
Code Readability Assessment and Improvement: Human and Automated Perspectives

Carlos Eduardo de Carvalho Dantas



UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Uberlândia
2025

Carlos Eduardo de Carvalho Dantas

**Code Readability Assessment and
Improvement: Human and Automated
Perspectives**

Tese de doutorado apresentada ao Programa de
Pós-graduação da Faculdade de Computação
da Universidade Federal de Uberlândia como
parte dos requisitos para a obtenção do título
de Doutor em Ciência da Computação.

Área de concentração: Ciência da Computação

Supervisor: Marcelo de Almeida Maia

Uberlândia

2025

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

D192 Dantas, Carlos Eduardo de Carvalho, 1982-
2025 Code Readability Assessment and Improvement: Human and
Automated Perspectives [recurso eletrônico] / Carlos Eduardo de
Carvalho Dantas. - 2025.

Orientador: Marcelo de Almeida Maia.

Tese (Doutorado) - Universidade Federal de Uberlândia, Pós-
graduação em Ciência da Computação.

Modo de acesso: Internet.

DOI <http://doi.org/10.14393/ufu.te.2025.555>

Inclui bibliografia.

1. Computação. I. Maia, Marcelo de Almeida, 1969-, (Orient.). II.
Universidade Federal de Uberlândia. Pós-graduação em Ciência da
Computação. III. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:

Gizele Cristine Nunes do Couto - CRB6/2091

Nelson Marcos Ferreira - CRB6/3074



ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Tese, 25/2025, PPGCO				
Data:	29 de Agosto de 2025	Hora de início:	13:00	Hora de encerramento:	16:30
Matrícula do Discente:	11723CCP002				
Nome do Discente:	Carlos Eduardo de Carvalho Dantas				
Título do Trabalho:	Code Readability Assessment and Improvement: Human and Automated Perspectives				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Engenharia de Software				
Projeto de Pesquisa de vinculação:	Smart Developer - Assistentes Automatizados para Apoio ao Desenvolvedor de Software. Manutenção de software inteligente.				

Reuniu-se por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Stéphane Julia - FACOM/UFU, Rafael Dias Araújo - FACOM/UFU, Fernando José Castor de Lima Filho- University of Twente , André Cavalcante Hora - DCC/UFGM, e Marcelo de Almeida Maia- FACOM/UFU, orientador do(a) candidato(a).

Os examinadores participaram desde as seguintes localidades: Fernando José Castor de Lima Filho - Enschede/Holanda, André Cavalcante Hora - Belo Horizonte/MG. Os outros membros da banca e o aluno participaram da cidade de Uberlândia.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Marcelo de Almeida Maia, apresentou a Comissão Examinadora e o candidato(a), agradeceu a presença do público, e concedeu ao(á) Discente a palavra para a exposição do seu trabalho. A duração da apresentação do(a) Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir ao candidato(a). Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato(a):

Aprovado

Esta defesa faz parte dos requisitos necessários à obtenção do título de Doutor.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.



Documento assinado eletronicamente por **Marcelo de Almeida Maia, Professor(a) do Magistério Superior**, em 03/09/2025, às 09:57, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Rafael Dias Araújo, Professor(a) do Magistério Superior**, em 03/09/2025, às 12:38, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Andre Cavalcante Hora, Usuário Externo**, em 03/09/2025, às 14:40, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Fernando José Castor de Lima Filho, Usuário Externo**, em 04/09/2025, às 06:46, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Stéphane Julia, Professor(a) do Magistério Superior**, em 04/09/2025, às 08:23, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **6590569** e o código CRC **24489C20**.

Acknowledgements

Dedico esta tese à memória da minha mãe, Jaida Abadia de Carvalho, e da minha avó paterna, Terezinha Macedo Dantas, que nos deixaram durante os momentos finais da escrita deste trabalho.

À minha avó, por me acolher em sua casa no início da minha vida profissional, gesto que me deu apoio para seguir adiante.

À minha mãe, que como professora, sempre acreditou na força transformadora da educação, dedicou-se incansavelmente aos seus dois filhos e tinha o grande desejo de vê-los cursando doutorado. Este desejo, mãe, eu consegui concretizar, graças a você.

Agradeço, do fundo do coração, pela compreensão e apoio incondicional da minha amada esposa, Amanda, dos meus queridos filhos, Eduardo e Fernando, e, claro, ao meu orientador, Professor Marcelo Maia.

I dedicate this thesis to the memory of my mother, Jaida Abadia de Carvalho, and my paternal grandmother, Terezinha Macedo Dantas, who passed away during the final stages of this work.

Thanks to my grandmother for welcoming me to her home at the beginning of my professional life, a gesture that gave me the support I needed to move forward.

To my mother, who, as a teacher, always believed in the transformative power of education, she devoted herself tirelessly to her two children and cherished the great desire to see them pursue doctoral studies. Mother, I have fulfilled this wish, thanks to you.

I am deeply grateful for the understanding and unconditional support of my beloved wife, Amanda, my dear sons, Eduardo and Fernando, and, of course, my advisor, Professor Marcelo Maia.

“We can only see a short distance ahead, but we can see plenty there that needs to be done.”

(Alan Turing)

Resumo

A legibilidade é uma característica crucial no desenvolvimento de software, pois os desenvolvedores frequentemente dedicam uma parte significativa do tempo lendo e compreendendo o código. No entanto, ainda existem desafios tanto na definição de métricas precisas para sua avaliação quanto na automação de refatorações que visam aprimorá-la. Com a ascensão dos modelos de linguagem de grande escala (LLMs), surge uma nova oportunidade: avaliar não apenas a legibilidade do código gerado por esses modelos, mas também sua capacidade de refatorar código existente, buscando torná-lo mais legível. Nesse contexto, o presente estudo avalia o desempenho de legibilidade de três LLMs (ChatGPT, DeepSeek e Gemini), comparando os trechos de código por eles gerados com aqueles escritos por humanos no Stack Overflow. Com isso, é apresentada uma abordagem híbrida que combina ferramentas automáticas de análise estática (ASATs) com refatorações baseadas em LLMs para melhorar a legibilidade do código. Além disso, é extraída uma taxonomia das melhorias de legibilidade realizadas por desenvolvedores em repositórios reais do GitHub, escritos nas linguagens Java e Python. Os resultados indicam que os LLMs, em geral, geram código com significativamente menos alertas de legibilidade em comparação com as recomendações do Stack Overflow. A combinação de LLMs e ASATs oferece um potencial valioso para refatorações orientadas à legibilidade, especialmente ao melhorar o código por meio de uma única interação com os LLMs. Modelos focados em geração de código, como o DeepSeek Coder, alcançaram o melhor desempenho no refinamento do código-fonte. Em código de projetos reais, os desenvolvedores tendem a melhorar a legibilidade ao esclarecer a intenção do código e reduzir redundâncias e trechos verbosos. Embora desenvolvedores de Python e Java compartilhem preocupações como a melhoria da nomenclatura, algumas diferenças são observadas: desenvolvedores Java focam mais na remoção de código redundante e desnecessário, enquanto desenvolvedores Python priorizam a documentação de código e a formatação correta usando padrões como as diretrizes PEP 8.

Palavras-chave: legibilidade de código, modelos de linguagem de grande escala, pull request, revisão de código, ferramentas automáticas de análise estática.

Abstract

Readability is a crucial characteristic in software development because developers often spend a significant amount of time reading and understanding code. However, there are still challenges both in accurately assessing code readability and in automatically applying refactoring operations to improve it. As large language models (LLMs) revolutionize the way code is written, a new opportunity emerges: to evaluate not only the readability of the code generated by these models but also their ability to refactor existing code into more readable versions. This study evaluates the readability of the code generated from three LLMs (ChatGPT, DeepSeek, and Gemini), comparing the code snippets they generate with those written by humans on Stack Overflow. Based on this evaluation, a hybrid approach is presented that combines automatic static analysis tools (ASATs) with LLM-based refactorings to improve code readability. In addition, a taxonomy of readability improvements is extracted from real-world GitHub repositories written in Java and Python. The results indicate that LLMs generate code with significantly fewer readability warnings compared to Stack Overflow recommendations. Combining LLMs and ASATs offers valuable potential for readability-oriented refactorings, especially when improving code through a single interaction with the LLMs. Code-generation-focused models, such as the DeepSeek Coder, achieved the best performance in refining source code. In real-world code, developers tend to improve readability by clarifying the intent of the code and reducing redundancy and verbosity. While Python and Java developers share concerns such as improving naming, some differences are observed: Java developers focus more on removing redundant and unnecessary code, whereas Python developers prioritize code documentation and correct formatting using standards such as the PEP 8 guidelines.

Keywords: code readability, large language models, pull request, code review, automatic static analysis tools.

List of Figures

Figure 1 – Code Readability Perspectives	31
Figure 2 – Example of a code smell rule that impacts code readability	49
Figure 3 – An example of commit performed on GitHub	52
Figure 4 – An example of Pull Request (PR) performed on GitHub	53
Figure 5 – Options of merging: (a) Merge Pull Request and (b) Squash and Merge	54
Figure 6 – Distribution of Python PRs mentioning code readability improvements, grouped by year.	55
Figure 7 – A graphql query and the extracted data using the GitHub interface . .	56
Figure 8 – A ChatGPT response for the query <i>"How to reverse a string in java"</i> .	59
Figure 9 – A merged PR using ChatGPT assistance to improve code	60
Figure 10 – Overview of the proposed methodology	65
Figure 11 – Input queries performed and evaluated by developers into CROKAGE tool	66
Figure 12 – The Top-3 recommended Code Snippets by CROKAGE for the input query ID = 208: "Find the second largest number	69
Figure 13 – Distribution of the selected Stack Overflow posts according to their ranking in CROKAGE	71
Figure 14 – Examples of a Non-Readability Rule and a Readability Rule identified in SonarLint	73
Figure 15 – Data set to apply Friedman tests followed by Nemenyi's post-hoc tests	75
Figure 16 – Boxplot of Lines of Code (LOC) for all 327 code snippets, grouped by baseline	77
Figure 17 – Boxplot of comments LOC for all 327 code snippets, grouped by baseline	77
Figure 18 – Boxplot of Cognitive Complexity for all 327 code snippets, grouped by baseline	78
Figure 19 – SonarLint code readability rules organized into five main categories and 21 subcategories.	81
Figure 20 – SonarLint code readability rules organized into 16 syntactic structures.	90

Figure 21 – Stack Overflow thread with the recommended post by CROKAGE for the input query ID = 164: “how to sort a list”	94
Figure 22 – ChatGPT-generated answer for the input query ID = 164: “how to sort a list”	95
Figure 23 – Overview of the proposed methodology to answer RQ ₁	105
Figure 24 – Example of User-Facing Readability, Tangled, Readability-Focused and Unexplained Readability PRs extracted from GitHub	108
Figure 25 – PR ID = 31936 from <i>spack/spack</i> . Extraction of developers’ description, code before and after the commit, producing three instances (types of changes)	111
Figure 26 – Word cloud of the developers descriptions on improving code readability in Java	112
Figure 27 – Word cloud of the developers descriptions on improving code readability in Python	112
Figure 28 – Distribution of number of stars, pull requests and contributors on Python repositories	117
Figure 29 – Distribution of number of stars, pull requests and contributors on Java repositories	117
Figure 30 – Types of readability improvements in the Clarify Code Intent category	119
Figure 31 – Types of readability improvements in the Prefer Concise Code category	128
Figure 32 – Types of readability improvements in the Clean Up category	136
Figure 33 – Types of readability improvements in the Improve Code Modularity .	138
Figure 34 – Types of readability improvements in the Improve Documentation category	142
Figure 35 – Types of readability improvements in the Improve Code Formatting category	143
Figure 36 – Overview of the proposed methodology to answer RQ ₁	157
Figure 37 – Improving Code Readability Example on Query ID = 26 “Odbc connection to Mysql in Java”	160

List of Tables

Table 1	– SonarLint recommended modifications on a code snippet.	30
Table 2	– Token-level features used in the Buse and Weimer readability model. . .	45
Table 3	– Examples of developer queries and the corresponding Stack Overflow post IDs recommended by CROKAGE.	58
Table 4	– The top-10 most frequent n-grams in the input queries used in this study	66
Table 5	– Number of readability warnings and affected code snippets grouped by the baseline ($n = 327$ snippets)	76
Table 6	– Readability classification according to the Scalabrino model, with counts and standardized residuals by baseline. ($n = 327$ snippets)	78
Table 7	– Number of readability warnings and affected code snippets per category and baseline (<i>Snip.</i> : number of code snippets affected; <i>Warn.</i> : total number of readability warnings). ($n = 327$ snippets)	81
Table 8	– Number of readability warnings and affected code snippets per subcat- egory and baseline in the <i>Clarify Code Intent</i> category (<i>Snip.</i> : number of code snippets affected; <i>Warn.</i> : total number of readability warnings). ($n = 327$ snippets)	82
Table 9	– Number of readability warnings and affected code snippets per subcat- egory and baseline in the <i>Prefer Concise Code</i> category (<i>Snip.</i> : number of code snippets affected; <i>Warn.</i> : total number of readability warnings). ($n = 327$ snippets)	85
Table 10	– Number of readability warnings and affected code snippets per subcate- gory and baseline in the <i>Clean Up</i> category (<i>Snip.</i> : number of code snip- pets affected; <i>Warn.</i> : total number of readability warnings). ($n = 327$ snippets)	86
Table 11	– Number of readability warnings and affected code snippets per sub- category and baseline in the <i>Convention</i> category (<i>Snip.</i> : number of code snippets affected; <i>Warn.</i> : total number of readability warnings). ($n = 327$ snippets)	87

Table 12 – Number of readability warnings and affected code snippets per subcategory and baseline in the <i>Improve Code Modularity</i> category (Snip.: number of code snippets affected; Warn.: total number of readability warnings). ($n = 327$ snippets)	89
Table 13 – Number of readability warnings and affected Code Snippets in ChatGPT and Stack Overflow, Grouped by Syntactic Structure (Snip.: number of code snippets affected; Warn.: total number of readability warnings). ($n = 327$ snippets)	91
Table 14 – Classification of Candidate PRs by Programming Language	109
Table 15 – Classification of Generative Pre-trained Transformer (GPT) 4.1 analysis as annotator	115
Table 16 – Distribution of Java and Python Pull Requests (PR) by category	146
Table 17 – Number of Subcategories for each category by programming language .	147
Table 18 – SonarLint recommended modifications on a Gemini code snippet.	159
Table 19 – Dataset used to extract statistical tests.	161
Table 20 – Total number of readability warnings before refactoring and after correction by each Large Language Models (LLM), along with the percentage reduction relative to the baseline. ($n = 1,109$ readability warnings) . . .	162
Table 21 – Total number of affected code snippets before refactoring and after correction by each LLM, along with the percentage reduction relative to the baseline. ($n = 611$ snippets)	163
Table 22 – Cochran’s Q test and McNemar’s pairwise comparisons among LLMs based on corrected code snippets (i.e., all warnings fixed and no new warnings introduced).	163
Table 23 – Introduced readability warnings by category and LLM (counts of warnings and affected snippets). Totals reflect unique snippets overall and may not equal the sum across categories.	165
Table 24 – Number of readability warnings by status (Corrected, Not Corrected, Introduced) for each SonarLint rule and LLM in the <i>Clean Up</i> category (Gem = Gemini 2.5 Pro, GPT = GPT 4.1, Deep = DeepSeek Coder). .	166
Table 25 – Number of readability warnings by status (Corrected, Not Corrected, Introduced) for each SonarLint rule and LLM in the <i>Clarify Code Intent</i> category (Gem = Gemini 2.5 Pro, GPT = GPT 4.1, Deep = DeepSeek Coder).	168
Table 26 – Number of readability warnings by status (Corrected, Not Corrected, Introduced) for each SonarLint rule and LLM in the <i>Prefer Concise Code</i> category (Gem = Gemini 2.5 Pro, GPT = GPT 4.1, Deep = DeepSeek Coder).	170

Table 27 – Number of readability warnings by status (Corrected, Not Corrected, Introduced) for each SonarLint rule and LLM in the *Convention* category (Gem = Gemini 2.5 Pro, GPT = GPT 4.1, Deep = DeepSeek Coder). 173

Table 28 – Number of readability warnings by status (Corrected, Not Corrected, Introduced) for each SonarLint rule and LLM in the *Improve Code Modularity* category (Gem = Gemini 2.5 Pro, GPT = GPT 4.1, Deep = DeepSeek Coder). 175

Table 29 – Temperature settings for each LLM. 176

Table 30 – Comparative summary of related work and this thesis. 190

List of Abbreviations

AST Abstract Syntax Tree

AI Artificial Intelligence

API Application Programming Interface

ASAT Automatic Static Analysis Tools

ASE International Conference on Automated Software Engineering

AOI Areas of Interest

CI Continuous Integration

EMSE Empirical Software Engineering

ESEM International Symposium on Empirical Software Engineering and Measurement

FSE ACM International Conference on the Foundations of Software Engineering

GPT Generative Pre-trained Transformer

IDE Integrated Development Environment

ICSE International Conference on Software Engineering

ICPC International Conference on Program Comprehension

ICSME International Conference on Software Maintenance and Evolution

IST Information and Software Technology

JLS Java Language Specification

JSS Journal of Systems and Software

JPA Java Persistence API

LLM Large Language Models

LOC Lines of Code

LSTM Long short-term Memory

MCR Modern Code Review

ML Machine Learning

MSR Mining Software Repositories

NMI Narrow meaning identifiers

PR Pull Request

PEP Python Enhancement Proposals

SAM Single Abstract Method

SE Software Engineering

SBES Brazilian Symposium on Software Engineering

SANER International Conference on Software Analysis, Evolution and Reengineering

TDD Test Driven Development

TD Technical Debt

TSE IEEE Transactions on Software Engineering

Contents

1	INTRODUCTION	27
1.1	Motivation	29
1.2	Objectives	31
1.2.1	Thesis Statement	32
1.3	Hyphoteses	32
1.4	Contributions	33
1.5	Publications	34
1.6	Other Academic Contributions	35
1.7	Thesis Outline	36
2	BACKGROUND	37
2.1	Code Readability	37
2.1.1	Definition	37
2.1.2	Standards and Guidelines	38
2.1.3	Improving Code Readability	43
2.1.4	Assessing Code Readability	44
2.2	GitHub Repositories	51
2.2.1	Commits and Branches	51
2.2.2	Code Review	52
2.3	Large Language Models (LLM)	55
2.4	Code Snippets	57
2.4.1	Human-Written Snippets	57
2.4.2	LLM-generated Snippets	58
2.5	Concluding Remarks	60
3	ASSESSING THE READABILITY OF CODE SNIPPETS . .	61
3.1	Context	61
3.2	The Proposed Methodology	63

3.2.1	Selecting Developer Input Queries	64
3.2.2	Code Snippet Extraction from Stack Overflow and LLM Models	65
3.2.3	Assessing Semantic Adherence of Code Snippets with Input Queries	68
3.2.4	Identifying Code Readability Warnings	71
3.2.5	Classifying Types of Code Readability Rules in SonarLint	72
3.2.6	Answering the Research Questions	74
3.3	Results and Findings	75
3.3.1	Number of Readability Warnings	75
3.3.2	Code Metrics	76
3.3.3	Readability Comparison Under Types of Warnings	80
3.3.4	Readability Comparison in Syntactic Elements Categorization	90
3.3.5	Assessing the Evolution of Code Readability Between ChatGPT 2023 and 2025	92
3.4	Discussion of the Results	93
3.5	Insights and Implications	97
3.5.1	Researchers	97
3.5.2	Practitioners	98
3.6	Threats to Validity	98
3.6.1	Construct Validity	98
3.6.2	Internal Validity	100
3.6.3	External Validity	100
3.7	Concluding Remarks	101
4	A TAXONOMY OF CODE READABILITY IMPROVEMENTS PERFORMED BY DEVELOPERS	103
4.1	Context	103
4.2	The Proposed Methodology	104
4.2.1	Mining Candidate Pull Requests	105
4.2.2	Selecting Readability-focused PRs from Candidates	107
4.2.3	Collecting the Types of Code Readability Improvements	109
4.2.4	Classifying the Types of Code Readability Improvements	113
4.3	Results and Findings	117
4.3.1	Taxonomy of Code Readability Improvements	118
4.3.2	Clarify Code Intent	118
4.3.3	Prefer Concise Code	127
4.3.4	Clean Up	135
4.3.5	Improve Code Modularity	137
4.3.6	Improve Documentation	141
4.3.7	Improve Code Formatting	142
4.3.8	Summary	146

4.4	Discussion of the Results	147
4.4.1	Comparing Subcategories in Java and Python	147
4.4.2	Linters Coverage of Readability Improvements	148
4.4.3	Use of GPT as an Annotator	149
4.5	Implications for Developers and Researchers	150
4.5.1	Researchers	150
4.5.2	Developers and Practitioners	150
4.6	Threats to Validity	151
4.6.1	Construct Validity	151
4.6.2	Internal Validity	152
4.6.3	External Validity	152
4.7	Concluding Remarks	152
5	AUTOMATED CODE READABILITY IMPROVEMENTS .	155
5.1	Context	155
5.2	The Proposed Methodology	157
5.2.1	Selecting Affected Code Snippets	157
5.2.2	Prompting LLMs for Code Readability Improvements	158
5.2.3	Statistical Tests	161
5.3	Results and Findings	161
5.3.1	Number of Readability Warnings	162
5.3.2	Number of Affected Code Snippets	162
5.3.3	Introduction of New Readability Warnings	164
5.3.4	Category Analysis	164
5.4	Discussion of the Results	175
5.5	Implications for Developers and Researchers	177
5.5.1	Researchers	177
5.5.2	Developers and Practitioners	177
5.6	Threats to Validity	177
5.6.1	Construct validity	178
5.6.2	Internal Validity	178
5.6.3	External Validity	178
5.7	Concluding Remarks	178
6	RELATED WORK	181
6.1	Understanding Developer Practices	181
6.1.1	Developers Perceptions	182
6.1.2	Developers Motivations	183
6.1.3	Developers Improvements	183
6.2	Code Readability Models	184

6.2.1 Readability Model Applications 184

6.2.2 Readability Models Limitations 185

6.3 Static Analysis Tools 186

6.3.1 Static Analysis Tools on Empirical Studies 186

6.3.2 Static Analysis tools combined with Neural Models 187

6.4 Mining GitHub Pull Requests 187

6.5 Large Language Models in Software Engineering Research . . . 188

6.5.1 LLM as an Annotator 189

6.6 Synthesis 190

7 CONCLUSIONS 191

7.1 Future Directions 193

BIBLIOGRAPHY 195

Introduction

Developers often have to work with source code written by others. This can occur in various ways, such as searching for reusable code snippets, that is, short fragments of source code that developers can reuse to solve specific programming tasks (KEIVANLOO; RILLING; ZOU, 2014; XIA et al., 2017), or in situations where a developer needs to improve a feature or correct a system bug originally implemented by another developer. In essence, software evolves through the accumulation of incremental changes (KEMERER, 1995; BENESTAD; ANDA; ARISHOLM, 2009; BENNETT; RAJLICH, 2000).

In this context, several desirable characteristics are expected from software to prevent incremental changes from becoming problematic. For example, software should be flexible to support evolution and minimize modifications to the existing system (BREIVOLD; CRNKOVIC; LARSSON, 2012). It should also be easy to test, enabling comprehensive test coverage and reducing the risk of bugs (ATHANASIOU et al., 2014). Most importantly, software should be easy to read and understand (GOSAVI; RAJLICH, 2004), as higher readability contributes to fewer errors, easier maintenance (AGGARWAL; SINGH; CHHABRA, 2002), faster comprehension, and greater consistency (HANEED, 1998).

Therefore, readability is a highly desirable characteristic in software evolution, as developers spend a significant amount of time reading and understanding code (MINELLI; AND; LANZA, 2015; TIARKS, 2011), particularly when evolving existing software often written by other developers (ERLIKH, 2000). Due to its importance for project maintainability, prior studies suggested adding a development phase in Software Engineering (SE) to assess and improve code readability (ELSHOFF; MARCOTTY, 1982; KNIGHT; MYERS, 1991).

The lack of code readability can motivate developers to perform refactoring operations to make the code easier for others to understand. Some of the motivations for improving readability have already been investigated. Previous studies have classified motivations for implementing refactoring operations, and some of them are closely related to improving code readability, such as renaming classes, attributes, methods, and other elements (PANTIUCHINA et al., 2020).

However, software projects such as those hosted on the GitHub repositories (GITHUB, 2025a) often contain a large amount of unreadable code (PIANTADOSI et al., 2020). Given the importance of this issue, several readability models have been proposed. Some of these models aim to assess whether the code is readable or unreadable (BUSE; WEIMER, 2010; POSNETT; HINDLE; DEVANBU, 2011; DORN, 2012; SCALABRINO et al., 2016; SCALABRINO et al., 2018; ROY et al., 2020), while others focus on automatically improving code readability based on predefined types of change (VITALE et al., 2023) or rules defined by Automatic Static Analysis Tools (ASAT), such as linters (LORIOT; MADEIRAL; MONPERRUS, 2022). In particular, linters such as SonarLint (SONARLINT, 2025b) are capable of detecting code smell issues in source code (CALIXTO; ARAÚJO; ALVES, 2024), including those directly related to the lack of code readability (BELLER et al., 2016).

Although significant progress has been made with automated tools, assessing code readability remains a challenge. For example, while developers generally acknowledge the relevance of linters for overall software improvement (ROMANO et al., 2022), previous studies show that only 13% of the issues reported by these tools are actually resolved in practice (MARCILIO et al., 2019), and some improvements made by developers are not detected (OLIVEIRA et al., 2025). Those tools also output false positive warnings (LENARDUZZI et al., 2023). Moreover, some readability models, particularly those developed in controlled settings, do not consistently detect whether readability has improved in real-world projects, revealing a gap between model predictions and the kinds of improvements made by developers (PANTIUCHINA; LANZA; BAVOTA, 2018; FAKHOURY et al., 2019; PIANTADOSI et al., 2020). Even models trained on real-world code have shown limitations, incorrectly classifying 98 out of 297 readability improvements (33%) (ROY et al., 2020). Even models that propose to automatically improve code readability often operate on a limited subset of code readability improvements, such as adding comments, renaming identifiers, or reorganizing code blocks (VITALE et al., 2023). These challenges may remain due to the complex syntax and semantics of the source code (BUSJAHN et al., 2015), or due to the difficulty of establishing reliable correlations between code readability and software metrics (SCALABRINO et al., 2017; SANTOS; GEROSA, 2018).

Given these challenges, new opportunities have emerged through the rapid adoption of LLM. Prominent examples include the GPT family of models (e.g., GPT-3, GPT-4) and applications built on top of them, such as ChatGPT (CHATGPT, 2025). These LLMs have gained significant attention among developers, as evidenced by numerous mentions of ChatGPT in GitHub commits, Pull Requests (PR), and even comments on source code where the solutions provided by ChatGPT were adopted and provided assistance to developers (XIAO et al., 2024; TUFANO et al., 2024; SILVA; DANTAS; MAIA, 2024). These LLMs also have the potential to contribute by performing refactoring operations on code (GUO et al., 2024) and helping developers understand it (NAM et al., 2024).

1.1 Motivation

As a motivational example, a LLM can be used in conjunction with existing tools, such as linters, to automatically improve code readability. Linters are known to display warnings, but are limited to issues explicitly defined by coded rules, as they lack the ability to understand the context of the code or the meaning behind it (DANTAS; ROCHA; MAIA, 2023a; OLIVEIRA et al., 2025). In contrast, LLMs can understand the context of the code to perform refactoring operations and provide feedback in natural language (GUO et al., 2024). However, they are not always accurate. They may hallucinate (ZUCCON; KOOPMAN; SHAIK, 2023), miss simple errors, or ignore coding standards if not properly guided (JAOUA; SGHAIER; SAHRAOUI, 2025).

For example, given the input query “Traipse a directory tree in Java?”, CROKAGE (CROKAGE, 2025), a tool that provides both code snippets and their corresponding explanations for each query, recommended a Stack Overflow snippet containing the code in Listing 1.1:

```
1 public static void walkin(File dir) {
2     File listFile[] = dir.listFiles();
3     if (listFile != null) {
4         for (int i = 0; i < listFile.length; i++) {
5             if (listFile[i].isDirectory()) {
6                 System.out.println("|\\t\\t");
7                 walkin(listFile[i]);
8             } else {
9                 System.out.println("+---" + listFile[i].getName()
10                    .toString());
11             }
12         }
13 }
```

Listing 1.1 – Stack Overflow Recommended Code Snippet

The SonarLint tool analyzed the recommended code snippet and identified the warnings listed in Table 1. Specifically, it detected that the array designators [] should be placed next to the type, following Java’s coding conventions for clarity, and that the call to `toString()` on the result of `getName()` is unnecessary.

These recommendations from SonarLint could be used as a prompt for the LLMs, specifically asking for automatic code refactoring driven by SonarLint instructions. A context is defined through the “*system*” prompt, as shown in the text box below, where [code_snippet] refers to the code snippet to improve readability, and [sonarlint_suggestions] contains the recommendations provided by SonarLint.

Table 1 – SonarLint recommended modifications on a code snippet.

Rule Key	Start Line	End Line	Message
java:S1197	2	2	Move the array designators [] to the type.
java:S1858	9	9	getName() returns a String; there is no need to call toString().

Prompt Templates used for readability improvements

System: "You are a software developer specialized in the Java programming language, skilled in performing refactoring operations, such as improving the code based on SonarLint recommendations."

User: "Based on the following Java code snippet: [code_snippet]. Sonarlint suggested the following refactoring operations: [sonarlint_suggestions]. Please provide a new version of this code snippet with the refactoring operations performed, as recommended by Sonarlint."

The DeepSeek Codr model generated the following code. Both SonarLint warnings were addressed as requested, and the rest of the code snippet remained unchanged, as shown in Listing 1.2:

```

1 public static void walkin(File dir) {
2     File[] listFile = dir.listFiles();
3     if (listFile != null) {
4         for (int i = 0; i < listFile.length; i++) {
5             if (listFile[i].isDirectory()) {
6                 System.out.println("|\\t\\t");
7                 walkin(listFile[i]);
8             } else {
9                 System.out.println("+---" + listFile[i].getName()
10                    );
11             }
12         }
13     }

```

Listing 1.2 – Stack Overflow Code Snippet changed by DeepSeek Codr model

When DeepSeek Codr is asked without context, using general queries such as *"Can you improve the readability of this code?"*, the response may include only partial readability improvements, such as adding comments to the method or renaming variables that were not intended to be changed. Although some of these refactorings might be useful, unguided experiments could be more difficult to evaluate in terms of effectiveness

when compared to changes guided by linter suggestions. For example, the LLM might hallucinate and generate undesirable code.

1.2 Objectives

This work proposes characterizing code readability from two complementary perspectives: (i) how code is suggested or generated by tools, and (ii) how developers actually modify code to improve its readability. The first perspective focuses on evaluating the readability of code suggested in online platforms such as Stack Overflow or by automated tools such as LLMs, based on input queries or prompts. It also examines whether LLMs can solve readability warnings according to linter-based recommendations. The second perspective investigates the types of readability improvements developers actually apply in practice, by analyzing incremental code changes in real-world repositories.

Figure 1 illustrates the general scope of this work. On the left, we analyze the recommendations made by tools and how they affect code readability. On the right, we observe how developers act in real-world projects to improve readability. This dual characterization provides information about both the recommendations offered by automated tools and the actual practices adopted by developers, helping bridge the gap between theoretical suggestions and practical refactorings.

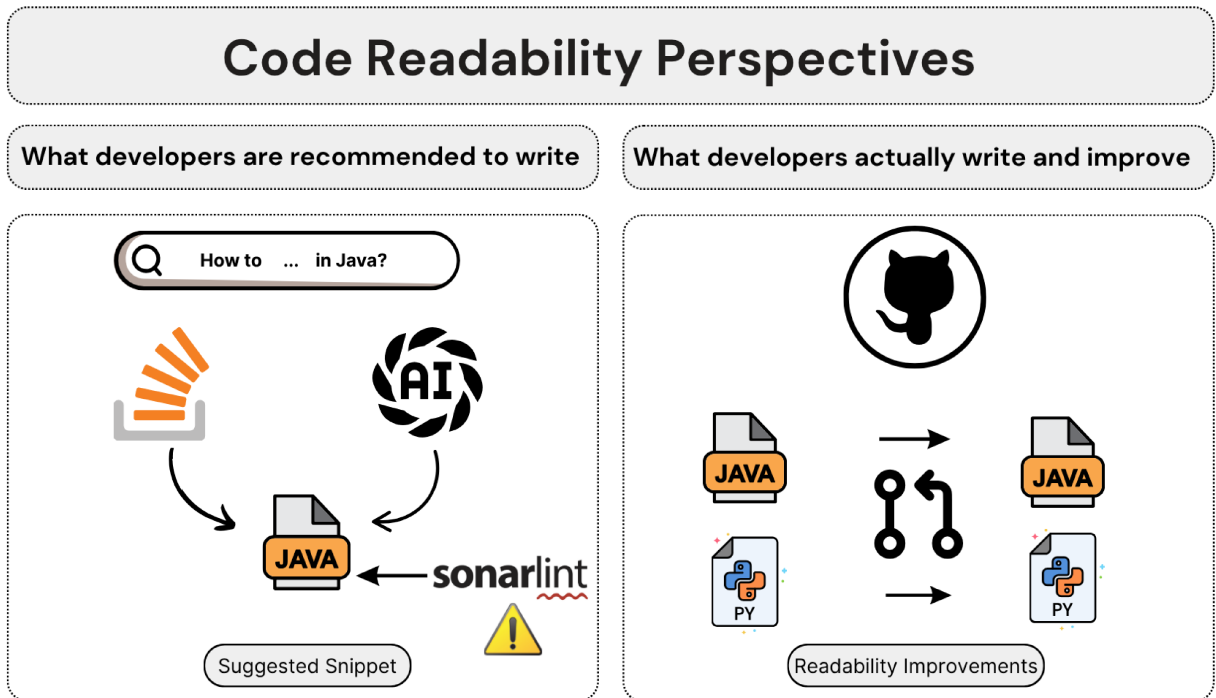


Figure 1 – Code Readability Perspectives

The objective of this thesis can be divided into five specific goals, which collectively contribute to achieving its overall purpose, as follows:

1. Compare the readability of code snippets generated by LLMs with those written by humans and recommended on Stack Overflow.
2. Assess the capability of LLMs to automatically improve code readability based on the linter recommendations.
3. Identify and classify the types of code readability improvements performed by developers in incremental changes.
4. Evaluate the types of readability improvements applied in different programming languages, such as Java and Python. Comparing readability improvements across these two widely used programming languages helps reveal both common patterns and language-specific practices.
5. Analyze potential gaps between the recommended readability improvements (from LLMs and linters) and the changes that developers actually apply in real-world repositories.

1.2.1 Thesis Statement

A hybrid approach combining LLMs and ASATs improves code readability by not only using the ability of LLMs to generate code with fewer readability warnings than human-written examples from Stack Overflow, but also using ASATs to eliminate residual warnings. This combination proved particularly effective with the DeepSeek Coder, a model specifically trained for code generation. Additionally, a taxonomy built from Java and Python repositories, describing strategies such as clarifying intent, eliminating redundancies, and reducing verbosity, provides an empirical basis for characterizing human refactoring practices. Although some improvements are captured by linter rules, these rules lack semantic depth, highlighting the importance of prompt design and suggesting that combining ASATs rules, patterns from real-world repositories and language conventions may enhance prompt effectiveness.

1.3 Hypotheses

In this section, four hypotheses are presented and investigated in the following chapters.

H1 The characterization of code readability improvements performed by developers in PRs from real-world GitHub (GITHUB, 2025a) repositories reveals readability perceptions that differ from those captured by recommendations made by automated tools such as ASATs, thus exposing potential gaps between human judgment and tool-based evaluation. Previous work has already characterize the motivations to

implement refactoring operations in GitHub, and some of them are related to improvements in readability (PANTIUCHINA et al., 2020).

- H2** Certain programming languages may present unique types of readability improvements, while differences in programming paradigms can reflect different motivations of the developer. Previous studies have investigated language-specific characteristics in software engineering, such as atoms of confusion in Java (LANGHOUT; ANICHE, 2021), Python (COSTA et al., 2023), Javascript (TORRES et al., 2023), Swift (CASTOR, 2018), and other languages (GOPSTEIN et al., 2017), as well as developer perceptions of code smells in Java (PALOMBA et al., 2014) and Elixir (VEGI; VALENTE, 2022).
- H3** Code snippets generated by LLMs could exhibit significantly different readability levels compared to the human-written solutions recommended on Stack Overflow. Previous work has already shown that model-generated code could be comparable in complexity and readability to code written by human pair programmers (MADI, 2023).
- H4** LLMs may be able to automatically improve code readability based on the recommendations provided by the linters. This combination of LLM and ASAT under the same model is already proposed in previous work, but to improve the code review process (JAOUA; SGHAIER; SAHRAOUI, 2025)

1.4 Contributions

This section describes the following contributions of this work.

1. A taxonomy of code readability improvements performed by developers in real-world Java and Python Github repositories.
2. A taxonomy of code readability rules recommended by linters such as SonarLint (SONARLINT, 2025b).
3. A comparative study between the readability of code snippets generated by LLMs such as ChatGPT, DeepSeek and Gemini with those written by developers on platforms such as Stack Overflow.
4. A novel approach that combines ASAT code readability recommendations with LLM-based generation to automatically refactor the code and improve its readability.
5. A comprehensive review of the literature on code readability.

6. A replication package containing all the scripts and data used to produce this thesis in (DANTAS; MAIA, 2025).

1.5 Publications

This thesis has resulted in the following publications, whose content has been integrated into its chapters.

1. Carlos Eduardo C. Dantas, Adriano M. Rocha and Marcelo A. Maia. **Assessing the Readability of ChatGPT Code Snippet Recommendations: A Comparative Study**. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering (SBES)*, 2023. *Awarded Paper — 21 citations* (GOOGLESCHOLAR, 2025a).

Article 1 proposes a comparative analysis of the readability of code snippets recommended by Stack Overflow through the CROKAGE search engine and those generated by ChatGPT. The results of this analysis, along with its extensions, are presented in Chapter 3.

2. Carlos Eduardo C. Dantas, Adriano M. Rocha and Marcelo A. Maia. **How do Developers Improve Code Readability? An Empirical Study of Pull Requests**. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2023, pp. 110–122. *7 citations* (GOOGLESCHOLAR, 2025c).

Article 2 proposes a taxonomy of the types of code readability improvements performed by developers in GitHub PRs. It also evaluates how these improvements align with recommendations made by SonarLint (SONARLINT, 2025b). The results of this article are presented in Chapters 4 and 5, although both chapters extend beyond the scope of this publication.

3. Carlos Eduardo C. Dantas and Marcelo A. Maia. **Readability and Understandability Scores for Snippet Assessment: An Exploratory Study**. In *9th Workshop on Software Visualization, Evolution and Maintenance (VEM)*, 2021. *14 citations* (GOOGLESCHOLAR, 2025f).
4. Carlos Eduardo C. Dantas and Marcelo A. Maia. **Readability and Understandability of Snippets Recommended by General-purpose Web Search Engines: A Comparative Study**. In *1st International Workshop on Automated Support to Improve Code Readability (AeSIR)*, 2021. *1 citation* (GOOGLESCHOLAR, 2025e).

Articles 3 and 4 involve the use of readability and understandability tools, such as Scalabrino’s model (SCALABRINO et al., 2018) and cognitive complexity (CAMPBELL, 2018), to assess how readable and understandable the code snippets recommended by CROKAGE and Google are compared to those from sites such as Stack Overflow. Although the results of these articles are not discussed in this thesis, the tools served as inspiration to adopt the readability and understandability metrics in the thesis.

5. Julyanara R. Silva, Carlos Eduardo C. Dantas, and Marcelo A. Maia. **What Developers Ask to ChatGPT in GitHub Pull Requests? An Exploratory Study.** In *Proceedings of the XII Workshop on Software Visualization, Evolution and Maintenance (VEM)*, Curitiba/PR, Brazil, SBC, 2024, pp. 125–136. *1 citation Awarded Paper.*

Article 5 presents an analysis of how developers are using ChatGPT to submit code changes through GitHub PRs. Although the results of this article are not included in the thesis, it reinforces the growing relevance and extensive use of LLMs in the daily activities of software developers.

6. Rodrigo F. Silva, Mohammad Masudur Rahman, Carlos Eduardo C. Dantas, Chanchal Roy, Foutse Khomh, and Marcelo A. Maia. **Improved Retrieval of Programming Solutions with Code Examples Using a Multi-featured Score.** *Journal of Systems and Software*, 181 (2021), 111063. *4 citations* (GOOGLESCHOLAR, 2025d).
7. Rodrigo F. Silva, Chanchal K. Roy, Mohammad M. Rahman, Kevin A. Schneider, Klérisson V. R. Paixão, Carlos Eduardo C. Dantas, and Marcelo A. Maia. **CROKAGE: Effective Solution Recommendation for Programming Tasks by Leveraging Crowd Knowledge.** *Empirical Software Engineering*, 25(6), 2020, pp. 4707–4758. *23 citations* (GOOGLESCHOLAR, 2025b).

Articles 6 and 7 propose the CROKAGE and CRAR search engines. These works are referenced not only because the author of this thesis slightly contributed to the development of these engines but also because the results produced by them served as motivation to evaluate the quality of the recommended code snippets, as discussed in Chapter 3.

1.6 Other Academic Contributions

During the Ph.D. period, the author also contributed to the academic community in the following ways:

- Member of the ICPC 2026 Organizing Committee serving as Web Co-Chair

- ❑ Research advisor for a high school student project that was awarded Best Paper at the XII Workshop on Software Evolution and Maintenance (VEM), part of CBSoft, 2024.
- ❑ Research advisor for three research papers accepted at CBSoft 2025, including two high school students and one by an undergraduate student finalist in the 6th Scientific Initiation Papers Competition of the Brazilian Symposium on Software Engineering (CTIC-ES).
- ❑ Reviewer for the journal *Transactions on Software Engineering* (Springer), 2023.
- ❑ Served as Program Committee Member for the Artifacts Festival, co-located with CBSoft 2025.
- ❑ Student Volunteer at the 39th International Conference on Software Maintenance and Evolution (ICSME), 2023.
- ❑ Student Volunteer at the XIII Congresso Brasileiro de Software (CBSoft), 2022.

1.7 Thesis Outline

This thesis is organized into the following chapters.

- ❑ **Chapter 2** introduces the fundamental concepts and theories that support the proposed research, including definitions, models, and background on code readability and related tools.
- ❑ **Chapter 3** investigates how readable the code snippets generated by automated approaches such as LLMs, compared to human-written snippets from Stack Overflow, as recommended by CROKAGE.
- ❑ **Chapter 4** proposes a taxonomy of the types of changes that Java and Python developers perform to improve code readability, based on an analysis of GitHub PRs.
- ❑ **Chapter 5** evaluates the extent to which LLMs are able to automatically improve code readability based on ASAT recommendations.
- ❑ **Chapter 6** provides an overview of the related work, focusing on the main strategies proposed to improve code readability and the application of automated tools in this context.
- ❑ **Chapter 7** summarizes the findings and highlights future perspectives on code readability.

Background

This chapter presents essential concepts to support the methodology in answering research questions.

2.1 Code Readability

This section presents definitions of code readability, followed by standards and guidelines that help define how to improve it. It also discusses how code readability can be assessed using readability models and their associated metrics, ASATs, and even LLMs. Code readability requires a specific definition, as it differs from natural language readability and involves a non-linear reading process (BUSJAHN et al., 2015).

2.1.1 Definition

Buse and Weimer (2010) define code readability as a human judgment of how easy code is to understand. This definition emphasizes that perceptions may vary, that is, what is easy to read for one developer may not be easy for another. Readability is closely related to several related concepts, including complexity, legibility, understandability, comprehensibility, and simplicity.

As mentioned by Buse and Weimer (2010), readability is not the same as complexity. Brooks Jr (1987) distinguishes between essential and accidental properties of software: the first is due to the inherent difficulty of the problem, while the latter arises from the tools and practices used in development. In this view, readability is an accidental property and can be improved through better coding practices. Oliveira et al. (2020) define legibility as the ease of identifying elements of a program, and readability as the combination of factors that make a program easier or harder to read. Posnett, Hindle and Devanbu (2011) distinguish between readability and understandability by making an analogy with syntactic and semantic analysis, that is, readability corresponds to syntactic aspects, while understandability refers to semantic aspects. Semantic understanding typically involves

the recognition of statements, beacons, or motifs in the code (SCALABRINO et al., 2017). As Boehm, Brown and Lipow (1976) notes, if the code is difficult to read, it is also difficult to understand. Rambally (1986) compares readability with comprehensibility, defining readability as the ease of reading and understanding the code, while comprehensibility involves the ease of maintaining, testing, and modifying it. Börstler, Caspersen and Nordström (2015) considers readability a prerequisite for understandability, highlighting that syntactical elements should be easy to recognize. He also suggested that simplicity may be a feature of readability.

Despite the nuanced distinctions made in the literature, developers do not always perceive these concepts in the same way. Some studies use keywords such as "*comprehension*", "*comprehensible*", "*understand*", "*understanding*" and "*clean up*" to identify commits related to code readability improvements and build readability-focused datasets (FAKHOURY et al., 2019). Readability may also involve semantic concerns, such as reducing nested loops (JOHNSON et al., 2019). As Scalabrino et al. (2018) observe, readability remains a subjective concept influenced by several factors, including complexity, design choices, formatting, code lexicon, and visual elements such as syntax styling (SCALABRINO et al., 2018).

Based on these prior definitions, we conceptualize code readability as a primarily syntactic property, influencing the extent to which source code can be understood and its behavior anticipated without the need for compilation or execution (OLIVEIRA et al., 2020; POSNETT; HINDLE; DEVANBU, 2011). However, as highlighted by Buse and Weimer (BUSE; WEIMER, 2010), readability entails a degree of subjectivity, reinforcing the need to contrast human judgments with tool-based recommendations.

2.1.2 Standards and Guidelines

The definition of code readability, by itself, does not specify which characteristics make code easier to read and understand. Although there is no formal documentation listing all these characteristics, several standards have been established either by the developer community (MARTIN, 2008; HOPKINS et al., 2008) or by the maintainers of programming languages (ROSSUM; HETTINGER, 2025; ORACLE, 2025; GOOGLE, 2025). Although these standards are not always created with readability as their primary focus, many of them are closely related to or have an impact on code readability (SANTOS; GEROSA, 2018; SMIT et al., 2011).

2.1.2.1 Python Guidelines

As mentioned, developers may have different perceptions of what constitutes readable code, as perception is highly individual (BOFF; KAUFMAN; THOMAS, 1986). To help align these perceptions, the Python community introduced the Python Enhancement Pro-

posals (PEP) (PYTHON, 2025), which are design documents that describe new features or conventions for the language. Some of these proposals focus specifically on improving code readability by establishing guidelines for aspects such as coding style and formatting (ROSSUM; HETTINGER, 2025), docstrings (GOODGER; ROSSUM, 2025), and the use of type annotations (ROSSUM; LEHTOSALO, 2014).

For example, PEP 8 defines conventions to enforce consistency, such as using four spaces for each level of indentation, limiting lines to 79 characters, adopting the *snake_case* pattern for variable and function names, inserting two blank lines before top-level functions and classes, and one blank line between methods within a class, among other conventions (ROSSUM; HETTINGER, 2025). PEP 20 (The Zen of Python) reinforces the importance of code clarity in the Python community through the principle "*Readability counts*" (PETERS, 2004).

Although PEP 8 provides guidelines for improving code readability in Python, many of its recommendations lacked empirical validation for a long time. However, recent studies have started to assess the actual impact of these guidelines on developer understanding. For example, a controlled eye-tracking experiment with 32 Python novices revealed that violations of the line break after an operator rule led to a 70% increase in eye regressions, indicating a higher cognitive load and reduced readability (OLIVEIRA et al., 2024). These findings support the idea that style conventions such as those in PEP 8 can help with code understanding. However, Pythonic idioms (ALEXANDRU et al., 2018), which can include functional constructs that have been reported to be potentially difficult to understand (ZID et al., 2024).

2.1.2.2 Java Guidelines

Java also has a set of conventions and guidelines, as defined by the programmers at Oracle (ORACLE, 2025) and Google (GOOGLE, 2025). Some of them explicitly mention readability in their descriptions. For example, Google programmers specify that a single blank line may be used to separate logically related code blocks, which helps improve readability by visually organizing the structure. They also recommend using grouping parentheses to improve readability, as relying on readers to remember operator precedence can cause misinterpretations. The Listing 2.1 illustrates an example.

```
1    boolean result = a && b || c; // possible misinterpretation
2    boolean result = (a && b) || c; //readable
```

Listing 2.1 – Java convention on grouping parentheses

Oracle programmers mention that code conventions improve the readability of the software, allowing developers to understand new code more quickly and with better understanding (ORACLE, 2025). They specify, for example, that each line length should not

be longer than 80 characters, or naming conventions make programs more understandable by making them easier to read. The Listing 2.2 illustrates an example.

```
1      int TotalVALUE;  
2      int totalValue;
```

Listing 2.2 – Java naming convention

The first name uses uppercase inconsistently, mixes naming styles (camelCase and UPPERCASE), and starts with an uppercase letter, which is conventionally reserved for class names. The second name uses lowercase for the first letter, follows the camelCase convention consistently, and provides a descriptive name.

2.1.2.3 Community Standards

The software development community also proposes coding standards, such as the nine rules of *Object Calisthenics* introduced by Jeff Bay (HOPKINS et al., 2008). As the author states, there are seven code qualities commonly recognized as important: cohesion, loose coupling, zero duplication, encapsulation, testability, readability, and focus. The proposed rules provide guidance for developers to apply these principles more effectively in code development.

1. *Use only one level of indentation per method:* the author recommends avoiding nested control structures in a single method. Instead, the methods should do exactly one thing. Listings 2.3 and 2.4 illustrates the **Board** class in two code snippets. The code snippet in Listing 2.3 contains nested control structures in the **board()** method. The code snippet in Listing 2.4 changes the logic by introducing the methods **collectRows()** and **collectRow()** to separate responsibilities between smaller and cohesive methods.

```
1      class Board {  
2          String board() {  
3              StringBuffer buf = new StringBuffer();  
4              for(int i = 0; i < 10; i++) {  
5                  for(int j = 0; j < 10; j++)  
6                      buf.append(data[i][j]);  
7                      buf.append("\n");  
8              }  
9              return buf.toString();  
10         }  
11     }
```

Listing 2.3 – Code Snippet containing nested control structures

```
1      class Board {
2          String board() {
3              StringBuffer buf = new StringBuffer();
4              collectRows(buf);
5              return buf.toString();
6          }
7
8          void collectRows(StringBuffer buf) {
9              for(int i = 0; i < 10; i++)
10                 collectRow(buf, i);
11          }
12
13          void collectRow(StringBuffer buf, int row) {
14              for(int i = 0; i < 10; i++)
15                 buf.append(data[row][i]);
16              buf.append("\n");
17          }
18      }
```

Listing 2.4 – Code Snippet extracting methods to separate responsibilities

2. *Don't use the **else** keyword:* This rule promotes the use of early return, encouraging developers to avoid unnecessary **else** blocks, especially in cases where nested conditionals become difficult to follow or when dealing with long **switch**-like structures. Listings 2.5 to 2.8 shows four code snippets. In code snippet of Listing 2.5, if the first condition is true, the **else** block ensures that a specific instruction is skipped. In contrast, the example in Listing 2.6 applies early return, that is, if the condition is true, the method returns immediately, making the use of **else** unnecessary. The same principle applies to the second pair of snippets Listing 2.7 and 2.8, which uses ternary operators to avoid redundant conditional structures.

```
1      public static void endMe() {
2          if (status == DONE) {
3              doSomething();
4          } else {
5              <other code>
6          }
7      }
```

Listing 2.5 – Code Snippet using else block

```
1      public static void endMe() {
2          if (status == DONE) {
```

```
3         doSomething();
4         return;
5     }
6     <other code>
7 }
```

Listing 2.6 – Code Snippet using the return early method

```
1     public static Node head() {
2         if (isAdvancing()) { return first; }
3         else { return last; }
4     }
```

Listing 2.7 – Code Snippet using else block

```
1     public static Node head() {
2         return isAdvancing() ? first : last;
3     }
```

Listing 2.8 – Code Snippet using ternary operations

3. *Don't abbreviate*: refers to avoid abbreviated names of classes, methods, or variables.

Robert C. Martin also introduced a set of principles focused on writing cleaner code, several of which offer important guidance on code readability (MARTIN, 2008). Examples include.

1. Without proper indentation, the programs would be virtually unreadable to humans.
2. The names in the software are responsible for 90% of its readability. Previous studies have proposed approaches to recommend more appropriate names for code elements (ALLAMANIS et al., 2014), (LIN et al., 2017), (THIES; ROTH, 2010).
3. The readability of an entire module can be improved by reducing code duplication.
4. One of the most effective ways to make a program more readable is to break complex calculations into intermediate values stored in variables with meaningful names.
5. The use of generics can significantly improve code readability.
6. Readability is perhaps even more important in unit tests than in production code.

2.1.3 Improving Code Readability

In addition to following language-specific conventions or community style guides, developers frequently perform refactoring operations, a form of structured code transformation, to improve readability and maintainability.

Refactoring is the process of improving the design quality of object-oriented software while preserving its external behavior (OPDYKE, 1992). An increased volume of refactoring operations typically results in improved software maintainability (SZÓKE et al., 2014). Fowler provides comprehensive documentation on dozens of refactorings (FOWLER, 1999). His catalog includes the name of each refactoring, the steps required to perform the transformation, and illustrative examples that demonstrate the intended changes.

Developers can identify structural changes in code using refactoring detection tools such as RefactoringMiner (TSANTALIS et al., 2018) which automatically identify the types of refactoring applied to the code. Version 3.0.10 of RefactoringMiner (REFACTORINGMINER, 2025) is capable of detecting 103 different refactoring operations in Java programs.

Some of the catalog of refactorings presented by Fowler (FOWLER, 1999) identified in the subsequent results of this study is described below.

1. *Extract Method*: Turn a code fragment, usually from a long method, into a separate method whose name clearly explains its purpose.
2. *Rename Variable/Attribute/Method/Parameter*: Rename a variable, class attribute, method, or parameter when its name does not convey its purpose. Names should communicate the intent behind their use.
3. *Change Variable/Attribute/Return/Parameter/Thrown Exception Type*: Change the type of a variable, class attribute, method return, parameter or thrown exception. This often involves replacing a concrete type with a superclass or interface to enable polymorphism.
4. *Extract Variable*: Extract an expression (or a part of it) into one or more variables to make the code more self-explanatory and easier to understand.
5. *Remove Parameter*: Remove a parameter that is no longer used in the method body. This simplification improves the readability of the method and reduces unnecessary complexity. This type of refactoring may also include: *Remove Method Annotation*, *Remove Variable Modifier*, *Remove Parameter Annotation*, *Remove Thrown Exception Type*, *Remove Method Modifier*, *Remove Attribute Annotation*, *Remove Attribute Modifier*, *Remove Class Annotation*.

6. *Parameterize Variable*: Convert a hardcoded variable in a method into a parameter, allowing the method to receive a value externally and become more reusable.
7. *Add Attribute/Variable/Parameter/Method Modifier*: Add a Java modifier keyword (e.g., *public*, *private*, *protected*, *final*, *static*, *transient*, *volatile*, *abstract*, *synchronized*) to an attribute, variable, method parameter, or method signature to explicitly define its behavior.
8. *Replace Anonymous with Lambda*: Replace anonymous classes with lambda expressions to make the code more concise.
9. *Replace Loop with Pipeline*: Replace traditional *for* or *while* loops with Java *stream API* to express data transformations in a more declarative way.
10. *Inline Method*: Remove a method whose body is so simple or obvious that it does not add any meaningful abstraction. The method calls are replaced by the actual content of the method.
11. *Inline Variable*: Remove a variable that holds a simple expression and replace its usages directly with the expression, reducing unnecessary indirection.

2.1.4 Assessing Code Readability

To assess whether a code snippet needs improvements in readability, developers can rely on tools that classify the code snippet as readable or unreadable, or provide specific suggestions on how to improve the code readability, or even propose automatically refactoring a code snippet aimed at improving readability.

2.1.4.1 Readability Models

Readability models are designed to assess the readability of source code, supporting goals such as improving code maintainability (KNIGHT; MYERS, 1991). Their purpose is analogous to the readability metrics in natural languages. For example, the *Flesch-Kincaid* index integrated into tools such as Microsoft Word, which estimates readability based on the average syllables per word and the average sentence length (FLESCH, 1948). In programming languages, readability models typically aim to assess whether a given code snippet is more likely to be readable or unreadable, based on features extracted from the source code. These features may include syntactic structure, lexical choices, and code formatting. The following subsections present representative readability models and the features they use for classification.

Buse and Weimer

The model proposed by Buse and Weimer (2010) was trained using a dataset of 100 code snippets (with an average length of 7.7 lines), evaluated by 120 developers, resulting in 12,000 judgments collected using a Likert scale (LIKERT, 1932). Snippets with an average rating below 3.14 were classified as less readable, while the remaining were labeled as more readable. A set of token-level features, summarized in Table 2, was extracted from the snippets and used in the training phase of a classifier. The authors used a logistic regression model to predict the readability of new code snippets. The most influential features in the model include average and maximum line length, average number of identifiers per line, average number of parentheses, and average number of periods.

Table 2 – Token-level features used in the Buse and Weimer readability model.

Feature	Average	Maximum
Line length (characters)	Yes	Yes
Number of identifiers	Yes	Yes
Identifier length	Yes	Yes
Indentation (preceding whitespace)	Yes	Yes
Number of keywords	Yes	Yes
Number of numbers	Yes	Yes
Number of comments	Yes	No
Number of periods	Yes	No
Number of commas	Yes	No
Number of spaces	Yes	No
Number of parentheses	Yes	No
Number of arithmetic operators	Yes	No
Number of assignments (=)	Yes	No
Number of branches (if)	Yes	No
Number of loops (for, while)	Yes	No
Number of blank lines	Yes	No
Occurrences of any single character	No	Yes
Occurrences of any single identifier	No	Yes

Posnett and Colleagues

The model proposed by Posnett, Hindle and Devanbu (2011) uses the same data set constructed for the Buse and Weimer study. The main objective was to outperform Buse and Weimer’s model by relying on a simpler set of features. These features include: (i) Halstead’s vocabulary metrics (total operators, unique operators, total operands and unique operands) (HALSTEAD, 1977), (ii) token-level entropy, and (iii) Lines of Code (LOC). The results indicate that the model proposed by Posnett and colleagues achieved higher accuracy compared to the Buse-Weimer model.

Scalabrino and Colleagues

The model proposed by Scalabrino et al. (2018) introduces seven new textual features designed to capture domain semantics (SCALABRINO et al., 2016), combined with structural features previously used in other works. The textual features are as follows:

1. Consistency of Comments and Identifiers: Measures the lexical similarity between the terms used in the comment of a method and those used in its body. Synonyms are not considered. This feature is inspired by the QALP model (BINKLEY et al., 2009).
2. Identifier Terms in the Dictionary: Evaluates how many identifiers are present in a standard dictionary. Identifiers composed of full words are expected to improve readability (LAWRIE et al., 2007).
3. Narrow Meaning Identifiers (NMI): Assess the semantic specificity of identifiers using WordNet trees, by comparing the number of hypernyms (general terms) and hyponyms (specific terms). Higher Narrow meaning identifiers (NMI) values reflect a higher use of specific (narrow) terms (MILLER, 1995).
4. Comment Readability: Apply the Flesch-Kincaid readability index (FLESCHE, 1948) to assess the readability of comments written in natural language.
5. Number of Meanings (NM): Use WordNet to count the number of possible meanings associated with each identifier. A higher count indicates greater code ambiguity (MILLER, 1995).
6. Textual Coherence: Estimates the number of distinct concepts implemented in a snippet. Based on Abstract Syntax Trees (ASTs) analysis, it computes all possible pairs of distinct syntactic blocks. A larger number of such blocks implies lower textual coherence.
7. Number of Concepts: Computes the number of concepts implemented at the inline level using the DBSCAN clustering algorithm (SANDER et al., 1998).

The model was evaluated using data sets from Buse and Weimer (2010), Dorn (2012), and a newly created data set consisting of 200 Java code snippets manually assessed by nine developers. The results indicate that the combination of structural and textual features leads to improved accuracy in the prediction the readability of the code.

Roy and Colleagues

The model proposed by Roy et al. (2020) follows a different approach compared to previous models, which were based on controlled experiments. Instead of relying on code

snippets annotated in artificial settings, their dataset was constructed from real-world code written by developers on GitHub. Specifically, they analyzed 5,104 pairs of classes, each representing the state of the class before and after a given commit on 2,665 commits collected from 76 GitHub repositories. These commits were manually labeled readability-improving or not. The model was trained using features extracted with the following tools, capturing the structural, lexical, and syntactic characteristics of the code in both versions.

1. *SourceMeter* (SOURCE METER, 2025) – Extracts metrics related to complexity such as McCabe’s Cyclomatic Complexity (MCCABE, 1976), Halstead Effort (HALSTEAD, 1977), coupling, cohesion, documentation, inheritance, and code size.
2. *CheckStyle* (CHECKSTYLE, 2025) – Reports warnings for issues such as magic numbers and computes the number of warnings before and after each code change.
3. *RSM* (RSM, 2025) – Calculates metrics using file diff information (before and after code change), focusing on documentation updates and formatting changes.
4. *PMD* (PMD, 2025) - addition of warning for poor programming practices and issues related to coding style design, and calculate the number of warnings produced before and after the code change.
5. *Coming* (MARTINEZ; MONPERRUS, 2019) – Mines code change patterns to detect fine-grained modifications introduced in each commit.
6. *ChangeDistiller* (FLURI; GALL, 2006) – Uses the ChangeDistiller algorithm (FLURI et al., 2007) to generate an AST and extract fine-grained code changes in 41 predefined change types.
7. *RefactoringMiner* (TSANTALIS et al., 2018) – Detects 21 types of refactoring performed on each commit.

In their study, the proposed model outperformed the other models, obtaining a precision of 79.2% and recall of 67% in incremental detection readability improvements performed by developers.

Vitale and Colleagues

The model proposed by Vitale et al. (2023), proposes to automatically improve code readability instead of classifying or scoring the code snippets. Their model transforms less readable code into a more readable version, performing specific types of readability improvement. These include renaming identifiers to more meaningful names, removing obsolete commented code, and unused variables. They use a T5 (text-to-text) approach

to learn how to rewrite code (RAFFEL et al., 2020). The model is trained on real-world examples collected from GitHub, where developers explicitly improved code readability. Each training pair consists of the original version of a code snippet and its revised, more readable counterpart. The model achieves 86% accuracy in correctly identifying commits to improve code readability.

2.1.4.2 Automatic Static Analysis Tools(ASAT)

ASATs analyze source code without execution (NIELSON; NIELSON; HANKIN, 2010), detecting potential issues such as logic errors, style violations, or poor design practices (BELLER et al., 2016). These tools assist developers by raising warnings during development, allowing early correction and contributing to better software quality (JOHNSON et al., 2013; HOVEMEYER; PUGH, 2004). For example, *FindBugs* is a ASAT focused on the detection of bugs in Java code (FINDBUGS, 2025).

ASATs can be integrated into Continuous Integration (CI) workflows to help maintain code quality during development. One of the most widely adopted tools in this context is SonarQube (VASSALLO et al., 2018), which supports more than 30 programming languages and provides over 700 static analysis rules for Java and 340 for Python (SONARLINT, 2025a). These rules are classified into three categories:

1. *Bugs*: incorrect logic or behavior.
2. *Vulnerabilities*: potential security risks.
3. *Code Smells*: maintainability issues, often associated with poor structure or clarity.

Many code smell rules are closely related to code readability. These include patterns such as highly complex methods, unclear naming, or unnecessary nesting. Figure 2 illustrates a SonarQube code smell rule whose description mentions improved code readability as a motivation.

Although SonarQube is often used in CI environments, developers can also receive real-time feedback through SonarLint, an Integrated Development Environment (IDE) plugin based on the same rule engine (SONARLINT, 2025b). Unlike SonarQube, which requires a server and per project setup, SonarLint can be applied locally and is more appropriate for large-scale analysis of open-source codebases. Other examples of linters include *ESLint* (a linter for JavaScript) (ESLINT, 2025), *PMD* (which includes linting and structural rules for Java) (PMD, 2025), and *Pylint* (a linter for Python) (PYLINT, 2025).

Some of the most common SonarLint rules detected in this work (SONARLINT, 2025a) are described below.

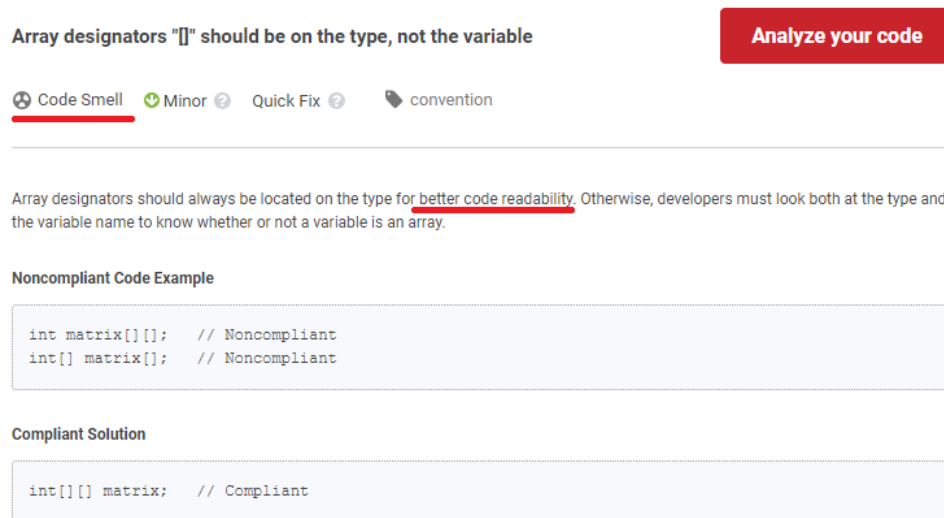


Figure 2 – Example of a code smell rule that impacts code readability

1. *Cognitive Complexity of methods should not be too high* (SONARLINT, 2025m): This rule recommends keeping the cognitive complexity of the methods low to improve the understandability of the code. Cognitive complexity, proposed by *Campbell* (CAMPBELL, 2018), is similar to McCabe’s cyclomatic complexity (MCCABE, 1976), but was designed to address some of its limitations, such as the inability to analyze nested structures (SARWAR; SHAHZAD; AHMAD, 2013), and to better handle modern language constructs such as `try/catch` blocks and lambdas. Barón, Wyrich and Wagner (2020) found a correlation between the cognitive complexity metric and subjective understandability assessments, providing empirical support for its relevance.
2. *Unused assignments should be removed* (SONARLINT, 2025k): Occurs when a value is assigned to a variable, but the assigned value is never used. These assignments should be removed, as they have no effect on the program’s behavior.
3. *Unused local variables should be removed* (SONARLINT, 2025g): If a local variable is declared but not used, it is dead code and should be removed.
4. *Local variables should not be declared and then immediately returned or thrown* (SONARLINT, 2025h): Happens when a variable is declared only to be immediately returned or thrown. SonarLint suggests returning or throwing the expression directly.
5. *The diamond operator (<>) should be used* (SONARLINT, 2025l): Recommends using the diamond operator to reduce verbosity in generic declarations. The Java compiler can infer the type automatically.

6. *Local variable and method parameter names should comply with a naming convention* (SONARLINT, 2025f): Recommends that the names follow the pattern `^[a-z][a-zA-Z0-9]*$`, starting with a lowercase letter and using only letters and digits.
7. *Lambdas should be replaced with method references* (SONARLINT, 2025i) - The method reference may be more readable compared to lambdas. For example, use `.map(B.class::cast)` instead of `.map(a -> (B) a)`.
8. *Multiple variables should not be declared on the same line* (SONARLINT, 2025j): Declaring multiple variables in a single line can reduce readability. It is recommended to declare one variable per line.
9. *Boolean literals should not be redundant* (SONARLINT, 2025b): Redundant boolean literals should be removed. For example, `if (booleanMethod() == true)` should be simplified to `if (booleanMethod())`.
10. *Modifiers should be declared in the correct order* (SONARLINT, 2025a): According to Section 8.1.1 of the Java Language Specification (JLS) (JAVA, 2025), modifiers should follow this recommended order: *Annotation, public, protected, private, abstract, static, final, sealed, non-sealed, strictfp*.
11. *"@Override" should be used in overriding and implementing methods* (SONARLINT, 2025e): Using `@Override` improves the readability of the code by clearly indicating that a method overrides or implements a method from a superclass or interface.
12. *Collection.isEmpty() should be used to test for emptiness* (SONARLINT, 2025d): Using `Collection.isEmpty()` makes the code more readable. For example, prefer `if (myCollection.isEmpty())` over `if (myCollection.size() == 0)`.
13. *"throws" declarations should not be superfluous* (SONARLINT, 2025c): Unnecessary `throws` declarations should be removed. For example, `void bar() throws Throwable, Exception {}` is redundant because `Exception` is a subclass of `Throwable`.

Models for Automatically Repairing Linter Suggestions

Linters typically identify parts of the source code that can be improved and provide suggestions for correction, but they do not apply the fixes automatically, and this task must be performed manually by the developer. Recent studies have proposed approaches to automate the repair of such suggestions. An example is the work by Loriot, Madeiral and Monperrus (2022), which introduces the Styler tool, which automatically fixes formatting violations detected by the linter Checkstyle (CHECKSTYLE, 2025), covering 24 different rules and reaching a repair rate of 41%.

2.2 GitHub Repositories

A repository manages the artifacts of a software project, including source code, and records all changes over time, preserving the complete development history. Repositories also contain metadata on code, contributors, and processes involved in software development (MUNAIAH et al., 2017).

GitHub (GITHUB, 2025a) is a large host of repositories and the primary platform for collaboration between developers and projects (CHACON; STRAUB, 2014). GitHub received 108 million new repositories in 2024 (GITHUB, 2024). GitHub has some popular open-source ecosystems as *Apache Software Foundation* (APACHE, 2025) with 2,800 repositories and approximately 1,500 members.

Although a large volume of GitHub data is available to researchers, some criteria would be necessary to filter these repositories. Some repositories are not related to software engineering, having other purposes (e.g. academic, storage, or experimental). Therefore, some filters might be applied to choose the appropriate Java projects. Some works implement filters such as the number of stars (stargazers), GitHub owners (e.g., Apache), and repositories with forks. The *reaper* framework (MUNAIAH et al., 2017) calculates a score for GitHub repositories. Higher scores contain evidence of an engineered software repository, and lower scores contain other types of repository (e.g. educational exercises). SEART is also another tool for searching for GitHub projects because it offers a comprehensive search criteria (DABIC; AGHAJANI; BAVOTA, 2021).

2.2.1 Commits and Branches

A commit represents a snapshot of the repository at a specific point in time. When a developer makes a commit, they record the changes made to the repository, which can include adding, removing, or modifying files, such as source code.

Figure 3 shows an example of a commit. The changed file is *src/test/java/org/apache/hadoop/ozone/security/TestOzoneBlockTokenSecretManager.java*. Red lines indicate content that was removed from the class, while green lines represent newly added content. The commit identifier is `8d8acbd19b26b2becd4acaf9d4e6bfe6cb6fe241`, and the commit message is *Add comment for better readability*.

Every repository has at least one branch. A branch represents a separate line of development and contains its own set of commits. Typically, GitHub repositories include a branch named *master* or *main*, which reflects the current code in the production environment. Other branches may represent parallel activities, such as bug fixes or new feature development (CHACON; STRAUB, 2014).



Figure 3 – An example of commit performed on GitHub

2.2.2 Code Review

Code review is the systematic examination of code changes, performed before or after the changes are applied (RIGBY; GERMAN; STOREY, 2008). These changes, typically submitted in commits by an author (developer), are manually reviewed by one or more reviewers (other developers). This process is referred to as a *peer code review* because it involves at least two developers, the author of the commit, and one or more reviewers evaluating the proposed modifications (EBERT et al., 2021). Code changes may include bug fixes, new features, or refactorings (PALOMBA et al., 2017). Modern Code Review (MCR) is characterized by being lightweight, tool-assisted, and asynchronous (BACCHELLI; BIRD, 2013).

2.2.2.1 Pull Request (PR)

The GitHub PR is a tool-assisted mechanism to submit contributions to the GitHub repositories. PRs are closely related to code reviews, as they promote a structured and collaborative review process (GITHUB, 2025). Each GitHub repository has a list of official collaborators, but external contributors can also participate by forking (cloning) the repository and making independent changes in their own fork. Once the changes are completed, typically in one or more commits, these contributors can open a PR to propose merging their changes into the main repository.

When opening a PR, the contributor provides a title and a description explaining the purpose of the changes. Each commit also includes a message that describes its specific update. A member of the project's core team reviews the PR, may provide feedback or suggestions using comments, and the contributor can respond or revise the code as needed. After the review process, the reviewer may approve the PR and merge it into the main branch (GOUSIOS et al., 2015).

Figure 4 shows an example of a merged PR. This PR involved three participants: the

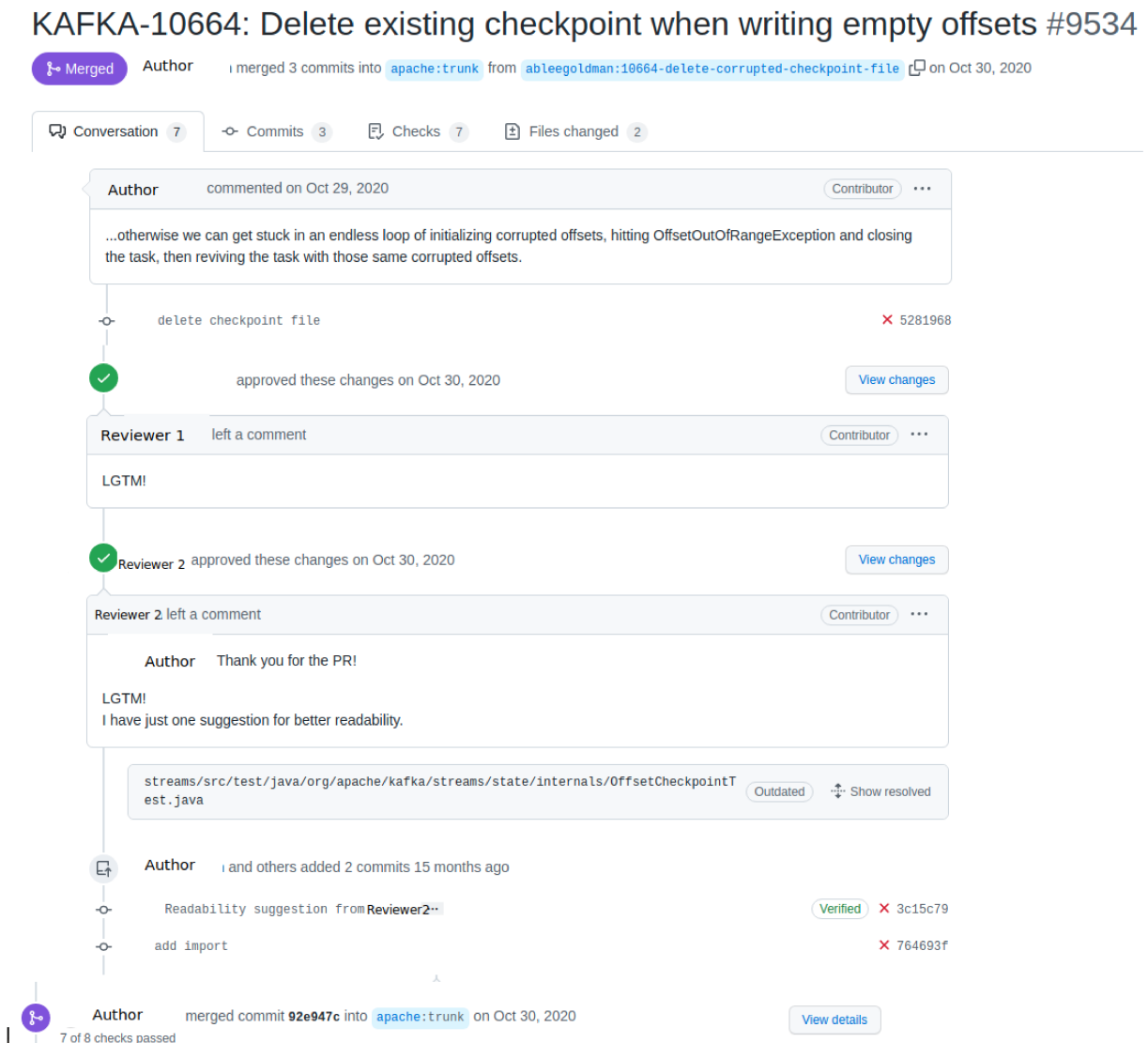


Figure 4 – An example of PR performed on GitHub

PR author and two reviewers. Their GitHub usernames have been replaced with generic names: "Author", "Reviewer 1", and "Reviewer 2". The PR is numbered #9534 and is titled: *KAFKA-10664: Delete the existing checkpoint by writing empty offsets #9534*. Its purpose is to resolve issue number 10664 in the *KAFKA* repository.

The author included a description in the PR: *"...otherwise we can get stuck in an endless loop of initializing corrupted offsets, hitting `OffsetOutOfRangeException` and closing the task, then reviving the task with those same corrupted offsets."* The initial commit had the short identifier 5281968 with the message: *delete checkpoint file*. Both reviewers approved the changes, but reviewer 2 suggested a readability improvement. The author applied this suggestion in commit 3c15c79, made an additional change in commit 764693f, and finally merged the PR into the main branch, named "trunk". In total, the PR contained three commits.

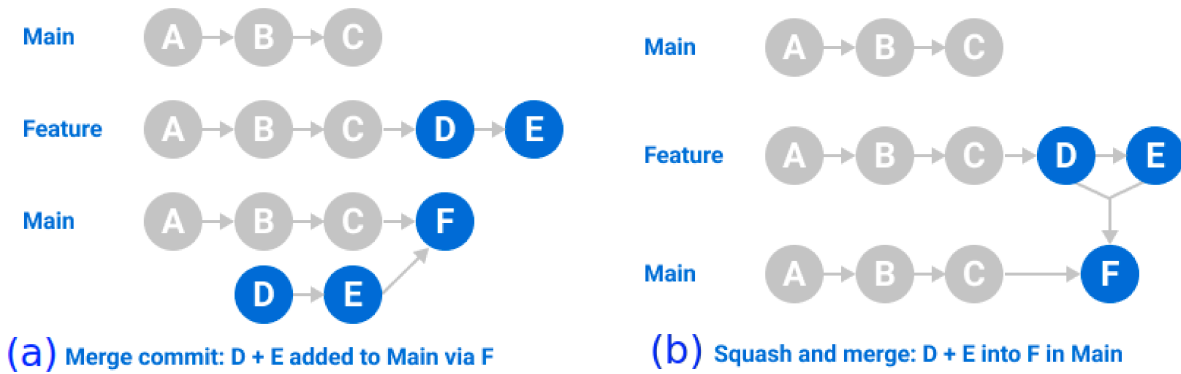


Figure 5 – Options of merging: (a) Merge Pull Request and (b) Squash and Merge

As shown in Figure 4, the merge commit identifier is `92e947c`. This commit merges the three commits from the PR into the *trunk* branch, which is the main branch of the repository.

Figure 5 illustrates two merge options available on GitHub: (a) *Merge Pull Request*, which adds the commits from the PR into the main branch in chronological order; and (b) *Squash and Merge*, which combines all commits from the PR into a single commit before merging it into the main branch.

2.2.2.2 Pull Requests as a Source for Software Engineering Datasets

In recent years, there has been a growing use of PRs to build datasets, including mine developer refactorings (COELHO et al., 2021), identify technical debt (CALIXTO; ARAÚJO; ALVES, 2024), automated tasks performed by developers using ChatGPT (TUFANO et al., 2024; SILVA; DANTAS; MAIA, 2024; XIAO et al., 2024) or, as proposed in this study, to investigate the types of code readability improvements performed in source code written in Java and Python.

In these studies, PR-build datasets typically contain code samples that were effectively merged into the main codebase and also promote a peer review process involving validation by multiple developers (SADOWSKI et al., 2018). Furthermore, PRs often provide a considerable amount of information to extract, as changes are explained and discussed between authors and reviewers (BACCHELLI; BIRD, 2013).

Figure 6 illustrates the number of PRs selected in the created data set of code readability improvements in Python (which will be further discussed in Chapter 4), grouped by year of creation. Most of these PRs are concentrated in the last three years (excluding 2025). This trend suggests that, as the number of PRs grows, it becomes more viable for software engineering research to rely on PR-based datasets with sufficient sampling to support specific investigations.

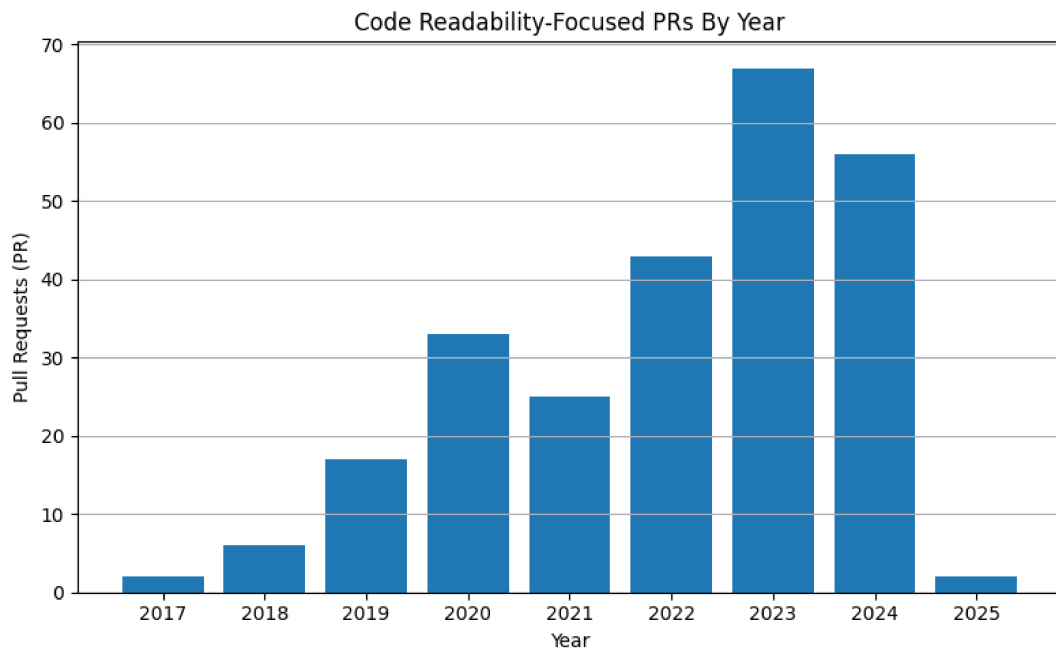


Figure 6 – Distribution of Python PRs mentioning code readability improvements, grouped by year.

2.2.2.3 Mining Merged PRs

GitHub provides extensive APIs for mining PRs, including the GitHub REST API v3 (GITHUB, 2025b) and the GitHub GraphQL API v4 (GITHUB, 2025c). Figure 7 illustrates a GraphQL query that extracts merged PRs from Java repositories whose title contains the keyword *readability*. The query returns up to 100 PR links along with pagination information. If more results are available, the `endCursor` value can be used to extract the next set of 100 PRs by querying again using that value.

2.3 Large Language Models (LLM)

In recent years, several LLMs based on the Transformer architecture have emerged (VASWANI et al., 2017). Some of them include ChatGPT (CHATGPT, 2025) from OpenAI (RADFORD et al., 2018), Gemini developed by Google (GOOGLE, 2025) and DeepSeek (DEEPSEEK, 2025b). Although these LLMs were designed for natural language processing tasks, they have also generated much interest in their use for coding problems, offering potential in providing solutions in software development environment (ZHANG et al., 2024; MASTROPAOLO et al., 2023). For example, developers can ask ChatGPT for code to solve a specific programming task or provide a detailed explanation of some concept (ERNST; BAVOTA, 2022).

Despite the relatively recent emergence of LLMs, their growing use in repositories such as GitHub has been documented. There are numerous mentions of ChatGPT in

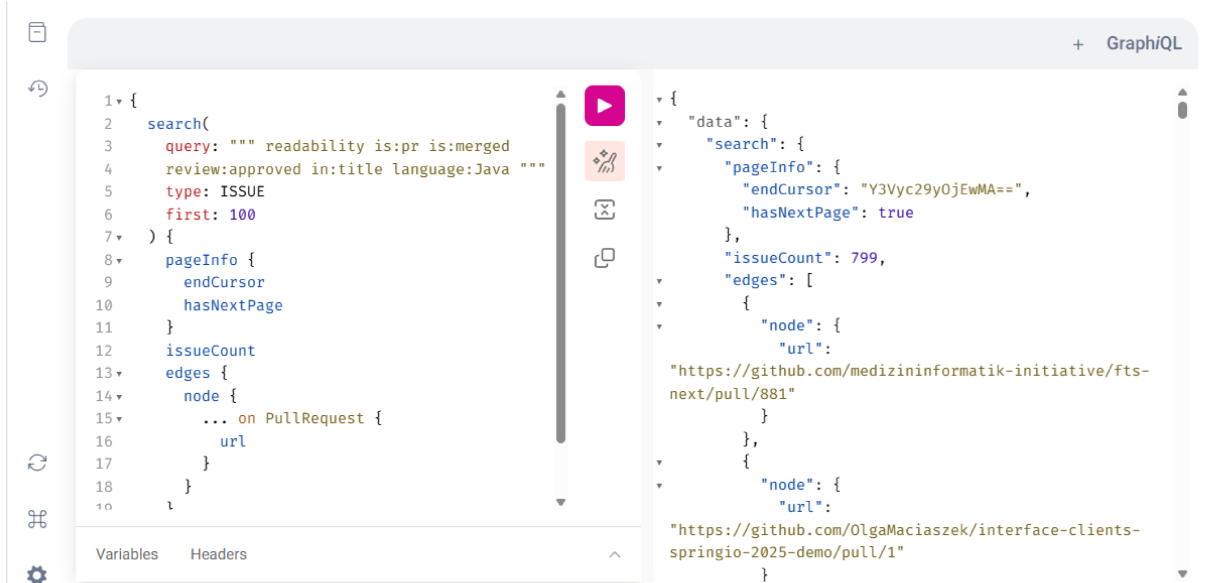


Figure 7 – A graphql query and the extracted data using the GitHub interface

commits, PRs, and even source code comments where ChatGPT provided a solution that developers used (XIAO et al., 2024; SILVA; DANTAS; MAIA, 2024). Recent studies have also shown that ChatGPT has been particularly useful for performing programming tasks such as refactoring operations into their source code, fixing bugs, and others (TUFANO et al., 2024), presenting an alternative by generating possibly original answers, including modified versions of the developer source code (EBERT; LOURIDAS, 2023).

In the scientific community, LLMs have reshaped the SE research landscape. However, these models present intrinsic characteristics that directly affect reproducibility and replicability. For example, OpenAI recently released ChatGPT 5.0 in August 2025, while legacy versions such as 3.5 are no longer accessible, making it difficult to replicate studies based on outdated models. Beyond model availability, LLM performance is highly sensitive to configuration parameters, such as temperature. Lower values make the model behave in a more deterministic way, whereas higher values produce more diverse outputs. Therefore, not reporting parameter settings can compromise the reproducibility of empirical studies involving LLMs.

Recently, a group of researchers proposed a taxonomy of study types and empirical guidelines for SE studies involving LLMs, highlighting the advantages and challenges of their use in each specific context (BALTES et al., 2025; BALTES et al., 2024). Their work shows that LLMs can support researchers in conducting empirical studies by acting, for example, as annotators (labeling artifacts based on a predefined coding guide), judges (evaluating properties of software artifacts, such as code readability), synthesis tools (integrating and interpreting information from multiple sources to generate higher-level insights), or even as subjects (producing responses that approximate those of human participants). Likewise, LLMs can assist software engineers directly, supporting tasks

such as code generation and debugging. This opens opportunities for studies focused on how developers adopt and use LLMs in practice, how LLMs can be integrated into new software engineering tools, and benchmarking their performance using standardized tasks and metrics.

2.4 Code Snippets

Code snippets, or code examples, are short and reusable fragments that demonstrate how to solve specific programming problems (KEIVANLOO; RILLING; ZOU, 2014). They improve learning (HOLMES et al., 2009), promote code reuse (HOLMES; WALKER, 2013), and can accelerate software development. Developers often choose an ad hoc query to describe their programming task (e.g., *How to reverse a string*), and then submit their query to search the Web for code examples and accompanying explanations to support their programming tasks (SADOWSKI; STOLEE; ELBAUM, 2015; UMARJI; SIM; LOPES, 2008), spending up to 20% of their time on this activity (NIU; KEIVANLOO; ZOU, 2017).

2.4.1 Human-Written Snippets

In practice, developers often rely on web search engines such as Google and Microsoft Bing to retrieve relevant code snippets, usually written by other developers (HORA, 2021b). However, the most pertinent part of the result, which is the code snippet itself, can be obscured by the surrounding content, and the top-ranked page does not always contain the most readable or reusable code snippet (HORA, 2021a).

To address the limitations of general-purpose web search engines, previous studies have proposed the development of specialized code search engines that extract source code snippets from repositories such as GitHub (GITHUB, 2025a) or question-and-answer platforms such as Stack Overflow (STACK OVERFLOW, 2025b). For example, the APISonar tool automatically extracts API usage examples from code repositories and ranks retrieved snippets according to readability and reusability metrics (HORA, 2021c). Similarly, tools such as Crokage (SILVA et al., 2020) and CRAR (SILVA et al., 2021) return code snippets along with the corresponding natural language explanations extracted from Stack Overflow posts. To make this possible, Crokage combines various information retrieval techniques, including BM25, Sent2vec, FastText, and TF-IDF, to identify and rank relevant posts based on a user query.

Table 3 illustrates examples of popular queries submitted by CROKAGE users on its website (CROKAGE, 2025). CROKAGE interprets each query and searches Stack Overflow for the most relevant posts, returning their content to the user. For example, in response to the query *"How to turn map keys to list"*, the first post retrieved by

CROKAGE is identified by ID = 9100828. The tool extracts the content of this post and displays it directly in its interface.

Table 3 – Examples of developer queries and the corresponding Stack Overflow post IDs recommended by CROKAGE.

Query	Stack Overflow posts recommended by CROKAGE				
	Top-1	Top-2	Top-3	Top-4	Top-5
How to remove an element from an array	11727522	3663949	10714351	25731624	11727508
How to reverse a string	12305723	20393516	12305682	—	—
How should I combine strings?	18740600	5591900	45338524	5591979	4424597
How to turn map keys to list	9100828	5426342	860578	44045625	2051691
How to remove duplicates	31024633	33127048	33126786	6174594	33126630
How to turn string into JSON	33667584	8596593	49089999	14205759	45110310
How to create a two dimensional array?	16350823	38732495	27061360	16437251	12231524
How would you read a file?	14169760	25028936	20227939	12961553	14089521

2.4.2 LLM-generated Snippets

LLMs, such as ChatGPT, have been increasingly used to generate code snippets from natural language queries, supporting tasks such as prototyping, refactoring, and bug fixing (TUFANO et al., 2024). Recent studies also indicate that developers interact with LLMs not only to request explanations or text refinements, but also to obtain customized code snippets for specific implementation needs (SILVA; DANTAS; MAIA, 2024). These interactions highlight the potential of LLMs to complement traditional sources of support, such as online forums, by providing on-demand code examples directly aligned with a developer’s request.

Figure 8 illustrates an example. When a developer asks “How to reverse a string in Java”, ChatGPT provides three different solutions and discusses their efficiency. For example, it highlights that `StringBuilder` is faster than `StringBuffer` due to lack of synchronization, but it does not clarify whether the first or third solution performs better. Figure 9 shows an example extracted from previous work (SILVA; DANTAS; MAIA, 2024), in which a developer created a PR to submit changes to a GitHub repository and included a link indicating that ChatGPT helped improve the code. The reviewer accepted the change and the code was merged into the codebase.

However, single-query responses from LLMs may not fully address the user’s needs. LLMs maintain a context window that relies on a series of interactions, or prompts, to generate the desired solution. This iterative process between the programmer and the

1. Using `StringBuilder` or `StringBuffer`

```
java Copy Edit  
  
public class ReverseString {  
    public static void main(String[] args) {  
        String original = "Hello World";  
        String reversed = new StringBuilder(original).reverse().toString();  
        System.out.println("Reversed: " + reversed);  
    }  
}
```

- **Note:** `StringBuilder` is faster than `StringBuffer` because it's not synchronized (better for single-threaded use).

2. Using a loop

```
java Copy Edit  
  
public class ReverseString {  
    public static void main(String[] args) {  
        String original = "Hello World";  
        String reversed = "";  
        for (int i = original.length() - 1; i >= 0; i--) {  
            reversed += original.charAt(i);  
        }  
        System.out.println("Reversed: " + reversed);  
    }  
}
```

- **Note:** This is less efficient for large strings because string concatenation creates new objects repeatedly.

3. Using character array

```
java Copy Edit  
  
public class ReverseString {  
    public static void main(String[] args) {  
        String original = "Hello World";  
        char[] chars = original.toCharArray();  
        for (int i = 0, j = chars.length - 1; i < j; i++, j--) {  
            char temp = chars[i];  
            chars[i] = chars[j];  
            chars[j] = temp;  
        }  
        String reversed = new String(chars);  
        System.out.println("Reversed: " + reversed);  
    }  
}
```

Let me know if you'd like the reversed string to ignore punctuation or spaces, or if you want to reverse words instead of characters.

Figure 8 – A ChatGPT response for the query *"How to reverse a string in java"*



Figure 9 – A merged PR using ChatGPT assistance to improve code

LLM involves refining and expanding the dialogue until the model generates a helpful and accurate response (EBERT; LOURIDAS, 2023) (MONDAL; BAPPON; ROY, 2024). This process requires *prompt engineering* since prompt quality affects the output generated by the LLM and can influence programmers' performance during programming tasks (FAGADAU et al., 2024). Previous work shows that developers, in general, execute more prompts to obtain generated code from ChatGPT compared to obtaining explanations or generating documentation (SILVA; DANTAS; MAIA, 2024). This indicates that generating source code to be merged into a PR may require more extensive prompt engineering work.

2.5 Concluding Remarks

This chapter presented the foundational background necessary to understand key concepts related to code readability. It discussed multiple definitions of the term, introduced a formal definition to guide the study, and reviewed state-of-the-art readability models, focusing on their features to clarify the metrics used in code readability assessment.

The explanation of the code review process using PRs is particularly relevant, as one of the datasets used in this thesis is based on PR information.

In addition, the definitions of refactoring and ASAT tools provide essential context for addressing some of the research questions defined in the next chapters.

Assessing the Readability of Code Snippets

This chapter aims to evaluate the readability of code snippets, whether authored by human developers and recommended on platforms such as Stack Overflow or generated by LLM such as ChatGPT, DeepSeek and Gemini. The analysis focuses on examining metrics, identifying the number and type of readability-related warnings detected by ASAT as the affected syntactic elements of the snippets.

3.1 Context

Code snippets are short fragments of source code that developers can reuse to solve specific programming tasks (KEIVANLOO; RILLING; ZOU, 2014). Developers often search for high-quality reusable code snippets online (XIA et al., 2017), and this activity can consume up to 20% of their total working time (NIU; KEIVANLOO; ZOU, 2017). To support this need, various websites provide code examples shared by other developers, such as Stack Overflow (STACK OVERFLOW, 2025b) and W3Schools (W3SCHOOLS, 2025), which are often recommended through search engines such as Google (GOOGLE, 2025).

Despite the availability of these websites, identifying code snippets that are both high-quality and contextually relevant remains a non-trivial task. Developers often rely on general-purpose search engines like Google to locate these examples, but retrieved results may require additional filtering and manual inspection. To aid in these tasks, several tools, such as Prompter (PONZANELLI et al., 2016), Postfinder (RUBEI et al., 2020), CRAR (SILVA et al., 2021), and CROKAGE (SILVA et al., 2020) have proposed customized code search engines to retrieve relevant code snippets from Stack Overflow based on the developer’s input query. CROKAGE, in particular, combines information retrieval algorithms to recommend Stack Overflow code examples along with natural language explanations.

However, with the rise of Generative Artificial Intelligence (AI) and the emergence of LLM based on the Transformer architecture (VASWANI et al., 2017) such as ChatGPT (CHATGPT, 2025), developed by OpenAI (RADFORD et al., 2018), developers and students have increasingly adopted these tools for solutions. Recent studies show that ChatGPT has proven effective in performing a wide range of programming tasks, including suggesting code changes, fixing bugs, and more (TUFANO et al., 2024). Furthermore, it offers an alternative approach by generating potentially original answers, including modified versions of the source code provided by developers (EBERT; LOURIDAS, 2023). Previous research has also shown that ChatGPT can produce high-quality educational code snippets that, in many cases, are easier to read than those written by humans on Stack Overflow (DANTAS; ROCHA; MAIA, 2023b).

More recently, new LLMs have been released, such as Google Gemini (GOOGLE, 2025) and DeepSeek (DEEPSEEK, 2025b). The latter, in particular, has drawn significant media attention due to its development in China and its impact on the financial market, contributing to the drop in the stock prices of major tech companies such as NVIDIA and Oracle (G1, 2025). Like ChatGPT, both LLMs are capable of generating source code to address tasks requested by developers.

With the rise of a new era in which various LLMs compete for developers' attention, there remains a lack of information about the source code generated by these tools. Evaluating the readability of such code snippets is essential to assess their quality. Readability is a key concern for developers during code writing (PIANTADOSI et al., 2020), as they often spend a significant portion of their time reading and understanding code (MINELLI; AND; LANZA, 2015).

One way to assess the readability of code snippets is through the use of ASAT, using linters such as *SonarLint* (SONARLINT, 2025b), *PMD* (PMD, 2025) and *CheckStyle* (CHECKSTYLE, 2025). These tools offer a set of rules that can show potential warnings in the code. Furthermore, developers consider the use of ASAT to be relevant in improving the quality of software (MARCILIO et al., 2019). Although previous work used readability models (SCALABRINO et al., 2018; POSNETT; HINDLE; DEVANBU, 2011; BUSE; WEIMER, 2010) to assess code readability (HORA, 2021c; HORA, 2021a; MORENO et al., 2015), these models only represent the probability that a code snippet is readable or unreadable. They do not offer a readability rating score nor specify which characteristics contribute negatively to the readability of the code (PIANTADOSI et al., 2020).

In this chapter, the objective is to investigate the readability of code snippets generated by LLM such as ChatGPT, Gemini, and DeepSeek compared to code snippets authored by human developers on Stack Overflow, which are recommended by CROKAGE, a state-of-the-art code search engine that has processed more than 15,000 queries from users in 80 countries. From this collection, we extracted 327 highly rated queries to obtain the corresponding snippets analyzed in this study. This study provides the following

contributions:

1. An evaluation of the effectiveness of LLM-generated code snippets to determine whether they are semantically aligned with the developer’s input query.
2. A comparison between the readability of code snippets generated by ChatGPT 3.5 in March 2023 (as extracted in the previous work (DANTAS; ROCHA; MAIA, 2023b)) and those generated by ChatGPT 4o in May 2025.
3. A comprehensive taxonomy and their respective categories for the SonarLint rules showing readability warnings.
4. A comparative study between the readability of code snippets generated by ChatGPT, Gemini, and DeepSeek and those written by human developers on Stack Overflow, considering the number and type of readability-related warnings detected by ASAT as the affected syntactic elements of the snippets.
5. An analysis of metrics applied to the code snippets, such as Cognitive Complexity (CAMPBELL, 2018), code readability (SCALABRINO et al., 2018), and LOC.
6. A replication package containing the SonarLint scripts used to extract readability warnings from code snippets, along with high-rated queries and their corresponding recommended code snippets from CROKAGE, DeepSeek, Gemini, and ChatGPT, aimed at supporting future research (DANTAS; MAIA, 2025).

3.2 The Proposed Methodology

This section aims to explain the methodology to answer the following research questions:

RQ₁: Do LLMs generate code snippets with significantly fewer readability warnings than those authored by human developers? This research question aims to statistically compare the number of readability warnings found in code snippets generated by each LLM and those authored by developers on Stack Overflow.

RQ₂: How do the code snippets recommended by LLMs compare to those authored by human developers in terms of readability and understandability metrics? This research question aims to compare the code snippets generated by LLM with those authored by humans on Stack Overflow using readability metrics such as (SCALABRINO et al., 2018), which is reported more effective on the context of code snippets (i.e., single snapshots (PIANTADOSI et al., 2020)), and cognitive complexity (CAMPBELL, 2018) which have some understandability aspects (BARÓN; WYRICH; WAGNER, 2020).

RQ₃: How does the readability of the code snippets recommended by LLMs compare to those authored by human developers, based on the type of warnings generated by ASAT?

This research question aims to investigate whether LLMs are capable of generating code snippets with higher readability compared to those authored by human developers on platforms like Stack Overflow. This study operationalizes readability through the type of warnings generated by an ASAT (SonarLint). The analysis focuses on warnings related to code content and usage, such as useless code, redundant code, and naming conventions, which are strong indicators of readability issues.

RQ₄: How does the readability of the code snippets recommended by LLMs compare to those authored by human developers under the perspective of the syntactic elements affected by the readability warnings? This research question focuses on analyzing the readability of code snippets by examining the syntactic elements affected by readability warnings. This analysis examines the code elements referenced by the warnings, such as variables, statements, and functions, to understand how LLM generated snippets compare to those written by human developers.

RQ₅: To what extent has ChatGPT improved in generating readable code snippets compared to its older version? Since the code snippets extracted by both versions (3.5 in March 2023 and 4o in May 2025) in response to the same set of programming queries, two years apart, this research question aims to assess whether the newer version of ChatGPT demonstrates improvements in terms of code readability, based on the warnings identified by the SonarLint rules.

Figure 10 illustrates the methodology proposed to address the research questions of this work. The following subsections describe the selection of input queries to extract code snippets (Section 3.2.1), extracted code snippets from LLM and Stack Overflow (Section 3.2.2), selected code snippets that have semantic adherence to the input queries (Section 3.2.3), identified the code readability warnings (Section 3.2.4), classified the types of code readability warnings (3.2.5) and address how to answer each proposed research question, based on the data extracted (3.2.6). A replication package is available, including the data set, scripts, assessments, and reproduction instructions (DANTAS; MAIA, 2025).

3.2.1 Selecting Developer Input Queries

To assess the readability of the code snippets generated by LLMs, the corresponding input queries were initially selected. These queries were obtained directly from the authors of the publicly available CROKAGE tool (CROKAGE, 2025). The CROKAGE user interface is shown in Figure 11. CROKAGE retrieves relevant code snippets and explanations from Stack Overflow in response to a given query. (SILVA et al., 2020).

The CROKAGE tool received a significant number of 15,865 queries from users in approximately 80 countries. This response from the community can be explained by the

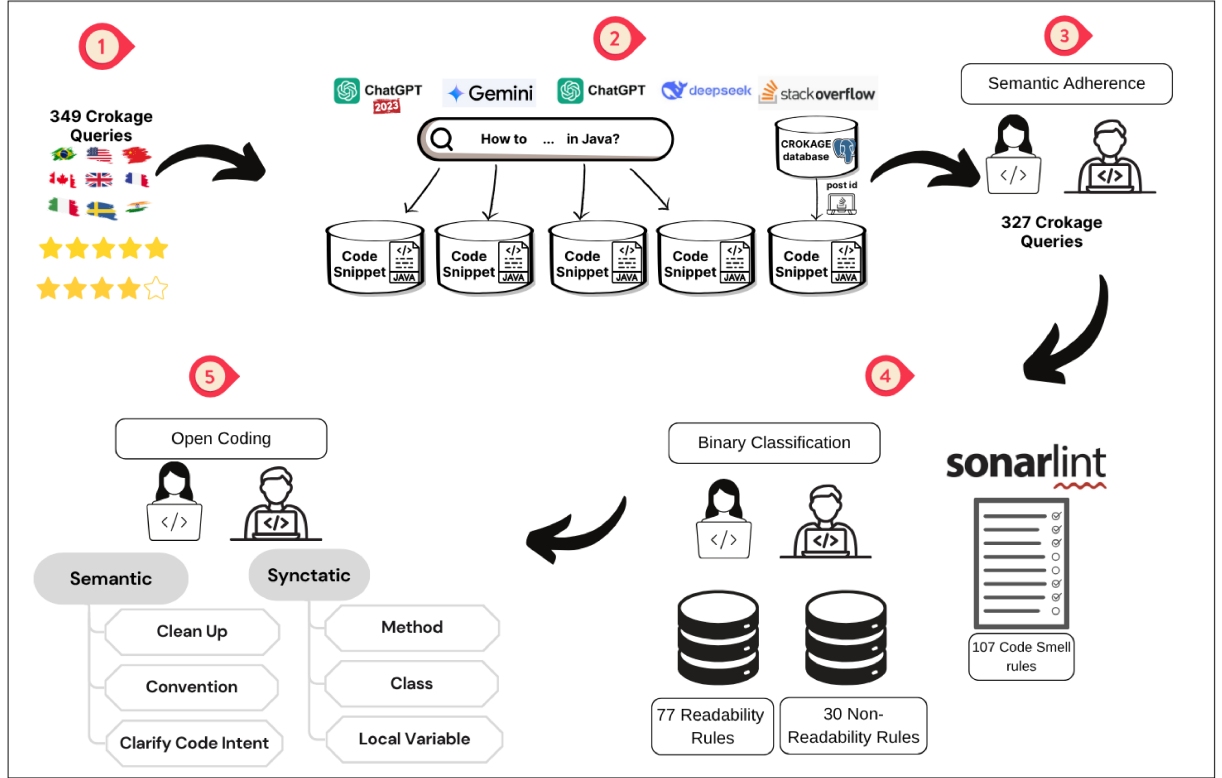


Figure 10 – Overview of the proposed methodology

exposure of the tool in the Stack Overflow blog (STACK OVERFLOW, 2025a). However, only 784 of these queries had code snippets that users evaluated, as shown in the *How do you like the overall result* section of Figure 11. This study focused exclusively on 4-star and 5-star ratings (that is, where the recommended code snippet from Stack Overflow is well accepted by users), resulting in a total of 435 queries. Finally, duplicated and small queries (those with less than two words) were removed.

To prevent duplicate evaluations of the same code snippet, one of two different queries that shared the same top-1 post id from Stack Overflow was excluded, resulting in a final set of 349 queries for analysis, as shown in step 1 of Figure 10.

Table 4 presents the top 10 most frequent 1-gram, 2-gram, and 3-gram searches found in input queries, excluding stop words. The 3-grams often start with *how to*, reflecting a focus on solving specific programming tasks. The 1-gram consists of popular structures commonly used in Java programming, including *array*, *string*, *file*, and *list*.

3.2.2 Code Snippet Extraction from Stack Overflow and LLM Models

To obtain the recommended and generated code snippets from both Stack Overflow and LLMs, a process was applied to each of the 349 input queries that had received



Figure 11 – Input queries performed and evaluated by developers into CROKAGE tool

Table 4 – The top-10 most frequent n-grams in the input queries used in this study

Rank	1-gram	2-gram	3-gram	3-gram with stop words
1	array(36)	element array (4)	remove element array (1)	how to create (16)
2	create(30)	linked list (4)	android view contact (1)	to create a(10)
3	string(30)	text file (3)	view contact lookupcolumn (1)	how do i(10)
4	file(23)	files folder (3)	build rest service (1)	how to get (7)
5	get(21)	convert string (3)	create service watch (1)	how to convert (6)
6	java(20)	two numbers (3)	pop elements array (1)	how to remove (5)
7	list(20)	date string(3)	read array properties (1)	how to add (5)
8	convert(16)	android view (2)	file spring mvc(1)	how to print (4)
9	add(12)	elements array (2)	encrypt something aes (1)	how to use (4)
10	sort(10)	sort array (2)	writing text file (1)	from an array (3)

positive developer ratings (4-star or 5-star).

3.2.2.1 Stack Overflow Posts Recommended by CROKAGE

The CROKAGE data set comprises a comprehensive collection of 4,106,137 posts, consisting of 1,543,653 questions and 2,562,484 answers. These posts were extracted from a March 2019 StackOverflow dump¹. The data set specifically includes questions tagged with Java and its derivatives, such as `<java-8>`. To build the dataset, the CROKAGE approach selected question-answer pairs that met specific criteria. Each pair had to receive at least one upvote from the Stack Overflow community, indicating approval from other users. Furthermore, every answer in the pair had to include at least one code segment, identified by the presence of `<code>` and `<pre>` tags (SILVA et al., 2020).

¹ <https://archive.org/details/stackexchange>

To extract the Stack Overflow code snippets recommended by CROKAGE, the post IDs available in its database were used. These posts correspond to the recommendations returned by the CROKAGE Web tool when a user submitted a query. When a query was submitted, CROKAGE retrieved a list of relevant Stack Overflow posts and saved the recommended ones in its database. For example, in the input query ID = 352: “how to convert dom to string”, CROKAGE recommended the following top 10 post IDs from Stack Overflow: 32739465, 32739440, 3140638, 31722446, 8484990, 5575306, 3098928, 12296802, 729662 and 13765293. In this case, the code snippets from each Stack Overflow post are extracted into new Java files.

However, in some cases, the CROKAGE recommended Stack Overflow posts contain only fragments of Java code, pseudocode, or code segments accompanied only by developer explanations. These situations were identified through manual analysis and the following methodology was adopted:

- ❑ **Complementary Java fragments:** In some posts, the answer provides only a fragment of code that is meaningful only when combined with the code already present in the corresponding question. In such cases, the question and answer snippets were manually merged to form a complete executable code snippet. Otherwise, the post was discarded. For example, in query ID = 219 (“How to output custom date string”), CROKAGE recommended the Top-1 Stack Overflow post ID = 9230991, which included a single-line Java snippet followed by an explanation. An inspection of the question showed that this line directly addressed the query and complemented the developers code, so the two snippets were merged.
- ❑ **Pseudocode or explanatory code segments only:** If the post includes only pseudocode or code fragments that are not executable (e.g., codes shared by developers such as `Integer i = ...` or consist only of explanations without meaningful code, the Stack Overflow post is discarded.

3.2.2.2 LLM Generated Code Snippets

For LLMs, the authors manually submitted each query, simulating the typical behavior of a developer or computer science student, by initiating a new chat for each question, as shown in Step 2 of Figure 10. To align the context with the Java programming language, the tokens ‘*in Java*’ are appended to each query to ensure consistency, as the CROKAGE recommendations specifically focused on Java code snippets extracted from Stack Overflow.

All code snippets were collected in May 2025 using their free versions, namely ChatGPT 4o, Gemini 2.5 Flash, and DeepSeek V3. Furthermore, the code snippets generated by ChatGPT 3.5 collected in March 2023 are also reused for the same data set of 349 queries from a previous study (DANTAS; ROCHA; MAIA, 2023b). In some cases, it was

observed that LLMs begins with pseudocode or step-by-step explanations before eventually providing a complete Java class. In such cases, the Java class is extracted into a new Java file. If the solution required multiple classes, all were placed in the same Java file, with only one declared public, usually the main class.

3.2.3 Assessing Semantic Adherence of Code Snippets with Input Queries

As mentioned in the previous section, only Stack Overflow posts recommended by CROKAGE that received positive ratings from developers (4-star and 5-star) were considered. However, in CROKAGE, users could evaluate the entire set of recommended posts for a given input query rather than individually evaluating each post. Therefore, it was necessary to assess the semantic adherence of each individual post to select the most appropriate ones for inclusion in the dataset. This assessment was manually performed by two evaluators who examined whether each post was semantically aligned with the corresponding input query, as detailed in this subsection.

To illustrate the need for semantic adherence to the CROKAGE recommendations, consider the input query ID = 208: “Find the second largest number” as illustrated in Figure 12. The CROKAGE tool recommended 10 Stack Overflow posts, with the top-1 post (ID = 12279849) containing a code snippet to find the third largest number, and the top-2 post (ID = 21926229) providing a solution for identifying the largest and smallest numbers. Only the top-3 posts (ID = 24428620) directly address the query by returning the second largest number, as requested. Although the entire set of recommendations received a 5-star rating from the user, the semantic alignment of each post was individually evaluated in this work. As a result, only the third post was retained for the dataset, while the first two were discarded due to misalignment with the query’s intent.

Another example of CROKAGE’s limitations in capturing the semantic meaning of a query is ID = 230: *how to create a sam*. While LLMs returned implementations related to the Single Abstract Method (SAM) (that is, the correct interpretation of SAM in this context), CROKAGE recommended posts that included objects with fields named as *Sam*, such as the top-1 post (ID = 47818178).

In addition to the code snippets extracted from Stack Overflow posts, those generated by the LLM were also evaluated for semantic adherence, that is, whether they are consistent with the intent of the original query. For example, in the case of query ID = 24: “spring boot default encoding”, Gemini and DeepSeek generated configuration snippets for `.properties` files such as `spring.http.encoding.charset = UTF-8` instead of Java code. Similarly, for query ID = 22: “cannot fetch multiple bags simultaneously”, while most LLMs returned code snippets related to object relational mapping in Java, ChatGPT 4o (2025) generated a class named `BagFetcher` with an attribute called `bag`, which



Figure 12 – The Top-3 recommended Code Snippets by CROKAGE for the input query ID = 208: “Find the second largest number

lacked semantic alignment with the actual query context.

To evaluate the ability of each LLM to solve programming tasks given by developers, two evaluators manually assessed the 349 input queries and their corresponding LLM code snippets, as shown in Step 3 in Figure 10. The evaluators evaluated each pair (query and code snippet) on a Likert scale from 1 to 5 based on the following criteria:

1. **Unrelated:** the code snippet is not related to the query. For example, in query ID = 22 (*cannot fetch multiple bags simultaneously in Java*), ChatGPT 2025 generated a thread to process multiple **String** objects concurrently. This response has no connection to the query, which was about handling bags in Java Persistence API (JPA).
2. **Weakly related:** the code snippet does not objectively address the query problem. For example, in query ID = 184 (*gradient descent in Java*), Stack Overflow posts recommended by CROKAGE provided non-executable code fragments mixed with pseudocode and comments, which do not present a clear and objective solution to the query.
3. **Related:** the code snippet is connected to the query but requires substantial modifications to fully address the problem. For example, in query ID = 24 (*spring boot default encoding*), Gemini and DeepSeek showed how to modify the **.properties** files, which are indeed part of the solution. However, they did not include the necessary changes in the Java classes.
4. **Understandable:** the code snippet partially addresses the query problem, requiring more specific code to fully solve it. For example, in query ID = 64 (*remove from a linked list in Java*), the first Stack Overflow post recommended by CROKAGE provided a code example using a custom **Seat** class. The snippet illustrates how to

create a `LinkedList<Seat>` and iterate over its elements with an `Iterator`, removing items that match a given row number. Although this example demonstrates the mechanism of element removal from a linked list, it is not a complete or directly applicable solution to the query, since it is tied to a specific domain class rather than showing a general-purpose removal operation.

5. **Straightforward:** the code snippet fully addresses the query problem. In the same query ID = 64 (*remove from a linked list in Java*) used as an example for the **Understandable** category, ChatGPT 2023 created a `LinkedList<String>` and demonstrated two types of removals: by index and by searching for a specific string. This snippet provides a direct and complete answer to the query.

To assess the semantic adherence of the code snippets, the two evaluators independently evaluated the 349 input queries and their respective code snippets from each baseline (Stack Overflow, ChatGPT 4o, Deep Seek, Gemini, and ChatGPT 3.5). Then, the average evaluation result was calculated. If the two evaluations differed by more than one Likert point and at least one of them was greater than 3, the answer was marked for reevaluation by both authors in an agreement phase. The marked pairs were then discussed, and if the evaluations still differed by more than one Likert point after discussion, the pair was discarded. The average rating was then recalculated for the marked pairs. A relevant pair is considered if its average rating is 4 or higher.

Following this approach, 327 relevant input queries and their respective code snippets were obtained, while 22 queries and associated snippets were discarded. Among the discarded cases, 18 were excluded due to lack of semantic alignment in any of the CROKAGE recommended Stack Overflow posts, and only four were excluded due to lack of semantic alignment in the responses generated by LLM. This indicates that LLM and Stack Overflow are both capable of addressing (partially or completely) the semantics of most of the queries. Kappa values were measured before and after the agreement phase, which were 0.5972 and 0.7059, respectively ($p\text{-value} < 0.05$), indicating an improvement from moderate to substantial agreement (LANDIS; KOCH, 1977).

Figure 13 illustrates the distribution of the selected Stack Overflow posts based on their ranking position in CROKAGE, where the final code snippets were extracted. More precisely, 203 out of 327 snippets (62%) were extracted from the top-1 recommended post, while 294 out of 327 (90%) came from posts ranked within the top 1 to 3 positions.

Finding 1) The majority of the recommended code snippets received positive ratings (4-star or higher) from both evaluators. However, in the case of Stack Overflow, 38% of the selected snippets were not the top-ranked recommendation provided by CROKAGE. This outcome suggests that LLMs are capable of generating code snippets

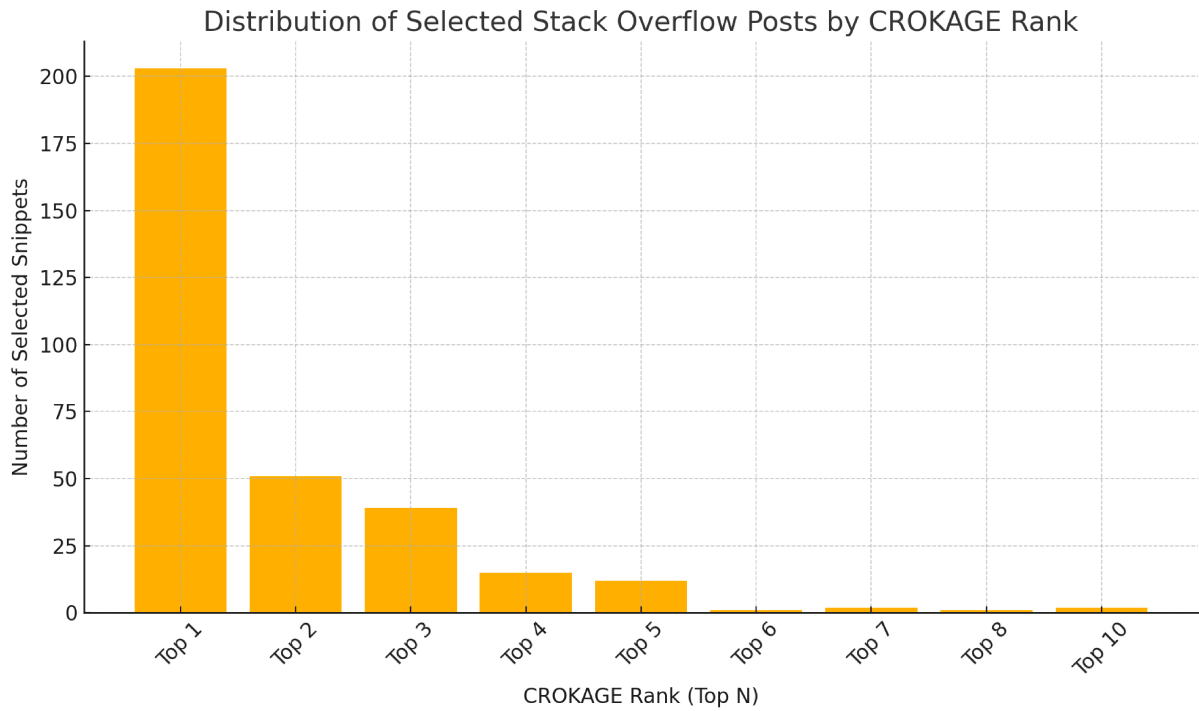


Figure 13 – Distribution of the selected Stack Overflow posts according to their ranking in CROKAGE

that semantically align with the developers intent, even when the corresponding Stack Overflow recommendations by CROKAGE are not ranked first. However, semantically aligned code snippets does not ensure correctness.

3.2.4 Identifying Code Readability Warnings

In this step, the 1,635 code snippets extracted and validated in the previous section (327 from each baseline: Stack Overflow, ChatGPT 3.5 2023, ChatGPT 4o 2025, Gemini 2.5 Flash, and DeepSeek V3) were evaluated by the SonarLint tool (SONARLINT, 2025b), aiming to identify possible code readability warnings. This tool can detect readability warnings based on the rules defined in its documentation (SONARLINT, 2025a).

To extract the code readability warnings (i.e., rule violations) identified by SonarLint, a standalone client script was developed that integrates with the SonarLint core (version 10.22) available on GitHub. The script processes each code snippet collected from the five baselines by submitting it to SonarLint and retrieving the list of detected code smell warnings. In total, SonarLint reported 5,910 warnings in 107 unique rules classified as code smells, as shown in Step 4 of Figure 10. SonarLint warnings related to bugs or vulnerabilities were discarded.

To ensure that SonarLint evaluated all 1,635 code snippets, all package declarations

were removed from the few code snippets that originally included them. Both LLM and Stack Overflow users typically omit package declarations in Java code snippets, but the remaining instances are removed to deliberately induce a warning from the rule *java:S1220*: "*The default unnamed package should not be used.*" in each of the 1,635 code snippets. This rule serves as a marker that confirms that the code snippet was successfully analyzed. Otherwise, in cases where no warnings are reported by SonarLint, one might incorrectly assume that the code snippet was properly analyzed, when, in fact, it may have been silently skipped by SonarLint. In this process, nine code snippets were failed analysis due to formatting errors (e.g., semicolons incorrectly placed after class definitions) introduced during the copy-pasting process. This verification step helped detect such warnings that otherwise could go unnoticed.

Of the 107 unique rules classified as code smells, each rule was categorized into one of two classes: readability or non-readability, as illustrated in step 4 of Figure 10. This classification is necessary because some code smells could affect other aspects, such as performance, for example. This classification was independently performed by two evaluators. During the agreement process, disagreements occurred in only nine of the 107 SonarLint rules, which were resolved through discussion among evaluators, supported by the SonarLint documentation, which contains descriptions, code examples, and tags for each rule (SONARLINT, 2025a). For example, seven rules labeled *performance* by SonarLint were discarded. As a result, 77 SonarLint rules that address readability-related warnings were identified, resulting in 1,458 readability warnings reported by SonarLint. In this process, 30 rules were discarded.

Figure 14 illustrates examples of a non-readability rule and a readability rule. In Figure 14(A), the SonarLint rule *java:S2864* recommends refactoring the loop interaction of the map by replacing the use of `keySet()` and manual search with `entrySet()`, citing performance improvements as the main motivation. Although this change may implicitly enhance readability, SonarLint developers explicitly categorized the rule as performance-oriented. In contrast, in Figure 14(B), the SonarLint rule *java:S3776* aimed to improve code readability by encouraging the refactoring of large code blocks into smaller and more understandable pieces, an intention clearly stated in the official description of the rule.

3.2.5 Classifying Types of Code Readability Rules in SonarLint

To address RQ₃ (that is, the number and type of SonarLint warnings), two evaluators conducted an open coding process, as illustrated in step 5 of Figure 10, assigning labels to each of the 77 SonarLint rules previously classified as improving code readability. These labels indicate the specific type of readability improvement associated with each rule. Initially, each SonarLint rule includes a set of internal tags (e.g. *redundant*, *clumsy*, *confusing*, *convention*, *obsolete*, *cert*, *brain overload*, *design*) that help distinguish the type of improvement the rule aims to address. These tags were used to support the

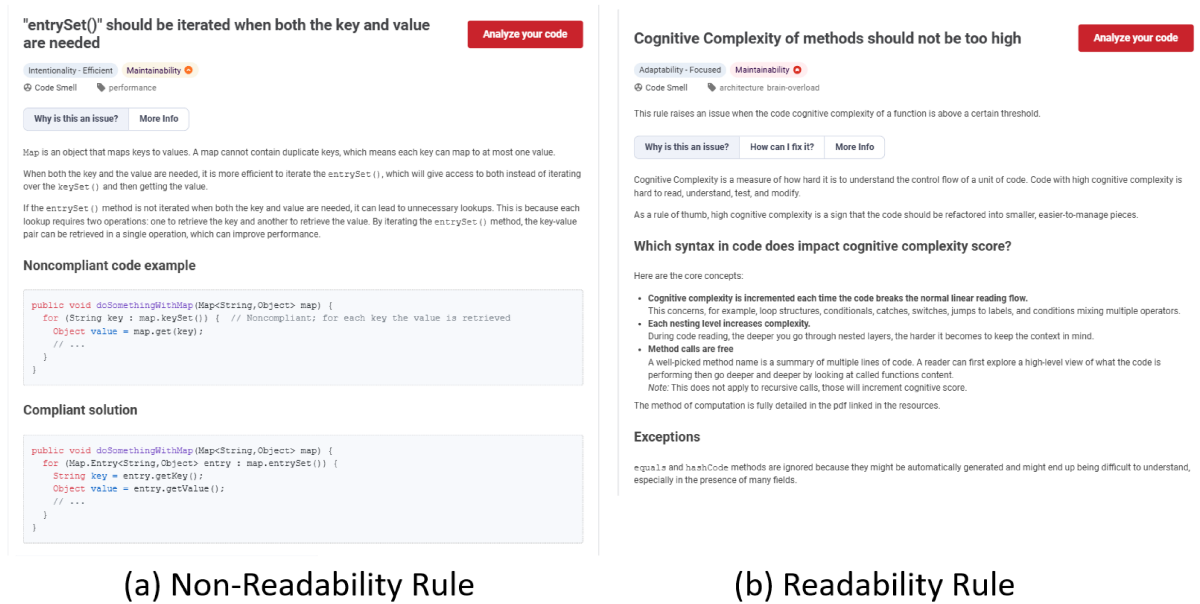


Figure 14 – Examples of a Non-Readability Rule and a Readability Rule identified in SonarLint

categorization of each rule. Moreover, Pantiuchina et al. (2020) has already employed labels such as *Clean Up* and *Improve Naming*, previously defined as common motivations for developers to improve code readability. In this study, these labels are also reused to ensure consistency with previous research.

Initially, the rules were organized into categories based on their predefined SonarLint tags. For example, nine out of the 77 rules were labeled *convention* and 15 as *clumsy*. However, during the open coding process, new labels were created, and the evaluators had to reevaluate and relabel the SonarLint rules multiple times as new labels were discovered, and others were abstracted. After labeling all 77 SonarLint rules, any conflicts were discussed and resolved, resulting in a taxonomy of 21 subcategories grouped into five different categories.

To address RQ₄ (that is, syntactic elements), both evaluators assessed descriptions, labels, and examples of SonarLint rules to examine their syntactic scope. Unlike the open coding process that involves the creation of categories, this analysis focused solely on extracting the scope of each rule based on their own description provided by SonarLint (SONARLINT, 2025a). For example, the rule *java:S1068 - Unused "private" fields should be removed* is classified as a *Clean Up* label, as it targets the elimination of useless code, but the explicit syntactic element involved is a *field*, as indicated in the description of the rule.

3.2.6 Answering the Research Questions

The previous subsections provided all the necessary data to answer the research questions. This subsection details the approach used to address each question.

3.2.6.1 Statistical Tests

To answer the RQ₁, the five baselines represent repeated measurements in the same set of 327 queries, characterizing a design within the subjects with more than two related groups. In this setting, the assumptions of normality and independence required by parametric tests such as repeated measures ANOVA are not satisfied: the data are non-normally distributed and paired, since each query produces five responses (one per baseline). Therefore, the non-parametric test Friedman (1937) was applied with a confidence level 95% (p -value < 0.05) to assess whether there are statistically significant differences between baselines. As a natural complement to the Friedman test, Nemenyi's post hoc procedure was applied to determine which specific pairs of baselines differed significantly.

For example, to verify whether a particular baseline exhibits significantly more readability warnings than the others, a data set was constructed as illustrated in Figure 15. In this example, the code snippet with ID = 3 contains three readability warnings in the Stack Overflow and ChatGPT 2023 versions, and no readability warnings in the ChatGPT 2025, DeepSeek, and Gemini versions. Using this paired dataset, the Friedman test followed by the post hoc Nemenyi test was performed to determine which baselines differ significantly from each other.

3.2.6.2 Readability and Understandability Metrics

To answer RQ₂, the LOC metric is extracted for each code snippet, counting the total number of lines and distinguishing between code lines, comments, and blank lines. Moreover, two complementary dimensions are considered: code readability and code understandability. Readability refers to the immediate ease of reading the code, that is, how clear and accessible it appears at a surface level. In contrast, understandability concerns the cognitive effort required to follow and reason about the underlying logic of the code.

To assess code readability, the model proposed by Scalabrino et al. (2018) is used. This model was chosen because it is specifically designed to evaluate single Java files, which aligns well with the structure of code snippets. Following Piantadosi et al. (2020), a snippet is considered readable when the model returns a score of score ≥ 0.6 or higher and unreadable when the score is score ≤ 0.416 or lower. Scores between 0.416 and 0.6 are classified as undetermined, which means that the model cannot confidently determine the readability of the snippet.

	A ▼	B	C	D	E	F
1	snippet_id	stackoverflow	chatgpt	chatgpt2025	deepseek	Gemini
2	2	1	1	0	0	0
3	3	3	0	0	0	0
4	4	2	0	0	0	0
5	5	0	9	2	2	4
6	6	0	0	3	2	1

Figure 15 – Data set to apply Friedman tests followed by Nemenyi’s post-hoc tests

To assess code understandability, the Cognitive Complexity metric proposed by Campbell (2018) is applied. Unlike Scalabrino’s model, this metric captures the mental effort required to understand the control flow and the nested structures. Each code snippet is analyzed using the implementation available in the SonarSource tool, which returns a complexity score. The higher the score, the greater the cognitive effort required to understand the snippet. The metric increases the score based on elements such as nested structures, including `if` and `loop blocks`, as well as control flow constructs such as `try/catch` and `lambdas`.

3.2.6.3 Types of Readability Warnings and Affected Syntactic Elements

To answer RQ₃ and RQ₄, the number of affected code snippets and the corresponding warnings are collected and grouped according to their classification. For RQ₃, the grouping is based on the categories resulting from the open coding process as previously described, while for RQ₄, it follows the affected syntactic elements.

3.2.6.4 Comparing ChatGPT versions

RQ₅ involves a direct comparison of SonarLint readability warnings in both versions of ChatGPT used in this study, to highlight the cases where a specific rule shows a marked difference.

3.3 Results and Findings

In this section, the results of the research questions are presented. A discussion is also included, based on the observations after manually analyzing 1,635 code snippets across five baselines.

3.3.1 Number of Readability Warnings

Table 5 presents the number of code snippets affected and the total readability warnings detected by SonarLint for each baseline. For example, 216 of the 327 Stack Overflow

snippets (66%) are affected by at least one readability warning, while the proportions are 29% for ChatGPT 2023, 24% for ChatGPT 2025, 34% for DeepSeek, and 31% for Gemini. In total, 611 of the 1635 snippets (37%) were affected by at least one readability warning.

When analyzing the total number of warnings, Stack Overflow snippets recommended by CROKAGE exhibit the highest values. DeepSeek is the second worst in terms of general warnings. ChatGPT 2023 snippets contain more warnings than Gemini, but affect fewer snippets overall. In contrast, ChatGPT 2025 produces the fewest warnings.

The Friedman test was performed using a confidence level of 95% (p -value < 0.05), and the post hoc Nemenyi tests confirmed that Stack Overflow contains significantly more readability warnings compared to all LLMs. No significant differences were found among the LLMs under the predetermined threshold, except ChatGPT 2025 which shows a significant difference compared to DeepSeek.

Table 5 – Number of readability warnings and affected code snippets grouped by the baseline ($n = 327$ snippets)

Stack Overflow		ChatGPT 2023		ChatGPT 2025		DeepSeek		Gemini	
Snippets	Warnings	Snippets	Warnings	Snippets	Warnings	Snippets	Warnings	Snippets	Warnings
216	623	97	225	80	151	114	261	104	198

RQ₁: Do LLMs generate code snippets with significantly fewer readability warnings than those authored by human developers? Stack Overflow code snippets recommended by CROKAGE exhibit significantly more readability warnings detected by SonarLint compared to those generated by the LLMs. Furthermore, DeepSeek shows significantly more readability warnings than ChatGPT 2025.

3.3.2 Code Metrics

Figure 16 shows the distribution of LOC between all baselines. Gemini tends to generate code snippets with a higher number of lines compared to the other baselines. The number of comment lines are also extracted separately, as shown in Figure 17, where Gemini also includes more comments. A Friedman test was performed using a confidence level of 95% (p -value < 0.05), and post hoc Nemenyi tests confirmed that Gemini generates significantly more LOC and comments than all other baselines, since human-written snippets found on Stack Overflow produced significantly fewer comments than any of the LLM.

Figure 18 shows the dispersion of cognitive complexity between all baselines. Gemini shows a distinct distribution, with a higher median and greater variability in the results, including several outliers compared to the other baselines. The Friedman test followed by the Nemenyi post hoc confirms that Gemini differs significantly from all other baselines using a confidence level of 95% (p -value < 0.05). These findings suggest that Gemini tends

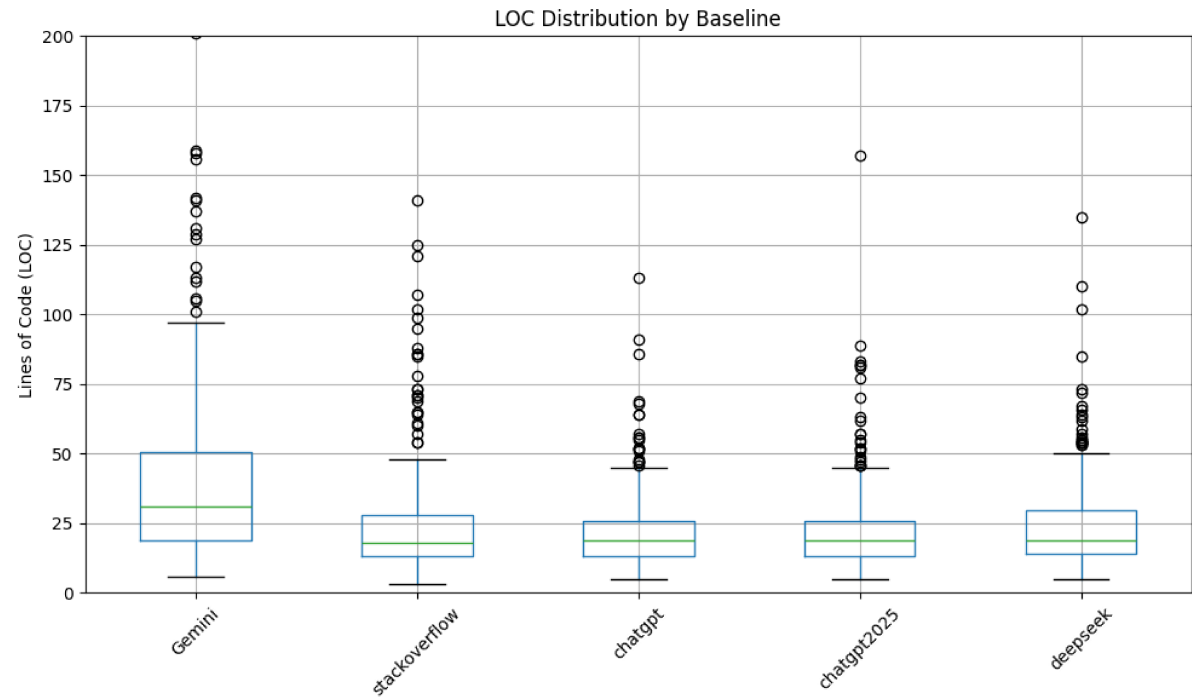


Figure 16 – Boxplot of LOC for all 327 code snippets, grouped by baseline

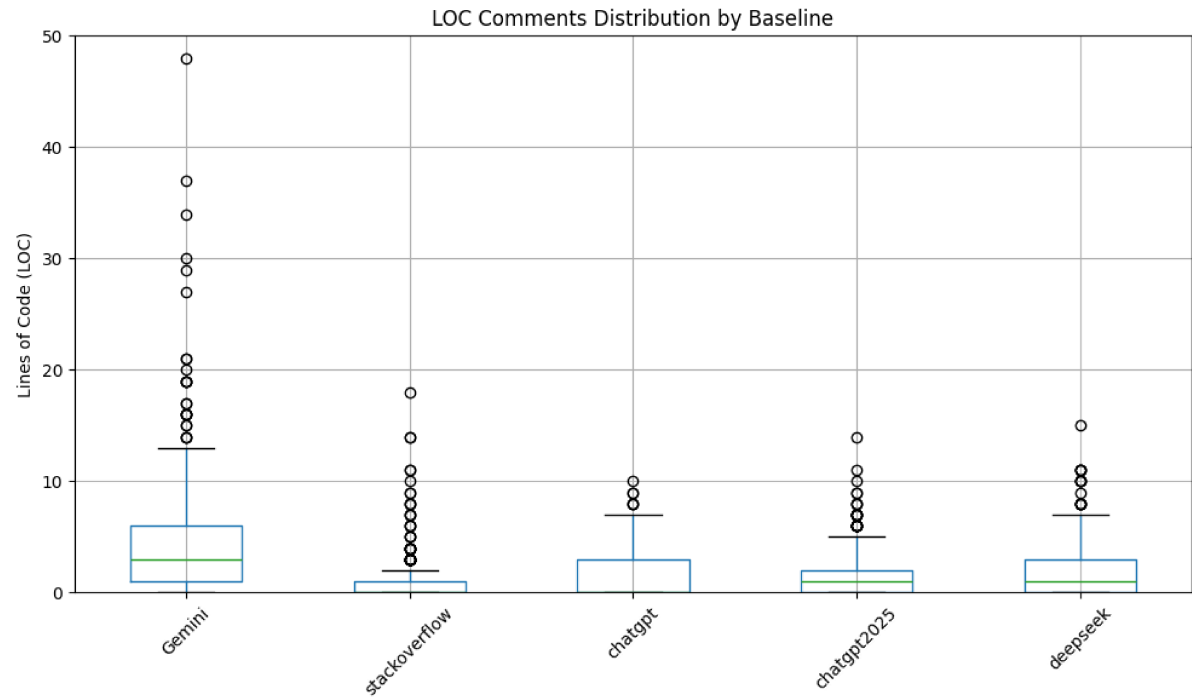


Figure 17 – Boxplot of comments LOC for all 327 code snippets, grouped by baseline

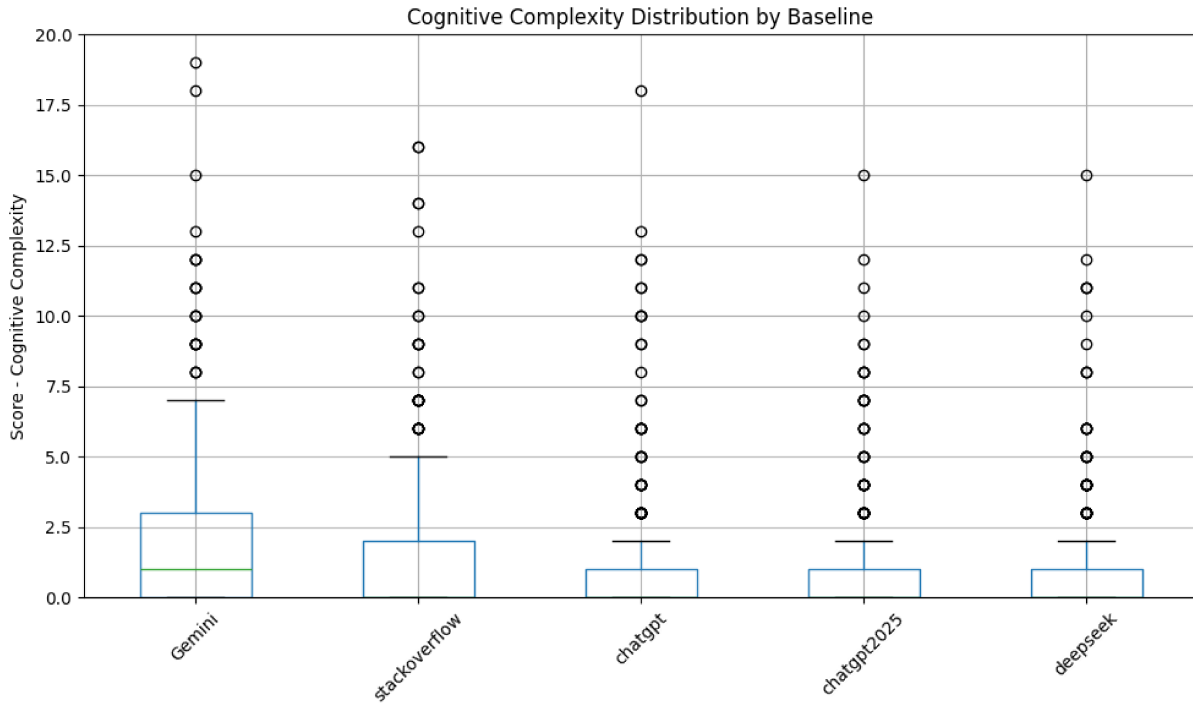


Figure 18 – Boxplot of Cognitive Complexity for all 327 code snippets, grouped by baseline

to produce snippets with higher cognitive complexity, that is, more control structures such as nested conditionals, loops, and others.

Table 6 presents the analysis of the code readability scores obtained using the Scalabrino et al. model (SCALABRINO et al., 2018) for the 327 code snippets from each baseline.

Table 6 – Readability classification according to the Scalabrino model, with counts and standardized residuals by baseline. ($n = 327$ snippets)

	S.Overflow		GPT 2023		GPT 2025		DeepSeek		Gemini	
	Count	Residual	Count	Residual	Count	Residual	Count	Residual	Count	Residual
Readable	224	-1.59	244	-0.32	276	1.69	260	0.68	242	-0.45
Undetermined	59	1.68	50	0.38	35	-1.80	43	-0.63	50	0.38
Unreadable	44	2.46	33	0.47	16	-2.61	24	-1.16	35	0.83
Total	327	-	327	-	327	-	327	-	327	-

The *Count* column shows that, in most cases, each baseline (including both LLM-generated and human-written snippets recommended by CROKAGE on Stack Overflow) achieved a readability rate between 68% and 84%. Although these results may appear to diverge from previous findings, such as those reported by Piantadosi et al. (2020), which revealed that software projects often contain a large amount of unreadable code, it is important to note that the contexts are different. The prior study analyzed real-world source code repositories with incremental changes, whereas this study focuses on single snapshot code snippets, which do not necessarily reflect production-ready code.

The *Residual* column shows the standardized residuals of the *Chi-squared* test. These values indicate the contribution of each cell to the overall test statistic by measuring the difference between observed and expected frequencies under the null hypothesis of independence. Conventionally, values beyond the range $[-2, +2]$ are considered statistically significant, suggesting that the observed frequency is unlikely to have occurred by chance. In this table, the *Unreadable* category for ChatGPT 2025 presents a standardized residual below -2 , suggesting that this LLM generated fewer unreadable snippets than expected. In contrast, Stack Overflow presents a standardized residual above $+2$ in the same category, indicating that it produced more unreadable snippets than expected.

3.3.2.1 Correlation Between Readability Warnings and Metrics

As shown in RQ₁, Stack Overflow contained significantly more readability warnings than the four LLM baselines. The metrics reported in RQ₂ indicate that Gemini produces code snippets with higher cognitive complexity, while Scalabrino’s model suggests that only Stack Overflow includes more unreadable code than expected. Based on these observations, we performed a correlation analysis between these metrics and the readability warnings to examine whether there is any relationship between them.

To verify if there is a correlation between the number of readability warnings and the Scalabrino’s readability metric, the non-parametric Kruskal and Wallis (1952) test was applied. This test was selected because the data distributions are non-normal and the readability metric is ordinal (readable, undetermined, and unreadable). In addition, when the test indicated significant differences among classes, the Dunn post hoc test was applied to identify which specific pairs of classes differed. The Kruskal–Wallis did not reveal significant differences between Scalabrino’s classes and the number of readability warnings for most baselines. The only exception was ChatGPT 2023, in which unreadable snippets contained significantly more warnings than readable and undetermined snippets. This indicates that the number of SonarLint warnings is generally not in line with Scalabrino’s readability classification.

To examine whether there is a correlation between the number of readability warnings and the understandability metric (cognitive complexity), the non-parametric Spearman (1904) test was applied. This method was chosen because the data distributions are non-normal and the variables under analysis are continuous. The results did not show significant correlation when all baselines were aggregated. At the individual level, Stack Overflow ($\rho = 0.123$, $p < 0.05$) and Gemini ($\rho = 0.167$, $p < 0.05$) presented weak but significant positive correlations, indicating that snippets with more warnings tended to exhibit greater cognitive complexity. In contrast, DeepSeek showed a weak but significant negative correlation ($\rho = -0.126$, $p < 0.05$), suggesting that warnings were more frequent in snippets with lower cognitive complexity. For both ChatGPT baselines, no significant correlations were observed.

When comparing the number of readability warnings and the LOC metric, Spearman’s test revealed a very weak ($\rho = 0.081$, $p < 0.05$) but statistically significant positive correlation, suggesting that larger snippets tend to contain slightly more warnings. Stack Overflow ($\rho = 0.188$, $p < 0.05$) and Gemini ($\rho = 0.266$, $p < 0.05$) showed weak but significant positive correlations, indicating that snippets with more LOC are more likely to include readability warnings. For ChatGPT 2023, ChatGPT 2025, and DeepSeek, no significant associations were found.

These results suggest that the number of SonarLint readability warnings is weakly influenced by the metrics. In other words, the presence of warnings does not consistently align with other indicators of readability and understandability.

RQ₂: How do the code snippets recommended by LLMs compare to those authored by human developers in terms of readability and understandability metrics? Gemini presents significantly higher values in both LOC and Cognitive Complexity metrics. Interestingly, however, it does not contain more readable or unreadable code snippets than expected. Stack Overflow code snippets include more unreadable examples than expected, while GPT 2025 shows fewer unreadable snippets than expected.

3.3.3 Readability Comparison Under Types of Warnings

SonarLint rules) The official documentation for each SonarLint rule is available at: [https://rules.sonarsource.com/java/RSPEC-\[RULE_ID\]>](https://rules.sonarsource.com/java/RSPEC-[RULE_ID]>). To consult a specific rule, it is sufficient to replace [RULE_ID] with the corresponding identifier. For example, the documentation for rule *java:S5843* can be accessed at: <https://rules.sonarsource.com/java/RSPEC-5843>>.

Figure 19 shows the results of the open coding process applied to 77 SonarLint rules related to code readability, as reported in Section “Classifying Types of Code Readability Rules in SonarLint”. This analysis revealed 21 different subcategories, which were grouped into five categories: *Clarify Code Intent*, *Prefer Concise Code*, *Clean Up*, *Convention*, and *Improve Code Modularity*. Each subcategory presents the number of associated rules. For example, *Improve Regex* includes four SonarLint rules: (*java:S5843*, *java:S6035*, *java:S6353*, and *java:S6397*). The total number of rules per category reflects the sum of its subcategories (e.g., Clarify Code Intent comprises 23 rules across eight subcategories).

Table 7 presents the number of readability warnings and the number of affected code snippets per category, grouped by baseline (Stack Overflow, ChatGPT 3.5, with code

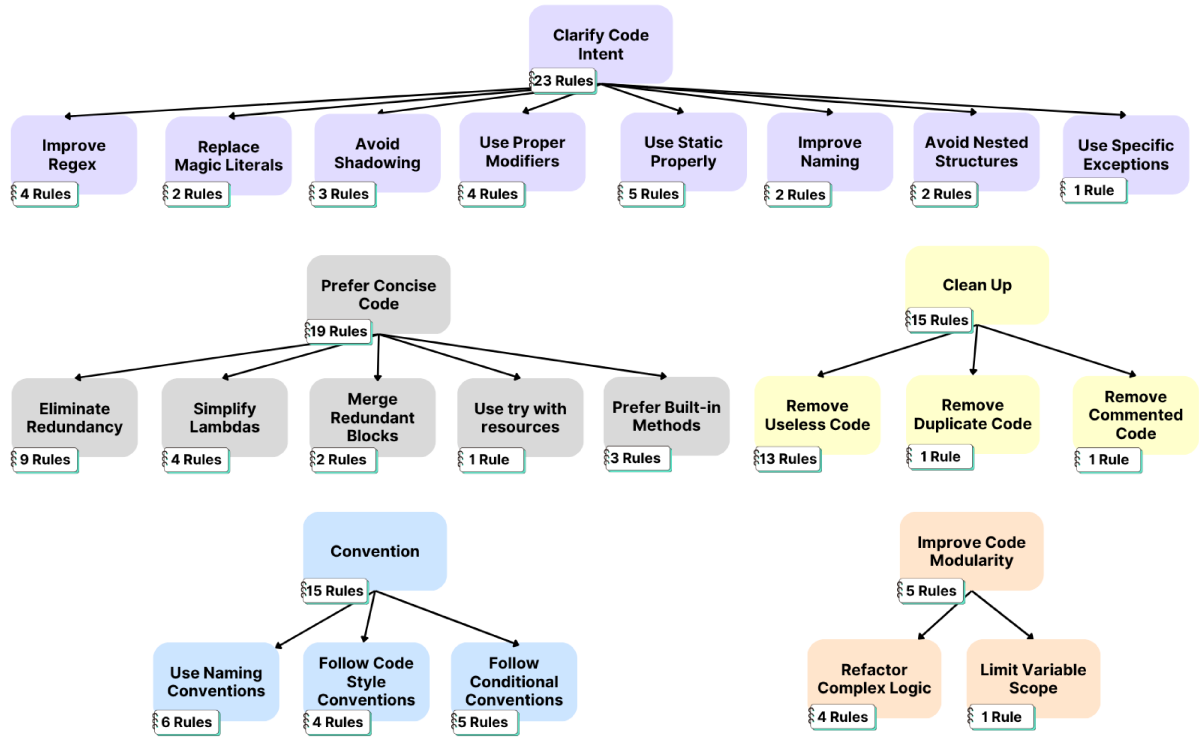


Figure 19 – SonarLint code readability rules organized into five main categories and 21 subcategories.

snippets extracted in March 2023 (DANTAS; ROCHA; MAIA, 2023b), ChatGPT 4o version 2025, DeepSeek, and Gemini). The column *Snip.* indicates the number of code snippets with at least one SonarLint readability warning (e.g., 216 out of 327 Stack Overflow snippets contained at least one warning). The column *Warn.* reports the total number of readability warnings found (e.g., Stack Overflow generated 623 readability warnings across the 216 affected snippets). While the *Warn.* count is the sum of warnings across all subcategories, the *Snip.* count may not equal the sum of affected snippets per subcategory, as a single code snippet could contain warnings from multiple subcategories.

Table 7 – Number of readability warnings and affected code snippets per category and baseline (*Snip.*: number of code snippets affected; *Warn.*: total number of readability warnings). ($n = 327$ snippets)

Category	S.Overflow		GPT 2023		GPT 2025		DeepSeek		Gemini	
	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.
Clarify Code Intent	54	90	19	24	21	26	24	31	39	54
Prefer Concise Code	70	91	31	35	11	12	14	17	21	26
Clean Up	132	303	59	155	45	98	81	209	53	97
Convention	51	130	5	7	6	11	2	3	6	14
Improve Code Modularity	4	9	4	4	3	4	1	1	6	7
Total	216	623	97	225	80	151	114	261	104	198

3.3.3.1 Clarify Code Intent

The *Clarify Code Intent* category addresses the importance of making code more explicit and expressive in terms of its purpose and functionality. For example, in five out of the 23 rules within this category, the keyword *confusing* is tagged by the SonarLint tool, indicating that these rules target code constructs that may be unclear and require easier-to-understand implementation.

Table 8 presents the number of readability warnings and snippets affected for each subcategory. Interestingly, the ChatGPT 3.5 baseline, whose code snippets were extracted in 2023, shows the lowest number of both affected snippets and readability warnings, although the difference is only slightly lower than that of ChatGPT 2025 and DeepSeek. In contrast, Gemini and Stack Overflow exhibit a higher number of affected code snippets and readability warnings.

Table 8 – Number of readability warnings and affected code snippets per subcategory and baseline in the *Clarify Code Intent* category (*Snip.*: number of code snippets affected; *Warn.*: total number of readability warnings). ($n = 327$ snippets)

Subcategory	S.Overflow		GPT 2023		GPT 2025		DeepSeek		Gemini	
	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.
Use Proper Modifiers	27	40	6	8	7	9	5	7	3	4
Replace Magic Literals	12	22	5	6	5	6	5	6	18	26
Use Static Property	7	8	4	6	6	6	5	5	8	10
Improve Regex	3	10	0	0	0	0	3	5	2	3
Use Specific Exception	4	5	2	2	2	3	2	4	6	8
Improve Naming	3	3	1	1	0	0	2	2	2	2
Avoid Nested Structures	1	1	0	0	1	1	1	1	1	1
Avoid Shadowing	1	1	1	1	1	1	1	1	0	0
Total	54	90	19	24	21	26	24	31	39	54

The *Use Proper Modifiers* subcategory includes four SonarLint rules (i.e., *java:S1444*, *java:S1165*, *java:S3740*, *java:S1161*) that emphasize the use of annotations such as `@Override` to explicitly indicate that a method is overriding a superclass method, modifiers like *final*, or declaring generic types without omitting their parameters (e.g., `List<String>` instead of `List`). The common goal of these rules is to make the code’s behavior and intent more explicit and easier to understand. In this context, Stack Overflow had the worst performance, with readability warnings found in 27 code snippets, totaling 40 instances. More precisely, most of the readability warnings involved the absence of generic type declarations in collections (e.g. `List list = new ArrayList();`). In contrast, Gemini achieved the best performance, largely because all other baselines failed to use the `@Override` annotation, whereas Gemini did not exhibit any readability warnings.

The *Replace Magic Literals* subcategory includes two SonarLint rules (i.e., *java:S4719* and *java:S1192*) that typically target the use of *hardcoded* string literals, recommending that these values be extracted into named constants or replaced with predefined API constants to improve readability. Interestingly, Gemini showed the worst performance

in this subcategory. For example, according to the rule *java:S1192* that discourages duplicated string literals, Gemini exhibited twice as many occurrences compared to others. In contrast, ChatGPT 2023, ChatGPT 2025 and DeepSeek had the best performance.

The *Use Static Property* subcategory includes five SonarLint rules (that is, *java:S3252*, *java:S2209*, *java:S1171*, *java:S2440*, and *java:S3010*) that relate to the proper use of static classes, methods, and fields (e.g. avoiding the instantiation of utility classes that contain only static members). Gemini had the worst performance in this subcategory, although the results in other baselines were relatively similar. For example, in query ID = 129: “how do you display an image?”, all baselines used the `javax.swing` API and included the line `frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`. SonarLint shows readability warnings for the rule *java:S3252* because the constant `EXIT_ON_CLOSE` is declared in the `WindowConstants` interface, which is implemented by the `JFrame` class. Therefore, SonarLint recommends referencing constants directly from their declaring class or interface.

The *Improve Regex* subcategory includes four SonarLint rules (i.e., *java:S5843*, *java:S6035*, *java:S6353*, and *java:S6397*) that refer to simplify the construct of regex expressions. For example, in query ID = 249 “regex to remove numbers”, all baselines produced a pattern that corresponds to numeric characters:

ChatGPT 2023: `"\d"`

ChatGPT 2025: `"\d+"`

DeepSeek: `"[0-9]"`

Gemini: `"[0-9]"`

StackOverflow: `"[0-9]*$"`

According to SonarLint rule *java:S6353*, regular expressions should be written concisely. From this perspective, only ChatGPT 2023 and ChatGPT 2025 provided more concise and recommended regex patterns. The equivalence is not observed among the patterns `[0-9]`, `\d`, and `\d+`. However, all LLMs produced functionally similar output, with ChatGPT 2023, Gemini, and DeepSeek generating character-by-character removals (i.e., `replaceAll()`) and ChatGPT 2025 using a sequence-based removal. In contrast, the Stack Overflow snippet did not fully address the query and was classified with semantic adherence level 4, as described in Section *Assessing Semantic Adherence of Code Snippets with Input Queries*.

The *Use Specific Exception* subcategory includes a single SonarLint rule (*java:S112*), which is activated when developers choose to throw generic exceptions (e.g. `java.util.Exception`) instead of using more specific or custom exceptions for application-level errors. Adopting specific exceptions leads to more accurate and informative error handling. For example, in query ID = 11: “How do I encrypt something with AES”, ChatGPT 2025, Gemini and DeepSeek generated methods such as `encrypt` and `decrypt` that throw

`java.util.Exception`. In contrast, the Stack Overflow snippet recommended by CROKAGE throws more specific exceptions, such as `UnsupportedEncodingException`. An interesting case is ChatGPT 2023, which places all the code inside the `main` method and throws a general exception from there. However, SonarLint does not show a readability warning of this rule when the generic exception is thrown directly from the `main` method.

The *Improve Naming* subcategory includes two SonarLint rules (*java:S1700* and *java:S6213*), which recommend improving naming in specific situations, for example, when a field has the same name as its enclosing class or when restricted identifiers such as `var` are used as variable names. Among all baselines, ChatGPT 2025 was the only one that did not show any readability warnings in this subcategory.

The last two subcategories had few instances. The *Avoid Nested Structures* subcategory includes two SonarLint rules (*java:S3358* and *java:S1141*) that aim to reduce nested code constructs, such as nested ternary operations or try-catch blocks. Finally, the *Avoid Shadowing* subcategory includes three SonarLint rules (*java:S1117*, *java:S2387*, and *java:S4977*) that focus on preventing naming conflicts where local variables, subclasses, or type parameters shadow existing identifiers, potentially leading to confusion.

Finding 2) In the Clarify Code Intent category, the two versions of ChatGPT and DeepSeek achieved the best results according to the readability warnings reported by SonarLint, as their warnings were more evenly distributed in the different subcategories. In contrast, Gemini showed a high number of warnings related to the use of duplicate string literals in its code snippets. Meanwhile, the Stack Overflow code recommended by CROKAGE exhibited a large number of readability warnings due to the omission of common Java modifiers, such as the use of generics in collections.

3.3.3.2 Prefer Concise Code

The *Prefer Concise Code* focuses on eliminating redundant or boilerplate code, reducing verbosity, and thus making the code easier to read. For example, in 11 out of the 20 rules within this category, the keywords *clumsy* or *redundant* are tagged by the SonarLint tool, indicating that these rules aim to refactor the code by removing syntactic clutter and simplifying its structure.

Table 9 presents the number of readability warnings and code snippets affected for each subcategory. In this subcategory, ChatGPT 2025 and DeepSeek show a lower number of affected code snippets and readability warnings compared to the other baselines.

The *Eliminate Redundancy* subcategory includes nine SonarLint rules (i.e., *java:S1858*, *java:S1125*, *java:S2293*, *java:S1905*, *java:S2129*, *java:S2130*, *java:S1126*, *java:S1488*, and *java:S3626*) that recommend removing redundant code, such as unnecessary calls to the `toString()` method, redundant casts, redundant `String` instantiations, and overly ver-

Table 9 – Number of readability warnings and affected code snippets per subcategory and baseline in the *Prefer Concise Code* category (*Snip.*: number of code snippets affected; *Warn.*: total number of readability warnings). ($n = 327$ snippets)

Subcategory	S.Overflow		GPT 2023		GPT 2025		DeepSeek		Gemini	
	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.
Eliminate Redundancy	49	58	16	19	0	0	4	4	3	3
Simplify Lambdas	17	20	8	8	7	8	9	12	7	11
Prefer Built-in Methods	4	7	4	4	4	4	1	1	8	9
Merge Redundant Blocks	4	4	3	3	0	0	0	0	1	1
Use Try With Resources	2	2	1	1	0	0	0	0	2	2
Total	70	91	31	35	11	12	14	17	21	26

bose boolean expressions, among others. In this subcategory, Stack Overflow and ChatGPT 2023 exhibited more readability warnings compared to the other baselines. For example, Stack Overflow shows 33 readability warnings of rule *java:S2293*, while ChatGPT 2023 shows 11. This rule recommends the use of the diamond operator `<>` when instantiating collections to reduce redundancy.

The *Simplify Lambdas* subcategory includes four SonarLint rules (*java:S1612*, *java:S1604*, *java:S1611*, and *java:S1602*) that recommend simplifying lambda expressions and method references, by refactoring anonymous inner classes or avoiding nested lambda operations that contain only a single statement. In this subcategory, Stack Overflow exhibited more readability warnings compared to the LLMs. For example, in query ID = 312: “ordered set”, all baselines constructed a `java.util.TreeSet` as the data structure. However, while ChatGPT (2023 and 2025) and DeepSeek used the default comparator for primitive types, both Stack Overflow and Gemini unnecessarily created an anonymous class for the comparison.

The *Prefer Built-in Methods* subcategory includes three SonarLint rules (*java:S7158*, *java:S2140*, and *java:S3012*) that encourage the use of built-in Java API methods rather than implementing custom algorithms for common tasks. For example, instead of manually writing a method to copy arrays, developers are encouraged to use `Arrays.copyOf()`. Interestingly, DeepSeek exhibited the lowest number of affected classes and readability warnings in this subcategory. In query ID = 162: “how to insert an element into an array at a given position”, ChatGPT (2023 and 2025) and Gemini used a `for` loop to manually copy elements from the original array to a new one. Only DeepSeek used the built-in `System.arraycopy()` method.

The *Merge Redundant Blocks* subcategory includes two SonarLint rules (i.e., *java:S1066* and *java:S2147*) that encourage merging collapsible `if` statements and `catch` blocks to reduce redundancy. Finally, the use of *try-with-resources* is represented by a single rule, *java:S2093*, which simplifies the traditional *try-catch-finally* structure when working with resources by reducing clutter code.

Finding 3) ChatGPT 4o (2025) produced code snippets with fewer clutter-related code warnings detected by SonarLint compared to ChatGPT 3.5 (2023). Similarly, DeepSeek generated fewer readability warnings than Gemini. The code written by humans on Stack Overflow contained more redundancy warnings than the code generated by the LLMs.

3.3.3.3 Clean Up

The *Clean Up* category focuses on eliminating unnecessary code, such as duplicated or unused elements that are never executed. In particular, eight out of the 15 rules within this category, the keyword *unused* is tagged by the SonarLint tool, indicating that these rules aim to improve the code by removing elements that do not serve a functional purpose. Table 10 presents the number of warnings and code snippets affected for each subcategory. ChatGPT 2025 and Gemini show a lower number of affected code snippets and readability warnings compared to the other baselines.

Table 10 – Number of readability warnings and affected code snippets per subcategory and baseline in the *Clean Up* category (*Snip.*: number of code snippets affected; *Warn.*: total number of readability warnings). ($n = 327$ snippets)

Subcategory	S.Overflow		GPT 2023		GPT 2025		DeepSeek		Gemini	
	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.
Remove Useless Code	124	280	56	150	41	94	74	197	31	66
Remove Commented Code	16	23	3	4	4	4	9	10	28	31
Remove Duplicate Code	0	0	1	1	0	0	1	2	0	0
Total	132	303	59	155	45	98	81	209	53	97

The *Remove Useless Code* subcategory includes 13 SonarLint rules (i.e., *java:S1481*, *java:S1854*, *java:S1068*, *java:S1144*, *java:S1172*, *java:S2326*, *java:S1130*, *java:S1116*, *java:S1186*, *java:S108*, *java:S2094*, *java:S4925*, and *java:S3457*) that recommend removing unused elements such as variables, assignments, fields, methods, empty statements, classes, and others. In this subcategory, Gemini exhibited fewer warnings compared to the other baselines. However, all baselines presented a number of useless code warnings. This is largely because many generated code snippets contain elements that are declared but never used.

For example, in query ID = 5: “final keyword”, all LLMs generated a Java class with a constant declared using the **final** keyword and assigned a value, but the constant was never used in practice, such as being printed or involved in any operation. As a result, warnings of rules such as *java:S1481 – Unused local variables should be removed* and *java:S1854 – Unused assignments should be removed* were shown across all baselines. Some baselines exhibited fewer warnings simply because the variables or assignments were later used to print their value on the screen. For example, in query ID = 159: “Cast an

int”, Gemini printed the cast variable, while the other baselines performed the cast but did not use the result in the rest of the code.

The *Remove Commented Code* subcategory includes only one SonarLint rule, *java:S125*, which recommends avoiding the inclusion of source code within comments. Interestingly, Gemini had the worst performance in this subcategory, as it tends to include alternative solutions as commented-out code. In contrast, ChatGPT and DeepSeek typically present alternatives as separate code snippets within their explanations. As a result, Gemini code snippets often have a higher number of LOC, cluttered with large sections of commented-out content.

The *Remove Duplicate* subcategory includes only one SonarLint rule, *java:S1871*, which targets duplicate implementations within conditional branches. It is important to note that this rule does not perform a comprehensive search for duplicate code across the entire code snippet, but rather focuses specifically on identifying and eliminating redundancy in conditional structures.

Finding 4) ChatGPT 4o (2025) produced cleaner code snippets compared to the 2023 version and DeepSeek, and performed on par with Gemini. Although Gemini tended to produce more comprehensive code, making proper use of variables and assignments, it also included a large amount of commented-out code, which contributed to increased unnecessary code.

3.3.3.4 Convention

The *Convention* category includes a set of stylistic and structural guidelines that the code snippets are expected to follow. In nine of the 15 rules within this category, the keyword *convention* is explicitly tagged by the SonarLint tool, indicating that these rules aim to enforce the adherence to established coding standards. Table 11 presents the number of readability warnings and affected code snippets for each subcategory. In this category, Stack Overflow accounts for the majority of warnings observed across the entire data set.

Table 11 – Number of readability warnings and affected code snippets per subcategory and baseline in the *Convention* category (*Snip.*: number of code snippets affected; *Warn.*: total number of readability warnings). ($n = 327$ snippets)

Subcategory	S.Overflow		GPT 2023		GPT 2025		DeepSeek		Gemini	
	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.
Use Naming Conventions	24	77	1	1	1	1	0	0	4	10
Follow Code Style Conventions	29	43	4	6	5	7	1	2	1	12
Follow Conditional Conventions	5	10	0	0	1	3	1	1	1	2
Total	51	130	5	7	6	11	2	3	6	14

The *Use Naming Conventions* subcategory includes six SonarLint rules (i.e., *java:S117*, *java:S100*, *java:S3008*, *java:S101*, *java:S116*, and *java:S115*), all of which recommend that variables, classes, methods, constants and field names follow naming patterns defined by regular expressions in SonarLint. These conventions include, for example, the use of *camelCase* for variables and methods, *PascalCase* for class names, and *lowercase* for attributes. Stack Overflow exhibited a significant number of warnings in this subcategory, mainly due to the frequent use of the *snake_case* pattern, which is common in other programming languages such as Python. This suggests that while LLMs are likely trained on examples that reinforce Java naming conventions, some Stack Overflow contributors may not be specialized in Java programming.

The *Follow Code Style Conventions* subcategory includes four SonarLint rules (i.e., *java:S1197*, *java:S1659*, *java:S1124*, and *java:S2234*), which all apply code style guidelines, such as avoiding the declaration of multiple variables on the same line, maintaining the correct order of modifiers, and other stylistic recommendations. For example, in query ID = 93: “Fibonacci sequence, both ChatGPT 2025 and Stack Overflow declared two variables on the same line, while the other baselines declared each integer on a separate line, better following the recommended style.

The *Follow Conditional Conventions* subcategory includes five SonarLint rules (i.e., *java:S3973*, *java:S128*, *java:S131*, *java:S1301*, and *java:S127*), which primarily reinforce code style conventions related to conditional structures. These include recommendations such as using curly braces even for single-line `if` statements, including `default` cases and `break` statements in `switch` constructs, among others. As in other convention-related subcategories, Stack Overflow exhibits a higher number of warnings in this subcategory.

Finding 5) Stack Overflow has the majority of warnings related to Java coding guidelines defined by SonarLint rules. One potential reason is that, while LLMs are trained in large data sets that reinforce standard Java conventions, the code written by humans on Stack Overflow can vary depending on the contributor’s background. In particular, developers who are more experienced in other programming languages may write Java code that does not strictly adhere to Java-specific style and structural conventions.

3.3.3.5 Improve Code Modularity

The *Improve Code Modularity* category involves deeper structural changes, either at the method or class level. This category is less likely to appear in code snippets, as they are typically short and often not complex enough to violate rules related to long methods or large classes. In particular, in three out of the five rules within this category, the keyword *brain-overload* is explicitly tagged by the SonarLint tool, indicating that these

rules aim to reduce cognitive load by promoting better modularity. Table 12 presents the number of warnings and code snippets affected for each subcategory. In this category, although Stack Overflow shows the highest number of warnings and Gemini the highest number of affected snippets, the differences among baselines are not substantial as in the other categories.

Table 12 – Number of readability warnings and affected code snippets per subcategory and baseline in the *Improve Code Modularity* category (*Snip.*: number of code snippets affected; *Warn.*: total number of readability warnings). ($n = 327$ snippets)

Subcategory	S.Overflow		GPT 2023		GPT 2025		DeepSeek		Gemini	
	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.
Refactor Complex Logic	3	8	2	2	1	1	0	0	5	6
Limit Variable Scope	1	1	2	2	2	3	1	1	1	1
Total	4	9	4	4	3	4	1	1	6	7

The *Refactor Complex Logic* subcategory includes four SonarLint rules (i.e., *java:S6541*, *java:S3776*, *java:S1479*, and *java:S135*) that aim to reduce cognitive complexity or the number of responsibilities concentrated within a single method. In query ID = 218: “convert JSON to CSV”, Gemini generated a method named `convertJsonArrayToCsv()` with a cognitive complexity score = 19 and a nesting depth = 4, involving multiple loops and conditional statements nested within each other. Although ChatGPT 2023 and 2025 also used loops, their methods had a nesting depth of 3 and did not show warnings of cognitive complexity.

The *Limit Variable Scope* subcategory includes a single SonarLint rule, *java:S1450*, which recommends limiting the scope of variables by converting fields that are only used within the methods into local variables.

Finding 6) As expected, SonarLint did not report a large number of code snippets affected by modularity-related warnings, since code snippets are typically short pieces of code designed to solve specific tasks, rather than full classes as found in production-ready software.

RQ₃: How does the readability of the code snippets recommended by LLMs compare to those authored by human developers, based on the number and type of warnings generated by ASAT? The open coding process resulted in a catalog of 21 types of code readability rules from SonarLint, grouped into five categories. Overall, the code snippets generated by four LLMs produce fewer warnings compared to those written by human developers on Stack Overflow, particularly in generating cleaner code and adhering more closely to Java coding conventions.

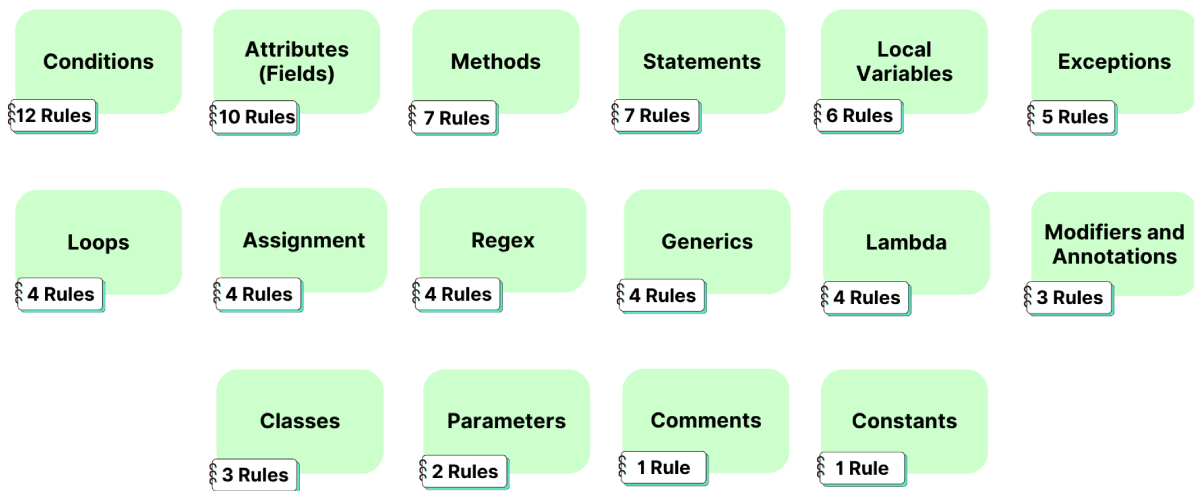


Figure 20 – SonarLint code readability rules organized into 16 syntactic structures.

3.3.4 Readability Comparison in Syntactic Elements Categorization

For this research question, 16 types of syntactic elements affected by the SonarLint rules were identified, as shown in Figure 20. The number of affected code snippets and the corresponding readability warnings detected by SonarLint were also extracted, as illustrated in Table 13. The syntactic elements perspective provides a complementary view of the types of readability warnings, that is, it reveals which structural aspects of the code are affected by readability warnings, in order to compare the snippets generated by LLMs with those authored by humans on Stack Overflow.

From a perspective of syntactic elements, all baselines exhibit a high number of warnings related to *Local Variables* and *Assignments*. The former is due to the frequent presence of unused variables in baselines, with Stack Overflow being particularly affected by variable names that do not conform to Java naming conventions. The latter is explained by the presence of unused assignments found in all baselines.

In the context of the *Generics* syntactic structure, Stack Overflow exhibits numerous warnings related to the use of raw types. For example, using `List myList;` instead of `List<String> myList;`, as well as omitting the diamond operator, such as writing `List<String> strings = new ArrayList<String>();` instead of `List<String> strings = new ArrayList<>();`. Interestingly, ChatGPT 2023 is the only LLM baseline that presents 14 occurrences of missing diamond operators, a pattern that is not observed in the other LLMs. In *Methods*, Stack Overflow presents more empty functions, but Gemini had more methods with high cognitive complexity.

In the *Statements* syntactic structure, Gemini exhibits a high number of duplicate string literals (24 occurrences), while Stack Overflow shows more warnings related to the omission of standard *charset*s, for example, using `string.getBytes("UTF-8")` instead of

Table 13 – Number of readability warnings and affected Code Snippets in ChatGPT and Stack Overflow, Grouped by Syntactic Structure (Snip.: number of code snippets affected; Warn.: total number of readability warnings). ($n = 327$ snippets)

syntactic Structure	S.Overflow		GPT 2023		GPT 2025		DeepSeek		Gemini	
	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.	Snip.	Warn.
Local Variables	119	190	49	70	35	45	59	89	25	39
Assignment	98	128	48	72	30	37	62	91	16	22
Generics	47	69	12	15	1	1	2	2	2	5
Methods	22	29	2	2	1	1	4	6	8	8
Statements	19	34	8	9	5	6	7	8	21	29
Attributes	16	43	10	17	15	24	15	20	12	17
Comments	16	23	3	4	4	4	9	10	28	31
Lambda	17	20	8	8	7	8	9	12	7	11
Conditions	11	16	3	3	2	4	3	4	5	6
Loops	11	17	6	6	3	3	1	1	7	8
Exceptions	10	11	6	6	4	5	2	4	10	12
Modifiers	9	15	4	6	6	8	4	5	1	1
Classes	5	6	3	5	2	2	1	1	2	2
Parameters	3	4	1	2	2	3	2	3	3	4
Regex	3	10	0	0	0	0	3	5	2	3
Constant	1	8	0	0	0	0	0	0	0	0
Total	216	623	97	225	80	151	114	261	104	198

`string.getBytes(StandardCharsets.UTF_8)`. In the *Attributes* category, Stack Overflow is the only baseline that includes field names that violate Java’s naming conventions. However, all baselines declare several private fields that are never used in the code. Regarding *Comments*, as mentioned in the previous section, both Stack Overflow and Gemini contain a higher number of code fragments commented on.

In the *Lambda* syntactic structure, DeepSeek shows a high number of unnecessary block usages for lambdas (8 occurrences), such as using `(a, b) -> return a + b;` instead of the more concise `(a, b) -> a + b`. Stack Overflow, on the other hand, exhibits more warnings related to the use of anonymous classes where lambdas could be used (12 occurrences). In the *Conditions* category, although it includes 10 associated rules, most of them have only a few warnings. The total number of warnings results from the aggregation of several distinct rules, but individually none of them show a significant number of occurrences. Regarding *Loops*, Stack Overflow presents more warnings related to empty nested blocks, whereas Gemini exhibits more cases where loops are used to perform tasks that could be replaced by built-in methods.

In the *Exceptions* syntactic structure, Gemini and Stack Overflow present a larger number of cases where generic exceptions are thrown. In the *Modifiers* category, interestingly, all baselines except Gemini lack the use of the `@Override` annotation. Stack Overflow is also the only baseline that receive warnings about the recommended order of modifiers, such as using `static public void main(String[] args)` instead of the conventional `public static void main(String[] args)`. Regarding *Classes*, ChatGPT

2023 includes instances where classes containing only static methods are unnecessarily instantiated. Stack Overflow also presents several warnings of Java’s class-naming conventions.

In the three remaining syntactic structures, the *Constants* category includes eight warnings related to naming conventions, all concentrated in a single class. The results for the *Regex* structure align with those observed in the open coding analysis. Lastly, the *Parameters* structure did not show any significant differences in the baselines.

RQ₄: How does the readability of the code snippets recommended by LLMs compare to those authored by human developers under the perspective of the syntactic elements affected by the readability warnings? Complementing the open coding analysis, the syntactic elements revealed that Stack Overflow exhibits the highest number of rule warnings in most syntactic structures. However, there are some isolated cases where the LLMs performed worse. For example, Gemini showed a higher number of duplicated string literals, while DeepSeek exhibited more redundant lambda expressions. Interestingly, both versions of ChatGPT, particularly the 2025 model, denote more balanced behavior. Although they do present some warnings, they do not stand out negatively in any specific syntactic aspect compared to the other baselines.

3.3.5 Assessing the Evolution of Code Readability Between ChatGPT 2023 and 2025

As mentioned previously, the code snippets generated by the free version of ChatGPT was collected at two different points in time. March 2023, as documented by (DANTAS; ROCHA; MAIA, 2023b), and May 2025. As reported in the RQ₁, the Nemenyi’s post hoc test with 95% (p -value < 0.05) was applied and found no statistically significant differences between the two versions. However, since ChatGPT 2025 produced fewer SonarLint readability warnings (151 versus 225) and affects fewer classes (80 versus 97), a discussion was held about the most notable differences observed between the two versions.

Of the 77 SonarLint rules associated with improving code readability, only 43 were violated by ChatGPT 2023 or ChatGPT 2025. Among these 43 rules, 20 had fewer warnings in ChatGPT 2025, 15 had fewer warnings in ChatGPT 2023, and 8 had the same number of warnings across both models.

Analyzing the 20 rules where ChatGPT 2025 had fewer warnings, the newer version demonstrates a reduction in unused assignments and local variables. For example, in query ID = 58: “Make HTTP call”, ChatGPT 2023 assigns the response code to a variable named `status` using the code `int status = con.getResponseCode();`, but this variable is never used elsewhere in the code, resulting in two warnings. In contrast,

ChatGPT 2025 retrieves the same response code, but proceeds to print both the status code and the response body to the screen. Although printing output with `System.out` may not be suitable for production environments, developers often seek code snippets that are didactic rather than production-ready.

In other cases, ChatGPT 2023 also produced more cluttered code, such as declaring unused parameters in methods, declaring a variable only to immediately return or throw it, using unnecessarily complex loops with multiple break or continue statements, and having multiple catch blocks that could be combined. A particularly interesting case is the use of the diamond operator: ChatGPT 2023 contains 13 warnings, while ChatGPT 2025 has none. Since this feature was introduced in Java 7 (released in 2011), it is not a recent addition that would justify its missing in a 2023 model. However, LLMs are trained on the code that developers commonly write, regardless of whether a feature is new or old. For example, lambdas were introduced in Java 8 (released in 2014), and while ChatGPT 2025 exhibits fewer warnings related to the use of anonymous classes, ChatGPT 2023 presents fewer warnings concerning the use of method references in place of lambda expressions.

In the 15 SonarLint rules in which ChatGPT 2023 raised fewer warnings, only two of the 15 showed a difference of less than three warnings in favor of ChatGPT 2023, that is, in most cases, the number of warnings was similar between the two models. For example, ChatGPT 2025 had two more occurrences of declaring unused private fields (13 versus 11). However, upon examining the code snippets, some of these fields were accompanied by comments such as `// Getters and setters`, suggesting that the developer is expected to implement them. Although this approach is intuitive, it results in generated code that requires additional action from the user to be used properly.

RQ₅: To what extent has ChatGPT improved in generating readable code snippets compared to its older version? Although there are no statistically significant differences in the number of readability warnings raised by SonarLint, ChatGPT 2025 tends to produce less clutter and unnecessary code. However, both models exhibit similar performance on most SonarLint rules.

3.4 Discussion of the Results

In this section, insights are presented derived from the analysis of SonarLint warnings shows for both LLM-generated and Stack Overflow code snippets. By combining the findings of RQ₃ and RQ₄, Stack Overflow developers generally produce more readability warnings related to naming conventions and code pattern guidelines. These snippets also tend to contain unnecessary elements, such as commented-out code, empty methods,

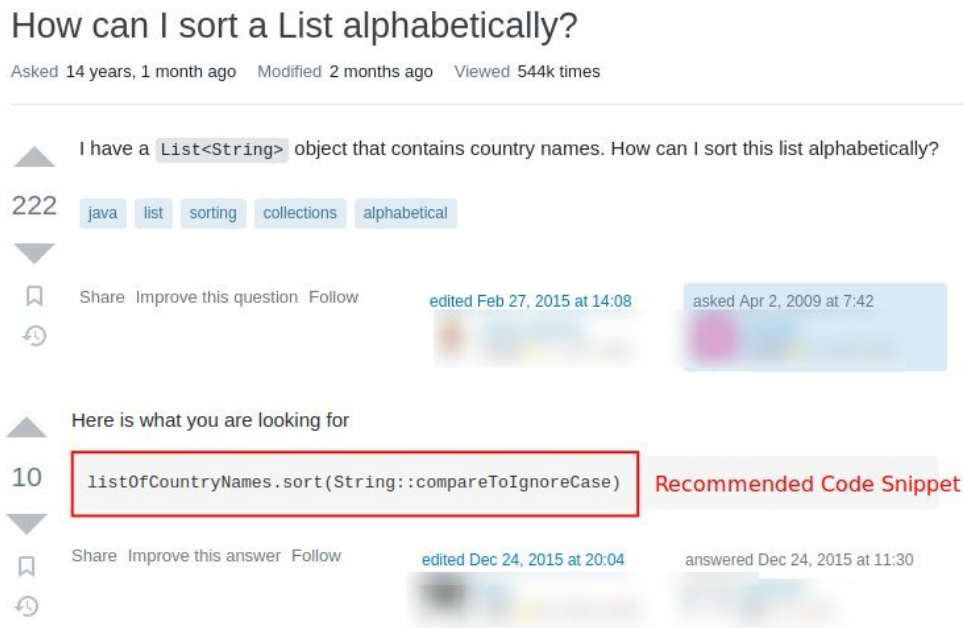


Figure 21 – Stack Overflow thread with the recommended post by CROKAGE for the input query ID = 164: “how to sort a list”

and other forms of clutter code. However, while LLM-generated code typically results in fewer SonarLint warnings, some outliers were identified. For example, Gemini frequently duplicates string literals, throws more exceptions, and includes excessively commented code. Similarly, DeepSeek often produces some unnecessarily verbose lambda expressions.

One of the key differences between human and automated approaches is in how they respond to programming queries. While search engines like CROKAGE aim to retrieve a link to a human-written code snippet and its accompanying explanation that most closely matches the query, LLMs generate code customized directly to the specific question asked. The semantic adherence analysis also reveals that CROKAGE frequently suggests Stack Overflow posts that reflect the specific environment or assumptions of the user who asked the question. In contrast, LLMs tend to provide more focused and context-independent responses. For example, when asked about sorting a list, CROKAGE recommended a Stack Overflow post that contained only a single-line call to the `sort` method in the specific context of country names (Figure 21). In contrast, ChatGPT 2023 generated a self-contained snippet and an explanation trying to directly address the query (Figure 22). Developer names were obfuscated to preserve privacy.

Finding 7) The code snippets recommended by LLMs tend to be more generic and didactic compared to those found on Stack Overflow. This distinction occurs because the Stack Overflow answers often contain code snippets linked to specific problems raised by the authors of the Stack Overflow questions.

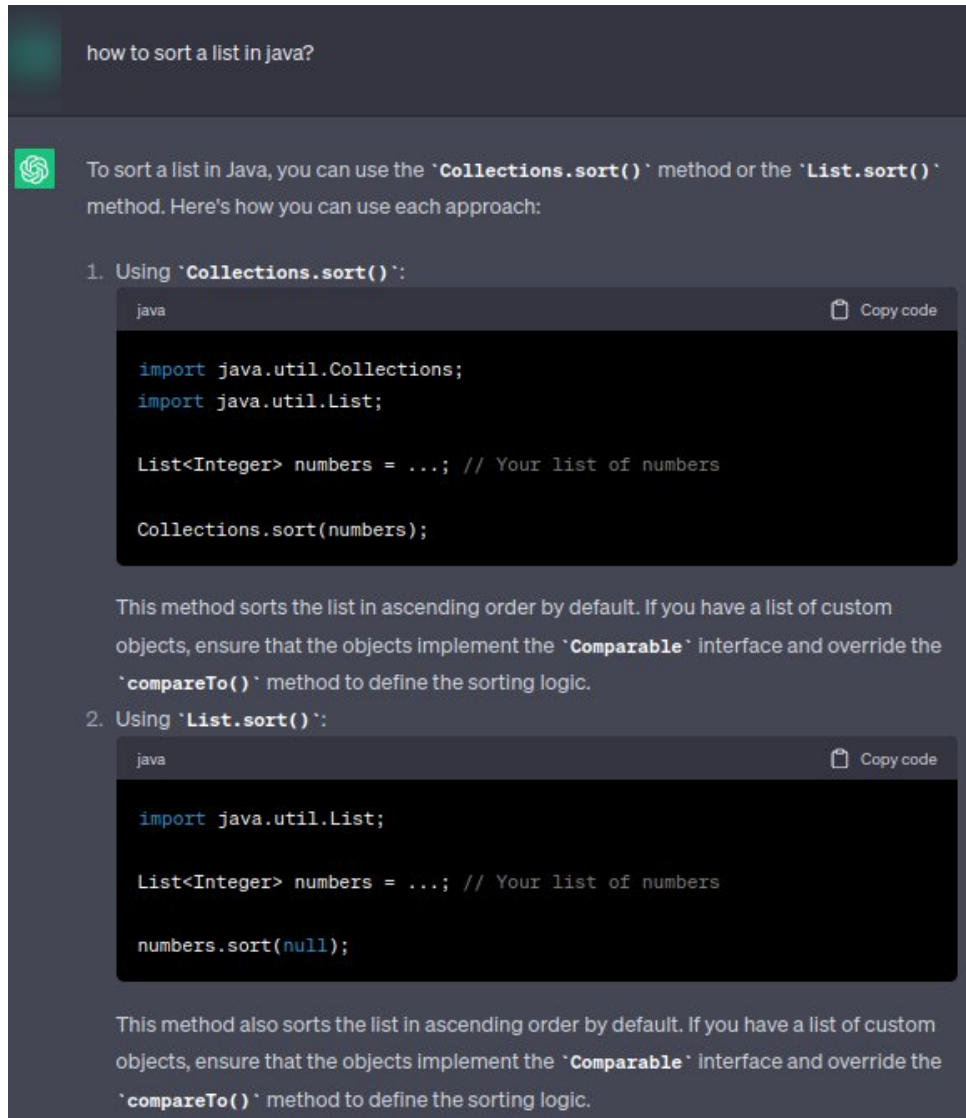


Figure 22 – ChatGPT-generated answer for the input query ID = 164: “how to sort a list”

Another observation concerns the behavior of LLMs with respect to code readability. LLMs generate code based on the internet crowd. As a result, they tend not often to make mistakes in well-established coding conventions widely adopted by the developer community, such as using the camel case for naming in Java. However, for more recent language features that still require time for widespread adoption, LLMs may produce code that shows warnings on these types of SonarLint rules. For example, the *try-with-resources* statement is used inconsistently: in some cases, the LLMs apply it correctly, while in others it omits it. Interestingly, some of the omissions occurred in code involving legacy Java objects, such as `javax.net.ssl.SSLSocket`, suggesting that the LLM may have been influenced by older examples from a time when developers had not yet widely adopted *try-with-resources*.

Another example comes from query ID = 315: “unsafe casting”. In 2023, ChatGPT

generated a variable named *valueOffset*, which violated Java naming conventions according to SonarLint. When repeating the same query in 2025, both ChatGPT and DeepSeek generated the same variable with the same readability warning. Assuming this is a rarely asked question, it is plausible that LLMs rely on a limited set of examples, leading to the replication of similar, potentially outdated code patterns, without particular attention to readability concerns.

An aspect in both cases is that although LLMs may initially ignore the importance of certain language features or readability rules, they are capable of incorporating them when explicitly prompted. For example, if the user asks the LLM to refactor the code to use a specific feature, such as *try-with-resources* or adherence to naming conventions, it will generally follow the suggestion. However, by default, LLMs seem to prioritize commonly observed patterns rather than proactively enforcing best practices that go beyond widespread conventions. Therefore, relying on the LLM first response without refinement or sufficient domain knowledge can lead to code that could contain readability warnings, as illustrated in this study.

Finding 8) Based on the code snippets of the dataset, LLMs do not seem to focus on making the code more readable, but instead aim to directly answer the developer's question by generating a solution based on patterns learned during training. This suggests that generating readable code may require prompt engineering and prior domain knowledge from the developer.

Finally, as shown in the RQ₂ results, a code snippet containing higher LOC or cognitive complexity does not necessarily show a higher number of readability warnings, as shown in the correlation results. For example, Listing 3.1 and Listing 3.2 (Query ID = 62: “parse html with regex in Java”) compare two code snippets generated by ChatGPT 2025 (A) and ChatGPT 2023 (B). Although neither snippet raised any warnings related to SonarLint readability rules, ChatGPT 2025-generated code exhibits higher cognitive complexity due to its `while` loop, and ChatGPT 2023 employs the use of *JSoup* API.

```

1 String html = "<html><body>"
2           + "<a href=\"https://example.com\">Example</a>"
3           + "<a href=\"https://openai.com\">OpenAI</a>"
4           + "</body></html>";
5
6 String regex = "<a\\s+[^>]*href\\s*=\\s*\"([^\"]*)\"[>]*";
7 Pattern pattern = Pattern.compile(regex, Pattern.CASE_INSENSITIVE
8           );
9 Matcher matcher = pattern.matcher(html);

```

```
10 List<String> links = new ArrayList<>();
11
12 while (matcher.find()) {
13     links.add(matcher.group(1));
14 }
```

Listing 3.1 – ChatGPT 2025-generated answer for the input query ID = 62: “*parse html with regex in Java*”

```
1 String html = "<html><body><h1>Hello, world!</h1><p>This is a
   paragraph.</p></body></html>";
2 Document doc = Jsoup.parse(html);
3 String text = doc.body().text();
4 System.out.println(text);
```

Listing 3.2 – ChatGPT 2023-generated answer for the input query ID = 62: “*parse html with regex in Java*”

3.5 Insights and Implications

This section discusses the insights and implications of the results obtained from the research questions for both researchers and practitioners.

3.5.1 Researchers

1. The combined use of SonarLint and LLMs creates opportunities for automated tasks such as performing refactoring operations, as linters could help identify and suggest syntactic improvements in code, while LLMs demonstrate the ability to understand contextual information. This is based on the premise that developers are increasingly using LLMs at scale, although they often require guidance through well-crafted prompts, an area in which linters can play a valuable role.
2. Although Scalabrino’s model indicated that more than 60% of the code snippets are readable and the observed cognitive complexity values have a relatively low median for a Java class (SonarLint considers cognitive complexity high when a method exceeds a value of 15), these metrics alone do not reveal the refactoring opportunities a developer needs to address readability issues in a class. Therefore, combining LLMs with linters may prove promising in the code review process.
3. Although the results mostly showed readable code with a manageable number of readability warnings in the LLMs, it is important to note that the context of this study is limited to code snippets, that is, relatively small classes. In real-world

projects, classes tend to be significantly larger and more complex, potentially leading to a higher number of readability warnings and reduced effectiveness of readability metrics.

3.5.2 Practitioners

1. Code snippets extracted from platforms such as question-and-answer sites may exhibit a considerably higher number of readability warnings, especially when authored by individuals with limited experience in Java who may be unfamiliar with some of its key coding conventions.
2. LLMs may require prompt engineering to effectively generate code that does not contain readability warnings.
3. ChatGPT in this research presented the best results compared to the other LLMs, as it generated significantly fewer readability warnings than DeepSeek, and substantially lower cognitive complexity and LOC compared to Gemini, under the same queries.

3.6 Threats to Validity

This section presents the main threats to validity, categorized into three types: construct validity, which concerns the alignment between theoretical concepts and their empirical measurement; internal validity, which relates to potential confounding factors within the study that could influence the results; and external validity, which addresses the extent to which the findings can be generalized beyond the specific context of this research.

3.6.1 Construct Validity

Correctness and Equivalence of Code Snippets: A potential threat concerns the correctness and equivalence of the generated code snippets. In this study, we did not formally evaluate whether each snippet was functionally correct; instead, we mitigated this limitation by selecting only those that were semantically adherent to the input query. Nevertheless, we acknowledge that comparing readability across snippets that are not fully equivalent in functionality can lead to confounding effects. A preferable approach for future work would be to report the proportion of correct outputs produced by each model and restrict readability comparisons to functionally equivalent snippets only.

SonarLint Readability Tool: This research relies on SonarLint as the primary tool for detecting code readability warnings. Although alternative ASAT such as PMD (PMD, 2025) and CheckStyle (CHECKSTYLE, 2025) were considered, SonarLint was chosen

due to its robust set of rules specifically focused on code readability and the didactic explanations provided in its official documentation. However, this dependency introduces potential threats. For instance, there is no guarantee that all possible readability warnings are covered by SonarLint, nor that the tool is entirely free from bugs that could lead to false positives or false negatives during the analysis. To mitigate these risks, the latest stable version of SonarLint available as of May 2025 was employed, and the environment was configured to ensure that at least one warning (e.g. missing package declaration) was raised per code snippet, confirming that all files were properly analyzed.

After manual inspection, no false positive warnings were found in the samples, as previously reported (LENARDUZZI et al., 2023). However, a few instances of false negatives were identified, that is, code that violates the SonarLint rules but was not detected by the linter. For example, Listing 3.3 and Listing 3.4 illustrate two code snippets of the query ID = 2: “How to remove an element from an array?”. ChatGPT 2023 (A) violates the rule *java:S3012* “Arrays and lists should not be copied using loops”, but DeepSeek (B) does not violate because it employs the *System.arraycopy* method. However, neither snippet raised any warnings related to SonarLint readability rules (including a manual inspection in the VSCode IDE).

```
1      // Create a new array with one less element
2      int[] newArray = new int[originalArray.length - 1];
3
4      // Copy all the elements of the original array into the new
      array
5      int newIndex = 0;
6      for (int i = 0; i < originalArray.length; i++) {
7          if (i != index) {
8              newArray[newIndex] = originalArray[i];
9              newIndex++;
10         }
11     }
```

Listing 3.3 – ChatGPT 2023-generated answer for the input query ID = 2: “How to remove an element from an array”

```
1      int[] result = new int[arr.length - 1];
2      System.arraycopy(arr, 0, result, 0, index);
3      System.arraycopy(arr, index + 1, result, index, arr.length -
      index - 1);
```

Listing 3.4 – DeepSeek-generated answer for the input query ID = 2: “How to remove an element from an array”

Query Dataset: CROKAGE has received numerous requests from developers worldwide, mainly due to the tool featured on the Stack Overflow blog (STACK OVERFLOW, 2025a), which has a broad reach within the developer community. However, despite the widespread distribution among developers, queries may not fully represent the most common questions users ask on the Web. The top 10 3-grams were referenced from queries with other related research (CAO et al., 2021), and some consistency was observed in the most popular tokens.

Stack Overflow recommended code snippets: In most of the queries used in this study, CROKAGE returned approximately 10 recommended code snippets per input query. The initial intention was to use only the recommended top-1 snippet to select for comparison with the LLMs generated code. However, it is not possible to guarantee that the developer who used CROKAGE actually rated that specific top-1 recommendation. To address this limitation, Stack Overflow snippets were also included in the semantic adherence analysis, with the highest-ranked snippet that was semantically aligned with the query being identified. Despite this, priority was given to the highest-ranked snippet, as it is the first one presented to the user and likely to exert a stronger influence on user ratings due to its immediate visibility.

3.6.2 Internal Validity

Code Snippet Retrieval: The retrieval of code snippets from Stack Overflow and LLMs, as well as their storage in separate Java files, was carried out manually by the authors. The evaluation of semantic adherence for the LLMs code snippets was carried out by two evaluators, as described in Section 3.3. The Stack Overflow code snippets were also independently checked by the authors to prevent any potential oversight or misinterpretation.

3.6.3 External Validity

Prompting LLMs: It is important to note that LLMs are actively evolving models. Other versions, such as the premium versions of each LLM may have had an impact on the results of this study. Moreover, future releases could yield different results. To ensure the reproducibility of the findings, some of the conversations with ChatGPT 2023 are publicly available in the replication package. In general, while this study could provide valuable results, it is important to consider the dynamic nature of LLMs, the influence of different questions, and the potential variations in the results based on the programming language and the characteristics of the data set.

Outdated CROKAGE data set: The CROKAGE dataset was built from a Stack Overflow dump extracted in 2019, meaning that all returned posts are limited to content published up to that year. This limitation can be partially mitigated by two factors. First,

most language features that improve code readability in Java were introduced before 2019. Second, the majority of Stack Overflow posts related to Java were concentrated between 2010 and 2015, suggesting that the dataset still captures a substantial portion of relevant community knowledge.

Programming Language: The scope of this study is limited to the Java programming language. This choice was primarily driven by the fact that CROKAGE is restricted to recommend only Stack Overflow posts with code snippets written in Java. Although Java is a widely adopted programming language, it is important to note that the findings of this study may not be directly applicable to other programming languages.

3.7 Concluding Remarks

In this study, LLM-generated code was compared to human-written Stack Overflow code recommended by the CROKAGE tool, focusing on identifying and analyzing potential code readability warnings using the SonarLint tool. The main findings of this investigation can be summarized as follows.

- ❑ LLMs generally demonstrate high effectiveness in generating didactic code snippets that semantically align with the input queries provided by users. However, the correctness of the generated code was not evaluated in this study. This aspect is considered highly relevant and should be addressed as a crucial direction for future research.
- ❑ In general, the code snippets generated by LLMs adhere to coding and naming guidelines and tend to produce more concise and streamlined code compared to the human-written code found on Stack Overflow.
- ❑ All LLMs produced significantly fewer readability warnings than the code snippets retrieved from Stack Overflow by CROKAGE. Remarkably, ChatGPT 4o produces significantly fewer warnings than DeepSeek.
- ❑ The code snippets generated by the LLMs and those recommended by CROKAGE are generally between 68% and 84% readable. Although Gemini tends to produce code with more lines and higher cognitive complexity, this did not result in unreadable code or a substantially higher number of readability-related warnings.
- ❑ Current LLMs may generate code that lacks readability improvements enabled by the newer Java API features, since their training data may not yet fully reflect these recent additions. As the availability of code examples using newer APIs increases on the Web, it is expected that future LLMs trained on such data will be more likely to generate up-to-date and potentially more readable snippets.

One potential opportunity for future research in this area is not only to analyze the code snippet recommendations generated by LLMs but also to compare the accompanying explanations with those provided by developers on platforms such as Stack Overflow. This expanded analysis would provide information on the quality and effectiveness of the explanations generated by LLM and how they compare to human-authored explanations. Furthermore, the methodology could be extended to encompass a wider range of programming languages, SonarLint warnings (bugs and vulnerabilities), and code generation tools such as GitHub Co-Pilot.

A Taxonomy of Code Readability Improvements Performed by Developers

The previous chapter assessed the readability of code snippets generated by LLMs or recommended by code search tools, based on the identification of readability warnings through linters. However, linters rely on predefined rules and may not fully capture how developers perceive code readability in practice. Therefore, this chapter aims to investigate how developers perform and perceive readability improvements in real-world projects, providing a human-centered perspective to the ASAT presented earlier. To address this objective, the readability improvements observed in merged GitHub PRs for Java and Python are grouped into categories, resulting in a taxonomy of developer-driven readability improvements.

This chapter is organized as follows: the motivation for building the taxonomy is first described, followed by the presentation of the research question and the methodology adopted to answer it. The results are then described and discussed in terms of their implications for researchers and practitioners, concluding with an examination of the study's threats to validity.

4.1 Context

Code reading is one of the most frequent activities for developers, as they spend a significant amount of time reading and understanding existing code (MINELLI; AND; LANZA, 2015; TIARKS, 2011). Given the importance of code readability, several studies have proposed methods to assess it. For example, some readability models have been developed to measure the effort required to read code in single snapshots (SCALABRINO et al., 2018; POSNETT; HINDLE; DEVANBU, 2011; BUSE; WEIMER, 2010), in code changes (ROY et al., 2020), or even predict the changes made by developers when improving readability in code (VITALE et al., 2023). However, assess code readability remains challenging due to the complex syntax and semantics involved (BUSJAHN et al., 2015;

SCALABRINO et al., 2017; PIANTADOSI, 2022). Moreover, most readability models do not capture real-world improvements made by developers (PANTIUCHINA; LANZA; BAVOTA, 2018; PIANTADOSI et al., 2020), resulting in a gap between automated assessments and actual readability improvements performed in practice (FAKHOURY et al., 2019).

Moreover, previous studies have focused mainly on source code written in Java. Based on the reviewed literature, there is a lack of research that assesses how developers perceive and improve code readability in programming languages such as Python, which, as of 2024, is the most popular language on GitHub, largely driven by growing interest in AI (GITHUB, 2024). Python also follows specific conventions and guidelines that help make code more understandable (OLIVEIRA et al., 2024). However, Python style guidelines also incorporate, for example, Pythonic idioms (ALEXANDRU et al., 2018), which can include functional constructs that have been reported to be potentially difficult to understand (ZID et al., 2024). Investigating how Python developers improve code readability in merged PRs can offer insights into the ways this aspect is perceived and addressed within the Python programming community.

Given the limitation of these experimental studies in identifying what makes code more or less readable or understandable in real-world projects, this chapter proposes characterizing how developers perceive and improve code readability in real-world Java and Python projects hosted on GitHub PR. This work decided to use PRs to identify readability improvements based on two main benefits:

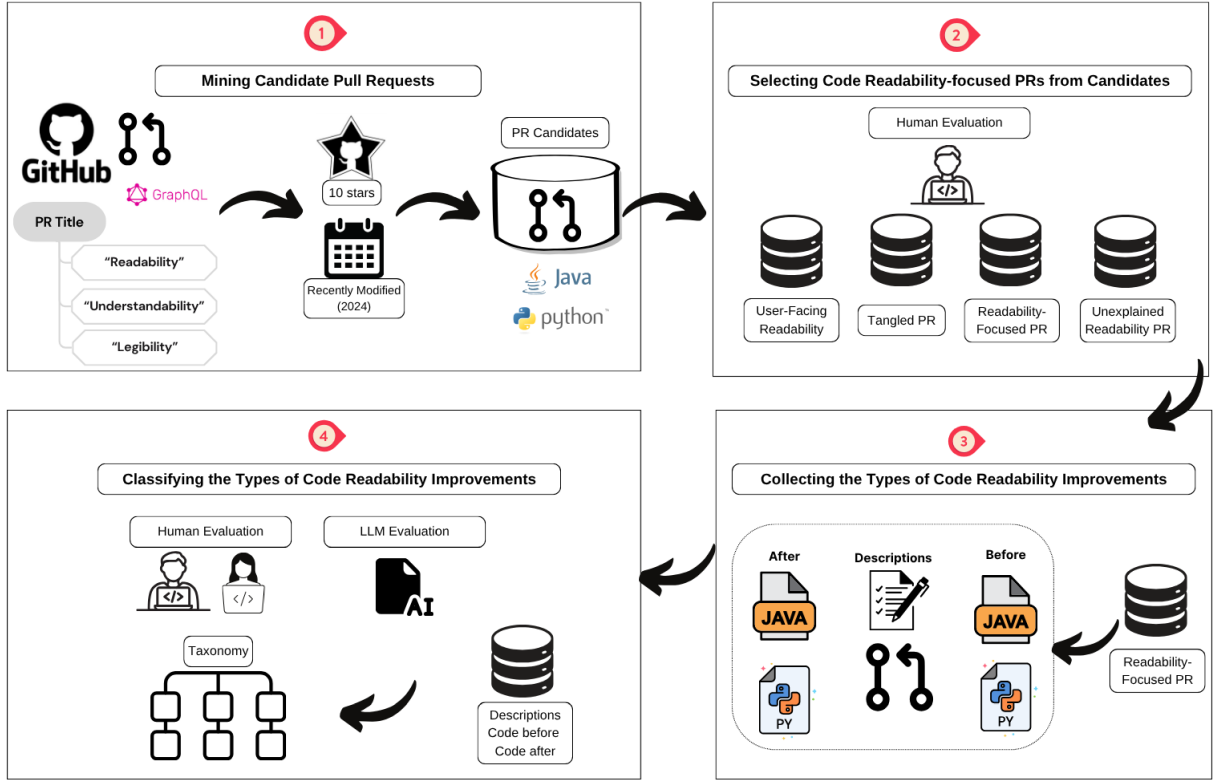
- ❑ PRs promote collaborative code review, where developers submit code changes and reviewers can suggest modifications before merging into the repository. This approach helps to ensure that commits with readability improvements are submitted to a peer code review process (SADOWSKI et al., 2018).
- ❑ Developers often provide detailed descriptions of their code changes in the PR to help reviewers better understand their work. These comments can offer valuable information about the developer's own perception of readability improvements (BACCHELLI; BIRD, 2013).

4.2 The Proposed Methodology

This section aims to explain the methodology to answer the following research question:

RQ₁: What types of code readability improvements do developers describe and implement in Python and Java Pull Requests (PRs)?

This research question aims to propose a taxonomy, identifying and categorizing the different types of code readability improvements made in Python and Java code. The

Figure 23 – Overview of the proposed methodology to answer RQ₁

objective is to understand the key readability factors that Python and Java developers address when they submit improvements to their PRs.

Figure 23 presents the overall approach adopted to answer the proposed research question. The following subsections describe the selection process of selected candidate PRs (Section 4.1), filtered readability-focused PRs (Section 4.2), collecting the instances (Section 4.3) and performing a manual evaluation (Section 4.4). A replication package, including data sets, scripts, classes, assessments, and reproduction instructions, is available (DANTAS; MAIA, 2025).

4.2.1 Mining Candidate Pull Requests

To identify merged PRs focused on improving code readability, a query was executed in January 2025 using the GitHub GraphQL API (GITHUB, 2025c) and the GitHub REST API (GITHUB, 2025b). The analysis focused on PRs that included one of the following keywords: “readability”, “understandability”, or “legibility” in their title field, since the title could indicate the motivation behind the PR. These keywords were selected based on their use in previous studies (ROY et al., 2020; FAKHOURY et al., 2019; DANTAS; ROCHA; MAIA, 2023a). Through this process, 1,293 merged PRs were identified in Python repositories (1,246 containing the keyword “readability”, 38 “legibility”, and 9 “understandability”), and 1042 merged PRs were identified from Java repositories (1009

“readability”, 23 “legibility”, and 10 “understandability”). Step 1 of Figure 23 illustrates this process.

Finding 1) Although three keywords, “readability”, “understandability”, and “legibility”, were used to search for candidate PRs, it was observed that in the vast majority of cases, developers mentioned the keyword “readability”. Although there is a slight semantic difference between these terms, it appears that developers have standardized the use of “readability” when referring to code that is easier to read and understand.

Instead of focusing on repositories from a specific ecosystem (e.g. Apache), which may follow particular readability practices, this study adopts a larger approach by selecting repositories from various ecosystems. To exclude potentially toy projects, PRs from GitHub repositories with fewer than 10 stars were removed. This filtering criterion has also been adopted in previous studies (TUFANO et al., 2024; OLIVEIRA et al., 2025). PRs were also excluded if they originated from forked repositories or from repositories that did not have an active code review process, defined as having no merged PRs in 2024. Furthermore, PRs were also excluded if they did not modify Python or Java files (e.g., those improving readability in textual files such as *README*, *.md* and *.rst* files) and excluded PRs that lacked at least two distinct participants (the PR author and a reviewer) to ensure a peer review process. Finally, PRs were removed if the PR body is written in the non-English language. After applying these criteria, 619 candidate PRs from Python and 577 candidates from Java repositories were identified.

To justify the decision to use only three keywords in the PR title for mining readability improvements, an initial experiment tested a more flexible search strategy. This approach included prefixes such as “readable” and “understandable”, applied to any field of PR, including title, body, commit messages, and reviewer comments. However, this broader search resulted in many false positives. For example, the keyword “readable” often appeared in unrelated contexts, and readability was sometimes mentioned in the comments of the reviewer without making any actual change to the code. A previous study had a similar issue. Although initially retrieved 2,693 Java PRs using this broader method, only 311 were kept after manual validation (DANTAS; ROCHA; MAIA, 2023a). To reduce false positives, the present work adopted a more focused strategy, restricting keyword searches to the PR title, which helped filter out irrelevant results and keep the most meaningful ones.

Previous work has used the data set provided by the *Reaper* tool, which analyzes GitHub repositories and filters out non-engineering or potentially trivial repositories (MUNAIAH et al., 2017). However, this data set was created in 2017 and does not include the large number of repositories created in subsequent years, many of which may contain PRs focused on improving code readability. Another available tool is *SEART*, which also

enables filtering of GitHub repositories (DABIC; AGHAJANI; BAVOTA, 2021), and has been used in more recent studies (OLIVEIRA et al., 2025). However, this thesis opted to retrieve data directly from GitHub, in order to avoid limitations similar to those reported with Reaper, particularly the exclusion of more recent repositories.

4.2.2 Selecting Readability-focused PRs from Candidates

In this subsection, the goal is to select PRs that were created exclusively to improve code readability. The primary motivation for this filtering is to reduce subjectivity, as a single Python or Java file can also include modifications not related to readability, potentially introducing tangled modifications into the analysis. When a PR is clearly focused on improving code readability, all modified files within that PR are considered samples, rather than trying to manually distinguish which changes are related or not related to readability.

Although the 619 candidate PRs on Python and 577 in Java mention readability-related keywords in their titles, some false positives were identified, that is, PRs that contain such keywords, but not actually focused on improving code readability. Therefore, the authors manually inspected the PR title and body fields of each of the candidate merged PRs into four categories as shown in step 2 of Figure 23.

1. User-Facing readability PR - Although readability keywords are mentioned, the PR title and PR body texts are not focused on improving code readability. For example, some PRs target improving readability for end users (e.g., improving console output, log messages, colors, styles, and others) rather than focusing on improving the code for other developers to have easier comprehension.
2. Tangled PR - In this case, the PR improves code readability but also includes other changes, such as adding new features, fixing bugs or other types of refactoring operations (e.g., improve performance) beyond code readability as described in previous work (PANTIUCHINA et al., 2020). These PRs are not only dedicated to enhancing code readability, as their focus is broader, rather than exclusively targeting specific classes for readability improvements.
3. Unexplained Readability PR - Some PRs mention readability improvements in vague terms, without clearly explaining what was actually performed. This study relies on what developers explicitly describe in the title, body fields, and commit messages of PR, rather than trying to infer intent from code changes, to minimize subjectivity in the methodology.
4. Readability-focused PR — These PRs are explicitly aimed at improving code readability, developers clearly describe such intentions in the PR title or body. All

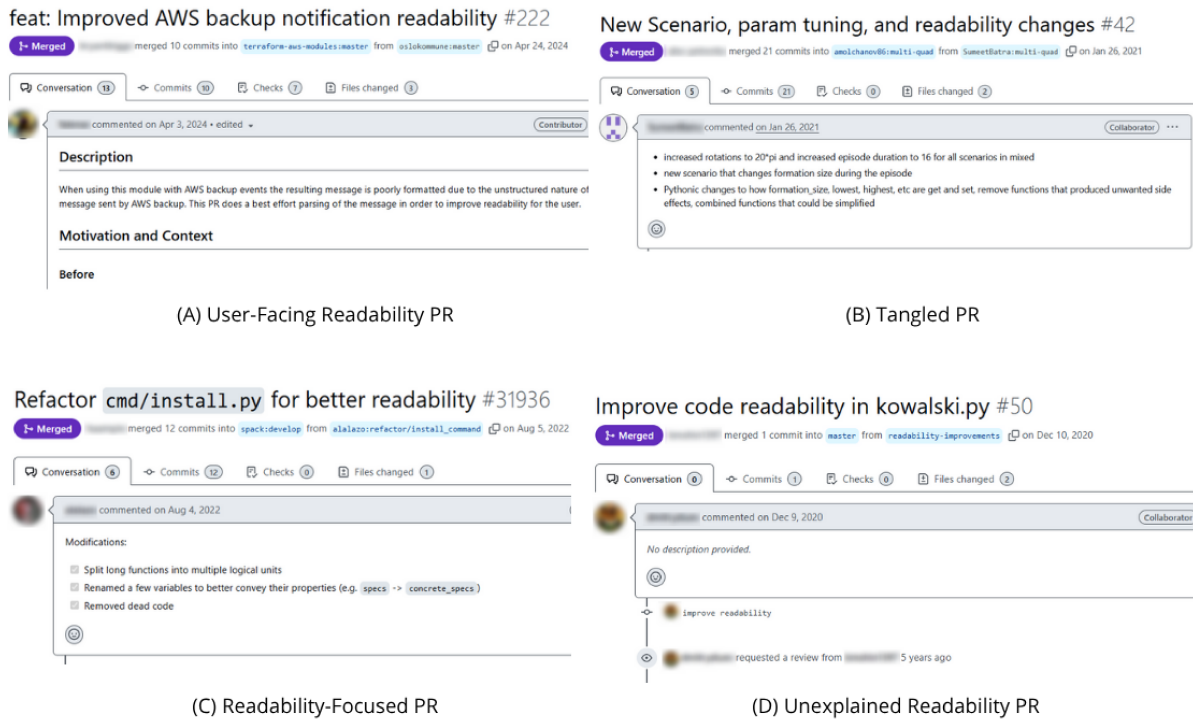


Figure 24 – Example of User-Facing Readability, Tangled, Readability-Focused and Unexplained Readability PRs extracted from GitHub

changes in these PRs are dedicated to improving the structure and clarity of the code, without introducing any other types of modification.

Figure 24 illustrates one example for each category. In the *User-Facing readability PR* (A), the developer’s motivation was to improve the readability of a notification message on the screen to the end user, which does not contribute to code readability for other developers. In *Tangled PR* (B), the developer improved code readability but also introduced unrelated changes, such as the inclusion of new features. The *Readability-Focused PR* (C) includes three code readability improvements: removing dead code, renaming variables, and splitting long functions. In the *Unexplained Code Readability PR* (D), the developer did not describe the specific type of code readability improvement made in any PR field (i.e., title, body, commit message, comment to the reviewer), although the code itself could suggest that such changes were made.

The classification is shown in Table 14. In Python, 264 merged PRs were identified, in which developers explicitly described code readability improvements in the PR. Furthermore, 161 *User-Facing Readability PRs*, 118 *Unexplained Code Readability PRs*, and 76 *Tangled PRs* were founded. It should be noted that although 382 of 619 (61%) candidates PRs mention code readability improvements, 31% of them (118 PRs) do not explain what kind of readability improvement was actually performed and will also be accepted by the reviewer and merged into the codebase.

In Java, 276 merged PRs were identified with explicit references to code readability

improvements in the PR. Additionally, 113 *User-Facing Readability* PRs, 79 *Unexplained Code Readability* PRs, and 109 *Tangled PRs* were found. Among the 355 out of 577 (61.52%) candidate PRs that mention readability improvements, 22% (79 PRs) do not clarify the specific type of improvement applied, despite being accepted and merged.

Finding 2) Among the merged PRs that improved code readability, 31% of the Python and 22% of the Java cases did not describe the readability changes in the title, body, or commit messages. Interestingly, these PRs were merged despite the lack of detailed explanation.

Table 14 – Classification of Candidate PRs by Programming Language

Category	Python		Java	
	PR Count	%	PR Count	%
Readability-Focused PR	264	42.39%	276	47.83%
User-Facing Readability PR	161	26.01%	113	19.58%
Unexplained Readability PR	118	19.42%	79	13.69%
Tangled PR	76	12.16%	109	18.89%
Total	619	100%	577	100%

4.2.3 Collecting the Types of Code Readability Improvements

In this step, for each of the 264 *Readability-focused PRs* in Python and 276 in Java identified in the previous step, the code readability improvements explicitly described by the developers in the PR were extracted into a field called *Developer Description*. Additionally, the corresponding pair of source code versions (before and after the commit) for the code snippet related to each improvement was collected. For example, in the PR illustrated in Figure 24 (C), the developer describes three types of readability improvements.

1. Split long functions into multiple logical units
2. Renamed a few variables to better convey their properties (e.g., `specs` → `concrete_specs`)
3. Removed dead code

For each type of improvement described by the developer, the corresponding code versions before and after the change were identified. Figure 25 presents the code diff between the modified classes of the PR shown in Figure 24 (C). In Example (A), the commit message “Extracted two functions in `cmd/install.py`” confirms that the functions were split to improve readability. Example (B) shows a variable renamed from

`specs` to `concrete_specs`, as described in the PR. Finally, in example (C), the developer justified the removal of dead code by explicitly indicating: “it doesn’t seem to be called anywhere”. This process is shown in step 3 of Figure 23.

Pull Requests (PRs). To access a specific PR, the following URL format can be used: `<https://github.com/[repository]/pull/[ID]>`. For example, the PR with ID = 5996 from the *apache/dubbo* repository is available at: `<https://github.com/apache/dubbo/pull/5996>`.

This approach verifies whether what the developers describe in title, description, and commit messages in the PR correspond to the changes made in the code. This verification is necessary because a PR may describe improvements that are not reflected in the code. For example, PR ID = 196 from *pyhoneybot/honeybot* repository described 17 types of code readability improvements. However, after verifying the code, only the renamed variable was implemented. The PR description, for example, mentions that the class `BotCore` was encapsulated, but that class was not modified at all. In such cases, the descriptions are discarded.

Following this approach, a total of 437 code readability improvements (instances) described by developers were identified in the 264 Python PRs and 456 in the 276 Java PRs. Each instance was created using the links: (i) a link to the GitHub merged PR, (ii) the developer description, and (iii) the code diff (before and after the commit).

To build the dataset of code readability improvements, only one instance of each type of change was considered per PR. For example, if a PR includes multiple instances of dead code removal, it is counted as a single instance. This approach prevents inflating the number of samples due to repetitive modifications made by the same developer within a single PR. Counting all occurrences could disproportionately assert certain types of change without capturing a broader consensus in different developers and projects.

Figures 26 and 27 illustrate a word cloud (with stop words removed) generated from the descriptions of Java and Python developers on how they improved code readability. As expected, several keywords commonly associated with readability improvements appear, such as “extract”, “remove”, “lambda”, “replace”, “reorder”, “add”, among others. However, each programming language has specific mentions of code readability improvements, such as:

1. Java: “javadoc”, “anonymous”, “assertion”
2. Python: “docstring”, “tuple”, “hint”

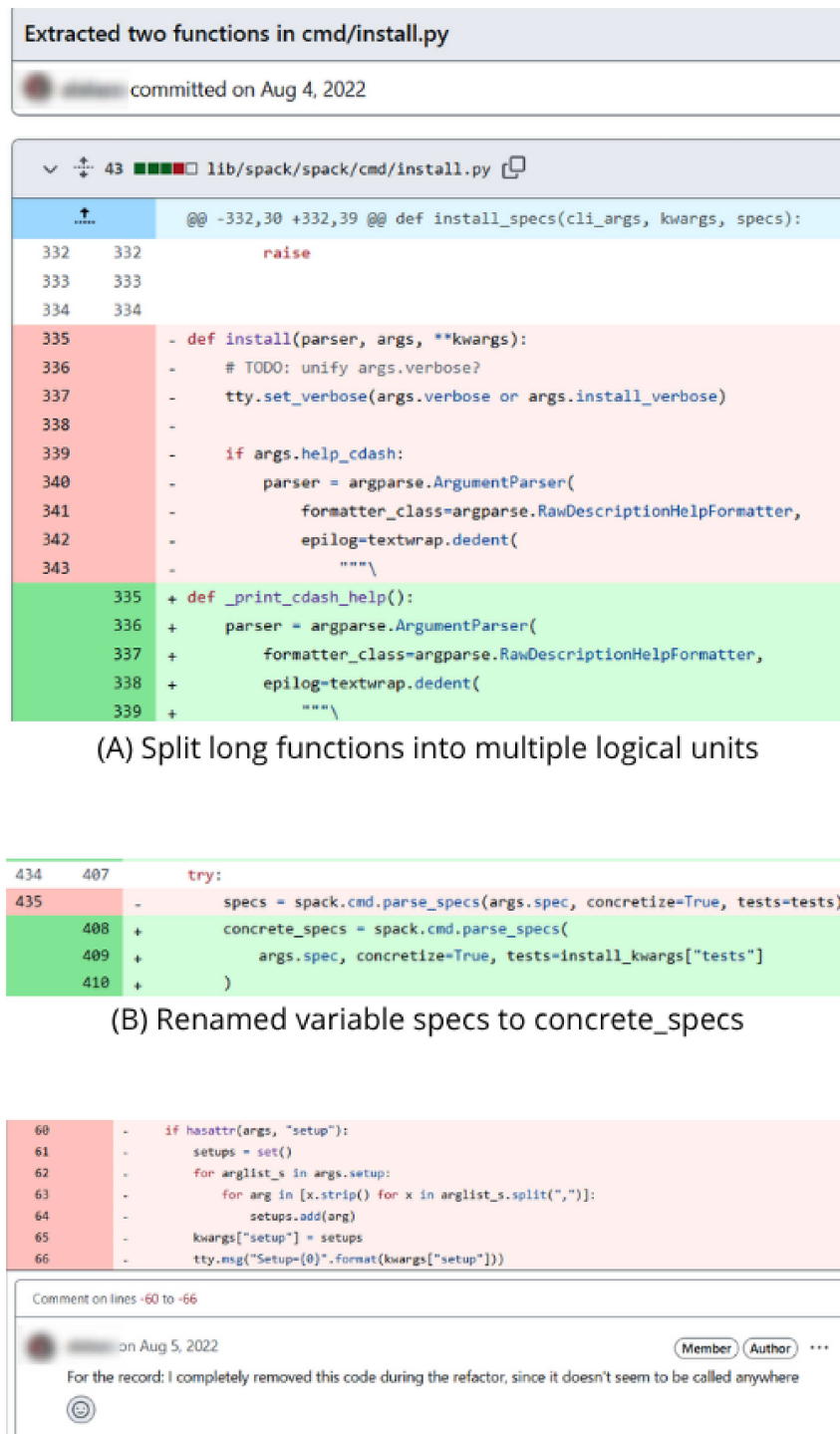
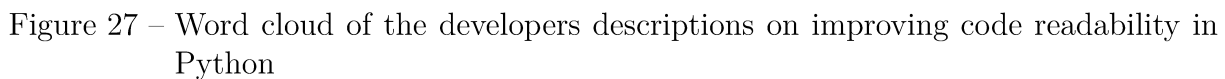
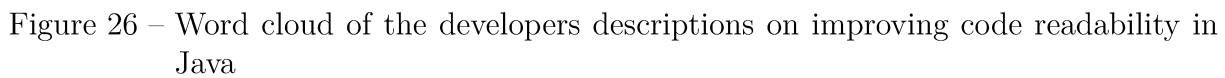


Figure 25 – PR ID = 31936 from *spack/spack*. Extraction of developers' description, code before and after the commit, producing three instances (types of changes)



4.2.4 Classifying the Types of Code Readability Improvements

To address RQ₁, two annotators (each with more than 15 years of experience in Java and 5 in Python programming languages) manually analyzed the 437 code readability improvements described in Python and the 456 Java instances, with the links and descriptions provided in the previous section.

To code the collected samples, the annotators independently analyzed the instances, assigning labels to each type of improvement implemented by the PR. As a starting point, the annotators used the labels defined in the SonarLint taxonomy of code readability warnings, as predefined in Chapter 3. These labels were shared in a common repository, and the annotators also shared their new labels in this repository while analyzing each instance. If an annotator found that an instance was not related to code readability improvements, it was labeled a false positive.

After tagging the instances, the two annotators discussed any conflicts in their tags and discarded the cases in which they could not reach an agreement. Furthermore, the use of two programming languages in the open coding process led to the relabeling of some instances, as the focus was on the semantic change applied in both languages rather than the specific Application Programming Interfaces (APIs) used. For example, in Java, the annotators identified instances where `for` loops were replaced by `streams`. The category was initially named *Replace Loops With Streams*. However, when analyzing Python instances, the annotators observed a similar behavior, although using Python-specific constructs such as functions from the `itertools` module. Rather than creating two separate types of change with the same underlying motivation, the annotators chose to define a single type of change that reflects what both Java and Python developers performed, highlighting each language-specific particularity. However, certain categories remained specific to each language, such as adding type hints in Python or converting lambda expressions to method references in Java.

During this process, 58 different labels were assigned to 437 Python instances and 61 to 456 Java instances, totaling 82 unique labels, 37 of which were shared between both programming languages, such as *improve naming*, *reduce conditional nesting* and others. In this process, we identified 21 exclusive labels for Python and 24 exclusive labels for Java.

4.2.4.1 GPT as a Code Readability Annotator

Before unifying the labels into broader categories, an additional step was introduced, involving a new annotator, specifically, the LLM GPT. This practice is discussed in recent research (AHMED et al., 2025) and is also discussed in the proposed guidelines for the use of LLMs in software engineering (BALTES et al., 2024). However, in this study, the role of the LLM is limited to supporting human annotators by revealing potential samples

that require reevaluation with label disagreements. The specific model used for labeling was GPT 4.1 using the *openai API*, configured with parameters *temperature* = 0.0, to obtain deterministic responses involving code refinement, as reported in previous work (GUO et al., 2024). The LLM was employed to assist in selecting the most appropriate label for each case, compared to the labels previously annotated by human annotators.

Initial prompt tests revealed that the labeling process is more effective when the LLM is provided with the predefined set of 82 labels produced. This helps guide the LLM to focus on the target classification and avoid generating labels with different semantics. For example, as shown in Figure 24 (C), the developer renamed a few variables. In such cases, the label *Improve Naming* was explicitly included as an option to the LLM. Without a predefined set, the LLM might produce a wide variety of suggestions that do not align with the intended taxonomy. Furthermore, the prompt was designed to allow LLM to propose new labels when appropriate, rather than restrict it exclusively to the list provided.

Another addition to the prompt was the inclusion of the code diff as a parameter, since the annotators observed that the developer’s description alone was often insufficient for the LLM to assign the correct label. For instance, in PR ID = 9265 from the *TheAlgorithms/Python* repository, the developer wrote: “Replaced repeated `len(board)` with a single variable `n` for efficiency in the `is_safe` function”. The human annotator read the description, examined the code diff, and correctly identified that a new variable was introduced to store the result of `len(board)`, classifying the change as *Extract Variable*. However, when only the description was provided, the LLM misinterpreted the change as the extraction of a `hardcoded` value, assigning the label *Extract Literals to Constants*. When both the description and the diff code were included in the prompt, the LLM assigned the same label as the human annotator in this case.

However, as observed, even in cases where the developer’s description alone was sufficient to identify the type of code readability improvement, GPT 4.1 did not have the same classification as humans. A representative example is PR ID = 67175 from the *elastic/elasticsearch* repository. The developer stated: “Improve readability of Node transform/forEach typed methods by moving the ‘type token’ to the front of the method, before the lambda declaration”. Human annotators labeled this as a *Reorder Elements* change. Surprisingly, GPT 4.1 classified it as an *Improve Naming* change. When the corresponding code was provided, GPT 4.1 aligned with the human classification.

The context was defined using a “system” prompt, as shown in the textbox below. In this prompt, [PROGRAMMING_LANGUAGE] refers to Java or Python. The field [LABELS] refers to the 61 for Java or 58 for Python types of code readability improvements previously identified by the human annotators. The field [DEVELOPER_DESCRIPTION] contains the developer’s description of a single readability improvement. [CODE_BEFORE] presents the code snippet before the change made in the PR, and [CODE_AFTER] shows the code after the change was applied.

Prompt Templates used for readability improvements

System: "You are a software developer specialized in [PROGRAMMING_LANGUAGE], skilled in reading GitHub Pull Request texts written by other developers and labeling the code readability improvements described by them using one of the following labels: [LABELS]. You may also suggest new labels if none of the provided labels accurately represents the improvement described by the developer."

User: "Based on this pull request with the developer description: [DEVELOPER_DESCRIPTION], the code was changed by the developer from [CODE_BEFORE] to [CODE_AFTER], please, provide the appropriate label that represents the code readability improvement described and performed by the developer. No explanations, only the name of the appropriate label"

After the first round, the GPT 4.1 generated labels were collected for each of the Java and Python instances provided as input. Then, these labels were compared with the label assigned by the human annotators. Table 15 summarizes the results. Labeling conflicts were found in the types of changes for 114 cases for Python (26% of instances) and 103 for Java (23% of instances). Then a manual investigation was conducted to verify the differences. Some of the divergences occur because of the fine-grained semantic differences between some labels, such as *remove redundant if-else* with *unwrap unnecessary else* or *Limit Line Length* with *Insert Line Break*, leading the annotators to reevaluate those labels. In other cases, the label was not sufficiently clear. For example, the inclusion of **comments** and **docstrings** was previously classified as *add comments*, and the modification of existing comments was classified as *clarify comments*. In the first round, GPT 4.1 classified some of the new comments as *clarify comments*, leading the annotators to rename the category to explicitly use the keyword *new*. In these cases, we revised the descriptions, the code diff, the predefined labels and provided a new input. After this process, 72 samples (17%) in Python and 49 samples (11%) in Java were then assigned the same label between LLMs and human annotators.

Table 15 – Classification of GPT 4.1 analysis as annotator

Category	Python		Java	
	Instances	%	Instances	%
Correct Match (Human × GPT 4.1)	323	74%	353	77%
Correct After Human Revised Label	42	10%	27	6%
Correct After Human Revised Prompt	30	7%	22	5%
Mismatch Classification	31	7%	46	10%
Discarded	11	2%	8	2%
Total	437	100%	456	100%

In 42 Python samples (9%) and 56 Java samples (12%), the divergence persisted, and instead of discarding each, the human annotators prefer to individually analyze each case

to decide whether the instance has to be discarded.

An illustrative example is PR ID = 5996 from the *apache/dubbo* repository. The developer’s description states: “replace some lambda expressions to method reference to improve code readability.” Listing 4.1 presents the code before and after the commit. However, in this case, the developers actually replaced an `anonymous class` (not a lambda expression) with a method reference. The annotators labeled this instance as *Replace Anonymous Class With Functional Style*. In contrast, GPT 4.1 labeled it as *Replace Lambda With Method Reference*, relying solely on the developer’s description rather than the code itself. Since the developer’s description was inaccurate, the annotators opted to discard this instance from the analysis.

```

1 //Code before the commit
2 ShutdownHookCallbacks.INSTANCE.addCallback(new
   ShutdownHookCallback() {
3     @Override
4     public void callback() throws Throwable {
5         DubboBootstrap.this.destroy();
6     }
7 });
8
9 //Code after the commit
10 ShutdownHookCallbacks.INSTANCE.addCallback(DubboBootstrap.this::
   destroy);

```

Listing 4.1 – Code before and after the commit of PR 5996 from *apache/dubbo* repository

In some cases, human annotators chose to retain the sample in the dataset, even when the LLM produced a divergent classification, interpreted as a mismatch classification. For example, in PR ID = 370 from the *SuffolkLITLab/docassemble-ALWeaver* repository, the annotators collected the developer description: “Limit interview_label length for readability, because 50 characters is still extraordinarily long and will render a lot of lines unreadable in the YAML, limiting to 35 characters instead.” The human annotators labeled this instance as *Limit Line Length*, whereas the LLM labeled it as *Clarify Numeric Constants*. In the second round (i.e., a second execution of the LLM with adjusted parameters), an additional code context was provided before and after the change. Even with this extra context, the LLM continued to classify the sample as *Clarify Numeric Constants*. Given the clear rationale provided by the developer, human annotators deemed the LLMs output a mismatch classification and kept the sample.

This process resulted in 44 Java and 38 Python labels observed in at least three different PRs, resulting in a total of 448 Java and 426 Python instances under 274 Java and 260 Python PRs. The annotators then grouped similar types of requests with similar purposes to create the initial taxonomy. This first draft included categories based on

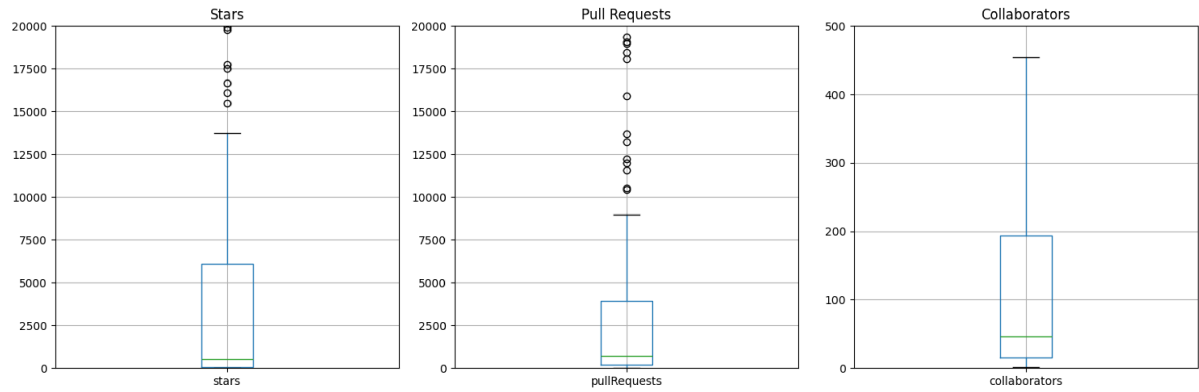


Figure 28 – Distribution of number of stars, pull requests and contributors on Python repositories

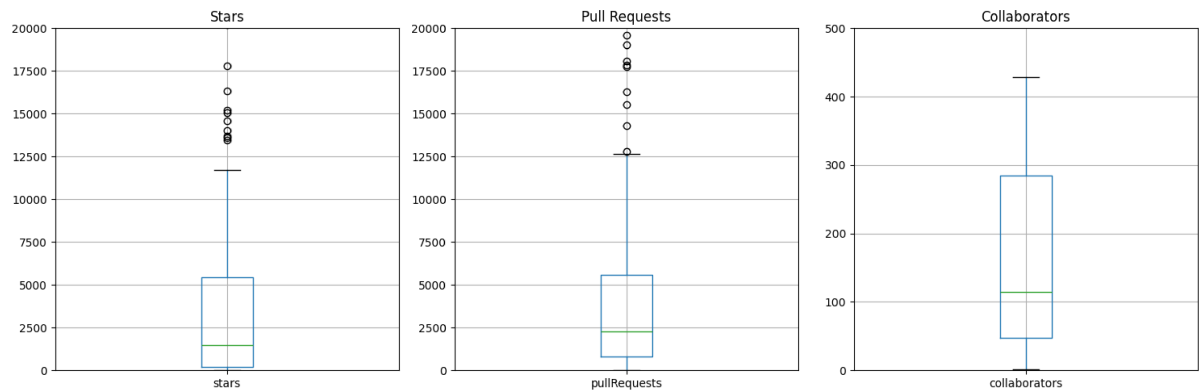


Figure 29 – Distribution of number of stars, pull requests and contributors on Java repositories

specific purposes, such as *improve if/else conditions* which were initially kept separate due to the high number of occurrences. However, this category was later merged into broader ones for improved semantic coherence. The final version of the taxonomy consists of six main categories.

Figures 28 and 29 show the distribution of stars, PRs, and contributors for the 149 Java repositories (with 274 merged PRs) and the 191 Python repositories (with 260 merged PRs). These PRs were opened by 214 Java and 226 Python developers (440 in total), and analyzed by a total of 1,036 (530 for Java and 506 for Python) distinct GitHub accounts, including both developers and reviewers.

4.3 Results and Findings

In this section, the results of the research questions are presented. A discussion is also included, based on the observations of the taxonomies produced. In the following, the numbers, PR examples, and particularities of Java (see  icon) and Python (see  icon)

will be discussed.

4.3.1 Taxonomy of Code Readability Improvements

The open coding process applied to the 426 Python and 448 Java instances identified 38 different types of changes for Python and 44 for Java, grouped into six categories: *Clarify Code Intent*, *Prefer Concise Code*, *Clean Up*, *Improve Formatting*, *Improve Code Modularity* and *Improve Documentation*. For each category and subcategory, the number of PRs that contain the change is observed. It is important to note that the number of PRs in each category does not necessarily correspond to the sum of its subcategories, as multiple subcategories may occur within the same PR. The types of code readability improvements (subcategories) observed in less than three PRs are classified as *others*.

4.3.2 Clarify Code Intent

This category appears in 125 of 274 PRs (46%) for Java and 125 of 260 PRs (48%) for Python. Specifically, it focuses on making the code more intuitive and self-explanatory, thus reducing the subjectivity involved in how other developers read and understand it. Unlike the *Prefer Concise Code* or *Clean Up* category, the improvements here do not necessarily involve removing tokens, but rather clarifying the intention of the code. Figure 30 illustrates five types of code readability improvements found exclusively in Java, six in Python, and nine that are common to both programming languages.

4.3.2.1 Improve Naming (📄 42 PRs) (♣️ 50 PRs)

This type of improvement has already been perceived in previous works (FAKHOURY et al., 2019; PANTIUCHINA et al., 2020), and shows how the quality of the identifier names is important to developers, with the aim of expressing their own intentions inside the project.

In Python, 31 PRs mention improvements in variable naming, whereas in Java only 10 such instances were found. However, 19 PRs in Java mention improvements in method names, compared to only 10 in Python. Examples:

- ❑ 📄 PR ID = 2381 from *apache/accumulo* repository: “Renamed variable names to enhance readability”.
- ❑ ♣️ PR ID = 12435 from *DataDog/integrations-core* repository: “improves code readability by renaming a variable that did not make sense”.
- ❑ 📄 PR ID = 4566 from *hyperledger/besu* repository: “Rename method `AbstractPendingTransactionsSorter::signalInvalidTransaction` to better express its semantic”.

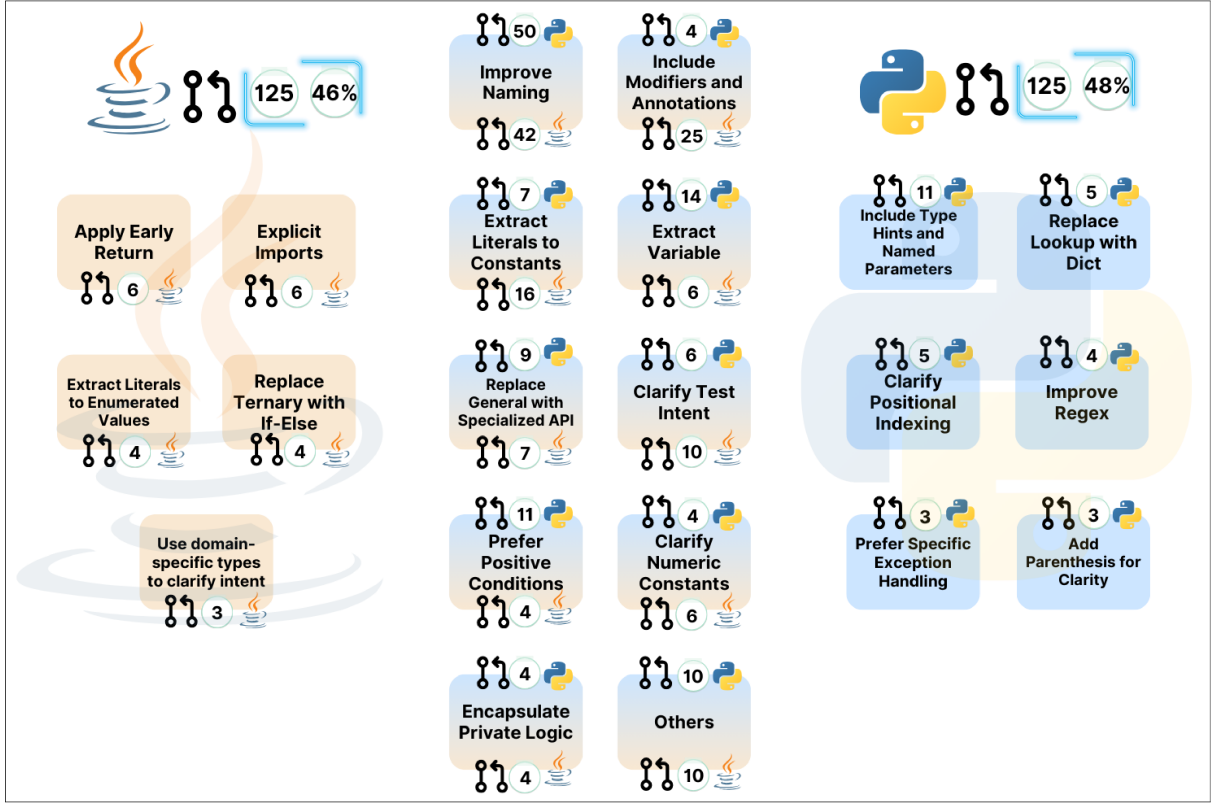


Figure 30 – Types of readability improvements in the Clarify Code Intent category

Moreover, in Java, five instances involved renaming class names, four involved renaming field names, two involved method arguments, and two involved renaming constants. In Python, five instances concerned renaming class names, two involved methods, and two involved constants.

4.3.2.2 Include Modifiers and Annotations (☞ 25 PRs) (☞ 4 PRs)

This type aims to expose the scope of the elements in a class. For example, in Java, when a developer includes the `final` modifier in a variable, other developers understand that the variable is immutable. The same idea is applied for `@NotNull` annotation on non-null fields or `@Override` annotation for overwriting methods from the superclass. This type of change noticed nine instances where the developer used the `@Deprecated` annotation, to mark deprecated variables and methods, four instances of `@Override`, three instances of `@NotNull`, four instances of `final`.

In Python, we only reported modifiers in the form of decorators, such as `@abstractmethod` and `@staticmethod`, which change the behavior or constraints of the methods.

4.3.2.3 Extract Literals to Constants (☞ 16 PRs) (☞ 7 PRs)

Occurs when the literal values whose meaning might not be obvious to the reader of the code (ANQUETIL et al., 2022). In Java, the PR ID = 254 from *Pi4J/pi4j* repos-

itory replaces a condition `if (accumulatedTime > 10 * 1000L)` with the constant `if (accumulatedTime > TEN_SECONDS)`. The developer explained his change: “I think the constant name could easier to read”.

In Python, PR ID = 74 from *JulianFrattini/cira* repository, the developer extracted the literal into a file named `constants.py`, and wrote: *This PR changes ‘graphconverter.py’ by using constants instead of hard-coded strings.*

4.3.2.4 Extract Variable (📦 6 PRs) (🔗 14 PRs)

This type of change was observed in cases where a single line contained significant complexity and the developer extracted parts of the expression into separate variables. For example, in Java PR ID = 2250 from the *apache/accumulo* repository, the line `var sjob = executors.get(job.getExecutor()).submit(myId, job, compactable, completion Callback);` contained three method calls `get()`, `getExecutor()` and `submit()` along the same line. The developer split this line into two, writing in the description: “Split up some single lines of code to make it easier to read”.

In Python, PR ID = 3837 from *equinor/ert* repository, the developer write: “We introduce a new variable representing the dict passed to rng config to increase clarity of the test in question.”, extracting a variable `rng_config_dict = ConfigKeys.RANDOM_SEED: random_seed`.

4.3.2.5 Replace Generic with Specialized API (📦 7 PRs) (🔗 9 PRs)

This subcategory describes *specialized APIs* to explicitly express the intent of the code. These samples have descriptions such as “provide more clarity”, “refactor to specialized functional interface”, and “improve code clarity using immutable interfaces instead of the generic collection interface”. In these scenarios, we observed a consistent pattern in which developers opted to replace a versatile API capable of serving multiple purposes with a specific API that effectively conveys intent.

For example, the PR ID = 4845 from *jenkinsci/jenkins* Java repository replaces a `Arrays.asList()` with `Collections.singletonList()` because the variable used only one element. In other words, employing `Collections.singletonList()` aids comprehension by explicitly indicating that the variable will contain only a single element, i.e., `Arrays.asList()` is the generic API, and the `Collections.singletonList()` is the specific API.

In Python, some examples include the use of a more specific mathematical function, such as `math.log10(len(options))` instead of `math.log(len(options), 10)`. This example was extrateded from PR ID = 130 from *openSUSE/opi* repository.

4.3.2.6 Clarify Test Intent (🔧 10 PRs) (🔧 6 PRs)

In this subcategory, most of the Java PRs improved from `assertEquals()` to `assertThat()` with the aim of adopting the fluent interface style (NAKAMARU et al., 2020). For example, in PR ID = 134 from the *jenkinsci/plugin-installation-manager-tool* repository, Listing 4.2 shows the comment left by the developer that illustrates this type of change.

```

1 //Improve the readability of tests. E.g.
2
3 assertThat(actualPlugins)
4     .containsExactlyInAnyOrder(
5         new Plugin("workflow-scm-step", "2.4", null, null),
6         new Plugin("workflow-step-api", "2.20", null, null));
7
8 //is easier to read than
9
10 List<Plugin> expectedPlugins = Arrays.asList(
11     new Plugin("workflow-scm-step", "2.4", null, null),
12     new Plugin("workflow-step-api", "2.20", null, null));
13 assertEquals(expectedPlugins, actualPlugins);
14
15 //and also more specific because it explicitly states that the
    order does not matter.

```

Listing 4.2 – An example of Simplify JUnit Asserts subcategory

In Python, the use of test parameterization was observed through the `@parameterized.expand` decorator. Listing 4.3 illustrates a parameterized test in PR ID = 4169 from the *streamlit/streamlit* repository. The use of `@parameterized.expand` between lines 1 to 8 creates a separate test case for each set of parameters, instead of grouping multiple scenarios within a single loop between lines 10 to 15.

```

1 @parameterized.expand(
2     [
3         ("code_location default", lambda: st.echo(), 0, 1),
4         ("code_location above", lambda: st.echo("above"), 0,
5             1),
6         ("code_location below", lambda: st.echo("below"), 1,
7             0),
8     ]
9 )
10 def test_echo(self, _, echo, echo_index, output_index):
11
12     def test_echo(self):

```

```

11     for echo, echo_index, output_index in [
12         (lambda: st.echo(), 0, 1),
13         (lambda: st.echo("above"), 0, 1),
14         (lambda: st.echo("below"), 1, 0),
15     ]:

```

Listing 4.3 – An example of test parametrization in Python

Although `@parameterized.expand` is also a decorator, the annotators chose to include it in the *Clarify Test Intent* category rather than *Include Modifiers and Annotations* because it typically requires multiple refactoring steps, such as changing the function signature and extracting the `for` loop logic into the decorator. Unlike other modifiers or annotations, its use is not simply an addition to infer behavior, but is part of a broader structural change that improves the clarity of the purpose of the test.

4.3.2.7 Prefer Positive Conditions (👉 4 PRs) (🔗 11 PRs)

In 15 cases, the developers mentioned that conditions involving negation are less readable. To improve readability, they refactored if statements to eliminate the use of the not operator (!) by inverting the conditions. For example, in PR ID = 13780 from the *prestodb/presto* Java repository, the developer originally wrote an `if-else` condition such as `if (!determinismEvaluator.isDeterministic(expression))`. This was later refactored by inverting the logic to remove the (!) operator, as explicitly described by the developer.

In Python, the developer on PR ID = 193 from *greenbone/python-gvm* repository called negation as a anti-pattern: “Address anti-pattern in conditionals: Use `if foo is not None` instead of `if not foo is None` and `if foo not in (set)` instead of `if not foo in (set)`”.

4.3.2.8 Include Type Hints and Named Parameters (🔗 11 PRs)

In Python, type hints indicate the expected type of function arguments or return values. The listing 4.4 presents an example from PR ID = 214 in the *google/capirca* repository. In this case, the method `RenderACL` initially had five parameters without explicit type information, making the expected types unclear. The updated code, shown on lines 1 to 8, introduces type hints to explicitly define the type of each parameter. Line 10 shows the function `RenderACL` without type hints.

```

1 def RenderACL(
2     acl_text: str,
3     acl_suffix: str,
4     output_directory: pathlib.Path,
5     input_file: pathlib.Path,
6     write_files: List[Tuple[pathlib.Path, str]],

```



```

7   binary: bool = False
8 ):
9
10 def RenderACL(acl_text, acl_suffix, output_directory, input_file,
    write_files, binary=False):

```

Listing 4.4 – An example of type hint in Python

Named parameters are arguments passed to a function using explicit names and values, which improves clarity in the order of the arguments. For example, in PR ID = 3754 from the *opsmill/infracore* repository, the developer replaced a function call written as (1299, "Arelion") with (asn=1299, organization="Arelion"), explicitly indicating the name associated with each argument.

4.3.2.9 Clarify Numeric Constants (📄 6 PRs) (♣ 4 PRs)

In 10 cases, the developers reported that numeric literals can be made more readable by using special characters. For example, using underscores in large numbers, such as writing 10_000 instead of 10000, was considered to improve readability.

4.3.2.10 Encapsulate Private Logic (📄 4 PRs) (♣ 4 PRs)

In this type of change, developers considered reducing the visibility of fields or methods to private as a way to improve readability. In PR ID = 35796 from the *Azure/azure-sdk-for-java* Java repository, the developer wrote: “Some minor changes to improve the readability of session receiver types implementation ... making few methods that don’t need to be package-internal as private”.

In Python, some methods are renamed to begin with an underscore (“_”), signaling that they are intended for internal use within the class. This convention was explicitly mentioned by the developer in PR ID = 68095 from the *getsentry/sentry* repository: “Class methods that are for internal use should begin with a _”.

4.3.2.11 Apply Early Return (📄 6 PRs) (♣ 1 PR)

This type of change is reported in Chapter 2 as one of the *Object Calisthenics* introduced by Jeff Bay (HOPKINS et al., 2008). Developers typically use early returns in `if-else` conditions to avoid deep-nested blocks. The description of developers in PRs includes texts such as “invert ifs and exit early” and “as a part of which early returns are introduced where suitable”.

4.3.2.12 Explicit Imports (📄 6 PRs)

This subcategory were founded only in Java PRs and refers to cases where developers avoid wildcard imports such as `import java.awt.*;` and instead explicitly specify the classes being used, such as `import java.awt.Color`, `import java.awt.Graphics2D`, and others.

4.3.2.13 Replace Lookup with Dict (📄 5 PRs)

In Python, improvements were observed where conditional logic based on raw literals was replaced by a dictionary mapping formats to their corresponding actions, as shown in Listing 4.5. In the lower part of the listing (lines 10 to 15), multiple `if-else` statements explicitly check the format and call the appropriate extraction function. In contrast, the upper part (lines 1 to 8) replaces this logic with a dictionary that maps each format string to its corresponding function. This example was extracted from PR ID = 530 of the *arc53/DocsGPT* repository.

```
1 extraction_functions = {
2     'py': extract_py,
3     'js': extract_js,
4     'java': extract_java
5 }
6
7 if formats in extraction_functions:
8     functions_dict, classes_dict = extraction_functions[formats](
9         dir)
10
11 if formats == 'py':
12     functions_dict, classes_dict = extract_py(dir)
13 elif formats == 'js':
14     functions_dict, classes_dict = extract_js(dir)
15 elif formats == 'java':
16     functions_dict, classes_dict = extract_java(dir)
```

Listing 4.5 – Replace Lookup with Dict in Python

4.3.2.14 Extract Literals to Enumerated Values (📄 4 PRs)

Initially, these changes were grouped under the category *Extract Literals to Constants*, previously referred to as *Replace Magic Literals*. However, the annotators observed two distinct behaviors. While *Extract Literals to Constants* involves moving individual literals into named constants, *Extract Literals to Enumerated Values* refers to creating an `enum` in Java that represents a set of related literal values.

For example, Listings 4.6 and 4.7 illustrate a refactoring performed by the developer in PR ID = 1506 from the *kolmafia/kolmafia* repository. The original code used a `switch` statement to handle constraint evaluations, where the value -1 indicated that the constraint was violated, 1 meant that the constraint was satisfied and 0 meant it was irrelevant. The developer introduced an `enum` to represent these states, making the `switch` statement easier to read and understand, as he described inside the PR: "Make `checkConstraints` return an `enum` describing what each number means".

```
1 switch (Maximizer.eval.checkConstraints(ModifierDatabase.  
    getEffectModifiers(effectId))) {  
2     case 1:  
3         continue;  
4     case 0:  
5         if (delta <= 0.0) continue;  
6         break;  
7     case -1:  
8         isSpecial = true;  
9 }
```

Listing 4.6 – Extract Literals to Enum before the refactoring

```
1 enum Constraint {  
2     /** Item violates a constraint, don't use it */  
3     VIOLATES,  
4     /** Item not relevant to any constraints */  
5     IRRELEVANT,  
6     /** Item meets a constraint, give it special handling */  
7     MEETS  
8 }  
9  
10 switch (Maximizer.eval.checkConstraints(ModifierDatabase.  
    getEffectModifiers(effectId))) {  
11     case MEETS:  
12         continue;  
13     case IRRELEVANT:  
14         if (delta <= 0.0) continue;  
15         break;  
16     case VIOLATES:  
17         isSpecial = true;  
18 }
```

Listing 4.7 – Extract Literals to Enum after the refactoring

4.3.2.15 Clarify Positional Indexing (🔗 5 PRs)

This subcategory clarifies the intent of indexing by explicitly using `-1` to access the last element in a sequence. For example, in PR ID = 130 from the *openSUSE/opi* repository, the code was changed from `release = option.split('-')[2]` to `release = option.split('-')[-1]`.

4.3.2.16 Improve Regex (🔗 4 PRs)

In Python, some developers simplified regular expressions, such as replacing `{0,}` with `\\d` to represent numbers, or removing nonessential characters.

4.3.2.17 Prefer Specific Exception Handling (🔒 1 PR) (🔗 3 PRs)

In some PRs, the developers changed the code to raise specific exceptions, indicating the intent. For example, in PR ID = 1554 from *thumbor/thumbor* Python repository, the code raised a general exception such as `raise Exception("x")`. The developer changed to `raise IOError("some-io-error")` and wrote: “Raising too general exception: Exception (broad-exception-raised)”

4.3.2.18 Replace Ternary with If-Else (🔒 4 PRs)

This subcategory shares a similar motivation with the *Extract Variable* subcategory, but applies to a different context. Some developers noted that ternary operators can be difficult to read, and therefore preferred to replace them with an equivalent `if-else` chain. Examples include.

- ❑ 🔒 PR ID = 2141 from *RPTools/maptool* repository: “overly complicated boolean expressions simplified”.
- ❑ 🔒 PR ID = 102 from *camel-tooling/camel-language-server* repository: “Write ternary with if-else to improve readability”.

4.3.2.19 Use domain-specific types to Clarify Intent (🔒 3 PRs)

This subcategory aims to enhance code clarity by specifying generic types explicitly, for instance, using `List<Pattern>` instead of the ambiguous `List<?>`.

4.3.2.20 Add Parenthesis for Clarity (🔗 3 PRs)

This subcategory aims include parenthesis, grouping elements. For example, using `(plot_func, d) = draw_cdf, db['data'][0]` instead of `plot_func, d = draw_cdf, db['data'][0]`.

4.3.2.21 Others (☞ 10 PRs) ☛ 10 PRs)

This work identified 10 instances in Java and 10 in Python of code readability improvements classified in the *Clarify Code Intent* category, driven by various types of change.

- ☐ ☞ Extracting fields into local variables inside methods - this change aligns with the SonarLint rule *java:S1450 - Private fields only used as local variables in methods should become local variables.*
- ☐ ☞ Introducing named overloads
- ☐ ☞ Creating customized exceptions
- ☐ ☞ Removing deprecated code
- ☐ ☞ Making generics explicit in type checks
- ☐ ☞ Replacing conditionals with ternary expressions
- ☐ ☛ Group Related Values into Named Structures
- ☐ ☛ Add Default Argument
- ☐ ☛ Extract Field to Local Variable

Interestingly, one case involved replacing an `if-else` statement with a ternary expression, which contradicts the typical pattern described in the subcategory *Replace Ternary with If-Else*.

Finding 3) Modifications that improve code clarity are present in almost half of the PRs aimed at improving code readability. While Python and Java developers share common practices, such as improving names, extracting literals into constants, and extracting variables, each language also presents its own particularities. For example, Python developers tend to focus on adding type hints and using named parameters and positive conditions, whereas Java developers often use modifiers and annotations to clarify code behavior.

4.3.3 Prefer Concise Code

This category appears in 96 of 274 Java PRs (35%) and 80 of 260 Python PRs (30%). Generally, developers describe verbosity as the presence of code that includes tokens that are not necessary to solve a task. Reducing code verbosity involves eliminating these superfluous elements, resulting in more concise and readable implementations. In general,

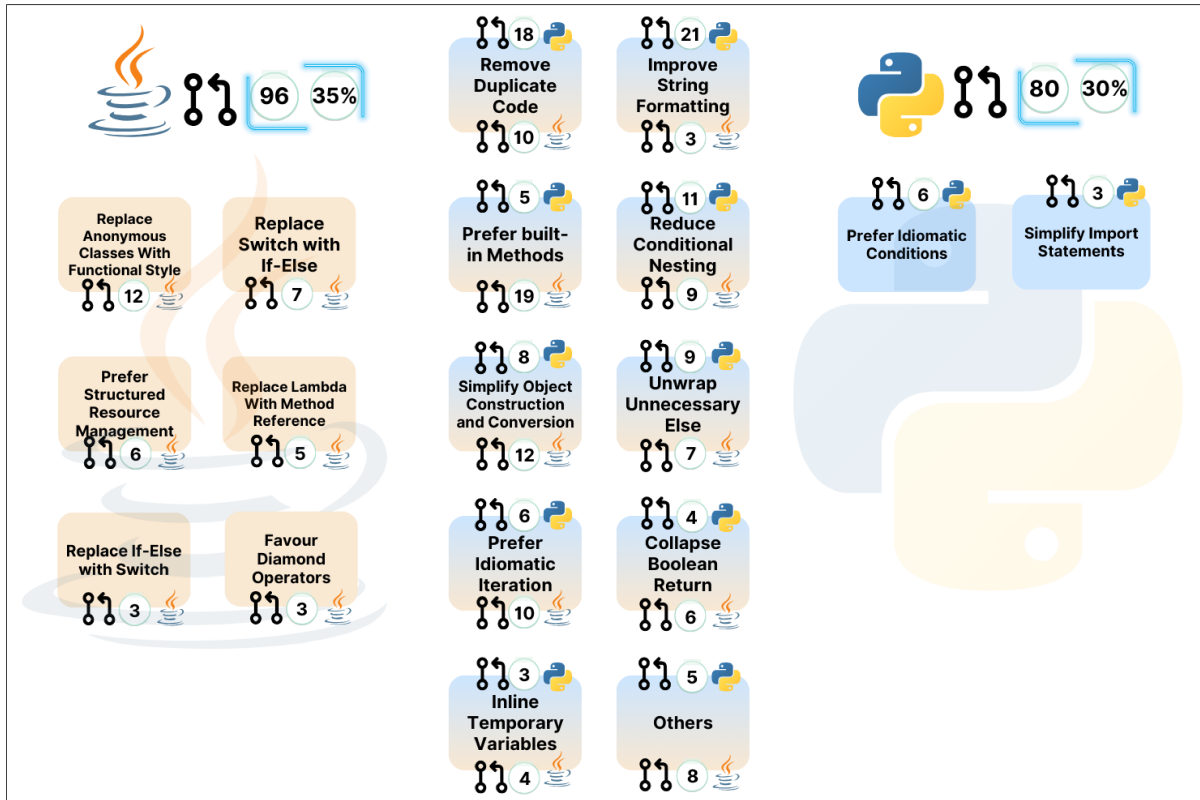


Figure 31 – Types of readability improvements in the Prefer Concise Code category

verbosity occurs when developers write additional code that is not essential for implementing a given feature. Figure 31 illustrates six types of code readability improvements found exclusively in Java, two in Python, and nine that are common to both programming languages.

4.3.3.1 Remove Duplicate Code (☕ 10 PRs) (🐍 18 PRs)

In this specific subcategory, the developer explicitly mentioned that extraction was motivated by removing code duplication, often using a refactoring operation such as the *extract method*. Examples:

- ☐ ☕ PR ID = 1271 from *apache/accumulo* repository: “Refactor `Compression.java` in order to remove duplicate code and restructure the class to make it easier to read overall.”
- ☐ 🐍 PR ID = 707 from *axolotl-ai-cloud/axolotl* repository: “As many tokenizer variations are added, its complexity is increasing and it is hard to understand what each code block means. We better clean up duplicated code and enhance code readability.”

4.3.3.2 Improve String Formatting (📄 3 PRs) (🔗 21 PRs)

This type of change occurs when developers concatenate string literals using operators such as “+”. In Python, such concatenations are mostly replaced with **f-strings** for improved readability. For example, the expression `intermediate = key + " : " + value` can be refactored to `intermediate = f"{key} : {value}"`, as seen in PR 3227 of the *prowler-cloud/prowler repository*. In Java, only three instances were found.

4.3.3.3 Prefer built-in Methods (📄 19 PRs) (🔗 5 PRs)

This occurs when the developer writes an algorithm to implement some feature that was already defined in the Java or Python API. The common API codes found in this work are the following:

1. 📄 `Collections.isEmpty()` instead of conditionals verifying if the size is equal to zero.
2. 📄 `str.isEmpty()`; instead of `"".equals(str)`;
3. 📄 `Math.min(binNumber,maxBinNumber)` instead of conditionals to verify the lowest number.
4. 📄 🔗 `String.join()` instead of loop appending text to String
5. 📄 `Collections.addAll()` instead of loop adding each element
6. 🔗 `endswith()` instead of string comparison by range

4.3.3.4 Reduce Conditional Nesting (📄 9 PRs) (🔗 11 PRs)

This subcategory refers to reducing the indentation level of control structures such as **if-else** statements or **for** loops. For example, in PR ID = 5256 from the *eXist-db/exist* Java repository, the developer explicitly mentioned reducing the indentation to improve the readability of the code, reducing **if** chain with depth = 2 to depth = 1. In Chapter 2, the *Object Calisthenics* principles introduced by Jeff Bay (HOPKINS et al., 2008) recommend using only one level of indentation per method. Although this subcategory does not necessarily defend a single level of indentation, it captures instances where developers reduce nesting to simplify the code structure.

Python PRs share similar motivations with Java PRs. In both programming languages, developers frequently mention the intention to “reduce nesting” or “reduce indentation”.

4.3.3.5 Simplify Object Construction and Conversion (👉 12 PRs) (👉 8 PRs)

This subcategory concentrates on removing boilerplate code, such as redundant object construction and conversion. Examples:

1. 👉 Removing redundant casts, such as unnecessary conversions from `java.lang.Object` to specific types or redundant calls such as `.longValue()`
2. 👉 Removing unnecessary calls such as `Collections.sort(profiles, Collections.<Profile>reverseOrder())`, converted to `profiles.sort(Collections.reverseOrder())`
3. 👉 Use of *Pattern Matching* resource, verifying the type and casting in the same line `if (named instanceof FieldAttribute fa)`
4. 👉 Simplifying the construct of dicts, such as from `visited = dict.fromkeys([source])` to `visited = {source: True}`

4.3.3.6 Unwrap Unnecessary Else (👉 7 PRs) (👉 9 PRs)

In this subcategory, the `else` block is considered redundant, as its condition can be extracted without affecting logic. This typically occurs when the `else` is the last condition in a method. For example, if an `else` block simply throws an exception, it can be removed and replaced with a standalone `throw` statement. Since it is the final condition, the code will execute the statement even if it is not enclosed in an `else` block.

4.3.3.7 Replace Anonymous Classes With Functional Style (👉 12 PRs)

The introduction of lambdas in Java is frequently reported by developers in PRs as a way to reduce boilerplate code and, consequently, improve code readability. It is common to find PRs referencing the “Java 8 style”, in recognition of the fact that lambdas were introduced in the Java 8 version. Anonymous classes are a typical example where the use of lambdas can eliminate large amounts of cluttered code.

Listing 4.8 and 4.9 illustrate the refactoring of anonymous classes into lambda, as described by the developer in the PR ID = 4703 from the *apache/pulsar* repository. The motivation described in PR: “The anonymous classes were converted to lambda at the places where it improves readability and decreases the lines of code”.

```

1      executor.submit(new Callable<Void>() {
2          @Override
3          public Void call() throws Exception {
4              barrier.await();
5              consumer.receive();

```



```

6         return null;
7     }
8 }));

```

Listing 4.8 – Anonymous class before refactoring into lambda

```

1     executor.submit((Callable<Void>) () -> {
2         barrier.await();
3         consumer.receive();
4         return null;
5     });

```

Listing 4.9 – Anonymous class after refactoring into lambda

4.3.3.8 Prefer Idiomatic Iteration (📄 10 PRs) (♣ 6 PRs)

Before analyzing Python instances, this category was initially named *Replace Statements with Streams*. However, since Python offers multiple ways to express functional-style iteration, such as list comprehensions, generator expressions, the `map/filter` functions, and the `itertools` module, the annotators opted to rename this subcategory as *Prefer Idiomatic Iteration*.

The listing 4.10 illustrates an example in PR ID = 6505 from *DIRACGrid/DIRAC* repository, where the imperative `for` loop in lines 1 to 5 was replaced by the use of operator `any` on line 7.

```

1 for possibleMsg in ["not exist", "not found", "No such file or
   directory"]:
2     if possibleMsg in errorMsg:
3         return True
4
5 return False
6
7 return any(possibleMsg in errorMsg for possibleMsg in ["not exist
   ", "not found", "No such file or directory"])

```

Listing 4.10 – Replacing Imperative Python loop with Functional Style

In Java PRs, this subcategory shares a similar motivation with the replacement of anonymous classes, but this focuses on replacing imperative control structures such as `for`, `while`, `do-while`, and `if-else` with stream-based constructs using lambdas. For example, in PR ID = 1155 from the *DataLinkDC/dinky* repository, a code snippet originally written with `LOC = 7` using `for` and `if` statements was refactored into a single line: `return Arrays.stream(str).allMatch(Asserts::isNullString)`.

4.3.3.9 Prefer Idiomatic Conditions (👉 2 PRs) (♣ 6 PRs)

This subcategory was initially named *Simplify Conditional Expressions*. However, during the GPT 4.1 evaluation, some overlaps with other subcategories related to `if-else` structures. Then, this category was renamed to *Prefer Idiomatic Conditions*. This involves simplifying Boolean expressions and avoiding unnecessary tokens or redundant constructs. Examples include.

1. 👉 `if (ok)` instead of `if (ok == true)`
2. ♣ `if not dictList:` instead of `if not len(dictList) > 0:`

4.3.3.10 Replace Switch with If-Else (👉 7 PRs) and Replace If-Else with Switch (👉 3 PRs)

These two subcategories may appear contradictory, but the analyzed instances revealed a relevant distinction between them. In cases where developers replaced a `switch` statement by `if-else`, the original `switch` typically contained only a few cases. In contrast, when developers refactored `if-else` chains into `switch` statements, the original code involved a long sequence of conditions. This observation aligns with the rule *java:S1301* from SonarLint, which declares: “switch statements should have at least 3 case clauses”.

Unlike in Java, no similar instances were found in Python.

4.3.3.11 Collapse Boolean Return (👉 6 PRs) (♣ 4 PRs)

In this subcategory, the developer converts an `if-else` chain into a single-line conditional expression to eliminate boilerplate code and make logic more concise. However, this is not the opposite of the subcategory *Replace Ternary with If-Else* previously discussed, as none of the four instances involves converting `if-else` statements into ternary expressions.

4.3.3.12 Prefer Structured Resource Management (👉 6 PRs) (♣ 1 PR)

This category was initially called *Use Try-with-Resources*. However, we found a single occurrence in Python that uses `with` to manage a resource through a context manager. To make the category name more general and applicable to both Java and Python, it was renamed to *Prefer Structured Resource Management*.

In Java, this subcategory also corresponds to a rule in SonarLint: *java:S209 — Try-with-resources should be used*. Although this rule is primarily associated with preventing resource leaks, it also reduces boilerplate code, which developers often associate with improved readability. For example, in PR ID = 286 from the *aaberg/sql2o* repository, the developer wrote: “Converted auto closable resources to use try with resources”. The

reviewer who accepted and merged the PR commented: “Looks good, this will increase the readability of our code base”.

Listings 4.11 and 4.12 show the code before and after the refactoring. In addition to managing resource closure more safely, the `try-with-resources` statement reduces the amount of code, making it much cleaner and easier to understand.

```
1      ResultSetIterable<T> iterable = null;
2      try {
3          iterable = executeAndFetchLazy(factory);
4          for (T item : iterable) {
5              list.add(item);
6          }
7      }
8      finally {
9          if (iterable != null) {
10             iterable.close();
11          }
12     }
```

Listing 4.11 – Try With Resources before refactoring

```
1      try (ResultSetIterable<T> iterable = executeAndFetchLazy(
2          factory)) {
3          for (T item : iterable) {
4              list.add(item);
5          }
6      }
```

Listing 4.12 – Try With Resources after refactoring

4.3.3.13 Inline Temporary Variables (4 PRs) (3 PRs)

This subcategory refers to cases where a developer introduces an unnecessary line by declaring a variable that could be inlined into the following statement. In some Java instances, it aligns with the SonarLint rule *java:S1488 - Local variables should not be declared and then immediately returned or thrown.* For example, the developer creates a variable only to return its value within a method.

However, this subcategory is broader. It also includes cases where a developer unnecessarily initializes a variable before assigning its actual value. For example, write `String serviceURL = null;` followed immediately by an assignment to that variable. In such situations, the initial declaration is redundant and can be removed to simplify the code.

4.3.3.14 Replace Lambda With Method Reference (☞ 5 PRs)

Although lambdas in Java can reduce boilerplate code, as shown in previous subcategories, in some cases developers consider method references to be more readable. For example, in PR ID = 488 from the *apache/uniffle* repository, the developer refactored `tid -> markFailedTask(tid)` to `this::markFailedTask`, and described the change as: “Improve readability by replacing lambda with method reference”.

4.3.3.15 Simplify Import Statements (☞ 2 PRs) (♣ 3 PRs)

In some cases, developers shorten imports to remove unnecessary qualifiers and improve code readability. For example, in Python, by writing `from hashlib import sha256`, they can call `sha256()` directly instead of using the fully qualified name `hashlib.sha256()`. In Java, using static imports to eliminate repetitive code, replacing expressions such as `Assert.assertTrue()` with the more concise `assertTrue()`.

4.3.3.16 Favour Diamond Operators (☞ 3 PRs)

This subcategory occurs only in Java and is similar to the SonarLint rule *java:S2293 - The diamond operator ("<>") should be used*. The developers describe as a redundant type definition, and the code should be refactored from `List<String> r = new ArrayList<String>()` to `List<String> r = new ArrayList<>()`.

4.3.3.17 Others (☞ 8 PRs) (♣ 5 PRs)

This work identified eight instances of Java and five for Python code readability improvements classified under the category *Prefer Concise Code*, driven by various types of change.

- ☐ ☞ Refactor to Lombok (LOMBOK, 2025)
- ☐ ☞ Inline Lambda Block
- ☐ ☞ Replace Null-check with Annotation
- ☐ ♣ Replace while with for
- ☐ ♣ Use Enumerate

Finding 4) Java and Python developers share similar concerns about writing concise `if-else` structures. However, the annotators observed a notable number of Java PRs focused on reducing code and boilerplate lines through Java 8 features, such as lambdas and streams. In contrast, Python developers showed a stronger emphasis

on writing and concatenating strings in a cleaner way, avoiding unnecessary tokens. The annotators found more PRs and a wider range of categories in Java compared to Python.

4.3.4 Clean Up

This category appears in 43 out of 274 Java PRs (16%) and 23 out of 260 Python PRs (9%). While *Prefer Concise Code* removes unnecessary additional code to implement a feature, this category removes unused code, that is, a custom method never called or a field never used. Figure 32 illustrates five types of code readability improvements found exclusively in Java, and two that are common to both programming languages.

4.3.4.1 Remove Unused Method/chunk of code (🗑️ 10 PRs) (♣️ 15 PRs)

This subcategory refers to the removal of unused code, such as unused methods (e.g., `main` methods) or unused code blocks. Examples:

- ❑ 🗑️ PR ID = 35796 from *Azure/azure-sdk-for-java* repository: “removing unnecessary method calls indirections and resulting unused methods”.
- ❑ ♣️ PR ID = 68095 from *getsentry/sentry* repository: “Functions that are not used anywhere should be removed, even if they were used previously”.

4.3.4.2 Remove Commented Code (🗑️ 14 PRs) ♣️ 1 PR)

Developers generally consider the removal of commented-out code as a readability improvement, since as such comments often contain legacy code that, rather than clarifying the current implementation, may increase the LOC and make the actual code harder to read. SonarLint also includes a rule addressing this issue in Java: *java:S125 — Sections of code should not be commented out*.

Interestingly, most of the Java PRs involving the removal of commented code comes from the same repository, namely *idempiere/idempiere*, suggesting that this practice may be concentrated among a few developers in the sample. In Python, a single instance was categorized as others.

4.3.4.3 Remove Unused Imports/Unnecessary Throws (🗑️ 11 PRs) (♣️ 3 PRs)

The *Remove Unused Imports* subcategory includes instances where the developer removes import statements that are not used in the class. Similarly, the *Remove Unused Throws* subcategory refers to cases where the developer removes unnecessary exception

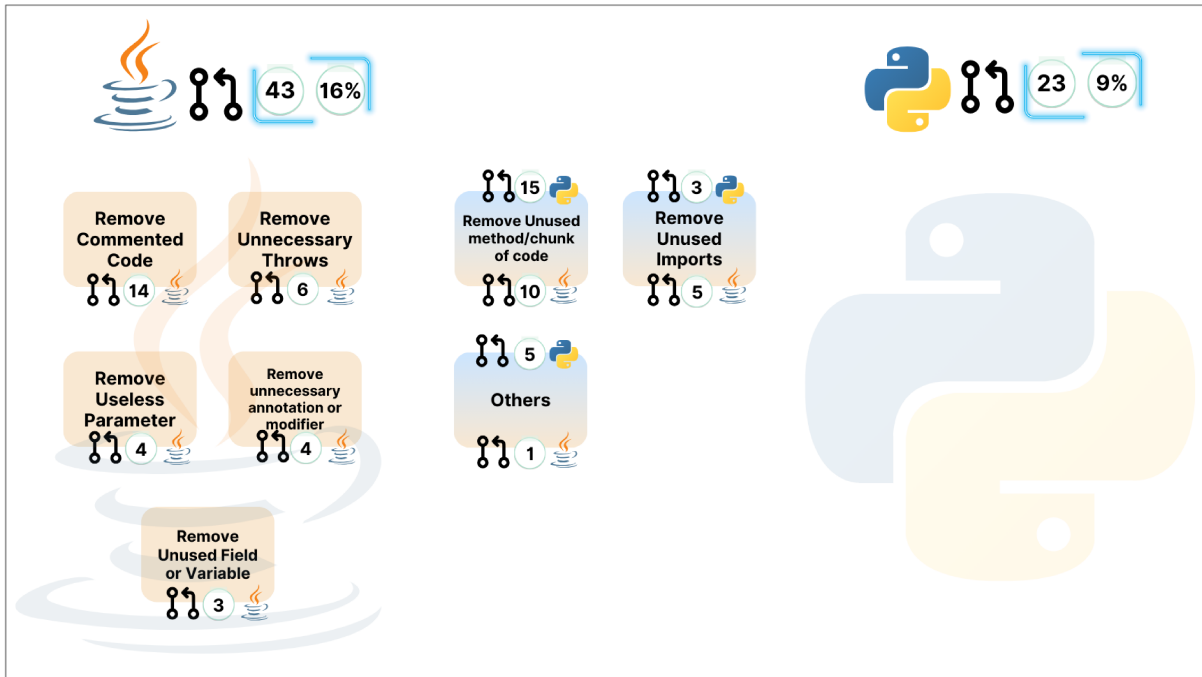


Figure 32 – Types of readability improvements in the Clean Up category

declarations, such as methods declaring a `throws` clause for exceptions that are never actually thrown. SonarLint includes rules for Java that issue warnings for unused imports, such as *java:S1128 — Unnecessary imports should be removed*. In Python, the two PRs are related to removing unused imports.

4.3.4.4 Others (☕ 12 PRs) (🐍 5 PRs)

Some of the others include:

1. ☕ 🐍 Variables, parameters or annotations not been used
2. 🐍 Constants not been used
3. ☕ Unnecessary `try-catch` structures

Finding 5) Although we found more instances in Java PRs related to cleanup by removing unnecessary code, both Python and Java developers share similar concerns about eliminating duplicated code and removing unnecessary chunks, such as entire methods. Notably, Java PRs included a significant amount of commented code, a pattern not observed in Python.

4.3.5 Improve Code Modularity

This category appears in 51 out of 274 Java PRs (19%) and 38 out of 260 Python PRs (14%). Modularity refers to the separation of the complexity of a program into smaller and more manageable chunks of code (BECK; DIEHL, 2011). It differs from other categories in that it does not involve polishing a code snippet as seen in categories such as *Prefer Concise Code*, or removing unused code as described in the *Clean Up* category. Although improving modularity may also contribute to clarifying the intention of the code, it also has a motivation to organize the codebase into coherent classes and methods. Figure 33 illustrates one type of code readability improvement found exclusively in Java, two in Python, and two that are common to both programming languages.

4.3.5.1 Extract Method from Large Function (📄 32 PRs) (🔗 19 PRs)

In this case, the developer considers the method to be violating the *Long Method* code smell and typically extracts parts of its logic into smaller and more focused methods. This differs from the *Remove Duplicate Code* subcategory previously discussed, as the code is not eliminated due to redundancy but rather reorganized into a separate method to improve readability and structure. In other words, the code is moved, not removed. Examples:

- ❑ 📄 PR ID = 4893 from *geoserver/geoserver* repository: “Break up the 272 lines long `CatalogItemProcessor.process()` method into smaller chunks for each `CatalogInfo` type needing specialized processing”.
- ❑ 🔗 PR ID = 31936 from *spack/spack* repository: “Split long functions into multiple logical units”.
- ❑ 📄 PR ID = 378 from *jenkinsci/remoting* repository: “The first PR extracts a few methods for improved readability. This makes it easier to isolate the different operations and see what is going on”.
- ❑ 🔗 PR ID = 3679 from *pytest-dev/pytest* repository: “Extract the `parametrize()` block of code into methods for better readability”.
- ❑ 📄 PR ID = 71876 from *elastic/elasticsearch* repository: “The hope here is that by drying up across both where possible and splitting the logic to different methods for both otherwise, the corner become easier to understand”.

4.3.5.2 Move logic to another file (📄 15 PRs) (🔗 10 PRs)

This category was previously labeled *Extract Class* because of the Java instances, but as the number of instances increased, it became evident that some examples involve

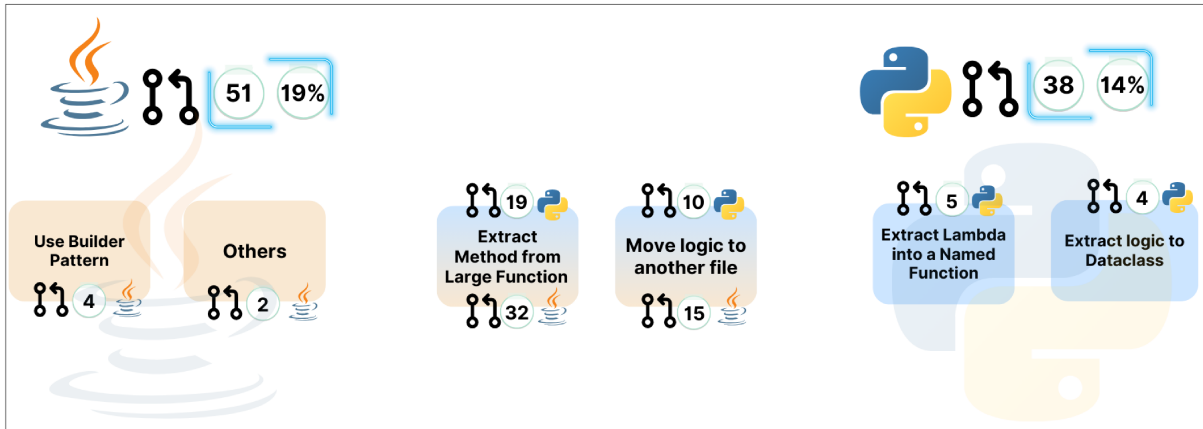


Figure 33 – Types of readability improvements in the Improve Code Modularity

extracting code from an existing class rather than creating an entirely new one. However, most cases still involve the creation of a new class, with a significant amount of code being extracted from another class.

- ❑ ☕ PR ID = 2256 from *CorfuDB/CorfuDB* repository: “move all index interfaces and definitions out of *CorfuTable* and into a separate class..”
- ❑ ☁ PR ID = 203 from *warpnet/salt-lint* repository: “Splitting the ‘*linter.py*’ and formatters in separate files”
- ❑ ☕ PR ID = 25 from *jenkinsci/office-365-connector-plugin* repository: “Moves two method into separate class to enable unit test (next PR). The code is same except one method that has been split into smaller one to improve readability”.
- ❑ ☁ PR ID = 264 from *CsqLang/Csq* repository: “Split Large Code Files for Improved Manageability and Readability Optimized codebase for manageability and readability by splitting large files into smaller”.

4.3.5.3 Extract Lambda into a Named Function (☁ 5 PRs)

In five Python PRs, developers replaced lambdas with local functions, motivated by the fact that lambda expressions can become unnecessarily complex and difficult to understand. As noted by a developer in PR ID = 18539 from the repository *getsentry/sentry*: “I know that a lot of similar classes use the `on_results=lambda x: serialize(x, ...)` pattern (I’m fine with lambdas generally) but this is one of the only multiline ones with a big compound expression”.

The listing 4.13 illustrates the refactoring in PR ID = 506 from *lichess-bot-devs/lichess-bot* repository. Line 1 shows the expressions using lambda before the refactoring, and lines 3 and 4 show the expressions extracted from lambda to a named function called `get_greeting()`.


```

1 get_greeting = lambda greeting: str(greeting_cfg.get(greeting) or
   " ").format_map(keyword_map)
2
3 def get_greeting(greeting):
4     return str(greeting_cfg.get(greeting) or " ").format_map(
       keyword_map)

```

Listing 4.13 – Example of Replacing Lambda with Local Function

4.3.5.4 Extract logic to Dataclass (♣ 4 PRs)

In some Python PRs, developers initially create anonymous dictionaries to store structured data, as shown in lines 1 to 4 of Listing 4.14. However, this approach may reduce readability, especially when the data structure is used in multiple functions. To improve clarity, some developers refactor the code by encapsulating the structure into a `@dataclass`, as shown in lines 6 to 17 of the same listing.

```

1 self._params.append({"name": field.name,
2                      "type": extract_string_type(field),
3                      "is_primitive": is_primitive_type(field)
4                      })
5
6 @dataclass
7 class Param:
8     name: str
9     type: str
10    is_primitive: bool
11
12 self._params.append(
13     Param(
14         name=field.name,
15         type=extract_string_type(field),
16         is_primitive=is_primitive_type(field))
17 )

```

Listing 4.14 – Example of Replacing Lambda with Local Function

4.3.5.5 Use Builder Pattern (🏗️ 4 PRs)

In four Java instances, developers reported the use of the *Builder* design pattern (GAMMA et al., 1994) as a means to improve code readability. For example, in PR ID = 528 from the *HubSpot/jinjava* repository, the developer wrote: “The JinjavaConfig

constructor had too many parameters. This replaces that constructor with one that just takes in a Builder. It is both easier to read and does not violate *Checkstyle* (which throws a style violation after 20+ parameters)”.

The listings 4.15 and 4.16 illustrate the constructor of a class before and after applying the *Builder* design pattern.

```
1  public JinjavaConfig(  
2      Charset charset,  
3      Locale locale,  
4      ZoneId timeZone,  
5      int maxRenderDepth) {  
6      this(  
7          charset,  
8          locale,  
9          timeZone,  
10         maxRenderDepth,  
11         new HashMap<>(),  
12         false,  
13         false,  
14         false,  
15         0,  
16         false,  
17         0,  
18         true,  
19         RandomNumberGeneratorStrategy.THREAD_LOCAL,  
20         false,  
21         0,  
22         new JinjavaInterpreterFactory(),  
23         new DefaultTokenScannerSymbols(),  
24         JinjavaInterpreterResolver.DEFAULT_RESOLVER_READ_ONLY,  
25         false,  
26         false);  
27 }
```

Listing 4.15 – Instantiation of an object before applying the Builder design pattern

```
1  public JinjavaConfig(  
2      Charset charset,  
3      Locale locale,  
4      ZoneId timeZone,  
5      int maxRenderDepth) {  
6      this(  
7          newBuilder()  
8      );  
9  }
```

```

8         .withCharset(charset)
9         .withLocale(locale)
10        .withTimeZone(timeZone)
11        .withMaxRenderDepth(maxRenderDepth)
12    );
13 }

```

Listing 4.16 – Instantiation of an object using the Builder design pattern

4.3.5.6 Others (📦 2 PRs)

This work identified two more instances of code readability improvements classified under the *Improve Code Modularity* category.

- ❑ 📦 Extract Class to Separate Package.

Finding 6) As expected, Python and Java programmers share concerns about extracting long methods and improving code modularization across classes. However, this category also reveals interesting language-specific patterns, such as improving the instantiation of objects with many fields in Java, and the use of dataclasses and the extraction of lambdas in Python.

4.3.6 Improve Documentation

This category appears in 38 out of 274 Java PRs (14%) and 65 out of 260 Python PRs (25%). The aim is to improve code readability using comments (STAPLETON et al., 2020) and documentation such as Python *docstring*. In Java, in 25 instances, the keyword “Javadoc” is mentioned, while in Python, the keyword “docstring” is mentioned in 49 instances. Figure 34 illustrates two types of code readability improvements that are common to both programming languages.

🔗 28 PRs inclusions of comments and *docstring*, 42 PRs clarifications of comments and *docstring*, some of them asking to fix typos (typographical errors) and a single instance asking to remove an unnecessary comment, arguing that the code is sufficiently intuitive.

📖 29 PRs inclusions of comments and *Javadoc*, 22 PRs focus on improving the quality of the existing comment text.

Examples include.

- ❑ 📖 PR ID = 11886 from *grpc/grpc-java* repository: “Add comments between logical blocks - Add comments explaining some xDS/gRPC nuances”.

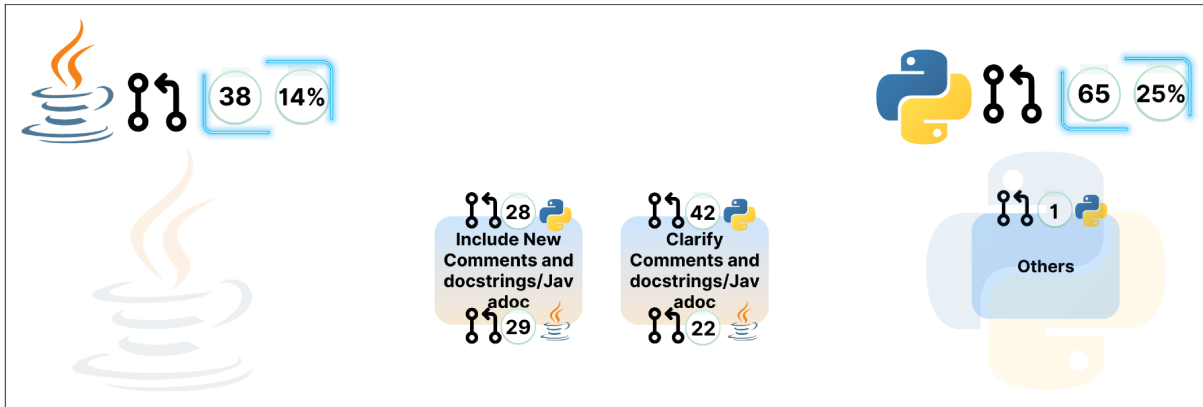


Figure 34 – Types of readability improvements in the Improve Documentation category

- ❑ ☕ PR ID = 20353 from *gradle* repository: “Rewrites comment to read better. Readability is increased which will allow for faster comprehension of the edited file”.
- ❑ 🍄 PR ID = 35 from *adswerve/GA4-Measurement-Protocol-Python* repository: “fixing some comment typos.”.

Interestingly, Python has a higher number of affected PRs, 65 compared to 38 in Java. This occurs because, while many Java PRs focus on creating and improving comments within the same PR, it is common for Python PRs to create or improve comments or documentation, but not both.

Another relevant aspect is that Python coding conventions emphasize method documentation. As a result, changes such as variable renaming often lead to updates in *docstrings*, which may also explain the higher number of readability improvements related to documentation.

Finding 7) Python developers create or clarify comments and docstrings in 25% of all PRs focused on code readability improvements, almost twice as often as Java developers. This may be attributed to Python conventions that encourage updating *docstrings* along with other code readability improvements.

4.3.7 Improve Code Formatting

This category appears in 21 out of 274 Java PRs (8%) and 41 out of 260 Python PRs (16%). Previous readability models have already addressed code formatting such as line length and indentation (BUSE; WEIMER, 2010; POSNETT; HINDLE; DEVANBU, 2011; DORN, 2012). This improvement aims to visually group related code blocks and make them easier to distinguish from each other. Figure 35 illustrates one type of code readability improvement found exclusively in Java, two in Python, and two that are common to both programming languages.

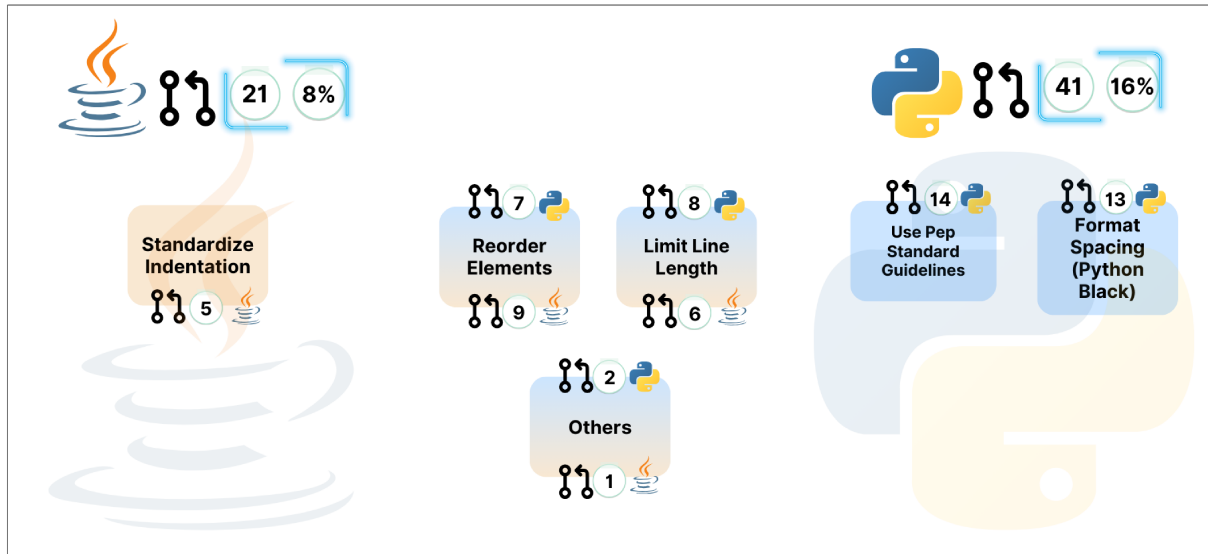


Figure 35 – Types of readability improvements in the Improve Code Formatting category

The main difference observed between Python and Java is that, while Java PRs tend to mention formatting improvements as isolated changes, Python PRs more frequently refer to adherence to coding standards, such as the PEP 8 guidelines (ROSSUM; HETTINGER, 2025) or the use of the Python Black formatter (PYTHONBLACK, 2025).

4.3.7.1 Reorder Elements (☞ 9 PRs) (☛ 7 PRs)

This subcategory focuses on organizing the order of elements, such as attributes and methods, within a Python file or a Java class. It also includes adherence to conventions related to access modifiers and structural consistency.

- ☐ ☛ PR ID = 68095 from *getsentry/sentry* repository: “Class methods that are for external use should be at the top of the class. Moves functions around to be in a more intuitive place. The ordering of class methods should generally follow the ordering in which they are called (to minimize big jumps when ctrl-clicking)”
- ☐ ☛ PR ID = 38093 from *getsentry/sentry* repository: “Only moving methods around to make reading these files easier (no logic changes). Re-ordering these methods to follow: 1. Implemented/Overridden methods. 2. Non ‘_’ prepended methods aka. ‘Public’ instance methods. 3. ‘_’ prepended methods aka. ‘Private’ instance methods. 4. @staticmethods”
- ☐ ☞ PR ID = 7944 from *grpc/grpc-java* repository: “Reordered arguments of `send Response()`, `verifyRequest()` to make them consistent between each other, and upcoming similar `verifyMetadata` helpers”.
- ☐ ☞ PR ID = 371 from *trinodb/trino* repository: “reorder methods so that helper methods are last”.

- ❑ 📄 PR ID = 336 from *oracle/tribuo* repository: “Reorder Access modifiers as per JSR standard”.

4.3.7.2 Limit Line Length (📄 6 PRs) (♣️ 8 PRs)

This subcategory enforces a limit on the number of characters per line, as specified in Oracle (ORACLE, 2025), Google (GOOGLE, 2025) and PEP 8 conventions (ROSSUM; HETTINGER, 2025).

Interestingly, although the purpose is the same in both languages, that is, to limit the number of characters per line so that the entire code remains visible without requiring horizontal scrolling, there appears to be no consensus on the ideal line length. For example:

- ❑ ♣️ PR ID = 213 from *ydb-platform/ydb-python-sdk* repository: “use line-length=120 for better code readability”
- ❑ ♣️ PR ID = 345 from *jarun/buku* repository: “Long lines (140 characters) were split for better readability.”
- ❑ 📄 PR ID = 6013 from *jenkins* repository: “Applies the rectangle rule to some extremely long lines to improve readability and sets the maximum line length at 240 columns”

4.3.7.3 Use PEP Standard Guidelines (♣️ 14 PRs)

In 14 Python PRs, the developers mentioned the use of PEP guidelines. In general, developers mention a set of style guidelines as mentioned in PR ID = 732 from *luong-komorebi/Awesome-Linux-Software* repository: “This commit includes modifications to the code in `asort.py` to improve code formatting and readability according to PEP 8 style guidelines. The changes include consistent indentation, proper spacing around operators, proper capitalization of comments, and using parentheses for print statements for compatibility with Python 3. Please note that these changes are purely stylistic and do not affect the functionality of the script. The goal is to improve code maintainability and readability.”

In some PRs, we observed commits with messages such as “fix styles”, often accompanied by changes in spacing, blank lines, and similar adjustments. In other words, while Java commits tend to focus on specific formatting tasks such as indentation or white space, Python commits more commonly follow conventions such as PEP 8 or simply refer to “style” changes, encompassing a wider range of formatting adjustments.

4.3.7.4 Format Spacing (Python Black) (🔗 13 PRs)

In some PRs, developers explicitly mentioned the use of the Python Black formatter to format files. Some of the improvements observed include the refactoring of placing each parameter of a method on a separate line, the proper use of blank spaces between tokens, and the inclusion of *docstrings* for each method. However, they are not counted under the *Improve Documentation* category, as they were not explicitly described by the developers in the PR descriptions.

However, in some cases, the developers did not mention the use of Python Black, yet the changes clearly reflect its formatting style. For example, Listing 4.17 shows a change from PR ID = 8 of the *jrialland/python-astar* repository, where the developer wrote: “Added line for better readability of nodes.” Initially, the annotators considered this change as a *Limit Line Length*, but GPT 4.1 categorized it as *Format Spacing (Python Black)*. Upon further inspection, the annotators agreed that the formatting reflects the pattern enforced by Python Black, where each item in a dictionary is placed on a new line, regardless of the overall character count. In contrast, PEP 8 allows multiple items per line as long as the 79-character limit is respected. Therefore, unless the developer explicitly mentions line length concerns, it is more appropriate to use the *Format Spacing (Python Black)* label in such cases.

```

1 //before commit
2 nodes = {'A': [('B', 100), ('C', 20)],
3          'C': [('D', 20)], 'D': [('B', 20)]}
4
5 //after commit
6 nodes = {'A': [('B', 100), ('C', 20)],
7          'C': [('D', 20)],
8          'D': [('B', 20)]}

```

Listing 4.17 – Formatting Example

4.3.7.5 Standardize Indentation (🔗 5 PRs)

In five cases, Java developers fixed the indentation, such as setting the indentation size to four spaces. These practices align with the convention of using a single blank line between method declarations, as specified by Oracle and Google, and previously discussed in Chapter 2.

4.3.7.6 Others (🔗 1 PR) (🔗 2 PRs)

This work identified three more instances of code readability improvements classified under the *Improve Code Formatting* category, driven by various types of change.

- ❑ 🛑 Disable Pylint warning.
- ❑ 🛑📄 Prefer One Statement Per Line.

Finding 8) Although we observed some similar formatting improvements among Python and Java developers, such as limiting the number of characters per line or reorganizing class elements, this category suggests a stronger adherence within the Python community to established coding standards like PEP 8. In most Java PRs, no explicit reference to a coding standard was found. This is further supported by the observation that Python PRs often bundle multiple formatting improvements together, whereas Java PRs tend to apply isolated changes without following a specific set of community-driven conventions.

4.3.8 Summary

Table 16 summarizes the categorization of Java and Python PRs. Both languages show a strong focus on clarifying code intent. However, while Java PRs tend to emphasize the elimination of redundant or unnecessary code, Python PRs more often focuses on improving documentation and code formatting according to style guidelines.

Table 16 – Distribution of Java and Python Pull Requests (PR) by category

Category	Python		Java	
	PR Count	%	PR Count	%
Clarify Code Intent	125	48%	125	46%
Prefer Concise Code	80	30%	96	35%
Clean Up	23	9%	43	16%
Improve Code Modularity	38	14%	51	19%
Improve Documentation	65	25%	38	14%
Improve Code Formatting	41	16%	21	8%
Total PRs	260	100%	274	100%

RQ₁ Answer: During the analysis, a comprehensive catalog of 56 different types of code readability improvements was produced, 38 of them identified in Python and 44 in Java, grouped into six distinct categories. The findings indicate that in most cases both Python and Java developers improved the readability of the code by making the code more intuitive. However, Python developers tended to focus on enhancing documentation and code formatting, while Java developers mainly aimed to reduce verbosity and make the code more concise. Some of the changes were subjective in nature and could represent a significant challenge for current readability tools and models to replicate.

4.4 Discussion of the Results

This section proposes to discuss the observed results that produce insights into the taxonomy produced.

4.4.1 Comparing Subcategories in Java and Python

In this work, a total of 56 subcategories with at least three occurrences in either of the programming languages were identified in the six established categories, as shown in Table 17. Among them, 26 subcategories (47%) were common to both languages. Furthermore, 12 subcategories were found only in Python PRs, while 18 were exclusive to Java.

Table 17 – Number of Subcategories for each category by programming language

Category	Python		Java		Shared		Total	
	Sub	%	Sub	%	Sub	%	Sub	%
Clarify Code Intent	6	30%	5	25%	9	45%	20	36%
Prefer Concise Code	2	12%	6	35%	9	53%	17	30%
Clean Up	0	0%	5	71%	2	29%	7	13%
Improve Code Modularity	2	40%	1	20%	2	40%	5	9%
Improve Documentation	0	0%	0	0%	2	100%	2	4%
Improve Code Formatting	2	40%	1	20%	2	40%	5	9%
Total	12	21%	18	32%	26	47%	56	100%

In the *Clarify Code Intent* category, nine subcategories are common to both programming languages. However, five subcategories, *Apply Early Return*, *Use domain-specific types to clarify intent*, *Extract Literals to Enumerated Values*, *Explicit Imports* and *Replace Ternary with If-Else* were observed exclusively in Java, each appearing in at least three PRs. On the other hand, six subcategories were unique to Python: *Include Type Hints and Named Parameters*, *Improve Regex*, *Clarify Positional Indexing*, *Prefer Specific Exception Handling*, *Replace Lookup with Dict* and *Add Parenthesis for Clarity*.

Some of these differences are due to language-specific features. For instance, type hints are optional in Python but unnecessary in Java due to its statically typed nature. Similarly, ternary operations in Python are expressed inline (e.g., `status = "adult" if age >= 18 else "minor"`), making them less likely to be replaced by `if-else` statements as seen in Java. Interestingly, while both languages support regular expressions, we found that Java PRs did not focus on improving regex usage, in contrast to Python, where such improvements were present.

In the *Prefer Concise Code* category, nine subcategories are shared between the programming languages. Python includes two exclusive subcategories: *Prefer Idiomatic Conditions*, and *Simplify Import Statements*. In contrast, Java presents six exclusive subcategories: *Replace Anonymous Classes With Functional Style*, *Replace Lambda With*

Method Reference, *Prefer Structured Resource Management*, *Replace Switch with If-Else*, *Replace If-Else with Switch*, and *Favour Diamond Operators*. It is interesting to observe that Python PRs often focus on eliminating redundant tokens from string manipulation, a behavior that is more rarely seen in Java. In contrast, Java PRs tend to simplify code by reducing verbosity through features such as lambdas, method references, *pattern matching*, and `try-with-resources`. This contrast suggests that such practices may already be well established in Python, which could explain the consistently higher number of conciseness-related changes observed in Java PRs.

In the *Clean Up* category, all two Python subcategories were also observed in Java. However, the remaining five Java subcategories were either not observed in Python or appeared in fewer than three PRs. Still, no behavior in this category was identified that could not potentially occur in Python as well.

In the *Improve Code Modularity* category, subcategories such as *Extract method from large function* and *Move logic to another file* were expected to have instances in both programming languages. However, the use of the builder design pattern occurs exclusively in Java PRs, maybe due to the adoption of design patterns in this programming language (GAMMA et al., 1994). In Python, we observed some specific behaviors, such as creating data classes and extracting lambdas.

In the *Improve Code Formatting* category, Java and Python shares the *Reorder Elements* and *Limit Line Length* subcategories. However, while Java contains instances related to *Standardize Indentation*, Python is often aligned with official style guides such as PEP, or enforced by formatting tools like Black.

4.4.2 Linter Coverage of Readability Improvements

In Section 4.3.1, some types of Java code readability improvements were mapped to their corresponding SonarLint rules, which detect such issues when they are not addressed in the code. In some of the samples, we also observed PRs explicitly referencing the use of linters such as SonarLint and Checkstyle for Java, or Pylint and Flake8 for Python, as well as the Black formatter, all of them in the *Improve Code Formatting* category.

Moreover, some types of code readability improvements can be recommended by readability models and/or tools. For example, subcategories such as (*“Limit Line Length”*, *“Standardize Indentation”*, *“Include New Comments and docstrings”*) have already been used as metrics in code readability models (BUSE; WEIMER, 2010; DORN, 2012). Other models could identify a complex and unreadable line of code, e.g., *“Replace Ternary with If-Else”* (SCALABRINO et al., 2018). The use of the cognitive complexity metric (CAMPBELL, 2018) can suggest minimizing the complexity of control flows (e.g., *“Reduce Conditional Nesting”*, *Prefer Idiomatic Iteration*, *“Replace Anonymous Classes With Lambda”*). Some tools can detect code smells such as *Large Class* and *Long Method*, suggesting the refactoring to extract a class or a method.

However, the findings indicate that some opportunities for improvement could be challenging to detect by code readability tools. For example, the “*Replace General with Specialized API*” could be an individual developer’s decision based on his/her own experience. In some observations, `java.util.Map` was replaced with `com.google.common.collect.ImmutableMap`, or `java.util.ArrayDeque` with `java.util.concurrent.ConcurrentLinkedDeque`. Another example is the *Include Modifiers and Annotations*, where an annotation or a modifier could both be considered redundant code if the variable does not need them. For example, the PR ID = 3152 from *apache/pinot* repository asked to remove the unused `@Nonnull` annotation to maintain readability.

An additional challenge is in improving the name, as most of the 92 Python and Java PRs modified names based on their semantic alignment with the intended purpose of the code. Tools such as SonarLint typically suggest naming improvements based on language conventions, which may not be sufficient to capture the developer’s intent or address the underlying motivation for the change.

4.4.3 Use of GPT as an Annotator

In this study, rather than relying solely on human evaluation, the open coding methodology incorporated GPT 4.1 as a secondary annotator. The goal was not to assess the quality of GPT 4.1 as an annotator or to rely on its output, but to use it as a tool to help surface potential inconsistencies or ambiguous cases among the 893 instances (437 in Python and 456 in Java), particularly since the human annotators had more experience with Java than with Python.

In general, the most valuable contribution of GPT 4.1 is to motivate reviewers to reevaluate labels that were semantically close. For example, GPT 4.1 often disagreed with human annotators when distinguishing between labels such as *Unwrap Unnecessary Else* and *Remove Redundant If-Else Conditions*, which involved slight semantic differences. In other situations, it helped clarify the distinction between *Format Spacing (Python Black)* and *Limit Line Length*, leading to a more consistent categorization. Even when disagreements did not result in changes, they led to a refinement of the prompts and input context provided to the model, and, in some cases, revealed that the annotators’ own understanding of a sample was not entirely clear, leading to the decision to discard those instances.

In cases of disagreement, we believe that providing broader context (e.g. the full class rather than a small code snippet) and adjusting parameters such as temperature could improve alignment between LLMs and human judgment. Exploring alternative models, such as DeepSeek or Gemini, could also be beneficial. However, since the purpose of this study was not to replace human annotators with LLMs in the open coding process, we opted for a more conservative, assistive use of the model.

4.5 Implications for Developers and Researchers

This section discusses the implications of this study for researchers and developers.

4.5.1 Researchers

For researchers, the taxonomy of 56 types of code readability improvements could provide valuable insight into improving readability models and ASAT. Despite some of these improvements being already observed in previous work (e.g., improve naming), others can offer a new set of features that align better with developers' perceptions of what constitutes readable code.

Moreover, the prevalence of each type reflects what developers primarily consider when aiming to improve code. For example, since 92 out of 534 PRs (17%) focus on improving naming, this highlights an opportunity for improvements in ASATs. Rather than relying solely on the production of readability warnings by naming patterns that do not match regular expressions, these tools could evolve to suggest more semantically appropriate names.

Although 47% of the cataloged improvements are shared between the Python and Java communities, the remaining 53% consists of code readability improvements that are exclusive to each language. This opens up potential discussions about the development of models that account for language-specific characteristics and conventions.

4.5.2 Developers and Practitioners

For developers, the samples demonstrate that developers frequently prioritize improving code readability to make it more intuitive, modular, and concise. We have observed a notable emphasis on producing a more intuitive code by improving variable and method names, simplifying control flows, reducing verbosity in the code, and improving modularity.

This work also identified several cases where developers improve code readability by using newer features in the Python and Java programming language evolution. Examples: `try-with-resources`, `lambdas`, `pattern matching` and others. These findings emphasize the importance of embracing and familiarizing yourself with these features to enhance code readability and take advantage of language advances.

While the Java community maps important formatting aspects, such as indentation, line length limits, method, and attribute organization within classes, we observed that the Python community tends to adopt formatting improvements in a more standardized way, often guided by conventions such as PEP 8 and automated tools such as Python Black. This contrast suggests that, unlike Python, Java could still lack a universally adopted and consistently applied formatting convention across the community.

4.6 Threats to Validity

As in the previous chapter, this section is structured around construct, internal, and external validity.

4.6.1 Construct Validity

- ❑ **Readability improvements dataset** - Studying the types of readability improvements performed by developers through mining software repositories has inherent limitations. One key limitation is that we can only observe cases where developers explicitly mention readability-related keywords in the PR description. However, there may be other instances in which readability improvements were made without explicitly being mentioned. Furthermore, as discussed in the proposed methodology section, the analysis was limited to the textual content of PRs, which could be requested via the GitHub API. Readability improvements may also be mentioned in other sections, such as code comments, which were not included in this study. Identifying such instances would require analyzing the entire source code of the projects and is suggested as a direction for future work.
- ❑ **Types of Code Readability Improvements Interpretation** - the interpretation of the types of changes performed by developers relies on reading and labeling their descriptions. This classification process is subject to potential subjectiveness and imprecision. As described in the proposed methodology section, this threat is mitigated by employing two independent annotators for each instance, followed by a collaborative resolution of any disagreements.
- ❑ **Readability-Focused PR construction** - The construction of this dataset is inherently subject to a degree of subjectivity and potential errors, such as false negatives, i.e., cases where a PR focused on improving readability may have been excluded due to misinterpretation by the annotators. However, since the process of identifying candidate PRs is highly susceptible to false negatives, as some readability improvements may rely on specific keywords, this approach prioritizes minimizing false positives over reducing false negatives.
- ❑ **LLM as a potential Annotators** — Although LLM outputs could have issues, particularly due to potential hallucinations (ZUCCON; KOOPMAN; SHAIK, 2023), the inclusion of GPT 4.1 in this analysis served more as a cross-checking mechanism than as a standalone annotator. For example, in cases where discrepancies were found, such as incomplete extracted code snippets, the authors revisited and corrected the data manually. In this sense, the LLM helped as a support tool to assist, rather than replace, human decision making.

4.6.2 Internal Validity

Developers Descriptions and Code Changes Retrieval: The retrieval of developer descriptions and code changes from GitHub PRs was performed manually by the authors. To mitigate potential data extraction errors and misinterpretations, these fields were also used as input in the LLM evaluation. In particular, instances in which the LLM produced a different categorization were manually reviewed, helping to minimize the impact of possible extraction inaccuracies.

4.6.3 External Validity

Programming Language: The scope of this study is limited to the Java and Python programming languages, which, despite sharing some fundamental concepts, differ in syntax, type systems, and idiomatic practices. These differences can influence how developers express readability improvements. Programming languages from other paradigms, such as functional or logic programming, are not included in this study.

4.7 Concluding Remarks

In this study, a taxonomy was produced with six categories and 56 documented types of code readability improvements performed by Python and Java programmers. The main findings of this investigation can be summarized as follows.

- ❑ Both Python and Java developers share some common motivations for improving code readability, such as clarifying the intention of the code, especially improving the names. Since names typically account for around 70% of the characters in the source code (DEISSENBOCK; PIZKA, 2005), it is expected that developers in both languages pay attention to naming conventions and semantic adherence to their purpose.
- ❑ Although both programming languages include instances of all categories identified in this study, Java developers demonstrated a higher concern with verbose and redundant code, which can potentially be reduced. In contrast, Python developers more frequently addressed code documentation and correct formatting using standards such as PEP 8 guidelines.
- ❑ Similar labels were found in both languages, especially those that involve the manipulation of `if-else` structures, along with language-specific labels. In Python, labels related to the use of type hints, parameter naming, and lambda extraction were identified. In contrast, Java showed a stronger focus on labels involving features such as code reduction using lambdas, streams, and method references, such as the use of design patterns like Builder, among others.

- Although readability improvements can be expressed using various keywords, we found that developers most frequently use the keyword *readability* when referring to changes aimed at making code easier to read and understand. Other terms such as *understandability* and *legibility* also refer to similar goals, but appear less frequently in merged PRs. In contrast, keywords used in previous studies such as *clarity*, *easier to read*, *comprehensible*, *readable*, and *understandable* are highly prone to false positives, often referring to aspects unrelated to code readability improvements. Automating this identification process remains a challenging task, as developers refer to code readability improvements in many ways.
- When properly guided, LLMs can serve as additional source code annotators, complementing human assessment and supporting the open coding process. However, the experience in this study revealed three key findings. First, LLMs requires as much context as possible, since developer-written descriptions were often insufficient and had to be supplemented with both before and after commit code. Second, while human annotators in open coding are generally free to create new labels as needed, we found this approach impractical with LLMs. Even simple name-based categorizations can result in an unmanageable number of labels, as developers often use their own terminology to describe their changes. And third, LLMs were more effective in identifying ambiguously interpretable samples, which is particularly helpful in open coding to avoid incorrect labeling. However, based on this experience, we do not recommend using LLMs for unsupervised coding.

Several directions for future work can be considered. For example, although Python and Java have distinct characteristics, it would be valuable to catalog readability improvements in languages from other paradigms, particularly fully functional ones such as Elixir. This language has already been the subject of research in other areas, such as refactoring operations (VEGI; VALENTE, 2022).

Another direction for future work could be to conduct a more refined investigation of the keywords most suitable to use for different types of study, such as those focused on readability, refactoring, performance, maintainability, and others. The goal would be to minimize the occurrence of false negatives, that is, PRs that should have been included as candidates in the sampling but were not selected due to keyword limitations.

Automated Code Readability Improvements

The previous chapters evaluated the readability of code generated by LLMs or recommended by code search tools (Chapter 3), as well as the types of readability improvements actually made by developers during incremental changes in GitHub repositories (Chapter 4). However, one of the main challenges is the implementation of automated improvements to the source code. This chapter proposes an approach that utilizes LLMs to improve the readability of the code, guided by recommendations from linters such as SonarLint.

This chapter is organized as follows: the motivation for proposing the automated model is first described, followed by the presentation of the research question and the methodology adopted to answer it. The results are then described and discussed in terms of their implications for researchers and practitioners, concluding with an examination of the study's threats to validity.

5.1 Context

Software projects, particularly those hosted on GitHub, often include code that is classified as unreadable according to automated readability models (PIANTADOSI et al., 2020). Even small code fragments, such as code snippets, frequently exhibit readability warnings raised by linters. As presented in Chapter 3, out of 1,635 code snippets analyzed, 611 (37%) contained readability warnings identified by SonarLint.

Although significant progress has been made in assessing if the code is readable or not (BUSE; WEIMER, 2010; POSNETT; HINDLE; DEVANBU, 2011; DORN, 2012; SCALABRINO et al., 2018; ROY et al., 2020), and in identifying code fragments that could potentially be improved (SONARLINT, 2025b; PMD, 2025; CHECKSTYLE, 2025), this does not guarantee that developers will actually improve the readability of the code. They may choose to follow these suggestions or simply ignore them.

Consequently, some models have been proposed to automate code readability improvements, with the aim of reducing the manual effort required by developers (VITALE et al., 2023; LORIOT; MADEIRAL; MONPERRUS, 2022). Although these approaches have shown promising results in improving the readability of the code automatically, they often operate on a limited subset of code readability improvements, such as adding comments, renaming identifiers, or reorganizing code blocks (VITALE et al., 2023). As discussed in Chapter 4, developers improve readability for a wide range of reasons, and tools such as SonarLint provide numerous recommendations related to improving code readability. In general, code readability remains a complex and multidimensional problem for which comprehensive automated support is still an open challenge.

As LLMs have been gaining increasing attention from the software development community, several studies have proposed evaluating their effectiveness in generating code for various tasks, including refactoring operations (ZHANG et al., 2024), bug fixing (SOBANIA et al., 2023), suggestions of code changes, and other programming activities on GitHub (TUFANO et al., 2024). Moreover, LLMs offer an alternative perspective by producing modified versions of the source code originally written by developers (EBERT; LOURIDAS, 2023). ChatGPT, in particular, has also shown potential in supporting code comprehension (NAM et al., 2024) and generating educational code snippets that are often more readable than those authored by developers on platforms like Stack Overflow (DANTAS; ROCHA; MAIA, 2023b).

However, as some recent studies have shown, LLMs are highly dependent on the quality and structure of the prompts. For example, in a recent work, ChatGPT provided correct answers to only 36.4% of the questions when using benchmark issue reports. However, when the prompts are increased using technical reports, its precision improves by 30% (TAMANNA et al., 2025).

In this work, an approach is proposed for using LLMs to automatically improve code readability based on SonarLint warnings. Similar combinations have been explored in other contexts, such as building a Long short-term Memory (LSTM) model guided by Checkstyle (CHECKSTYLE, 2025) recommendations (LORIOT; MADEIRAL; MONPERRUS, 2022), or leveraging ASATs to enhance the code review process in conjunction with LLMs (JAOUA; SGHAIER; SAHRAOUI, 2025). However, to our knowledge, no previous work has specifically explored the combination of LLMs and ASATs with the explicit goal of improving the readability of the code. This study provides the following contributions.

- ❑ A study analyzing to what extent LLMs automatically improve code readability by comparing correction, persistence, or even the introduction of new readability warnings in code snippets.
- ❑ A qualitative analysis of the types of code readability improvements that LLMs were

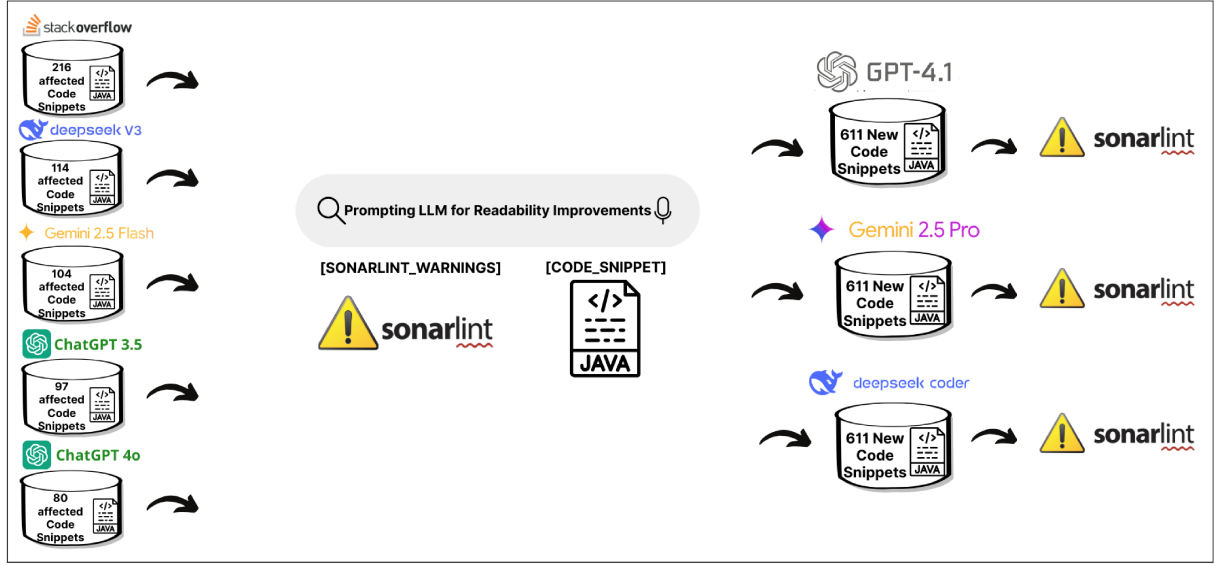


Figure 36 – Overview of the proposed methodology to answer RQ₁

most and least effective in addressing the SonarLint warnings.

- ❑ A discussion involving the changes performed by the LLM and their impact on the functional behavior of the code snippet
- ❑ A replication package containing the SonarLint scripts used to extract readability warnings from code snippets, along with the scripts and statistical tests to evaluate the performance of each LLM (DANTAS; MAIA, 2025).

5.2 The Proposed Methodology

This section aims to explain the methodology to answer the following research question:

RQ₁: Can LLMs improve code readability based on SonarLint warning recommendations?

This question investigates whether LLMs can remove SonarLint-identified code readability warnings without introducing new ones.

Figure 36 presents the overall approach adopted to answer the proposed research question. The following subsections describe the process of prompting LLMs to extract the new code snippets and analyze them.

5.2.1 Selecting Affected Code Snippets

In Chapter 3, as illustrated in Figure 36, a total of 611 code snippets presented readability warnings detected by SonarLint. These include 216 Stack Overflow snippets recommended by CROKAGE, 114 from DeepSeek, 104 from Gemini, 97 from ChatGPT 3.5

(extracted in March 2023), and 80 from ChatGPT 4o. Table 18 provides an example of the warnings generated by SonarLint. For the query with ID = 8, “How to create a service to watch for directory changes in Java”, SonarLint reported nine warnings in the Gemini-generated snippet:

1. *java:S1220* on line 1, it is not classified as a readability warning
2. *java:S1068* on line 15, it is classified as a readability warning
3. *java:S106* on line 82, it is not classified as a readability warning, since code snippets generally contains **System** messages to show the behavior of the code
4. *java:S3776* on line 120, it is classified as a readability warning, with cognitive complexity = 32, i.e., a number of loops and statements inside a method that is higher than the threshold = 15, as determined by SonarLint tool.
5. *java:S135* on line 121, it is classified as a readability warning
6. *java:S3457* on line 160, it is classified as a readability warning
7. *java:S1066* on line 164, it is classified as a readability warning
8. *java:S1192* on line 200, it is classified as a readability warning
9. *java:2142* on line 275, it is not classified as a readability warning, since this is a classified by SonarLint as a bug rule.

The number of affected code snippets and the corresponding warning types were recorded by rule key and are presented in Chapter 3. The information extracted from SonarLint, such as the description of the readability warning and the affected line of code, could be used as input to the LLM, with the goal of generating a revised version of the code that aims to remove the warnings.

5.2.2 Prompting LLMs for Code Readability Improvements

In this process, each of these 611 affected code snippets was provided as input to the LLMs GPT, DeepSeek and Gemini, with the aim of correcting the identified warnings. As a result, 1,833 new code snippets were generated, 611 by each LLM.

While the 611 code snippets with readability warnings were collected from Stack Overflow and through manual interactions with free versions of LLMs such as ChatGPT 3.5, ChatGPT 4o, Gemini 2.5 Flash, and DeepSeek V3, this chapter examines 1,833 code snippets automatically extracted via the APIs of GPT-4.1, Gemini 2.5 Pro, and DeepSeek Coder, a model specifically designed for code-related tasks (DEEPSEEK, 2025a). In this

Table 18 – SonarLint recommended modifications on a Gemini code snippet.

Rule Key	Start Line	End Line	Message
java:S1220	1	1	Move this file to a named package.
java:S1068	15	15	Remove this unused <code>directoryToWatch</code> private field.
java:S106	82	82	Replace this use of <code>System.err</code> with a logger.
java:S3776	120	120	Refactor this method to reduce its Cognitive Complexity from 32 to the 15 allowed.
java:S135	121	121	Reduce the total number of <code>break</code> and <code>continue</code> statements in this loop to at most one.
java:S3457	160	160	Use <code>%n</code> instead of <code>\n</code> to produce the platform-specific line separator.
java:S1066	164	164	Merge this <code>if</code> statement with the enclosing one.
java:S1192	200	200	Define a constant instead of duplicating the literal " <code>Created directory: </code> " three times.
java:S2142	275	275	Either re-interrupt this method or rethrow the <code>InterruptedException</code> that can be caught here.

context, GPT refers to the actual underlying transformer models accessed via API, while “ChatGPT” denotes the web-based application layer built atop such models.

Figure 37 illustrates this process. In the query with ID = 8, as shown in Table 18, the code snippets generated by ChatGPT, DeepSeek, and Stack Overflow did not raise any readability warnings. However, Gemini raised five warnings and ChatGPT 3.5 raised one. Then these code snippets were submitted to the three LLMs in an attempt to address the detected warnings. The warning raised by ChatGPT 3.5 was eliminated by all three LLMs. In contrast, regarding the warnings raised by Gemini, only the version generated by Gemini 2.5 Pro itself managed to remove them. DeepSeek Coder failed to resolve the *java:S135* warning, while GPT 4.1 did not fix either the *java:S135* or *java:S3776* warnings and even introduced a new one: *java:S3626*.

The prompt used to ask the LLMs to improve code readability was created using a context defined through the system prompt, as shown in the text box below. In this prompt, [CODE_SNIPPET] represents the code snippet to be improved, and [SONARLINT_WARNINGS] contains the warnings raised by SonarLint, including the message, start, and end lines. A script was created to construct the prompts and extract the 1,833 code snippets using the *openai API* for GPT 4.1 and DeepSeek coder versions, and the *google.generativeai API* for the Gemini 2.5 Pro version. All using the parameter *MAX_TOKENS* = 8,192, that is, each request is limited to this number of tokens, and *TEMPERATURE* = 0.0, as

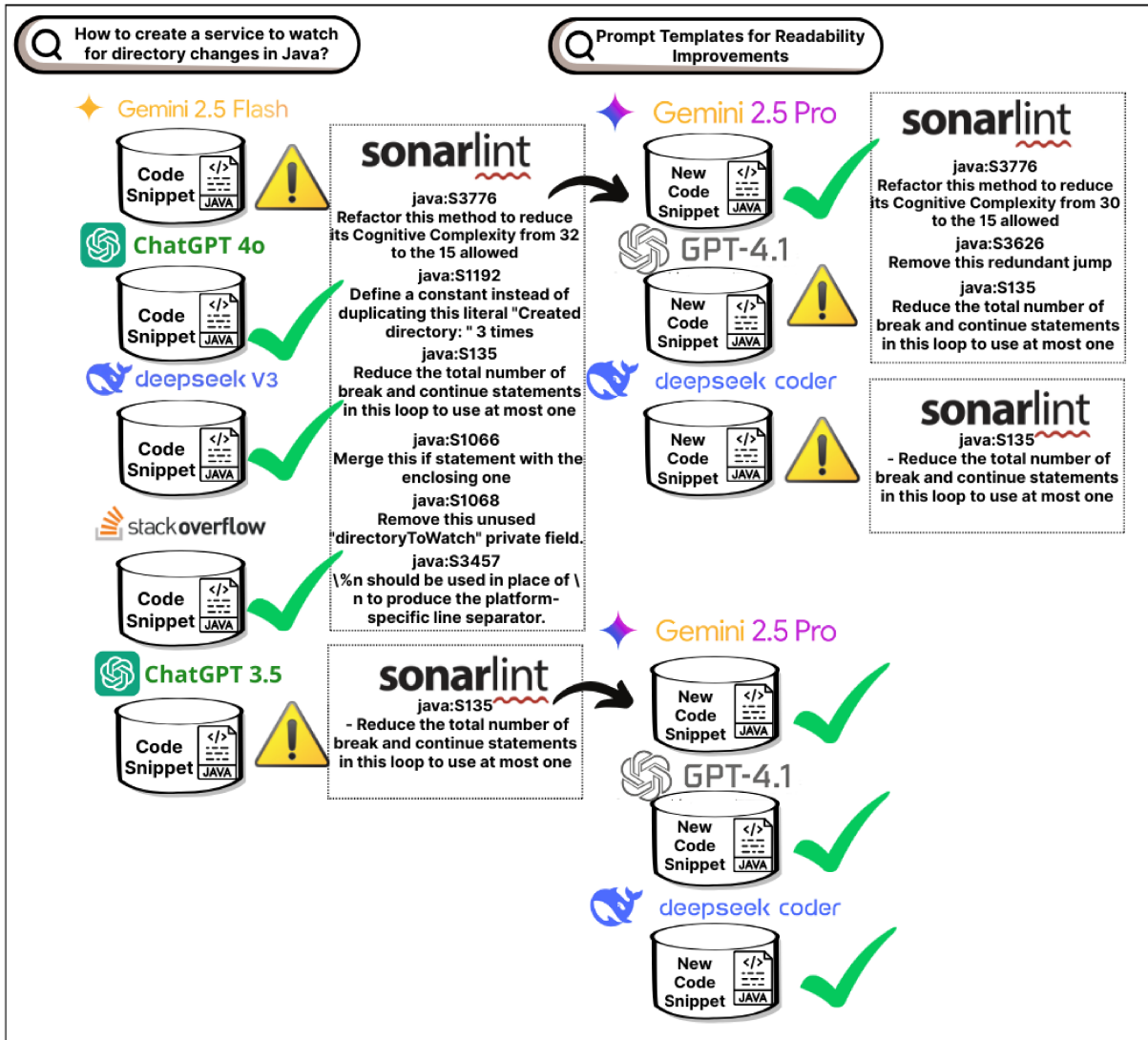


Figure 37 – Improving Code Readability Example on Query ID = 26 “Odbc connection to Mysql in Java”

observed in previous work that lower temperature settings for code refinement tasks tend to produce better and more stable results (GUO et al., 2024).

Prompt Templates used for readability improvements

System: "You are a software developer specialized in the Java programming language, with expertise in improving the readability of the code based on a list of recommendations."

User: "Given the following Java code snippet: [CODE_SNIPPET], the following improvements are recommended: [SONARLINT_WARNINGS]. Please provide a revised version of the code snippet that applies the recommended improvements. Only the revised code snippet, without additional text."

5.2.3 Statistical Tests

Statistical tests are conducted from two complementary perspectives. The first verifies whether a given LLM corrected significantly more readability warnings than the others. For this analysis, the Friedman test followed by the Nemenyi post hoc procedure was applied, as in Chapter 3. The second perspective examines whether a given LLM corrected significantly more code snippets than the others. For this purpose, a dataset was created as shown in Table 19, which includes all 611 affected code snippets (e.g., query ID = 100 contains readability warnings in both the ChatGPT 2023 and DeepSeek versions). For each LLM, a binary classification was performed to indicate whether it completely removed all readability warnings without introducing new ones. In this example, the snippet corresponding to query ID = 100 was successfully corrected by GPT-4.1, DeepSeek Coder, and Gemini 2.5 Pro, with all warnings eliminated and no additional ones introduced.

The Cochran Q test (COCHRAN, 1950) was then applied to assess whether the three paired binary sets (i.e., GPT 4.1, DeepSeek Coder, and Gemini 2.5 Pro) differ in their ability to correct code snippets. This non-parametric test is suitable for binary outcomes in a within-subject design, where each of the 611 code snippets with readability warnings represents a paired observation across the three LLMs. If the global null hypothesis of equality is rejected, the test indicates that at least one model differs from the others, but does not specify which one. To complement this analysis, McNemar’s test was applied for pairwise comparisons between LLMs (e.g., GPT vs. DeepSeek). McNemar’s test examines whether the number of disagreements between two classifiers is symmetric, providing a finer analysis of whether one model significantly outperforms another on the same set of binary results. Thus, Cochran’s Q establishes the overall difference among multiple LLMs, while McNemar’s test identifies the specific pairs responsible for such differences.

Table 19 – Dataset used to extract statistical tests.

Query and LLM code snippet	GPT 4.1	DeepSeek Coder	Gemini 2.5 Pro
100 – ChatGPT 2023	1	1	1
100 – DeepSeek	0	1	0
101 – Stack Overflow	1	1	1
103 – Stack Overflow	0	1	1

5.3 Results and Findings

This section presents the results separated by the number of warnings and the number of code snippets corrected by each LLM, as well as a qualitative analysis of each category defined in Chapter 3.

5.3.1 Number of Readability Warnings

Table 20 presents the number of readability warnings before and after correction by each LLM. The 611 code snippets affected by readability warnings, as reported in Chapter 3, contained a total of 1,109 warnings. GPT 4.1 reduced this number to 281, achieving a reduction of 74.7%, while DeepSeek Coder achieved a reduction of 82.6%, and Gemini 2.5 Pro a reduction of 81.2%. In particular, both DeepSeek Coder and Gemini 2.5 Pro eliminated a substantially larger proportion of warnings compared to GPT-4.1. This result is particularly interesting given that the free version of ChatGPT produced code snippets with the fewest readability warnings in Chapter 3, whereas GPT-4.1 accessed via the API showed the weakest performance when refactoring those snippets based on SonarLint recommendations, especially in the *Clean Up* category.

To verify whether the remaining warnings differed significantly between models, the Friedman statistical test was applied. The test returned $p = 6.51 \times 10^{-7}$, rejecting the null hypothesis and confirming that there are differences. A subsequent Nemenyi post hoc test indicated that the difference between DeepSeek Coder and GPT-4.1 was nearly significant ($p = 0.06$).

Another relevant aspect is the reduction rate per category, which shows that LLMs eliminated the vast majority of warnings related mainly to *Clarify Code Intent* and *Prefer Concise Code*. Gemini 2.5 Pro proved to be more effective than the others in reducing warnings related to conventions and modularity, whereas DeepSeek Coder performed better in the *Clean Up* category.

Table 20 – Total number of readability warnings before refactoring and after correction by each LLM, along with the percentage reduction relative to the baseline. ($n = 1,109$ readability warnings)

Category	Before	GPT-4.1		DeepSeek Coder		Gemini 2.5 Pro	
	Count	Count	Reduction	Count	Reduction	Count	Reduction
Clean Up	635	220	65.4%	153	75.9%	193	69.6%
Clarify Code Intent	179	15	91.6%	11	93.9%	6	96.6%
Prefer Concise Code	158	18	88.6%	7	95.6%	3	98.1%
Convention	116	22	81.0%	16	86.2%	5	95.7%
Improve Code Modularity	21	6	71.4%	6	71.4%	2	90.5%
Total	1109	281	74.7%	193	82.6%	209	81.2%

5.3.2 Number of Affected Code Snippets

Table 21 presents the number of code snippets affected by readability warnings before and after correction by each LLM. In GPT 4.1, 179 code snippets are still affected

by readability warnings, which corresponds to a reduction of 70.7%. DeepSeek Coder achieved a reduction of 79.4%, while Gemini 2.5 Pro achieved a reduction of 77.7%. Similarly to the overall number of readability warnings, both DeepSeek Coder and Gemini 2.5 Pro eliminated a considerably larger proportion of affected snippets compared to GPT-4.1.

Table 21 – Total number of affected code snippets before refactoring and after correction by each LLM, along with the percentage reduction relative to the baseline. ($n = 611$ snippets)

Category	Before	GPT-4.1		DeepSeek Coder		Gemini 2.5 Pro	
	Count	Count	Reduction	Count	Reduction	Count	Reduction
Clean Up	370	148	60.0%	99	73.2%	125	66.2%
Clarify Code Intent	169	14	91.7%	11	93.5%	6	96.4%
Prefer Concise Code	156	16	89.7%	7	95.5%	3	98.1%
Convention	106	17	84.0%	15	85.8%	5	95.3%
Improve Code Modularity	19	3	84.2%	4	78.9%	1	94.7%
Total	611	179	70.7%	126	79.4%	136	77.7%

Table 22 presents the results of the Cochran Q and McNemar tests used to assess whether there are statistically significant differences between the 611 new code snippets generated by each of the three LLMs. The Cochran Q test (p -value < 0.01) indicates a significant difference in the LLMs to correct the code (i.e., fix all readability warnings raised by SonarLint without introducing new ones).

Table 22 – Cochran’s Q test and McNemar’s pairwise comparisons among LLMs based on corrected code snippets (i.e., all warnings fixed and no new warnings introduced).

Test	Comparison	none	first	second	all	p-value
Cochran’s Q	All (3 LLMs)	-	-	-	-	4.19×10^{-7}
McNemar’s	GPT 4.1 vs DeepSeek C.	103	23	76	409	8.53×10^{-8}
	GPT 4.1 vs Gemini Pro	104	32	75	400	3.92×10^{-5}
	DeepSeek C. vs Gemini Pro	72	64	54	421	0.407

The pairwise McNemar tests revealed statistically significant differences between GPT 4.1 and DeepSeek Coder, and between GPT 4.1 and Gemini 2.5 Pro. In the comparison between GPT 4.1 and DeepSeek Coder, neither model corrected all readability warnings in 103 out of 611 snippets, while both succeeded in 409 cases. GPT 4.1 corrected 23 code snippets that DeepSeek Coder did not, whereas DeepSeek Coder corrected 76 snippets that GPT 4.1 did not address. Given the (p -value < 0.01), the difference is statistically

significant, indicating that DeepSeek Coder and Gemini 2.5 Pro successfully corrected significantly more code snippets than GPT 4.1.

Finding 1) Despite the fact that their free versions 3.5 and 4o generated code with fewer SonarLint warnings, as shown in Chapter 3, the GPT model in version 4.1 was the least effective LLM in addressing and correcting those warnings when asked.

5.3.3 Introduction of New Readability Warnings

Although the previous subsections provided information on the reduction of warnings and the resolution of affected code snippets, they did not address the perspective of newly introduced warnings that were not present in the original code. For example, among the 611 snippets affected by readability warnings, 370 contained issues related to the *Clean Up* category. After applying LLMs to remove these warnings, 148 snippets still exhibited *Clean Up* issues. However, a portion of these 148 cases corresponded to snippets that were not among the original 370, indicating that new warnings were introduced during the correction process.

Table 23 illustrates the number of newly introduced readability warnings and affected code snippets for each LLM. GPT-4.1 introduced 109 new warnings in 89 snippets, DeepSeek Coder introduced 67 across 58 snippets, and Gemini 2.5 Pro introduced 75 across 71 snippets.

Combining the information from Tables 20, 21, and 23, it can be observed that GPT-4.1 ended with 281 readability warnings in total, of which 105 were newly introduced, while the remaining 176 correspond to original warnings that were not corrected. A similar reasoning applies to the number of code snippets: 179 snippets still contained readability warnings, 89 of which had newly introduced warnings, whereas the other 90 correspond to cases where existing warnings were not fully resolved and no new warnings were introduced.

5.3.4 Category Analysis

To better understand the performance of the LLMs in specific types of code readability improvement, 79 SonarLint rules are shown grouped by their respective semantic category. In this process, two SonarLint rules were raised in the new code snippets produced by the LLM that was not raised in the original version, and the remaining 77 rules were extracted from Chapter 3.

Table 23 – Introduced readability warnings by category and LLM (counts of warnings and affected snippets). Totals reflect unique snippets overall and may not equal the sum across categories.

Category	GPT-4.1		DeepSeek Coder		Gemini 2.5 Pro	
	Warn.	Snip.	Warn.	Snip.	Warn.	Snip.
Clean Up	79	72	50	46	69	66
Clarify Code Intent	5	5	1	1	1	1
Prefer Concise Code	12	12	6	6	2	2
Convention	8	8	8	8	2	2
Improve Code Modularity	1	1	2	2	1	1
Total	105	89	67	58	75	71

5.3.4.1 Clean Up

Table 24 shows the number of readability warnings corrected, uncorrected, and introduced in the 16 SonarLint rules classified as *Clean Up* (15 extracted from Chapter 3 and one new rule introduced, i.e., *java:S3985*). To analyze the table, consider the rule *java:S1854*. After submitting these snippets to the LLMs for readability improvement, 180 instances were corrected by GPT 4.1, 195 by DeepSeek Coder, and 182 by Gemini 2.5 Pro. However, 63 instances were not corrected by GPT 4.1, 48 of DeepSeek Coder, and 61 of Gemini 2.5 Pro. Additionally, this rule was introduced 8 times by GPT 4.1, 5 by DeepSeek Coder, and 3 by Gemini 2.5 Pro. “Total” row summarizes the number of readability warnings corrected, not corrected, and introduced by the LLMs. For example, GPT 4.1 corrected 494 readability warnings, not corrected 141, and 79 new readability warnings from *Clean Up* category were introduced that were not originally affected.

The rules *java:S1854* and *java:S1481* are related to unused assignments and variables not used in the code. For example, in query ID = 2 *How to remove an element from an array in Java?*, the original code produced by ChatGPT in 2023 contains a variable that receives the array without the element removed `originalArray = newArray`. However, the code does not contain any operations that use this variable after this assignment, raising the warning *java:S1854*. When the LLM are asked to fix the warning, this assignment is excluded, stop raising the warning.

A potential issue with removing a variable or assignment in a code snippet is that the excluded element may represent the final result or a key part of the intended solution. In this regard, DeepSeek Coder produced fewer problematic cases, primarily because instead of deleting the entire line of code, it simply removed the assignment to the unused variable. For example, in query ID = 144, “how to create a GeoTIFF in Java?”, shown in Listing 5.1, the variable `transformToEpsg4326` is declared but never used. As a result, the assignment appears redundant and raises both SonarLint issues *java:S1854* and *java:S1481*. GPT 4.1 and Gemini 2.5 Pro addressed the warnings by removing the entire line `MathTransform`

Table 24 – Number of readability warnings by status (Corrected, Not Corrected, Introduced) for each SonarLint rule and LLM in the *Clean Up* category (Gem = Gemini 2.5 Pro, GPT = GPT 4.1, Deep = DeepSeek Coder).

Rule	Corrected			Not Corrected			Introduced		
	GPT	Deep	Gem	GPT	Deep	Gem	GPT	Deep	Gem
java:S1854	180	195	182	63	48	61	8	5	3
java:S1481	177	193	178	61	45	60	10	8	6
java:S125	54	60	59	6	0	1	8	1	12
java:S1186	8	9	9	1	0	0	32	22	35
java:S1068	26	25	27	1	2	0	1	2	2
java:S2094	6	5	8	3	4	1	13	7	8
java:S1172	7	7	7	0	0	0	5	3	0
java:S108	4	7	9	5	2	0	0	0	0
java:S1144	8	8	8	0	0	0	0	0	0
java:S4925	7	7	6	0	0	1	0	0	0
java:S1130	5	4	5	0	1	0	1	1	2
java:S3457	5	5	5	0	0	0	0	0	0
java:S1116	3	3	3	0	0	0	0	0	0
java:S2326	2	2	3	1	1	0	0	0	0
java:S1871	2	2	2	0	0	0	1	0	1
java:S3985	0	0	0	0	0	0	0	1	0
Total	494	532	511	141	103	124	79	50	69

`transformToEpsg4326 = CRS.findMathTransform(targetCRS, sourceCRS)`, but this caused new warnings in the remaining variables `sourceCRS` and `targetCRS`, which also became unused. In contrast, DeepSeek Coder preserved the line, but removed only the assignment to `transformToEpsg4326`, avoiding the introduction of new warnings.

```

1 // Original Stack Overflow Code Snippet
2 CoordinateReferenceSystem sourceCRS = CRS.decode("EPSG:4326",
    true);
3 CoordinateReferenceSystem targetCRS = CRS.decode("EPSG:32612");
4 MathTransform transformToEpsg4326 = CRS.findMathTransform(
    targetCRS, sourceCRS);
5
6 // Refactoring ChatGPT 2025 and Gemini
7 CoordinateReferenceSystem sourceCRS = CRS.decode("EPSG:4326",
    true);
8 CoordinateReferenceSystem targetCRS = CRS.decode("EPSG:32612");
9
10 //Refactoring DeepSeek
11 CoordinateReferenceSystem sourceCRS = CRS.decode("EPSG:4326",
    true);

```

```

12 | CoordinateReferenceSystem targetCRS = CRS.decode("EPSG:32612");
13 | CRS.findMathTransform(targetCRS, sourceCRS);

```

Listing 5.1 – Original code and the refactorings performed by DeepSeek Coder, GPT 4.1 and Gemini 2.5 Pro for Query ID = 144"

Most of the commented code warnings (*java:S125*) were removed. However, in some cases, they were actually introduced. This occurs because, instead of deleting the unused line, GPT 4.1 and Gemini 2.5 Pro sometimes simply commented on it. Similarly, warnings such as *java:S1186* (empty method) and *java:S2094* (empty class) were introduced as a side effect of removing unused variables or attributes. As a result, methods or entire classes may become empty, thus raising these new warnings.

Some warnings were fully addressed by completely removing problematic code elements. These include unused fields (*java:S1068*), empty nested blocks such as `for` loops without a body (*java:S108*), unused private methods (*java:S1144*), unnecessary lines like `Class.forName()` calls (*java:S4925*), and empty statements (*java:S1116*).

In summary, some of the SonarLint rules on *Clean Up* category could not be corrected or led to the introduction of new warnings. This suggests that addressing these issues may require not only improvements to the messages provided by SonarLint, but also more sophisticated prompt engineering capable of guiding LLM through iterative corrections until the model generates a code without the presence of SonarLint warnings.

Finding 2) Some of the *Clean Up* rules may require an iterative prompt engineering process to fully remove the warnings, as in many cases new warnings are introduced as a consequence of resolving other types of readability warnings.

5.3.4.2 Clarify Code Intent

Table 25 presents the number of readability warnings that were corrected, not corrected, and introduced for the 23 SonarLint rules classified under the *Clarify Code Intent* category. In most cases, the warnings were corrected by the LLMs, with only 15 instances from GPT 4.1, 11 from DeepSeek Coder and 6 from Gemini 2.5 Pro still containing readability warnings. Examples include duplicated literals extracted into constants (*java:S1192*), avoid generic exceptions (*java:S112*), access to members of a static base class via derived types (*java:S3252*), missing `@Override` annotations (*java:S1161*), raw types such as `List integers = new ArrayList<>()` (*java:S3740*), and missing use of `final` keyword when static fields are public (*java:S1444*), among others.

In this process, we examined whether the LLMs actually corrected the code or simply removed the portion that raised the SonarLint warning. Surprisingly, in most cases, LLMs effectively corrected the code while preserving its functional behavior. Examples include.

Table 25 – Number of readability warnings by status (Corrected, Not Corrected, Introduced) for each SonarLint rule and LLM in the *Clarify Code Intent* category (Gem = Gemini 2.5 Pro, GPT = GPT 4.1, Deep = DeepSeek Coder).

Rule	Corrected			Not Corrected			Introduced		
	GPT	Deep	Gem	GPT	Deep	Gem	GPT	Deep	Gem
java:S1192	37	39	39	2	0	0	1	0	0
java:S112	29	26	28	0	3	1	2	1	1
java:S3252	25	25	25	0	0	0	0	0	0
java:S1161	19	20	20	1	0	0	0	0	0
java:S3740	20	20	20	0	0	0	0	0	0
java:S4719	8	8	8	0	0	0	0	0	0
java:S1444	7	7	6	0	0	1	1	0	0
java:S6213	1	3	6	5	3	0	1	0	0
java:S6353	6	6	6	0	0	0	0	0	0
java:S1141	3	3	3	0	0	0	0	0	0
java:S2387	2	2	2	0	0	0	0	0	0
java:S1700	1	1	1	1	1	1	0	0	0
java:S2440	1	1	1	1	1	1	0	0	0
java:S1117	1	1	1	0	0	0	0	0	0
java:S1165	1	1	1	0	0	0	0	0	0
java:S1171	1	1	1	0	0	0	0	0	0
java:S3010	1	0	1	0	1	0	0	0	0
java:S2209	1	1	1	0	0	0	0	0	0
java:S5843	1	0	0	0	1	1	0	0	0
java:S4977	1	1	1	0	0	0	0	0	0
java:S6035	1	1	1	0	0	0	0	0	0
java:S6397	1	1	1	0	0	0	0	0	0
java:S3358	1	1	1	0	0	0	0	0	0
Total	169	169	174	10	10	5	5	1	1

1. In rule *java:S1192*, duplicate literals such as “State changed to RUNNING” were extracted into constants, for example: `private static final String STATE_CHANGED_TO_RUNNING = "State changed to RUNNING";`.
2. In rule *java:S112*, the original Stack Overflow code threw generic exceptions such as `RuntimeException` when a field was not found. DeepSeek Coder and Gemini 2.5 Pro modified the code to throw `NoSuchFieldException`, while GPT 4.1 introduced a custom exception named `FieldNotFoundException`.
3. In rule *java:S3252*, most LLMs replaced accesses to constants like `EXIT_ON_CLOSE` through concrete classes (e.g., `JFrame`) with accesses via their declaring interface `javax.swing.WindowConstants`, as recommended.
4. In rule *java:S1161*, when the original code involved inheritance (e.g., `class Dog`

- `extends Animal`) and lacked the `@Override` annotation on overridden methods (e.g., `sound()`), all LLMs corrected the issue by adding the appropriate annotation.
5. In rule *java:S3740*, the LLMs corrected the usage of raw type by transforming code such as `List<String> list = new ArrayList()` into `List<String> list = new ArrayList<>()`, using the diamond operator.
 6. In rule *java:S1444*, variables that should be constants, such as `private static double PI_VALUE = 3.14159`, were updated to include the `final` modifier, improving readability.

In few cases, GPT 4.1 and DeepSeek Coder failed to correct specific readability warnings. For example, in rule *java:S6213*, which recommends avoiding the use of restricted identifiers such as `var` or `record`, the original Gemini 2.5 Flash generated code included a loop declaration `for (InterfaceRecord record : batch)`, where SonarLint raised a warning suggesting renaming the variable `record`. Gemini 2.5 Pro addressed the issue by renaming it `interfaceRecord`, aligning it with the class name. However, GPT 4.1 did not modify the identifier and the warning persisted. DeepSeek Coder removed the entire loop because it was empty and had no effect on logic.

Finding 3) In the *Clarify Code Intent* category, the LLMs generally resolved the warnings raised by SonarLint, including the extraction of magic literals, the addition of modifiers and annotations, among others. An interesting observation is that, in most cases, the warnings were addressed with a single prompt, without the need for iterative prompt engineering. The clear and objective descriptions provided by the SonarLint rules may have contributed to this effective removal by the LLMs.

5.3.4.3 Prefer Concise Code

Table 26 shows the number of readability warnings corrected, uncorrected, and introduced in the 20 SonarLint rules classified as *Prefer Concise Code* (19 extracted from Chapter 3 and one new rule introduced, i.e., *java:S6395*). Like the *Clarify Code Intent* category, most readability warnings were corrected by the LLMs, remaining incorrect only 18 from GPT 4.1, 7 from DeepSeek Coder, and 3 from Gemini 2.5 Pro.

1. In rule *java:S2293*, collections such as `List<String[]> rows = new ArrayList<String[]>()` were refactored to use the diamond operator, resulting in `List<String[]> rows = new ArrayList<>()`.
2. In rule *java:S1604*, anonymous classes were converted into lambda expressions by all LLMs. For example, in query ID = 247, “class override exception handler in

Table 26 – Number of readability warnings by status (Corrected, Not Corrected, Introduced) for each SonarLint rule and LLM in the *Prefer Concise Code* category (Gem = Gemini 2.5 Pro, GPT = GPT 4.1, Deep = DeepSeek Coder).

Rule	Corrected			Not Corrected			Introduced		
	GPT	Deep	Gem	GPT	Deep	Gem	GPT	Deep	Gem
java:S2293	45	46	46	1	0	0	2	1	0
java:S1604	23	24	24	1	0	0	1	1	0
java:S1602	18	18	18	0	0	0	1	0	0
java:S3012	12	14	14	2	0	0	0	0	0
java:S1612	8	8	8	0	0	0	3	3	1
java:S1488	8	8	8	0	0	0	1	0	0
java:S2129	6	6	6	0	0	0	0	0	0
java:S1066	5	5	4	0	0	1	1	1	0
java:S2093	5	5	5	0	0	0	0	0	0
java:S1905	3	4	4	1	0	0	0	0	0
java:S2140	3	4	4	1	0	0	0	0	0
java:S2147	3	3	3	0	0	0	2	0	0
java:S1126	3	2	3	0	1	0	0	0	0
java:S7158	3	3	3	0	0	0	0	0	0
java:S1125	2	2	2	0	0	0	0	0	0
java:S1611	2	2	2	0	0	0	0	0	0
java:S3626	1	1	1	0	0	0	1	0	0
java:S1858	1	1	1	0	0	0	0	0	0
java:S2130	1	1	1	0	0	0	0	0	0
java:S6395	0	0	0	0	0	0	0	0	1
Total	152	157	157	6	1	1	12	6	2

Java”, Listing 5.2 illustrates how GPT 4.1, DeepSeek Coder, and Gemini 2.5 Pro replaced an anonymous class with a lambda expression.

3. Rule *java:S1602* recommends that lambda expressions containing only a single statement should not be encapsulated within a code block. In this case, the LLMs correctly removed the opening and closing braces of the lambda bodies.
4. Rule *java:S3012* recommends avoiding the use of loops to copy arrays or lists. In this case, we analyzed whether the LLMs refactored the code using appropriate built-in methods. As shown in the Listing 5.3, for query ID = 302, “Recursion sort in Java”, Gemini 2.5 Pro and DeepSeek Coder replaced the loop with a call to `System.arraycopy()`, while GPT 4.1 did not provide an alternative.
5. Rule *java:S1612* suggests replacing lambda expressions with method references when possible. In some cases, this warning was introduced as a side effect of previous changes. Listing 5.4 illustrates an example from query ID = 29, “getimage url applet in Java”. The original code raised a warning to convert an anonymous class

to a lambda expression (*java:S1604*). After GPT 4.1 and DeepSeek Coder applied this transformation, the resulting code raised a new warning (*java:S1612*) suggesting the use of a method reference instead. Interestingly, only Gemini 2.5 Pro directly generated a method reference, without requiring a second prompt.

```

1 // Original Stack Overflow Code Snippet
2 public MyThread() {
3     setUncaughtExceptionHandler(new UncaughtExceptionHandler() {
4         @Override
5         public void uncaughtException(Thread t, Throwable e) {
6             System.out.println("[THREAD SPECIFIC] My exception
7                 handler");
8         }
9     });
10 }
11 //Gemini 2.5 Pro and DeepSeek Coder refactoring
12 public MyThread() {
13     setUncaughtExceptionHandler((t, e) -> System.out.println("[
14         THREAD SPECIFIC] My exception handler"));
15 }
16 //GPT 4.1 refactoring
17 public MyThread() {
18     setUncaughtExceptionHandler(
19         (t, e) -> System.out.println("[THREAD SPECIFIC] My
20             exception handler")
21     );
22 }

```

Listing 5.2 – Original code and the refactorings performed by DeepSeek Coder, GPT 4.1 and Gemini 2.5 Pro for Query ID = 247"

```

1 //original code, and GPT 4.1 did not changed
2 for (int i = left; i <= right; i++) {
3     temp[i] = arr[i];
4 }
5
6 //Gemini 2.5 Pro and DeepSeek Coder code
7 System.arraycopy(arr, left, temp, left, right - left + 1);

```

Listing 5.3 – Original code and the refactorings performed by DeepSeek Coder, GPT 4.1 and Gemini 2.5 Pro for Query ID = 302"

```

1 //original code in Stack Overflow
2 try {
3     SwingUtilities.invokeLater(new Runnable() { @Override
4         public void run() { createGUI(); } });
5 }
6 //GPT 4.1 and DeepSeek Coder version
7 try {
8     SwingUtilities.invokeLater(() -> createGUI());
9 }
10
11 //
12 //Gemini 2.5 Pro version
13 try {
14     SwingUtilities.invokeLater(this::createGUI);
15 }

```

Listing 5.4 – Original code and the refactorings performed by DeepSeek Coder, GPT 4.1 and Gemini 2.5 Pro for Query ID = 29"

Finding 4) In general, most of the *Prefer Concise Code* rules were successfully addressed using a single prompt. However, there are cases in which a single prompt was not sufficient to fully remove the SonarLint warnings, particularly when multiple related rules apply sequentially. For example, the combination of rules *java:S1604* (“Anonymous inner classes containing only one method should become lambdas”) and *java:S1612* (“Lambdas should be replaced with method references”) often requires additional context. This suggests that, in some cases, the input provided to the LLM should be improved to account for overlapping scenarios, in order to prevent the resolution of one warning from introducing another.

5.3.4.4 Convention

Table 27 shows the number of readability warnings that were corrected, not corrected, and introduced across the 15 SonarLint rules classified under the *Convention* category. The rules *java:S117* (variable names), *java:S116* (field names), *java:S115* (constant names), *java:S100* (method names), and *java:S101* (class names) refer to the use of Java naming conventions for identifiers. Interestingly, LLMs often suggested names that follow these conventions, such as:

1. `doc_List` to `docList` in query ID = 35

2. Robot MYROBOT to private Robot myRobot in query ID = 133

3. `public static int StackMove(int N)` to `public static int stackMove(int diskCount)` in query ID = 88

Table 27 – Number of readability warnings by status (Corrected, Not Corrected, Introduced) for each SonarLint rule and LLM in the *Convention* category (Gem = Gemini 2.5 Pro, GPT = GPT 4.1, Deep = DeepSeek Coder).

Rule	Corrected			Not Corrected			Introduced		
	GPT	Deep	Gem	GPT	Deep	Gem	GPT	Deep	Gem
java:S117	29	29	29	0	0	0	5	3	0
java:S116	21	23	23	3	1	1	0	0	0
java:S1659	21	22	22	1	0	0	0	0	0
java:S1197	12	13	13	1	0	0	0	0	0
java:S2234	0	0	4	6	6	2	0	0	0
java:S115	1	1	1	0	0	0	3	5	2
java:S100	3	3	3	0	0	0	0	0	0
java:S101	3	3	3	0	0	0	0	0	0
java:S1124	3	3	3	0	0	0	0	0	0
java:S1301	2	3	3	1	0	0	0	0	0
java:S3008	2	2	2	0	0	0	0	0	0
java:S127	1	1	2	1	1	0	0	0	0
java:S128	2	2	2	0	0	0	0	0	0
java:S131	1	2	2	1	0	0	0	0	0
java:S3973	1	1	1	0	0	0	0	0	0
Total	102	108	113	14	8	3	8	8	2

However, in some cases, the LLMs introduced variable or constant names that do not follow Java naming conventions. For example, in query ID = 88, “towers of hanoi in Java”, the original Stack Overflow code includes a static variable declared as `public static Stack<Integer>[] integer = new Stack[4]`. SonarLint raised a *java:S1444* warning, recommending that this variable be converted to a constant. Although all LLMs applied the appropriate modification by making the variable `final`, none of them renamed it according to the Java convention for constants, which requires uppercase letters with underscores (e.g., `INTEGER`).

In other rules, such as *java:S1659*, which recommends that multiple variables should not be declared on the same line, LLMs correctly refactored the code placing each variable on a separate line. Similarly, rule *java:S1197* enforces conventions for array designators. For instance, in query ID = 133, the original Stack Overflow code declares the array as `Color SCREEN[] []`, and DeepSeek refactored it to `private Color[] [] screen`, aligning with Java conventions for array declarations and identifier naming.

One particularly case is rule *java:S2234*, which recommends that “parameters should be passed in the correct order”. This warning is raised when a parameter is named for a specific position in a method signature, but a variable with the same name is passed in a different position in the method call. As illustrated in query ID = 88 (Listing 5.5), this warning appears in a recursive implementation where the order of the parameters naturally changes between recursive calls. In this case, the LLM correctly chose not to modify the parameter order despite the presence of the warning order to preserve the functional behavior of the code and avoid introducing bugs.

```

1 public static int move(int N, int a, int b, int c) {
2     if (N > 0) {
3         int numberMovesA = move(N - 1, a, c, b);
4         int d = integer[a].pop();
5         integer[c].push(d);
6         PrintStack();
7         int numberMovesB = move(N - 1, b, a, c);
8         return (numberMovesA + numberMovesB + 1);
9     }
10    return 0;
11 }

```

Listing 5.5 – Original code for Query ID = 88"

Finding 5) Although the LLM occasionally produced identifier names that violated Java naming conventions, it was generally effective in addressing naming and coding standard violations as defined by the SonarLint rules.

5.3.4.5 Improve Code Modularity

Table 28 shows the number of readability warnings corrected, uncorrected, and introduced in the 5 SonarLint rules classified as *Improve Code Modularity*. The only scenario that introduced warnings is the *java:S1450* - “Private fields only used as local variables in methods should become local variables”, because in some snippets, the LLMs created fields only to be used in a specific method, raising the readability warning.

In this category, two sonarLint rules are correlated: *java:S3776* - “Cognitive Complexity of methods should not be too high” and *java:S6541* - “Methods should not perform too many tasks (aka Brain method)”. The query ID = 133: “How do I automate mouse clicks in Java”, the Stack Overflow code snippet contains a method `void typeHuman(String letters)` with LOC = 162, cognitive complexity = 18, number of variables = 8, and nesting level = 4. This method raises both rules *java:S3776* and *java:S6541*. Both

Table 28 – Number of readability warnings by status (Corrected, Not Corrected, Introduced) for each SonarLint rule and LLM in the *Improve Code Modularity* category (Gem = Gemini 2.5 Pro, GPT = GPT 4.1, Deep = DeepSeek Coder).

Rule	Corrected			Not Corrected			Introduced		
	GPT	Deep	Gem	GPT	Deep	Gem	GPT	Deep	Gem
java:S1450	7	7	7	0	0	0	1	2	1
java:S3776	6	6	7	2	2	1	0	0	0
java:S135	3	3	4	1	1	0	0	0	0
java:S1479	0	0	1	1	1	0	0	0	0
java:S6541	0	1	1	1	0	0	0	0	0
Total	16	17	20	5	4	1	1	2	1

DeepSeek Coder and Gemini 2.5 Pro implemented a *extract method* refactoring to reduce the complexity of the method.

Finding 6) The LLMs were also capable of modifying relatively large classes (e.g., with LOC = 810) and long methods (e.g., LOC = 162), by applying modularization strategies such as method extraction. This demonstrates the potential of LLMs to perform more complex refactorings, beyond simple edits that involve a few characters.

RQ₁ Answer: While the free version of ChatGPT 4o generate code snippets with fewer readability warnings detected by SonarLint, DeepSeek Coder and Gemini 2.5 Pro are more effective at completely removing the warnings recommended by SonarLint in code snippets. In most cases, the LLMs are capable of refactoring the code to address readability warnings, including modular transformations such as *extract method* refactorings. However, some rules and their associated messages may require iterative prompt engineering to fully resolve all warnings, since the correction of one rule may lead to the introduction of new warnings from others.

5.4 Discussion of the Results

One of the initial objectives of this study was to evaluate the impact of the randomness parameter, specifically to determine the most appropriate temperature setting to generate responses, as shown in Table 29. LLMs include a response randomness parameter known as *temperature*. In general, the closer this value is to 0, the more deterministic and predictable the output becomes. In contrast, higher values tend to produce more random or creative responses. For example, according to the DeepSeek documentation (DEEPSEEK,

2025c), a temperature of 0.0 is recommended for code-related and mathematical tasks, 1.0 for data analysis, and 1.5 for creative writing or poetry.

Table 29 – Temperature settings for each LLM.

Setting Level	GPT-4.1	DeepSeek Coder	Gemini 2.5 Pro
Deterministic	0.0	0.0	0.0
Moderate	1.0	1.0	1.0
Creative	2.0	1.5	2.0

However, previous work has shown that lower temperature values often lead to better and more stable results in code refinement tasks (GUO et al., 2024). However, this assumption may not be generalized to all scenarios involving source code. For example, using LLMs for suggesting variable names may benefit from exploring a wider range of temperature settings, as variable naming often requires a degree of creativity to better align semantically with the intent of the code. This suggests that the optimal temperature can vary depending on the nature of the task, whether it requires strict adherence to structure or allows for semantic flexibility.

The results observed in this study may not fully generalize to production-ready source code. One key reason is that code snippets are typically short and didactic, often lacking the semantic complexity found in functional requirements from real-world systems. As shown in Chapter 3, many of the analyzed code snippets contain around 50 lines of code, which does not reflect the structure or behavior of production code. Therefore, the effectiveness achieved by LLMs in refactoring code snippets may differ when applied to larger, more complex production-ready codebases.

An important aspect is that the code snippets extracted in Chapter 3 were manually obtained from free versions of the models, such as ChatGPT 3.5, ChatGPT 4o, DeepSeek, and Gemini. In contrast, the refactorings presented in this chapter were performed using paid models accessed through APIs, including GPT 4.1, DeepSeek Coder, and Gemini 2.5 Pro. This distinction reflects two different use cases: free versions simulate developers manually querying an LLM through its user interface to obtain a code snippet that solves a particular problem, while paid API versions represent a potential evolution of these models as tools such as IDE plugins, which would rely on API access and the most up-to-date model available. Although DeepSeek Coder showed superior performance in code refactoring, this does not necessarily imply better results in manual code generation scenarios.

Another important point is that the evaluation focused on verifying whether the readability warnings were removed by refactoring the code, rather than simply deleting class content. No analysis was conducted to assess whether the changes introduced bugs or security vulnerabilities, in addition to readability improvements, an aspect that could be valuable to investigate. While the LLMs may have been successful in removing readability

warnings, they might also have introduced other types of issue, such as functional bugs or security flaws.

5.5 Implications for Developers and Researchers

5.5.1 Researchers

The results show that combining LLMs with linters to improve code readability can lead to well-structured refactorings, as long as the process is properly guided. A curated set of rules extracted from multiple linters such as SonarLint (SONARLINT, 2025b), PMD (PMD, 2025) and Checkstyle (CHECKSTYLE, 2025) could provide a more comprehensive and representative view of code readability.

Using a lower temperature parameter makes the LLM more deterministic and predictable, but also often literal. As shown in the analysis, when instructed to remove a specific element, the model simply removes it, often without considering that the change might raise another readability warning. This highlights the need for better prompt design to avoid excessively long refinement cycles between LLM and developers. These findings open opportunities for developing new IDE plugins that combine understanding the context of the code with well-defined static analysis rules to guide the refactoring process more effectively.

Finally, specialized code generation models, such as DeepSeek-Coder, appear particularly promising, as they outperform more general-purpose models such as GPT 4.1 and Gemini 2.5 Pro. This serves as a valuable insight for future research, highlighting the potential benefits of using domain-specific LLMs for software engineering tasks.

5.5.2 Developers and Practitioners

As shown in the analysis, many SonarLint rules were resolved with a single prompt, suggesting that its recommendations can serve as an effective guide for developers when prompting LLMs to refactor the code.

Although ChatGPT is currently the most popular LLM, the results obtained by DeepSeek and Gemini using APIs demonstrate that they are equally useful models for source code improvement tasks.

5.6 Threats to Validity

This section presents a discussion of the key limitations of this study and the measures taken to mitigate their potential impact.

5.6.1 Construct validity

A threat to the validity of construction is the dependency on the SonarLint rules to define the readability improvements performed by LLM, as previously mentioned in the threats of validity in Chapter 3. Although SonarLint provides a large set of rules to detect potential readability warnings, it may not capture all types of readability warning present in the code. As a result, some relevant aspects of code readability could have been omitted. Other tools, such as Checkstyle and PMD, could provide additional recommendations that complement those detected by SonarLint.

Another threat refers to the manual process of extracting source code from LLMs, which is prone to human error. To mitigate this, validation steps were introduced throughout each process. For example, if a code snippet contains errors that prevent its analysis, SonarLint throws an exception and stops execution. These validations were implemented to ensure that all code snippets were indeed analyzed.

5.6.2 Internal Validity

A threat to internal validity relates to the evaluation of whether LLM-generated modifications preserve the original behavior of the code. In previous work (VITALE et al., 2023), their model introduced behavioral changes in approximately 30% of the samples. Although this study attempts to detect such cases, the absence of robust functional validation (e.g., through automated test suites) limits the ability to confidently assess behavioral preservation.

5.6.3 External Validity

Finally, a threat to external validity consists in the use of isolated code snippets as input source code to the LLMs. Improving the readability of short snippets could be fundamentally different from modifying source code from real-world projects, which is often more complex and context-dependent. As a result, the effectiveness of LLM-based readability improvements in codebases remains uncertain.

5.7 Concluding Remarks

In this study, a novel approach is proposed that combines the semantic capabilities of LLMs with recommendations from static analysis tools, such as linters, to refactor 611 code snippets affected by readability warnings. The main findings of this investigation can be summarized as follows.

- The LLMs refactor the code removing readability warnings produced by SonarLint without introducing new warnings in more than 70% of the code snippets.

- ❑ DeepSeek Coder, a model specifically trained for coding-related tasks, generally outperformed the more general-purpose models Gemini 2.5 Pro and especially GPT 4.1 in automatically improving code readability based on SonarLint warnings. This provides an interesting insight that domain-specific LLMs trained for source code may yield even more promising results.
- ❑ Many SonarLint rules have clear and descriptive messages that allow LLMs to refactor the code in a single shot (i.e., with a single prompt). However, some rules may require multiple prompts or involve interdependent warnings, where fixing one rule may trigger new warnings from others. This highlights the need for prompt engineering strategies that can refine the input to the LLM in order to achieve better and more consistent results.

There are many opportunities for future studies. For example, combining metrics and rules from different linters could improve the prompts given to LLMs, allowing greater coverage of code readability aspects. The application of this combination (LLM with linters) can also evolve beyond code snippets to production ready code, allowing for more comprehensive evaluations. Additionally, future work could involve linters for other programming languages, such as Python Pylint, to assess the generalizability of the approach across ecosystems.

Related Work

This chapter presents some of the most relevant work in establishing the foundations of this thesis. The selected studies address key topics, including the assessment and improvement of code readability, code review practices, and the main approaches that support the proposed methodology.

The related work discussed in this chapter was selected from venues recognized as highly relevant by the software engineering research community. We considered top-tier international conferences such as International Conference on Software Engineering (ICSE), International Conference on Program Comprehension (ICPC), Mining Software Repositories (MSR), International Conference on Automated Software Engineering (ASE), ACM International Conference on the Foundations of Software Engineering (FSE), International Symposium on Empirical Software Engineering and Measurement (ESEM), International Conference on Software Analysis, Evolution and Reengineering (SANER), and International Conference on Software Maintenance and Evolution (ICSME), as well as established journals including Information and Software Technology (IST), Journal of Systems and Software (JSS), IEEE Transactions on Software Engineering (TSE), and Empirical Software Engineering (EMSE). In addition, we also examined research published in national venues, such as the Brazilian Symposium on Software Engineering (SBES), which often reflects the local research context. The search focused on studies retrieved using keywords directly related to our research scope, such as “readability”, “refactoring”, “code snippet”, LLM, PR, and ASAT.

6.1 Understanding Developer Practices

Understanding how developers perceive and interact with source code is a key aspect of empirical software engineering. Numerous studies have investigated developer behavior, including their motivations for performing code changes, perceptions of design quality, and reactions to specific code properties. This section presents related work reporting such findings, providing information on the factors that drive developer decisions and

practices.

6.1.1 Developers Perceptions

Some studies aim to understand the perception of developers about specific topics in software engineering, which are often influenced by their individual interpretations and definitions of each subject. For example, previous research has presented developers' perceptions based on their opinions about code smells, architectural roles, or class co-change patterns, typically grounded in source code that they themselves authored. These perceptions may at times diverge from the theoretical principles established in the literature.

Oliveira et al. (2022) investigated the effectiveness of Machine Learning (ML) techniques in detecting developer-sensitive code smells, that is, code smells whose identification may vary depending on the individual perception of developers. In their study, 63 experienced Java developers were asked to classify 1,800 code fragments in the presence of ten well-known code smells, such as the Long Method and the God Class. In particular, the Kappa agreement coefficient (COHEN, 1960) indicated only slight to fair agreement among developers, confirming the idea that perception of code quality can be highly individual (BOFF; KAUFMAN; THOMAS, 1986).

Palomba et al. (2014) conducted a study to understand the perception of developers of code smells (FOWLER, 1999), specifically the extent to which developers recognize code smells as serious design problems and the potential gap between theory (which is believed to be problematic) and practice (which developers actually perceive as problematic). In the experiment, developers were shown classes containing code smells and asked if they identified any design or implementation issues. Their responses were analyzed to assess how they interpret potential design problems in the code.

In another study, Aniche, Gerosa and Treude (2016) examined the developers perception of aspects of object-oriented design related to architectural roles, and whether these perceptions aligned with source code metric analysis. Through a survey, developers were asked to express opinions such as *"Controllers are more coupled than Services"* or *"Entities are more complex than Repositories"*. These perceptions were then compared with the actual code metrics extracted from the corresponding classes.

Although the taxonomy proposed in this thesis is derived from developers' perceptions, specifically, how they objectively interpret and express code readability, this thesis does not aim to classify those perceptions or compare them against theoretical definitions in the literature. It is not intended to analyze divergences in perception among developers themselves.

6.1.2 Developers Motivations

Other studies focus on the motivations of developers, defined as the reasons behind their actions during software development. These motivations reflect how developers interpret and prioritize various quality attributes of software in specific contexts. For example, motivations such as improving maintainability, preventing bugs, or code readability often lead to observable actions such as performing refactoring operations. In this sense, developers' motivations act as a bridge between their subjective perceptions of code quality and the concrete changes they apply to the codebase.

Pantiuchina et al. (2020) manually analyzed 551 merged PRs and found that 468 of them included improvements in code readability and understandability, including common improvements such as variable renaming and code cleanup. Other motivations for refactoring included bug prevention, preparation for future changes, improvement of test code quality, and improvements in overall code design. Their study influenced this thesis by showing that developers actively perform code readability improvements in repositories through refactoring operations.

In another study, Silva, Tsantalis and Valente (2016) used thematic analysis to explore developers' motivations to perform refactorings. After monitoring 748 Java repositories for 61 days, the authors detected refactoring operations and contacted developers by email to inquire about their motivations. The responses collected were analyzed using thematic analysis (CRUZES; DYBA, 2011), resulting in the identification of 44 distinct motivations in 12 refactoring types.

This thesis differs from previous studies because it does not aim to propose a taxonomy of developer motivations. Instead, it focuses on the improvements that were made as a consequence of those motivations. The taxonomy proposed in this thesis is more specific, as it focuses on cases where the motivation is explicitly to improve code readability, as one of the main reasons for performing refactoring operations previously identified by Pantiuchina et al. (2020).

6.1.3 Developers Improvements

While perceptions refer to how developers interpret specific aspects of the code, motivations represent the underlying reasons that lead them to take action, such as improving code readability and design, preventing bugs, etc. (PANTIUCHINA et al., 2020). In consequence, actions correspond to the concrete changes applied to the code, reflecting what was actually done by the developer as a result of those motivations and perceptions.

Oliveira et al. (2025) analyzed 2,401 code review comments from Java projects and proposed a taxonomy comprising eight categories of understandability smells, based on 323 accepted review suggestions. Their findings showed that 83.9% of the suggestions related to understandability were accepted and merged into the codebase. Furthermore,

they observed that four widely used Java linters failed to detect most of these issues. Although some problems could potentially be addressed by adding new linter rules, others, such as those involving documentation clarity, may require natural language understanding, a gap that could be addressed by LLMs, as proposed in this study. Unlike their work, the taxonomy proposed in this thesis includes examples from the Python programming language and explores differences in languages from distinct paradigms. Furthermore, this thesis also focuses on merged PRs explicitly aimed at improving code readability, thus complementing the analysis based on code review comments, defining higher-order themes by grouping semantically unified motivations (BRADLEY; CURRY; DEVERS, 2007).

6.2 Code Readability Models

Although this thesis does not propose an entirely new model for code readability, existing approaches, their applications, and limitations should be discussed as related work, as they inspire the proposal of a new taxonomy focused on code readability improvements performed by developers in real-world repositories.

Previous studies have focused on automatic assessment of code readability (BUSE; WEIMER, 2010; POSNETT; HINDLE; DEVANBU, 2011; DORN, 2012; SCALABRINO et al., 2016; SCALABRINO et al., 2018). These approaches typically rely on machine learning techniques that define a set of features extracted from source code and train a binary classifier to distinguish between readable and unreadable code. To train such classifiers, these approaches require large datasets that contain human judgments about code readability. These datasets follow a similar construction process: the authors selected a set of code snippets, asked several developers to rate each snippet’s readability using a Likert scale, and then classified each snippet as either readable or unreadable based on a threshold. All of these studies also reported metrics for agreement among evaluators to validate consistency.

6.2.1 Readability Model Applications

Several code search engines have been proposed that incorporate readability models as part of their ranking methodology. Hora (2021b) investigated how Google, a general-purpose web search engine, ranks code snippets in terms of readability and reusability features. The findings indicate that readable and reusable code snippets are not necessarily top-ranked. Instead, factors such as didactic quality or the presence of multiple snippets on a single page tend to influence ranking more strongly.

In another study, Hora (2021c) developed the API Sonar tool, which extracts code snippets from 100 Java APIs on GitHub to generate collections of API usage examples. Readability is used as a criterion to rank the most readable snippets. Moreno et al. (2015)

introduced the Muse approach, which ranks code snippets by computing an overall score based on readability and reusability features.

These works demonstrate that readability models are frequently used in practice, such as code search and recommendation systems. It also highlights the importance of improving the accuracy and reliability of readability assessments in these models. This reinforces the relevance of improving existing readability models as well as developing new ones.

6.2.2 Readability Models Limitations

Some studies have attempted to assess whether state-of-the-art readability models are capable of capturing readability improvements explicitly mentioned by developers in PR descriptions or commit messages. For example, Pantiuchina, Lanza and Bavota (2018) analyzed 1,282 valid commits, evaluating changes in quality metrics such as cohesion, coupling, complexity, and readability before and after each commit. Their findings indicate that these metrics did not capture the subjective interpretation of code quality by developers, suggesting that quantitative indicators should be complemented by developer feedback.

Fakhoury et al. (2019) investigated readability improvements in 548 commits and 2,323 classes by manually analyzing the commit messages and source code. Their findings confirm those of previous studies: readability metrics often fail to capture the readability improvements performed by developers in commits. Moreover, they observed that complexity metrics such as *McCabe* (MCCABE, 1976) tend to decrease when readability improvements are made, and that refactoring operations such as the Extract Method frequently occur in readability-related commits.

Scalabrino et al. (2017) conducted a comprehensive evaluation of 121 different metrics and found that none of them showed a significant correlation with code understandability or even with code readability itself. Oliveira et al. (2020) reported that much of the research on code readability has been evaluated through human-centered studies, often relying on surveys and questionnaires. These studies provide valuable insight by revealing the discrepancies between what is reported and what actually occurs in real-world software repositories.

Piantadosi et al. (2020) introduced a model to analyze the evolution of code readability in GitHub classes. Their findings show that when a class is initially created with low readability, it tends to remain unreadable over time. Moreover, large commits or changes involving multiple classes often lead to a decrease in code readability. Importantly, the study also highlights that Scalabrino's model exhibits a reduced precision when evaluating incremental changes. This limitation occurs because the tool was designed to assess isolated code snapshots, such as individual snippets, rather than the evolution of the Java file over time.

Given that prior studies have reported that existing metrics are not effectively able to capture what developers actually consider code readability, this limitation motivated the proposal of a new taxonomy in this thesis. The goal is to better represent the types of changes performed by developers when readability is explicitly stated as a motivation in GitHub PRs, providing an comprehensive perspective on code readability improvements.

6.3 Static Analysis Tools

Since this thesis relies on the use of ASATs such as linters, mainly to identify code elements lacking readability, it is essential to include a review of related literature that examines both the effectiveness of these tools in detecting readability-related issues and the practical contexts in which they have been applied.

Lenarduzzi et al. (2023) conducted a large-scale investigation to assess which types of code quality issues can be effectively detected by six popular ASATs: Better Code Hub, CheckStyle, Coverity Scan, FindBugs, PMD, and SonarQube in 47 projects. Their study resulted in a taxonomy of warnings produced by these ASATs, including naming and formatting, exception handling, code complexity, and others. The tools demonstrated low agreement, with different ASATs often detecting distinct issues even when analyzing the same source code. Furthermore, the precision of the tools varied considerably, ranging from 18% to 86%, with many false positives and warnings that were not useful in real-world scenarios. In this thesis, we also chose to discard some rules, such as the rule discouraging the use of `System.out`, especially in code snippets, where such usage is often acceptable and serves a didactic purpose.

Romano et al. (2022) conducted two controlled experiments involving 92 participants to assess the impact of using the SonarLint tool during Test Driven Development (TDD). Their findings show that the SonarLint significantly reduced code smells, lowered estimated technical debt, and improved code understandability. However, participants also reported that TDD became more cognitively demanding with the tool enabled.

Marcilio et al. (2019) investigated whether and how developers actually fix static analysis violations by analyzing issue data and surveying practitioners. Although most developers recognized the relevance of these tools in improving overall software quality, the study found that only 13% of the reported issues had been effectively resolved in the analyzed systems.

6.3.1 Static Analysis Tools on Empirical Studies

Although some studies have shown that ASATs may produce false negatives and is not always actively used by developers, it is common to find related work that relies on these tools to extract valid metrics for empirical research. This connects directly

with the investigation proposed in this thesis, which uses SonarLint to extract potential readability-related violations.

Calixto, Araújo and Alves (2024) analyzed Technical Debt (TD) in 2,035 merged PRs from 12 Apache Java projects using the SonarQube tool. Their findings show that 96.26% of the PRs exhibited some form of TD. While some PRs removed TD and others increased it, approximately half remained unchanged. The connection between this study and the present thesis lies in the shared reliance on Sonar-based tools to extract rule violations. However, while Calixto, Araújo and Alves (2024) adopted a general perspective on code smells, this thesis takes a more specific approach, focusing specifically on code smells that impact readability.

6.3.2 Static Analysis tools combined with Neural Models

This subsection presents some models that inspired this thesis to propose an automated refactoring of warnings raised by SonarLint.

Loriot, Madeiral and Monperrus (2022) introduced Styler, an approach that addresses readability-related improvements such as code styling. Their method collects issues from Checkstyle ASAT tool for each class and trains a neural model to apply the linter's recommendations automatically. This thesis has a similar objective to automatically improve readability, but focuses on SonarLint's readability-specific rules and evaluates how effectively each LLM (ChatGPT, DeepSeek and Gemini) can refactor code snippets according to these suggestions.

Jaoua, Sghaier and Sahraoui (2025) proposed a hybrid approach that combines linters with LLM to improve the quality of the code review process. Our approach differs by selecting a different linter tool and focusing specifically on code readability, rather than improving the precision of review comments.

6.4 Mining GitHub Pull Requests

Since this thesis relies on GitHub PRs to collect source code from real-world repositories, this section presents related studies that adopt a similar approach.

Tufano et al. (2024) proposed a taxonomy that describes the automated tasks developers request from ChatGPT, categorizing 467 instances (165 commits, 159 pull requests, and 143 issues) that explicitly mention the term "*ChatGPT*". To investigate how developers use ChatGPT in real-world scenarios, they extracted references from GitHub PRs, commits, and code reviews. Similarly, the DevGPT repository (XIAO et al., 2024) was constructed based on developer interactions on GitHub that involve the use of ChatGPT. Chouchen et al. (2024) and Hao et al. (2024) both proposed taxonomies derived from shared ChatGPT conversations using the DevGPT dataset. The former analyzed

243 ChatGPT prompts embedded in PRs, while the latter examined 210 GitHub pull requests and 370 issues.

Coelho et al. (2021) analyzed refactoring-inducing PRs, that is, cases in which refactoring operations were introduced in commits made between the initial PR commit and the final merge. These intermediate commits often reflect interactions between developers and reviewers. The authors found that 30,2% of the 1,845 analyzed PRs included refactoring-inducing operations, and such PRs tend to exhibit distinct characteristics, such as longer time to merge, stronger reviewer involvement, more extensive discussions (e.g., PR comments) and a higher number of file changes. Although this thesis also leverages merged PRs, the focus differs: rather than identifying refactoring-inducing operations, our aim is to collect and classify types of code readability improvements explicitly addressed by developers in real-world PR workflows.

6.5 Large Language Models in Software Engineering Research

Although LLMs are relatively recent, the evaluation of these models has already attracted considerable attention from SE researchers. For example, Nam et al. (2024) developed an IDE plugin for VS Code, called GILT, to investigate whether integrating an LLM into the development environment through a conversational UI could improve code understanding, particularly in unfamiliar domains such as the Open3D (3D rendering) library. The study involved 32 participants who performed tasks using the GILT plugin or a traditional web search. The results showed a statistically significant improvement in task completion rate when using the plugin, although no significant differences were observed in task completion time. A key finding from the study is that the quality of the prompts is essential to the effective use of such tools.

Guo et al. (2024) conducted an empirical study to assess the effectiveness of ChatGPT in automating code refinement based on real code review scenarios. Their results indicate that parameter configuration, particularly temperature, significantly affects performance, with lower temperatures generally leading to more consistent outputs. Prompts with concise and well-defined scenarios lead to better results, and ChatGPT performs best when reviews include concrete and high-quality suggestions. The model shows strong performance in code refactoring tasks, but is less effective in refining documentation and functional aspects. Their work influenced this thesis in terms of parameter configuration, such as prompt design and temperature settings, mainly because of the same focus on use LLMs to perform refactoring operations.

Nguyen and Nadi (2022) conducted an evaluation of the precision and understandability of the code suggestions made by GitHub Copilot. Their evaluation of understandability used the cognitive complexity metric (CAMPBELL, 2018), which correlates with certain

aspects of understandability (BARÓN; WYRICH; WAGNER, 2020). Their findings concluded that the code snippets in GitHub Copilot had low complexity, attributable to the limited number of control flows. In contrast, this thesis uses 77 readability metrics to evaluate code snippets, also including cognitive complexity (SONARLINT, 2025m).

Madi (2023) focused on analyzing the readability of GitHub Copilot code snippets, compared to code written by human programmers. The evaluation of readability in his work incorporated a combination of metrics such as cyclomatic complexity, Microsoft Visual Studio Maintainability Index, and others, along with human aspects such as tracking eye movement to examine token-level Areas of Interest (AOI). His findings indicated that GitHub Copilot demonstrated similar complexity and readability to code authored entirely by human programmers, emphasizing the importance of reading and inspecting model-generated code to avoid unwarranted confidence. In contrast, our study compares multiple LLMs and introduces a method to automatically refactor code snippets by leveraging linter suggestions as input prompts to the LLMs.

6.5.1 LLM as an Annotator

Recently, several Software Engineering researches proposed the *LLM Guidelines for SE*, i.e., a draft of community guidelines for empirical studies in software engineering involving LLMs (BALTES et al., 2024). These guidelines include discussions around the use of LLMs as a tool for software engineering research, such as using LLMs as annotators, judges, synthesis, subjects, and others.

Based on the mentioned guidelines, the use of LLM as an annotator can rapidly and cost-effectively label qualitative data, potentially replacing or augmenting human annotators. Studies show that LLMs can reduce costs by 50-96% and complete tasks in days versus weeks, often with higher accuracy and agreement between codes than human crowd workers. Their agreement with each other often correlates with the agreement of human model, suggesting suitability for tasks where humans agree well.

However, challenges include reliability issues, sensitivity to prompt variations, and biases, making them less reliable for high-stakes tasks or those requiring complex contextual understanding. Although the pooling of multiple LLMs can improve reliability, it increases computational cost. Therefore, their optimal use might be to increase human efforts rather than completely replace them, which requires careful verification of their output. For this reason, in Chapter 4 of this work, ChatGPT was used as an annotator to complement the human perspective, instead of replacing them.

As an example, Ahmed et al. (2025) investigated the feasibility of using LLMs to replace or complement human annotators in software engineering evaluation tasks. By applying six LLMs to ten annotation tasks on five datasets, ranging from code summarization and semantic similarity to static analysis warnings, the authors found that in several cases, human-model agreement was comparable to human-human agreement.

However, their study considered only discrete LLM responses (multiple choice). In the study proposed in Chapter 4, ChatGPT 4.1 received a set of 58 labels for Java and 61 for Python, but the prompt explicitly mentioned that the LLM could introduce new labels if any of the provided labels did not describe the code readability improvement performed. This design decision reinforces the choice to use the LLM as a complementary annotator, acknowledging both the risk of hallucinated output and the possibility that LLMs may require even more contextual information than humans to accurately annotate certain samples.

6.6 Synthesis

Table 30 summarizes the main findings of the literature review and compares them with the focus of this thesis.

Table 30 – Comparative summary of related work and this thesis.

Author(s), Year	Scope	Limitations	Improvement Proposed
Scalabrino et al. (2018)	Model for classifying code readability	Lower precision for incremental changes	Identifies features that developers consider readable in real-world scenarios
Roy et al. (2020)	Classification of readability on incremental changes	High false-positive rate	Identifies features that developers consider readable in real-world scenarios
Nguyen and Nadi (2022)	Evaluation of the precision and understandability of code suggestions	Limited to GitHub Copilot	Assessment of the readability of code generated by three LLMs
Loriot, Madeiral and Monperrus (2022)	Combines an ASAT with a neural model	Restricted to Checkstyle rules (mostly formatting)	Model that combines SonarLint recommendations with LLMs
Vitale et al. (2023)	Automatic code readability improvement	Covers only specific improvements	Model that combines SonarLint recommendations with LLMs
Oliveira et al. (2025)	Taxonomy of understandability smells in real-world repositories	Restricted to Java repositories	Extends to Python, exploring cross-paradigm differences

Conclusions

In this chapter, the main findings and future research are discussed. Therefore, the objective of this chapter is to unify these insights and present a consolidated view of the work as a whole.

Firstly, this study evaluated the code snippets generated by LLMs compared to the human-written code snippets extracted from Stack Overflow. Although both sources produced readable code in approximately 68% to 84% of the cases, the code generated by LLMs contained significantly fewer readability warnings, particularly regarding established patterns such as coding style and naming conventions. Among the models evaluated, ChatGPT was the most prominent, producing code with significantly fewer LOC than Gemini and significantly fewer readability warnings than DeepSeek. Overall, there were no substantial differences between the 2023 and 2025 versions of the ChatGPT models, although the 2025 version produced slightly fewer readability warnings.

Secondly, by analyzing the types of code readability improvements made by developers in real-world GitHub projects, a taxonomy of 56 documented improvement types (12 exclusive for Python, 18 exclusive for Java and 26 shared between them), grouped into six categories, was extracted from 893 instances across 534 PRs in 340 repositories, involving a total of 1,036 developers responsible for opening, merging, and reviewing PRs. In both languages, the most prominent category was Clarify Code Intent, which included numerous cases in which developers improved the naming of variables, methods, and fields. However, while Python presented a higher number of instances involving documentation enhancements (e.g., comments) and formatting aligned with Python-specific guidelines, Java showed a higher frequency of clean-up actions and removal of redundant code. This analysis also reinforces the value of using ChatGPT as a complementary annotator: in the initial run, the human-model agreement exceeded 70% of the instances, and after refining the label set and adjusting the parameters, the agreement increased to approximately 90%, thus improving the reliability of the open coding process.

Finally, since LLMs can generate code with significantly fewer readability warnings, an additional evaluation was performed to verify whether they are also capable of fixing

such warnings when explicitly requested. In this experiment, 611 code snippets containing readability warnings detected by the SonarLint tool were submitted to ChatGPT, DeepSeek, and Gemini, asking the models to refactor the code and improve its readability by removing the warnings. In this scenario, the three models corrected the readability warnings between 70% and 79% of the code snippets without introducing newer warnings into them.

Interestingly, although DeepSeek had previously generated more readability warnings than ChatGPT, in this context it was able to remove a significantly higher number of warnings. It is important to note that the original code samples from DeepSeek were generated using the free version (V3) with manual prompts, as a typical developer would use. In contrast, the refactorings were performed using the DeepSeek Coder model through the API, a version specifically designed for code generation.

This contrast gives us an important insight: In Chapter 3, both ChatGPT 2023 and 2025 outperformed Gemini and DeepSeek in generating readable code using publicly available interfaces. In Chapter 5, however, when the refactorings were performed through APIs, representing a more automated support scenario, Gemini and DeepSeek performed better than ChatGPT. This difference was intentional. Chapter 3 aimed to simulate how a developer might use free LLM tools to get help with coding. Chapter 5, on the other hand, explored how automated tools, integrated through APIs, can help developers automatically refactor code.

This work has shown promising perspectives regarding the use of LLMs, both for generating code to solve programming tasks in Chapter 3 and for refining existing code based on SonarLint recommendations in Chapter 5. However, as demonstrated throughout the study, LLMs should be guided by properly designed prompts, and the messages produced by the linters with adequate refinement can be used as guidance.

Although this work did not focus on evaluating LLMs using Python, mainly because our set of questions submitted to CROKAGE was limited to the Java ecosystem, the inclusion of Python in Chapter 4 was useful to illustrate that a subset of readability issues is common among developers in different ecosystems. This suggests the possibility of defining a base set of readability rules that is language-independent. However, as shown, Python, like any other language, also has its own specific characteristics that should be properly addressed.

Although the study did not conduct a systematic evaluation to verify whether the linter rules are consistently aligned with human perspectives, Chapter 4 highlighted that a subset of the readability improvements performed by Java developers are actually covered by existing linter rules. However, these rules have limitations. For example, in naming suggestions, linters tend to enforce naming conventions rather than proposing semantically meaningful names. This reinforces the importance of prompt design: while linter-based rules can serve as a useful foundation, they should be complemented with additional rules

to better guide LLMs toward more suitable refinements.

Summarizing the main insights of this thesis.

- ❑ LLMs can generate code with significantly fewer readability warnings than human-written code from Q&A platforms such as Stack Overflow.
- ❑ Although there is a common set of readability improvement types between languages, Java and Python exhibit many language-specific characteristics.
- ❑ Some readability improvements can be detected by linters, but others, such as naming suggestions, are highly dependent on semantics.
- ❑ While LLMs proved to be useful as annotators, code generators, and for refactoring tasks, they are heavily dependent on well-crafted prompts and rich contextual information.

7.1 Future Directions

As mentioned in earlier chapters, one future direction for this work involves the advancement of the prompt used with LLMs in Chapter 5 toward the development of a prompt engineering model specifically focused on code readability. This model would integrate assessments from different linters, recommendations derived from real-world readability improvements made by developers (as presented in Chapter 4), and established standards and conventions in languages such as Python and Java. In this way, developers could receive feedback and automated suggestions grounded in a richer set of readability-oriented metrics, ultimately supporting more effective code refinement.

Beyond a simple combination of linter rules, the proposed model should also consider semantic aspects that are generally not covered by linters, such as recommending variable names that go beyond syntactic conventions, correcting typos, and refining comments. An additional research avenue would be to evaluate this model as a baseline for comparison with emerging LLM-based agents that aim to automatically manage and modify the source code in GitHub repositories.

Another direction for future research is to expand the analysis of code snippets, considering not only readability and semantic adherence, as discussed in Chapter 3, but also correctness, that is, whether the snippet effectively addresses the intent of the query. This extension would ensure that comparisons are made between code examples that are properly aligned with the original prompt. Furthermore, future work could examine not only the code recommendations generated by LLMs but also contrast the accompanying explanations with those written by developers on platforms such as Stack Overflow. This expanded analysis would provide more comprehensive insight into the quality and effec-

tiveness of the explanations generated by LLMs and how they compare to the explanations authored by humans.

Finally, although Chapter 4 raised a valid concern about ensuring that the sampling did not include false positives, this type of research often overlooks potential false negatives that could also be systematically explored. A promising direction for future work would be not only to design a more comprehensive set of keywords tailored to each search criterion involving code refactoring in general, but also to expand the use of LLMs as annotators. In this context, LLMs could be used further as judges, automatically filtering large-scale samples of PRs based on predefined criteria.

Bibliography

AGGARWAL, K.; SINGH, Y.; CHHABRA, J. An integrated measure of software maintainability. In: **Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.02CH37318)**. [s.n.], 2002. p. 235–241. Available at: <<https://doi.org/10.1109/RAMS.2002.981648>>.

AHMED, T. et al. Can llms replace manual annotation of software engineering artifacts? In: **Proceedings of the 22nd International Conference on Mining Software Repositories (MSR)**. [s.n.], 2025. Available at: <<https://doi.org/10.1109/MSR66628.2025.00086>>.

ALEXANDRU, C. V. et al. On the usage of pythonic idioms. In: **Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software**. New York, NY, USA: Association for Computing Machinery, 2018. (Onward! 2018), p. 1–11. ISBN 9781450360319. Available at: <<https://doi.org/10.1145/3276954.3276960>>.

ALLAMANIS, M. et al. Learning natural coding conventions. In: **Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2014. (FSE 2014), p. 281–293. ISBN 9781450330565. Available at: <<https://doi.org/10.1145/2635868.2635883>>.

ANICHE, M.; GEROSA, M. A.; TREUDE, C. Developers' perceptions on object-oriented design and architectural roles. In: **Proceedings of the 30th Brazilian Symposium on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2016. (SBES '16), p. 63–72. ISBN 9781450342018. Available at: <<https://doi.org/10.1145/2973839.2973846>>.

ANQUETIL, N. et al. What do developers consider magic literals? a smalltalk perspective. **Inf. Softw. Technol.**, Butterworth-Heinemann, USA, v. 149, n. C, sep 2022. ISSN 0950-5849. Available at: <<https://doi.org/10.1016/j.infsof.2022.106942>>.

APACHE. **Apache GitHub Repository**. 2025. <<https://github.com/apache>>. Accessed: 2025-02-20.

ATHANASIOU, D. et al. Test code quality and its relation to issue handling performance. **IEEE Transactions on Software Engineering**, v. 40, n. 11, p. 1100–1125, 2014. Available at: <<https://doi.org/10.1109/TSE.2014.2342227>>.

BACCHELLI, A.; BIRD, C. Expectations, outcomes, and challenges of modern code review. In: **Proceedings of the 2013 International Conference on Software Engineering**. IEEE Press, 2013. (ICSE '13), p. 712–721. ISBN 9781467330763. Available at: <<https://doi.org/10.1109/ICSE.2013.6606617>>.

BALTES, S. et al. **Guidelines for Empirical Studies in Software Engineering involving Large Language Models**. 2025. Available at: <<https://arxiv.org/abs/2508.15503>>.

_____. **LLM Guidelines for Software Engineering Research**. 2024. <<https://llm-guidelines.org/study-types/#llms-as-annotators>>. Accessed: 2025-04-01.

BARÓN, M. M.; WYRICH, M.; WAGNER, S. An empirical validation of cognitive complexity as a measure of source code understandability. In: **Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. New York, NY, USA: Association for Computing Machinery, 2020. (ESEM '20). ISBN 9781450375801. Available at: <<https://doi.org/10.1145/3382494.3410636>>.

BECK, F.; DIEHL, S. On the congruence of modularity and code coupling. In: **Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2011. (ESEC/FSE '11), p. 354–364. ISBN 9781450304436. Available at: <<https://doi.org/10.1145/2025113.2025162>>.

BELLER, M. et al. Analyzing the state of static analysis: A large-scale evaluation in open source software. In: **2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [s.n.], 2016. v. 1, p. 470–481. Available at: <<https://doi.org/10.1109/SANER.2016.105>>.

BENESTAD, H. C.; ANDA, B.; ARISHOLM, E. Understanding software maintenance and evolution by analyzing individual changes: a literature review. **J. Softw. Maint. Evol.**, John Wiley & Sons, Inc., USA, v. 21, n. 6, p. 349–378, nov. 2009. ISSN 1532-060X. Available at: <<https://doi.org/10.1002/smr.412>>.

BENNETT, K. H.; RAJLICH, V. T. Software maintenance and evolution: a roadmap. In: **Proceedings of the Conference on The Future of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2000. (ICSE '00), p. 73–87. ISBN 1581132530. Available at: <<https://doi.org/10.1145/336512.336534>>.

BINKLEY, D. W. et al. Increasing diversity: Natural language measures for software fault prediction. **J. Syst. Softw.**, v. 82, p. 1793–1803, 2009. Available at: <<https://doi.org/10.1016/j.jss.2009.06.036>>.

BOEHM, B. W.; BROWN, J. R.; LIPOW, M. Quantitative evaluation of software quality. In: **Proceedings of the 2nd International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society Press, 1976. (ICSE '76), p. 592–605. Available at: <<https://dl.acm.org/doi/10.5555/800253.807736>>.

BOFF, K. R.; KAUFMAN, L.; THOMAS, J. P. (Ed.). **Handbook of Perception and Human Performance: Volume I. Sensory Processes and Perception**. New York: Wiley-Interscience, 1986. ISBN 978-0-471-80458-9.

- BÖRSTLER, J.; CASPERSEN, M.; NORDSTRÖM, M. Beauty and the beast: on the readability of object-oriented example programs. **Software Quality Journal**, v. 24, 02 2015. Available at: <<https://doi.org/10.1007/s11219-015-9267-5>>.
- BRADLEY, E. H.; CURRY, L. A.; DEVERS, K. J. Qualitative data analysis for health services research: developing taxonomy, themes, and theory. **Health services research**, v. 42 4, p. 1758–72, 2007. Available at: <<https://doi.org/10.1111/j.1475-6773.2006.00684.x>>.
- BREIVOLD, H. P.; CRNKOVIC, I.; LARSSON, M. A systematic review of software architecture evolution research. **Inf. Softw. Technol.**, Butterworth-Heinemann, USA, v. 54, n. 1, p. 16–40, jan. 2012. ISSN 0950-5849. Available at: <<https://doi.org/10.1016/j.infsof.2011.06.002>>.
- BROOKS JR, F. No silver bullet essence and accidents of software engineering. **IEEE Computer**, v. 20, p. 10–19, 04 1987. Available at: <<https://doi.org/10.1109/MC.1987.1663532>>.
- BUSE, R. P.; WEIMER, W. R. Learning a metric for code readability. **IEEE Transactions on Software Engineering**, v. 36, n. 4, p. 546–558, 2010. Available at: <<https://doi.org/10.1109/TSE.2009.70>>.
- BUSJAHN, T. et al. Eye movements in code reading: Relaxing the linear order. In: **2015 IEEE 23rd International Conference on Program Comprehension**. [s.n.], 2015. p. 255–265. Available at: <<https://doi.org/10.1109/ICPC.2015.36>>.
- CALIXTO, F.; ARAÚJO, E.; ALVES, E. How does technical debt evolve within pull requests? an empirical study with apache projects. In: **Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software**. Porto Alegre, RS, Brasil: SBC, 2024. p. 212–223. ISSN 2833-0633. Available at: <<https://doi.org/10.5753/sbes.2024.3368>>.
- CAMPBELL, G. A. Cognitive complexity — an overview and evaluation. In: **2018 IEEE/ACM International Conference on Technical Debt (TechDebt)**. [s.n.], 2018. p. 57–58. Available at: <<https://doi.org/10.1145/3194164.3194186>>.
- CAO, K. et al. Automated query reformulation for efficient search based on query logs from stack overflow. In: . [s.n.], 2021. p. 1273–1285. Available at: <<https://doi.org/10.1109/ICSE43902.2021.00116>>.
- CASTOR, F. Identifying confusing code in swift programs. In: **6th Workshop on Software Visualization, Evolution and Maintenance (VEM)**. [s.n.], 2018. Available at: <<https://vem2018.github.io/program.html>>.
- CHACON, S.; STRAUB, B. **Pro Git: 2nd edition**. Apress, 2014. Available at: <<https://git-scm.com/book/en/v2>>.
- CHATGPT. **ChatGPT Web Page**. 2025. <<https://chatgpt.com/>>. Accessed: 2025-02-20.
- CHECKSTYLE. **CheckStyle Web Page**. 2025. <<https://checkstyle.sourceforge.io/>>. Accessed: 2025-02-20.

- CHOUCHEN, M. et al. **How Do Software Developers Use ChatGPT? An Exploratory Study on GitHub Pull Requests**. 2024. Available at: <<https://doi.org/10.1145/3643991.3645084>>.
- COCHRAN, W. G. The comparison of percentages in matched samples. **Biometrika**, Oxford University Press, v. 37, n. 3/4, p. 256–266, 1950. Available at: <<https://doi.org/10.2307/2332378>>.
- COELHO, F. et al. An empirical study on refactoring-inducing pull requests. **Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**, 2021. Available at: <<https://doi.org/10.1145/3475716.3475785>>.
- COHEN, J. A coefficient of agreement for nominal scales. **Educational and Psychological Measurement**, v. 20, p. 37 – 46, 1960. Available at: <<https://doi.org/10.1177/00131644600200010>>.
- COSTA, J. A. S. da et al. Seeing confusion through a new lens: on the impact of atoms of confusion on novices' code comprehension. **Empirical Software Engineering**, v. 28, n. 4, p. 81, 2023. ISSN 1573-7616. Available at: <<https://doi.org/10.1007/s10664-023-10311-0>>.
- CROKAGE. **Crokage Tool Web Page**. 2025. <<http://isel.ufu.br:9000/>>. Accessed: 2025-02-20.
- CRUZES, D. S.; DYBA, T. Recommended steps for thematic synthesis in software engineering. In: **2011 International Symposium on Empirical Software Engineering and Measurement**. [s.n.], 2011. p. 275–284. Available at: <<https://doi.org/10.1109/ESEM.2011.36>>.
- DABIC, O.; AGHAJANI, E.; BAVOTA, G. Sampling projects in github for msr studies. **2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)**, p. 560–564, 2021. Available at: <<https://doi.org/10.1109/MSR52588.2021.00074>>.
- DANTAS, C. C.; ROCHA, A. M.; MAIA, M. A. How do developers improve code readability? an empirical study of pull requests. In: **2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. Los Alamitos, CA, USA: IEEE Computer Society, 2023. p. 110–122. Available at: <<https://doi.org/10.1109/ICSME58846.2023.00022>>.
- DANTAS, C. E.; ROCHA, A. M.; MAIA, M. A. Assessing the readability of chatgpt code snippet recommendations: A comparative study. In: **Proceedings of the XXXVII Brazilian Symposium on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2023. (SBES '23), p. 283–292. ISBN 9798400707872. Available at: <<https://doi.org/10.1145/3613372.3613413>>.
- DANTAS, C. E. C.; MAIA, M. A. **Code Readability Assessment and Improvement: Human and Automated Perspectives**. [S.l.]: GitHub, 2025. <<https://github.com/carloseduardoxp/thesis>>.
- DEEPSEEK. **Deep Seek Coder Model**. 2025. <<https://deepseekcoder.github.io/>>. Accessed on June 10, 2025.

_____. **DeepSeek Web Page**. 2025. <<https://www.deepseek.com>>. Accessed: 2025-02-20.

_____. **Temperature Parameter**. 2025. <https://api-docs.deepseek.com/quick_start/parameter_settings>. Accessed on April 10, 2025.

DEISSENBOCK, F.; PIZKA, M. Concise and consistent naming [software system identifier naming]. In: **13th International Workshop on Program Comprehension (IWPC'05)**. [s.n.], 2005. p. 97–106. Available at: <<https://doi.org/10.1007/s11219-006-9219-1>>.

DORN, J. **A General Software Readability Model**. 2012. Available at: <<https://api.semanticscholar.org/CorpusID:14098740>>.

EBERT, C.; LOURIDAS, P. Generative AI for software practitioners. **IEEE Software**, v. 40, p. 30–38, 07 2023. Available at: <<https://doi.org/10.1109/MS.2023.3265877>>.

EBERT, F. et al. An exploratory study on confusion in code reviews. **Empirical Software Engineering**, v. 26, p. 1–48, 2021. Available at: <<https://doi.org/10.1007/s10664-020-09909-5>>.

ELSHOFF, J. L.; MARCOTTY, M. Improving computer program readability to aid modification. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 25, n. 8, p. 512–521, aug 1982. ISSN 0001-0782. Available at: <<https://doi.org/10.1145/358589.358596>>.

ERLIKH, L. Leveraging legacy system dollars for e-business. **IT Professional**, v. 2, n. 3, p. 17–23, 2000. Available at: <<https://doi.org/10.1109/6294.846201>>.

ERNST, N. A.; BAVOTA, G. AI-driven development is here: Should you worry? **IEEE Software**, Institute of Electrical and Electronics Engineers (IEEE), v. 39, n. 2, p. 106–110, mar 2022. Available at: <<https://doi.org/10.1109/MS.2021.3133805>>.

ESLINT. **ESLint Web Page**. 2025. <<https://eslint.org/>>. Accessed: 2025-02-20.

FAGADAU, I. D. et al. Analyzing prompt influence on automated method generation: An empirical study with copilot. In: **Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension**. New York, NY, USA: Association for Computing Machinery, 2024. (ICPC '24), p. 24–34. ISBN 9798400705861. Available at: <<https://doi.org/10.1145/3643916.3644409>>.

FAKHOURY, S. et al. Improving source code readability: Theory and practice. In: **Proceedings of the 27th International Conference on Program Comprehension**. IEEE Press, 2019. (ICPC '19), p. 2–12. Available at: <<https://doi.org/10.1109/ICPC.2019.00014>>.

FINDBUGS. **Findbugs Web Page**. 2025. <<https://findbugs.sourceforge.net/>>. Accessed: 2025-02-20.

FLESCH, R. A new readability yardstick. **Journal of Applied Psychology**, p. 221–233, 1948. Available at: <<https://doi.org/10.1037/h0057532>>.

FLURI, B.; GALL, H. Classifying change types for qualifying change couplings. In: **14th IEEE International Conference on Program Comprehension (ICPC'06)**. [s.n.], 2006. p. 35–45. Available at: <<https://doi.org/10.1109/ICPC.2006.16>>.

FLURI, B. et al. Change distilling: tree differencing for fine-grained source code change extraction. **IEEE Transactions on Software Engineering**, v. 33, 2007. Available at: <<https://doi.org/10.1109/TSE.2007.70731>>.

FOWLER, M. **Refactoring: Improving the Design of Existing Code**. Boston, MA, USA: Addison-Wesley, 1999. ISBN 0-201-48567-2.

FRIEDMAN, M. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. **Journal of the American Statistical Association**, ASA Website, v. 32, n. 200, p. 675–701, 1937. Available at: <<https://doi.org/10.2307/2279372>>.

G1. **App chinês DeepSeek supera ChatGPT nos EUA e derruba ações de empresas ligadas à IA**. 2025. <<https://acesse.one/tvIsb>>. Accessed: 2025-02-20.

GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. Boston, MA: Addison-Wesley, 1994. ISBN 978-0201633610.

GITHUB. **The 2024 State of the Octoverse – GitHub Report**. 2024. Available at: <<https://octoverse.github.com/>>.

GITHUB. **Github**. 2025. <<https://github.com/>>. Accessed: 2025-02-20.

_____. **Github API V3**. 2025. <<https://developer.github.com/v3/>>. Accessed: 2025-02-20.

_____. **Github GraphQL V4**. 2025. <<https://developer.github.com/v4/>>. Accessed: 2025-02-20.

GITHUB. **GitHub Pull Requests**. 2025. Available at: <<https://docs.github.com/en/pull-requests/collaborating-with-pull-requests>>.

GOODGER, D.; ROSSUM, G. van. **PEP 257 – Docstring Conventions**. 2025. <<https://peps.python.org/pep-0257/>>. Accessed: 2025-02-20.

GOOGLE. **Gemini Web Page**. 2025. <<https://gemini.google.com/?hl=pt-BR>>. Accessed: 2025-02-20.

GOOGLE. **Google Java Style Guide**. 2025. <<https://google.github.io/styleguide/javaguide.html>>. Accessed: 2025-02-20.

GOOGLE. **Google Search**. 2025. <<https://www.google.com/>>. Accessed: 2025-02-20.

GOOGLESCHOLAR. **Assessing the readability of ChatGPT code snippet recommendations: a comparative study - Citations**. 2025. <https://scholar.google.com.br/scholar?cites=1013109633119088303&as_sdt=2005&scioldt=0,5&hl=pt-BR/>. Accessed on June 19, 2025.

_____. **CROKAGE: effective solution recommendation for programming tasks by leveraging crowd knowledge - Citations**. 2025. <https://scholar.google.com.br/scholar?cites=17847239508875336956&as_sdt=2005&scioldt=0,5&hl=pt-BR>. Accessed on June 19, 2025.

_____. **How do developers improve code readability? an empirical study of pull requests - Citations**. 2025. <https://scholar.google.com.br/scholar?cites=5600753986694327836&as_sdt=2005&scioldt=0,5&hl=pt-BR/>. Accessed on June 19, 2025.

_____. **Improved retrieval of programming solutions with code examples using a multi-featured score - Citations**. 2025. <https://scholar.google.com.br/scholar?hl=pt-BR&as_sdt=0%2C5&q=Improved+Retrieval+of+Programming+Solutions+with+Code+Examples+Using+a+Multi-featured+Score&btnG=>>. Accessed on June 19, 2025.

_____. **Readability and Understandability of Snippets Recommended by General-purpose Web Search Engines - Citations**. 2025. <https://scholar.google.com.br/scholar?cites=16557284020627472748&as_sdt=2005&scioldt=0,5&hl=pt-BR/>. Accessed on June 19, 2025.

_____. **Readability and understandability scores for snippet assessment: An exploratory study - Citations**. 2025. <https://scholar.google.com.br/scholar?cites=9560195858097445568&as_sdt=2005&scioldt=0,5&hl=pt-BR/>. Accessed on June 19, 2025.

GOPSTEIN, D. et al. Understanding misunderstandings in source code. In: **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2017. (ESEC/FSE 2017), p. 129–139. ISBN 9781450351058. Available at: <<https://doi.org/10.1145/3106237.3106264>>.

GOSAVI, P.; RAJLICH, V. Incremental Change in Object-Oriented Programming . **IEEE Software**, IEEE Computer Society, Los Alamitos, CA, USA, v. 21, n. 04, p. 62–69, jul. 2004. ISSN 1937-4194. Available at: <<https://doi.org/10.1109/MS.2004.17>>.

GOUSIOS, G. et al. Work practices and challenges in pull-based development: The integrator's perspective. In: **2015 IEEE/ACM 37th IEEE International Conference on Software Engineering**. [s.n.], 2015. v. 1, p. 358–368. Available at: <<https://doi.org/10.1145/2884781.2884826>>.

GUO, Q. et al. Exploring the potential of chatgpt in automated code refinement: An empirical study. In: **Proceedings of the IEEE/ACM 46th International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2024. (ICSE '24). ISBN 9798400702174. Available at: <<https://doi.org/10.1145/3597503.3623306>>.

HALSTEAD, M. H. **Elements of Software Science**. New York: Elsevier North-Holland, 1977. v. 7. (Operating and Programming Systems Series, v. 7). ISBN 9780444002051.

HANEEF, N. J. Software documentation and readability: a proposed process improvement. **SIGSOFT Softw. Eng. Notes**, Association for Computing Machinery, New York, NY, USA, v. 23, n. 3, p. 75–77, maio 1998. ISSN 0163-5948. Available at: <<https://doi.org/10.1145/279437.279470>>.

HAO, H. et al. An empirical study on developers' shared conversations with chatgpt in github pull requests and issues. **Empirical Softw. Engg.**, Kluwer Academic Publishers, USA, v. 29, n. 6, set. 2024. ISSN 1382-3256. Available at: <<https://doi.org/10.1007/s10664-024-10540-x>>.

HOLMES, R. et al. The end-to-end use of source code examples: An exploratory study. In: **2009 IEEE International Conference on Software Maintenance**. [s.n.], 2009. p. 555–558. Available at: <<https://doi.org/10.1109/ICSM.2009.5306387>>.

HOLMES, R.; WALKER, R. J. Systematizing pragmatic software reuse. **ACM Trans. Softw. Eng. Methodol.**, Association for Computing Machinery, New York, NY, USA, v. 21, n. 4, fev. 2013. ISSN 1049-331X. Available at: <<https://doi.org/10.1145/2377656.2377657>>.

HOPKINS, N. F. et al. **The ThoughtWorks Anthology: Essays on Software Technology and Innovation**. Raleigh, NC: Pragmatic Bookshelf, 2008. ISBN 9780978739218.

HORA, A. Characterizing top ranked code examples in google. **Journal of Systems and Software**, v. 178, p. 110971, 2021. ISSN 0164-1212. Available at: <<https://doi.org/10.1016/j.jss.2021.110971>>.

_____. Googling for software development: What developers search for and what they find. In: **2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)**. [s.n.], 2021. p. 317–328. Available at: <<https://doi.org/10.1109/MSR52588.2021.00044>>.

HORA, A. C. Apisonar: Mining api usage examples. **Software: Practice and Experience**, v. 51, p. 319 – 352, 2021. Available at: <<https://doi.org/10.1002/spe.2906>>.

HOVEMEYER, D.; PUGH, W. Finding bugs is easy. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 39, n. 12, p. 92–106, dec 2004. ISSN 0362-1340. Available at: <<https://doi.org/10.1145/1052883.1052895>>.

JAOUA, I.; SGHAIER, O. B.; SAHRAOUI, H. Combining Large Language Models with Static Analyzers for Code Review Generation . In: **2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)**. Los Alamitos, CA, USA: IEEE Computer Society, 2025. p. 174–186. Available at: <<https://doi.org/10.1109/MSR66628.2025.00038>>.

JAVA. **Java Language Specification Web Page**. 2025. <<https://docs.oracle.com/javase/specs/jls/se17/html/jls-8.html#jls-8.1.1>>. Accessed: 2025-02-20.

JOHNSON, B. et al. Why don't software developers use static analysis tools to find bugs? In: **2013 35th International Conference on Software Engineering (ICSE)**. IEEE Press, 2013. (ICSE '13), p. 672–681. ISBN 9781467330763. Available at: <<https://doi.org/10.1109/ICSE.2013.6606613>>.

JOHNSON, J. et al. An empirical study assessing source code readability in comprehension. In: **2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [s.n.], 2019. p. 513–523. Available at: <<https://doi.org/10.1109/ICSME.2019.00085>>.

- KEIVANLOO, I.; RILLING, J.; ZOU, Y. Spotting working code examples. In: **Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014), p. 664–675. ISBN 9781450327565. Available at: <<https://doi.org/10.1145/2568225.2568292>>.
- KEMERER, C. Software complexity and software maintenance: A survey of empirical research. **Ann. Software Eng.**, v. 1, p. 1–22, 12 1995. Available at: <<https://doi.org/10.1007/BF02249043>>.
- KNIGHT, J. C.; MYERS, E. A. Phased inspections and their implementation. **SIGSOFT Softw. Eng. Notes**, Association for Computing Machinery, New York, NY, USA, v. 16, n. 3, p. 29–35, jul 1991. ISSN 0163-5948. Available at: <<https://doi.org/10.1145/127099.127101>>.
- KRUSKAL, W. H.; WALLIS, W. A. Use of ranks in one-criterion variance analysis. **Journal of the American Statistical Association**, Taylor & Francis, v. 47, n. 260, p. 583–621, 1952. Available at: <<https://doi.org/10.2307/2280779>>.
- LANDIS, J. R.; KOCH, G. G. The measurement of observer agreement for categorical data. **Biometrics**, v. 33, n. 1, p. 159–174, 1977. Available at: <<https://doi.org/10.2307/2529310>>.
- LANGHOUT, C. V.; ANICHE, M. F. Atoms of confusion in java. **2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)**, p. 25–35, 2021. Available at: <<https://doi.org/10.1109/ICPC52881.2021.00012>>.
- LAWRIE, D. J. et al. Effective identifier names for comprehension and memory. **Innovations in Systems and Software Engineering**, v. 3, p. 303–318, 2007. Available at: <<https://doi.org/10.1007/s11334-007-0031-2>>.
- LENARDUZZI, V. et al. A critical comparison on six static analysis tools: Detection, agreement, and precision. **Journal of Systems and Software**, v. 198, p. 111575, 2023. ISSN 0164-1212. Available at: <<https://doi.org/10.1016/j.jss.2022.111575>>.
- LIKERT, R. A technique for the measurement of attitudes. **Archives of Psychology**, v. 40, p. 44–53, 1932. Available at: <<https://www.jstor.org/stable/1471067>>.
- LIN, B. et al. Investigating the use of code analysis and nlp to promote a consistent usage of identifiers. In: **2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [s.n.], 2017. p. 81–90. Available at: <<https://doi.org/10.1109/SCAM.2017.17>>.
- LOMBOK. **Lombok Framework**. 2025. <<https://projectlombok.org/>>. Accessed on April 10, 2025.
- LORIOT, B.; MADEIRAL, F.; MONPERRUS, M. Styler: learning formatting conventions to repair checkstyle violations. **Empirical Software Engineering**, v. 27, 08 2022. Available at: <<https://doi.org/10.1007/s10664-021-10107-0>>.
- MADI, N. A. How readable is model-generated code? examining readability and visual inspection of github copilot. In: **Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering**. New York, NY,

USA: Association for Computing Machinery, 2023. (ASE '22). ISBN 9781450394758. Available at: <<https://doi.org/10.1145/3551349.3560438>>.

MARCILIO, D. et al. Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In: **2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)**. [s.n.], 2019. p. 209–219. Available at: <<https://doi.org/10.1109/ICPC.2019.00040>>.

MARTIN, R. C. **Clean Code: A Handbook of Agile Software Craftsmanship**. Upper Saddle River, NJ: Prentice Hall, 2008. ISBN 9780132350884.

MARTINEZ, M.; MONPERRUS, M. Coming: A tool for mining change pattern instances from git commits. In: **2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)**. [s.n.], 2019. p. 79–82. Available at: <<https://doi.org/10.1109/ICSE-Companion.2019.00043>>.

MASTROPAOLO, A. et al. On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot . In: **2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)**. Los Alamitos, CA, USA: IEEE Computer Society, 2023. p. 2149–2160. Available at: <<https://doi.org/10.1109/ICSE48619.2023.00181>>.

MCCABE, T. A complexity measure. **IEEE Transactions on Software Engineering**, SE-2, n. 4, p. 308–320, 1976. Available at: <<https://doi.org/10.1109/TSE.1976.233837>>.

MILLER, G. A. Wordnet: A lexical database for english. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 38, n. 11, p. 39–41, nov 1995. ISSN 0001-0782. Available at: <<https://doi.org/10.1145/219717.219748>>.

MINELLI, R.; AND, A. M.; LANZA, M. I know what you did last summer: an investigation of how developers spend their time. In: **Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension**. IEEE Press, 2015. (ICPC '15), p. 25–35. Available at: <<https://doi.org/10.1109/ICPC.2015.12>>.

MONDAL, S.; BAPPON, S. D.; ROY, C. Enhancing user interaction in ChatGPT: Characterizing and consolidating multiple prompts for issue resolution. In: **Proceedings of the International Conference on Mining Software Repositories (MSR 2024)**. [s.n.], 2024. Available at: <<https://doi.org/10.1145/3643991.3645085>>.

MORENO, L. et al. How can i use this method? In: **Proceedings of the 37th International Conference on Software Engineering - Volume 1**. IEEE Press, 2015. (ICSE '15), p. 880–890. ISBN 9781479919345. Available at: <<https://doi.org/10.1109/ICSE.2015.98>>.

MUNAIAH, N. et al. Curating github for engineered software projects. **Empirical Softw. Engg.**, Kluwer Academic Publishers, USA, v. 22, n. 6, p. 3219–3253, dec 2017. ISSN 1382-3256. Available at: <<https://doi.org/10.1007/s10664-017-9512-6>>.

NAKAMARU, T. et al. An empirical study of method chaining in java. In: **Proceedings of the 17th International Conference on Mining Software Repositories**. New York, NY, USA: Association for Computing Machinery, 2020. (MSR '20), p. 93–102. ISBN 9781450375177. Available at: <<https://doi.org/10.1145/3379597.3387441>>.

NAM, D. et al. Using an llm to help with code understanding. In: **Proceedings of the IEEE/ACM 46th International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2024. (ICSE '24). ISBN 9798400702174. Available at: <<https://doi.org/10.1145/3597503.3639187>>.

NGUYEN, N.; NADI, S. An empirical evaluation of github copilot's code suggestions. In: **Proceedings of the 19th International Conference on Mining Software Repositories**. New York, NY, USA: Association for Computing Machinery, 2022. (MSR '22), p. 1–5. ISBN 9781450393034. Available at: <<https://doi.org/10.1145/3524842.3528470>>.

NIELSON, F.; NIELSON, H. R.; HANKIN, C. **Principles of Program Analysis**. Springer Publishing Company, Incorporated, 2010. ISBN 3642084745. Available at: <<https://dl.acm.org/doi/book/10.5555/1965094>>.

NIU, H.; KEIVANLOO, I.; ZOU, Y. Learning to rank code examples for code search engines. **Empirical Softw. Engg.**, Kluwer Academic Publishers, USA, v. 22, n. 1, p. 259–291, fev. 2017. ISSN 1382-3256. Available at: <<https://doi.org/10.1007/s10664-015-9421-5>>.

OLIVEIRA, D. et al. Developers' perception matters: machine learning to detect developer-sensitive smells. **Empirical Softw. Engg.**, Kluwer Academic Publishers, USA, v. 27, n. 7, dez. 2022. ISSN 1382-3256. Available at: <<https://doi.org/10.1007/s10664-022-10234-2>>.

_____. Evaluating code readability and legibility: An examination of human-centric studies. In: **2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [s.n.], 2020. p. 348–359. Available at: <<https://doi.org/10.1109/ICSME46990.2020.00041>>.

_____. Understanding code understandability improvements in code reviews. **IEEE Transactions on Software Engineering**, v. 51, n. 1, p. 14–37, 2025. Available at: <<https://doi.org/10.1109/TSE.2024.3453783>>.

OLIVEIRA, P. et al. Assessing python style guides: An eye-tracking study with novice developers. In: **Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software**. Porto Alegre, RS, Brasil: SBC, 2024. p. 136–146. ISSN 2833-0633. Available at: <<https://doi.org/10.5753/sbes.2024>>.

OPDYKE, W. F. **Refactoring Object-Oriented Frameworks**. Tese (Doutorado) — University of Illinois at Urbana-Champaign, USA, 1992. UMI Order No. GAX93-05645.

ORACLE. **Oracle's Code Conventions for the Java Programming Language**. 2025. <<https://www.oracle.com/java/technologies/javase/codeconventions-contents.html>>. Accessed: 2025-02-20.

PALOMBA, F. et al. Do they really smell bad? a study on developers' perception of bad code smells. In: **2014 IEEE International Conference on Software Maintenance and Evolution**. [s.n.], 2014. p. 101–110. Available at: <<https://doi.org/10.1109/ICSME.2014.32>>.

_____. An exploratory study on the relationship between changes and refactoring. In: **2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)**. [s.n.], 2017. p. 176–185. Available at: <<https://doi.org/10.1109/ICPC.2017.38>>.

PANTIUCHINA, J.; LANZA, M.; BAVOTA, G. Improving code: The (mis) perception of quality metrics. In: **2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [s.n.], 2018. p. 80–91. Available at: <<https://doi.org/10.1109/ICSME.2018.00017>>.

PANTIUCHINA, J. et al. Why developers refactor source code: A mining-based study. **ACM Trans. Softw. Eng. Methodol.**, Association for Computing Machinery, New York, NY, USA, v. 29, n. 4, set. 2020. ISSN 1049-331X. Available at: <<https://doi.org/10.1145/3408302>>.

PETERS, T. **PEP 20 – The Zen of Python**. 2004. <<https://peps.python.org/pep-0020/>>. Accessed: 2025-02-20.

PIANTADOSI, V. On the evolution of code readability. In: **2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [s.n.], 2022. p. 597–601. Available at: <<https://doi.org/10.1109/ICSME55016.2022.00082>>.

PIANTADOSI, V. et al. How does code readability change during software evolution? **Empirical Software Engineering**, v. 25, p. 1–39, 11 2020. Available at: <<https://doi.org/10.1007/s10664-020-09886-9>>.

PMD. **Pmd**. 2025. <<https://pmd.github.io/>>. Accessed: 2025-02-20.

PONZANELLI, L. et al. Prompter: Turning the ide into a self-confident programming assistant. **Empirical Software Engineering**, Springer, v. 21, p. 2190–2231, 2016. Available at: <<https://doi.org/10.1007/s10664-015-9397-1>>.

POSNETT, D.; HINDLE, A.; DEVANBU, P. A simpler model of software readability. In: **Proceedings of the 8th Working Conference on Mining Software Repositories**. New York, NY, USA: Association for Computing Machinery, 2011. (MSR '11), p. 73–82. ISBN 9781450305747. Available at: <<https://doi.org/10.1145/1985441.1985454>>.

PYLINT. **Pylint Web Page**. 2025. <<https://pypi.org/project/pylint/>>. Accessed: 2025-02-20.

PYTHON. **Python Enhancement Proposals (PEPs)**. 2025. <<https://peps.python.org>>. Accessed: 2025-02-20.

PYTHONBLACK. **Python Black site**. 2025. <<https://github.com/psf/black>>. Accessed on April 10, 2025.

RADFORD, A. et al. Language models are unsupervised multitask learners. 2018. Available at: <https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf>.

RAFFEL, C. et al. Exploring the limits of transfer learning with a unified text-to-text transformer. **J. Mach. Learn. Res.**, JMLR.org, v. 21, n. 1, jan. 2020. ISSN 1532-4435. Available at: <<http://jmlr.org/papers/v21/20-074.html>>.

RAMBALLY, G. K. The influence of color on program readability and comprehensibility. In: **SIGCSE '86**. [s.n.], 1986. Available at: <<https://doi.org/10.1145/5600.5702>>.

REFACTORINGMINER. **Refactoring Miner Tool**. 2025. <<https://github.com/tsantalis/RefactoringMiner/>>. Accessed: 2025-02-20.

RIGBY, P. C.; GERMAN, D. M.; STOREY, M.-A. Open source software peer review practices: A case study of the apache server. In: **Proceedings of the 30th International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2008. (ICSE '08), p. 541–550. ISBN 9781605580791. Available at: <<https://doi.org/10.1145/1368088.1368162>>.

ROMANO, S. et al. Do static analysis tools affect software quality when using test-driven development? In: **Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement**. New York, NY, USA: Association for Computing Machinery, 2022. (ESEM '22), p. 80–91. ISBN 9781450394277. Available at: <<https://doi.org/10.1145/3544902.3546233>>.

ROSSUM, G. van; HETTINGER, R. **PEP 8 – Style Guide for Python Code**. 2025. <<https://peps.python.org/pep-0008/>>. Accessed: 2025-02-20.

ROSSUM, G. van; LEHTOSALO, J. **PEP 484 – Type Hints**. 2014. <<https://peps.python.org/pep-0484/>>. Accessed: 2025-02-20.

ROY, D. et al. A model to detect readability improvements in incremental changes. In: _____. **Proceedings of the 28th International Conference on Program Comprehension**. New York, NY, USA: Association for Computing Machinery, 2020. p. 25–36. ISBN 9781450379588. Available at: <<https://doi.org/10.1145/3387904.3389255>>.

RSM. **RSM**. 2025. <<https://msquaredtechnologies.com/>>. Accessed on February 6, 2025.

RUBEI, R. et al. Postfinder: Mining stack overflow posts to support software developers. **Information and Software Technology**, v. 127, p. 106367, 2020. ISSN 0950-5849. Available at: <<https://doi.org/10.1016/j.infsof.2020.106367>>.

SADOWSKI, C.; STOLEE, K. T.; ELBAUM, S. How developers search for code: a case study. In: **Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2015. (ESEC/FSE 2015), p. 191–201. ISBN 9781450336758. Available at: <<https://doi.org/10.1145/2786805.2786855>>.

SADOWSKI, C. et al. Modern code review: A case study at google. In: **2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)**. [s.n.], 2018. p. 181–190. Available at: <<https://doi.org/10.1145/3183519.3183525>>.

SANDER, J. et al. Density-based clustering in spatial databases: The algorithm gdbscan and its applications. **Data Mining and Knowledge Discovery**, v. 2, n. 2, p. 169–194, 1998. Available at: <<https://doi.org/10.1023/A:1009745219419>>.

SANTOS, R. M. a. dos; GEROSA, M. A. Impacts of coding practices on readability. In: **Proceedings of the 26th Conference on Program Comprehension**. New York, NY, USA: Association for Computing Machinery, 2018. (ICPC '18), p. 277–285. ISBN 9781450357142. Available at: <<https://doi.org/10.1145/3196321.3196342>>.

SARWAR, M. M. S.; SHAHZAD, S.; AHMAD, I. Cyclomatic complexity: The nesting problem. In: **Eighth International Conference on Digital Information Management (ICDIM 2013)**. [s.n.], 2013. p. 274–279. Available at: <<https://doi.org/10.1109/ICDIM31829.2013>>.

SCALABRINO, S. et al. Automatically assessing code understandability: How far are we? In: **2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [s.n.], 2017. p. 417–427. Available at: <<https://doi.org/10.1109/TSE.2019.2901468>>.

_____. Improving code readability models with textual features. In: **2016 IEEE 24th International Conference on Program Comprehension (ICPC)**. [s.n.], 2016. p. 1–10. Available at: <<https://doi.org/10.1109/ICPC.2016.7503707>>.

_____. A comprehensive model for code readability. **Journal of Software: Evolution and Process**, v. 30, 06 2018. Available at: <<https://doi.org/10.1002/smr.1958>>.

SILVA, D.; TSANTALIS, N.; VALENTE, M. T. Why we refactor? confessions of github contributors. In: **Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2016. (FSE 2016), p. 858–870. ISBN 9781450342186. Available at: <<https://doi.org/10.1145/2950290.2950305>>.

SILVA, J.; DANTAS, C.; MAIA, M. What developers ask to chatgpt in github pull requests? an exploratory study. In: **Anais do XII Workshop de Visualização, Evolução e Manutenção de Software**. Porto Alegre, RS, Brasil: SBC, 2024. p. 125–136. ISSN 0000-0000. Available at: <<https://doi.org/10.5753/vem.2024.3913>>.

SILVA, R. F. et al. Improved retrieval of programming solutions with code examples using a multi-featured score. **Journal of Systems and Software**, v. 181, p. 111063, 2021. ISSN 0164-1212. Available at: <<https://doi.org/10.1016/j.jss.2021.111063>>.

SILVA, R. F. G. da et al. CROKAGE: effective solution recommendation for programming tasks by leveraging crowd knowledge. **Empir. Softw. Eng.**, v. 25, n. 6, p. 4707–4758, 2020. Available at: <<https://doi.org/10.1007/s10664-020-09863-2>>.

SMIT, M. et al. Code convention adherence in evolving software. In: **2011 27th IEEE International Conference on Software Maintenance (ICSM)**. [s.n.], 2011. p. 504–507. Available at: <<https://doi.org/10.1109/ICSM.2011.6080819>>.

SOBANIA, D. et al. An Analysis of the Automatic Bug Fixing Performance of ChatGPT . In: **2023 IEEE/ACM International Workshop on Automated Program Repair (APR)**. Los Alamitos, CA, USA: IEEE Computer Society, 2023. p. 23–30. Available at: <<https://doi.org/10.1109/APR59189.2023.00012>>.

SONARLINT. **Rule 1124**. 2025. <<https://rules.sonarsource.com/java/RSPEC-1124/>>. Accessed: 2025-02-20.

_____. **Rule 1125**. 2025. <<https://rules.sonarsource.com/java/RSPEC-1125/>>. Accessed: 2025-02-20.

_____. **Rule 1130**. 2025. <<https://rules.sonarsource.com/java/RSPEC-1130/>>. Accessed: 2025-02-20.

_____. **Rule 1155**. 2025. <<https://rules.sonarsource.com/java/RSPEC-1155/>>. Accessed: 2025-02-20.

_____. **Rule 1161**. 2025. <<https://rules.sonarsource.com/java/RSPEC-1161/>>. Accessed: 2025-02-20.

_____. **Rule 117**. 2025. <<https://rules.sonarsource.com/java/RSPEC-117/>>. Accessed: 2025-02-20.

_____. **Rule 1481**. 2025. <<https://rules.sonarsource.com/java/RSPEC-1481/>>. Accessed: 2025-02-20.

_____. **Rule 1488**. 2025. <<https://rules.sonarsource.com/java/RSPEC-1488/>>. Accessed: 2025-02-20.

_____. **Rule 1612**. 2025. <<https://rules.sonarsource.com/java/RSPEC-1612/>>. Accessed: 2025-02-20.

_____. **Rule 1659**. 2025. <<https://rules.sonarsource.com/java/RSPEC-1659/>>. Accessed: 2025-02-20.

_____. **Rule 1854**. 2025. <<https://rules.sonarsource.com/java/RSPEC-1854/>>. Accessed: 2025-02-20.

_____. **Rule 2293**. 2025. <<https://rules.sonarsource.com/java/RSPEC-2293/>>. Accessed: 2025-02-20.

_____. **Rule 3776**. 2025. <<https://rules.sonarsource.com/java/RSPEC-3776/>>. Accessed: 2025-02-20.

SONARLINT. **Sonarlint Rules Web Page**. 2025. <<https://rules.sonarsource.com/java/type/Code%20Smell/>>. Accessed: 2025-02-20.

_____. **Sonarlint Web Page**. 2025. <<https://www.sonarsource.com/products/sonarlint/>>. Accessed: 2025-02-20.

SOURCE METER. **Source Meter Web Page**. 2025. <<https://www.sourcemeter.com/>>. Accessed: 2025-02-20.

SPEARMAN, C. The proof and measurement of association between two things. **The American Journal of Psychology**, University of Illinois Press, v. 15, n. 1, p. 72–101, 1904. Available at: <<https://doi.org/10.2307/1412159>>.

STACK OVERFLOW. **Stack Overflow Blog**. 2025. <<https://stackoverflow.blog/2019/08/14/crokage-a-new-way-to-search-stack-overflow/>>. Accessed: 2025-02-20.

_____. **Stack Overflow Web Page**. 2025. <<https://stackoverflow.com/>>. Accessed: 2025-02-20.

STAPLETON, S. et al. A human study of comprehension and code summarization. In: **Proceedings of the 28th International Conference on Program Comprehension**. New York, NY, USA: Association for Computing Machinery, 2020. (ICPC '20), p. 2–13. ISBN 9781450379588. Available at: <<https://doi.org/10.1145/3387904.3389258>>.

SZÓKE, G. et al. Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? In: **2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation**. [s.n.], 2014. p. 95–104. Available at: <<https://doi.org/10.1109/SCAM.2014.18>>.

TAMANNA, S. B. et al. Chatgpt Inaccuracy Mitigation During Technical Report Understanding: Are we There Yet? . In: **2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)**. Los Alamitos, CA, USA: IEEE Computer Society, 2025. p. 2290–2302. Available at: <<https://doi.org/10.1109/ICSE55347.2025.00145>>.

THIES, A.; ROTH, C. Recommending rename refactorings. In: **Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2010. (RSSE '10), p. 1–5. ISBN 9781605589749. Available at: <<https://doi.org/10.1145/1808920.1808921>>.

TIARKS, R. What programmers really do - an observational study. **Softwaretechnik-Trends**, v. 31, 2011. Available at: <<https://dl.gi.de/server/api/core/bitstreams/e6bef151-fd86-4acd-a2ad-81f66aa4b101/content>>.

TORRES, A. et al. An investigation of confusing code patterns in javascript. **Journal of Systems and Software**, v. 203, p. 111731, 2023. ISSN 0164-1212. Available at: <<https://doi.org/10.1016/j.jss.2023.111731>>.

TSANTALIS, N. et al. Accurate and efficient refactoring detection in commit history. In: **Proceedings of the 40th International Conference on Software Engineering**. New York, NY, USA: ACM, 2018. (ICSE '18), p. 483–494. ISBN 978-1-4503-5638-1. Available at: <<https://doi.org/10.1145/3180155.3180206>>.

TUFANO, R. et al. Unveiling chatgpt's usage in open source projects: A mining-based study. In: **Proceedings of the 21st International Conference on Mining Software Repositories**. New York, NY, USA: Association for Computing Machinery, 2024. (MSR '24), p. 571–583. ISBN 9798400705878. Available at: <<https://doi.org/10.1145/3643991.3644918>>.

UMARJI, M.; SIM, S. E.; LOPES, C. Archetypal internet-scale source code searching. In: RUSSO, B. et al. (Ed.). **Open Source Development, Communities and Quality**. Boston, MA: Springer US, 2008. p. 257–263. ISBN 978-0-387-09684-1. Available at: <https://doi.org/10.1007/978-1-4614-6596-6_3>.

VASSALLO, C. et al. Context is king: The developer perspective on the usage of static analysis tools. In: **2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [s.n.], 2018. p. 38–49. Available at: <<https://doi.org/10.1109/SANER.2018.8330195>>.

- VASWANI, A. et al. Attention is all you need. In: **Proceedings of the 31st International Conference on Neural Information Processing Systems**. Red Hook, NY, USA: Curran Associates Inc., 2017. (NIPS'17), p. 6000–6010. ISBN 9781510860964. Available at: <<https://dl.acm.org/doi/10.5555/3295222.3295349>>.
- VEGI, L. F. d. M.; VALENTE, M. T. Code smells in elixir: early results from a grey literature review. In: **Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension**. New York, NY, USA: Association for Computing Machinery, 2022. (ICPC '22), p. 580–584. ISBN 9781450392983. Available at: <<https://doi.org/10.1145/3524610.3527881>>.
- VITALE, A. et al. Using deep learning to automatically improve code readability. In: **2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. Los Alamitos, CA, USA: IEEE Computer Society, 2023. p. 573–584. Available at: <<https://doi.org/10.1109/ASE56229.2023.00112>>.
- W3SCHOOLS. **W3Schools Online Web Tutorials**. 2025. <<https://www.w3schools.com/>>. Accessed: 2025-02-20.
- XIA, X. et al. What do developers search for on the web? **Empir Software Engineering**, Kluwer Academic Publishers, USA, v. 22, n. 6, p. 3149–3185, dez. 2017. ISSN 1382-3256. Available at: <<https://doi.org/10.1007/s10664-017-9514-4>>.
- XIAO, T. et al. DevGPT: Studying Developer-ChatGPT Conversations. In: **Proceedings of the International Conference on Mining Software Repositories (MSR 2024)**. [s.n.], 2024. Available at: <<https://doi.org/10.1145/3643991.3648400>>.
- ZHANG, Z. et al. Automated refactoring of non-idiomatic python code with pythonic idioms. **IEEE Transactions on Software Engineering**, PP, p. 1–22, 11 2024. Available at: <<https://doi.org/10.1109/TSE.2024.3420886>>.
- ZID, C. et al. A study on the pythonic functional constructs' understandability. In: **Proceedings of the IEEE/ACM 46th International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2024. (ICSE '24). ISBN 9798400702174. Available at: <<https://doi.org/10.1145/3597503.3639211>>.
- ZUCCON, G.; KOOPMAN, B.; SHAIK, R. Chatgpt hallucinates when attributing answers. In: **Proceedings of the Annual International ACM SIGIR Conference on Research and Development in Information Retrieval in the Asia Pacific Region**. New York, NY, USA: Association for Computing Machinery, 2023. p. 46–51. ISBN 9798400704086. Available at: <<https://doi.org/10.1145/3624918.3625329>>.