UNIVERSIDADE FEDERAL DO MARANHÃO

UNIVERSIDADE FEDERAL DO PIAUÍ

Doutorado em Ciência da Computação Associação UFMA/UFPI

Otávio Cury da Costa Castro

Source Code Expertise: Improving Knowledge Models and Assessing Generative AI Impact

Orientador: Guilherme Amaral Avelino

Co-orientador: Pedro de Alcantara dos Santos Neto

Teresina - PI
Agosto, 2025

Otávio Cury da Costa Castro

Source Code Expertise: Improving Knowledge Models and Assessing Generative AI Impact

TESE DE DOUTORADO

Tese apresentada como requisito parcial para obtenção do título de Doutor em Ciência da Computação, ao Doutorado em Ciência da Computação, Associação UFMA/UFPI.

Orientador: Guilherme Amaral Avelino
Co-orientador: Pedro de Alcantara dos Santos Neto

Teresina - PI
Agosto, 2025

FICHA CATALOGRÁFICA
Universidade Federal do Piauí
Biblioteca Comunitária Jornalista Carlos Castello Branco
Divisão de Representação da Informação

C355s Castro, Otavio Cury da Costa.
Source code expertise : improving knowledge models and assessing generative ai impact / Otavio Cury da Costa Castro. – 2025.
110 f.

Tese (Doutorado) – Universidade Federal do Maranhão, Universidade Federal do Piauí, Centro de Ciências da Natureza, Programa de Pós-Graduação em Ciências da Computação, Teresina, 2025.
"Orientador: Guilherme Amaral Avelino."
"Co-orientador: Pedro de Alcantara dos Santos Neto."

1. Software repository mining. 2. Code expertise. 3. Knowledge concentration. 4. Generative artificial intelligence. I. Avelino, Guilherme Amaral. II. Santos Neto, Pedro de Alcantara dos. III. Título.

CDD 005.1

Bibliotecário: Gésio dos Santos Barros – CRB3/1469

Otávio Cury da Costa Castro

Source Code Expertise: Improving Knowledge Models and Assessing Generative AI Impact

A presente Tese de Doutorado foi avaliada e aprovada por banca examinadora composta pelos seguintes membros:

Guilherme Amaral Avelino
Orientador
Universidade Federal do Piauí

Prof. Dr. Pedro de Alcantara dos Santos Neto
Co-orientador
Universidade Federal do Piauí

Prof. Dr. Vinicius Ponte Machado
Examinador Interno
Universidade Federal do Piauí

Prof. Dr. Romuere Rodrigues Veloso e Silva
Examinador Interno
Universidade Federal do Piauí

Prof. Dr. Lincoln Souza Rocha
Examinador Externo
Universidade Federal do Ceará

Prof. Dr. André Cavalcante Hora
Examinador Externo
Universidade Federal de Minas Gerais

Certificamos que esta é a versão original da Tese de Doutorado que foi julgada adequada para obtenção do título de Doutor em Ciência da Computação.

gov.br Documento assinado digitalmente
GUILHERME AMARAL AVELINO
Data: 26/08/2025 14:36:35-0300
Verifique em https://validar.iti.gov.br

Guilherme Amaral Avelino
Orientador

gov.br Documento assinado digitalmente
ANDRE CASTELO BRANCO SOARES
Data: 26/08/2025 14:56:21-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Andre Castelo Branco Soares
Coordenador

Teresina - PI, 22 de Agosto de 2025

Ao meu irmão Arthur Cury (in memoriam), minha fonte de motivação.

Acknowledgements

Primeiramente, gostaria de agradecer a Deus por todas as bênçãos e oportunidades que marcaram o meu caminho até aqui. Agradeço a toda a minha família pelo apoio incondicional. Aos meus pais, especialmente à minha querida mãe, Gemilia, pelo amor e pela orientação que nunca me faltaram. Meus agradecimentos também ao meu irmão Álvaro, aos meus tios, tias e primos, por todo o carinho que sempre me acompanhou.

Agradeço imensamente à minha companheira, Alyne, por todo o amor e pela motivação nos momentos em que mais precisei. Agradeço ainda ao meu irmão Arthur (in memoriam), pelos anos que passamos juntos e por, mesmo não estando fisicamente presente, continuar sendo uma grande fonte de motivação e inspiração para superar as dificuldades encontradas.

Agradeço ao meu coorientador, Prof. Pedro de Alcântara, pelo esforço e supervisão neste trabalho. Um agradecimento especial ao meu orientador, Prof. Guilherme Avelino: obrigado por toda a preocupação, dedicação e confiança durante todos esses anos de colaboração. Sem sua ajuda e orientação, este trabalho e seus méritos certamente não seriam possíveis.

Por fim, agradeço a todos os colegas e professores da Universidade Federal do Piauí. Obrigado por todos os momentos preciosos que vivi ao longo da minha trajetória acadêmica.

Muito obrigado!

Abstract

Identifying developer expertise in source code is valuable in various Software Engineering contexts. Knowledgeable developers are best suited to perform tasks such as code review and onboarding. Numerous models have been proposed to estimate source code knowledge, making it a well-explored topic; however, important gaps remain that affect the accuracy and applicability of these models. Moreover, the increasing use of Generative Artificial Intelligence (GenAI) tools may influence how code expertise is acquired and measured. This study aims to develop more accurate models for identifying source code experts. We first investigate the correlation between development history variables and developers' knowledge of source code files. We extract metrics from public and private repositories and survey developers about the files they contributed to. Based on these data, we propose a linear model and train machine learning classifiers, comparing their performance with existing models. We also apply the proposed models to the Truck Factor (TF) metric to assess their practical implications in identifying critical developers. To examine the impact of GenAI, we build a dataset combining code expertise metrics with information on ChatGPT-generated code integrated into open-source projects. We simulate different usage scenarios by assigning a portion of contributions to GenAI instead of developers and survey developers about their perception of GenAI's effects on code comprehension. Our results show that First Authorship and Recency of Modification are the variables most strongly correlated with source code knowledge. The proposed machine learning models outperform linear baselines, achieving F-scores between 71% and 73%. When applied to the TF algorithm, they improved developer identification, reaching a best average F-score of 74%. GenAI usage negatively affected TF reliability, even in low proportions. Developers reported mixed perceptions, with concerns, especially about use by novice programmers.

Keywords: software repository mining, code expertise, knowledge concentration, generative artificial intelligence.

List of Figures

List of Tables

List of abbreviations and acronyms

VCS	*Version Control System*
AI	*Artificial Intelligence*
ML	*Machine Learning*
RF	*Random Forest*
GBM	*Gradient Boosting Machines*
SVM	*Support Vector Machines*
KNN	*K-Nearest Neighbors*
TP	*True Positive*
TN	*True Negative*
FP	*False Positive*
FN	*False Negative*
FM	*F-Measure*
TF	*Truck Factor*
GenAI	*Generative Artificial Intelligence*
DOK	*Degree of Knowledge*
DOA	*Degree of Authorship*
DOI	*Degree of Interest*
DOE	*Degree of Expertise*
OSS	*Open-source software*
LoC	*Lines of Code*
FA	*First Authorship*
RQ	*Research Question*
API	*Application Programming Interface*
URL	*Uniform Resource Locator*

Contents

1 Introduction

1.1 Motivation and Problem

Software maintenance and evolution is an iterative process primarily driven by source code changes across various development activities (RAJLICH, 2014). This process demands seamless collaboration and efficient team management due to the continuous emergence of bugs and issues (SUN et al., 2017). Managing these activities becomes challenging in large projects, where managers require comprehensive information about their development to coordinate their teams effectively. In this context, it is crucial to identify which developers possess expertise in specific parts of the source code. This need has become even more critical with the rise of remote work, which has grown and solidified in many software organizations following the COVID-19 pandemic (ROT; SOBINSKA; BUSCH, 2023; RALPH et al., 2020).

Identifying developer expertise is valuable in many aspects of the software development process. For instance, it aids in task assignments, determining who can assist newcomers (CANFORA et al., 2012), and who is best suited to fix a bug (KAGDI; HAMMAD; MALETIC, 2008). Additionally, this information enables project managers to monitor knowledge concentration within the source code, where a few team members may hold critical knowledge. This concentration poses a significant risk to the project if those key individuals leave, potentially leading to project discontinuation (FERREIRA; VALENTE; FERREIRA, 2017; ROBILLARD, 2021).

However, tracking project file familiarity poses a significant challenge for developers and managers due to the sheer volume of information related to code changes (FRITZ et al., 2014). To assist with this task, we can leverage data available in Version Control Systems (VCS), which log a substantial portion of developer-file interactions (ZOLKIFLI; NGAH; DERAMAN, 2018). Previous research has utilized this information to tackle the problem of identifying file experts (FRITZ et al., 2014; MCDONALD; ACKERMAN, 2000; MOCKUS; HERBSLEB, 2002; MINTO; MURPHY, 2007; SILVA et al., 2015).

In a related study, Avelino et al. (2018) compared the performance of three models for identifying file experts. The authors identified opportunities to improve the existing models by incorporating information on *file size* and the *recency of modifications*. In this thesis, we address this research gap by developing more accurate models for identifying source code expertise and exploring their practical applications in software development contexts.

Additionally, in recent years, Generative Artificial Intelligence (GenAI) has garnered

significant attention and interest from Software Engineering research and practice (NGUYEN-DUC et al., 2023). This interest is justifiable due to the immense potential of GenAI to transform not only the software profession but also a wide range of other domains, including marketing, gaming, and language processing (GOZALO-BRIZUELA; GARRIDO-MERCHÁN, 2023). The ability of GenAI to generate human-like text, code, and other content opens up new possibilities and challenges across various fields, driving innovation and altering traditional workflows.

Specifically in software development, the increasing integration of GenAI into the development environment has brought numerous benefits. GenAI tools like OpenAI's ChatGPT[1], and GitHub Copilot[2] have increased productivity, generated code snippets, facilitated testing (DOHMKE; IANSITI; RICHARDS, 2023; SOHAIL et al., 2023; ZIEGLER et al., 2024). Despite these benefits, practitioners and researchers have expressed concerns and highlighted several challenges associated with adopting these technologies (ERNST; BAVOTA, 2022).

One drawback is that the use of these GenAI tools to generate source code may lead to developer over-reliance, resulting in a lack of understanding of the recommendations and an inclination to accept code generated by these tools without sufficient understanding (ZHANG et al., 2023; YILMAZ; YILMAZ, 2023; PRATHER et al., 2023; RUSSO, 2024). Based on these insights, this thesis also investigates how developers' use of these tools might affect expertise identification models. This will allow us to understand how integrating AI-generated code influences the efficacy and reliability of these source code knowledge models.

1.2 Proposed Thesis

In this thesis, we aim to answer three overarching questions related to creating more accurate expertise identification models, their applications in real software development contexts, and the reliability and efficiency of these source code knowledge models in projects that use Generative AI for source code generation. We start by comprehensively analyzing historical development variables to create new, more accurate knowledge models for identifying source code experts (**Q1**). Our goal includes performing a comparative study of these new models against existing ones documented in the literature. With these improvements, we apply these models in practice in a software development context, specifically focusing on a well-studied knowledge concentration metric (**Q2**). Finally, we collect extensive data on the use of GenAI tools, such as ChatGPT, in open-source projects, enabling us to examine how integrating AI-generated code affects the effectiveness of these knowledge models (**Q3**). These overarching questions are

[1] https://chatgpt.com
[2] https://github.com/features/copilot

described in detail below:

Q1: Can we improve existing source code knowledge models?

Several works in the literature use different repository-based metrics to infer developers' knowledge of source code files. However, we did not identify large-scale studies correlating these variables with source code knowledge. By addressing this gap, we aim to understand how these variables are related to knowledge, which can guide the development of models for identifying source code file experts. Additionally, given the successful application of machine learning classifiers in the software engineering literature, we evaluated whether applying machine learning classifiers can improve the performance of identifying experts compared to other techniques used in previous works. Chapter 4 describes this study and includes the following related publication:

- *Cury, Otávio, et al. "Identifying Source Code File Experts." Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. 2022.*

Q2: Can we improve a knowledge concentration metric with new source code knowledge models?

Avelino et al. (2016) proposed a greedy algorithm to compute the Truck Factor of software repositories, which has since been adopted in several studies (AVELINO et al., 2019; JABRAYILZADE et al., 2022; CALEFATO et al., 2022; ALMARIMI et al., 2021; CANEDO et al., 2020; FERREIRA; VALENTE; FERREIRA, 2017). However, the original work pointed out a key limitation: the algorithm fails to include developers with recent contributions in the list of *Truck Factor Developers*. To address this issue, we enhanced Avelino's algorithm by integrating knowledge models that account for the recency of contributions and other relevant variables. This modification aims to provide a more accurate and comprehensive estimation of the Truck Factor. We evaluated the improved algorithm using a dataset of open-source developers previously missed by the original method, and through a survey with developers from a private project, comparing the accuracy of both algorithms. Chapter 5 presents this study and includes the following related publications:

- *Cury, Otávio, et al. "Source code expert identification: Models and application." Information and Software Technology 170 (2024).*
- *Cury, Otávio; Avelino, Guilherme. Knowledge Islands: Visualizing Developers Knowledge Concentration. In: Simpósio Brasileiro de Engenharia de Software (SBES), 38. , 2024, Curitiba/PR.*

Q3: What is the impact of Generative AI-assisted code generation on the accuracy and reliability of source code knowledge models?

The growing popularity of GenAI tools for source code generation has raised concerns about developers' understanding of the generated and integrated code (DENNY et al., 2024; YILMAZ; YILMAZ, 2023; MA; CHEN; KONOMI, 2024; RUSSO, 2024). These changes are likely to affect knowledge models designed to identify source code expertise. To the best of our knowledge, no prior studies have examined the relationship between GenAI-assisted code generation and source code knowledge models. To address this gap, we conducted an exploratory study to assess the impact of GenAI usage on these models and their applications. We built a dataset that combines developer expertise metrics with information about the integration of GenAI-generated code into open-source projects. This allowed us to estimate the potential loss of code authorship caused by GenAI and its effects on a proposed knowledge model and its implementation in a Truck Factor algorithm. Additionally, we surveyed to gather developers' perceptions of how GenAI affects code comprehension and to understand how they deal with this issue in practice. Chapter 6 presents this study and includes the following related publication:

- *Cury, Otávio; Avelino, Guilherme. The Impact of Generative AI on Code Expertise Models: An Exploratory Study. In: Simpósio Brasileiro de Engenharia de Software (SBES), 2025, Recife/PE.*

1.3 Thesis Structure

This thesis comprises six additional chapters, structured as follows. The next chapter provides the necessary background concepts to support the understanding of this work. Chapter 3 reviews the main studies related to the topics addressed in this thesis. Chapter 4 presents a study that analyzes the correlation between development history variables and proposes more accurate source code knowledge models. Chapter 5 describes a practical application of the proposed models to improve a well-established measure of knowledge concentration. Chapter 6 examines the effects of using Generative AI on knowledge models. Finally, Chapter 7 concludes this work with a comprehensive summary and suggestions for future research directions.

2 Background

This section briefly presents the main concepts explored in this thesis. Section 2.1 explains what Version Control Systems are and how they work. Section 2.2 describes different types of developer expertise discussed in the literature. Section 2.3 introduces machine learning, its main types, and the classifiers used in this work. Section 2.4 presents the performance metrics used to evaluate binary classifiers, as well as the concept of cross-validation and how it works. Section 2.5 examines the Truck Factor concept and its relevance to software development. Finally, Section 2.6 defines Generative Artificial Intelligence and discusses its applications in software engineering.

2.1 Version Control Systems

A *Version Control System* (VCS) is a type of software that tracks and manages different versions of files throughout a project's lifecycle (LOELIGER; MCCULLOUGH, 2012). Its benefits in software development are numerous, making it an essential tool in modern practices. As multiple developers make frequent changes to project files, it becomes crucial to maintain a detailed history of what was changed, who made the changes, and when they occurred (OTTE, 2009; SPINELLIS, 2005).

VCSs streamline collaboration by enabling developers to share project files efficiently, accelerating and simplifying the development process. Developers can work concurrently on the same codebase and later commit their changes. When conflicting modifications occur, VCSs provide mechanisms, such as merge operations, to detect and resolve them.

Within a VCS, *commits* play a central role by recording changes made to project files. Each commit creates a new version of the affected files (SPINELLIS, 2005). By analyzing commit histories, it is possible to trace the evolution of the code and identify the specific changes made to each line. To retrieve a particular version of the codebase, developers use the *checkout* command, which updates local files to reflect the desired version or the latest changes from other team members.

Another fundamental concept in VCS is that of branches. Branches represent independent lines of development within the same project (PILATO; COLLINS-SUSSMAN; FITZPATRICK, 2008). They originate from a main line of development, commonly called *main* or *master*, and share a common commit history up to the point where they diverge. Branches are especially useful for implementing new features or testing changes without compromising the stability of the main codebase.

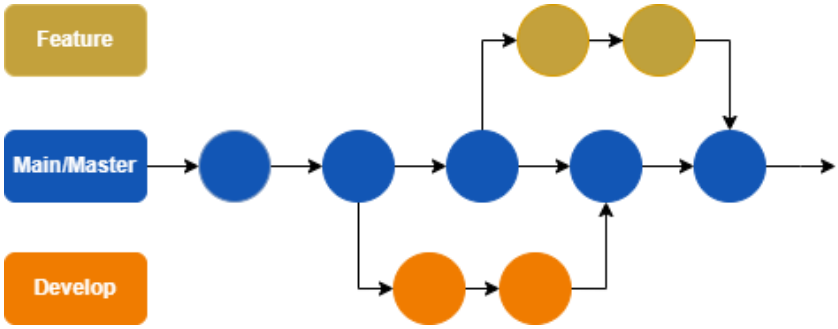


Figure 1 – Illustration of branch merge history in a version control system.

Version Control Systems play a central role in this thesis, as all the techniques investigated are based on information extracted from *Git*[1] repositories, currently the most widely used VCS (OVERFLOW, 2022). The process of extracting and processing information from the VCS of the selected repositories is detailed in the following chapters.

2.2 Source Code Expertise

The term *expertise* has several synonyms in the related literature, such as experience, knowledge, authorship, familiarity, and ownership (KRÜGER et al., 2018). Due to its widespread use, the term *expertise* is adopted consistently throughout this thesis. In the context of software development, different types of *expertise* are discussed in the literature, each reflecting a particular kind of knowledge that a person accumulates about an artifact or process.

Some studies focus on identifying *Review Expertise* (HANNEBAUER et al., 2016; SÜLÜN; TÜZÜN; DOĞRUSÖZ, 2019; SÜLÜN; TÜZÜN; DOĞRUSÖZ, 2020), which aims to determine which developers have the most experience reviewing specific parts of the source code. This line of research supports the automation of review assignment, a key area in modern code review practices (KIM; LEE, 2018; BADAMPUDI; BRITTO; UNTERKALMSTEINER, 2019).

Other research efforts propose techniques to infer *Usage Expertise* (SCHULER; ZIMMERMANN, 2008; MANI; PADHYE; SINHA, 2016), which relates to a developer's understanding of the purpose of a software artifact and how its services can be applied to solve problems. This type of expertise is often connected to developer skill profiling (OLIVEIRA; VIGGIATO; FIGUEIREDO, 2019; MONTANDON; SILVA; VALENTE, 2019).

This thesis, however, focuses on identifying *Implementation Expertise* (ANVIK; MURPHY, 2007), also referred to as *Modification Expertise* (HANNEBAUER et al., 2016). This form of expertise concerns how familiar a developer is with a given source code artifact, including its structure and behavior. Such knowledge is typically acquired through

[1] https://git-scm.com

direct modification of the source code (ANVIK; MURPHY, 2007). Developers' expertise can be analyzed at different levels of granularity, such as lines, methods, files, or packages. In this work, we specifically consider file-level expertise, that is, the degree to which a developer is knowledgeable about the contents of a given source code file.

2.3 Machine Learning

Machine Learning is a subfield of Artificial Intelligence (AI) (RUSSELL; NORVIG, 2009) that focuses on developing methods capable of improving performance or making more accurate predictions based on *experience*, meaning past information available to the learning method (LUGER, 2004; KULKARNI, 2012).

There are three main types of learning: supervised, unsupervised, and semi-supervised. In supervised learning, each instance in the training dataset is associated with a corresponding output label, enabling the model to learn a mapping from inputs to outputs. In contrast, unsupervised learning does not provide output labels, requiring the algorithm to discover patterns or structures within the data on its own (KULKARNI, 2012). Semi-supervised learning combines both labeled and unlabeled data during training, often aiming to leverage the abundance of unlabeled data to improve performance.

When the output belongs to a finite set of categories, the problem is referred to as *classification*. A common subtype is *binary classification*, which involves only two possible output classes (RUSSELL; NORVIG, 2009), for example, classifying developers as either file maintainers or non-maintainers, a task addressed in the following chapters.

Machine learning has been applied across a wide range of domains (MOHRI; ROSTAMIZADEH; TALWALKAR, 2018). In the context of software engineering, it is used for tasks such as software quality assessment (AL-JAMIMI; AHMED, 2013), development cost estimation (WEN et al., 2012), software testing (DURELLI et al., 2019), and defect prediction (SHEPPERD; BOWES; HALL, 2014), among others. The integration of machine learning techniques with software repository mining has become a growing research area within the field (GÜEMES-PEÑA et al., 2018).

2.3.1 Machine Learning Algorithms

In the following chapters of this thesis, we employ five widely used supervised learning algorithms to support the identification of source code expertise. The selection of these algorithms is grounded in their popularity within software engineering research and their proven effectiveness in addressing binary classification problems (MONTANDON; SILVA; VALENTE, 2019; SATAPATHY; RATH, 2017).

2.3.1.1 Random Forest

Random Forest is a machine learning algorithm based on decision trees. This algorithm can be used for regression and classification tasks and is highly regarded for its performance and robustness (ALI et al., 2012; SATAPATHY; RATH, 2017; HASANLUO; GHAREHCHOPOGH, 2016).

In essence, *Random Forest* constructs multiple individual decision trees and combines their outputs to achieve greater predictive power (BREIMAN, 2001). The data samples and the variables used to build each tree are selected randomly. When new data is introduced, the algorithm aggregates the responses from these individual trees to produce a single result, with the method of combination depending on the specific nature of the problem.

This algorithm has numerous applications in software engineering, including failure prediction (KAUR; MALHOTRA, 2008), software quality assessment (FOLLECO et al., 2008), effort estimation (SATAPATHY; ACHARYA; RATH, 2016), and identifying the *expertise* of developers (MONTANDON; SILVA; VALENTE, 2019).

2.3.1.2 Gradient Boosting Machines

Gradient Boosting Machine (GBM) is a powerful machine learning algorithm that has demonstrated strong performance in several application areas (NATEKIN; KNOLL, 2013). Some applications of this algorithm can be found in the field of software engineering (SATAPATHY; ACHARYA; RATH, 2014; NASSIF et al., 2012; SATAPATHY; RATH, 2017).

Like *Random Forest*, GBM combines the predictive power of several smaller models into one robust model. However, unlike *Random Forest*, which combines several models built randomly (*bagging*), GBM builds models sequentially so that each model improves the performance of the previous ones (*boosting*). Typically, the models used are decision trees.

2.3.1.3 Support Vector Machines

Support Vector Machines (SVM) is a supervised learning algorithm used for classification or regression problems (GUNN et al., 1998). It is widely used, particularly due to its performance in problems involving many attributes (*features*) and its robustness to *outliers*.

Generally, this algorithm is based on finding a hyperplane (function) that best separates data from two classes. This is achieved by calculating the distance (margin) of the closest points of both classes to the hyperplane. The optimal hyperplane is the one that maximizes this distance, effectively finding a boundary that maximizes the distinction between the two classes. To handle data that is not linearly separable, the algorithm uses

a *Kernel trick* to mathematically map the data into higher-dimensional spaces, enabling easier separation (GUENTHER; SCHONLAU, 2016).

This algorithm is widely used in software engineering research, with applications in areas such as failure prediction (ELISH; ELISH, 2008), software quality (XING; GUO; LYU, 2005), and developer expertise identification (MONTANDON; SILVA; VALENTE, 2019).

2.3.1.4 K-Nearest Neighbors

K-Nearest Neighbors (KNN) is a simple supervised machine learning algorithm that can be used for classification and regression problems (PETERSON, 2009). However, it is commonly used in classification problems and has several applications in software engineering (HASANLUO; GHAREHCHOPOGH, 2016; ALKHALID; LUNG; AJILA, 2013).

The algorithm is based on the similarity between data points. Given a new input that needs to be classified, the algorithm calculates the distance from that input to all previously classified (labeled) data. This distance is typically calculated using the *Euclidean Distance* of the *features* values of each data point (CHOMBOON et al., 2015). After calculating the distances, the new data point is classified based on the majority class among the K closest data points.

2.3.1.5 Logistic Regression

Logistic Regression is a statistical technique that can be applied in machine learning, typically used for binary classification problems (BISHOP, 2006).

In short, this technique uses a logistic function fitted to the training data to return the probability of a given data point belonging to the main class of the problem. With the probability associated with new data, a *threshold* is used to determine if the data belongs to the main class.

2.4 Performance of Binary Classifiers

Several measures can be used to analyze the performance of binary classifiers, such as those used in this thesis. In this section, we explain some of these measures, how they are obtained, and the contexts in which they are applied.

2.4.1 Confusion Matrix

The confusion matrix is a method used to organize the classification results produced by an algorithm, aiding in the analysis of its performance (POWERS, 2020). In binary classifications, this table consists of two rows and two columns that display the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Table 1 provides an example of a confusion matrix for binary classifications.

		Actual Values	
		x	y
Predicted Values	x	True Positives	False Positives
	y	False Negatives	True Negatives

Table 1 – Confusion matrix example.

In Table 1, True Positives (**TP**) indicate the number of elements of class *x* that were correctly classified as *x*. True Negatives (**TN**) indicate the number of elements of class *y* that were correctly classified as *y*. False Positives (**FP**) indicate the number of elements of class *y* that were incorrectly classified as elements of class *x*, and False Negatives (**FN**) indicate the number of elements of class *x* that were incorrectly classified as *y*.

2.4.2 Precision

The *Precision* of a classifier refers to the proportion of correct positive predictions among all instances predicted as positive (POWERS, 2020). Based on Table 1, Precision measures the proportion of elements predicted as class *x* (i.e., True Positives + False Positives) that actually belong to class *x* (i.e., True Positives). It is calculated using Equation 2.1. Precision is a particularly useful performance metric when the cost of False Positives is high in the given application.

$$Precision = \frac{VP}{VP + FP} \tag{2.1}$$

2.4.3 Recall

The *Recall* of a classifier represents the proportion of actual positive instances that were correctly identified by the model (POWERS, 2020). This metric assesses how effectively the classifier can detect elements belonging to the positive class. Based on Table 1, Recall measures the proportion of elements from class x (i.e., True Positives + False Negatives) that were correctly predicted as class x (i.e., True Positives). It is computed using Equation 2.2. Recall is especially useful when the cost of False Negatives is high in the application.

$$Recall = \frac{VP}{VP + FN} \tag{2.2}$$

2.4.4 F-Score

The *F-Score* (or *F-Measure*) is a performance metric that reflects the balance between Precision and Recall (POWERS, 2020). It is calculated as the harmonic mean of these two metrics, as shown in Equation 2.3. In this thesis, the F-Score is used to evaluate

the effectiveness of the proposed techniques for identifying file maintainers, as well as the performance of the modified Truck Factor algorithm.

$$FM = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{2.3}$$

2.5 Truck Factor

Truck Factor (TF) is defined as *"the number of people on your team that have to be hit by a truck (or quit) before the project is in serious trouble"* (WILLIAMS; KESSLER, 2003). We can find different names in the literature, such as Bus Factor, Bus Number, or Lottery Factor (HARATIAN et al., 2023; JABRAYILZADE et al., 2022; KLIMOV et al., 2023).

This metric is used to identify key developers, verify the concentration of knowledge in a project, and allow risk assessment in the event of turnover in the development team. Ideally, to avoid potential problems and comply with the *"collective code ownership"* principle of Extreme Programming (XP) (BECK, 2000), the TF of a project should be as high as possible (RICCA; MARCHETTO, 2010). Techniques like *pair programming* can increase knowledge sharing and therefore increase the TF (ZAZWORKA et al., 2010).

There are some shared characteristics among most Truck Factor computing techniques. Most are based solely on development history information present in version control systems (COSENTINO; IZQUIERDO; CABOT, 2015; RIGBY et al., 2016; AVELINO et al., 2016; ZAZWORKA et al., 2010), while others combine this information with additional data, such as the number of review meetings (JABRAYILZADE et al., 2022). Another characteristic is that to estimate the Truck Factor, it is necessary to consider how many files are abandoned after a group *g* of developers leaves the project. For a team of *n* developers this involves $\binom{n}{g}$ calculations, which is infeasible for large projects (RIGBY et al., 2016). Therefore, approximate methods such as *Greedy Algorithms* or *Monte Carlo Simulations* are used (AVELINO et al., 2016; RIGBY et al., 2016).

In this thesis, we propose an enhancement to the Truck Factor algorithm, grounded in a study of source code knowledge models. Specifically, we use the Avelino Truck Factor algorithm (AVELINO et al., 2016), a well-established algorithm in the literature, to achieve this.

2.6 Generative Artificial Intelligence

Generative Artificial Intelligence (GenAI) refers to computational techniques capable of creating new content like text, images, or audio from training data (FEUERRIEGEL et al., 2024). These techniques are used for different purposes and can assist humans in intelligent question-answering systems (FEUERRIEGEL et al., 2024). GenAI is based on

generative modeling, which differs from discriminative modeling often used in data-driven decision support (NG; JORDAN, 2001; FEUERRIEGEL et al., 2024).

Generative AI uses large datasets to create new versions of text, images, or predicted data at the user's request (EUCHNER, 2023). These algorithms, including deep neural networks, transformers, generative adversarial networks, and variational autoencoders, excel at modeling high-dimensional probability distributions of language or images (LECUN; BENGIO; HINTON, 2015; CRESWELL et al., 2018; MURPHY, 2022).

Tools like GitHub *Copilot*[2] and *ChatGPT*[3] have significantly assisted developers by helping with programming, website building, and documentation. This democratizes coding for non-technical users, representing a major technological advance (GOZALO-BRIZUELA; GARRIDO-MERCHÁN, 2023). In video generation, GenAI aids in digital human videos, and human motion capture, though text-to-video models are still developing (GOZALO-BRIZUELA; GARRIDO-MERCHÁN, 2023).

Generative AI also simplifies 3D design, with applications in game creation, the metaverse, and urban planning. Models like *Adobe Firefly*[4] generate 3D models from various inputs (GOZALO-BRIZUELA; GARRIDO-MERCHÁN, 2023). Image generation technologies, such as *DALL-E 2*[5] and *Midjourney*[6], have advanced photorealism and creative art (GOZALO-BRIZUELA; GARRIDO-MERCHÁN, 2023).

AI-Driven Development Environments (AIDEs) integrate modern AI into IDEs like *Visual Studio Code*[7] and JetBrains *IntelliJ*[8], automating routine programming tasks (ERNST; BAVOTA, 2022). GenAI is poised to transform the software profession more than any recent technology, as evidenced by the release of ChatGPT to the public in 2022, which spurred the AI race (EBERT; LOURIDAS, 2023; ROOSE, 2023).

In addition to studying knowledge models and their applications, this thesis seeks to understand how the use of GenAI affects these models. Specifically, we investigated whether using ChatGPT to generate source code impacts the estimates of these models and what the consequences of this are.

[2] https://github.com/features/copilot
[3] https://openai.com/chatgpt
[4] https://www.adobe.com/br/products/firefly.html
[5] https://openai.com/index/dall-e-2
[6] https://www.midjourney.com/home
[7] https://code.visualstudio.com
[8] https://www.jetbrains.com/pt-br/idea

3 Related Works

This thesis investigates new source code knowledge models, their applications in the context of software development, and the impact of generative artificial intelligence on these models. Therefore, this Chapter was divided into three Sections. The first, Section 3.1, discusses proposals and comparisons of source code knowledge models in the literature. Next, Section 3.2 presents works that use different algorithms for computing Truck Factor from software repositories, and tools that implement them. Finally, Section 3.3 explores works that relate the use of generative AI with source code knowledge.

3.1 Source Code Knowledge Models

Many studies in the literature have focused on proposing and comparing source code knowledge models. These studies introduce models that use different variables to measure the knowledge of software developers.

Some of these works rely primarily on information about code changes. One of the first studies in this area is by McDonald e Ackerman (2000), where the *Line 10 rule* heuristic is used to identify the developer responsible for the last change in the code of a module and, from this, infer expertise. The idea behind this heuristic is that the developer who modified a file last has the most knowledge about that file. This concept was also used in later studies, such as the *iMacPro* approach proposed by Hossen, Kagdi e Poshyvanyk (2014), which recommends appropriate developers to resolve software change requests.

Other works focus on metrics for the number of changes made to an artifact to identify experts. These changes were studied at different granularities (THONGTANUNAM; TANTITHAMTHAVORN, 2024). Many studies use the number of a developer's *commits* in a file as a knowledge *proxy* (MILANO; CAFEO, 2024; BIRD et al., 2011; KAGDI; POSHYVANYK, 2009; BOCK et al., 2023; CANFORA et al., 2012). At a finer granularity, Mockus e Herbsleb (2002) considers the number of line-level changes to rank developers by expertise, using a tool called *Expertise Browser*. Girba et al. (2005) and Rahman e Devanbu (2011) consider a developer to have greater knowledge in a file if they are the author of the highest percentage of lines in the file. At an even finer granularity, the *Syde* tool, proposed by Hattori et al., records every change made by a developer (HATTORI; LANZA, 2009; HATTORI; LANZA, 2010). A change is recorded when the developer saves the file with modifications, without needing to perform a *commit*. This tool can identify experts in a file based on the number of small changes made. Again, the developer with the most knowledge is the one who has made the most changes to a given file.

Some studies combine commit history with other supplementary information

to identify expertise. Minto e Murphy (2007) uses the number of changes to a file as a measure of knowledge, but also considers the relationship between files that were changed together. Falcão et al. (2020), in a study of technical and social factors that influence the introduction of bugs, considers the number of commits made by the developer and other developers in inferring knowledge in files. Costa et al. consider commits made in different *branches* to identify familiarity with source code files and recommend developers for merge operations (COSTA et al., 2016; COSTA et al., 2019). Other works, such as those proposed by Sülün, Tüzün e Doğrusöz (2019), use the number of commits in the artifact of interest and other related artifacts to calculate knowledge and recommend code reviewers.

The studies mentioned in the previous paragraphs primarily rely on information about changes, such as the number of commits and who made the last change, to identify expertise. However, studies indicate that these variables alone are insufficient to identify expertise accurately (AVELINO et al., 2018; KRÜGER et al., 2018). For this reason, this thesis analyzes additional variables and their relationships with developers' source code knowledge.

However, some models in the literature combine change information with other data from Version Control Systems (VCS) to assess source code knowledge. The *Degree of Knowledge* (DOK), a model proposed by Fritz et al. (2014), integrates information related to the authorship (*Degree of Authorship*, DOA) that the developer has with the file, and the number of interactions such as selections and edits the developer made to the file, referred to as the *Degree of Interest* (DOI) (MURPHY; KERSTEN; FINDLATER, 2006; KERSTEN, 2007). Calculating the DOK model requires plugins installed in the development environment, which makes its application and comparison unfeasible for the study presented in this thesis. Additionally, the DOK model does not directly address the recency of interactions or consider file size in its knowledge calculation. These two factors (recency and file size) have been identified as significant in previous studies on knowledge calculation (KRÜGER et al., 2018; AVELINO et al., 2018).

Some techniques calculate the impact of time on source code knowledge, specifically focusing on recency. Lucas et al. (2020) use a hyperbolic function that models forgetting and relearning, which, combined with the number of interactions with the artifact, calculates the developer's knowledge. Silva et al. (2015) presented a model that calculates the developer's expertise in an artifact based on the number of changes made by the developer, using time windows. Other approaches that consider the recency of changes appear in studies focused on recommending developers to resolve change requests. Kagdi et al. (2012) proposed an approach that identifies source code files relevant to a given change request and identifies experts in those files, prioritizing developers who have made the most commits (KAGDI; HAMMAD; MALETIC, 2008; KAGDI; POSHYVANYK,

2009). Studies by Sülün, Tüzün e Doğrusöz (2021) and Asthana et al. (2019) use information about the recency of modifications to recommend code reviewers. Similarly, Chouchen et al. (2021) uses the recency of review comments to recommend the most suitable developers to review a code change. However, these studies did not provide an in-depth analysis showing how the variables were used to make these recommendations.

On the source of data used in the models, this work focuses exclusively on data present in VCS, specifically Git[1]. Some approaches use other sources, such as interactions with the code artifact (LUCAS et al., 2020; FRITZ; MURPHY; HILL, 2007), code review data (ASTHANA et al., 2019; HANNEBAUER et al., 2016), and the number of meetings related to a file (JABRAYILZADE et al., 2022). While all of these data sources are valid, they rely on tools such as plugins installed in the development environment or other specific tools used in companies. Due to the universality of VCSs like Git, its use as a data source becomes more practical, making the models that use it more generalizable.

Regarding the use of machine learning, Montandon, Silva e Valente (2019) investigated the performance of supervised and unsupervised classifiers in identifying library experts. While our study employs a similar data collection and analysis process, our objectives differ. We use classifiers to identify experts in source code files, while Montandon focuses on identifying experts in libraries and frameworks. Other examples of machine learning applications in the context of developer expertise address the bug-assignment problem (SAJEDI-BADASHIAN; STROULIA, 2020), which is distinct from the problem investigated here.

Some authors have compared expertise identification techniques. Avelino et al. (2018) compared the performance of the *Commits*, *Blame*, and *DOA* techniques in identifying maintainers of source code files. The results showed that these three techniques have similar performance and highlighted the importance of considering the recency of modifications and file size as strategies to improve these techniques. Hannebauer et al. (2016) compared the performance of eight algorithms for recommending code reviewers, with six based on modification expertise and two on review expertise. Anvik e Murphy (2007) compared two approaches to determining appropriate developers to resolve bug reports: one uses the *Line 10 rule* to define which developers are experts in the files associated with the bug report, and the other uses data from bugnets. The work presented in this article differs from these three studies in three main aspects: purpose, compared techniques, and scope.

Regarding the purpose of the research, our work aligns with Avelino et al. (2018) and Anvik e Murphy (2007), as we also compare expert identification techniques. However, we compare different techniques, particularly machine learning classifiers. Regarding the compared techniques, Avelino et al. (2018), Krüger et al. (2018), and Hannebauer et al.

[1] https://www.git-scm.com

(2016) used some of the techniques selected by us. However, we also investigate the performance of machine learning models. Additionally, our study covers a broader scope, utilizing data from more repositories than the previous studies.

3.2 Truck Factor Computation

Truck Factor, also known as *Bus Factor*, is a topic explored in several studies within the software quality literature. Some studies aim to develop and compare developer metrics, while others focus on implementing these algorithms.

3.2.1 Truck Factor Algorithms

We can highlight three studies that investigated ways to estimate the Truck Factor. First, Cosentino, Izquierdo e Cabot (2015) proposed that the Truck Factor of software can be determined using the concepts of *primary and secondary developers*. Primary developers (P) are those who have modified a minimum of X percent of the artifact, and secondary developers (S) are those who have modified at least $Y < X$. The artifact's Truck Factor is given by $|P\ U\ S|$. Second, Rigby et al. (2016) used the authorship of lines to calculate the Truck Factor. A line is *owned* by the developer who last modified it, and a file is considered abandoned when at least 90% of its lines are *not owned*. The authors analyzed Truck Factor scenarios by removing developers in a Monte Carlo simulation. Finally, Avelino et al. (2016) used the *Degree of Authorship* to identify *authors* of files and calculate the Truck Factor. They proposed an algorithm that simulates the abandonment of the project's main developers and identifies the impact by the number of files without *authors* (i.e., developers with high authorship on a file). The algorithm stops and returns the Truck Factor when more than 50% of the project files lose their authors.

These three approaches were compared in a study by Ferreira, Valente e Ferreira (2017). Their performances were evaluated in terms of Truck Factor computation and accuracy in identifying Truck Factor developers. The results showed that Avelino's algorithm is the most accurate in these two aspects.

Other studies have utilized the algorithm proposed by Avelino et al. (2016). Avelino et al. (2019) applied it to investigate open-source projects that faced abandonment by core developers, examining how many survived and why. Jabrayilzade et al. (2022) used it as a basis for comparing the accuracy of a proposed algorithm derived from it. Calefato et al. (2022) used it to identify core developers of open-source software to investigate their downtime and disengagement from projects. Canedo et al. (2020) leveraged it to gain insights into issues of segregation of women in OSS communities. Almarimi et al. (2021) used it to detect community issues in software projects. Finally, Karlsson, Andreas (2023) studied this algorithm and its variations to analyze and compare Truck Factors from public

and private projects, establishing foundations for implementations in the CodeScene[2] tool.

In this thesis, we modified the algorithm proposed by Avelino et al. (2016) to use knowledge models other than the *Degree of Authorship*. This allows us to investigate the possibility of improving this algorithm, which is considered the most accurate in the mentioned study.

Furthermore, it is possible to calculate the Truck Factor using information different from that in the VCS. Jabrayilzade et al. (2022) modified the *Degree of Authorship* to include information about the number of reviews made by the developer and the time spent in meetings about the commit. While this enhances the inference of developers' knowledge, this information is more difficult to obtain, making it challenging to generalize its use.

3.2.2 Truck Factor Tools

SonarQube[3] is an open-source platform designed to manage and improve code quality by identifying poorly written code that violates best practices (CAMPBELL; PAPAPETROU, 2013). The platform comes equipped with a range of functionalities in its default installation and can be further expanded through free and commercial plugins.

One of the extensions that complement *SonarQube* is the *SoftVis3D*[4] plugin, which allows users to view the directory and file structure of a project as if it were a city. Using a combination of colors and building heights, the tool indicates critical points in the code according to different metrics such as coverage, complexity, and the number of authors. The metric for the number of authors of a file is related to the concept of Truck Factor, but unlike the metrics used in this thesis, it only considers the *blame* measurements of the files, not other variables that would help identify the most robust expertise.

In addition to plugins, web tools are specialized in code analysis. *CodeScene*[5] is a proprietary code analysis tool that offers a variety of code quality metrics (TORNHILL, 2015; TORNHILL, 2018). Among these metrics, some are related to knowledge concentration, identification of experts, and calculation of lost knowledge, simulating a Truck Factor situation. However, like the previously mentioned tools, CodeScene only uses the number of *lines of code* (LoC) to identify authors (Karlsson, Andreas, 2023), representing a gap for tools that could implement improvements.

There are also other less commercial tools aimed at specific studies. For example, Avelino et al. (2016) presents a tool[6], along with a new algorithm to calculate the Truck Factor. Haratian et al. (2023) presented *BFSig*[7], another tool for estimating the Truck

[2] https://codescene.com
[3] http://www.sonarqube.org
[4] https://softvis3d.com
[5] https://codescene.com
[6] https://github.com/aserg-ufmg/Truck-Factor
[7] https://github.com/JetBrains-Research/file-importance

Factor, which considers the importance of software components in the calculation. Almarimi et al. (2021) in the study present the tool named *CsDetector*[8], which, among other community smells, is capable of estimating the Truck Factor. Finally, Klimov et al. (2023) introduce *Bus Factor Explorer*[9], a web application with an interface and an API to analyze and visualize Truck Factor information using the *Degree of Authorship* (DOA) as the knowledge model. However, even though each of these tools represents an advancement in the field, they are either not web-based, which makes their use by practitioners more difficult, or they do not implement the knowledge model and file importance metrics used in this study.

3.3 Generative Artificial Intelligence and Source Code Knowledge

To the best of our knowledge, no research examines the relationship between source code knowledge models and the use of Generative Artificial Intelligence (GenAI). However, some authors have raised concerns about how using GenAI to generate source code might impact programming knowledge and learning.

Researchers have already addressed this concern. Ernst e Bavota (2022) recognizes that novice programmers can benefit from the integration of *AI-driven Development Environments*, but there is a risk of not fully understanding the recommendations of these tools and simply accepting them. Denny et al. (2024), when discussing the challenges and opportunities faced by computing educators in the GenAI era, also cites *learner over-reliance*, trusting in the generated code without fully understanding it, as a key risk in student learning, a problem also pointed out by *Codex* developers (CHEN et al., 2021).

This concern is also evident among programming students. Yilmaz e Yilmaz (2023) analyzed the advantages and limitations of using ChatGPT in the learning process from the perspective of undergraduate students. Among the perceived limitations, the most common concern was that ChatGPT could lead programmers to become lazy and negatively impact their algorithmic thinking skills.

Similar findings were reported in a study involving students in a Python programming course, where some participants expressed concerns that ChatGPT could hinder the learning process by fostering dependency on the tool (MA; CHEN; KONOMI, 2024). Likewise, Java students using ChatGPT in their learning process shared similar concerns (RAHIM; RAHIM; MD, 2024). Finally, Prather et al. (2023) observed and interviewed students using Copilot in an introductory programming course to understand their perceptions of its benefits and risks. Several participants feared that relying on

[8] https://github.com/Nuri22/csDetector
[9] https://github.com/JetBrains-Research/bus-factor-explorer

Copilot could impede their learning by preventing them from fully understanding the auto-generated code.

This concern also extends to practitioners. Zhang et al. (2023) studied the perspectives of practitioners using Copilot during software development and cited concerns about understanding the code generated and integrated by the tool. Similarly, Russo (2024) studied the adoption of Generative AI tools by surveying 100 software engineers. Some respondents expressed concerns about programmers relying too much on these tools, which could lead to a decline in code understanding.

Even ChatGPT itself recognizes this issue. In a literature survey, Anagnostopoulos (2023) combines ChatGPT responses to questions about the implications of its use with literature studies related to the themes of each question. *Dependence on AI-generated code without proper understanding* was cited by ChatGPT as one of the most important limitations of its use in programming and training programmers.

Some studies have employed a methodology similar to this thesis. Grewal et al. (2024) analyzed ChatGPT conversation shared links to examine how generated code is integrated into GitHub projects. Using *Levenshtein* distance, the authors measured the extent to which generated code was incorporated into GitHub repositories. Similarly, (JIN et al., 2024) investigated ChatGPT's effectiveness in assisting developers with code generation in real-world scenarios. Applying a comparable approach, they found that in 16.8% of conversations, the generated code snippets were exact matches to code found in the main branch of the analyzed projects.

Although we did not identify studies addressing the relationship between generative AI and knowledge models, the existing literature highlights a notable research gap. It suggests that using generative AI tools for source code generation may lead to developers not fully comprehending the code generated and integrated into their projects. This should be reflected in source code knowledge models and, consequently, in knowledge concentration metrics. This thesis explores this gap, providing greater insights into the impact of GenAI on this software engineering domain.

4 Knowledge Models Proposal

This chapter outlines the initial phase of this thesis, which is focused on proposing new models for identifying expertise in source code files. It begins with creating a dataset that encapsulates authorship data relevant to source code knowledge. We then describe how this dataset was leveraged to build knowledge models through a correlation analysis of development history variables and their relationship to source code expertise. Finally, we evaluate the performance of these proposed models against widely used models in the literature. The chapter specifically addresses the following research questions:

RQ1: How do repository-based metrics correlate with developer's knowledge?

RQ2: How do machine learning classifiers compare with traditional techniques for identifying source code experts?

4.1 Study Design

A ground truth with data on source code experts is required to analyze which variables relate to source code knowledge and propose models of expertise. We built this oracle by extracting data from both open-source and industrial projects. This ground truth comprises the developers' knowledge of source code files and variables from development history.

4.1.1 Target Subjects

We selected open-source repositories from the GitHub platform.[1] To select these repositories, we adopted a procedure similar to other studies that investigate GitHub data (AVELINO et al., 2016). First, for six popular programming languages[2] - Java, Python, Ruby, JavaScript, PHP, and C++ - we selected the 50 most popular repositories as indicated by their number of stars. This measure is commonly used by researchers in the selection of repositories (AVELINO et al., 2016; RAY et al., 2014; PADHYE; MANI; SINHA, 2014; HILTON et al., 2016; MAZINANIAN et al., 2017; JIANG et al., 2017; NIELEBOCK; HEUMÜLLER; ORTMEIER, 2019; RIGGER et al., 2018; CASTRO; SCHOTS, 2018) and is perceived by developers as a proxy of popularity (BORGES; VALENTE, 2018).

After cloning the repositories, we performed a filtering step based on the number of commits, the number of files, and the number of developers. As in a previous work (AVELINO et al., 2016), for each language, we removed repositories in the first quartile of

[1] https://github.com

[2] Six popular programming languages https://octoverse.github.com/#top-languages

Table 2 – First quartiles of *commits*, *files*, and *number of developers* by programming language.

Language	Commits	Files	Developers
Python	510.75	87.50	45.25
Java	829.25	318.00	39.25
PHP	823.50	89.00	97.50
Ruby	1,650.25	152.75	198.25
C++	2,010.25	706.00	113.50
JavaScript	1,455.75	113.25	129.25

Table 3 – Summary of the target open source repositories with metrics on repository count, programming languages, number of developers, commits, and files.

Language	Repos	Devs	Commits	Files
Python	25	18,936	192,587	39,154
Java	17	5,733	138,473	62,429
PHP	16	9,802	144,092	29,902
Ruby	25	38,036	605,546	88,869
C++	14	11,467	350,345	72,991
JavaScript	14	10,319	109,541	21,477
Total:	111	94,293	1,540,584	314,822

the distribution for each metric, resulting in the intersection of the remaining sets. In other words, repositories with few commits, files, and developers were excluded. Table 2 presents the first quartile values for the three metrics across each programming language.

Additionally, we discarded repositories whose history suggests that most of the software was developed outside of GitHub by removing repositories where more than half of its files were added in a few commits. As few commits, we considered the outliers of the distribution of the number of files added in each commit of a repository; we discarded repositories if most of their files (>50%) were added by the set of outliers commits, following the procedure done by Avelino et al. (2016). The resulting dataset comprises 111 GitHub repositories distributed over the six programming languages, which have a relevant number of developers, files, and commits. Table 3 summarizes the characteristics of these 111 repositories.

We also utilized data from two industrial projects developed by the multinational networking and telecommunications company Ericsson[3], which was made available to us through a collaborator in this study who the company employs. Both projects were implemented in the Java programming language. Project #1 has a development history comprising 74,078 commits, 17,329 files, and 513 contributors, while Project #2 includes 26,678 commits, 15,930 files, and 262 contributors. These projects were selected because

[3] https://www.ericsson.com/en

they meet the same criteria used to choose the open-source repositories.

4.1.2 Oracle Construction

Extracting Development History. The data was extracted by collecting commits from the *master/default* branch of the repositories. First, we ran *git log –no-merge –find-renames*[4] to extract data from the commit logs. From each commit, we extracted: (1) changed files; (2) the name and email of the commit's author; and (3) the type of change: addition, modification, or rename. Then, we discarded files that did not contain source code and third-party libraries using the *Linguist* tool.[5] Finally, we handled developer aliases by following the procedure adopted in other works (AVELINO et al., 2018; AVELINO et al., 2016; AVELINO et al., 2017; AVELINO et al., 2019). We merged developer histories with the same email and used the Levenshtein distance maximum modification threshold to merge histories of developers with similar names (CURY et al., 2022; NAVARRO, 2001).

Generating Survey Sample. We used information extracted from development history to create sample pairs *(developer, file)* for each repository. These samples are necessary to elaborate the survey we applied. The samples were created by performing the following steps for each repository:

1. We randomly selected a file and retrieved the list of developers who touched it (created or modified).
2. We discarded the file if at least one of these developers reached the maximal limit (5) of files we plan to send to a developer (*file_limit*), asking about their expertise on the file. Otherwise, we added the file to each developer's list.
3. We repeated steps 1 and 2 until there were no more files to be verified.

Step 2 ensures that a file will be added to the sample only if it is possible to add all developers who modified it. This step aims to maximize the possibility of obtaining answers from all developers who touched a file. We established a *file_limit* of five to avoid discouraging developers from responding to the survey or responding randomly.

Ultimately, we created 20,564 pairs for the open-source repositories (including 7,803 developers, with an average of 2.64 files per developer) and 394 pairs for the Ericsson repositories (including 92 developers, with an average of 4.34 files per developer).

Sending the Survey. The developers were invited via email to evaluate their knowledge of the files on their list. To do so, we asked them to use a scale from 1 (one) to 5 (five), where

[4] https://git-scm.com/docs/git-log/1.5.6
[5] https://github.com/github/linguist/blob/master/lib/linguist/languages.yml

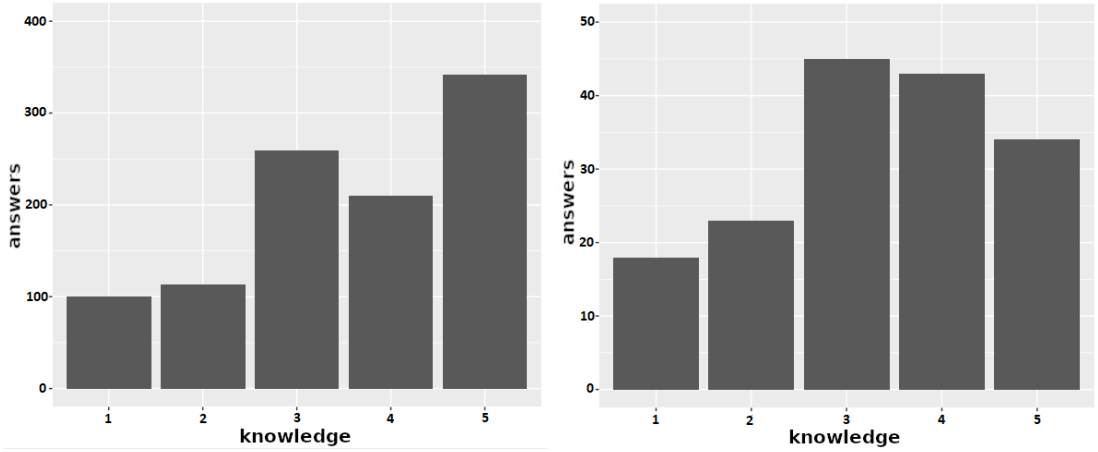


Figure 2 – Distribution of developers' self-assessed expertise levels, ranging from 1 (lowest) to 5 (highest), in the public project dataset (left) and the private project dataset (right).

(1) means you have no knowledge about the file's code; (3) means you would need to perform some investigations before reproducing the code; and (5) means you are an expert on this code. For the open-source repositories, we sent 7,803 emails, and 501 responses were received, representing a response rate of 7%. For the private repositories, we sent 92 emails, and 38 responses were received, representing a response rate of 41%.

Processing the Answers. We processed the answers by classifying each *(developer, file)* pair into one of two disjoint sets: *declared experts* (O_m) and *declared non-experts* ($O_{\overline{m}}$). A developer is considered a *declared expert* if they reported a knowledge level greater than 3 for a given file; otherwise, they are classified as a *declared non-expert*. This classification approach was also used in a previous study by Avelino et al. (2018) for a similar purpose. Figure 2 shows the distribution of responses in the public and private datasets. In the public dataset, 54% of the entries are *declared experts* and 46% are *declared non-experts*. In contrast, in the private dataset, 47% are *declared experts* and 53% are *declared non-experts*. As shown, the proportions of declared experts and non-experts are relatively balanced in both datasets. The complete dataset comprises 1,187 evaluations, with an average of 2.51 files evaluated per developer. The number of evaluations per developer has first, second, and third quartiles of 1, 1, and 3, respectively.

4.1.3 Development Variables

Extracting Variables. We selected 12 variables (or features) that can be extracted from the development history. These variables and their definitions are presented in Table 4.

The variables *Adds*, *Dels*, *Mods*, and *Conds* are extracted using the *git diff* command.[6] In this work, a modification is defined as a set of removed lines followed by

[6] https://git-scm.com/docs/git-diff

Table 4 – Variables extracted from the development history. The variable descriptions are given considering a developer *d* and a file *f* in its last version.

Variable	**Meaning**
Adds	Number of lines added by a developer *d* on a file *f*
Dels	Number of lines deleted by a developer *d* on a file *f*
Mods	Number of lines modified by a developer *d* on a file *f*
Conds	Number of conditional statements added by a developer *d* on a file *f*
Amount	Sum of number of added and deleted lines of a developer *d* on a file
FA	**F**irst **A**uthorship. Binary variable that indicates whether a developer *d* added the file *f* to the repository
Blame	Number of lines authored by a developer *d* that are in a file *f*
NumCommits	Number of commits made by a developer *d* on a file *f*
NumDays	Recency of Modification. Number of days since the last commit of a developer
NumModDevs	Number of developers that committed on the file *f* since the last commit of a developer *d*
Size	Number of lines of code (LOC) of the file *f*
AvgDaysCommits	Average number of days between the commits of a developer *d* on a file *f*

a set of added lines of the same size (IBIAPINA et al., 2017). We use *Levenshtein distance* (NAVARRO, 2001) to identify which pairs *(remove, add)* are modifications between two versions of the files. The algorithm takes the removed and added lines as input and returns the number of characters that must be modified to transform the removed line into the added line. In this work, a change is a modification if the returned value is less than a certain percentage (threshold) of the size of the removed line. A threshold of 40% was used, as suggested by Canfora, Cerulo e Penta (2007).

4.1.4 Compared techniques

This section outlines the techniques employed in this study to identify code experts. Their performance will be compared in the subsequent sections.

Commits. This technique counts the number of commits as a measure of knowledge that a developer has in a code file. The idea is that a developer gains knowledge in a file by modifying it. In other words, more commits are interpreted as more knowledge. This technique is widely used in assessing developer expertise, both individually (AVELINO et al., 2018; HANNEBAUER et al., 2016) and in combination with other methods (FRITZ et al., 2014; SUN et al., 2017). In this study, we use the number of commits individually as an expert identification technique and refer to it as *NumCommits*.

Blame. This technique infers the knowledge a developer has in a file by counting the number of lines they added in the latest version of the file. Blame-like tools such as *git-blame* command are used to identify the authors of each line.[7] As in other works (AVELINO et al., 2018; KRÜGER et al., 2018; HANNEBAUER et al., 2016), we use the percentage of lines associated with a developer as a measure of knowledge.

Degree of Authorship (DOA). Fritz et al. (2014) proposed that the knowledge on a source code file depends on the file's authorship, number of contributions, and number of changes made by others. The authors combined these variables into a linear model called *Degree of Authorship* (DOA). The weights associated with each variable in this model were defined through an empirical study based on the development history data from two Java systems. The authors used multiple linear regression to find the weights associated with each variable, and they claim that this model is robust enough to be applied in different analyses. In the studies by Avelino (AVELINO et al., 2016) and Ferreira (FERREIRA; VALENTE; FERREIRA, 2017), *DOA* is used to classify developers into experts using classification thresholds. The *DOA* value that a developer *d* has in the version *v* of a *f* file is calculated using Equation 4.1.

$$\textbf{DOA(d, f(v))} = 3.293 + 1.098 * FA + 0.164 * DL - 0.321 * ln(1 + AC) \tag{4.1}$$

where,

- **FA**: 1 if developer *d* is the creator of the file *f*, 0 otherwise; **DL**: is the number of changes made by developer *d* until version *v* of file *f*; **AC**: is the number of changes made by other developers in the file *f* up to version *v*.

Degree of Expertise (DOE). In many applications, it's important not only to identify experts but also to measure their level of expertise for ranking purposes (AKTER, 2021). To address this need, we used the development variables *Adds*, *FA*, *Size*, and *NumDays* (Table 4) to propose a linear model that calculates the knowledge of developers in source code files. The rationale for these choices is given in Section 4.2, where we present the results of the *RQ1*. We named this model *Degree of Expertise* (DOE).

Following the same approach as the *Degree of Authorship* (DOA), we propose using linear regression to define this model. To address scale differences among variables, a *log transformation* was applied to *Adds*, *NumDays*, and *Size*. The knowledge of a developer *d* in version *v* of file *f* is defined by Equation 4.2. To enable the model's

[7] https://git-scm.com/docs/git-blame

Table 5 – Table of estimated coefficients for the *Degree of Expertise* model.

	Estimate	**Std. Error**	**p-value**
Intercept (C)	5.28223	0.19685	<2e-16
Adds	0.23173	0.02724	<2e-16
FA	0.36151	0.10037	0.000329
NumDays	-0.19421	0.02400	1.46e-15
Size	-0.28761	0.03020	<2e-16
Residual std. error: 1.152; **R-squared**: 0.2256; **F-statistic**: 86.11; **p-value**: <2.2e-16			

application in real development contexts, we estimated its coefficients using the entire dataset for training. The final values are presented in Table 5.

$$\textbf{DOE(d, f(v))} = C + b_0 * ln(1 + \textbf{Adds}^{d,f(v)}) + b_1 * (\textbf{FA}^{f}) - b_2 * ln(1 + \textbf{NumDays}^{d,f(v)}) - b_3 * ln(\textbf{Size}^{f(v)}) \quad (4.2)$$

In addition to identifying key developers, the *DOE* model can be used to identify important files within a project. The underlying assumption is that files that have undergone significant modifications throughout the project's evolution tend to be more important, a notion supported by prior studies (MAEN; COLLARD; MALETIC, 2010; KPODJEDO et al., 2008). In an application of this model, presented in the next chapter, we define the *importance score* of a file as the sum of the *DOE* values of its contributors. However, we do not present this *importance score* as a contribution of this work, since evaluating the accuracy of this metric would require a more in-depth analysis, which is beyond the scope of this thesis. Moreover, the literature already includes more sophisticated metrics that consider additional aspects of software development (ŞORA; CHIRILA, 2019). Here, the *importance score* is used merely to demonstrate yet another applicability of the model.

Machine Learning Classifiers. We also investigate the applicability of machine learning algorithms for identifying source code experts, formulating the problem as a binary classification task. Based on the dataset described in Section 4.1.2, each *(developer, file)* pair was labeled as either non-expert (knowledge levels 1–3, according to survey responses) or expert (levels 4–5). We used the same variables employed in the *DOE* model as features to train the machine learning classifiers. Based on these feature values, the models aim to classify whether a developer is an expert on a given source code file.

We evaluated five well-known machine learning classifiers: Random Forest (LIAW; WIENER et al., 2002), Support Vector Machines (SVM) (WESTON; WATKINS, 1998), K-Nearest Neighbors (KNN) (PETERSON, 2009), Gradient Boosting Machine (GBM)

(FRIEDMAN, 2001), and Logistic Regression (JR; LEMESHOW; STURDIVANT, 2013). To assess their performance, we applied *10-fold cross-validation* with three repetitions. This method involves randomly partitioning the dataset into ten complementary subsets, using each subset once for validation while training on the remaining nine. The results are aggregated to produce a more robust estimate of model performance. Repeating the procedure three times with different random splits further reduces variability, and the average performance is reported. We use the *F-score* as the primary evaluation metric. A grid search was conducted to tune the hyperparameters and determine the optimal configuration for each classifier.

4.1.5 Linear Techniques Evaluation

To evaluate the performance of *NumCommits*, *Blame*, *DOA*, and *DOE* as classifiers, we need to define a *k* classification threshold for each one of them. To achieve this, we normalized the values of these metrics to a common range. Then, we tested the same set of thresholds for all metrics to find the best one and compare their performances in identifying experts. For the normalization process, we adopted a procedure from related work (AVELINO et al., 2018).

For this purpose, we set as 1 the expertise of a developer *d* in file *f* if *d* has the highest value for a given technique in *f*; otherwise, we set a proportional value. For example, suppose that for a given file *f* developers *d1*, *d2*, and *d3* have a *Blame* value of 10, 15, and 20. Their normalized values for blame regarding the file *f* are as follows: *expertise(d1, f)* = 10/20 = 0.5, *expertise(d2, f)* = 15/20 = 0.75, and *expertise(d3, f)* = 20/20 = 1. We apply this normalization process to all techniques. After normalization, a developer *d* is classified as an expert of a file *f* if its *expertise (d, f)* is higher or equal than a threshold *k*; otherwise, the developer is considered a *non-expert*. In this example, considering a threshold *k* = 0.7, developers *d2* and *d3* would be considered experts of the *f* file accordingly with the *Blame* technique. This normalization is done to facilitate the application of different classification thresholds for all techniques, which have different ranges of values, as explained in the next paragraphs.

Evaluating Performance. We compared the performance of the technique in their best settings. For the linear techniques, this requires setting the classification threshold *k* that maximizes the correct identification of file experts. Thus, we first compute the performance of each technique by adopting 11 different thresholds, i.e., by varying the threshold *k* from 0 to 1, under 0.1 steps. At the end of this process, the best performance of the linear techniques is obtained together with their associated thresholds *k*. For each threshold, we applied *10-fold Cross-Validation* to compute the *F-Score* of the techniques using

Equations 4.3, 4.4, and 4.5.

$$Precision(k) = \begin{cases} \frac{|(d,f) \in O_m \mid \textit{expertise(d, f)} > k|}{|\textit{expertise(d, f)} > k|}, & \text{if } k = 0 \\ \frac{|(d,f) \in O_m \mid \textit{expertise(d, f)} \geq k|}{|\textit{expertise(d, f)} \geq k|}, & \text{otherwise} \end{cases} \tag{4.3}$$

$$Recall(k) = \begin{cases} \frac{|(d,f) \in O_m \mid \textit{expertise(d, f)} > k|}{|O_m|}, & \text{if } k = 0 \\ \frac{|(d,f) \in O_m \mid \textit{expertise(d, f)} \geq k|}{|O_m|}, & \text{otherwise} \end{cases} \tag{4.4}$$

$$F - Score(k) = 2 * \frac{Precision(k) * Recall(k)}{Precision(k) + Recall(k)} \tag{4.5}$$

where O_m is the set of declared experts, as we inferred from the survey responses (Section 4.1.2).

Thresholds Calibration. Figure 3 shows the *F-Score* results for each linear technique at the analyzed thresholds under the public and private datasets. As we can see, *Blame* reaches its best performance when we use the lowest possible threshold ($k = 0$), in both datasets. In other words, by adopting $k = 0$, *Blame* reaches its best performance when it is used to classify as an expert any developer who has at least one line of code in the last version of the file. Similarly, *NumCommits* achieves the best performance by adopting the thresholds *k* = 0.1 and *k* = 0.3, in the public and private datasets, respectively. The low thresholds indicate that both techniques perform better by relying on minimal changes to consider developers as file experts. On the other hand, *DOA* has its best performance with a threshold *k* = 0.1. on public data; and *k* = 0.7, on private data. Finally, *DOE* had the best results with high thresholds of *k = 0.7* with public data, and *k = 0.8* with private data.

4.2 Results

This section presents the results of the two research questions proposed in this chapter. We present the correlation analysis between the extracted variables and developers' knowledge. Next, we present the performance of the techniques in our two datasets.

***RQ1:* How do repository-based metrics correlate with developer's knowledge?**

We analyze the correlations between the development variables and the developers' knowledge, as obtained in the survey. In the remainder of this chapter, the survey responses will be represented by a variable named *Knowledge*. We address *RQ1* and identify which variables could be included in a knowledge model.

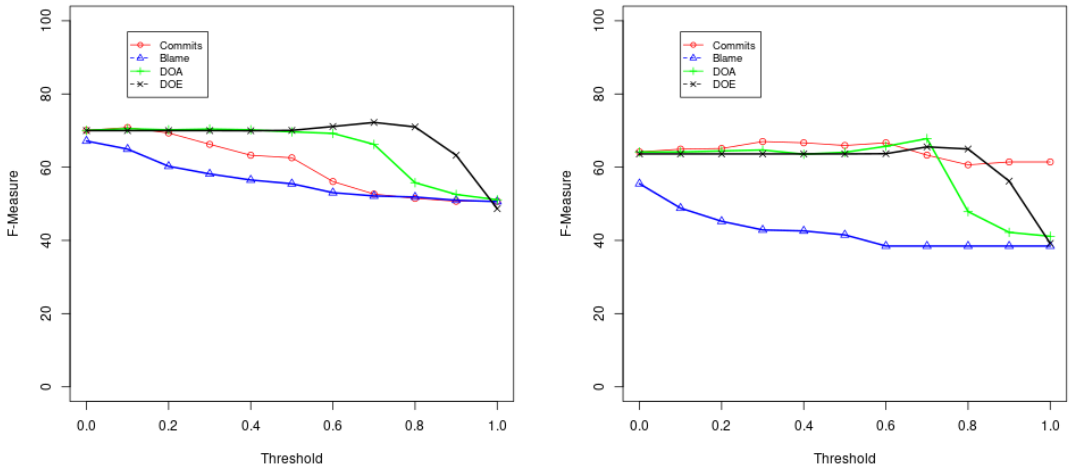


Figure 3 – Performance of techniques across analyzed thresholds using public and private datasets.

Table 6 – Correlation between extracted variables and *Knowledge*.

Variable	Corr. with Knowledge	p-value
NumDays	**- 0.24**	<0.001
Size	**- 0.21**	<0.001
NumModDevs	- 0.21	<0.001
Mods	0.01	0.515
Dels	0.02	0.369
Cond	0.19	<0.001
NumCommits	0.20	<0.001
AvgDaysCommits	0.21	<0.001
Amount	0.28	<0.001
Blame	0.29	<0.001
Adds	**0.31**	<0.001
FA	**0.33**	<0.001

Table 6 shows the directions and intensities of the correlations between the extracted variables and the *Knowledge* variable by applying Spearman's ρ. We consider results statistically significant when $p < 0.05$. Even though *NumCommits* is the most used variable for inferring a developer's knowledge, it does not show the strongest positive correlation with the *Knowledge* variable. The variable with the highest positive correlation is *FA*, closely followed by *Adds*. This suggests that a more fine-grained measure of changes like *Adds* can be more important than *NumCommits* to compose a knowledge model. Finally, the variable that shows the highest negative correlation is *NumDays*, followed by *NumModDevs* and *Size*. These results reinforce the importance of recency for inferring knowledge about a source code.

We also analyze the correlation between the extracted variables using Spearman's

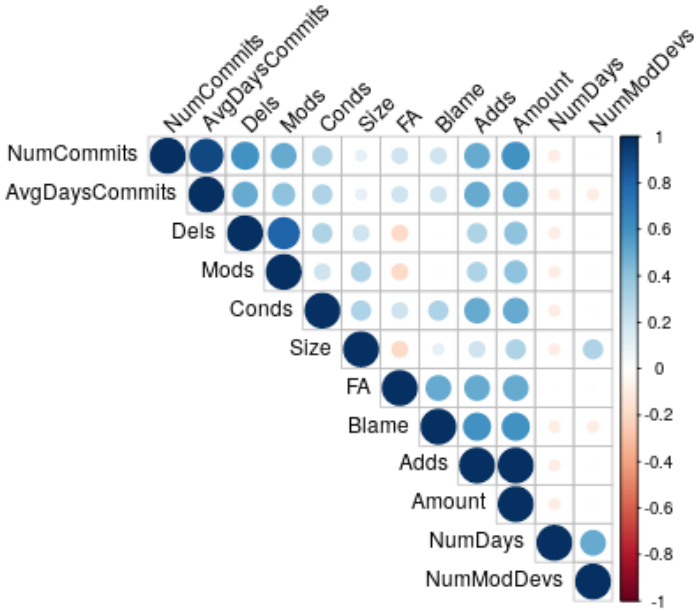


Figure 4 – Spearman correlation strength between the dependent variables.

ρ. These correlations are represented in Figure 4, where colors close to 1 and -1 represent higher positive and negative correlations, respectively. To interpret the correlations in terms of *weak*, *moderate*, and *strong*, we used the interpretation commonly used in some works (OVERHOLSER; SOWINSKI, 2008; SCHOBER; BOER; SCHWARTE, 2018).

As expected, *NumCommits* has a moderate correlation with *Adds*, *Dels*, *Mods*, *Amount*, and *AvgDaysCommits* ($\rho \geq 0.5$). Therefore, any one of these five variables can be used to measure the number of changes made by a developer throughout the history of a file. The variables *NumModDevs* and *NumDays* also show a moderate positive correlation ($\rho \geq 0.5$) with each other.

> **Summary of RQ1:** *First Authorship* (FA) has the highest positive correlation with knowledge in source code files, suggesting that file creators tend to have high knowledge of the files they create. On the other hand, *Recency of Modification* has the highest negative correlation, indicating that a developer's knowledge of a source code file decreases over time, since its last modification.

Variable Selection Rationale. In previous research, Avelino et al. (2018) described how the variables *file size* and *recency* influence the developer's knowledge on source code files. Thus, models that use this information tend to be more accurate. This previous

Table 7 – Performance of linear and machine learning techniques across public and private datasets.

		Public			Private		
	Metrics	**Precision**	**Recall**	**F-Score**	**Precision**	**Recall**	**F-Score**
Linear Metrics	**DOE**	0.60	0.91	0.72	0.50	0.95	0.64
	DOA	0.55	0.97	0.70	0.61	0.77	0.68
	Commits	0.60	0.87	0.70	0.55	0.86	0.67
	Blame	0.56	0.83	0.67	0.50	0.62	0.55
ML Classifiers	**Random Forest**	0.73	0.74	0.73	0.59	0.62	0.60
	SVM	0.71	0.75	0.73	0.63	0.60	0.61
	KNN	0.71	0.71	0.71	0.70	0.69	0.67
	GBM	0.73	0.73	0.73	0.65	0.58	0.60
	Logistic Regression	0.69	0.75	0.72	0.70	0.51	0.58

finding is reinforced by the results in Table 6, where these variables achieved the highest negative correlation with *Knowledge*. Therefore, we include the variables *Size* and *NumDays* in the proposed machine learning models. Considering the variables *Adds*, *Amount*, and *Blame*, we choose the variable *Adds*, as it has the highest positive correlation with *Knowledge* among the three. No more than one of the three variables is chosen because these variables are conceptually related, as explained in Section 4.1.3.

The variable *NumCommits* is not included in the models. It can be viewed as just a macro way of accounting for changes made by a developer to a source code file, and the variable *Adds* has already been chosen since it has a higher positive correlation with *Knowledge*. Additionally, *Adds* has a moderate correlation with *NumCommits* ($\rho \geq 0.5$, Figure 4). For a similar reason, *AvgDaysCommits* is not included in the model. We also add the variable *FA*, which has the highest positive correlation with *Knowledge* among all variables. The variable *NumModDevs* is not included due to its moderate correlation with *NumDays* ($\rho \geq 0.5$).

***RQ2:* How do machine learning classifiers compare with traditional techniques in identifying source code experts?**

After identifying and selecting the variables with the highest correlations with source code knowledge, we train the Machine Learning (ML) models and the Linear Regression model (DOE) using them as features. We then compared the performance of these models in identifying experts. Table 7 presents the performance of linear techniques and machine learning classifiers, in the two analyzed scenarios.

Among the linear techniques, using the public dataset (Table 7 - **Public**), *DOE* had the best *F-Score* (72%) closely followed by *DOA* and *NumCommits* (70%), and *Blame* had the lowest *F-Score* of 67%. The technique with the best *Recall* was *DOA* (97%), and the one with the lowest *Recall* in the same scenario is *Blame* (83%). Finally, the techniques with the highest *Precision* are *DOE* and *NumCommits* (60%), while *DOA* has the lowest *Precision* (55%). Using the private dataset (Table 7 - **Private**), *DOA* had the highest value

for *F-Score* (68%), closely followed by *NumCommits* (67%). As in the other scenario, *Blame* had the worst *F-Score* (55%). The best *Recall* was from *NumCommits* (86%), with *Blame* presenting the lowest *Recall* (62%). Regarding *Precision*, *DOA* achieved the best result (61%), and *Blame* the lowest one (50%).

Considering the machine learning models, using the public dataset (Table 7 - **Public**), *Random Forest*, *SVM*, and *GBM* had the best *F-Score* (73%), with the other two classifiers reaching close values. The classifiers with the highest *Recall* were *SVM* and *Logistic Regression* (75%), and *KNN* obtained the lowest result (71%). In terms of *Precision*, *GBM* and *Random Forest* have the highest values (73%). The lowest *Precision* is from *Logistic Regression* (69%). With the private dataset (Table 7 - **Private**), the classifier with the highest *F-Score* was *KNN* (67%) and *Logistic Regression* had the lowest one (58%). The best *Recall* was also obtained by *KNN* (69%) and *Logistic Regression* had the lowest one (51%). Finally, the best *Precision* was achieved by *Logistic Regression* and *KNN* (70%). *Random Forest* had the lowest *Precision* (59%).

> **Summary of RQ2:** In the public dataset, in general, machine learning classifiers outperformed the linear techniques (*F-Score* = 71% to 73%). In the private dataset, this advantage is not clear, with F-Score ranging from 55% to 68% for the linear techniques and 58% to 67% for *ML* techniques.

4.3 Discussion

The linear techniques and machine learning classifiers achieved similar performance, especially when considering the *F-Score*. Therefore, the choice of the best technique depends on the user's tolerance for false positives and false negatives. We envision two specific application scenarios. First, when the goal is to identify a small number of experts (or even just one) for tasks such as selecting a person to effectively onboard new developers (KAGDI; POSHYVANYK, 2009), machine learning classifiers are preferred due to their higher precision.

An example where greater precision is crucial is in identifying the concentration of knowledge within a project, which is related to measuring the Truck Factor. As discussed in our practical application in the next chapter, metrics that overestimate a project's Truck Factor—indicating a higher reliance on individual developers than exists—can provide a false sense of security. This suggests that the project is more resilient and reliant on a larger number of developers than it truly is (HARATIAN et al., 2023).

Second, when the application requires additional developer expertise, such as during the evolution or maintenance of non-complex features (HOSSEN; KAGDI; POSHYVANYK, 2014), tolerating some false positives becomes more justifiable. In these

scenarios, employing linear techniques is often beneficial, as they offer a wider selection of developers possessing greater knowledge in the files.

4.4 Threats to Validity

Construct Validity. Some developers who participated in the survey may have inflated their self-assessed knowledge due to concerns about the commercial use of this information. To address this issue, we clarified in the survey that their answers would be used only for academic purposes. There is also the possibility of random responses. To mitigate this risk and enhance the response rate, we included a small number of files for developers to assess their knowledge, with an average of 2.64 and 4.34 for open-source and private projects, respectively. The size of the questionnaire is a demotivating factor for the quality of responses in a survey (LINAKER et al., 2015). Related to this, there is the possibility that only a few developers from a project responded to the survey. This may introduce bias depending on the quality of the responses and expertise of the respondents. However, since the responses relate to hundreds of files, this concern is somewhat mitigated. In other words, our strategy tends to prioritize file coverage rather than the completeness of the responses per file.

Regarding the risk of misinterpretation of the knowledge scale (1 to 5) used in the first survey, we provided a guide to explain the score's meaning. However, the concept of *expert* is subjective, which remains a threat. Another issue is defining *expert* from the survey responses. Developers with knowledge scores above three were considered file experts. We believe this is a more conservative division of the knowledge scale, as applied in a similar study (AVELINO et al., 2018).

Internal Validity. Other factors may influence a developer's knowledge of a source code file, which were not considered in this study (JABRAYILZADE et al., 2022; FRITZ et al., 2014). However, we limited our scope to identifying experts using information from Version Control Systems (VCS), which are widely used tools, making the evaluated techniques applicable to most projects. Other variables can also be extracted from code repositories and may play a role in identifying file experts. For instance, interactions with files in the same module and the complexity of the changed code are relevant but more complex to compute and dependent on the programming language. For this reason, we focused on variables less dependent on the specific characteristics of the software being analyzed.

The knowledge models used in this chapter are ultimately based on source code modifications made by project contributors and are sensitive to loss of history in migrations to code hosting platforms. In selecting the repositories, we mitigated this threat by using a heuristic to detect systems where part of the development was performed outside the platform and later migrated to it, resulting in a partial loss of development history.

Conclusion Validity. For the statistical tests, we used Spearman's ρ to analyze the correlation between the extracted variables. This coefficient was chosen because it does not require a normal distribution or a linear relationship between the variables. There is no consensus on the interpretation of the correlation values returned by the test; however, we relied on interpretations commonly used in studies from different fields (OVERHOLSER; SOWINSKI, 2008). We used 10-fold cross-validation to assess the performance of the machine learning models, noting that results may vary somewhat depending on the number of folds chosen.

Another potential threat to our conclusions is selection bias in cross-validation. There is the possibility of concentrating evaluations from the same developer in the training folds, which could introduce bias. To address this issue, we used 10-fold cross-validation with three repetitions. This approach allows performance to be estimated with several different divisions of training and testing sets, thereby reducing the possibility of bias.

External Validity. We aimed to maximize the generalizability of our results. For open-source systems, we used six popular programming languages. By not limiting the projects to a single programming language, we intended to reduce the impact that certain languages may have on developers' familiarity assessments and, consequently, on our results. The small amount of data from private projects makes it difficult to generalize the results for this type of project, but these results can be generalized to projects with characteristics similar to those analyzed.

4.5 Conclusion

In this chapter, we investigated the use of knowledge models for identifying source code file expertise based on information in Version Control Systems. From a dataset composed of public and industrial projects, we identified the variables most related to code knowledge. *First Authorship* and *Number of Lines Added* showed the highest positive correlations with knowledge, while *Recency of Modification* and *File Size* had the highest negative correlations. Using variables from this correlation study, we evaluated a proposed linear model named *Degree of Expertise* and several *Machine Learning* models for identifying experts. We compared the performance of these models with other techniques, finding that three *Machine Learning* algorithms (*Random Forest*, *SVM*, and *Gradient Boosting Machines*) achieved the best *F-Score* and *Precision*.

Our findings can support the design and implementation of tools that automate the process of identifying file experts in software projects. We also provided insights into the variables most relevant to deriving new knowledge models. The models evaluated in

this chapter can be part of future investigations and research in repository analysis. Additionally, future research can assess the performance of the studied techniques in industrial contexts. One application context of these models is studied in the next chapter, where we use them in a metric for identifying the concentration of source code expertise.

5 Improving a Knowledge Concentration Measure

In this chapter, we describe a practical application of the new models proposed in Chapter 4, aiming to improve the results of an algorithm for Truck Factor estimation. To achieve this, we investigated the performance of the algorithm proposed by Avelino et al. (2016) for computing the Truck Factor of a project using two different models: *Degree of Expertise* (DOE) and *Random Forest* (RF). We chose these models because they performed best among the linear techniques and machine learning classifiers in the largest dataset used in Section 4.2. Specifically, this chapter aims to answer the following research question.

RQ3: Can the proposed models improve a state-of-the-art Truck Factor algorithm?

5.1 Study Design

In this section, we present how our study evaluated the modified algorithm using two complementary scenarios. First, in a quantitative analysis, we assessed its performance on an extended version of the dataset from Avelino's study (AVELINO et al., 2016), where previously missing *Truck Factor Developers* were added to the reference set generated using the *DOE* model. Second, in a qualitative analysis, we surveyed development leaders from an industrial project and compared the accuracy of the *Truck Factor Developer* lists produced by the evaluated models.

5.1.1 Expert Identification and Truck Factor

We start with a description of the Truck Factor algorithm and how we modified it in this study. Avelino's Truck Factor algorithm follows a greedy approach that relies on an authorship metric to identify the top authors of a system and iteratively estimate the impact of removing the top developers.

First, the experts of each file in the project are identified by adopting the *Degree of Authorship* (DOA) metric. Then, the algorithm iteratively removes the developer who is the expert in the largest number of files and checks how many files become abandoned, i.e., files without experts after the removal. When more than half of the project's files have no expert, the algorithm stops, returning the Truck Factor estimation of the number of experts removed. This procedure is represented in Algorithm 1.

Algorithm 1 High-level pseudocode of the algorithm proposed by Avelino for computing the Truck Factor of a software project.

```
1:  E ← getExperts()
2:  F ← getFiles(E)
3:  tf ← 0
4:  while E ≠ ∅ do
5:      coverage ← getcoverage(F, E)
6:      if coverage ≤ 0.5 then
7:          break;
8:      E ← removeTopAuthor()
9:      tf ← tf + 1;
10: return tf;
```

Therefore, as a practical application of the results found in Chapter 4, we investigate the results of modifying the original algorithm, that adopts the *DOA* model, by replacing it with *DOE* and *Random Forest* models for classifying developers into experts (line 1). For *DOE*, we used the coefficients presented in Table 5 and the best threshold pointed in Section 4.1.4. Regarding Random Forest, we also obtained a final model by training the model with all data. We checked Random Forest's performance by running the procedure using *10-Fold Cross-Validation* with all the data, and the results were the same using only the public data: *Precision = 72%*, *Recall = 73%* and *F-Score = 72%*.

5.1.2 Data Gathering

In this section, we describe how we collected the data for the two analyses we performed. First, we explain how we extracted data from public and private projects. Then, we detail how we mined the comments that identified missing important developers in the list of Truck Factor Developers generated in the previous study, and how we designed and applied the survey with the private project development leaders.

Extracting Development Data. In the original study, Avelino et al. (2016) computed the Truck Factor of 133 open-source projects using an algorithm that uses *DOA* to identify expertise. The list of projects can be found in the original paper.[1] We obtained these repositories in the versions used by Avelino to compute the truck factor using *DOE* and Random Forest for each repository, we cloned it and checkout using the command *git checkout 'git rev-list -1 –before=date #branchName'*. By doing so, we ensured that our analysis used the same project versions as in Avelino's study. However, we were unable to *checkout* three projects used: *ass/sass*,[2] *driftyco/ionic*,[3] *cucumber/cucumber*,[4] probably

[1] Publicly available http://aserg.labsoft.dcc.ufmg.br/truckfactor
[2] https://github.com/sass/sass
[3] https://github.com/driftyco/ionic
[4] https://github.com/cucumber/cucumber

Table 8 – Summary of the target repositories with metrics on repository count, programming languages, number of developers, commits, and files.

Language	Repos	Devs	Commits	Files
Python	22	7,682	218,673	12,988
Ruby	31	17,672	254,266	21,009
JavaScript	22	5,259	88,641	8,707
PHP	17	3,096	116,596	18,962
Java	20	3,980	287,431	92,138
C/C++	18	16,448	725,259	61,350
Total	130	54,137	1,690,866	215,154

due to internal changes in their historic structure. As a result, our analysis was conducted on 130 open-source projects for Truck Factor analysis. Table 8 summarizes the number of repositories for each programming language, along with the counts of commits, files, and developers included in our analysis.

Additionally, we also investigate the Truck Factor of a software project from a Brazilian health tech company[5]. The selected project is a *web system* developed in the *Java* programming language, using a framework developed by the company. The project has nine years of development history, and we analyzed *12,996 commits*, from *6,369 files*, with *94 contributors*.

To extract the development histories of these projects, we used the same strategy described in Section 4.1.2. For the private project, we handled part of the file selection manually, as *Linguist* ended up excluding some files with extensions used by the internal framework used by the company. We manually added the files with extensions *'jhm.xml'* and *'hbm.xml'*, which are used by the framework for creating screens and configuring persistence models respectively. We also discard some commits related to code migrations and systematic changes caused by library version updates. We understand that these changes do not require significant knowledge from developers and may potentially introduce errors in results, such as considering a developer with very few contributions as one of the project's main experts. To address this, we manually inspected the commits and decided to remove commits with more than *90 files changed*. Similar filtering has been done in related work to detect migrations in projects (AVELINO et al., 2016; AVELINO et al., 2019). This threshold works well for our analysis, however, we emphasize its context-specific nature and do not claim generalizability.

Extracting Previous Survey Comments. In Avelino's study, the authors applied a survey with the developers of the analyzed projects (AVELINO et al., 2016). The survey sought to validate whether the *Truck Factor Developers* identified by the algorithm were the main

[5] https://inas.maida.health

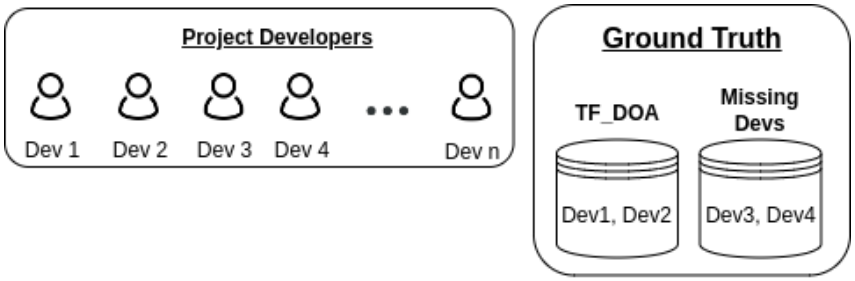
developers of the project and whether their departure would bring serious problems to the project, validating the Truck Factor estimation. This survey was applied by opening GitHub issues in the target projects.[6] Despite the majority of participants fully or partially agreeing with the listings presented, some answers also included important developers who contributed more recently to the projects. For example, comments such as: "*On the ipython/ipython repo, yes, though I would add Matthias Bussonier who is the expert on many pieces of the code.*", or "*I would have added @DayS and @WonderCsabo as main developers.*"

We looked for similar comments on the provided answers for the original study. Thus, we manually reviewed each answer, searching for comments that claimed deficiencies in the list of developers presented and pointed to missing developers. We identified 15 comments present in 14 projects, pointing to 36 missing developers from the presented lists. To ensure relevance, we opted to consider only comments from project contributors who were active at the time of commenting, applying the same inactivity threshold of one year without commits used in prior studies with public projects (AVELINO et al., 2019). By applying this filter, we excluded one comment present in *silexphp/Silex*,[7] leaving a total of 14 comments in 13 projects, citing 34 developers, as we couldn't find the comment authors contribution.

As in Avelino's study (AVELINO et al., 2016), we relied on respondents' opinions to verify the ability of our techniques to include these missing developers. We consider the union of the list of *Truck Factor Developers* generated by the algorithm using *DOA* plus the list of developers named in the comments as *ground truth* to verify the performance of the Truck Factor algorithm using the proposed models. To illustrate this evaluation, Figure 5 presents a project with *n* contributors. Using *DOA*, the algorithm points as *Truck Factor Developers dev1* and *dev2*. *Dev3* and *dev4* are pointed out by other contributors as important developers who should also be on *Truck Factor Developers*. Using the union of these lists as ground truth, we can evaluate *Precision*, *Recall* and *F-Score* of the *Truck Factor Developers* pointed by the algorithm using *DOE* and *Random Forest*.

In addition to the 15 comments mentioning missing Truck Factor Developers, we identified 20 comments in 18 projects that pointed to the lack of the recency factor in the results. We can cite as an example: "*I would say that this list is kind of old? I'm in the community for a while and never seem to most of those guys. Maybe we might need another metric other then just files updated?*" in *fog/fog* survey[8], or "*I'm definitely not qualified to be giving advice in this topic, but it could be a good idea to look at recent history as well.*" in *dropwizard/dropwizard* survey[9]. These comments were not used in our

[6] Publicly available http://aserg.labsoft.dcc.ufmg.br/truckfactor/survey.html
[7] https://github.com/silexphp/Silex/issues/1212
[8] https://github.com/fog/fog/issues/3654
[9] https://github.com/dropwizard/dropwizard/ issues/1207



	Truck Factor Devs	Precision	Recall	F-Score
DOE	Dev1, Dev2, Dev3	1	0.75	0.86
RF	Dev1, Dev2, Dev3, Dev4, Dev6	0.8	1	0.89

Figure 5 – Computing *Precision* and *Recall* based on the *Truck Factor Developers List* generated using the *DOE* and *Random Forest* models.

evaluations, but they demonstrate the lack of the recency factor in the computation of these Truck Factors using the original approach.

5.1.3 Survey Design and Application

In addition to the quantitative analysis, we conducted a qualitative analysis comparing the Truck Factors of a private company project using Avelino's algorithm with *DOA*, *DOE*, and *Random Forest* models. After computing the Truck Factor with these three models, we administered an online survey to six development leaders familiar with the project. Among them, two have more than five years of project experience, one has between two and three years, and three have between one and two years of experience.

In the survey introduction, we briefly explain the Truck Factor concept and the survey's goal. We also make clear the absence of any professional impact on the answers given. After that, we introduce the questionnaire with a question about the number of years of experience of the respondent in the analyzed project, followed by the three questions about the truck factor estimations. Following good survey design practices, we ran a feedback loop between the survey author and two other authors of this thesis until we reached a consensus, and ensured clarity and consistency in our questionnaire. The three questions and aims are presented as follows.

Question 1: In your opinion, which of the listings below best represents the main developers of the project?

Rationale: This question aims to assess which of the generated listings of primary developers most accurately reflects the reality of the project. It helps determine whether our models can identify crucial developers that the original approach may have overlooked.

Question 2: Do you agree that if all the developers in the chosen list abandoned the project it will face serious maintenance and evolution problems?

Rationale: This second question aligns closely with the Truck Factor concept and aims to determine whether the developers listed are truly essential for the project's continuity. Participants were provided with four response options on a Likert Scale: Strongly Disagree, Disagree, Agree, and Strongly Agree. We excluded the neutral option to prevent potential misinterpretation or its use as a catch-all response, a consideration supported by various studies (CHYUNG et al., 2017).

Question 3: Would you add any other developers to your chosen listing? Please provide the name(s) of the developer(s) if your answer is yes.

Rationale: The proposed models may not consider some aspects and leave out some important developers in the truck factor computation. With this third question, we seek to verify deficiencies in the lists presented. In addition to the three questions, we also left an open space to collect comments on any related topic.

5.2 Results

This section presents the results of both quantitative and qualitative evaluations, addressing our third research question *RQ3: Can the proposed models improve a state-of-the-art Truck Factor algorithm?* First, we present the performance results obtained by applying the proposed models and compare these findings with the original Truck Factor study. The comparison is conducted on an extended dataset containing missing developers, as identified by the system experts. Following this, we present the results from the survey conducted with developers of a private project.

Mined Comments Missing Developers: Initially, we analyze whether the Truck Factor algorithm using *DOE* and *Random Forest* (RF) manages to list the developers present in our extended ground truth of the 13 selected projects. For this purpose, we compute *Precision*, *Recall*, and *F-Score* of the Truck Factor variations for each project. Table 9 shows Truck Factor using *DOE*, *RF* and our ground truth composed by the original Truck Factor *DOA* plus the missing developers (Devs+), the performances using *DOE* and *RF* in terms of *Precision*, *Recall* and *F-Score*.

Analyzing first the results of *Precision*, the algorithm with *DOE* achieves the best performance (100%) for the *mailpile/Mailpile* and *ReactiveX/RxJava* systems. Conversely, it exhibits the poorest performance (28%) for the *chef/chef* system. Notably,

Table 9 – Performance of variations of Avelino's Truck Factor algorithm on the extended ground truth dataset.

Project	Truck Factor				Precision		Recall		F-Score	
	GroundTruth		DOE	RF	DOE	RF	DOE	RF	DOE	RF
	DOA	Devs+								
fog/fog	12	1	21	10	0.57	1	0.92	0.77	0.71	0.87
chef/chef	6	2	29	20	0.28	0.40	1	1	0.43	0.57
emberjs/ember.js	6	6	30	21	0.40	0.49	1	0.92	0.57	0.67
spotify/luigi	6	1	15	14	0.47	0.50	1	1	0.64	0.67
androidannotations/ androidannotations	2	2	6	4	0.67	1	1	1	0.80	1
mailpile/Mailpile	1	1	2	2	1	1	1	1	1	1
ReactiveX/RxJava	1	1	2	1	1	1	1	0.50	1	0.67
ipython/ipython	4	1	10	6	0.40	0.57	0.80	0.80	0.53	0.72
yiisoft/yii2	3	1	9	7	0.44	0.57	1	1	0.62	0.72
elasticsearch/ elasticsearch	2	10	11	8	0.72	0.88	0.67	0.58	0.70	0.70
elasticsearch/ logstash	1	6	6	4	0.67	1	0.57	0.57	0.62	0.72
Respect/Validation	2	1	5	4	0.60	0.75	1	1	0.75	0.86
dropwizard/metrics	1	1	5	2	0.40	0.50	1	0.50	0.57	0.50
Avg.	**3.62**	**2.62**	**11.62**	**7.92**	**0.59**	**0.74**	**0.92**	**0.82**	**0.69**	**0.74**

RF demonstrates superior overall performance, achieving a *Precision* of 100% in five systems with the worst result being 40% for the *chef/chef* system. The reverse happens when we analyze the results of *Recall*. Using *DOE*, the algorithm obtained maximum performance (100%) in nine systems, with the worst result of 57%, while using *RF* it had 100% of *Recall* in six, with the worst result of 50% (*ReactiveX/RxJava* and *dropwizard/metrics*).

As we can observe, when using *DOE*, the algorithm classifies more developers as experts which results in high *Recalls*, with an average of 92%, and lower *Precisions*, with an average of 59%, resulting an average *F-Score* of 69%. Using *RF* we had an average *Precision* of 74% and an average *Recall* of 82%, which, in general, resulted in better *F-Scores*, with an average of 74%. These results show that, in most cases, the variations using the proposed models managed to include the missing developers, as evidenced by the high *Recalls*. However, they also added other developers, which were considered false positives.

We obtain some cases of much larger Truck Factor using the proposed techniques, as is the case of the *chef/chef* ($DOA = 6$, $DOE = 29$, and $RF = 20$) and *emberjs/ember.js* ($DOA = 6$, $DOE = 30$, and $RF = 21$) projects. However, even in cases where the difference is small, as the *mailpile/Mailpile* ($DOA = 1$, $DOE = 2$, and $RF = 2$) and *androidannotations/androidannotations* ($DOA = 2$, $DOE = 6$, and $RF = 4$) the improved algorithms can include exactly the developers mentioned in the comments. In cases where the difference between the Truck Factor of the variations and the *DOA* is quite large like

Table 10 – Measures on Truck Factors of the 130 open-source projects.

	First quartile	Second Quartile	Third Quartile	Mean	Std. Deviation
DOA	1	2	4	3.97	6.15
DOE	3	7	17	18.01	34.60
Random Forest	2	6	14	14.58	40.10

the two mentioned above, we found comments that indicate a greater number of missing important developers, such as: "*Everyone else listed is definitely a 'main developer', but there are quite a few other people who have enough knowledge to navigate and maintain the codebase.*" (*chef/chef*) and "*Losing those four people would certainly have an impact on the project, but I don't think it would be fatal - again, this is due to the growing set of major contributors.*" (*emberjs/ember.js*). These observations suggest that there are additional key developers beyond those explicitly named in the comments, which could potentially improve the *Precision* results of our models.

Furthermore, using *RF*, we obtain a smaller average Truck Factor of *7.92*, and with *DOE* a higher average of *11.62*. This tendency is reinforced in the computation of the Truck Factor of all the 130 selected open-source projects. Table 10 presents the results of computing the truck factor for the 130 projects. We can see that, in this analysis, the algorithm using *RF* has the mean and the first, second, and third quartiles greater than those reported in the original study using *DOA*, and smaller than those using the *DOE*. This reinforces the earlier findings regarding these techniques' *Recall* and *Precision*.

This large variation found in the Truck Factors computed using *DOE* and *RF* is explained by the presence of outliers (Figure 6) leaving the distribution quite asymmetric.[10] Looking at why there were these outliers, we take as an example the two biggest ones: *torvalds/linux* and *rails/rails*.[11,12] Using *DOE* as an example, we have *linux* with a *TF = 311*, and *rails* with a *TF = 185*, and with Random Forest we have *linux* with a *TF = 423*, and *rails* with a *TF = 143*. In comparison, using *DOA* we have *linux* with a *TF = 57*, and *rails* with a *TF = 9*. The exceptionally high Truck Factors observed in these two example projects are attributable to the extended convergence time of the algorithm using our models. In other words, it takes too many iterations, removing experts, until the project's file coverage drops below 50% (Algorithm 1).

Investigating these two projects, we decided to plot the percentage of expertise of each developer to all analyzed files of the project, i.e., the number of files in which each developer is an expert divided by the total number of files analyzed in the project. Figure 7 shows this expertise percentage distribution for the two projects using the algorithm with

[10] The truck factor values are publicly available https://doi.org/10.5281/zenodo.10806775
[11] https://github.com/torvalds/linux
[12] https://github.com/rails/rails

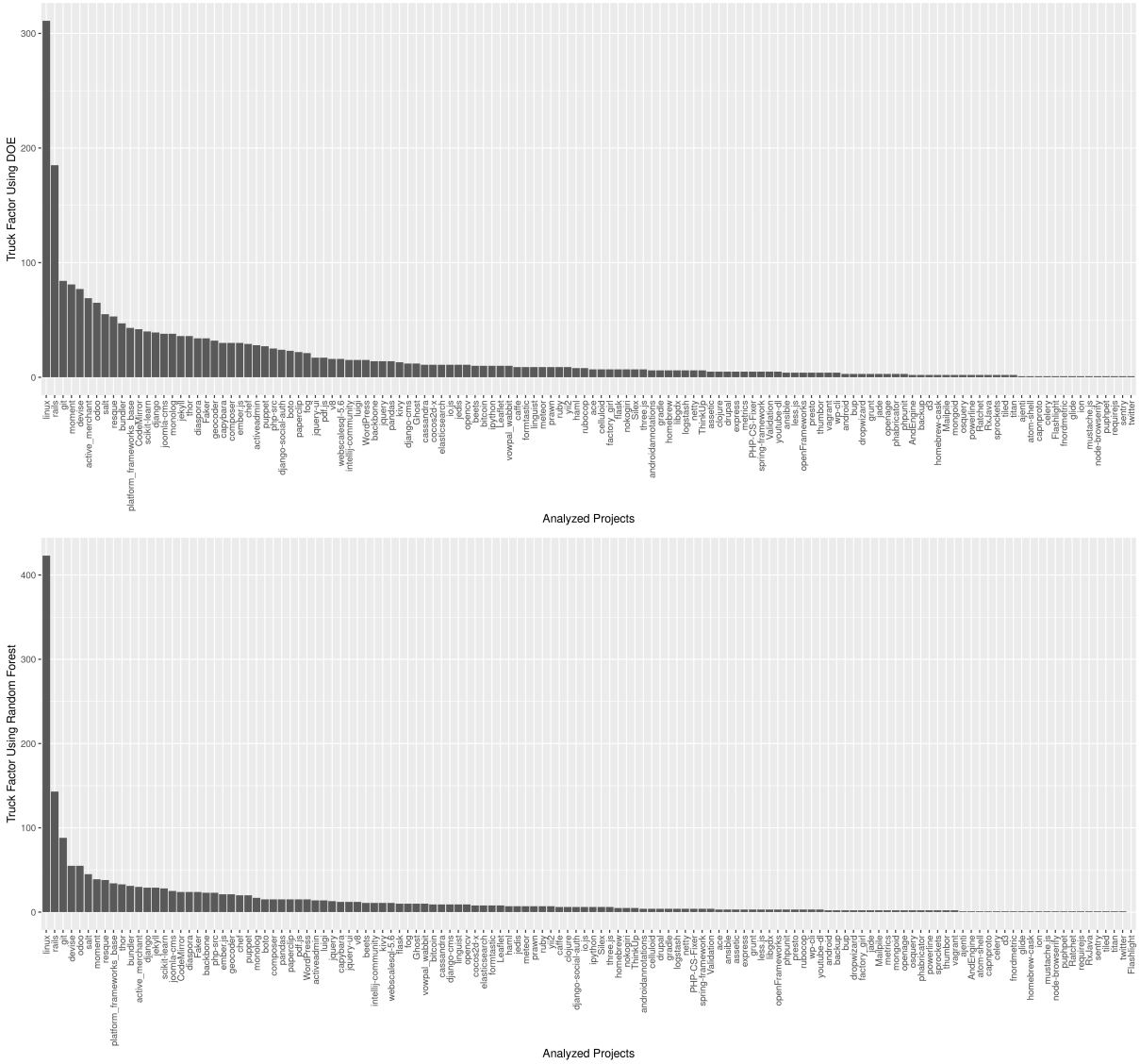


Figure 6 – Truck Factor distribution using *Degree of Expertise* (DOE) and *Random Forest* of all 130 selected open-source projects.

DOE as an example. We can see that the distribution of expertise in these projects is very uneven. We have a few developers with expertise in a considerable part of the system files, while many others in only a small portion. This is consistent with the results of the authorship distribution analysis in the Linux kernel from a previous study (AVELINO et al., 2017).

Furthermore, the proposed techniques manage to attribute expertise in cases that *DOA* does not. In Figure 8, we can see the proportions of developers considered experts in at least one project file of the analyzed projects using the algorithm with the two proposed models. Using *DOE*, most target systems have a significant portion of developers identified as experts in some project file with the first, second, and third quartiles of 56%, 69%, and 81% respectively, and with *RF* we have 56%, 65%, and 72% for the quartiles respectively. These values are much higher than the quartiles reported by

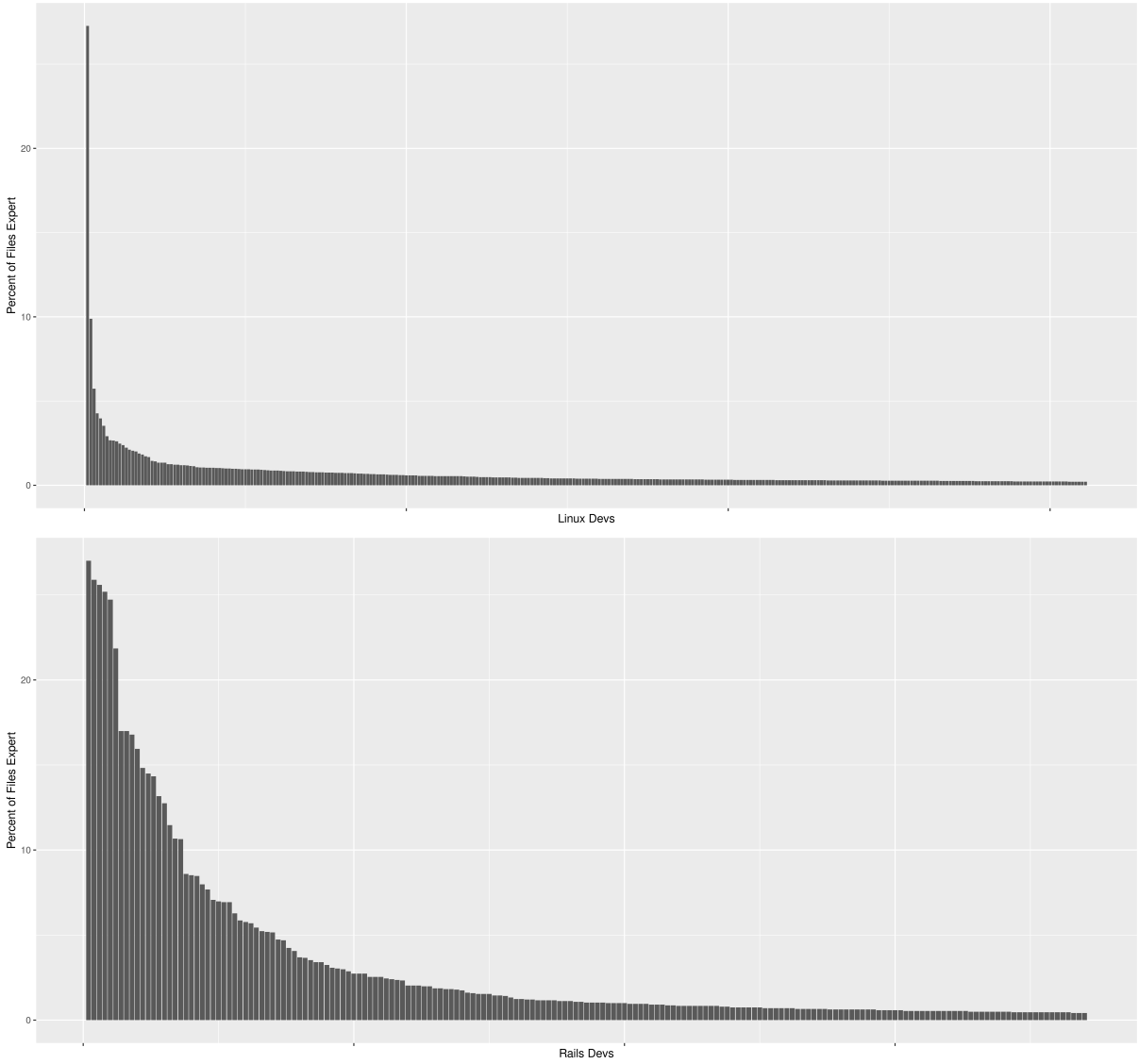


Figure 7 – Expertise distribution among Linux and Rails developers using the *Degree of Expertise* (DOE) model.

Avelino et al. (2016) of 16%, 23%, and 36%. The higher proportions contribute to the long tail observed in expertise distributions, sometimes complicating algorithm convergence despite the removal of developers with greater knowledge.

Based on the situation exemplified in Figure 7, a possible solution for these Truck Factors outliers is to define thresholds of expertise to include a developer in the Truck Factor computation. This *p* threshold would indicate the minimum percentage of files that the developer would have to have expertise to be considered in the calculation of Truck Factor. For example, using the examples previously presented, if we use a low threshold *p = 1%*, in the computation of *rails* we would end up with a *TF = 97*, excluding 48% of the developers from the computation without the threshold, and with *linux* we would have a *TF = 60* excluding 85% of the developers of the original computation. This is equivalent to saying that the 50% abandoned files threshold used in the original algorithm is not suitable

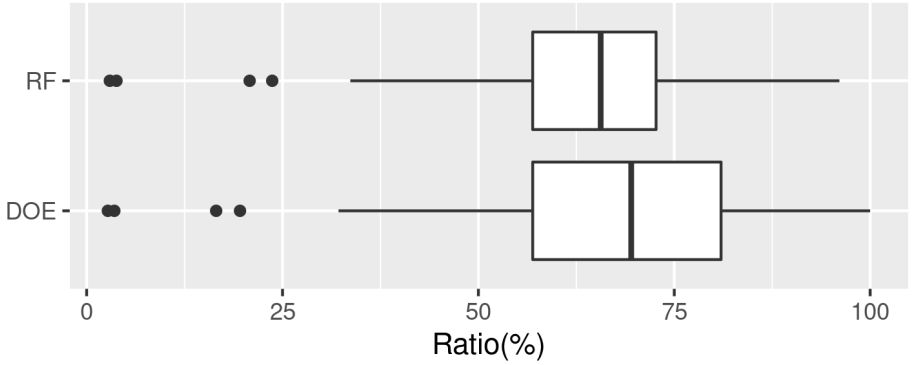


Figure 8 – Proportion of developers ranked as experts.

for all cases, and the ideal algorithm should be flexible according to the distribution of expertise in the project.

Survey Results. We also computed the Truck Factor of the industrial project described in Section 5.1.3 with variations of Algorithm 1 using *DOA*, *DOE* and *RF*. With *DOA*, following the analysis pattern of the open-source projects, we had the lowest *TF = 4*, with *DOE* we had the highest of *TF = 12*, and with *Random Forest* a *TF = 7*. Analyzing the Truck Factor developers lists generated by each variation, we observed that they were subsets of each other, i.e., the set of Truck Factor developers computed using *DOA* (O_{doa}) is contained in the set with *RF* (O_{rf}) which in turn is contained in the set with *DOE* (O_{doe}) - $O_{doa} \subset O_{rf} \subset O_{doe}$. As in the previous analysis, we have a correspondence between the models, with the proposed models ranking the same developers as *DOA*, but adding other ones that *DOA* cannot identify their expertise.

With the Truck Factor developer listings generated, we applied the survey described in Section 5.1.2. For *Question 1* - "*In your opinion, which of the listings below best represents the main developers of the project?*" - four participants pointed out the list generated using *DOE* as the most representative, while two participants chose the one using *RF*. This result shows that the proposed techniques could include relevant developers that *DOA* could not. This is reinforced when we look at the developers included by *DOA* and find that all four have already left the project. Of the four, the developer with the most recent contribution was two years before this computation. Thus, according to the algorithm using *DOA*, the project had experienced a Truck Factor event, or a *Truck Factor Developers Detachment (TFDD)* (AVELINO et al., 2019). In contrast, using both *DOE* and *RF*, the algorithm was able to include important developers who had already left the project, and two relevant developers still active in the project. This corresponds more with the reality of the project, which continues to evolve, and having the two developers appointed as technical references for the project. To better investigate this behavior, we computed the Truck Factor of the project over the past seven years aiming to verify the

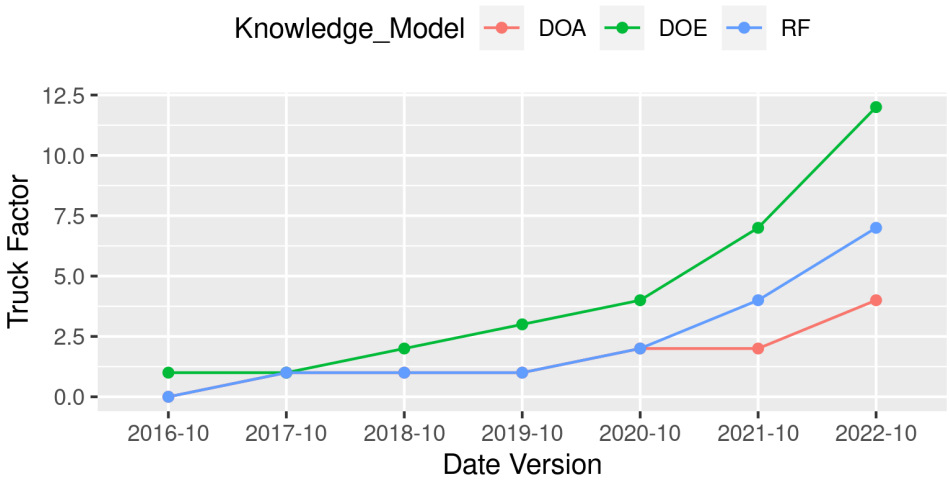


Figure 9 – Evolution over time of Truck Factor values according to the three studied models.

evolution of the Truck Factor according to the 3 metrics. Starting from the analysis date, we iteratively backward year by year performing the *checkout* of the project and computing the Truck Factor using each of the three models. We can verify in Figure 9 that by adopting the *DOA* algorithm there was not much evolution of the Truck Factor across the years, always concentrated in the initial relevant developers of the project, unlike the other algorithms that can include new important contributors.

For *Question 2* - "*Do you agree that if all the developers in the chosen list abandoned the project, it would face serious problems in its maintenance and evolution?*" - four of the respondents *Agreed* and two *Strongly agreed* that if the chosen developers abandoned the project, it would enter a Truck Factor situation. This current situation where the project knowledge is hanging on a few developers is reinforced by a respondent's comment: "*I've been at the company for two and a half years and I'm already one of the oldest devs. When I joined I had reference devs to guide me on the project. Devs coming in today should struggle with this.*"

And for *Question 3* - "*Would you add any other developers to your chosen listing? Please provide the name(s) of the developer(s) if your answer is yes.*" - one respondent informed a developer. The informed developer is active in the project, with 306 commits, an expert in 233 files - 4.5% of the project files - according to *DOE*, and an expert in 192 files according to *RF*. For *DOE*, the informed developer is the 13th contributor with the most expertise in project files, just one place after 12 in the calculated Truck Factor, and 10th for *RF*, three places after 7 in the computed Truck Factor, which shows that the models approached the respondent's perception

> **Summary of RQ3:** Our models improved Avelino's Truck Factor algorithm by including important developers with recent contributions. In a quantitative analysis, the algorithm had an average *F-Score* of 69% using the *Degree of Expertise* and 74% using *Random Forest*. The improvement in the Truck Factor estimation was further validated in a qualitative study, in which the lists of *Truck Factor Developers* generated using our models were perceived as more accurate in a survey with six development leaders of an industrial project.

5.3 Discussion

We begin by highlighting the main finding of the results presented: *the models proposed in this work can be applied to successfully improve the performance of algorithms to identify knowledge concentration in software projects*. In summary, the quantitative analysis reveals that the algorithm successfully incorporates significant developers previously overlooked when augmented with our models. Additionally, the qualitative analysis indicates that the computed Truck Factors are more accurate compared to those derived from the original model.

In general, the results of this practical application were consistent with those presented in Chapter 4, in Section 4.2. When using *DOE*, the algorithm classifies more developers as experts which results in higher *Recalls* and lower *Precisions*, which means more false positives and fewer false negatives. We had the opposite with *RF*, with higher *Precisions* and lower *Recalls* and better *F-Scores*.

In the results of the first question of the survey, the list of Truck Factor developers generated with *DOE* was perceived as the most accurate by the majority, with others pointing to the list generated by *RF*. If we look at the literature, one of the main applications of Truck Factor is the identification of bottlenecks in development. This happens when a few developers are responsible for a part of the system, and they are overloaded with other activities or are not available. This situation is associated with the community smell known as *code red* and is one of the most widespread community smells (JABRAYILZADE et al., 2022). There is an interpretation in the literature that a Truck Factor algorithm that reports fewer false positives is more desirable, as it reduces the risk of giving a false sense of security when the Truck Factor is overestimated (HARATIAN et al., 2023). Following this interpretation, the variation of the algorithm that uses *RF* is the most appropriate for practical use, as it presents fewer false positives and greater precision in all studies. According to the best of our knowledge, this is the only interpretation relating Truck Factors results to false positive and negative measurements. Still, we are not excluding the existence of applications less sensitive to false positives, a topic that can be explored in future studies.

5.4 Threats to Validity

Construct Validity. Some developers who participated in the survey may have concerns about choosing certain Truck Factor developers for professional reasons. As done in the survey in Chapter 4, we clarified that their answers would be used only for academic purposes. Another risk is relying on developer comments to build ground truth in the Truck Factor application. To mitigate this risk, we only considered comments from active developers, bringing more reliability to the opinions, a similar approach used in a related study (AVELINO et al., 2016).

Internal Validity. The knowledge models used are based on source code modifications made by project contributors and are sensitive to loss of history during migrations to code hosting platforms. In selecting repositories in Chapter 4, we mitigated this threat by using a heuristic to detect systems that had part of their development performed outside the platform and migrated to it, thus losing some development history. Avelino reports that a similar procedure was used in the selection of projects for his study (AVELINO et al., 2016), the same projects used in the Truck Factor study.

5.5 Knowledge Islands: Visualizing Developers Knowledge Concentration

In this section, we present *Knowledge Island*, a web tool designed to identify the concentration of knowledge within source code repositories, using our improvements in Avelino's Truck Factor algorithm described in this Chapter. This open-source tool[13] enables developers and project managers to clone GitHub repositories asynchronously and identify knowledge concentrations by calculating the Truck Factor of each project component. By pinpointing components with a low Truck Factor, Knowledge Island highlights areas where knowledge is concentrated in a single developer or small group, potentially posing a risk to project stability (CURY; AVELINO, 2024).

5.5.1 Implementation Overview

Figure 10 provides a visual overview of the main actions and components of Knowledge Island. The tool follows a simple client-server architecture divided into a front-end and a back-end. The front-end is responsible for acquiring user input data, such as information from the repository to be cloned and analyzed, and presenting the processed results. The back-end consists of a *RESTful API* and access to a relational

[13] https://github.com/OtavioCury/knowledge-islands

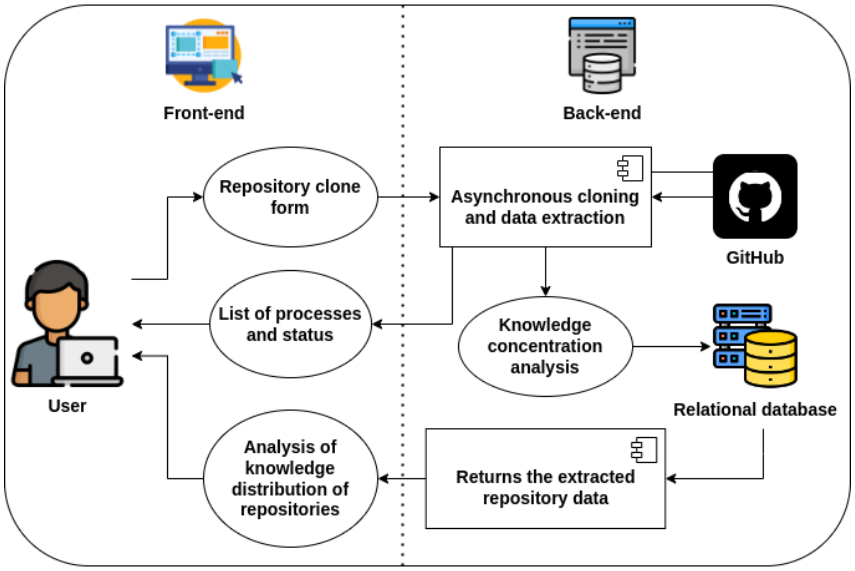


Figure 10 – Knowledge Islands operation diagram.

database to store user data and their repositories.

The front-end is implemented using the *React*[14] Javascript library, version 18. To help with the build of web components such as tables and forms and their styling, we mainly use the component libraries *React Bootstrap*[15] and *Material UI*[16]. The main components of the front-end include a form for repository cloning and analysis, a list of cloned repositories with their analysis process status, a detail page containing repository knowledge concentration information, and pages/components for user registration and authentication.

On the other hand, the back-end is implemented in the *Java* programming language (version 17) and uses a *PostgreSQL* database for data persistence. We employ the *Spring Boot*[17] framework to build the API and its endpoints. The back-end of Knowledge Islands provides a set of endpoints for managing and retrieving information about GitHub repository cloning and analysis results, as presented in Table 11. The first endpoint allows users to initiate the cloning and analysis process for a specified GitHub repository by providing the GitHub URL and specific branch name. The second one retrieves a list of all cloning and analysis processes started by a particular user, enabling users to track the status of their requests. The third endpoint provides access to the analysis results of a specific repository, including the knowledge concentration information and Truck Factor calculations.

To clone and manipulate repositories in the back-end, we utilize the Java library

[14] https://react.dev
[15] https://react -bootstrap.netlify.app
[16] https://mui.com
[17] https://spring.io/projects/spring-boot

Table 11 – Main endpoints of the Knowledge Islands API.

Endpoint	Method	Description
/git-repository-version-process/start-git-repository-version-process	POST	Initiates the cloning and analysis of a repository. Receives JSON form with GitHub URL of the repository, and the name of the branch.
/git-repository-version-process/user/<id>	GET	Lists the repository analysis processes initiated by a user.
/git-repository-version/<id>	GET	Returns the analysis results of a repository.

JGit[18]. This powerful library allows for efficient handling of Git operations programmatically. We employ a set of publicly available scripts to assist in extracting development history data, which is essential for identifying knowledge within the code. These scripts are included in our GitHub repository[19] and play a crucial role in analyzing and processing the historical data needed for our analysis.

Finally, in the implementation of Knowledge Islands, we used the *Degree of Expertise* (DOE, Section 4.1.4) to identify expertise and calculate the projects' Truck Factor, as required. To calculate the Truck Factor of repositories, we used Avelino's algorithm (AVELINO et al., 2016), which is presented in this chapter (Section 5.1.1).

5.5.2 Usage Scenario

In this section we present a usage scenario following the main features of Knowledge Islands, briefly represented in Figure 10. In this example, we will use data from the *Spring-Data-JPA*[20] project, an important repository in the *Spring Ecosystem*[21], and *Apache Kafka*[22], another popular project for web development.

After completing the registration and authentication process, the user is directed to the Knowledge Islands *Home* page (see Figure 11). This page features a form that initiates a process of cloning and analysis of a public GitHub repository. The form, which interacts with the first endpoint described in Table 11, includes two fields: a mandatory field for entering the GitHub URL of the repository to be analyzed and an optional field for specifying the desired branch to be cloned and analyzed. The project's main branch will be cloned by default if the branch field is blank.

The analysis process begins asynchronously once the user clicks the 'Start Analysis' button (see Figure 11). The initiated tasks are then listed in a table beneath the form, allowing users to manage multiple analyses simultaneously. Each row in the table provides detailed information about the ongoing tasks, including the repository URL, the date and time the analysis started, and the current stage of the process. The stages range from 'Process Initialized' to 'Process Finished', providing a clear and concise overview of

[18] https://www.eclipse.org/jgit
[19] https://github.com/SERG-UFPI/knowledge-islands
[20] https://github.com/spring-projects/spring-data-jpa
[21] https://github.com/spring-projects
[22] https://github.com/apache/kafka

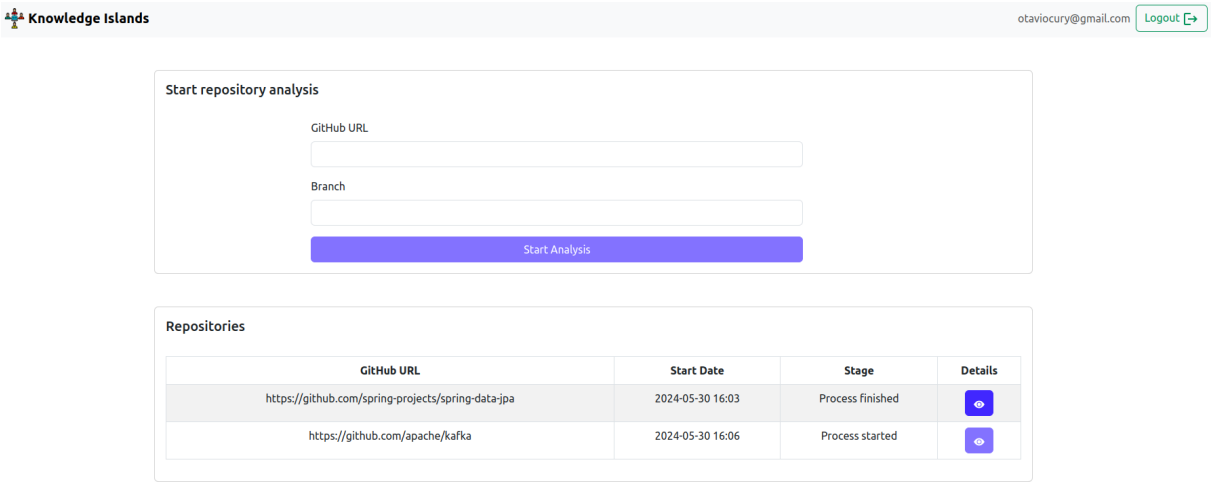


Figure 11 – Knowledge Islands page for cloning and listing analyzed repositories.

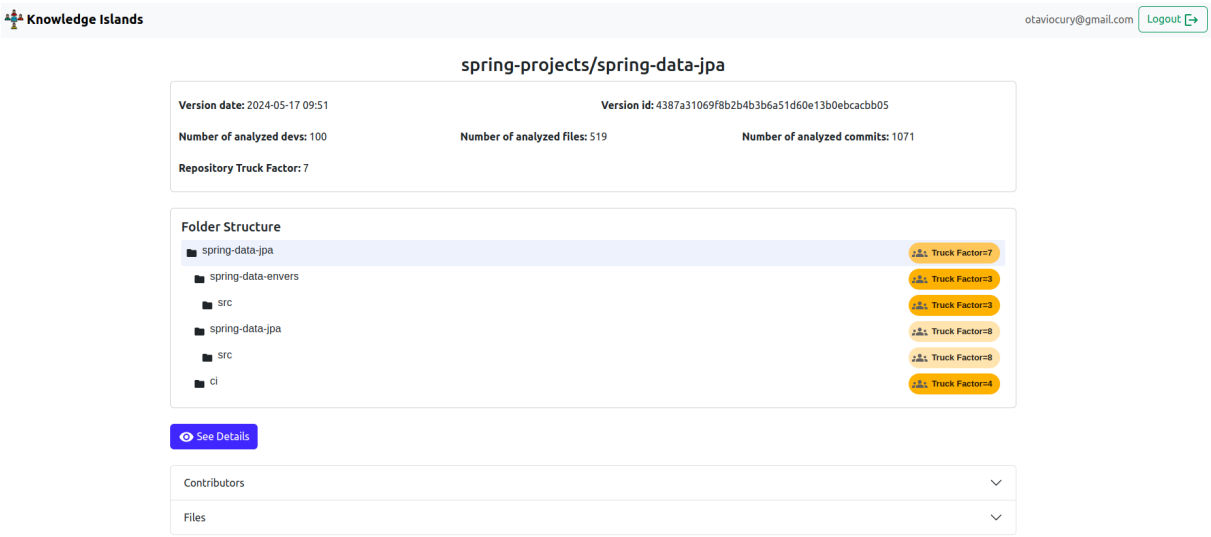


Figure 12 – Detailed analysis page of a repository in the Knowledge Islands tool.

the analysis progress for each repository. This setup ensures that users can easily track the status of their analyses and manage their workflows efficiently.

Only after a process has been completed ('Process Finished') can the user access the detailed analysis results via a button in the last column of the table (see Figure 11). Upon clicking the 'Details' button for a repository listed on the home page (Figure 11), the user is directed to the repository analysis detail page (Figure 12). This page presents a comprehensive view of the analyzed repository.

At the top of this repository detail page, the users find key information such as the analyzed version data, the repository size, measured by the number of developers, commits, and files, and the overall project Truck Factor. At the bottom of the page, the user will find a list of all developers and analyzed files.

At the center of the page, a tree component displays the project's directory structure, allowing users to navigate through folders and files. Alongside each item

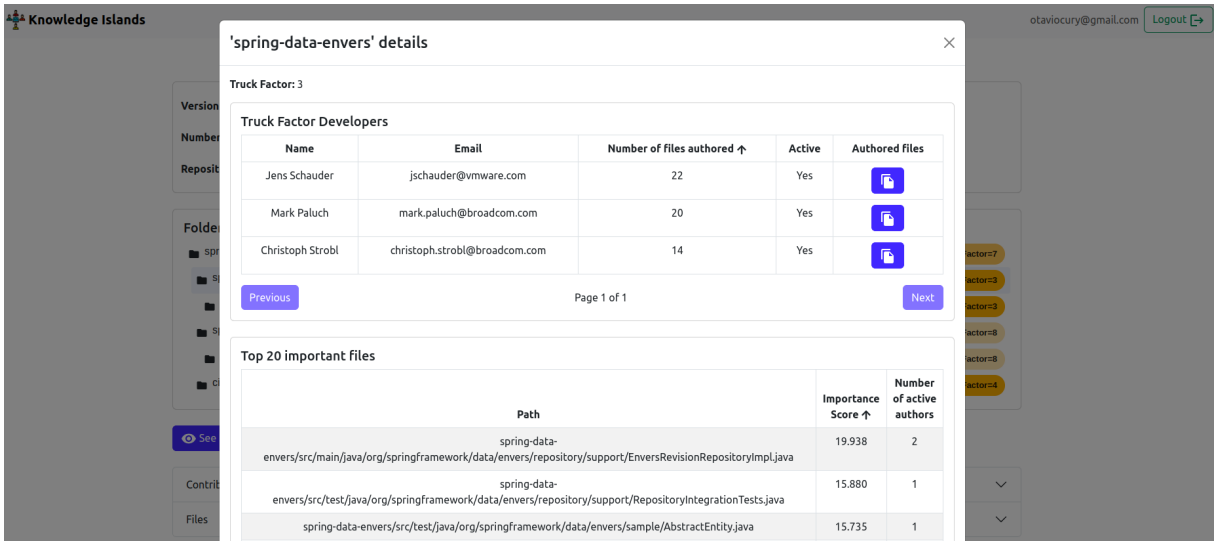


Figure 13 – Knowledge Islands modal displaying the Truck Factor details of a specific artifact.

(directory or file), a readily visible Truck Factor value indicates the level of knowledge concentration within that component (see Figure 12).

The label visually represents the Truck Factor of each item using a gradient scale of orange hues. A darker shade of orange signifies a lower Truck Factor, indicating a greater risk of knowledge loss or project disruption. This visual cue facilitates the identification of potential knowledge islands and enables proactive risk mitigation efforts.

The users can select a specific item in the tree component and click the 'See Details' button to access more in-depth information about its Truck Factor. A modal window then displays a table listing the Truck Factor developers responsible for that component, their names, email addresses, and the number of files they authored. This table is sorted in descending order by the number of files authored (see Figure 13). The user can then click the button in the last column 'Authored files' to expand the row and see a list of files. In short, this feature, besides indicating the main developers, indicates how many and which files they maintain.

The Truck Factor developer table includes a column indicating whether each developer is currently active in the project. The tool defines, by default, inactive developers as those who have not made any commits within the past year, following established criteria from the literature (AVELINO et al., 2019; FERREIRA; SILVA; VALENTE, 2020; CALEFATO et al., 2022). This information enables users to identify potential knowledge gaps and address them proactively.

Still in the Truck Factor details modal, beneath the list of Truck Factor developers, users will find a list of files along with their corresponding *importance scores*, as explained in Section 4.1.4, and the number of active authors associated with each file. This feature allows users to identify files with greater significance to the project and correlate this with

the number of developers knowledgeable about these files. Consequently, users can pinpoint key files at risk of being left without experts, potentially leading to project progress and maintenance complications.

5.5.3 Limitations

Knowledge Island effectively analyzes a software repository's source code knowledge concentration. However, we acknowledge limitations in the current version of the tool that will be addressed in future implementations.

Currently, the repository directory structure under analysis is presented only using the tree component, as shown in Figure 12. There are other ways to present this information, such as zoomable bubble plots, which can facilitate the visualization of the structure and knowledge concentration information. Different repository viewing options will provide users with a more flexible and intuitive experience.

Additionally, the knowledge model used in the tool could offer more options for users. As explained in Section 5.5.1, the tool currently employs the Degree of Expertise in Avelino's Truck Factor algorithm. Consequently, our implementation inherits all the limitations of the model discussed in this thesis. Incorporating other knowledge models would allow users to conduct knowledge concentration assessments considering different development history variables.

Additionally, although the simulation explores different usage scenarios, it does not necessarily reflect the full complexity of real-world contexts, particularly those encountered by large development teams. Nevertheless, we argue that this simulation serves as an exploratory study that highlights the sensitivity of the studied metrics under specific usage conditions.

Another limitation is that Knowledge Island lacks direct integration with GitHub, such as through a login process. The tool could benefit from using the GitHub API with OAuth 2.0 to authenticate users, which would streamline the process of analyzing data from their repositories.

5.6 Conclusion

In this chapter, we describe the practical application of the proposed source code knowledge models by using them to compute the Truck Factor of public and private systems. We demonstrate that with our models, Avelino's Truck Factor Algorithm identified relevant missing developers in open-source projects, achieving an average *F-Score* of 74%. Additionally, six development leaders perceived it as more accurate in computing the Truck Factor of a private repository.

Practitioners can significantly benefit from adopting the models proposed in this thesis, which excel in identifying source code experts (Chapter 4) and have applications in computing a project's Truck Factor, thereby mitigating risks associated with developer loss. Given the widespread recognition of Avelino's Algorithm in Truck Factor computation, our models can seamlessly integrate into software development workflows.

To exemplify an implementation, we also introduce Knowledge Islands, a tool for visualizing the concentration of knowledge in software repositories. Combining Avelino's Truck Factor Algorithm with one of the proposed models, Knowledge Islands assists developers and software managers in decision-making by providing metrics on the importance of developers and files in a repository. In the presented use case, utilizing data from public repositories, we demonstrate the process of downloading, extracting, and presenting data to the user. Knowledge Islands effectively showcases the repository's Truck Factors at various granularities: project, module, and file levels. The tool also highlights the top developers associated with each artifact and the most critical files.

6 Impacts of Generative Artificial Intelligence on Knowledge Models

With the increasing integration of Generative Artificial Intelligence into the software development process, concerns have emerged regarding the side effects of this automation. As explained in Section 3.3, one concern among practitioners and researchers is that using GenAI tools for source code generation can negatively impact developers' understanding of the integrated code. We believe that the knowledge models and their applications, studied in the previous chapters of this thesis, should reflect this impact.

Therefore, in this final research chapter, we present an exploratory study that investigates the impact of using Generative Artificial Intelligence (GenAI) tools for source code generation on knowledge models and knowledge concentration metrics. In addition, we aim to gather insights into how open-source developers perceive the influence of these tools on their ability to understand the code they work with. Specifically, we address the following two research questions.

RQ4: How does the use of GenAI tools for source code generation affect the accuracy and reliability of models that identify developer expertise in source code?

RQ5: What is the perception of developers regarding the relationship between using GenAI tools for source code generation and expertise in the generated code?

We begin by describing our use of the GitHub and ChatGPT APIs to collect data on the integration of GenAI-generated code into open-source repositories. This data is then combined with developer expertise metrics to address our research questions. Additionally, we leverage the mined data to conduct a survey targeting developers who have integrated ChatGPT-generated code into their projects.

6.1 Study Design

Our study design consists of four main steps. First, we select source code files that contain shared ChatGPT links. Next, we mine the development history of these files and retrieve the code generated in the associated ChatGPT conversations. By combining these two sources of information, we identify the extent to which Generative Artificial Intelligence (GenAI)-generated code was integrated into the project files. Based on this information, we

identify developers who have incorporated ChatGPT-generated code to some degree and invite them to participate in a survey by sending a questionnaire with six questions via email.

We then conduct a statistical analysis of the code integrations. Drawing on the insights from this analysis, we simulate various scenarios to assess the potential impact of GenAI on knowledge models, attributing a portion of the authorship to GenAI instead of developers. Finally, we analyze the survey responses, examining how developers' answers relate to the level of GenAI authorship in their files. Figure 14 provides an overview of the study design.

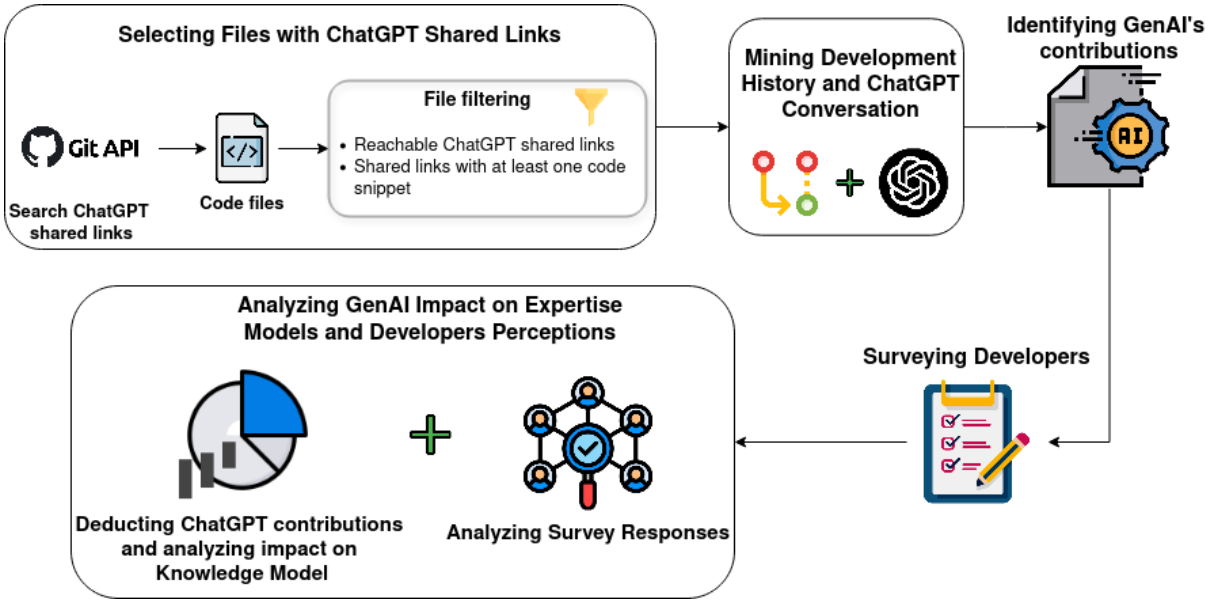


Figure 14 – Overview of the methodology for assessing the impact of GenAI on knowledge models and developer comprehension.

6.1.1 Selecting Files with ChatGPT Shared Links

In May 2023, OpenAI introduced a feature that allows users to share their conversations with ChatGPT via shareable links[1]. This feature enables developers to share their chats with ChatGPT, containing specific solutions integrated into the source code. These shared links appear in GitHub artifacts, and they have already been extracted in previous work (XIAO et al., 2024) and used in related studies (GREWAL et al., 2024; JIN et al., 2024; HAO et al., 2024).

Figure 15 presents an example of a shared link embedded in a Java source code file, along with the corresponding ChatGPT conversation. In this example, the code in the file is nearly identical to the snippet generated by ChatGPT.

[1] https://help.openai.com/en/articles/7925741-chatgpt-shared-links-faq

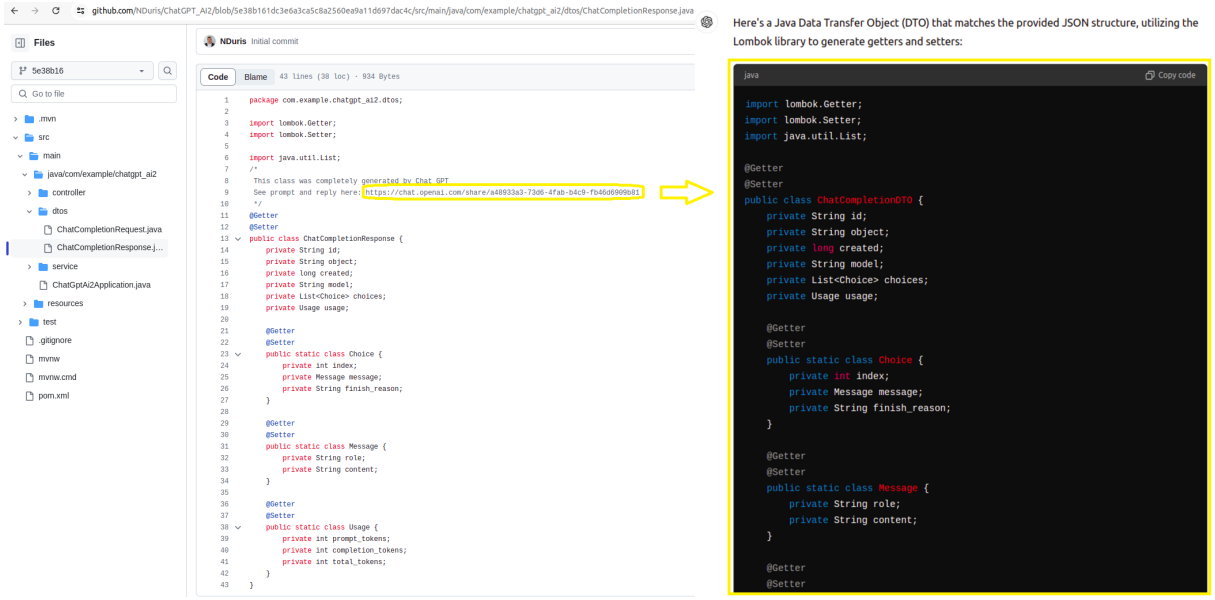


Figure 15 – Example of a ChatGPT shared link in a GitHub file and the associated conversation code snippet.

Inspired by the study of Xiao et al. (2024), in this study, we construct a dataset focused on shared links within GitHub *source code files*. We chose the source code files artifact because it is more closely related to the knowledge metrics used, facilitating the analysis. For our initial search, we utilized the GitHub REST API[2] to identify keywords that indicate the presence of ChatGPT shared links within the source code file contents. The following endpoint was used to locate these chat links in the source code: *<https://api.github.com/search/code?q=#chatgpt_url+language:#language>*.

We searched for links using two keywords (#chatgpt_url). The first, *<https://chat.openai.com/share/>*, was originally introduced by OpenAI (XIAO et al., 2024). The second, *<https://chatgpt.com/share/>*, emerged in 2024. To focus on source code files, we applied a programming language filter (#language), targeting the 10 most popular languages on GitHub in 2022[3]: JavaScript, Python, Java, TypeScript, C#, C++, PHP, Shell, C, and Ruby. This search, conducted in November 2024, identified 2579 files with shared links.

Due to the limitations of the API, which do not allow regex searches, we manually filtered the results to identify artifacts containing links that exactly match the shared link pattern using the following regular expressions: *<^https://chat.openai.com/share/[a-zA-Z0-9-]{36}>$*, and *<^https://chatgpt.com/share/[a-zA-Z0-9-]{36}>$*. After applying these to the file contents, we refined our dataset to 2502 source code files from 1354 repositories, totaling 2036 shared links.

Following the initial search and regex filtering, we applied two additional filters.

2 https://docs.github.com/en/rest?apiVersion=2022-11-28
3 https://octoverse.github.com/2022/top-programming-languages

Table 12 – Number of source code files and shared links per programming language before and after applying the filters.

	Before Filtering		After Filtering	
Language	**Files**	**Shared Links**	**Files**	**Shared Links**
Python	988	803	781	629
JavaScript	542	483	476	414
C++	251	195	223	169
Java	247	183	210	146
TypeScript	178	134	146	108
Shell	100	90	88	79
C#	80	62	69	51
C	68	63	59	53
PHP	45	27	41	22
Ruby	3	4	1	1

First, since shared links can be disabled after being created[4], we checked the reachability of each valid shared link by fetching the page content and checking for errors. Second, we verified that each successful fetch contained at least one code snippet from ChatGPT. We found 198 shared links without code snippets and 172 disabled links. As a result, our dataset was reduced to 2094 files in 1135 repositories, containing 1666 shared links. The number of source code files and shared links per programming language before and after applying these filters is shown in Table 12.

The results show a significantly higher number of links in Python files, 60% more than in the second most common language, JavaScript. This finding aligns with Xiao et al. (2024). Additionally, Python and JavaScript are among the languages whose users most frequently utilize ChatGPT and Copilot (PESLAK; KOVALCHICK, 2024; ZIEGLER et al., 2024).

After applying these filters, the distribution of the number of files with shared links per repository has a first quartile (Q1) of 1, a median (Q2) of 1, and a third quartile (Q3) of 1. Similarly, the shared link distribution per repository has Q1 = 1, Q2 = 1, and Q3 = 2. These statistics indicate sparse distributions, where most repositories contain only one file with shared links, and most of these files contain only a single shared link.

The entire process of searching and selecting files containing ChatGPT shared links was performed using the *Java* programming language. The implementation is available in a module of the *Knowledge Islands*[5] tool, presented in Section 5.5 of the previous chapter.

[4] https://help.openai.com/en/articles/7925741-chatgpt-shared-links-faq
[5] https://github.com/OtavioCury/knowledge-islands

6.1.2 Extracting Development History and GenAI Conversations

After selecting and filtering all relevant links and associated files, we constructed a dataset to analyze how the solutions provided in the links were integrated into their respective files. This step allowed us to assess the extent to which the generated code was adopted and how much of the developers' contributions could be attributed to ChatGPT.

Using the successfully fetched shared links described in Section 6.1.1, we first extracted the developers' prompts with the corresponding ChatGPT responses that included generated code snippets. We then cloned 1,135 repositories to collect the development history of the files containing the shared links. The vast majority of these repositories were relatively small, with a median of 1 contributor (Q1 = 1, Q3 = 2), 24 commits, and 34 files.

6.1.3 Identifying GenAI's Contributions

In addition to mining data to calculate the author's contributions to the files that contain the links, we also needed to assess how much of the code from each link was integrated into the file. Following the study by Grewal et al. (2024), we identified *matched lines* between code snippets and lines added in the same commit as the shared link. While their study used *Levenshtein Distance* for fuzzy matching, we take a more conservative approach and consider *only exact matches*. Our analysis is based on two key assumptions, explained below with their supporting rationale.

Assumption 1: *The exact matching lines identified correspond to instances of code directly copied and pasted from ChatGPT-generated outputs.*

Rationale: Since there is no known direct integration of ChatGPT with development environments, we consider lines that match the generated code as instances of *copying*. In the context of *Copilot*, matched lines are often referred to in research as *accepted* lines. A previous study by Dohmke, Iansiti e Richards (2023) found that developer satisfaction and productivity increase as the acceptance rate rises. With the growing use of ChatGPT for code generation (OVERFLOW, 2024; BRACHMAN et al., 2025), we can assume similar trends, although this may reduce developers' autonomy over the code (BIRD et al., 2023). Moreover, research on Copilot has already shown an increase in copied and repeated code (HARDING, 2025), while studies on ChatGPT indicate that more than 50% of generated code snippets are integrated without modifications (GREWAL et al., 2024).

Assumption 2: *Developers do not acquire the same level of knowledge from integrating copied ChatGPT-generated code.*

Rationale: Concern about the effects of GenAI on developers' learning and understanding of integrated code is well documented in the literature, reinforcing this assumption. This concern is reflected in recent reviews, which emphasize the risk of overconfidence eroding developers' core competencies when they passively accept model-generated suggestions (RAY, 2025; SINGH et al., 2025). Such passivity may impair code comprehension and foster cognitive dissonance regarding one's problem-solving abilities, potentially leading to an illusion of competence (PRATHER et al., 2023; PRATHER et al., 2024). When developers, especially juniors, adopt code without critical scrutiny, they forgo the opportunity to engage in the problem-solving process that is essential for internalizing and truly understanding a solution (LEHMANN; CORNELIUS; STING, 2024). Moreover, the very act of copying and pasting has been associated with shallower comprehension and reduced cognitive effort (LEHMANN; CORNELIUS; STING, 2024). Prior research suggests that actively retyping a solution improves comprehension and learning outcomes (GAWEDA et al., 2020; SKRIPCHUK et al., 2023). For instance, active code retyping has been a key component in methodologies designed to increase cognitive engagement with AI-generated code (KAZEMITABAAR et al., 2024). We argue that this potential loss of knowledge should be considered in knowledge models and reflected in their applications.

Building on these assumptions and the methodology described, we quantify the extent of copied code for each *file–shared link* pair. This provides statistical evidence of how ChatGPT-generated code is integrated into open-source projects, allowing us to simulate different usage scenarios by attributing varying levels of authorship to GenAI.

6.1.4 Analyzing GenAI Conversations and Code Integration

Among the 2,235 file–shared link pairs selected and filtered (Section 6.1.1), 1,699 (76%) contained at least one matched line. However, manual inspection revealed that some of these matches consisted only of single-character overlaps, such as individual symbols or keys. To improve data quality, we applied an additional filter to retain only pairs with at least one matched line containing more than one character. After this refinement, 1,672 pairs remained (74%), and only these will be considered in the subsequent analysis.

With this filter applied, we primarily analyzed the distribution of the percentage of code copied from ChatGPT by developers and how this varies across programming languages. Additionally, we assessed the number of conversation turns present in these solutions, a characteristic frequently explored in related studies.

6.1.5 Impact Simulation Design

As demonstrated in the previous sections, the dataset we constructed is sparse regarding the *link–file–repository* relationship. In most cases, a single link is added in a single commit to a single file within a repository. As a result, a direct analysis of the impact

of copied code from these links on files and repository history would be limited and not representative of realistic usage scenarios. Therefore, drawing on the statistical findings from the real data (presented in Section 6.2.1), we performed a *simulated* analysis to assess the potential impact of copied code.

Instead of simulating the impact on the repositories containing shared links, most of which are small and have limited relevance, we focused on more prominent repositories. Specifically, we selected the five most-starred GitHub projects for each of the ten chosen programming languages. We excluded 14 non-software repositories (e.g., code collections and roadmaps) due to their limited relevance for knowledge concentration analysis, and 12 additional projects were excluded based on size, following methodologies from related studies (AVELINO et al., 2016; CURY et al., 2022). Ultimately, 24 repositories were selected, as shown in Table 15 (Section 4.2).

We simulated the impact by applying a *uniform code copy rate*, derived from the statistical analysis presented in Section 6.2.1. For this simulation, we adopted the *Degree of Expertise* (DOE) model for two primary reasons. First, *DOE* estimates a developer's expertise in a file partially based on the number of lines they have added, which enables us to discount lines attributed to GenAI. This level of granularity is not supported by models like *Degree of Authorship* (DOA), which rely solely on commit counts. Second, as demonstrated in previous chapters, *DOE* outperforms other models in identifying developer expertise and in computing the Truck Factor of software projects (CURY et al., 2024).

We analyzed the impact of code copying on two key aspects within the scope of this study. First, we investigated how code copying affects the *DOE* values by evaluating the impact of excluding such contributions on developers' expertise scores. Second, we examined how these changes influence the Truck Factor of the projects, considering both variations in the Truck Factor values and shifts in the ranking of developers identified as key knowledge holders. To quantify changes in developer rankings, we applied the *Kendall Rank Correlation Coefficient*, which measures the degree of similarity between two ranked lists (ABDI, 2007).

For the Truck Factor analysis, we evaluated the impact by varying the percentage of files affected by code copying to 10%, 20%, 30%, 40%, and 50%. In each iteration, the affected files for each developer were randomly selected. Then, in every commit made by that developer on these files, we deducted the uniform code copy rate of the added lines, simulating GenAI authorship. Figure 16 illustrates the procedure. In each iteration, the Truck Factor was compared to the original results without GenAI impact. For the first analysis, which examines the impact on the expertise values calculated by the *DOE* model, we focus exclusively on the 50% impact scenario. This choice is justified by our interest in analyzing statistical metrics, such as the overall average impact on *DOE* values.

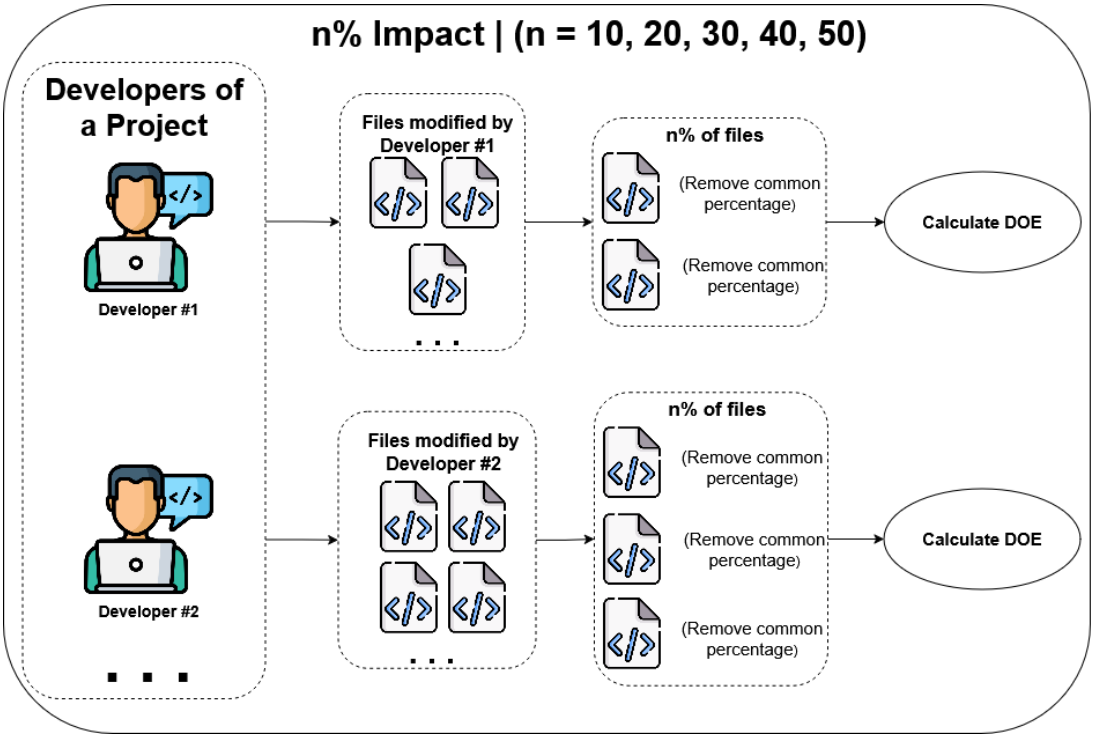


Figure 16 – Overview of the approach used to simulate the impact of GenAI code copying across different usage scenarios.

6.1.6 Survey Design and Application

With the development history, percentage of copied code, and developer expertise information in hand, we proceeded to the final step of our methodology before analyzing the results: surveying the authors of the shared links. This survey aimed to gather open-source developers' perspectives on the topics discussed in this chapter and to gain insights into how GenAI-generated code is integrated into their projects.

Given the scope of the study, all communication with participants was conducted via email, following the same approach used in the knowledge models study described in Chapter 4. Email remains a practical and cost-effective option among the various survey methods available in software engineering, despite its known limitation of low response rates (PUNTER et al., 2003; GHAZI et al., 2018).

We selected participants from the remaining 1,672 file–shared link pairs obtained through the filtering process described in Section 6.1.4. This ensured that all selected developers had integrated at least some GenAI-generated code into their projects. After further filtering for unique developers and valid email addresses (excluding *noreply* addresses), a final set of 653 developers was selected for participation. Using the Gmail API, we sent personalized survey emails to these 653 developers.

The survey consisted of six multiple-choice questions covering topics such as developer profiles, GenAI usage patterns, code integration strategies, and familiarity with

files containing GenAI-generated code. Each question, along with its corresponding rationale, is presented below.

Question 1: What is your experience level in software development?

(A) Junior (up to 2 years of experience).

(B) Mid-level (between 2 and 5 years of experience).

(C) Senior (over 5 years of experience).

Rationale: This question aims to gather information about the seniority of participating developers to enable a stratified analysis based on experience. This allows us to investigate whether the integration of GenAI-generated code varies according to the developer's level of experience.

Question 2: How often do you use generative AI tools (e.g., ChatGPT, Copilot, Gemini) to generate code?

(A) Never.

(B) Rarely (less than once a week).

(C) Occasionally (once or twice a week).

(D) Frequently (almost every day).

(E) Always (every day)

Rationale: This question aims to determine how frequently developers use GenAI for code generation. By cross-referencing it with other questions, such as the one on developer seniority, we can identify which types of developers are more likely to rely on GenAI tools.

Question 3: Do you believe using generative AI for code generation affects your understanding of the integrated code?

(A) Positively – It helps me understand the code better.

(B) No significant impact – My understanding is unaffected.

(C) Mixed – It depends on the context or scenario.

(D) Negatively – It diminishes my understanding or learning of the code

(E) Unsure.

Rationale: This question aims to gather developers' perceptions of the impact of GenAI on code comprehension. We can assess practitioners' opinions on whether AI-generated code facilitates or hinders their understanding by analyzing the responses.

Question 4: How do you typically integrate code generated by generative AI into your projects?

(A) I integrate the code with minimal scrutiny, ensuring it works without a detailed review.

(B) I integrate the code with a general understanding of its functionality (e.g., inputs and outputs)

(C) I integrate the code after a detailed review, understanding its implementation thoroughly.

(D) I use the code as inspiration or a reference but rarely incorporate it directly

(E) Other: [Please specify]

Rationale: This question aims to gather information on how survey participants typically integrate code generated by ChatGPT. It seeks to determine the extent of developers' understanding of the code they incorporate into their projects.

Question 5: Have you encountered difficulties maintaining AI-generated code (e.g., ChatGPT, Copilot) previously integrated into your project, such as modifying, extending, or debugging it?

(A) Yes, frequently

(B) Yes, occasionally

(C) No, but I rarely integrate AI-generated code

(D) No, never

Rationale: This question explores the maintainability of AI-generated code. We seek to understand whether developers frequently encounter difficulties modifying AI-generated code, which may indicate a lack of comprehension.

Question 6: Based on your past contributions, how confident do you feel about maintaining the following file: *%FILE_GITHUB_LINK*

(A) I have no familiarity with this file and wouldn't be able to work on it

(B) I remember contributing to it, but I would need to re-familiarize myself before making changes.

(C) I understand the general purpose and logic, and could make small changes.

(D) I'm confident I can maintain this file effectively.

(E) I'm the go-to expert on this file and know it in depth.

Rationale: With this final question, we aim to capture the level of expertise the developer had in the file version that both added the ChatGPT link and integrated the generated code. This allows us to investigate whether there is a relationship between the extent of code copying and the developer's perception of the code's maintainability. To facilitate this, the variable *%FILE_GITHUB_LINK* for each developer was replaced with a URL pointing to the specific version of the file, following the pattern: <https://github.com/%REPOSITORY_FULL_NAME/blob/%COMMIT_HASH/%FILE_PATH>, where *%REPOSITORY_FULL_NAME* is the full GitHub repository name, *%COMMIT_HASH* is the full commit hash, and *%FILE_PATH* is the file's path in the repository. The response options were generally aligned with the intent of the survey questions presented in Chapter 4.

The emails were sent in April 2025. We received 35 responses, representing 5% of

the 653 emails sent. This response rate is considered low, consistent with the one obtained in the survey conducted in Chapter 4. Despite the low participation, this level of engagement is common in email-based surveys in software engineering research and is sufficient to support exploratory insights.

To analyze the survey responses, we conducted a descriptive statistical analysis of each question, calculating the percentage of participants who selected each response option. To explore potential relationships between responses across questions, we applied the *Chi-Squared test* for independence to all pairs of categorical questions (PANDIS, 2016). For questions with ordinal response options, specifically Questions 1, 2, and 6, we employed *Spearman's* ρ to identify potential monotonic correlations. In the Spearman correlation analysis, we also considered the percentage of code copied by the developer in the corresponding file version, as referenced in Question 6.

Of the 35 survey respondents, only 16 provided additional comments that expanded on their answers and offered deeper insights into the themes addressed in the survey. We analyzed these open-ended responses using the *Grounded Theory Coding* method (STOL; RALPH; FITZGERALD, 2016), in which qualitative data is interpreted through a systematic process of assigning codes to text segments. These codes are then grouped into higher-level categories to identify themes and build an understanding of the participants' perspectives.

First, we excluded from the analysis any parts of the comments that lacked analytical value, such as greetings, expressions of thanks, and farewells. Second, in the labeling process, we segmented the comments into individual sentences, analyzed each one independently, and assigned labels that summarized the perspective conveyed in that specific sentence. Finally, we grouped these labels into broader categories that emerged from the data. This categorization enabled a descriptive analysis to characterize the participants' overall views on the topics addressed in the survey.

6.2 Results

Following the study design outlined earlier, we present the results in three parts. First, we provide a statistical analysis characterizing the conversations with ChatGPT and the corresponding code integrations. Second, we assess the impact of GenAI usage on developer expertise values, using the Degree of Expertise (DOE) model, and subsequently evaluate its influence on the Truck Factor metric. Finally, we present the results of the survey conducted with developers.

Table 13 – Percentage of copied code by programming language.

Language	Percentage	Num
C	59%	53
C#	51%	59
Shell	44%	69
Java	42%	175
Python	41%	612
C++	38%	186
JavaScript	34%	373
TypeScript	33%	109
PHP	32%	35

6.2.1 Code Integration Statistics

First, to characterize the conversations in the shared links, we briefly analyzed the number of *turns* involved in the proposed solutions. A *turn* consists of a user prompt followed by a ChatGPT response, following the same terminology used by Hao et al. (2024), for example. The distribution of the number of turns in the shared solutions has a first quartile (Q1) of 4, a median (Q2) of 8, a third quartile (Q3) of 16, and a mean of 14.29.

Second, we analyzed the distribution of the percentage of copied code. The distribution has a mean of 39%, with the first quartile (Q1) at 14%, the median (Q2) at 31%, and the third quartile (Q3) at 66%. Table 13 presents a breakdown of the mean copy percentage and the number of file–shared link pairs, by programming language. As there is only one Ruby file in the dataset, we excluded it from this analysis and the table.

As described in Section 6.1.5, the impact analysis requires a code copy rate to simulate the attribution of authorship to GenAI. For this purpose, in the following sections, we adopt the mean of the code copy rate distribution of 39% as the *uniform code copy rate*.

6.2.2 Impact on Degree of Expertise

In this initial analysis, we focus exclusively on the 50% impact scenario, as described in Section 6.1.5. This choice is justified by our interest in the overall average impact on *DOE* values rather than individual cases explored in the Truck Factor analysis. Table 14 presents the distribution of *DOE* values across developer-file pairs affected by GenAI usage, both with and without code copying. The third column (**Difference**) shows the distribution of the differences between the *DOE* values in these two conditions.

The project-level analysis also did not reveal any significant differences. We calculated the mean difference in *DOE* values for each project with and without copied code. The standard deviation of these means was low (0.0017), indicating that the values were tightly clustered around the overall mean of 0.125. Similarly, when grouping projects

Table 14 – Quartile distribution of original DOE values, DOE values affected by code copying, and their differences.

	Original DOE	Copy Affected DOE	Difference
Q1	2.248	2.118	0.113
Q2	2.793	2.666	0.118
Q3	3.397	3.277	0.160
Mean	2.842	2.716	0.126

by programming language, the mean differences also showed no notable variation.

We also segmented the analysis by comparing the differences in *DOE* values between core and peripheral developers. Core developers, defined here as those identified in the Truck Factor, experienced smaller losses, with an average reduction of 0.124. In contrast, peripheral developers showed a slightly higher average loss of 0.126. A *Wilcoxon Signed-Rank Test*, a non-parametric test used to assess whether the median difference between paired observations differs from zero (REY; NEUHÄUSER, 2011), comparing the *DOE* differences between these two groups yielded a statistically significant result ($p < 0.005$), suggesting a systematic disparity in the impact.

We also applied the Wilcoxon Signed-Rank Test to the distribution of *DOE* variations within each project, finding a statistically significant difference from zero in all cases ($p < 0.005$). This suggests that, despite the small magnitude of individual changes, there is a consistent and systematic effect across projects. The following sections explore how these differences affect a higher-level metric.

6.2.3 Impact on Truck Factor

First, we present a Truck Factor analysis of the selected projects. The distribution of Truck Factor values shows a mean of 108.75 and a median of 17. Due to the presence of outliers, such as *torvalds/linux* and *ohmyzsh/ohmyzsh*, the median offers a more representative measure of central tendency. The original (unimpacted) Truck Factor values are listed in the **TF** column of Table 15. To evaluate the impact of GenAI usage on these values, we consider two main aspects: (1) whether the Truck Factor value itself changes, and (2) whether the value remains stable but the developer ranking is affected.

We computed 120 Truck Factor values, covering 24 projects under 5 scenarios. The results are shown in the **TF_*N*** columns, where *N* indicates the proportion of impacted files, as illustrated in Figure 16. Of these, 87 values (73%) changed, impacting 20 of the 24 projects. Among the 87 changes, 86 showed a reduction in the Truck Factor, with a median decrease of 2 developers. Only one case showed an increase, by a single developer.

Additionally, 85 Truck Factors (71%) exhibited differences in their ranking order across 21 projects. The distribution of τ (tau) values, representing ranking similarity, has a

Table 15 – Truck Factor values of the target repositories across different impact scenarios.

Repository	TF	TF_10	TF_20	TF_30	TF_40	TF_50
discourse/discourse	23	22	22	21	21	21
electron/electron	16	15	15	15	15	15
facebook/react	6	6	7	6	6	6
facebook/react-native	68	66	66	65	64	60
freeCodeCamp/ freeCodeCamp	15	14	13	13	13	13
godotengine/godot	58	56	55	56	52	52
huginn/huginn	7	6	7	6	6	6
jellyfin/jellyfin	13	13	12	11	11	11
laravel/framework	212	189	188	189	181	173
mastodon/mastodon	5	5	5	4	4	4
microsoft/PowerToys	21	20	20	19	19	18
microsoft/terminal	6	6	6	6	6	6
microsoft/vscode	18	17	17	17	16	16
netdata/netdata	3	3	3	3	3	3
ohmyzsh/ohmyzsh	845	632	615	649	630	641
PowerShell/ PowerShell	13	13	12	12	12	12
rails/rails	455	399	400	391	380	374
redis/redis	33	31	31	30	30	30
Significant-Gravitas/ AutoGPT	6	6	6	6	6	6
tensorflow/tensorflow	149	146	146	144	144	144
torvalds/linux	604	593	589	580	574	564
twbs/bootstrap	22	18	19	17	17	19
vercel/next.js	10	10	10	10	10	9
vuejs/vue	2	2	2	2	2	2

mean of 0.43, with the first quartile (Q1) at 0.26, the median (Q2) at 0.37, and the third quartile (Q3) at 0.60. Among the 21 affected projects, 12 began to show differences, either in the value of the Truck Factor or in the order, in the first 10% impact scenario. Seven projects exhibited changes specifically at the 20% impact scenario, while one project showed changes only at the 30% scenario, and another at the 50% scenario.

We found a moderate positive correlation of $\rho = 0.41$ between the original Truck Factor size and the frequency of changes across scenarios. However, this does not mean that changes were exclusive to projects with higher Truck Factors. To investigate whether projects with lower Truck Factors were also affected, we selected those with a Truck Factor

less than or equal to 7, approximately the first quartile (6.75) of the original Truck Factor distribution. This subset represents 35 (29%) of the 120 calculated values. Even within this group, 17 (49%) exhibited changes in their ranking order or Truck Factor value.

> **Summary of RQ4:** In the simulated scenarios, developers experienced only a small absolute loss in Degree of Expertise (DOE), with an overall mean reduction of 0.125. Nevertheless, this decrease was sufficient to impact the Truck Factor computation, either by altering the Truck Factor value itself (observed in 73% of the 120 tested cases) or by changing the ranking of developers who concentrate the most knowledge (in 71% of the scenarios). These effects were observed even in scenarios with relatively limited integration of GenAI-generated code and were present across both high and low Truck Factor projects.

6.2.4 Survey Results

We divided the presentation of results into two sections. The first focuses on the survey responses from a quantitative perspective, analyzing the selected options. The second provides a qualitative analysis of the comments left by the developers.

6.2.4.1 Quantitative Results

Regarding *Question 1*, which asked about the developers' experience levels, the majority identified as *Senior* developers (48.6%), followed by *Junior* developers (31.4%). In *Question 2*, which addressed the frequency of using GenAI tools for code generation, 40% reported using them *Frequently*, followed by 25.7% using *Always*, with 11.4% and 22.9% using *Rarely* and *Occasionally*, respectively. A summary of the responses to all survey questions is presented in Table 16.

For *Question 3*, which investigated whether the use of GenAI affects code comprehension, most participants were divided between option *C* — "*Mixed – it depends on the context or scenario*," selected by 45.7% of respondents, and option *A* (*Positively*), chosen by 40%. The remaining responses were more evenly distributed across the other alternatives. Only a few developers provided comments elaborating on the specific scenarios referred to in option *C*; these will be presented later.

For *Question 4*, which examined how developers integrate GenAI-generated code, the majority of respondents (42.9%) reported that they *review in detail the integrated code* (option *C*). The second most selected option was *B* (*general understanding*), chosen by 25.7% of participants. Additionally, 17.1% indicated they use the GenAI-generated code as *inspiration* (option *D*), while 14.3% of participants selected option *E*.

Responses to Question 5, regarding the maintainability of GenAI-integrated code,

Table 16 – Summary of Developer Survey Responses.

Question and Theme	Response Options	Percentage
#1 - Developer experience	(A) Junior	31.4%
	(B) Mid-level	20.0%
	(C) Senior	48.6%
#2 - Frequency of use of GenAI tool	(B) Rarely	11.4%
	(C) Occasionally	22.9%
	(D) Frequently	40%
	(E) Always	25.7%
#3 - Impacto da GenAI on code understanding	(A) Positively	40%
	(B) No significant impact	2.9%
	(C) Mixed	45.7%
	(D) Negatively	8.6%
	(E) Unsure	2.9%
#4 - GenAI Code Integration	(B) General understanding	25.7%
	(C) Detailed review	42.9%
	(D) Code as inspiration	17.1%
	(E) Other	14.3%
#5 - Difficulty in maintaining GenAI code	(A) Yes, frequently	8.6%
	(B) Yes, occasionally	62.9%
	(C) No, but I rarely integrate AI-generated code	11.4%
	(D) No, never	17.1%
#6 - Confidence in maintaining file with GenAI code	(B) Need to re-familiarize	5.7%
	(C) Understand the general purpose	5.7%
	(D) Maintain effectively	34.3%
	(E) Go-to expert	54.3%

were concentrated on option *B* (62.9%), suggesting that most developers *occasionally* encounter difficulties in maintaining such code. Finally, Question 6 assessed whether developers could maintain the specific file versions in which GenAI-generated code was integrated. Most participants indicated no issues, with the majority selecting options *D* (34.3%) and *E* (54.3%).

The *Chi-Squared test* revealed three statistically significant associations ($p < 0.05$): between Questions 1 and 5, Questions 2 and 3, and Questions 2 and 6. Additionally, the association between Questions 2 and 6 was the only one to exhibit a statistically significant linear correlation according to *Spearman's* ρ. These results suggest three key relationships: (i) a developer's experience level is associated with the frequency of reported difficulties when maintaining GenAI-generated code (Q1 × Q5); (ii) the frequency of GenAI usage correlates with perceived impacts on code comprehension (Q2 × Q3); and (iii) frequency of usage is also related to the developer's confidence in maintaining the specific file that contains GenAI-generated code (Q2 × Q6).

We constructed proportion tables for the corresponding question pairs to further

explore these associations, which are visualized as stacked bar charts in Figures 17, 18, and 19. In Figure 17, crossing Questions 1 and 5, 81.8% of *Junior* developers (option *A*) reported experiencing *occasional* difficulties (option *B*). *Mid-level* developers (option *B*) were evenly split between *occasional difficulties* and *no, but I rarely integrate GenAI code* (option *C*). Among *Senior* developers (option *C*), 52% also reported occasional difficulties, while 29% stated they *never* encountered difficulties (option *D*), with the remaining responses distributed across the other options.

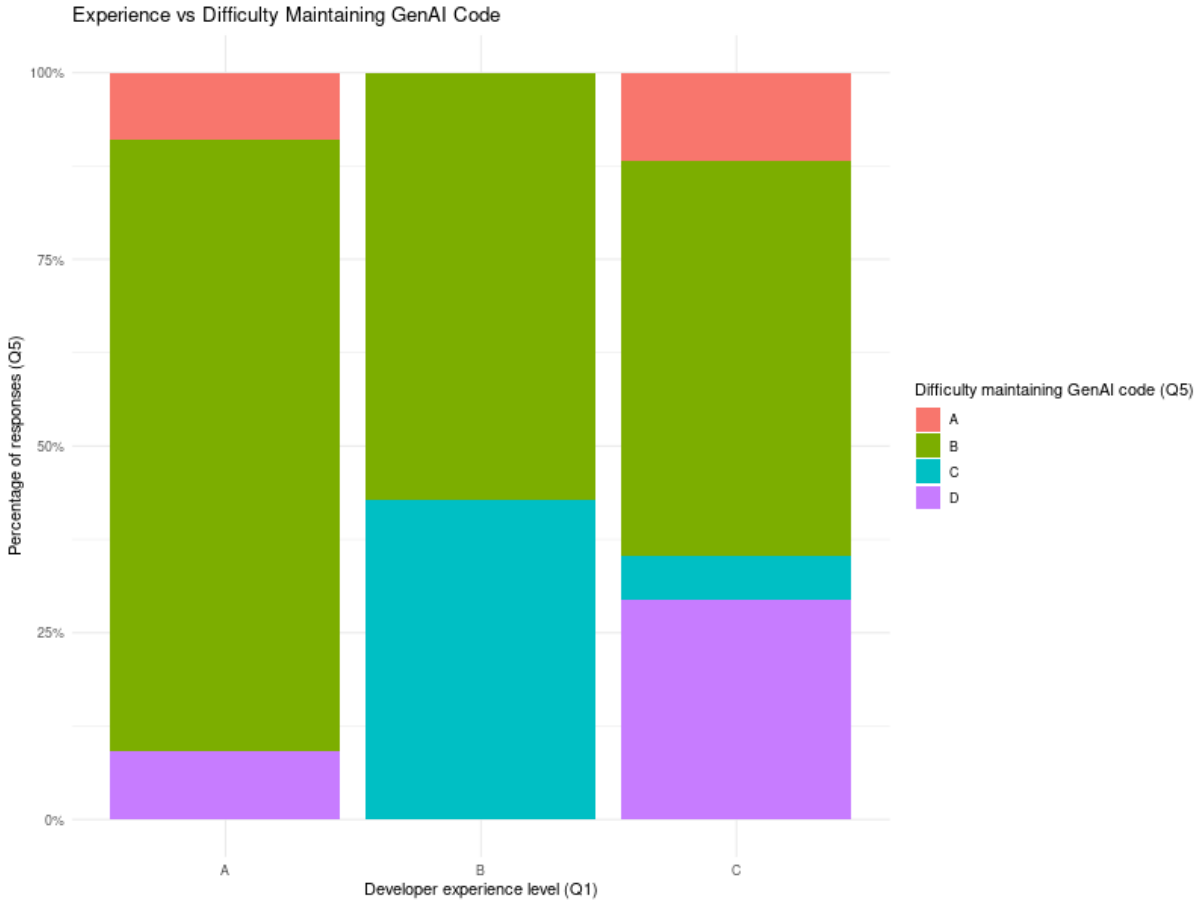


Figure 17 – Distribution of perceived difficulty in maintaining GenAI-generated code (Q5) across developer experience levels (Q1).

The proportion table for Questions 2 and 3, visualized in Figure 18, shows that 87.5% of participants who reported using GenAI *occasionally* (option *C*) selected a *mixed* impact on comprehension (option *C*), with the remaining respondents reporting a positive perception. Among *frequent* users, half also indicated a *mixed* perception, while 35.7% reported a *positive* impact and 14.3% a *negative* one. Respondents who reported *always* using GenAI presented a more polarized view: 77.8% stated that GenAI usage *helps them understand the code better*, while 11.1% reported a *negative* impact and 11.1% a *mixed* one.

Finally, the proportion table for Questions 2 and 6, shown in Figure 19, indicates that 50% of developers who use GenAI *occasionally* reported low familiarity with the file,

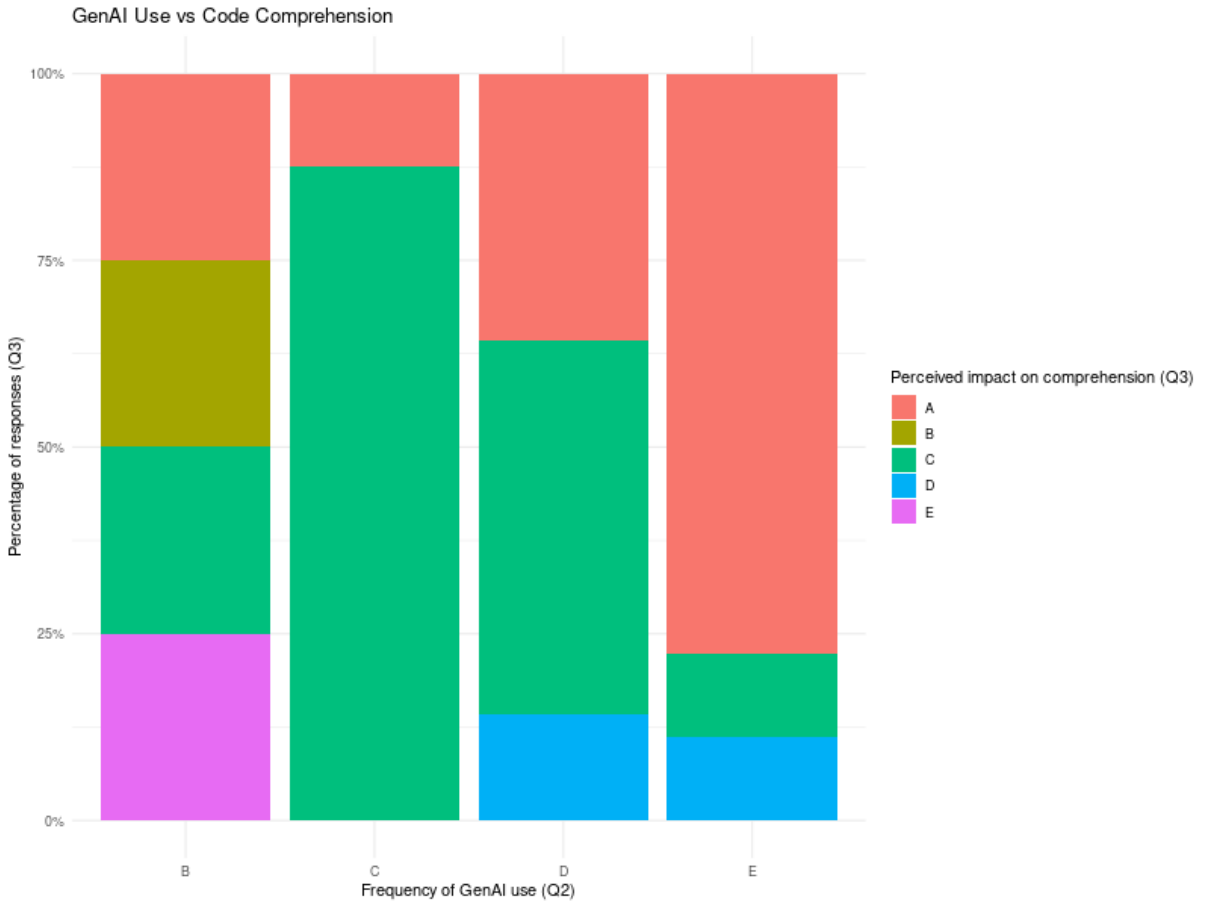


Figure 18 – Distribution of perceived impact on code comprehension (Q3) across different frequencies of GenAI usage for code generation (Q2).

indicating a need to re-familiarize themselves with it. Only 25% of these participants expressed confidence in maintaining the file (options *D* and *E*). As the frequency of GenAI usage increases, so does the developer's confidence: more than half of the *frequent* users (57.1%) and 88.9% of the *always* users identified themselves as the *go-to expert* for the file. This trend is further supported by a moderate positive Spearman correlation, with $\rho = 0.513$.

> **Summary of RQ5.1:** Most developers (45% of respondents) expressed mixed opinions regarding the impact of GenAI on code expertise, with many reporting occasional difficulties in maintaining AI-generated code. These challenges were particularly prevalent among junior developers, 81% of whom reported such difficulties. On the other hand, developers who used GenAI more frequently tended to view its impact more positively, and senior developers reported greater confidence in maintaining code produced with its assistance.

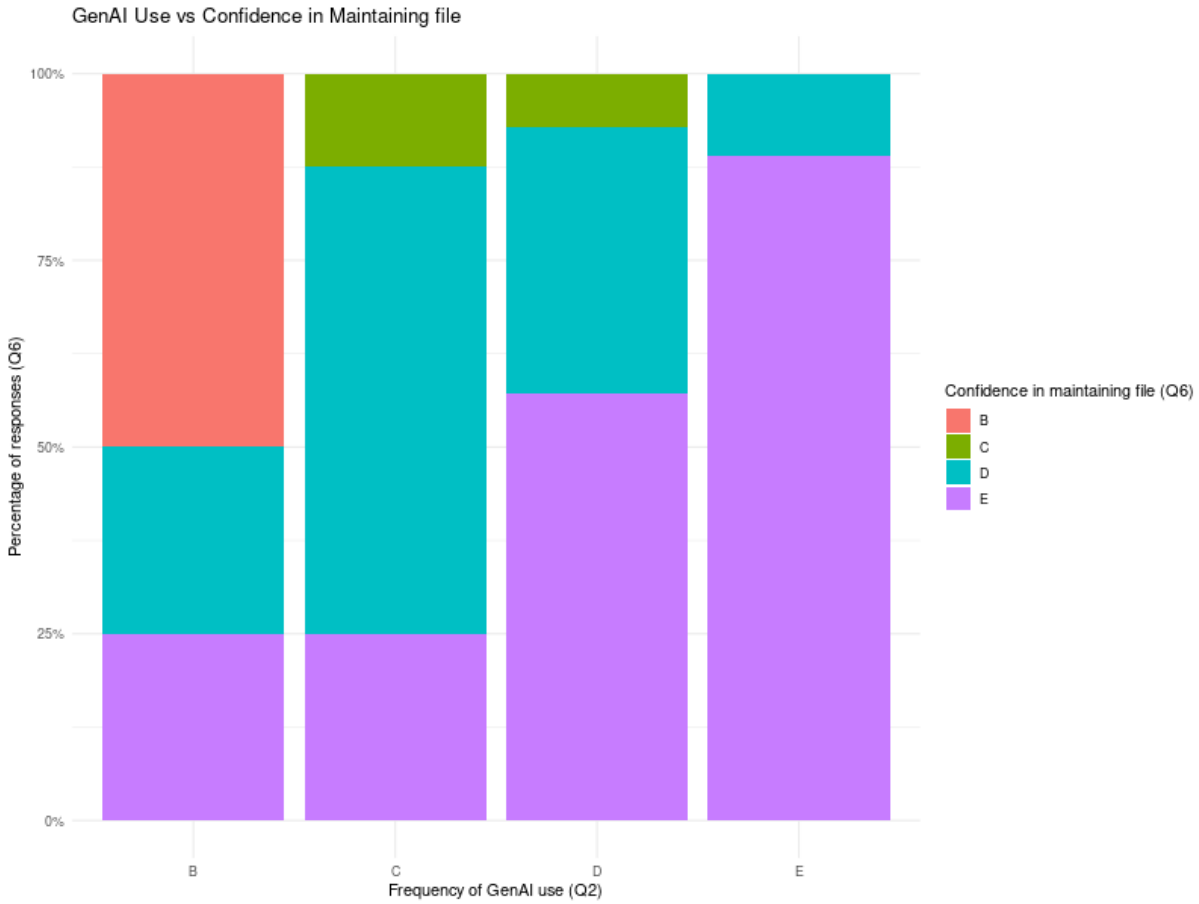


Figure 19 – Confidence in maintaining the given file (Q6) as reported by participants with varying levels of GenAI usage frequency (Q2).

6.2.4.2 Qualitative Results

From the responses of 16 participants, we extracted 35 individual statements for analysis. During the coding process, we identified 28 distinct codes, which were grouped into 5 overarching categories. Table 17 presents the statements, their corresponding codes, and categories. When a statement is associated with more than one code or category, they are listed separated by semicolons. The five categories that emerged from the analysis are:

1. **METHOD OF INTEGRATING GENAI-GENERATED CODE** – Comments describing how GenAI-generated code is typically integrated into projects.
2. **DIFFICULTIES AND INEFFICIENCY** – Statements expressing challenges or inefficiencies related to the use of GenAI tools.
3. **CONDITIONS FOR EFFECTIVE USE OF GENAI** – Comments identifying specific conditions under which GenAI tools are effectively used for code generation.

Table 17 – Comments excerpts, codes and categories identified.

Excerpts from comments	Codes	Categories
sometimes i'm only interested in solving the problem rather than understanding why	OCCASIONAL USE WITHOUT CONCERN FOR UNDERSTANDING	METHOD OF INTEGRATING GENAI-GENERATED CODE
if the code is for one-off scripts, i skip validation	UNREVIEWED USE IN TRIVIAL OR NON-CRITICAL TASKS	METHOD OF INTEGRATING GENAI-GENERATED CODE
As a teacher's assistant and tutor for mathematics, psychology, and computer science, AI has had a negative effect on the understanding of the materials for students.	NEGATIVE PERCEPTION OF GENAI'S IMPACT ON UNDERSTANDING	IMPACT OF GENAI USE ON CODE UNDERSTANDING
They (students) over rely on AI systems to do their code for them and do not understand the code.	CONCERN ABOUT OVER-RELIANCE	IMPACT OF GENAI USE ON CODE UNDERSTANDING
AI can be a good tool, but computer science students now are using it instead of learning, and as a result do not understand anything that they should.	OVERALL POSITIVE PERCEPTION; NEGATIVE PERCEPTION OF GENAI'S IMPACT ON UNDERSTANDING	PERCEIVED BENEFITS; IMPACT OF GENAI USE ON CODE UNDERSTANDING;
on the whole, I think I do overall get better at using software engineering by using LLMs.	OVERALL POSITIVE PERCEPTION	PERCEIVED BENEFITS
However, this option is absolutely true: "Mixed – It depends on the context or scenario" - there are times, for example, when I am lazy or don't have time, I will use generated code without even looking at it, and this feels atrophic.	OCCASIONAL USE WITHOUT CONCERN FOR UNDERSTANDING; NEGATIVE PERCEPTION OF GENAI'S IMPACT ON UNDERSTANDING	METHOD OF INTEGRATING GENAI-GENERATED CODE; IMPACT OF GENAI USE ON CODE UNDERSTANDING
I found trying to understand AI code sometimes hard because they make something like a pseudo-code where it does looks like actual code but most of the time I found it it to be absolutely useless	PERCEPTION OF USELESSNESS IN SOME CASES	DIFFICULTIES AND INEFFICIENCY
For me, AI can understand if it's single file doing single thing, but in most big projects I did it's useless.	EFFICIENCY LIMITED TO SMALL/SIMPLE COMPONENTS; PERCEPTION OF USELESSNESS IN MANY CASES	CONDITIONS FOR EFFECTIVE USE OF GENAI; DIFFICULTIES AND INEFFICIENCY
Sometimes I even find it faster to implement it myself.	PERCEPTION OF USELESSNESS IN SOME CASES	DIFFICULTIES AND INEFFICIENCY
when I usually ask question or use Agent/Edit mode, most of the time I just throw their code away because they don't understand a thing of what my codebase does.	PERCEPTION OF USELESSNESS IN MOST CASES	DIFFICULTIES AND INEFFICIENCY
Maybe if I still use Python, JavaScript/TypeScript like what most of the AI tools learn it I would be fine.	PERCEIVED USEFULNESS DEPENDS ON PROGRAMMING LANGUAGE	CONDITIONS FOR EFFECTIVE USE OF GENAI
But recently since I started using Rust I found that just getting general idea is better.	PERCEIVED USEFULNESS DEPENDS ON PROGRAMMING LANGUAGE; USE AS A STARTING POINT/REFERENCE	CONDITIONS FOR EFFECTIVE USE OF GENAI; METHOD OF INTEGRATING GENAI-GENERATED CODE
I take it as a starting point and then clean it up and fix/improve it.	USE AS A STARTING POINT/REFERENCE	METHOD OF INTEGRATING GENAI-GENERATED CODE
Yes, occasionally (especially with edge cases or non-standard patterns); I've noticed that AI-generated code often lacks context-specific documentation or modularity, which can hinder long-term maintainability.	NEGATIVE PERCEPTION OF GENERATED CODE QUALITY	DIFFICULTIES AND INEFFICIENCY
While it's a useful productivity tool, developers must prioritize deep code review to ensure integration aligns with project architecture and best practices.	POSITIVE PERCEPTION ON PRODUCTIVITY; NEED FOR THOROUGH CODE REVIEW	PERCEIVED BENEFITS; METHOD OF INTEGRATING GENAI-GENERATED CODE
Besides code generation, there are a lot of other very helpful applications that help software engineers like explaining code and error messages, reviews, learning, brain storming	POSITIVE PERCEPTION ON VARIOUS USE CASES	PERCEIVED BENEFITS
Auto generating unit tests is - on the other hand in my opinion - harmful,even if it sounds reasonable for inexperienced engineers or managers.	NEGATIVE PERCEPTION OF USING GENAI FOR UNIT TESTS	DIFFICULTIES AND INEFFICIENCY
I strongly believe that it won't replace an engineer soon.	SKEPTICISM ABOUT JOB REPLACEMENT BY GENAI	DIFFICULTIES AND INEFFICIENCY
I know that there are some tasks where an LLM will make me gain some time, generally simple or trivial tasks.	POSITIVE PERCEPTION ON PRODUCTIVITY, ESPECIALLY FOR SIMPLE TASKS	PERCEIVED BENEFITS; CONDITIONS FOR EFFECTIVE USE OF GENAI
If the task is too complex for an LLM, then I generally give up and do it myself from scratch, maybe sometimes using the LLM's attempt as a base.	PERCEPTION OF USELESSNESS IN SOME CASES; USE AS A STARTING POINT/REFERENCE	DIFFICULTIES AND INEFFICIENCY; METHOD OF INTEGRATING GENAI-GENERATED CODE
But it's often simpler to just work through the hard problems myself and use the LLM to generate boilerplate and make simpler changes.	PERCEPTION OF USELESSNESS IN SOME CASES; USE AS A STARTING POINT/REFERENCE	DIFFICULTIES AND INEFFICIENCY; METHOD OF INTEGRATING GENAI-GENERATED CODE
Frequently (almost every day), It is fine for boring work and low-risk summaries	POSITIVE PERCEPTION FOR BORING OR LOW-RISK TASKS	CONDITIONS FOR EFFECTIVE USE OF GENAI
Negatively – It diminishes my understanding or learning of the code, once I paste it, I stop thinking about it unless it breaks.	NEGATIVE PERCEPTION OF GENAI'S IMPACT ON UNDERSTANDING	IMPACT OF GENAI USE ON CODE UNDERSTANDING
Positively – It helps me understand the code better If the prompt is carefully set, I will enforce code readability	POSITIVE PERCEPTION OF GENAI'S IMPACT ON UNDERSTANDING DEPENDING ON PROMPT QUALITY	IMPACT OF GENAI USE ON CODE UNDERSTANDING; CONDITIONS FOR EFFECTIVE USE OF GENAI; PERCEIVED BENEFITS
I integrate the code with a general understanding of its functionality (e.g., inputs and outputs) - For example, the backend will have a test case that the AI needs to pass, and the frontend will carefully test the behavior through interaction.	FOCUS ONLY ON CODE FUNCTIONALITY	METHOD OF INTEGRATING GENAI-GENERATED CODE
Sometimes the AI will somehow modify the file you don't want.	CONCERN ABOUT INAPPROPRIATE USAGE BEHAVIOR	DIFFICULTIES AND INEFFICIENCY
Other, sometimes in a rush I just plug it in directly, making sure inputs/outputs are correct and just get it over with without learning or understanding it, other times I will make a much more thorough effort.	OCCASIONAL USE WITHOUT CONCERN FOR UNDERSTANDING	METHOD OF INTEGRATING GENAI-GENERATED CODE; IMPACT OF GENAI USE ON CODE UNDERSTANDING
Yes, occasionally, especially if someone else on my team used it, it's better when I used it myself since i know the context better, but reading unfamiliar AI code is more difficult as sometimes it contains extraneous logic	DIFFICULTY MAINTAINING GENAI CODE INTEGRATED BY ANOTHER DEVELOPER	IMPACT OF GENAI USE ON CODE UNDERSTANDING
3, depends on what I'm working on; Depends on what I'm working on. If it's sensitive/nuanced, then 4. If it's a throwaway tool, then 1 or 2. If it's a minor incremental change, then 3. If I'm optimizing for maintainability, I treat it differently than if it's a throwaway single-use tool	UNREVIEWED USE IN TRIVIAL OR NON-CRITICAL TASKS; CAUTIOUS USE OF GENAI FOR IMPORTANT/COMPLEX TASKS	METHOD OF INTEGRATING GENAI-GENERATED CODE; CONDITIONS FOR EFFECTIVE USE OF GENAI
but I only ask for smaller components, so that they are easily testable. If they do not arrive small, I refactor.	USE LIMITED TO SMALL/SIMPLE COMPONENTS; OCCASIONAL REFACTORING OF GENAI-GENERATED CODE	METHOD OF INTEGRATING GENAI-GENERATED CODE; CONDITIONS FOR EFFECTIVE USE OF GENAI
If the code is for a personal or unimportant project I do not apply much scrutiny, but if the project is important I try my best to thoroughly examine and understand the generated code.	UNREVIEWED USE IN TRIVIAL OR NON-CRITICAL TASKS; CAUTIOUS USE OF GENAI FOR IMPORTANT/COMPLEX TASKS	METHOD OF INTEGRATING GENAI-GENERATED CODE; CONDITIONS FOR EFFECTIVE USE OF GENAI
I discuss architecture and use code of common functionality/patterns.	USE FOR COMMON CODE GENERATION	METHOD OF INTEGRATING GENAI-GENERATED CODE;
Always understand code before use.	CONCERN ABOUT CODE UNDERSTANDING	METHOD OF INTEGRATING GENAI-GENERATED CODE
Always be critical of the ai because it doesn't know the entire scope of the project and it's systems (I work in game dev)	CONCERN ABOUT CODE UNDERSTANDING	METHOD OF INTEGRATING GENAI-GENERATED CODE; DIFFICULTIES AND INEFFICIENCY

4. **IMPACT OF GENAI USE ON CODE UNDERSTANDING** – Statements addressing how, or to what extent, using GenAI for code generation affects code comprehension.

5. **PERCEIVED BENEFITS** – Comments highlighting benefits associated with the use of GenAI tools.

Of the 35 analyzed sentences, 16 were categorized under *method of integrating genai-generated code*. Among them, 7 described *occasional use without concern*, emphasizing *unreviewed* integration driven primarily by *functionality*, often justified by *haste*, *laziness*, the *triviality* of the task, or the *perceived lack of importance* of the code, with one comment in this group also noted a perceived *atrophy of skills* resulting from excessive reliance on GenAI tools. Six other comments reflected a *cautious approach*, involving *code review* and *efforts to understand* the generated output, particularly in the context of *complex* or *important* tasks. One participant in this group also mentioned an overall *positive perception on productivity* of GenAI tools. Finally, four sentences indicated the use of GenAI as a *starting point* or source of *inspiration*, with three of these expressing a *perception of uselessness* of the tool in certain scenarios.

In the category *difficulties and inefficiency*, 11 sentences were identified. Among them, 6 expressed a perception of *uselessness*, at varying levels, particularly in contexts such as *large projects* and *hard problems*, or when dealing with specific programming languages like *Rust*. Additionally, 2 comments conveyed a negative perception of the generated code, one criticizing the quality of *documentation*, and another regarding the generation of *unit tests*. Other remarks included concerns about GenAI's potential for *unauthorized modifications* in the code, *skepticism about job replacement*, and distrust in the overall *code quality* produced, especially when the tools lack full understanding of the broader codebase.

We identified 9 sentences that describe the *conditions for effective use of GenAI*. The majority (6 sentences) indicated that there is no issue with using GenAI for *simple*, *small components*, *trivial*, or *low-risk* tasks, as well as for repetitive and *boring* work. Two other comments stated that effectiveness depends on the programming language, being more favorable for widely used languages such as *Python*, *JavaScript*, or *TypeScript*. One additional response emphasized the importance of a *quality prompt* as a key condition for effective use.

In the category *impact of GenAI use on code understanding*, which is most directly aligned with the focus of our study, we identified 8 sentences, 7 expressing negative perceptions and 1 positive. Among the 7 negative responses, concerns varied, including the risk of developing *over-reliance*, the potential *atrophy* of coding skills, and the reduction in understanding due to behaviors such as *copying and pasting* generated code without reflection. The only positive perception mentioned that GenAI can improve code *readability*, depending on the quality of the prompt.

Lastly, in the category *perceived benefits*, we identified 6 sentences. Two of them are general and do not specify particular advantages, simply stating that GenAI can be beneficial overall and can help engineers become *better*. Two others highlight gains in *productivity*, one lists benefits such as explaining code and error messages, and another

mentions that GenAI can improve code *readability*.

> **Summary of RQ5.2:** Most sentences about GenAI integration described *unreviewed* use, primarily driven by *functionality* and justified by *task triviality* or *urgency*. In the *difficulties* category, the dominant perception was one of *uselessness* in complex scenarios. Regarding *effective use*, most sentences highlighted the suitability of GenAI for *simple* or *low-risk tasks*. The impact on *code understanding* was largely negative, with concerns about *skill atrophy* and *reduced comprehension*. *Productivity* and *readability* were the most commonly cited benefits.

6.3 Discussion

6.3.1 GenAI Code Integration

Firstly, regarding the number of turns in the mined conversations, our results differ from those reported by Hao et al. (2024), where most conversations consisted of only a single turn. A key difference lies in the datasets: our study focuses specifically on code generation that was at least partially integrated into real projects. A more comparable reference is the work by Jin et al. (2024), who reported an average of 10.4 turns per code generation conversation. In contrast, our dataset shows a considerably higher average of 14.291 turns. However, the 100th percentile of our distribution is 326, indicating the presence of outliers. In this context, the median value of 8 is a more representative central tendency and aligns more closely with results from other studies.

Based on the distribution of copied code percentages, with an average of 39%, we observe a scenario that suggests co-authorship between developers and ChatGPT, although the majority of the code is still authored by developers. It is important to note that our findings are limited to instances indicating the use of GenAI in source code specifically from ChatGPT. Therefore, this likely reflects a case of underreporting, as ChatGPT's use for code generation is likely much more widespread across GitHub files.

Additionally, some tools are already fully integrated into Integrated Development Environments (IDEs), which streamlines both code generation and acceptance. Considering this growing adoption and tighter integration, we highlight that future research may uncover different percentages of code being accepted without modification.

The data on the percentage of copied code by programming language suggests that lower-level languages, such as *C* and *Shell*, exhibit the highest rates. In contrast, higher-level languages like *JavaScript* and *TypeScript* show lower percentages. This trend may be associated with the level of abstraction provided by each language, with higher-level languages tending to be more concise, potentially reducing the need for

directly copying code snippets.

6.3.2 Impact on Knowledge Model

Regarding the impact analysis, the difference in developers' Degree of Expertise (DOE) values within the files they contributed to was consistent but minor, as shown in Section 4.2. This outcome is expected, given that the simulation removed 39% of only one variable in the DOE model, *number of lines added*, which, as discussed in Section 4.2, is not the most influential factor in assessing code knowledge. Among the stratifications performed, none showed a notable difference, except that core developers appeared to lose slightly less knowledge and were less affected overall. Nonetheless, even these subtle changes impacted the Truck Factor calculations, affecting both higher and lower values, with a strong tendency toward a decrease in the Truck Factor. In such scenarios, the algorithm tends to indicate a greater concentration of knowledge among key developers.

Considering the *assumptions*, in the simulated scenarios, the team effectively experienced knowledge loss due to the use of GenAI for code generation. However, management may be misled into believing that knowledge is more concentrated than it actually is, which could unnecessarily alarm the team. Although this is arguably a better scenario than an increase in the Truck Factor, since the latter might convey a false sense of security, it still reflects a misleading interpretation of the team's actual resilience, as discussed in Section 5.3, Chapter 5. This finding highlights that the Truck Factor metric is sensitive to variations in developers' expertise, even under moderate usage conditions. As such, it *raises concerns about its reliability when GenAI-generated code is integrated under certain conditions*.

However, this issue extends beyond the Truck Factor. Authorship metrics are widely used across a variety of software engineering tasks. As previously discussed, measures such as commit counts and lines of code contribute to identifying suitable code reviewers (HANNEBAUER et al., 2016) and recommending developers to fix bugs (KHATUN; SAKIB, 2016). These metrics are also commonly employed to evaluate developer performance, both in academic research and in industry settings (BELLER et al., 2025). As GenAI tools become increasingly prevalent in code production, we argue that all direct applications of these authorship-based metrics should be revisited and carefully reassessed.

Furthermore, we argue that the very notion of developer expertise may be redefined in the coming years if the trend toward automated code generation persists. This shift could be further intensified in a *black box case* scenario of AI interaction, where engineers engage with generative agents solely through high-level specifications, and the resulting code is deployed without direct human inspection of the source code (ROBLES et al., 2024). In such a context, developer expertise may transition from traditional coding skills to higher-order competencies, emphasizing strategic decision-making, creative problem

solving, and system-level reasoning. This transformation aligns with predictions from recent studies that envision a future where software engineering becomes increasingly abstracted and centered on intent specification rather than implementation (RASHEED et al., 2024; SAUVOLA et al., 2024).

This shift in the focus of developer skills is also expected to impact both the training of software engineers and the qualifications valued by the job market. On the industry side, the integration of software development with GenAI tools makes it increasingly difficult to assess the actual technical capabilities of candidates, with some companies already prioritizing experience in effectively leveraging these tools (CHEN et al., 2024). As for education, curricula are likely to evolve, placing greater emphasis on high-level skills such as optimization, systems design, prompt engineering, and critical evaluation of AI-generated outputs (HAQUE, 2025).

6.3.3 Survey on Developers' Perspectives

In the related literature, surveys report mixed perceptions regarding the use of GenAI in software development and code comprehension, echoing the results of *Question 3* in our survey. Some studies emphasize gains in productivity and assistance with redundant or tedious tasks (ULFSNES et al., 2024; YILMAZ; YILMAZ, 2023), which can allow developers to focus on more meaningful activities (BANH; HOLLDACK; STROBEL, 2025), a point also reflected in some of the analyzed comments. Others suggest that GenAI tools support learning by helping users better understand software development concepts (WASEEM et al., 2023). Conversely, concerns about not fully understanding the generated and integrated code are also frequently mentioned by students (PRATHER et al., 2023), reinforcing the correlation we found between *Question 1* (experience level) and *Question 5* (frequency of difficulty in maintaining GenAI-generated code).

Most of our survey participants reported concerns regarding the need for a *detailed review* of code generated by GenAI. This concern is well-founded, as LLMs generally lack a deep understanding of the underlying logic of the code, which can lead to incorrect outputs (HAQUE, 2025). Even with the undeniable benefits, recent studies have provided evidence of *code quality erosion*, potentially compromising the *maintainability* of software systems (HARDING, 2025).

The second most common response to *Question 4* indicates that developers typically form only a *general understanding* before integrating GenAI-generated code. While this approach may be adequate for repetitive or trivial tasks, it carries the risk of *skill atrophy*, as pointed out in some additional comments. This risk is particularly critical for *junior* developers, who may fail to internalize essential computing concepts and instead accept the suggested implementation without deeper evaluation, a concern also highlighted in a recent review on *vibe coding* practices (RAY, 2025). When combined with

known risks of GenAI-related errors and *hallucinations* (AGARWAL et al., 2024), these findings reinforce the importance of maintaining a *human-in-the-loop* approach, ensuring that developers remain actively engaged rather than merely acting as passive recipients of automated solutions.

Finally, our results also showed a *positive correlation* between the *frequency of GenAI use* and both the *confidence in the generated code* and the *reduced difficulty in maintaining it*. This perception may be attributed to the accumulation of experience with these tools. Over time, developers tend to improve their skills in *prompt engineering*, enabling them to obtain more satisfactory outputs, an effect also observed in a study on the satisfaction of programming students with GenAI tools (ZVIEL-GIRSHIN, 2024).

6.4 Threats to Validity

Construct Validity. In simulating the impact of GenAI usage on the knowledge model and the Truck Factor, we relied on two assumptions outlined in Section 6.1.3, which may not necessarily hold in all analyzed cases. Regarding *Assumption 1*, the presence of accepted code from ChatGPT does not necessarily imply a direct copy-and-paste action. To adopt a conservative approach, we considered only exact matches between the added lines and those generated by ChatGPT, unlike related work such as Grewal et al. (2024), which employed approximate matching techniques based on *Levenshtein Distance*. Similarly, for *Assumption 2*, we do not assert that developers completely lose knowledge of code that was copied and pasted without thorough review; this simplification was adopted for exploratory purposes. Nonetheless, prior research lends support to this interpretation, as discussed in the rationale for the assumption.

Similar to the two surveys conducted in the previous chapters, this one is also subject to common threats associated with email-based surveys, such as misunderstandings, reluctance to respond, and overestimated self-assessments of comprehension. As with the previous surveys, we took steps to mitigate these issues by clearly explaining the purpose of the research, how the data was obtained, and how it would be handled.

Conclusion Validity. In the analyses reported in this chapter, we combined descriptive statistics with inferential tests to support our findings. We used Spearman's ρ to detect monotonic relationships, Kendall's τ to assess changes in Truck Factor rankings, and the χ^2 test to examine associations among categorical variables derived from survey responses. Given the risk of misinterpretation when results lack statistical significance, we limited our interpretations to statistically significant results ($p < 0.005$), thereby increasing the robustness and reliability of our conclusions.

External Validity. The investigations presented in this chapter also pose certain threats to the generalizability of the results. In the GenAI code integration analysis, we used integration data from files written in ten different programming languages, aiming to maximize variability within the dataset. However, in the impact simulation, we employed only one knowledge model among several available in the literature. As previously explained, this choice was guided by the granularity and variables used in the selected model, which align with the design of the impact analysis and correspond to the model proposed in this thesis. Nevertheless, this constitutes a limitation regarding external validity.

We also conducted simulations using large projects written in different languages and developed in diverse contexts. Still, we acknowledge that the wide range of possible team configurations and patterns of GenAI usage in real-world software development scenarios go beyond what can be captured in simulation, representing an additional threat to generalizability.

Similarly, the survey we conducted had a limited number of respondents, which is common in email-based surveys due to their typically low response rates. However, the variability among the responses provided by developers with different levels of experience and diverse perspectives helps to mitigate this limitation partially.

6.5 Conclusion

In this chapter, we presented a study on the impact of using Generative Artificial Intelligence (GenAI) for code generation on a developer expertise identification model and a Truck Factor algorithm. Based on statistics reflecting the percentage of code copied from ChatGPT, our simulations demonstrate that expertise identification models are sensitive to authorship loss. Although the quantitative reduction in expertise values is relatively small, applications such as the Truck Factor algorithm are nonetheless affected, even under scenarios of limited GenAI usage. These findings highlight a potential loss of reliability in such metrics as GenAI tools become increasingly widespread in software development and code generated by GenAI becomes more commonly integrated, depending on how these tools are used.

Additionally, we surveyed developers to explore their perceptions regarding the impact of using GenAI tools for code generation on code understanding. We performed both quantitative and qualitative analyses of the responses. Overall, most developers expressed a mixed perception, acknowledging both benefits and drawbacks. Junior developers reported greater difficulty in working with GenAI-generated code, whereas senior developers showed more confidence in maintaining such code. In the open-ended responses, some participants voiced concerns about over-reliance and reduced understandability, while others highlighted benefits such as increased productivity

and improved code readability.

This exploratory study opens several avenues for future research. As discussed, the impact of losing developer authorship to GenAI can be further investigated in a variety of software engineering activities. Future work could examine how the use of GenAI tools influences areas such as code review effectiveness, bug triage and resolution, developer productivity, and alternative expertise metrics. Additionally, expanding this analysis to different project types, team structures, and domains may offer broader insights into the generalizability of these effects.

7 Conclusion

This chapter concludes the thesis by revisiting each overarching question introduced in Chapter 1 and discussing our main contributions. We also outline the continuation of this research, providing a schedule of activities to be completed leading up to the presentation of the final version.

7.1 Contributions

This thesis explores the development of more accurate expertise identification models, their applications in a software development context, and the impact of generative artificial intelligence on these models. In this section, we summarize the contributions of this work by revisiting the overarching questions posed at the beginning.

Q1: Can we improve existing source code knowledge models?

In the study presented in Chapter 4, we demonstrated that more accurate source code knowledge models can be developed through a detailed analysis of variables. From a dataset of public and industrial projects, we identified variables most related to code knowledge. *First Authorship* and *Number of Lines Added* showed the highest positive correlations with knowledge, while *Recency of Modification* and *File Size* had the highest negative correlations. Using these variables, we evaluated a proposed linear model named *Degree of Expertise* and several *Machine Learning* models for identifying experts. We compared the performance of these models with other techniques, finding that three *Machine Learning* algorithms (*Random Forest*, *SVM*, and *Gradient Boosting Machines*) achieved the best *F-Score* and *Precision*.

Q2: Can we improve a knowledge concentration metric with new source code knowledge models?

In Chapter 5, we apply the models proposed in Chapter 4 (Q1) to a well-known Truck Factor algorithm and verify its performance both quantitatively and qualitatively. Avelino's Truck Factor Algorithm identified relevant missing developers in open-source projects, achieving an average *F-Score* of 74%. Additionally, six development leaders perceived it as more accurate in computing the Truck Factor of a private repository. We also demonstrate an implementation of this modified algorithm in a web application called Knowledge Islands, describing a usage scenario.

Q3: What is the impact of Generative AI tools for code generation on source code knowledge models?

In Chapter 6, we present an exploratory study that investigates the impact of using Generative AI (GenAI) tools for code generation on source code knowledge models. We began by collecting integration statistics from several open-source GitHub projects across different programming languages. Based on these insights, we simulated usage scenarios involving the integration of GenAI-generated code and assessed the resulting loss of authorship using the *Degree of Expertise* (DOE) model and the Truck Factor algorithm. While the changes in DOE values were relatively small, the Truck Factor results proved highly sensitive to this knowledge loss, revealing potential reliability issues in such models under certain assumptions. To complement the quantitative analysis, we conducted a survey with developers to gather their perceptions of GenAI's influence on code understanding. Most respondents expressed a mixed view, acknowledging both benefits and drawbacks. Junior developers reported greater difficulty in maintaining GenAI-generated code, whereas frequent GenAI users tended to view its impact more positively. Additionally, senior developers expressed greater confidence in maintaining code produced with GenAI support.

7.2 Publications

During the development of this work, four full papers and one extended abstract were produced, as listed in Table 18. Of these, four have been published. The first paper was presented at the *International Symposium on Empirical Software Engineering and Measurement* (ESEM) (CURY et al., 2022), and the second was published in the journal *Information and Software Technology* (IST) (CURY et al., 2024). Two additional papers were presented at the *Brazilian Symposium on Software Engineering* (SBES) (CURY; AVELINO, 2024; CURY; AVELINO, 2025). Furthermore, a summary of this thesis, in the form of a poster and extended abstract, was presented at the *LATAM School in Software Engineering*, as part of SBES.

Table 18 – Studies published and accepted as part of this thesis.

Publication	Type	Qualis	Status
Source code expert identification: Models and application. Information and Software Technology. 2024.	Journal	A1	Published
Identifying source code file experts. International Symposium on Empirical Software Engineering and Measurement. 2022.	Symposium	A2	Published
Knowledge Islands: Visualizing Developers Knowledge Concentration. Brazilian Symposium on Software Engineering (Tools Track). 2024.	Symposium	A3	Published
Source Code Expertise: Improving Knowledge Models and Assessing Generative AI Impact. Brazilian Symposium on Software Engineering (LATAM School in Software Engineering). 2024.	Symposium	-	Published
The Impact of Generative AI on Code Expertise Models: An Exploratory Study	Symposium	A3	Published

Bibliography

ABDI, H. The kendall rank correlation coefficient. **Encyclopedia of measurement and statistics**, Sage Thousand Oaks, CA, v. 2, p. 508–510, 2007.

AGARWAL, V.; PEI, Y.; ALAMIR, S.; LIU, X. Codemirage: Hallucinations in code generated by large language models. **arXiv preprint arXiv:2408.08333**, 2024.

AKTER, S. **Recommending expert developers using usage and implementation expertise**. Tese (Doutorado) — University of Lethbridge (Canada), 2021.

AL-JAMIMI, H. A.; AHMED, M. Machine learning-based software quality prediction models: state of the art. In: IEEE. **2013 International Conference on Information Science and Applications (ICISA)**. [S.l.], 2013. p. 1–4.

ALI, J.; KHAN, R.; AHMAD, N.; MAQSOOD, I. Random forests and decision trees. **International Journal of Computer Science Issues (IJCSI)**, International Journal of Computer Science Issues (IJCSI), v. 9, n. 5, p. 272, 2012.

ALKHALID, A.; LUNG, C.-H.; AJILA, S. Software architecture decomposition using adaptive k-nearest neighbor algorithm. In: IEEE. **2013 26th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)**. [S.l.], 2013. p. 1–4.

ALMARIMI, N.; OUNI, A.; CHOUCHEN, M.; MKAOUER, M. W. csdetector: an open source tool for community smells detection. In: **Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. [S.l.: s.n.], 2021. p. 1560–1564.

ANAGNOSTOPOULOS, C.-N. Chatgpt impacts in programming education: A recent literature overview that debates chatgtp responses. **arXiv preprint arXiv:2309.12348**, 2023.

ANVIK, J.; MURPHY, G. C. Determining implementation expertise from bug reports. In: IEEE. **Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)**. [S.l.], 2007. p. 2–2.

ASTHANA, S.; KUMAR, R.; BHAGWAN, R.; BIRD, C.; BANSAL, C.; MADDILA, C.; MEHTA, S.; ASHOK, B. Whodo: automating reviewer suggestions at scale. In: **Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. [S.l.: s.n.], 2019. p. 937–945.

AVELINO, G.; CONSTANTINOU, E.; VALENTE, M. T.; SEREBRENIK, A. On the abandonment and survival of open source projects: An empirical investigation. In: IEEE. **2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. [S.l.], 2019. p. 1–12.

AVELINO, G.; PASSOS, L.; HORA, A.; VALENTE, M. T. A novel approach for estimating truck factors. In: IEEE. **2016 IEEE 24th International Conference on Program Comprehension (ICPC)**. [S.l.], 2016. p. 1–10.

AVELINO, G.; PASSOS, L.; HORA, A.; VALENTE, M. T. Assessing code authorship: The case of the linux kernel. In: SPRINGER, CHAM. **IFIP International Conference on Open Source Systems**. [S.l.], 2017. p. 151–163.

AVELINO, G.; PASSOS, L.; PETRILLO, F.; VALENTE, M. T. Who can maintain this code?: Assessing the effectiveness of repository-mining techniques for identifying software maintainers. **IEEE Software**, IEEE, v. 36, n. 6, p. 34–42, 2018.

BADAMPUDI, D.; BRITTO, R.; UNTERKALMSTEINER, M. Modern code reviews-preliminary results of a systematic mapping study. In: **Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering**. [S.l.: s.n.], 2019. p. 340–345.

BANH, L.; HOLLDACK, F.; STROBEL, G. Copiloting the future: How generative ai transforms software engineering. **Information and Software Technology**, Elsevier, v. 183, p. 107751, 2025.

BECK, K. **Extreme programming explained: embrace change**. [S.l.]: addison-wesley professional, 2000.

BELLER, M.; PARK, A.; NAKAD, K.; PATEL, A.; MOHANTY, S.; GARBERSON, F.; MALONE, I. G.; GARG, V.; VERROKEN, H.; KENNEDY, A. et al. What's dat? three case studies of measuring software development productivity at meta with diff authoring time. **arXiv preprint arXiv:2503.10977**, 2025.

BIRD, C.; FORD, D.; ZIMMERMANN, T.; FORSGREN, N.; KALLIAMVAKOU, E.; LOWDERMILK, T.; GAZIT, I. Taking flight with copilot. **Communications of the ACM**, ACM New York, NY, USA, v. 66, n. 6, p. 56–62, 2023.

BIRD, C.; NAGAPPAN, N.; MURPHY, B.; GALL, H.; DEVANBU, P. Don't touch my code! examining the effects of ownership on software quality. In: **Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering**. [S.l.: s.n.], 2011. p. 4–14.

BISHOP, C. M. **Pattern recognition and machine learning**. [S.l.]: springer, 2006.

BOCK, T.; ALZNAUER, N.; JOBLIN, M.; APEL, S. Automatic core-developer identification on github: a validation study. **ACM Transactions on Software Engineering and Methodology**, ACM New York, NY, v. 32, n. 6, p. 1–29, 2023.

BORGES, H.; VALENTE, M. T. What's in a github star? understanding repository starring practices in a social coding platform. **Journal of Systems and Software**, Elsevier, v. 146, p. 112–129, 2018.

BRACHMAN, M.; EL-ASHRY, A.; DUGAN, C.; GEYER, W. Current and future use of large language models for knowledge work. **arXiv preprint arXiv:2503.16774**, 2025.

BREIMAN, L. Random forests. **Machine learning**, Springer, v. 45, n. 1, p. 5–32, 2001.

CALEFATO, F.; GEROSA, M. A.; IAFFALDANO, G.; LANUBILE, F.; STEINMACHER, I. Will you come back to contribute? investigating the inactivity of oss core developers in github. **Empirical Software Engineering**, Springer, v. 27, n. 3, p. 1–41, 2022.

CAMPBELL, G. A.; PAPAPETROU, P. P. **SonarQube in action**. [S.l.]: Manning Publications Co., 2013.

CANEDO, E. D.; BONIFÁCIO, R.; OKIMOTO, M. V.; SEREBRENIK, A.; PINTO, G.; MONTEIRO, E. Work practices and perceptions from women core developers in oss communities. In: **Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. [S.l.: s.n.], 2020. p. 1–11.

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

CANFORA, G.; PENTA, M. D.; OLIVETO, R.; PANICHELLA, S. Who is going to mentor newcomers in open source projects? In: **Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering**. [S.l.: s.n.], 2012. p. 1–11.

CASTRO, D.; SCHOTS, M. Analysis of test log information through interactive visualizations. In: **Proceedings of the 26th Conference on Program Comprehension**. [S.l.: s.n.], 2018. p. 156–166.

CHEN, A.; HUO, T.; NAM, Y.; PORT, D.; PERUMA, A. The impact of generative ai-powered code generation tools on software engineer hiring: Recruiters' experiences, perceptions, and strategies. **arXiv preprint arXiv:2409.00875**, 2024.

CHEN, M.; TWOREK, J.; JUN, H.; YUAN, Q.; PINTO, H. P. D. O.; KAPLAN, J.; EDWARDS, H.; BURDA, Y.; JOSEPH, N.; BROCKMAN, G. et al. Evaluating large language models trained on code. **arXiv preprint arXiv:2107.03374**, 2021.

CHOMBOON, K.; CHUJAI, P.; TEERARASSAMEE, P.; KERDPRASOP, K.; KERDPRASOP, N. An empirical study of distance metrics for k-nearest neighbor algorithm. In: **Proceedings of the 3rd international conference on industrial application engineering**. [S.l.: s.n.], 2015. p. 280–285.

CHOUCHEN, M.; OUNI, A.; MKAOUER, M. W.; KULA, R. G.; INOUE, K. Whoreview: A multi-objective search-based approach for code reviewers recommendation in modern code review. **Applied Soft Computing**, Elsevier, v. 100, p. 106908, 2021.

CHYUNG, S. Y.; ROBERTS, K.; SWANSON, I.; HANKINSON, A. Evidence-based survey design: The use of a midpoint on the likert scale. **Performance Improvement**, Wiley Online Library, v. 56, n. 10, p. 15–23, 2017.

COSENTINO, V.; IZQUIERDO, J. L. C.; CABOT, J. Assessing the bus factor of git repositories. In: IEEE. **2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.], 2015. p. 499–503.

COSTA, C.; FIGUEIREDO, J.; MURTA, L.; SARMA, A. Tipmerge: recommending experts for integrating changes across branches. In: **Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. [S.l.: s.n.], 2016. p. 523–534.

COSTA, C. de S.; FIGUEIREDO, J.; PIMENTEL, J.; SARMA, A.; MURTA, L. Recommending participants for collaborative merge sessions. **Trans Softw Eng**, p. 1–1, 2019.

CRESWELL, A.; WHITE, T.; DUMOULIN, V.; ARULKUMARAN, K.; SENGUPTA, B.; BHARATH, A. A. Generative adversarial networks: An overview. **IEEE signal processing magazine**, IEEE, v. 35, n. 1, p. 53–65, 2018.

CURY, O.; AVELINO, G. Knowledge islands: Visualizing developers knowledge concentration. In: SBC. **Simpósio Brasileiro de Engenharia de Software (SBES)**. [S.l.], 2024. p. 789–795.

CURY, O.; AVELINO, G. **The Impact of Generative AI on Code Expertise Models: An Exploratory Study**. 2025. Disponível em: <https://arxiv.org/abs/2507.08160>.

CURY, O.; AVELINO, G.; NETO, P. S.; BRITTO, R.; VALENTE, M. T. Identifying source code file experts. In: **Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement**. [S.l.: s.n.], 2022. p. 125–136.

CURY, O.; AVELINO, G.; NETO, P. S.; VALENTE, M. T.; BRITTO, R. Source code expert identification: Models and application. **Information and Software Technology**, Elsevier, p. 107445, 2024.

DENNY, P.; PRATHER, J.; BECKER, B. A.; FINNIE-ANSLEY, J.; HELLAS, A.; LEINONEN, J.; LUXTON-REILLY, A.; REEVES, B. N.; SANTOS, E. A.; SARSA, S. Computing education in the era of generative ai. **Communications of the ACM**, ACM New York, NY, USA, v. 67, n. 2, p. 56–67, 2024.

DOHMKE, T.; IANSITI, M.; RICHARDS, G. Sea change in software development: Economic and productivity analysis of the ai-powered developer lifecycle. **arXiv preprint arXiv:2306.15033**, 2023.

DURELLI, V. H.; DURELLI, R. S.; BORGES, S. S.; ENDO, A. T.; ELER, M. M.; DIAS, D. R.; GUIMARAES, M. P. Machine learning applied to software testing: A systematic mapping study. **IEEE Transactions on Reliability**, IEEE, v. 68, n. 3, p. 1189–1212, 2019.

EBERT, C.; LOURIDAS, P. Generative ai for software practitioners. **IEEE Software**, IEEE, v. 40, n. 4, p. 30–38, 2023.

ELISH, K. O.; ELISH, M. O. Predicting defect-prone software modules using support vector machines. **Journal of Systems and Software**, Elsevier, v. 81, n. 5, p. 649–660, 2008.

ERNST, N. A.; BAVOTA, G. Ai-driven development is here: Should you worry? **IEEE Software**, IEEE, v. 39, n. 2, p. 106–110, 2022.

EUCHNER, J. Generative ai. **Research-Technology Management**, Taylor & Francis, v. 66, n. 3, p. 71–74, 2023.

FALCÃO, F.; BARBOSA, C.; FONSECA, B.; GARCIA, A.; RIBEIRO, M.; GHEYI, R. On relating technical, social factors, and the introduction of bugs. In: IEEE. **2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.], 2020. p. 378–388.

FERREIRA, F.; SILVA, L. L.; VALENTE, M. T. Turnover in open-source projects: The case of core developers. In: **Proceedings of the 34th Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2020. p. 447–456.

FERREIRA, M.; VALENTE, M. T.; FERREIRA, K. A comparison of three algorithms for computing truck factors. In: IEEE. **2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)**. [S.l.], 2017. p. 207–217.

FEUERRIEGEL, S.; HARTMANN, J.; JANIESCH, C.; ZSCHECH, P. Generative ai. **Business & Information Systems Engineering**, Springer, v. 66, n. 1, p. 111–126, 2024.

FOLLECO, A.; KHOSHGOFTAAR, T. M.; HULSE, J. V.; BULLARD, L. Software quality modeling: The impact of class noise on the random forest classifier. In: IEEE. **2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)**. [S.l.], 2008. p. 3853–3859.

FRIEDMAN, J. H. Greedy function approximation: a gradient boosting machine. **Annals of statistics**, JSTOR, p. 1189–1232, 2001.

FRITZ, T.; MURPHY, G. C.; HILL, E. Does a programmer's activity indicate knowledge of code? In: **Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering**. [S.l.: s.n.], 2007. p. 341–350.

FRITZ, T.; MURPHY, G. C.; MURPHY-HILL, E.; OU, J.; HILL, E. Degree-of-knowledge: Modeling a developer's knowledge of code. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM New York, NY, USA, v. 23, n. 2, p. 1–42, 2014.

GAWEDA, A. M.; LYNCH, C. F.; SEAMON, N.; OLIVEIRA, G. Silva de; DELIWA, A. Typing exercises as interactive worked examples for deliberate practice in cs courses. In: **Proceedings of the Twenty-Second Australasian Computing Education Conference**. [S.l.: s.n.], 2020. p. 105–113.

GHAZI, A. N.; PETERSEN, K.; REDDY, S. S. V. R.; NEKKANTI, H. Survey research in software engineering: Problems and mitigation strategies. **IEEE Access**, IEEE, v. 7, p. 24703–24718, 2018.

GIRBA, T.; KUHN, A.; SEEBERGER, M.; DUCASSE, S. How developers drive software evolution. In: IEEE. **Eighth international workshop on principles of software evolution (IWPSE'05)**. [S.l.], 2005. p. 113–122.

GOZALO-BRIZUELA, R.; GARRIDO-MERCHÁN, E. C. A survey of generative ai applications. **arXiv preprint arXiv:2306.02781**, 2023.

GREWAL, B.; LU, W.; NADI, S.; BEZEMER, C.-P. Analyzing developer use of chatgpt generated code in open source github projects. In: IEEE. **2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)**. [S.l.], 2024. p. 157–161.

GÜEMES-PEÑA, D.; LÓPEZ-NOZAL, C.; MARTICORENA-SÁNCHEZ, R.; MAUDES-RAEDO, J. Emerging topics in mining software repositories. **Progress in Artificial Intelligence**, Springer, v. 7, n. 3, p. 237–247, 2018.

GUENTHER, N.; SCHONLAU, M. Support vector machines. **The Stata Journal**, SAGE Publications Sage CA: Los Angeles, CA, v. 16, n. 4, p. 917–937, 2016.

GUNN, S. R. et al. Support vector machines for classification and regression. **ISIS technical report**, v. 14, n. 1, p. 5–16, 1998.

HANNEBAUER, C.; PATALAS, M.; STÜNKEL, S.; GRUHN, V. Automatically recommending code reviewers based on their expertise: An empirical comparison. In: **Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering**. [S.l.: s.n.], 2016. p. 99–110.

HAO, H.; HASAN, K. A.; QIN, H.; MACEDO, M.; TIAN, Y.; DING, S. H.; HASSAN, A. E. An empirical study on developers shared conversations with chatgpt in github pull requests and issues. **arXiv preprint arXiv:2403.10468**, 2024.

HAQUE, M. A. Llms: A game-changer for software engineers? **BenchCouncil Transactions on Benchmarks, Standards and Evaluations**, Elsevier, p. 100204, 2025.

HARATIAN, V.; EVTIKHIEV, M.; DERAKHSHANFAR, P.; TÜZÜN, E.; KOVALENKO, V. Bfsig: Leveraging file significance in bus factor estimation. In: **31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. [S.l.: s.n.], 2023. p. 1926–1936.

HARDING, W. **AI Copilot Code Quality: Evaluating 2024's Increased Defect Rate via Code Quality Metrics**. [S.l.], 2025.

HASANLUO, M.; GHAREHCHOPOGH, F. S. Software cost estimation by a new hybrid model of particle swarm optimization and k-nearest neighbor algorithms. **Journal of Electrical and Computer Engineering Innovations (JECEI)**, Shahid Rajaee Teacher Training University, v. 4, n. 1, p. 49–55, 2016.

HATTORI, L.; LANZA, M. Mining the history of synchronous changes to refine code ownership. In: IEEE. **2009 6th ieee international working conference on mining software repositories**. [S.l.], 2009. p. 141–150.

HATTORI, L.; LANZA, M. Syde: a tool for collaborative software development. In: **Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2**. [S.l.: s.n.], 2010. p. 235–238.

HILTON, M.; TUNNELL, T.; HUANG, K.; MARINOV, D.; DIG, D. Usage, costs, and benefits of continuous integration in open-source projects. In: IEEE. **2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.], 2016. p. 426–437.

HOSSEN, M. K.; KAGDI, H.; POSHYVANYK, D. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In: **Proceedings of the 22nd International Conference on Program Comprehension**. [S.l.: s.n.], 2014. p. 130–141.

IBIAPINA, I. M. S.; ALVES, F. V. M.; LIRA, W. A. L.; SILVA, G. A.; NETO, P. A. S. Inferência da familiaridade de código por meio da mineração de repositórios de software. **Simpósio Brasileiro de Qualidade de Software - SBQS**, 2017.

JABRAYILZADE, E.; EVTIKHIEV, M.; TÜZÜN, E.; KOVALENKO, V. Bus factor in practice. **arXiv preprint arXiv:2202.01523**, 2022.

JIANG, J.; LO, D.; MA, X.; FENG, F.; ZHANG, L. Understanding inactive yet available assignees in github. **Information and Software Technology**, Elsevier, v. 91, p. 44–55, 2017.

JIN, K.; WANG, C.-Y.; PHAM, H. V.; HEMMATI, H. Can chatgpt support developers? an empirical evaluation of large language models for code generation. **arXiv preprint arXiv:2402.11702**, 2024.

JR, D. W. H.; LEMESHOW, S.; STURDIVANT, R. X. **Applied logistic regression**. [S.l.]: John Wiley & Sons, 2013. v. 398.

KAGDI, H.; GETHERS, M.; POSHYVANYK, D.; HAMMAD, M. Assigning change requests to software developers. **Journal of software: Evolution and Process**, Wiley Online Library, v. 24, n. 1, p. 3–33, 2012.

KAGDI, H.; HAMMAD, M.; MALETIC, J. I. Who can help me with this source code change? In: IEEE. **2008 IEEE International Conference on Software Maintenance**. [S.l.], 2008. p. 157–166.

KAGDI, H.; POSHYVANYK, D. Who can help me with this change request? In: IEEE. **2009 IEEE 17th International Conference on Program Comprehension**. [S.l.], 2009. p. 273–277.

Karlsson, Andreas. **Driving Development Resilience: Analyzing Truck Factors across Proprietary and Open-Source Projects**. 2023. (LU-CS-EX). Student Paper.

KAUR, A.; MALHOTRA, R. Application of random forest in predicting fault-prone classes. In: IEEE. **2008 International Conference on Advanced Computer Theory and Engineering**. [S.l.], 2008. p. 37–43.

KAZEMITABAAR, M.; HUANG, O.; SUH, S.; HENLEY, A. Z.; GROSSMAN, T. Exploring the design space of cognitive engagement techniques with ai-generated code for enhanced learning. **arXiv preprint arXiv:2410.08922**, 2024.

KERSTEN, M. **Focusing knowledge work with task context**. Tese (Doutorado) — University of British Columbia, 2007.

KHATUN, A.; SAKIB, K. A bug assignment technique based on bug fixing expertise and source commit recency of developers. In: IEEE. **2016 19th International Conference on Computer and Information Technology (ICCIT)**. [S.l.], 2016. p. 592–597.

KIM, J.; LEE, E. Understanding review expertise of developers: A reviewer recommendation approach based on latent dirichlet allocation. **Symmetry**, Multidisciplinary Digital Publishing Institute, v. 10, n. 4, p. 114, 2018.

KLIMOV, E.; AHMED, M. U.; SVIRIDOV, N.; DERAKHSHANFAR, P.; TÜZUÜ, E.; KOVALENKO, V. Bus factor explorer. In: IEEE. **2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.], 2023. p. 2018–2021.

KPODJEDO, S.; RICCA, F.; GALINIER, P.; ANTONIOL, G. Not all classes are created equal: toward a recommendation system for focusing testing. In: **Proceedings of the 2008 international workshop on Recommendation systems for software engineering**. [S.l.: s.n.], 2008. p. 6–10.

KRÜGER, J.; WIEMANN, J.; FENSKE, W.; SAAKE, G.; LEICH, T. Do you remember this source code? In: IEEE. **2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)**. [S.l.], 2018. p. 764–775.

KULKARNI, P. **Reinforcement and systemic machine learning for decision making**. [S.l.]: John Wiley & Sons, 2012. v. 1.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. **nature**, Nature Publishing Group UK London, v. 521, n. 7553, p. 436–444, 2015.

LEHMANN, M.; CORNELIUS, P. B.; STING, F. J. Ai meets the classroom: When does chatgpt harm learning? **Available at SSRN 4941259**, 2024.

LIAW, A.; WIENER, M. et al. Classification and regression by randomforest. **R news**, v. 2, n. 3, p. 18–22, 2002.

LINAKER, J.; SULAMAN, S. M.; HÖST, M.; MELLO, R. M. de. Guidelines for conducting surveys in software engineering v. 1.1. **Lund University**, 2015.

LOELIGER, J.; MCCULLOUGH, M. **Version Control with Git: Powerful tools and techniques for collaborative software development**. [S.l.]: " O'Reilly Media, Inc.", 2012.

LUCAS, E. M.; OLIVEIRA, T. C.; SCHNEIDER, D.; ALENCAR, P. S. C. Knowledge-oriented models based on developer-artifact and developer-developer interactions. **IEEE Access**, v. 8, p. 218702–218719, 2020.

LUGER, G. F. **Inteligência Artificial-: Estruturas e estratégias para a solução de problemas complexos**. [S.l.]: Bookman, 2004.

MA, B.; CHEN, L.; KONOMI, S. Enhancing programming education with chatgpt: A case study on student perceptions and interactions in a python course. In: SPRINGER. **International Conference on Artificial Intelligence in Education**. [S.l.], 2024. p. 113–126.

MAEN, H.; COLLARD, M.; MALETIC, J. Measuring class importance in the context of design evolution. In: **Program Comprehension (ICPC), IEEE 18th International Conference on. IEEE**. [S.l.: s.n.], 2010.

MANI, S.; PADHYE, R.; SINHA, V. S. Mining api expertise profiles with partial program analysis. In: **Proceedings of the 9th India Software Engineering Conference**. [S.l.: s.n.], 2016. p. 109–118.

MAZINANIAN, D.; KETKAR, A.; TSANTALIS, N.; DIG, D. Understanding the use of lambda expressions in java. **Proceedings of the ACM on Programming Languages**, ACM New York, NY, USA, v. 1, n. OOPSLA, p. 1–31, 2017.

MCDONALD, D. W.; ACKERMAN, M. S. Expertise recommender: a flexible recommendation system and architecture. In: **Proceedings of the 2000 ACM conference on Computer supported cooperative work**. [S.l.: s.n.], 2000. p. 231–240.

MILANO, K.; CAFEO, B. Navigating expertise in configurable software systems through the maze of variability. **arXiv preprint arXiv:2401.10699**, 2024.

MINTO, S.; MURPHY, G. C. Recommending emergent teams. In: IEEE. **Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)**. [S.l.], 2007. p. 5–5.

MOCKUS, A.; HERBSLEB, J. D. Expertise browser: a quantitative approach to identifying expertise. In: IEEE. **Proceedings of the 24th International Conference on Software Engineering. ICSE 2002**. [S.l.], 2002. p. 503–512.

MOHRI, M.; ROSTAMIZADEH, A.; TALWALKAR, A. **Foundations of machine learning**. [S.l.]: MIT press, 2018.

MONTANDON, J. E.; SILVA, L. L.; VALENTE, M. T. Identifying experts in software libraries and frameworks among github users. In: IEEE. **2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)**. [S.l.], 2019. p. 276–287.

MURPHY, G. C.; KERSTEN, M.; FINDLATER, L. How are java software developers using the eclipse ide? **IEEE software**, IEEE, v. 23, n. 4, p. 76–83, 2006.

MURPHY, K. P. **Probabilistic machine learning: an introduction**. [S.l.]: MIT press, 2022.

NASSIF, A. B.; CAPRETZ, L. F.; HO, D.; AZZEH, M. A treeboost model for software effort estimation based on use case points. In: IEEE. **2012 11th International Conference on Machine Learning and Applications**. [S.l.], 2012. v. 2, p. 314–319.

NATEKIN, A.; KNOLL, A. Gradient boosting machines, a tutorial. **Frontiers in neurorobotics**, Frontiers, v. 7, p. 21, 2013.

NAVARRO, G. A guided tour to approximate string matching. **ACM computing surveys (CSUR)**, ACM New York, NY, USA, v. 33, n. 1, p. 31–88, 2001.

NG, A.; JORDAN, M. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. **Advances in neural information processing systems**, v. 14, 2001.

NGUYEN-DUC, A.; CABRERO-DANIEL, B.; PRZYBYLEK, A.; ARORA, C.; KHANNA, D.; HERDA, T.; RAFIQ, U.; MELEGATI, J.; GUERRA, E.; KEMELL, K.-K. et al. Generative artificial intelligence for software engineering–a research agenda. **arXiv preprint arXiv:2310.18648**, 2023.

NIELEBOCK, S.; HEUMÜLLER, R.; ORTMEIER, F. Programmers do not favor lambda expressions for concurrent object-oriented code. **Empirical Software Engineering**, Springer, v. 24, n. 1, p. 103–138, 2019.

OLIVEIRA, J.; VIGGIATO, M.; FIGUEIREDO, E. How well do you know this library? mining experts from source code analysis. In: **Proceedings of the XVIII Brazilian Symposium on Software Quality**. [S.l.: s.n.], 2019. p. 49–58.

OTTE, S. Version control systems. **Computer Systems and Telematics**, p. 11–13, 2009.

OVERFLOW, S. Stack overflow developer survey 2022. **URL: https://survey. stackoverflow. co**, 2022.

OVERFLOW, S. Overflow: 2024 state of development survey. **https://survey.stackoverflow.co/2024/**, 2024.

OVERHOLSER, B. R.; SOWINSKI, K. M. Biostatistics primer: part 2. **Nutrition in clinical practice**, Wiley Online Library, v. 23, n. 1, p. 76–84, 2008.

PADHYE, R.; MANI, S.; SINHA, V. S. A study of external community contribution to open-source projects on github. In: **Proceedings of the 11th Working Conference on Mining Software Repositories**. [S.l.: s.n.], 2014. p. 332–335.

PANDIS, N. The chi-square test. **American journal of orthodontics and dentofacial orthopedics**, Elsevier, v. 150, n. 5, p. 898–899, 2016.

PESLAK, A.; KOVALCHICK, L. Ai for coders: An analysis of the usage of chatgpt and github copilot. **Issues in Information Systems**, International Association for Computer Information Systems, v. 25, n. 4, p. 252–260, 2024.

PETERSON, L. E. K-nearest neighbor. **Scholarpedia**, v. 4, n. 2, p. 1883, 2009.

PILATO, C. M.; COLLINS-SUSSMAN, B.; FITZPATRICK, B. W. **Version control with subversion: next generation open source version control**. [S.l.]: " O'Reilly Media, Inc.", 2008.

POWERS, D. M. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. **arXiv preprint arXiv:2010.16061**, 2020.

PRATHER, J.; REEVES, B. N.; DENNY, P.; BECKER, B. A.; LEINONEN, J.; LUXTON-REILLY, A.; POWELL, G.; FINNIE-ANSLEY, J.; SANTOS, E. A. "it's weird that it knows what i want": Usability and interactions with copilot for novice programmers. **ACM Transactions on Computer-Human Interaction**, ACM New York, NY, v. 31, n. 1, p. 1–31, 2023.

PRATHER, J.; REEVES, B. N.; LEINONEN, J.; MACNEIL, S.; RANDRIANASOLO, A. S.; BECKER, B. A.; KIMMEL, B.; WRIGHT, J.; BRIGGS, B. The widening gap: The benefits and harms of generative ai for novice programmers. In: **Proceedings of the 2024 ACM Conference on International Computing Education Research-Volume 1**. [S.l.: s.n.], 2024. p. 469–486.

PUNTER, T.; CIOLKOWSKI, M.; FREIMUT, B.; JOHN, I. Conducting on-line surveys in software engineering. In: IEEE. **2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.** [S.l.], 2003. p. 80–88.

RAHIM, A. binti A.; RAHIM, W. A. H. bin A.; MD, N. binti. Exploring student perceptions and interactions with chatgpt in java programming learning. **environment**, v. 14, p. 15, 2024.

RAHMAN, F.; DEVANBU, P. Ownership, experience and defects: a fine-grained study of authorship. In: **Proceedings of the 33rd International Conference on Software Engineering**. [S.l.: s.n.], 2011. p. 491–500.

RAJLICH, V. Software evolution and maintenance. In: **Proceedings of the on Future of Software Engineering**. [S.l.: s.n.], 2014. p. 133–144.

RALPH, P.; BALTES, S.; ADISAPUTRI, G.; TORKAR, R.; KOVALENKO, V.; KALINOWSKI, M.; NOVIELLI, N.; YOO, S.; DEVROEY, X.; TAN, X. et al. Pandemic programming: how covid-19 affects software developers and how their organizations can help (2020). **arXiv preprint arXiv:2005.01127**, 2020.

RASHEED, Z.; WASEEM, M.; SAMI, M. A.; KEMELL, K.-K.; AHMAD, A.; DUC, A. N.; SYSTÄ, K.; ABRAHAMSSON, P. Autonomous agents in software development: A vision paper. In: SPRINGER NATURE SWITZERLAND CHAM. **International Conference on Agile Software Development**. [S.l.], 2024. p. 15–23.

RAY, B.; POSNETT, D.; FILKOV, V.; DEVANBU, P. A large scale study of programming languages and code quality in github. In: **Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering**. [S.l.: s.n.], 2014. p. 155–165.

RAY, P. P. A review on vibe coding: Fundamentals, state-of-the-art, challenges and future directions. **Authorea Preprints**, Authorea, 2025.

REY, D.; NEUHÄUSER, M. Wilcoxon-signed-rank test. In: **International encyclopedia of statistical science**. [S.l.]: Springer, 2011. p. 1658–1659.

RICCA, F.; MARCHETTO, A. Are heroes common in floss projects? In: **Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement**. [S.l.: s.n.], 2010. p. 1–4.

RIGBY, P. C.; ZHU, Y. C.; DONADELLI, S. M.; MOCKUS, A. Quantifying and mitigating turnover-induced knowledge loss: case studies of chrome and a project at avaya. In: IEEE. **2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)**. [S.l.], 2016. p. 1006–1016.

RIGGER, M.; MARR, S.; KELL, S.; LEOPOLDSEDER, D.; MÖSSENBÖCK, H. An analysis of x86-64 inline assembly in c programs. In: **Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments**. [S.l.: s.n.], 2018. p. 84–99.

ROBILLARD, M. P. Turnover-induced knowledge loss in practice. In: **Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. [S.l.: s.n.], 2021. p. 1292–1302.

ROBLES, G.; TREUDE, C.; GONZALEZ-BARAHONA, J. M.; KULA, R. G. The role of code proficiency in the era of generative ai. **arXiv preprint arXiv:2405.01565**, 2024.

ROOSE, K. How chatgpt kicked off an ai arms race. **International New York Times**, International Herald Tribune, p. NA–NA, 2023.

ROT, A.; SOBINSKA, M.; BUSCH, P. Programming teams in remote working environments: an analysis of performance and productivity. In: IEEE. **2023 13th International Conference on Advanced Computer Information Technologies (ACIT)**. [S.l.], 2023. p. 376–381.

RUSSELL, S.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. 3rd. ed. USA: Prentice Hall Press, 2009. ISBN 0136042597.

RUSSO, D. Navigating the complexity of generative ai adoption in software engineering. **ACM Transactions on Software Engineering and Methodology**, ACM New York, NY, 2024.

SAJEDI-BADASHIAN, A.; STROULIA, E. Guidelines for evaluating bug-assignment research. **Journal of Software: Evolution and Process**, Wiley Online Library, p. e2250, 2020.

SATAPATHY, S. M.; ACHARYA, B. P.; RATH, S. K. Class point approach for software effort estimation using stochastic gradient boosting technique. **ACM SIGSOFT Software Engineering Notes**, ACM New York, NY, USA, v. 39, n. 3, p. 1–6, 2014.

SATAPATHY, S. M.; ACHARYA, B. P.; RATH, S. K. Early stage software effort estimation using random forest technique based on use case points. **IET Software**, IET, v. 10, n. 1, p. 10–17, 2016.

SATAPATHY, S. M.; RATH, S. K. Empirical assessment of machine learning models for agile software development effort estimation using story points. **Innovations in Systems and Software Engineering**, Springer, v. 13, n. 2-3, p. 191–200, 2017.

SAUVOLA, J.; TARKOMA, S.; KLEMETTINEN, M.; RIEKKI, J.; DOERMANN, D. Future of software development with generative ai. **Automated Software Engineering**, Springer, v. 31, n. 1, p. 26, 2024.

SCHOBER, P.; BOER, C.; SCHWARTE, L. A. Correlation coefficients: appropriate use and interpretation. **Anesthesia & analgesia**, Wolters Kluwer, v. 126, n. 5, p. 1763–1768, 2018.

SCHULER, D.; ZIMMERMANN, T. Mining usage expertise from version archives. In: **Proceedings of the 2008 international working conference on Mining software repositories**. [S.l.: s.n.], 2008. p. 121–124.

SHEPPERD, M.; BOWES, D.; HALL, T. Researcher bias: The use of machine learning in software defect prediction. **IEEE Transactions on Software Engineering**, IEEE, v. 40, n. 6, p. 603–616, 2014.

SILVA, J. R. da; CLUA, E.; MURTA, L.; SARMA, A. Niche vs. breadth: Calculating expertise over time through a fine-grained analysis. In: IEEE. **2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.], 2015. p. 409–418.

SINGH, A.; TANEJA, K.; GUAN, Z.; GHOSH, A. Protecting human cognition in the age of ai. **arXiv preprint arXiv:2502.12447**, 2025.

SKRIPCHUK, J.; BENNETT, N.; ZHANG, J.; LI, E.; PRICE, T. Analysis of novices' web-based help-seeking behavior while programming. In: **Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1**. [S.l.: s.n.], 2023. p. 945–951.

SOHAIL, S. S.; FARHAT, F.; HIMEUR, Y.; NADEEM, M.; MADSEN, D. Ø.; SINGH, Y.; ATALLA, S.; MANSOOR, W. Decoding chatgpt: a taxonomy of existing research, current challenges, and possible future directions. **Journal of King Saud University-Computer and Information Sciences**, Elsevier, p. 101675, 2023.

ŞORA, I.; CHIRILA, C.-B. Finding key classes in object-oriented software systems by techniques based on static analysis. **Information and Software Technology**, Elsevier, v. 116, p. 106176, 2019.

SPINELLIS, D. Version control systems. **IEEE Software**, IEEE, v. 22, n. 5, p. 108–109, 2005.

STOL, K.-J.; RALPH, P.; FITZGERALD, B. Grounded theory in software engineering research: a critical review and guidelines. In: **Proceedings of the 38th International conference on software engineering**. [S.l.: s.n.], 2016. p. 120–131.

SÜLÜN, E.; TÜZÜN, E.; DOĞRUSÖZ, U. Reviewer recommendation using software artifact traceability graphs. In: **Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering**. [S.l.: s.n.], 2019. p. 66–75.

SÜLÜN, E.; TÜZÜN, E.; DOĞRUSÖZ, U. Rstrace+: Reviewer suggestion using software artifact traceability graphs. **Information and Software Technology**, Elsevier, v. 130, p. 106455, 2020.

SÜLÜN, E.; TÜZÜN, E.; DOĞRUSÖZ, U. Rstrace+: Reviewer suggestion using software artifact traceability graphs. **Information and Software Technology**, Elsevier, v. 130, p. 106455, 2021.

SUN, X.; YANG, H.; XIA, X.; LI, B. Enhancing developer recommendation with supplementary information via mining historical commits. **Journal of Systems and Software**, Elsevier, v. 134, p. 355–368, 2017.

THONGTANUNAM, P.; TANTITHAMTHAVORN, C. Code ownership: The principles, differences, and their associations with software quality. In: . [S.l.: s.n.], 2024.

TORNHILL, A. Your code as a crime scene: use forensic techniques to arrest defects, bottlenecks, and bad design in your programs. **Your Code as a Crime Scene**, The Pragmatic Bookshelf, p. 1–218, 2015.

TORNHILL, A. Assessing technical debt in automated tests with codescene. In: IEEE. **2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [S.l.], 2018. p. 122–125.

ULFSNES, R.; MOE, N. B.; STRAY, V.; SKARPEN, M. Transforming software development with generative ai: empirical insights on collaboration and workflow. In: **Generative AI for effective software development**. [S.l.]: Springer, 2024. p. 219–234.

WASEEM, M.; DAS, T.; AHMAD, A.; LIANG, P.; FEHMIDEH, M.; MIKKONEN, T. Chatgpt as a software development bot: a project-based study. **arXiv preprint arXiv:2310.13648**, 2023.

WEN, J.; LI, S.; LIN, Z.; HU, Y.; HUANG, C. Systematic literature review of machine learning based software development effort estimation models. **Information and Software Technology**, Elsevier, v. 54, n. 1, p. 41–59, 2012.

WESTON, J.; WATKINS, C. **Multi-class support vector machines**. [S.l.], 1998.

WILLIAMS, L.; KESSLER, R. R. **Pair programming illuminated**. [S.l.]: Addison-Wesley Professional, 2003.

XIAO, T.; TREUDE, C.; HATA, H.; MATSUMOTO, K. Devgpt: Studying developer-chatgpt conversations. In: **Proceedings of the 21st International Conference on Mining Software Repositories**. [S.l.: s.n.], 2024. p. 227–230.

XING, F.; GUO, P.; LYU, M. R. A novel method for early software quality prediction based on support vector machine. In: IEEE. **16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)**. [S.l.], 2005. p. 10–pp.

YILMAZ, R.; YILMAZ, F. G. K. Augmented intelligence in programming learning: Examining student views on the use of chatgpt for programming learning. **Computers in Human Behavior: Artificial Humans**, Elsevier, v. 1, n. 2, p. 100005, 2023.

ZAZWORKA, N.; STAPEL, K.; KNAUSS, E.; SHULL, F.; BASILI, V. R.; SCHNEIDER, K. Are developers complying with the process: an xp study. In: **Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement**. [S.l.: s.n.], 2010. p. 1–10.

ZHANG, B.; LIANG, P.; ZHOU, X.; AHMAD, A.; WASEEM, M. Demystifying practices, challenges and expected features of using github copilot. **arXiv preprint arXiv:2309.05687**, 2023.

ZIEGLER, A.; KALLIAMVAKOU, E.; LI, X. A.; RICE, A.; RIFKIN, D.; SIMISTER, S.; SITTAMPALAM, G.; AFTANDILIAN, E. Measuring github copilot's impact on productivity. **Communications of the ACM**, ACM New York, NY, USA, v. 67, n. 3, p. 54–63, 2024.

ZOLKIFLI, N. N.; NGAH, A.; DERAMAN, A. Version control system: A review. **Procedia Computer Science**, Elsevier, v. 135, p. 408–415, 2018.

ZVIEL-GIRSHIN, R. The good and bad of ai tools in novice programming education. **Education Sciences**, MDPI AG, v. 14, n. 10, p. 1089, 2024.