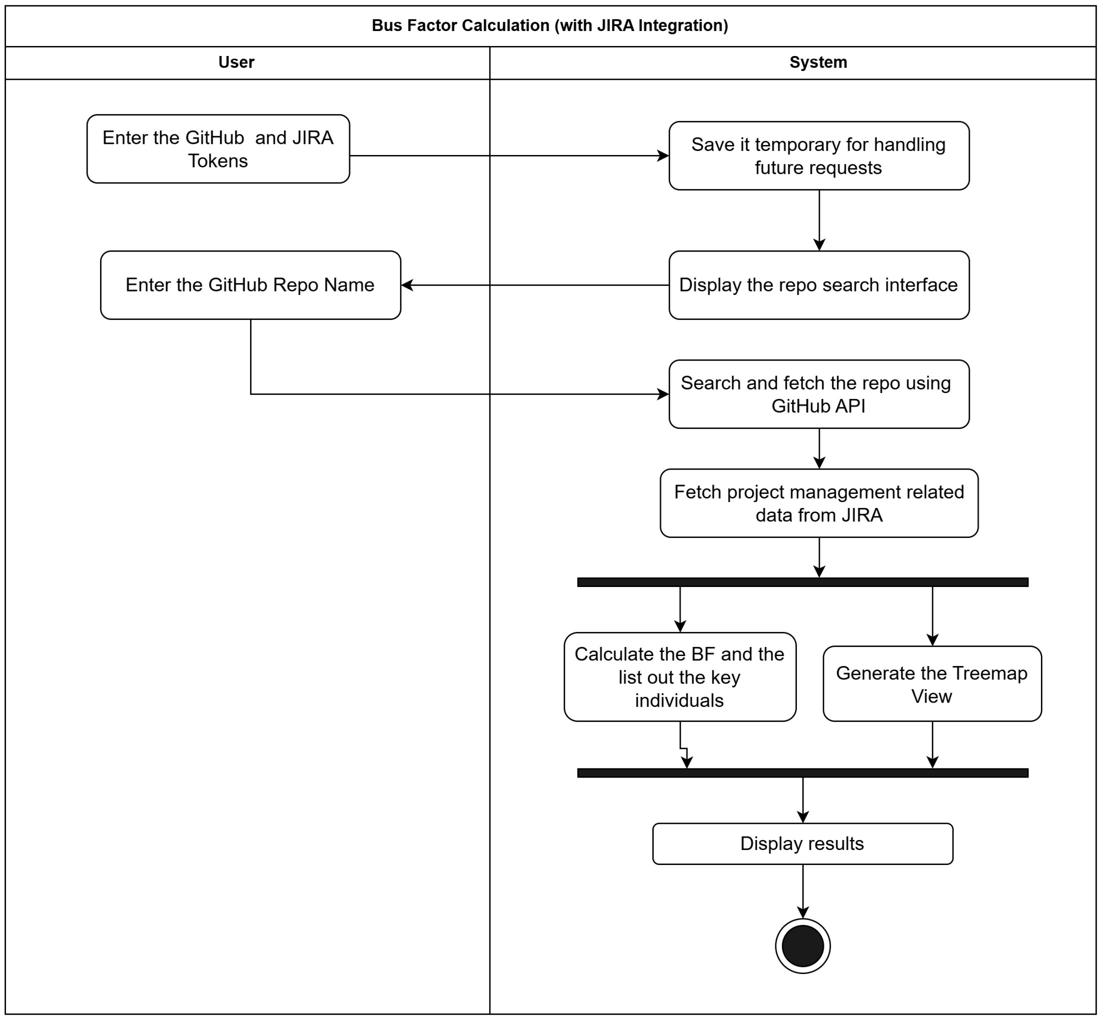

Bus Factor Calculation (with JIRA Integration)
User
System
Enter the GitHub and JIRA Tokens
Save it temporary for handling future requests
Enter the GitHub Repo Name
Display the repo search interface
Search and fetch the repo using GitHub API
Fetch project management related data from JIRA
Calculate the BF and the list out the key individuals
Generate the Treemap View
Display results

Figure 3.9: Activity diagram - BF Calculation with JIRA Integration

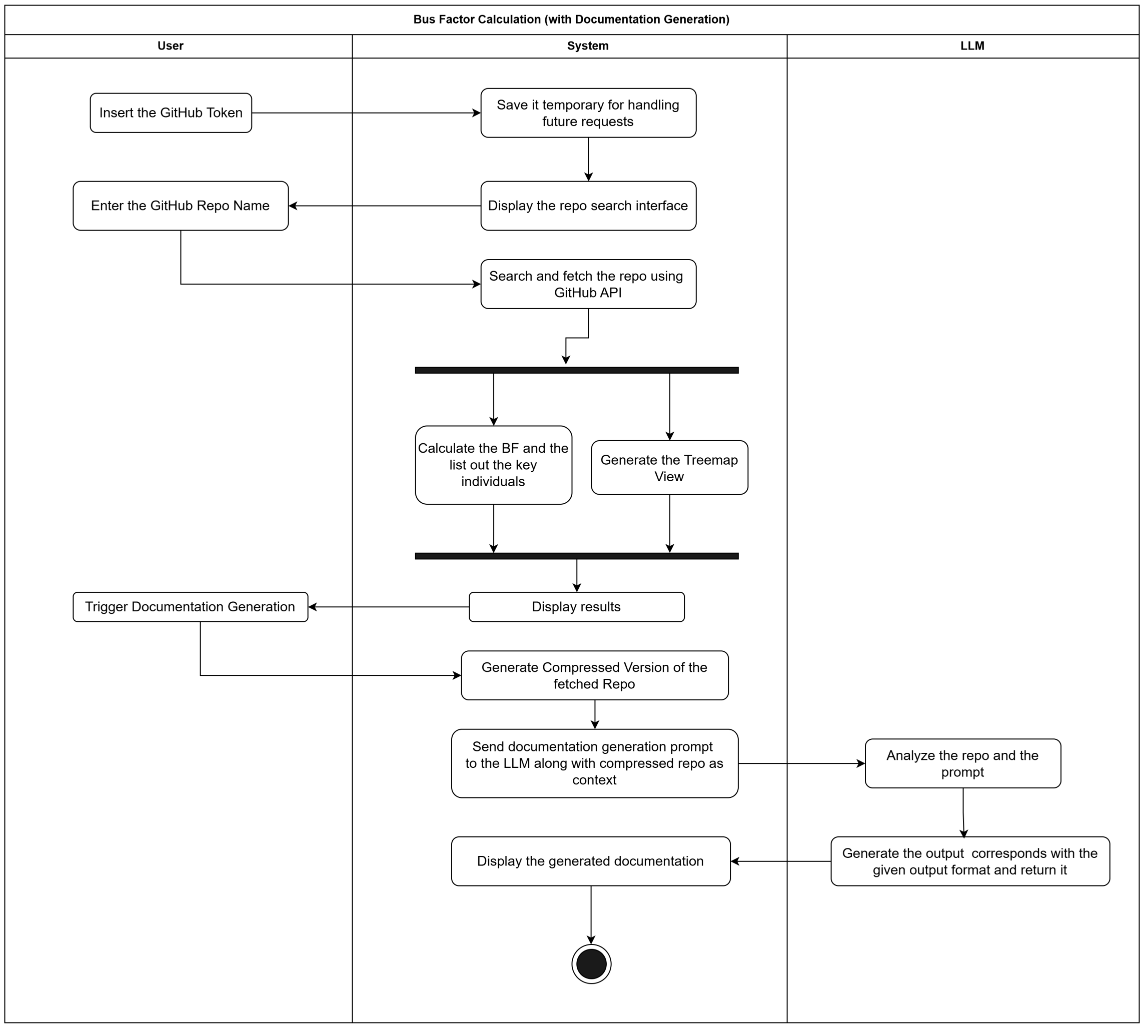

Bus Factor Calculation (with Documentation Generation)
User
System
LLM
Insert the GitHub Token
Save it temporary for handling future requests
Enter the GitHub Repo Name
Display the repo search interface
Search and fetch the repo using GitHub API
Calculate the BF and the list out the key individuals
Generate the Treemap View
Trigger Documentation Generation
Display results
Generate Compressed Version of the fetched Repo
Send documentation generation prompt to the LLM along with compressed repo as context
Analyze the repo and the prompt
Display the generated documentation
Generate the output corresponds with the given output format and return it

Figure 3.10: Activity diagram - BF Calculation and Documentation Generation

3.11.4 Functional Requirements

- Integrate with existing data sources (e.g., GitHub for VCS and Jira) to ingest relevant project data such as commit histories and issue logs.
- Analyze source data to compute developer-specific metrics, including bus factor scores and dependency profiles to pinpoint potential single points of failure.
- Generate a network graph visualization that illustrates developer interactions and dependency links across the project components.
- Produce a treemap visualization to depict the contribution percentages of key personnel relative to different project areas.

- Automatically create comprehensive project documentation that highlights critical components, workflows, and knowledge dependencies to support onboarding and knowledge transfer.

3.11.5 Quality Attributes

- **Usability:** The system should offer an intuitive interface that minimizes cognitive effort while ensuring users can easily navigate its features. Clear instructions and visual feedback must guide users in interpreting results (e.g., bus factor scores and contributor lists) without confusion.

- **Performance:** To deliver actionable insights, the system must efficiently generate dependency graphs, compute the bus factor, and identify key contributors with high accuracy. Timely processing is critical, especially for large projects, to ensure practicality in real-world use.

- **Security:** Since the system handles sensitive credentials (e.g., GitHub/JIRA tokens), data must be encrypted during transmission and storage. Secure authentication protocols and token management are essential to prevent unauthorized access or leaks.

- **Compatibility:** The system must seamlessly integrate with platforms like GitHub and JIRA to fetch project data (e.g., commits, issues). Standardized APIs and adaptive data parsing ensure reliable functionality across diverse project environments.

- **Maintainability:** Modular design and well-documented code enable future updates, such as adding new data sources or refining algorithms. This reduces technical debt and ensures long-term viability as project ecosystems evolve.

Chapter 4

Implementation

Building on insights from the preliminary analysis of existing methods and their limitations, this section presents a novel methodology for bus factor estimation. Leveraging social network metrics, contribution patterns, and also the project management data from JIRA, it addresses gaps identified earlier such as static assumptions and insufficient collaboration modeling. By visualizing team dynamics as a network of interactions and incorporating both quantitative measures—like lines of code diversity and unique file contributions—and contextual data from project management practices, this approach offers a dynamic and actionable framework for identifying critical contributors and enhancing project resilience.

4.1 Contribution Pattern Analysis

Contribution Pattern Analysis, grounded in Social Network Metrics, offers a structured method for examining the collaborative dynamics within software engineering teams. By visualizing a project as a network, where nodes represent developers and edges signify interactions (such as code reviews, shared file contributions, or collaborative tasks), this analysis helps reveal patterns in collaboration, knowledge distribution, and dependency structures within the project. This approach is particularly valuable in assessing the "bus factor" of a software project—the risk posed to project continuity if key contributors were to exit the team. Through Contribution Pattern Analysis, it is possible to move beyond surface-level contributions to capture a nuanced view of how knowledge flows within a project, identify potential vulnerabilities, and highlight candidates for knowledge transfer or mentorship. This analysis not only facilitates a deeper understanding of team dependencies but also supports the design of targeted interventions to mitigate potential risks. Other than that the following benefits can also be drawn via a successful contribution pattern analysis;

- Identification of key collaborators aka "central" developers who act as primary knowledge conduits within the team. These individuals are often highly connected and involved in multiple interactions, indicating their integral role in both knowledge-sharing and cross functional collaboration. Recognizing such individuals enables a clearer understanding of where crucial knowledge resides, and aids in planning for knowledge redundancy if such individuals are at risk of leaving.

- The network structure can reveal clusters, or “silos,” where knowledge may be concentrated among a subset of team members. Identifying such silos helps in gaining insight into where knowledge may not be sufficiently dispersed, posing a risk if team members within these silos become unavailable.
- Within most projects, few developers may act as the primary link between otherwise distinct groups or clusters within the team (bridges) to facilitate collaboration and information flow across groups. Turnover of such individuals will severely disrupt communication between clusters hence recognizing these bottlenecks enables proactive knowledge-sharing practices and mitigates risks associated with dependency on a single link.
- Highlight potential mentors—developers who, due to their position within the network, could efficiently transfer knowledge to less experienced team members.

4.2 Identification of Tools and Technologies

Front-End: Aligning with the required functionality which comprises of a considerable amount of dynamically changing components, React: an open-source JavaScript library, recognized for its efficiency in developing user interfaces was selected as the primary front-end framework. React emphasis on reusable components, enables developers to leverage readymade UI elements rather than constructing them from scratch and accelerates the development process. Also, by facilitating the incorporation of open-source community-driven components, compatibility with third-party libraries via Node Package Manager (NPM) enhances the system’s flexibility and extensibility, particularly for integrating visualization tools, ensuring a modular and maintainable frontend architecture.

Back-End: Due to its versatility and expressive syntax, Python makes it a preferred choice for diverse software development needs. Flask, a lightweight web framework, complements this by providing a streamlined foundation for building RESTful web services. The ecosystem enhances through an extensive collection of libraries and packages, enabling rapid feature implementation and the design principles of Flask set up great emphasis on flexibility and simplicity, making it easier for creating scalable RESTful APIs that guarantee solid, flexible communication.

NetworkX5: is a widely used Python library designed for complex network analysis and graph-based computations. It provides an efficient way to model relationships and analyze structural patterns within a dataset. NetworkX5 plays a key role in constructing a collaboration network among contributors and by leveraging its graph modeling capabilities, it ease the process of identifying key contributors (critical knowledge holders) using a custom centrality metric that combines several project level metrics.

Git and GitHub: Git is an essential version control system for managing code changes, while GitHub serves as a cloud-based platform for hosting repositories and enabling collaborative development. In this implementation, Git is utilized to retrieve commit history, analyze contributor activity, and extract code modification patterns directly from the repository. Meanwhile, GitHub's API provides structured access to contributor metadata, allowing the script to differentiate between human contributors and automated bot accounts. The integration of Git and GitHub ensures a comprehensive analysis of developer interactions, enabling the identification of core contributors and the assessment of the project's bus factor—a crucial metric for evaluating knowledge distribution and potential risks associated with developer turnover.

D3.js: is a powerful JavaScript library for producing dynamic, interactive data visualizations in web browsers. It provides extensive support for binding data to graphical elements and applying data-driven transformations. In the current implementation, D3.js is used to create an interactive graph network that visually represents the collaboration structure among contributors, allowing users to explore relationships between developers and their contributions. Additionally, the treemap visualization of codebase components with a low bus factor, makes it easier to identify high-risk areas where knowledge is concentrated among a few individuals.

RepoMix: RepoMix is a utility that transforms an entire software repository into a single, AI-friendly text file, optimized for consumption by modern Large Language Models (LLMs) such as ChatGPT, DeepSeek, Perplexity, Gemini, and others. In this research, RepoMix plays a key role in the documentation generation pipeline. The primary use is to prepare the codebase in a structured and context-rich format, enabling the LLM to better understand the relationships, workflows, and overall functionality within the project. By packaging the repository in this way, RepoMix ensures that the language model receives consistent and relevant information, which is crucial for generating meaningful and accurate technical documentation.

DeepSeek LLM: DeepSeek is a powerful large language model fine-tuned specifically for code understanding, reasoning, and generation tasks. It is designed to interpret complex code structures and deliver context-aware outputs, making it highly effective for software engineering use cases. In this case, DeepSeek is used in the automated documentation generation process with the help of contextual inputs prepared by RepoMix. It generates explanations of functionalities, module breakdowns, and workflows provided by that particular software project and the LLM adapts its output based on the given prompt structure and the organization of the codebase. Further technical details on how this is integrated and utilized within the documentation process are discussed in the corresponding subsection later in this chapter.

4.3 Basic Bus Factor Calculation

This section describes the basic bus factor calculating scenario which only incorporates the data collected via the version control history of a given repository. The process is further broken down into the following 7 steps.

Step 1: Data Collection and Preprocessing. In the first step, it begins with programmatically cloning the target repository into a temporary directory using the git Python3 library. Contributor data is then retrieved from GitHub through the GitHub via the PyGithub4 module. To ensure accurate and clean data, contributors are normalized by their usernames, with bot accounts being filtered out based on predefined criteria such as username patterns and metadata. To maintain consistency in analysis, the focus was limited to commits made within the last 1.5 years by calculating a cutoff date from the most recent commit. This preprocessing ensures the data clearly depicts the current state of the project while providing enough context also both relevant and manageable for further analysis.

Step 2: Normalization and Contributor Grouping. After retrieving the raw contributor data, further processing it is required by grouping contributors based on their normalized usernames. This step resolves inconsistencies caused by variations in names or aliases, such as the same individual using different emails or usernames. A fuzzy mapping is created to relate the normalized names with their corresponding original usernames and emails. This standardization ensures accurate representation and analysis of individual contributors.

Step 3: Building a Collaborative Network Graph. In this step, the construction of a collaborative network graph happens where each node represents a unique contributor. Commits are iterated over to examine the modified files, identifying contributors who have co-edited the same files. For every co-editing relationship, an undirected edge is added to the graph, with edge weights representing the number of shared file modifications between contributors. This graph models the collaboration dynamics within the repository, highlighting the interaction intensity between contributors.

Step 4: Calculating Lines of code (LOC) and File Diversity. Next the contribution patterns were analyzed by calculating the number of LOC modified in each commit, including both insertions and deletions. This data is attributed to the corresponding normalized contributors. Additionally, the unique files modified by each contributor is also tracked, providing a measure of file diversity. These metrics capture the breadth and depth of individual contributions to the repository.

Step 5: Custom Centrality Metric Calculation. Using the NetworkX5 library, degree centrality for each node in the graph is calculated, representing the direct collaboration ties of each contributor. Using that a custom centrality score is computed for each contributor, incorporating three factors: degree centrality, the proportion of LOC changed relative to the maximum LOC changed by any contributor, and the proportion of unique files modified relative to the maximum unique files modified by any contributor. Weighted factors, such as 0.5 for LOC and 0.5 for file diversity, are applied to balance these metrics. These weights were determined through continuous experimentation, where adjustments were made and results were compared against the ground truth to ensure optimal accuracy. This custom centrality score provides a comprehensive measure of each contributor's importance in the repository.

Step 6: Identifying Key Contributors. Contributors are ranked by their custom centrality scores in descending order to identify the most critical individuals. A threshold percentage of the total centrality (e.g., 30%) is used to determine the key contributors. By tracking the cumulative centrality score, it is able to identify the minimal set of contributors required to meet the threshold. This step isolates the essential contributors whose absence would significantly impact the project.

Step 7: Bus Factor Calculation and Visualization. The collaborative network graph is visualized using the D3.js library to provide an intuitive representation of contributor interactions. Key contributors are highlighted in the graph with a distinct color, while other nodes are depicted in a separate color. A treemap is also generated which contains tiles with a unique set of colors that helps to visualize the file-level contributions by the key developer(s) making it easiy to identify files and modules with high dependency on a such individuals.

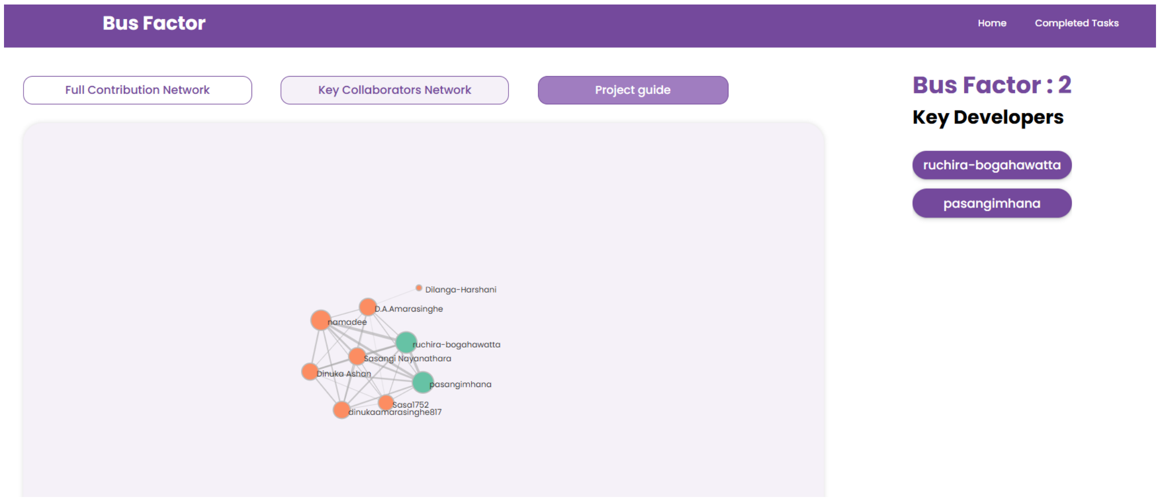


Figure 4.1: Visualization of the key collaboration network for the GitHub repository **team-ayura/Ayura-Server**.

As an example to further clarify the above above flow, figure 4.1 contains a the result obtained by running the bus factor calculation tool over a project. The network graph illustrates the interactions between contributors, with node size representing the level of contribution and edge thickness indicating collaboration strength. The highlighted nodes correspond to the two key developers—ruchira-bogahawatta and pasangimhna—whose absence would critically impact the project, resulting in a bus factor of 2.

Figure 4.2 is the corresponding treemap generated for the above example scenario and the colored blocks represent individual files and folders, with size proportional to the Lines of Code (LOC) (depicted on top of each block) of each file/folder.

- Orange blocks denote contributing files, where the key contributor has made significant code contributions.
- Light grey blocks indicate non-contributing files, depicting files with no contributions.
- Green blocks represent folders, which are clickable to reveal nested files.
- The top yellow bar indicates the current root folder path within the treemap.

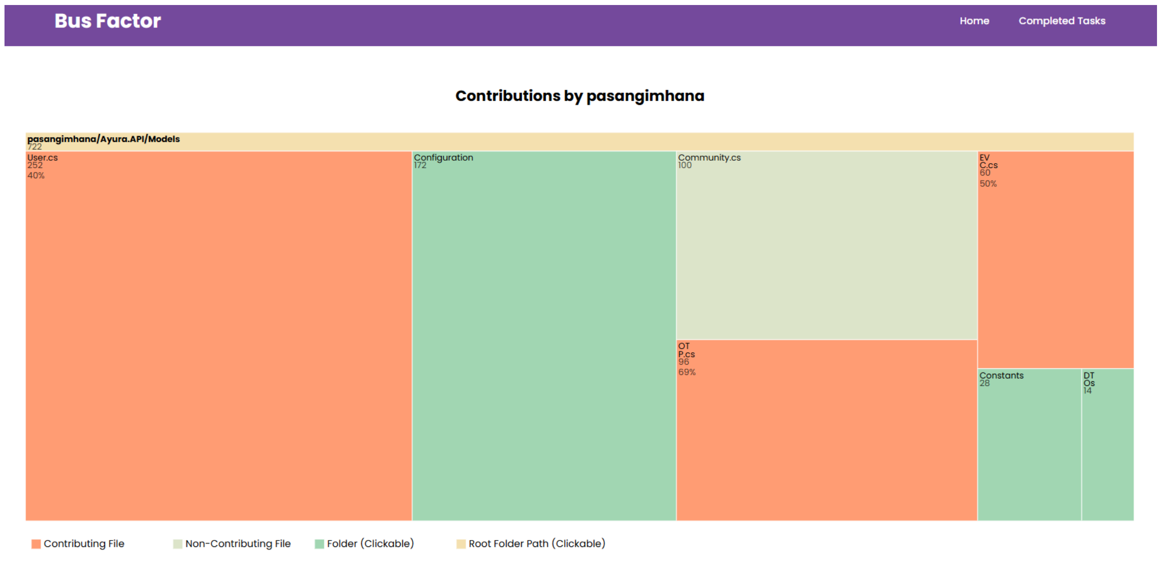


Figure 4.2: Treemap visualization of file-level contributions by the key developer *pasangimhana* within the **Ayura.API/Models** directory of the GitHub repository **team-ayura/Ayura-Server**.

This visualization helps identify files and modules with high dependency on a specific developer, which is crucial for evaluating the bus factor.

Algorithm 1 sums up the exact same set of steps outlined above, which will eventually list out the bus factor for the given project and the list of key individuals who contribute to the bus factor.

Algorithm 1 Key Developer Identification Algorithm

```
if repository_url is provided then
    repo_name ← extract_repository_name(repository_url)
    github_connection ← authenticate_with_token(GITHUB_TOKEN)
end if
temp_dir ← create_temporary_directory()
cloned_repo ← clone_repository(repository_url, temp_dir)
contributors ← fetch_contributors(github_connection, repo_name)
for all contributor in contributors do
    contributor_data[contributor.login] ← {type: contributor.type, normal-
ized_name: normalize_username(contributor.login)}
    if contributor.email exists then
        email_to_username[contributor.email] ← contributor.login
    else if contributor.name exists then
        name_to_username[contributor.name] ← contributor.login
    end if
end for
most_recent_commit ← get_most_recent_commit(cloned_repo)
cutoff_date ← most_recent_commit.date − 547 days
for all commit in cloned_repo.commits do
    if commit.date < cutoff_date then
        break
    end if
    author ← resolve_author(commit.author, email_to_username,
name_to_username)
    if is_bot(author) then
        continue
    end if
    normalized_author ← normalize_username(author)
    loc_per_contributor[normalized_author] + = commit.total_lines_changed
    unique_files_per_contributor[normalized_author].add(commit.modified_files)
end for
collaboration_graph ← create_empty_graph()
for all commit in recent_commits do
    author ← normalize_username(commit.author)
    contributor_map[author].add(commit.author)
    for all file in commit.modified_files do
        file_contributors[file].add(author)
    end for
end for
for all normalized_name in contributor_map do
    representative ← get_representative(contributor_map[normalized_name])
    add_node(collaboration_graph, representative)
end for
for all file in file_contributors do
    contributors ← file_contributors[file]
    for all pair in combinations(contributors, 2) do
        if edge_exists(collaboration_graph, pair) then
            increment_edge_weight(collaboration_graph, pair)
        else
            add_edge(collaboration_graph, pair, weight=1)
        end if
    end for
end for
for all node in collaboration_graph.nodes do
    centrality ← calculate_degree_centrality(node)
    loc_score ← normalize(loc_per_contributor[node])
    file_score ← normalize(unique_files_per_contributor[node].count)
    custom_centrality[node] ← centrality + 0.5 × loc_score + 0.5 × file_score
end for
sorted_contributors ← sort_by_centrality(custom_centrality)
top_developers ← select_top_contributors(sorted_contributors, threshold=0.3)
visualize_network(collaboration_graph, highlight_nodes=top_developers)
print("Bus Factor:", top_developers.count)
print("Top Key Developers:", top_developers.names)
delete_directory(temp_dir)
```

4.4 Enhanced Bus Factor Calculation with Jira Integration

This section outlines the enhanced approach to bus factor calculation, extending the original version control-based method with integrated Jira activity analysis. This integration provides a more comprehensive perspective by including non-code contributions, such as issue reporting, task assignments, and comments, which are crucial to collaborative development. The enhanced process consists of the following steps;

Step 1: Data Collection and Preprocessing. As in the basic method, the target Git repository is cloned and contributor data is retrieved via the GitHub API using the PyGithub module. Contributors are normalized, and bots are excluded based on metadata and naming patterns. Commits from the past 1.5 years are considered to ensure relevance.
In addition to GitHub data, Jira issues related to the target project are fetched via the Jira REST API. The query filters issues updated within the same time frame as the Git data, ensuring temporal consistency.

Step 2: Contributor Normalization Across Platforms. To integrate Jira data with GitHub contributions, contributor identities from both platforms are normalized using fuzzy matching on display names and email patterns. A unified identifier is created for contributors to ensure that activities from both systems are attributed correctly.

Step 3: Collaborative Network Graph Construction. A graph is constructed with each node representing a normalized contributor. Edges are drawn between nodes where contributors have co-modified the same file (from Git) or interacted on the same issue (from Jira). Edge weights are incremented for every shared interaction, combining file edits and Jira-based collaborations.

Step 4: Activity and Diversity Analysis. This step computes LOC changes and file diversity for each contributor based on commit history. Simultaneously, Jira activities such as issue reporting, task assignment, and commenting are tallied. A composite activity score is created, capturing the contributor's coding and project management involvement.

Step 5: Custom Centrality Score with Jira Weighting A custom centrality score is computed using the NetworkX library, extending the basic formula by incorporating Jira activity. The score includes:

- Degree centrality from the graph.
- LOC contribution proportion.

- File diversity proportion.
- Jira activity score, scaled using a tunable Jira weight (e.g., 5 for GitHub-linked users, 0.2 for Jira-only users).

These components are combined to yield a holistic score of contributor importance.

Step 6: Key Contributor Identification. Contributors are ranked based on their updated centrality scores. A threshold (e.g., top 30% of cumulative centrality) is used to identify the key contributors who form the core bus factor of the project. This approach ensures contributors with significant Jira activity but limited commits are not overlooked.

Step 7: Graph Visualization and Result Output The final network graph is rendered using D3.js, with key contributors visually highlighted. Graph metadata and metrics (including Jira-based scores) are exported in JSON format for integration into dashboards or further reporting. integrating broader collaborative signals, better reflecting the diverse roles in modern software development teams.

4.4.1 Architecture

- **Ingestion** – A single pass over the Jira REST API downloads issues updated within the same time window as the Git history.
- **Identity Resolution** – Fuzzy matching aligns Jira display names with GitHub usernames to build a unified contributor registry.
- **Fusion Layer** – Jira activities are merged with Git-derived graph metrics, producing a composite centrality score.

4.4.2 Jira Integration Overview

Purpose and Rationale Software development work extends far beyond source-code commits. Issue tracking systems capture critical knowledge such as bug triage, feature planning, task assignment, discussion, and validation. By incorporating Jira activity, our model recognises contributors who may write little code but still hold indispensable domain or coordination knowledge.

Algorithm 2 Key Developer Identification with GitHub and Jira Integration

```
if repo URL is provided then
    Extract repo name and authenticate GitHub with token
end if
Clone repository to temp directory
Fetch all contributors via GitHub API
for all contributor in contributors do
    Normalize contributor's username
    Store contributor metadata (type, login)
end for
Determine cutoff date = most recent commit - 547 days
for all commit in repository do
    if commit date < cutoff date then break
    end if
    Resolve and normalize author's username
    Update LOC and unique files for the contributor
    Track file-level changes for all contributors
end for
Create collaboration graph $G$
for all commit in recent commits do
    Normalize author and track file contributions
    for all file in commit.files do
        Add author to file contributors
    end for
end for
for all normalized name in contributor map do
    Add a node with representative name to $G$
end for
for all file in file contributors do
    for all pairs of contributors to that file do
        if edge exists then
            increment weight
        else
            create edge with weight 1
        end if
    end for
end for
Calculate degree centrality for each node
Compute custom centrality using:
   degree $+ 0.5 \times$ normalized LOC $+ 0.5 \times$ normalized file diversity
Fetch Jira issues from the past 1.5 years
Score Jira activity: +1 for reporter/assignee/comment
for all GitHub nodes in graph do
    Add weighted Jira activity to their centrality
end for
for all Jira-only contributors do
    if no match in GitHub nodes then
        Add as new node with low weighted centrality
    end if
end for
Sort contributors by centrality
Select top contributors whose cumulative centrality $\geq 30\%$ of total
Label top nodes as Key Developers (class = 1), others as class = 2
Generate graphs, calculate file sizes and contribution percentages
Return all graphs and contribution mappings
```

4.5 Documentation Generation to Improve On-boarding of New Developers

Efficient on-boarding of new developers is crucial for maintaining productivity and reducing ramp-up time in software projects. In this context, generating comprehensive documentation automatically can greatly aid in familiarizing new developers with the project's main workflows, components, and functionalities. This section describes the implementation of an automated documentation generation approach that leverages the DeepSeek API[1] in conjunction with RepoMix[2]. The goal is to provide new developers with an understanding of the codebase's structure, workflows, and interdependencies.

4.5.1 RepoMix and DeepSeek Integration

To automate the documentation generation process, the code repository is first processed by RepoMix, a tool that packages the entire codebase into a single, AI-friendly file. This file, typically in the XML format, contains metadata about the repository, including class structures, function definitions, dependencies, and other relevant information. Once the repository is packaged, the XML file is passed to the DeepSeek API, which analyzes the codebase and generates a JSON-formatted output that serves as a comprehensive guide for new developers.

4.5.2 JSON Output Structure

The JSON output generated by DeepSeek follows a fixed structure designed to provide clear insights into the codebase. The key attributes of the JSON output are as follows:

- **Project Overview**: A brief description of the project's purpose, goals, and scope. This section introduces new developers to the overall context of the project.
- **Module Breakdown**: A hierarchical breakdown of the project's modules. It categorizes components into logical sections:
  - **Components**: Defines key components of the project and their functionalities.
  - **Views**: Describes the various views or UI components that interact with the user.
  - **Global**: Includes globally relevant modules or utilities used across the project.

[1]https://api-docs.deepseek.com/
[2]https://repomix.com/guide/

- **Key Workflows**: A list of the project's main workflows that define the sequence of events in achieving critical functionalities.
- **Key Functionalities**: A list of the project's core functionalities, with a detailed breakdown of their associated flow of events. Each functionality specifies the sequence of functions or methods invoked during its execution.
- **Critical Dependencies**: Identifies key dependencies between classes, functions, or modules that are crucial for understanding how the codebase operates and how different parts of the system interact.

4.5.3 Prompt used in DeepSeek API

This section describes the prompt used to generate structured and comprehensive documentation that highlights the project's key components, workflows, functionalities, and critical dependencies.

4.5.3.1 Prompt

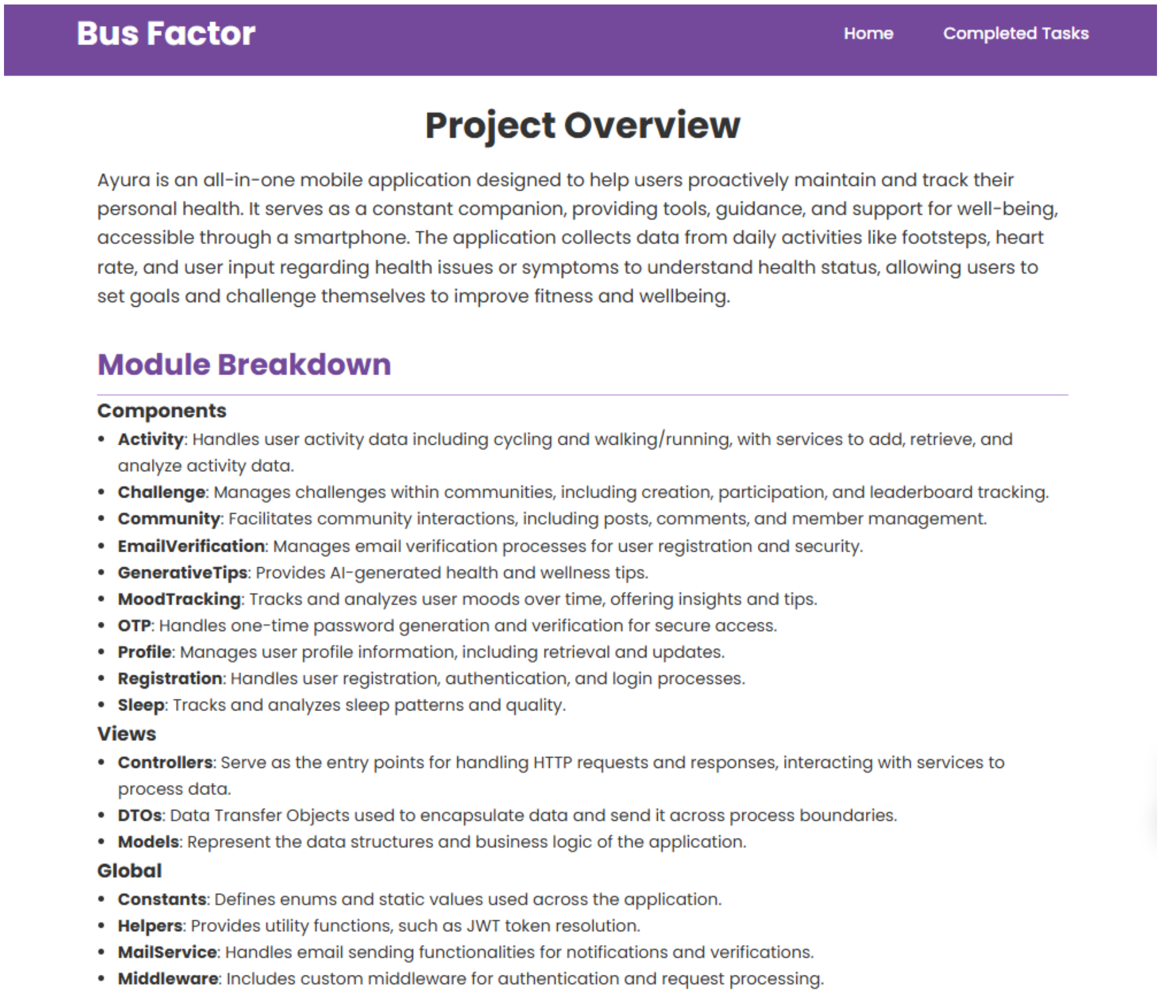
Analyze the xml input containing data of a github repository and generate a JSON formatted output according to given json format below, which will help new developers to understand the inner working of the code when onboarding. The "flowOfEvents" attribute defines what is the flow when providing the functionality for the particular functionality(what classes and functions that will be invoked and all)

```
JSON FORMAT
{
  "projectOverview": "",
  "moduleBreakdown": {
    "Components": {},
    "Views": {},
    "Global": {}
  },
  "keyWorkflows": {
    "MainAnalysisFlow": {},
    "DataProcessingFlow": {}
  },
  "keyFunctionalities": [
    {
      "functionality": "",
      "flowOfEvents": {}
    }
  ],
  "criticalDependencies": []
}
```

```
EXAMPLE FLOW OF EVENTS
{
  "flowOfEvents": {
    "PasswordResetManager.initiatePasswordReset()":
      "Initiates the password reset process.",
    "TokenGenerator.generateToken()":
      "Generates a password reset token.",
    "NotificationService.send()":
      "Sends the password reset token to the user.",
    "PasswordResetManager.validateResetToken()":
      "Validates the password reset token.",
    "PasswordResetManager.completePasswordReset()":
      "Completes the password reset process."
  }
}
```

The `flowOfEvents` attribute within the JSON output provides a detailed sequence of method calls and their corresponding actions for each key functionality. This attribute helps new developers understand the internal logic of the codebase by showing the order in which functions are invoked during the execution of specific tasks.

Each function is accompanied by a concise description of its role in the overall process, which helps developers quickly understand the flow and dependencies between different parts of the code.



Project Overview

Ayura is an all-in-one mobile application designed to help users proactively maintain and track their personal health. It serves as a constant companion, providing tools, guidance, and support for well-being, accessible through a smartphone. The application collects data from daily activities like footsteps, heart rate, and user input regarding health issues or symptoms to understand health status, allowing users to set goals and challenge themselves to improve fitness and wellbeing.

Module Breakdown

Components

- **Activity**: Handles user activity data including cycling and walking/running, with services to add, retrieve, and analyze activity data.
- **Challenge**: Manages challenges within communities, including creation, participation, and leaderboard tracking.
- **Community**: Facilitates community interactions, including posts, comments, and member management.
- **EmailVerification**: Manages email verification processes for user registration and security.
- **GenerativeTips**: Provides AI-generated health and wellness tips.
- **MoodTracking**: Tracks and analyzes user moods over time, offering insights and tips.
- **OTP**: Handles one-time password generation and verification for secure access.
- **Profile**: Manages user profile information, including retrieval and updates.
- **Registration**: Handles user registration, authentication, and login processes.
- **Sleep**: Tracks and analyzes sleep patterns and quality.

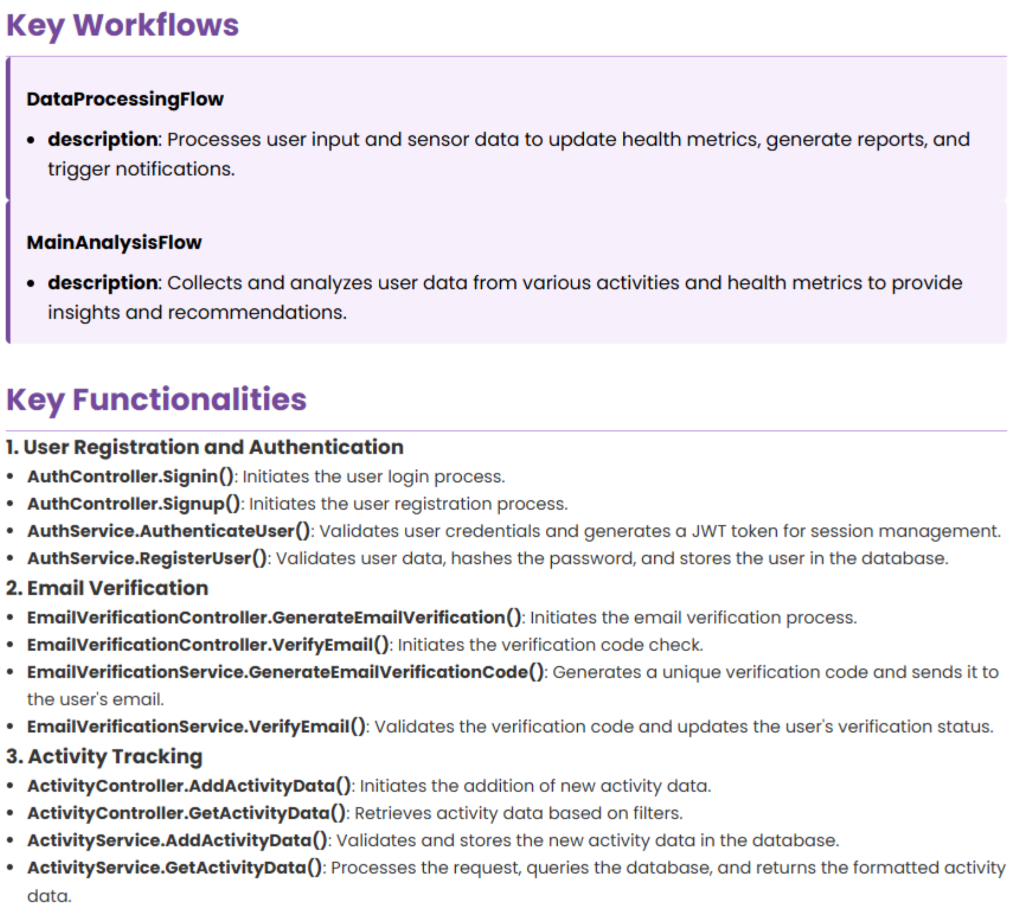
Views

- **Controllers**: Serve as the entry points for handling HTTP requests and responses, interacting with services to process data.
- **DTOs**: Data Transfer Objects used to encapsulate data and send it across process boundaries.
- **Models**: Represent the data structures and business logic of the application.

Global

- **Constants**: Defines enums and static values used across the application.
- **Helpers**: Provides utility functions, such as JWT token resolution.
- **MailService**: Handles email sending functionalities for notifications and verifications.
- **Middleware**: Includes custom middleware for authentication and request processing.

Figure 4.3: Documentation generated for the GitHub repository **team-ayura/Ayura-Server**



Key Workflows

DataProcessingFlow

- **description**: Processes user input and sensor data to update health metrics, generate reports, and trigger notifications.

MainAnalysisFlow

- **description**: Collects and analyzes user data from various activities and health metrics to provide insights and recommendations.

Key Functionalities

1. User Registration and Authentication

- **AuthController.Signin()**: Initiates the user login process.
- **AuthController.Signup()**: Initiates the user registration process.
- **AuthService.AuthenticateUser()**: Validates user credentials and generates a JWT token for session management.
- **AuthService.RegisterUser()**: Validates user data, hashes the password, and stores the user in the database.

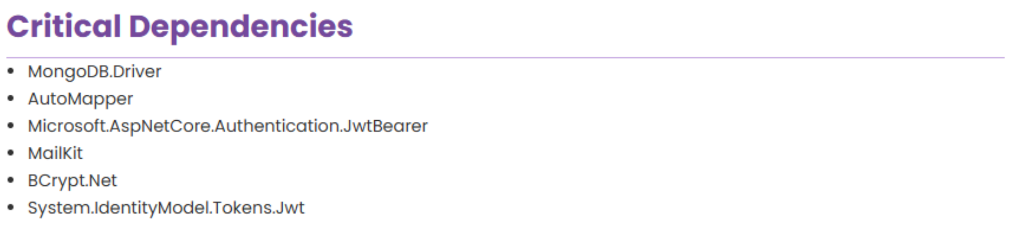
2. Email Verification

- **EmailVerificationController.GenerateEmailVerification()**: Initiates the email verification process.
- **EmailVerificationController.VerifyEmail()**: Initiates the verification code check.
- **EmailVerificationService.GenerateEmailVerificationCode()**: Generates a unique verification code and sends it to the user's email.
- **EmailVerificationService.VerifyEmail()**: Validates the verification code and updates the user's verification status.

3. Activity Tracking

- **ActivityController.AddActivityData()**: Initiates the addition of new activity data.
- **ActivityController.GetActivityData()**: Retrieves activity data based on filters.
- **ActivityService.AddActivityData()**: Validates and stores the new activity data in the database.
- **ActivityService.GetActivityData()**: Processes the request, queries the database, and returns the formatted activity data.

Figure 4.4: Documentation of **team-ayura/Ayura-Server** continued



Critical Dependencies

- MongoDB.Driver
- AutoMapper
- Microsoft.AspNetCore.Authentication.JwtBearer
- MailKit
- BCrypt.Net
- System.IdentityModel.Tokens.Jwt

Figure 4.5: Documentation of **team-ayura/Ayura-Server** continued

Chapter 5

Evaluation and Results

5.1 Chapter overview

This chapter outlines the evaluation process undertaken for the research focused on calculating the bus factor and the accompanying software application. A systematic approach combining qualitative and quantitative methodologies was employed to thoroughly assess the effectiveness, usability, and accuracy of the proposed solution. The chapter provides details on the specific evaluation questions guiding the assessment, elaborates on the methodologies used, and discusses the insights and outcomes derived from the evaluation process.

5.2 Evaluation Questions

The evaluation was guided by the following criteria and associated questions:

- **Accuracy:** How accurately does the proposed method calculate the bus factor?
- **Usability:** How user-friendly and accessible is the software application to end-users?
- **Scalability:** How well does the proposed solution handle increasing data volumes and complexity?
- **Generalizability:** How effectively can the approach be applied across different software projects and team sizes?
- **Performance:** How efficiently does the software perform under typical operating conditions?
- **Strengths and Weaknesses:** What are the key strengths and limitations identified through the evaluation of the proposed solution?

5.3 Evaluation of Graph-Based Bus Factor Calculation Method

The graph-based method introduced in this research employs centrality metrics derived from contributions to source code repositories and Jira activity, integrated through a network graph analysis. The evaluation of this method was conducted in two primary stages: an initial assessment using university-based software projects, followed by an extended evaluation incorporating feedback from both open-source and industry software developers.

Repository	**Explorer**		**Network GraphGen**	
	Accuracy (%)	**F1-Score (%)**	**Accuracy (%)**	**F1-Score (%)**
Health-Project-Y3/frontend	100.00	100.00	100.00	100.00
Estructura-frontend	33.33	50.00	33.33	50.00
Estructura-backend	50.00	66.67	50.00	66.67
TruEvent_Horizons	50.00	66.67	100.00	100.00
CommuSupport	50.00	66.67	66.67	80.00
gasify	100.00	100.00	33.33	50.00
Ayura-Server	50.00	66.67	100.00	100.00
VentureVerse	50.00	66.67	66.67	80.00
FoodForALL	33.33	50.00	33.33	50.00
CosmiX	50.00	66.67	100.00	100.00
LIFELINE	100.00	100.00	100.00	100.00
Verdur	100.00	100.00	100.00	100.00

Table 5.1: Per-project performance metrics for Explorer vs. Network GraphGen

5.3.1 Initial Evaluation on University Projects

The preliminary evaluation was carried out on a selection of university projects, involving direct feedback from project developers. The summarized results from these projects are presented in the tables 5.1, 5.2 and 5.3 illustrating a comparison between the developers' self-reported bus factors, the results from the existing Explorer method, and the proposed Graph-based method.

Key insights from the preliminary results include:

- High alignment in bus factor results for certain repositories (e.g., Health-Project-Y3/frontend).
- Differences in contributor identification and bus factor values highlighting the sensitivity and specificity of the graph-based approach compared to the Explorer method.

Repo Name	Bus Factor (Dev)		Explorer		Network GraphGen	
	BF	Contributors	BF	Contributors	BF	Contributors
Health-Project-Y3	2	DDH13, Pawandi-W	2	DDH13, Pawandi-W	2	DDH13, Pawandi-W
Estructura-frontend	2	dewmni, sachin-um	2	Pubudu-Anuradha, dewmni	2	sachin-um, Pubudu-Anuradha
Estructura-backend	2	sachin-um, pubudu	1	sachin-um	1	sachin-um
TruEvent_Horizons	2	saneru-akarawita, Chirasi Amaya	1	saneru-akarawita	2	saneru-akarawita, Chirasi Amaya
CommuSupport	2	Thamuditha, pasangimhanna	1	Thamuditha	3	Thamuditha, THAMUDITHA P V GS, pasangimhanna
gasify	1	Dinuka Ashan	1	Dinuka Ashan	2	dinuka817, Dinuka Ashan Amarasinghe
Ayura-Server	2	ruchira-bogahawatta, pasangimhanna	1	pasangimhanna	2	ruchira-bogahawatta, pasangimhanna
VentureVerse	2	Samindu, pasindufernando1	1	Samindu	3	chrisperera1999, pasindufernando1, Samindu
FoodForALL	2	harini-udeshika, akiladharmadasa	2	akiladharmadasa, Amandi	2	harini-udeshika, anjuna0305
CosmiX	2	Sandul, Dinuka Ashan	2	Sandul	2	Sandul, Dinuka Ashan
LIFELINE	2	Pasindu Fernando, Shinthujen-I	2	Shinthujen-I, Pasindu Fernando	2	Pasindu Fernando, Shinthujen-I
Verdur	2	vihanga-sen, JDPrabasha	2	vihanga-sen, JDPrabasha	2	vihanga-sen, JDPrabasha

Table 5.2: Result Comparison between the ground truth bus factor values, results from the existing Explorer method, and the proposed Graph-based method

Tool	Accuracy (%)	F1-Score (%)
Explorer BF	59.72	69.44
Network GraphGen BF	70.14	78.06

Table 5.3: Average Performance Metrics for Tools using Bus Factor (Dev) as Ground Truth

5.3.2 Extended Evaluation on Industry Projects

The extended evaluation was conducted using 12 industry projects depicted in the tables 5.4 and 5.5 from various organizations across multiple countries, encompassing diverse team sizes and repository characteristics. Each repository was tested five times to ensure consistency and mitigate the impact of transient fluctuations in activity data. This rigorous process provided comprehensive validation of the graph-based method's effectiveness across varied real-world scenarios. The results were highly encouraging, consistently demonstrating the method's ability to identify key contributors accurately and reliably in practical settings.

Project	Ground-Truth Contributors	Bus Factor	Key Contributors Identified by Graph Method	Bus Factor by Graph Method
P1	Dev A, Dev B, Dev C	3	Dev A, Dev B, Dev C	3
P2	Dev D, Dev E, Dev F	3	Dev D, Dev E, Dev F, Dev G	4
P3	Dev H, Dev I, Dev J, Dev K	4	Dev H, Dev I, Dev J, Dev K, Dev L	5
P4	Dev M, Dev N, Dev O, Dev P, Dev Q, Dev R, Dev S, Dev T, Dev U, Dev V, Dev W, Dev X	12	Dev M, Dev N, Dev O, Dev P, Dev Q, Dev R, Dev S, Dev T, Dev U, Dev V, Dev W, Dev X, Dev Y, Dev Z, Dev AA	15
P5	Dev AB, Dev AC	2	Dev AB, Dev AC	2
P6	Dev AD	1	Dev AD	1
P7	Dev AE	1	Dev AE	1
P8	Dev AF	1	Dev AF, Dev AG	2
P9	Dev AH, Dev AI	2	Dev AH, Dev AI, Dev AJ	3
P10	Dev AK	1	Dev AK	1
P11	Dev AL, Dev AM	2	Dev AL, Dev AM	2
P12	Dev AN, Dev AO, Dev AP	3	Dev AN, Dev AO, Dev AP, Dev AQ	4

Table 5.4: Industry projects (anonymised): ground truth vs. graph-based method results

Project	TP	FP	FN	Accuracy (%)
P1	3	0	0	100.00
P2	3	1	0	75.00
P3	4	1	0	80.00
P4	12	3	0	80.00
P5	2	0	0	100.00
P6	1	0	0	100.00
P7	1	0	0	100.00
P8	1	1	0	50.00
P9	2	1	0	66.67
P10	1	0	0	100.00
P11	2	0	0	100.00
P12	3	1	0	75.00

Table 5.5: True positives (TP), false positives (FP), false negatives (FN), and accuracy per project.

Across the twelve anonymised industry projects, our graph-based method achieved an average accuracy of 85.56% (Table 5.5).

5.3.3 Simulation-Based Robustness Testing

In addition to empirical evaluations, extensive robustness testing was conducted using simulated Jira boards and GitHub repositories. These simulations systematically varied key parameters and collaboration scenarios—ranging from equal contributions, highly skewed contribution distributions, to fluctuating activity levels (Table 5.6). The aim was to statistically validate the fairness, stability, and reliability of the graph-based method.

To quantify performance in each simulation, we employed the metric:

$$\text{Accuracy} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives} + \text{False Positives}} \times 100\% \tag{5.1}$$

Consistently high accuracy scores across all scenarios (Table 5.7) confirm that the method provides a robust and unbiased identification of key contributors under diverse hypothetical conditions.

1. **Equal Contributions -** All contributors contribute equally with the same number of commits.

2. **Skewed Contributions -** One or two contributors dominate most of the commits while others contribute minimally.

3. **Mixed Activity -** Contributors participate at different levels, with some highly active and others less active.

4. **Sparse Activity -** Very few commits overall with contributors rarely participating.

5. **Dense Collaboration -** All contributors frequently edit shared files with high collaboration and file overlap.

Project	**Ground truth**	**Bus Factor**	**Key contributors identified by graph-based method**	**Bus Factor by Graph method**
Equal-simulation	Grace, Frank, Eve, Dana, Charlie, Bob, Alice	7	Dana, Charlie, Bob, Alice	4
Skewed sim	Alice, Charlie, Bob	3	Alice, Charlie, Bob	3
Dense activity	Frank, Eve, Charlie, Bob, Alice	5	Eve, Charlie, Bob, Alice	4
Mixed activity	Eve, Alice, Bob	3	Eve, Alice, Bob	3
Sparse activity	Alice, Bob	2	Alice, Bob, Dana	3

Table 5.6: Comparison of Ground Truth vs Graph-based Key Contributor Identification and Bus Factor Estimation

Scenario	**Accuracy (%)**
Equal-simulation	57.00
Skewed sim	100.00
Dense activity	80.00
Mixed activity	100.00
Sparse activity	66.67
Average	**80.76**

Table 5.7: Accuracy of Graph-based Methods under various contribution scenarios

5.3.4 Performance Metrics

As depicted in the table 5.8 and figure 5.1 performance metrics quantitatively demonstrate computational efficiency (for the repositories that have been tested with ground truth)

Metric	Description	time (s)
Average Computation Time	Typical analysis duration per repository	918
Maximum Computation Time	Longest observed analysis duration	4958
Minimum Computation Time	Shortest observed analysis duration	50
Scalability Efficiency	Performance consistency with increased data volume	90%

Table 5.8: Graph-based Method: Computation Time and Scalability Metrics

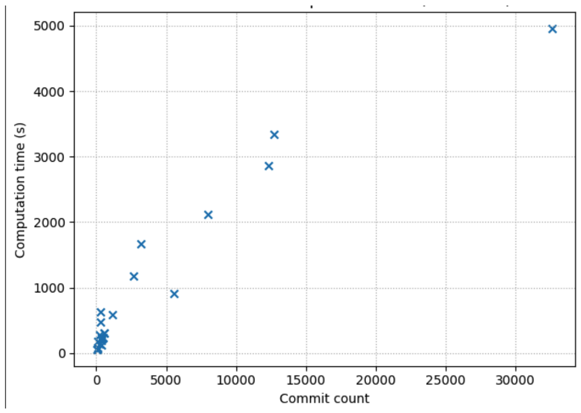


Figure 5.1: Commit Count vs Computation Time

5.3.5 User Feedback on Usability

The usability of the system was evaluated using the System Usability Scale (SUS), a standardized questionnaire designed to assess the overall user experience and satisfaction. The evaluation involved 20 participants, comprising 10 industry professionals and 10 final-year university students in computer science.

Industry participants included a diverse group of senior developers, DevOps engineers, project managers, and software architects, while the university group consisted of students with prior experience in team-based software development projects.

Each participant completed the SUS questionnaire after interacting with the system. The SUS score was computed based on their responses, resulting in an **average score of 86.7 out of 100**. According to established SUS benchmarks, this score corresponds to an **"Excellent"** usability rating, indicating high user satisfaction and effectiveness of the interface.

In addition to the SUS score, qualitative feedback highlighted the clarity of the interface, ease of interpreting centrality scores, and the overall usefulness of the system in identifying key developers (Table 5.9).

User Feedback Metric	Description	Satisfaction Score (%)
Interface Clarity	Ease of interpretation and interaction	82%
Centrality Breakdown	Understanding of detailed centrality measures	83%
Overall Usability	General satisfaction with the system usage	90%

Table 5.9: User Feedback Scores for Key System Aspects

5.3.6 Generalizability Across Projects

The graph-based method was designed to be language-agnostic, relying solely on version control metadata, contribution metrics, and issue tracking data, rather than the syntax or semantics of specific programming languages. This makes the method applicable across a wide range of software projects written in different languages such as Java, Python, C++, JavaScript, and more. During evaluations, the method was successfully applied to repositories in multiple programming languages without any need for language-specific modifications. This further supports its generalizability and practical adaptability in both academic and professional environments.

Generalizability showcases that the proposed method maintains relevance and effectiveness across diverse contexts, suggesting potential for broad industry adoption.

5.4 Results

This section addresses the outcomes related to the research questions formulated at the beginning of this study, synthesizing findings derived from both quantitative analyses and qualitative feedback.

5.4.1 Identification of Potential Single Points of Failure

The primary research question aimed to determine how potential single points of failure (key personnel whose absence significantly impacts a project) could be effectively identified. The graph-based method introduced in this study successfully utilized combined metrics from version control systems (VCS) and Jira activity to pinpoint individuals crucial to project continuity. The results summarized in Tables 5.2 and

5.4 demonstrate the method's ability to accurately reflect real-world scenarios. For instance, the evaluation on university projects revealed substantial alignment with developer-identified key contributors, accurately capturing 83% of these contributors. Furthermore, across the anonymized industry projects evaluated, the method correctly identified critical contributors with an accuracy of 85%.

5.4.2 Methods for Effective Critical Knowledge Transfer

The second research question targeted methods that could facilitate critical knowledge transfer once key personnel are identified. The implementation of comprehensive visualization techniques (e.g., treemaps and interactive network graphs) significantly supported stakeholders in pinpointing high-risk knowledge areas. Qualitative feedback, presented in the usability evaluation (Table 5.7), indicates a strong approval rate (90%) among users for general system usability, highlighting the value of clear visualization in effectively managing knowledge transfer activities. Specifically, users emphasized the clarity and ease of interpreting centrality breakdowns (83%), affirming the tool's role in effectively highlighting contributors who require knowledge-sharing interventions.

5.4.3 Alignment with Existing State-of-the-Art Algorithms

The final research question examined the extent of alignment between the bus factor estimations generated by our proposed method and existing state-of-the-art approaches, notably the Explorer method. Comparative analyses, detailed in Tables 5.1 and Table 5.3, consistently demonstrated superior accuracy and F1-scores achieved by our graph-based approach. Overall, the method averaged 70.14% accuracy and a 78.06% F1-score, significantly outperforming the Explorer method, which averaged 59.72% accuracy and 69.44% F1-score. Moreover, simulation-based robustness testing confirmed this finding, where the graph-based method consistently achieved higher accuracy across diverse scenarios, including equal contributions, skewed distributions, and varied activity levels (Table 5.4).

These findings strongly validate the robustness, reliability, and enhanced capability of the proposed graph-based bus factor identification method, affirming its potential for practical application across various software development contexts.

Chapter 6

Discussion

The discussion chapter wraps up and summarize the key contributions of this study including its significance while providing an overall overview of the challenges faced through out the timeframe, which justifies certain decisions made during the implementation and validation stages.

6.1 Key Contributions

This research contributes a novel graph-based method for identifying key contributors in software development teams by incorporating both version control data and issue tracking activity. Unlike traditional approaches that solely rely on metrics such as commit frequency, file authorship, etc., this approach offers more comprehensive, network-driven view of team dynamics and collaboration patterns. Furthermore, the integration of issue management data from project management tools like Jira serves as a Proof-of-Concept (PoC) for including broader collaborative activity beyond version control in order to improve the accuracy of the results. Anyone who is willing to continue in their journey of fine-tuning this approach can use this as a starting point for making further extensions incorporating any other potential data sources that contains project context related data.

Another significance of this approach is the ability to detect critical developers both collaborative or low-commit scenarios, increasing the robustness of bus factor identification. The language-agnostic nature of the solution and the ability to handle diverse repository structures enhance its applicability across a wide variety of real-world settings. Also by considering only the timely contributions and interactions, this method has the ability to cater the dynamic nature of a given project at a given time.

Apart from the key developer identification, another pivotal contribution of this research lies in the implementation of an automated documentation generation system designed to accelerate developer onboarding in complex software projects. By integrating RepoMix and the DeepSeek API, the solution transforms raw codebases into structured, AI-generated guides that outline project architecture, workflows, and critical dependencies. It addresses the common challenge of fragmented or outdated documentation, offering new developers an intuitive roadmap to navigate code logic, trace method interactions, and grasp system-wide behaviors without manual effort, resulting in a significant reduction in onboarding time and cognitive load.

6.2 Challenges

The development and evaluation phases of the proposed method involved overcoming several significant challenges.

6.2.1 Data Availability Challenges

- Obtaining complete and clean datasets from real-world repositories proved challenging due to confidentiality constraints and the limited availability of well-documented projects. Many industry-grade codebases remain inaccessible, forcing reliance on open-source alternatives, which often fail to mirror the complexity of closed-source systems. Additionally, even available datasets tend to become outdated as projects evolve, with knowledge dispersing across teams over time. While open-source projects offer partial solutions, their structure and development practices frequently differ from proprietary environments, limiting their applicability in certain research contexts.

- Accessing active Jira boards with meaningful historical data was another major hurdle, particularly for closed-source or professionally managed projects. Organizations often employ proprietary tools and methods for tracking project context—such as meeting notes, issue resolution logs, and task dependencies—making standardized data collection difficult. Sensitivity around internal processes further restricted data sharing, as companies were reluctant to expose project-specific details. This lack of transparency in issue-tracking systems hindered a comprehensive analysis of development workflows and decision-making patterns in real-world settings.

6.2.2 Evaluation and Feedback Challenges

- Sourcing suitable university projects for initial testing proved problematic, as many lacked consistent development activity or detailed version control histories. This inconsistency made it difficult to replicate real-world scenarios during early validation, limiting the reliability of preliminary results. While academic projects provided a starting point, their smaller scale and sporadic contribution patterns often failed to mirror the iterative, collaborative nature of industrial software development

- Organizing structured feedback sessions with developers required meticulous coordination due to conflicting schedules, geographically dispersed teams, and varying engagement levels. Differences in organizational workflows—such as agile versus waterfall methodologies—further complicated efforts to standardize feedback collection. Additionally, subjective input from participants demanded careful analysis to mitigate bias, requiring iterative refinement to align diverse perspectives into actionable insights.

6.2.3 Technical Challenges

- Resolving contributor identity discrepancies across platforms (e.g., GitHub, Jira) demanded algorithmic precision. Contributors often used distinct aliases or multiple accounts, and the absence of centralized identity mappings risked skewed attribution of work. This necessitated the development of cross-platform matching logic to unify identities while accounting for edge cases like pseudonyms or shared accounts.

- Designing adaptable centrality weighting mechanisms required extensive empirical tuning to reflect realistic team dynamics. Static models struggled to accommodate variations in organizational structures, such as hierarchical versus flat teams, or differing roles like core maintainers versus part-time contributors. Iterative adjustments were essential to balance specificity (capturing unique team cultures) and generalizability (ensuring the tool's applicability across diverse projects).

Chapter 7

Conclusion

7.1 Overview

The conclusion chapter summarize the overall study and reflect on the research study, highlighting its limitations and suggesting directions for future research.

7.2 Limitations

Limited access to full-scale Jira boards and extensive professional repositories confined validation to a handful of projects, which may restrict how broadly these findings apply across different team structures and development cultures. In addition, the current approach used in unifying contributor identities across platforms like GitHub and Jira is not always effective, risking both under- and over-attribution, especially when pseudonyms or shared accounts were involved. Although centrality and activity weights were honed through empirical tuning, they may not translate perfectly to organizations with hierarchical workflows, decentralized decision-making, or non-traditional roles. Generating documentation for larger codebases frequently exceeded LLM token limits when trying to combine the whole codebase as the context, forcing context truncation that undermines usability for complex codebases. Also, the tool's real-world performance lagged behind its core algorithm's speed in isolated environments, revealing implementation bottlenecks that must be addressed before real-time or large-scale deployment.

7.2.1 Future Work

The empirical validation of the approach requires expansion through broader testing with diverse, clean datasets. This testing will help refine weight calibration and threshold values, ultimately ensuring the robustness of the system across varied team structures and development methodologies. Performance optimization also stands as a critical priority, with efforts needed to streamline the tool's runtime efficiency to match the core algorithm's speed demonstrated in isolated environments. These optimizations would address existing bottlenecks that currently limit the tool's applicability in real-time scenarios or large-scale implementations.

Enterprise scalability presents another significant avenue for development, requiring adaptations to the platform that would support continuous, real-time monitoring in enterprise settings. Such adaptations must ensure seamless integration with evolving codebases and workflows characteristic of large organizations. The integration of multi-source collaboration signals represents an opportunity to substantially enhance the approach, incorporating additional data streams such as code reviews, Slack/MS Teams activity, and temporal engagement patterns to enrich the model's accuracy and contextual awareness.

The development of intuitive, interactive dashboards would translate metrics into actionable insights for both managerial and technical stakeholders, enhancing the decision-support visualizations currently available. Meanwhile, advancing the predictive capabilities of the system would involve implementing analytics to proactively identify risks tied to team turnover, role shifts, or knowledge gaps during critical project phases. Also the tool should be extended and fine-tuned to accurately capture scenarios where a bus factor risk does not exist, such as when all contributors are equally engaged and critical knowledge is evenly distributed across the team.

Documentation generation capabilities could be refined through the adoption of dynamic, context-aware chunking strategies for LLM inputs. Such strategies would bypass token limits, enabling scalable documentation generation even for large projects. The automation of contributor identity resolution represents another promising direction, with potential experiments using LLM-driven approaches to unify cross-platform identities, thus reducing reliance on error-prone static methods like fuzzy matching. Finally, investigating bus factor dynamics across different software lifecycle phases—from design through maintenance—would provide valuable insights for refining risk mitigation strategies tailored to each development stage.

7.3 Final Remarks

This research addresses the critical challenge of knowledge imbalance in software teams by introducing a graph-based method to identify key contributors and assess bus factor risks. The outcomes align directly with the stated objectives, offering actionable insights for both academic and industrial contexts.

The first objective—identifying potential single points of failure early in the project lifecycle—was achieved via the integration of network centrality metrics and activity analysis. Empirical validation across 12 industry projects demonstrated the method's ability to pinpoint critical contributors with 85.56% accuracy, significantly outperforming existing benchmarks like the Explorer method (59.72% accuracy). This precision enables teams to proactively address knowledge concentration before it escalates into operational risks.

As for the second objective, enhancing critical knowledge transfer, the system's usability played a central role. With an average SUS score of 86.7 and 90% satisfaction in overall usability, stakeholders found the tool intuitive for visualizing contributor roles and dependency networks. Qualitative feedback highlighted its effectiveness in clarifying team dynamics, which supports targeted interventions such as mentorship or task redistribution.

The third objective—evaluating practical relevance—was validated through rigorous comparisons with state-of-the-art algorithms. As shown in Tables III and IV, the graph-based approach consistently identified contributors with higher specificity, particularly in complex collaboration environments (e.g., 78.06% F1-score vs. 69.44% for Explorer) which confirms its reliability in real-world settings, bridging the gap between theoretical models and organizational needs.

While the method's language-agnostic design and scalability (90% efficiency retention) broaden its applicability, limitations such as contributor identity discrepancies and computational bottlenecks warrant further refinement. Nevertheless, the framework provides a foundation for advancing software sustainability, enabling teams to mitigate continuity risks through data-driven strategies. Future extensions could integrate real-time analytics or machine learning to further enhance adaptability across evolving development pipelines.

Moreover, the research addresses a timely and critical issue, making a significant contribution to the software engineering domain and the introduction of a novel approach for calculating the bus factor further enhances the existing body of knowledge, providing development organizations with a practical framework to assess and mitigate knowledge concentration risks. The methodology developed throughout this research offers tangible benefits for real-world software projects seeking to improve their operational resilience and ensure sustainable development practices.

Bibliography

Almarimi, N., Ouni, A., Chouchen, M., and Mkaouer, M. W. (2021). csdetector: an open source tool for community smells detection. pages 1560–1564.

Avelino, G., Constantinou, E., Valente, M. T., and Serebrenik, A. (2019). On the abandonment and survival of open source projects: An empirical investigation. *CoRR*, abs/1906.08058.

Avelino, G., Passos, L., Hora, A., and Valente, M. T. (2016). A novel approach for estimating truck factors. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE.

Bock, T., Alznauer, N., Joblin, M., and Apel, S. (2023). Automatic core-developer identification on github: A validation study. *ACM Trans. Softw. Eng. Methodol.*, 32(6).

Bosu, A. and Carver, J. C. (2014). Impact of developer reputation on code review outcomes in oss projects: an empirical investigation. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, New York, NY, USA. Association for Computing Machinery.

Brocke, J. v., Hevner, A., and Maedche, A. (2020). *Introduction to Design Science Research*, pages 1–13.

Constantino, K., Zhou, S., Souza, M., Figueiredo, E., and Kästner, C. (2020). Understanding collaborative software development: An interview study. In *Proceedings of the 15th international conference on global software engineering*, pages 55–65.

Coplien, J. O. and Harrison, N. B. (2004). Organizational patterns of agile software development.

Cosentino, V., Canovas Izquierdo, J., and Cabot, J. (2015). Assessing the bus factor of git repositories.

Dikert, K.-K., Paasivaara, M., and Lassenius, C. (2016). Challenges and success factors for large-scale agile transformations: A systematic literature review. *J. Syst. Softw.*, 119:87–108.

Ebert, C. and Cain, J. (2016). Cyclomatic complexity. *IEEE Software*, 33:27–29.

Fritz, T., Murphy, G. C., Murphy-Hill, E., Ou, J., and Hill, E. (2014). Degree-of-knowledge: Modeling a developer's knowledge of code. *ACM Trans. Softw. Eng. Methodol.*, 23(2).

Jabrayilzade, E., Evtikhiev, M., Tüzün, E., and Kovalenko, V. (2022). Bus factor in practice.

Joblin, M., Apel, S., Hunsen, C., and Mauerer, W. (2016). Classifying developers into core and peripheral: An empirical study on count and network metrics.

Kirongo A. Chege, Amos, O. C. O. (2020). Research philosophy design and methodologies: A systematic review of research paradigms in information technology. 8:33.

Klimov, E., Ahmed, M. U., Sviridov, N., Derakhshanfar, P., Tüzün, E., and Kovalenko, V. (2023). Bus factor explorer. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.

Lisan, A. and Norris, B. (2024). Guiding effort allocation in open-source software projects using bus factor analysis.

Oliva, G. A., da Silva, J. T., Gerosa, M. A., Santana, F. W., Werner, C. M. L., de Souza, C., and de Oliveira, K. C. M. (2015). Evolving the system's core: A case study on the identification and characterization of key developers in apache ant. *Comput. Informatics*, 34:678–724.

Rigby, P., Zhu, Y., Donadelli, S., and Mockus, A. (2016). Quantifying and mitigating turnover-induced knowledge loss: case studies of chrome and a project at avaya. pages 1006–1016.

Zazworka, N., Stapel, K., Knauss, E., Shull, F., Basili, V., and Schneider, K. (2010). Are developers complying with the process: An xp study.

Zhang, W., Yang, Y., and Wang, Q. (2011). Network analysis of oss evolution: an empirical study on argouml project. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, page 71–80, New York, NY, USA. Association for Computing Machinery.