CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

The Impact of Compiler Warnings on Code Quality in C++ Projects

Master's thesis in Software Engineering and Technology

ALBIN JOHANSSON
CARL HOLMBERG

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

The Impact of Compiler Warnings on Code Quality in C++ Projects

ALBIN JOHANSSON
CARL HOLMBERG

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Interaction Design and Software Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

The Impact of Compiler Warnings on Code Quality in C++ Projects
ALBIN JOHANSSON
CARL HOLMBERG

Supervisor: Philipp Leitner, Department of Computer Science and Engineering
Examiner: Lucas Gren, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Division of Interaction Design and Software Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2023

The Impact of Compiler Warnings on Code Quality in C++ Projects
ALBIN JOHANSSON
CARL HOLMBERG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The usage of compiler warnings has widely been assumed to have a positive effect on code quality, but this notion has little evidence in research. This study investigated the relationship between the usage of compiler warnings and overall code quality in C++ projects. A purposive sampling approach was used to collect 127 C++ projects, which were subsequently analyzed with SonarCloud. Bayesian data analysis was utilized to investigate the correlation between compiler warning usage and measured code quality. The results indicated a correlation between stricter compiler warning usage and improved code quality for some code quality metrics, such as bugs, code smells, and technical debt. Projects that used compiler warnings generally performed better than projects with no warnings enabled, even for code quality metrics that were deemed unlikely to be affected by compiler warnings, such as the number of security hotspots and duplicated lines. Notably, projects that treated warnings as errors performed substantially better than similar projects that did not treat their warnings as errors. One proposed explanation was that this could be caused by a tendency among developers to ignore compiler warnings. It was concluded that the usage of stricter compiler warnings and improved code quality are correlated, while external factors such as engineering culture also likely contributed to the results.

Keywords: compiler warnings, code quality, C++, static code analysis.

Acknowledgments

First and foremost, we would like to thank our supervisor Philipp Leitner for his support and feedback during the writing process. His feedback helped the thesis grow far beyond what it could have been without him. We would also like to thank Hampus Broman and William Levén for their excellent Master's thesis, which inspired our process. Lastly, we also thank our examiner Lucas Gren for his valuable insights, especially regarding Bayesian statistical methods.

Albin Johansson & Carl Holmberg, Gothenburg, June 2023

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis, listed in alphabetical order:

BDA	Bayesian data analysis
CI	Credible interval
CSV	Comma-separated values (file format)
DAG	Directed acyclic graph
ELPD	Expected log predictive density (Estimate of predictive prowess of a statistical model given new data)
GCC	GNU Compiler Collections
KLOC	1,000 lines of code
LOO	Leave-one-out (model cross-validation strategy)
MCMC	Markov chain Monte Carlo
MSVC	Microsoft Visual C++
SE	Standard error
SD	Standard deviation
UB	Undefined behavior

Contents

List of Acronyms **ix**

List of Figures **xv**

List of Tables **xix**

1 Introduction **1**
1.1 Motivating Example . . . 1
1.2 Purpose and Research Questions . . . 3
1.3 Limitations . . . 3

2 Background **5**
2.1 Compiler Warnings and Errors . . . 5
2.2 Code Quality . . . 5
2.3 Bayesian Data Analysis . . . 6
2.4 Causal Inference . . . 6
2.5 Related Work . . . 7
2.5.1 Compiler Warnings . . . 7
2.5.2 Compiler Errors . . . 8
2.5.3 Static Code Analysis . . . 9
2.5.4 Summary . . . 10

3 Method **13**
3.1 Data Collection . . . 13
3.1.1 Sampling Strategy . . . 14
3.1.2 Exclusion Criteria . . . 15
3.2 Categorization of Compiler Warning Usage . . . 15
3.2.1 Classification of Warnings Across Compilers . . . 15
3.2.2 Compiler Warning Categories . . . 16
3.3 Code Quality Analysis . . . 17
3.3.1 Code Quality Metrics . . . 18
3.4 Handling of Recent Warning Usage Changes . . . 19
3.5 Causal Analysis . . . 20
3.6 Model Design . . . 20
3.6.1 Prior Selection . . . 21
3.6.2 Model Template . . . 21

4 Results **23**
4.1 Collected Data . . . 23
4.2 Analysis . . . 26
4.2.1 Bugs . . . 26
4.2.2 Code Smells . . . 28
4.2.3 Critical Violations . . . 32
4.2.4 Major Violations . . . 34
4.2.5 Minor Violations . . . 38
4.2.6 Security Hotspots . . . 40
4.2.7 Vulnerabilities . . . 42
4.2.8 Cyclomatic Complexity . . . 44
4.2.9 Cognitive Complexity . . . 46
4.2.10 Duplicated Lines . . . 48
4.2.11 Technical Debt Ratio . . . 50
4.3 Summary . . . 54

5 Discussion **55**
5.1 Threats to Validity . . . 57
5.1.1 Conclusion Validity . . . 57
5.1.2 Internal Validity . . . 58
5.1.3 Construct Validity . . . 59
5.1.4 External Validity . . . 59
5.2 Future Research . . . 60

6 Conclusion **63**

Bibliography **65**

A Resources **I**

B Included Projects **III**
B.1 Category N . . . III
B.2 Category A . . . IV
B.3 Category AE . . . IV
B.4 Category AP . . . V
B.5 Category AEP . . . V
B.6 Category AE+ . . . V
B.7 Category AEP+ . . . V
B.8 Category AP+ . . . VI
B.9 Category E+ . . . VI
B.10 Category A+ . . . VI
B.11 Category E . . . VI

C Excluded projects **VII**
C.1 Non-CMake projects . . . VII
C.2 Non-compilable projects . . . VIII
C.3 Projects with non-English documentation . . . VIII

C.4 Exercises, tutorials, demos, courses IX
C.5 Miscellaneous IX

D Data Summary **XI**
D.1 Bugs XI
D.2 Code Smells XI
D.3 Critical Violations XII
D.4 Major Violations XIII
D.5 Minor Violations XIV
D.6 Security Hotspots XV
D.7 Vulnerabilities XVI
D.8 Cyclomatic Complexity XVII
D.9 Cognitive Complexity XVIII
D.10 Duplicated Lines XIX
D.11 Technical Debt Ratio XX

List of Figures

3.1 Overview of the methodology applied in the study. 13
3.2 The DAG that encodes the causal assumptions regarding parameters considered in this study. 20
3.3 Template for all statistical model definitions. $\hat{N}_Y$ is the standardized outcome, and $\hat{X}_n$ refers to a standardized parameter. $\alpha_{[i]}$ represents an intercept for each of the warning categories. 21

4.1 Distribution of samples across all warning categories. Some categories are too small to be representative, so a subset of the categories is used to make statistical deductions. These are referred to as the "major" categories, featuring N, A, AE, AEP, and AEP+. 24
4.2 Mean lines of code per project for each major category. 25
4.3 Mean age in days of projects in each major category. Projects in stricter warning categories tend to be older. 25
4.4 Mean number of stargazers of projects in each major category. 26
4.5 Credible intervals of B/KLOC (standardized) for each major category, from model B1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. . 28
4.6 Credible intervals of B/KLOC (standardized) for projects that use any warnings and projects that use none, from model B1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. The projects that use warnings generally feature lower bug scores than projects that use no warnings. 28
4.7 Credible intervals of CS/KLOC (standardized) for each major category, from model CS6 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. . 30
4.8 Credible intervals of CS/KLOC (standardized) for each major category, from model CS6 using a sample that excludes `clipp` and `hana`. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 31
4.9 Credible intervals of CS/KLOC (standardized) for projects that use any warnings and projects that use none, from model CS6 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 32

4.10 Credible intervals of CS/KLOC (standardized) for projects that use any warnings and projects that use none, from model CS6 using a sample that excludes `clipp` and `hana`. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 32

4.11 Credible intervals of CV/KLOC (standardized) for each major category, from model CV1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 34

4.12 Credible intervals of CV/KLOC (standardized) for projects that use any warnings and projects that use none, from model CV1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. Projects that do use warnings feature fewer CV/KLOC. 34

4.13 Credible intervals of MaV/KLOC (standardized) for each major category, from model MAV5 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 36

4.14 Credible intervals of MaV/KLOC (standardized) for projects that use any warnings and projects that use none, from model MAV5 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 36

4.15 Credible intervals of MaV/KLOC (standardized) for each major category, obtained from model MAV5 using a sample that excludes `clipp`. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 37

4.16 Credible intervals of MaV/KLOC (standardized) for projects that use any warnings and projects that use none, obtained from model MAV5 using a sample that excludes `clipp`. The inner and outer regions represent 50% and 90% of the probability mass, respectively. . 38

4.17 Credible intervals of MiV/KLOC (standardized) for each major category, from model MIV1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 39

4.18 Credible intervals of MiV/KLOC (standardized) for projects that use any warnings and those that use none, from model MIV1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 40

4.19 Credible intervals of SH/KLOC (standardized) for each major category, from model S2 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. . 41

4.20 Credible intervals of SH/KLOC (standardized) for projects that use warnings and those that do not, from model S2 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 42

4.21 Credible intervals of V/KLOC (standardized) for each major category, from model V1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. . 43
4.22 Credible intervals of V/KLOC (standardized) for projects that use any warnings and projects that use none, from model V1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 44
4.23 Credible intervals of Cyc/KLOC (standardized) for each major category, from model CYC7 using a sample that excludes `better-enums`. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 46
4.24 Credible intervals of Cyc/KLOC (standardized) for projects that use any warnings and projects that use none, obtained from model CYC7 using a sample that excludes `better-enums`. The inner and outer regions represent 50% and 90% of the probability mass, respectively. . 46
4.25 Credible intervals of Cog/KLOC (standardized) for each major category, from model COG5 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 48
4.26 Credible intervals of Cog/KLOC (standardized) for projects that use any warnings and those that use none, from model COG5 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 48
4.27 Credible intervals of DL/KLOC (standardized) for each major category, from model D6 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. . 50
4.28 Credible intervals of DL/KLOC (standardized) for projects using warnings and those that do not, obtained from model D6 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 50
4.29 Credible intervals of TDR for each major category, from model TD1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 52
4.30 Credible intervals of TDR for projects that use warnings and those that do not, obtained from model TD1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 52
4.31 Credible intervals of TDR for each major category, from model TD1 using a sample that excludes `clipp`. The inner and outer regions represent 50% and 90% of the probability mass, respectively. 53
4.32 Credible intervals of TDR for projects that use warnings and those that do not, obtained from model TD1 using a sample that excludes `clipp`. The inner and outer regions represent 50% and 90% of the probability mass, respectively.. 54

D.1 Overview of bugs per KLOC for each sample in the major categories. XI

D.2 Overview of code smells per KLOC for each sample in the major categories. . . . XII
D.3 Overview of critical violations per KLOC for each sample in the major categories. . . . XIII
D.4 Overview of major violations per KLOC for each sample in the major categories. . . . XIV
D.5 Overview of minor violations per KLOC for each sample in the major categories. . . . XV
D.6 Overview of security hotspots per KLOC for each sample in the major categories. . . . XVI
D.7 Overview of vulnerabilities per KLOC for each sample in the major categories. . . . XVII
D.8 Overview of cyclomatic complexity per KLOC for each sample in the major categories. . . . XVIII
D.9 Overview of cognitive complexity per KLOC for each sample in the major categories. . . . XIX
D.10 Overview of duplicated lines per KLOC for each sample in the major categories. . . . XX
D.11 Overview of the technical debt ratio for each sample in the major categories. . . . XXI

List of Tables

3.1 Overview of assumptions regarding equivalent warning flags in terms of severity for the GCC, Clang, and MSVC compilers. 16

3.2 Overview of translation strategies for compiler-specific warning flags. 16

3.3 List of warning categories established from the samples. The index column contains the numerical identifiers associated with each category. The indices are used extensively in the source code for the statistical analysis. 17

3.4 Overview of recent compiler warning flag changes in sampled projects, including the Git commits that introduced the changes. 19

4.1 Summary of available metrics for the samples. The acronyms assigned to the different code quality metrics are provided in parentheses. . . . 23

4.2 Summary of code quality metrics scaled according to the code size of project samples. The TDR metric is not included since it is already adjusted according to code size. 24

4.3 Overview of candidate models for the bug metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. B1, the model with no parameters, is the model that produces the best predictions. 27

4.4 Summary of results for major categories, from model B1 using the complete sample. Category N has the worst mean at 0.97 B/KLOC, and category AEP has the best mean at 0.37 B/KLOC. 27

4.5 Overview of candidate models for the code smell metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. CS6 is the model that produces the best predictions and is therefore utilized in further analysis steps. 29

4.6 Summary of results for major categories, from model CS6 using the complete sample. Category AEP has the worst mean at 87.95 CS/KLOC, and category AEP+ has the best mean at 54.31 CS/KLOC. . . 29

4.7 Summary of results for major categories, from model CS6 using a sample that excludes `clipp` and `hana`. Category N has the worst mean at 73.83 CS/KLOC, and category AEP+ has the best mean at 44.20 CS/KLOC. 31

4.8 Overview of candidate models for the critical violation metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. CV1 is the model that produces the best predictions and is subsequently used for further analysis. 33
4.9 Summary of results for major categories, from model CV1 using the complete sample. Category N has the worst mean at 23.97 CV/KLOC, and category AEP+ has the best mean at 11.89 CV/KLOC. . . 33
4.10 Overview of candidate models for the major violation metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. MAV5 produces the best predictions and is used in subsequent analysis steps. 35
4.11 Summary of results for major categories, from model MAV5 using the complete sample. Category AEP has the worst mean at 52.49 MaV/KLOC, and category AEP+ has the best mean at 17.90 MaV/KLOC. An outlier is likely causing the poor results for category AEP. . 35
4.12 Summary of results for major categories, from model MAV5 using a sample excluding `clipp`. Category A now has the worst mean at 29.48 MaV/KLOC, and category AEP+ still has the best mean of 16.53 MaV/KLOC. 37
4.13 Overview of candidate models for the minor violation metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. MIV2 and MIV1 perform very similarly, so MIV1 is used in further analysis despite not having the highest ELPD score. 38
4.14 Summary of results for major categories, from model MIV1 using the complete sample. Category N has the worst mean at 24.10 MiV/KLOC, and category AEP+ has the best mean at 16.74 MiV/KLOC. . 39
4.15 Overview of candidate models for the security hotspot metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. S2 is the model that produces the best predictions and is used in further analysis. 40
4.16 Summary of results for major categories, from model S2 using the complete sample. Category N has the worst mean at 0.97 SH/KLOC, while category AE has the best mean at 0.48 SH/KLOC. 41
4.17 Overview of candidate models for the vulnerability metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. V1 has the greatest predictive accuracy and is used in additional analysis. 42
4.18 Summary of results for all categories, from model V1 using the complete sample. Category A has the highest mean at 0.022476 V/KLOC, and category AEP+ has the lowest mean at 0.000951 V/KLOC. . . 43
4.19 Overview of candidate models for the cyclomatic complexity metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. CYC7 marginally outperforms CYC4 and is therefore used in subsequent analysis. 44

4.20 Summary of results for major categories, from model CYC7 using the complete sample. N has the best mean at 137.50 Cyc/KLOC, while A has the worst estimated mean at 178.42 Cyc/KLOC. 45
4.21 Summary of results for major categories, from model CYC7 using a sample that excludes `better-enums`. Category A remains the worst category at 177.50 Cyc/KLOC, while AEP+ becomes the best category at 132.80 Cyc/KLOC, marginally better than N with 138.31 Cyc/KLOC. 45
4.22 Overview of candidate models for the cognitive complexity metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. COG5 provides the best predictions and is therefore chosen for use in subsequent analysis. 47
4.23 Summary of results for major categories, from model COG5 using the complete sample. AEP has the worst estimate at 141.08 Cog/KLOC, closely followed by A at 139.07 Cog/KLOC. AEP+ has the best mean at 75.95 Cog/KLOC. 47
4.24 Overview of candidate models for duplicated lines. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. D6 has the best ELPD score and is selected for further analysis. 49
4.25 Summary of results for all categories, from model D6 using the complete sample. Category N has the highest estimated mean at 116.38 DL/KLOC, while category AEP+ has the best estimate at 58.28 DL/KLOC. 49
4.26 Overview of candidate models for the TDR metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. 51
4.27 Summary of results for major categories, from model TD1 using the complete sample. Categories N and AEP+ have the worst and best means, with TDR scores of 2.16 and 1.40, respectively. 51
4.28 Summary of results for major categories, from model TD1 using a sample that excludes `clipp`. AEP+ and N are respectively the best and worst categories. 53
4.29 Summary of mean metric results, listing the best and worst out of the major categories. Metrics that have been analyzed using both the full and a culled sample have both cases are provided in the table. In general, AEP+ and N are the best and worst categories, respectively. 54

1
Introduction

Modern compilers for programming languages such as C and C++ can diagnose suspicious code constructs at compile-time through *compiler warnings*. Compiler-issued warnings serve as an early aid for developers to avoid subtle but potentially serious mistakes that could otherwise have severe consequences. However, most warnings are disabled by default in modern compilers, such as GCC [1] and Clang [2], and must be explicitly enabled by specifying warning flags.

While compilers can catch both trivial and non-trivial issues and warn about them at compile-time, these warnings are by definition not errors. Therefore, the programmer is not required to address the warnings to compile their program. Unfortunately, syntactically valid code is not necessarily semantically correct (otherwise, there would be no bugs in compiled code). In other words, just because you can write something does not mean you should. As a result, codebases that do not use compiler warnings may feature syntactically valid code that suffers from subtle semantical issues that compilers could otherwise diagnose.

Some programmers are enthusiastic about compiler warnings as a tool to prevent bugs, while others are less convinced of their usefulness [3]. This study aims to provide evidence for practicing software developers to use as a basis to motivate for or against the use of compiler warnings in their projects. Additionally, researchers may use this study as a baseline for further research in the compiler warning field.

1.1 Motivating Example

C++ is a high-level programming language. However, it notably affords significantly more freedom (and responsibility) to the programmer in contrast to other high-level languages. In other words, C++ is a high-level language subject to low-level intricacies, such as manual memory management. This opens up a wide array of possible mistakes and pitfalls.

Undefined behavior (UB) is something that all C++ programmers have to avoid when writing programs. That is, violating the rules of the language such that the output of a program becomes undefined. A program exhibiting UB is free to do whatever it wants. It might run as expected, corrupt your memory, or do nothing at all. Unfortunately, accidentally invoking UB is not particularly difficult. The following

trivial C++ program exhibits UB.

```
1 int main() {
2   bool b;
3   if (b) {
4     // Branch 1
5   } else {
6     // Branch 2
7   }
8 }
```

Readers familiar with other imperative languages may expect `b` to be default-initialized to `false` and therefore assume that branch 2 will be executed. This is incorrect. UB is triggered on line 3 when `b` is evaluated. C++ scalars are *uninitialized* by default, i.e., they have undefined values. Reading uninitialized memory is forbidden and UB.

Compiling the above program with Clang 15, a popular C++ compiler, can be done with `clang++ ub.cpp`, where `ub.cpp` is the source file containing the above program. No warnings will be emitted. To compile with some warnings enabled, we can instead use `clang++ -Wall ub.cpp`. Which will result in the following warning message being issued.

```
1 ub.cpp:4:7: warning: variable 'b' is uninitialized when used here
    [-Wuninitialized]
2   if (b) {}
3       ^
4 ub.cpp:3:9: note: initialize the variable 'b' to silence this
    warning
5   bool b;
6         ^
7          = false
8 1 warning generated.
```

The compiler explains that `b` is uninitialized when used and provides a solution to silence the warning. A version of the program free from UB is listed below, where `b` is initialized to `false`, leading to the expected behavior.

```
1 int main() {
2   bool b = false;
3   if (b) {
4     // Branch 1
5   } else {
6     // Branch 2
7   }
8 }
```

This demonstrates the fact that even simple and seemingly innocuous pieces of code may hide subtle issues. As shown, modern compilers can detect and diagnose many of these issues during compilation. Therefore, it is of interest to obtain empirical data on the impact of compiler warnings regarding code quality.

1.2 Purpose and Research Questions

The purpose of this study is to investigate the relationship between the usage of compiler warnings and overall code quality in C++ projects. A common assumption is that the use of strict compiler warnings should have a positive effect on code quality. However, there is surprisingly little published research to back this notion [4]. This study aims to provide additional empirical evidence to expand on previous research in this neglected area of study.

This study subsequently aims to answer the following research questions. The sample of C++ projects needs to be categorized according to their usage of compiler warnings to provide insight into the typical warning configurations. Some examples of measurable code quality metrics include code smells, likely bugs, and cognitive complexity. In RQ3, the *optimal* compiler warnings are those that lead to the greatest overall improvements in code quality, if any.

RQ1: To what extent are compiler warnings used in C++ projects?
RQ2: Does compiler warning usage correlate with C++ code quality?
RQ3: What is the optimal level of compiler warnings in C++ projects?

1.3 Limitations

The study only considers public C++ projects hosted on GitHub that use the CMake build system. Only CMake projects are considered to simplify code quality analysis using SonarCloud. The build system limitation does exclude some sample candidates, although CMake is the de facto standard build system for C++ projects.

Due to the manual work necessary to reliably categorize projects in relation to used compiler warning flags, the overall sample size is constrained in regard to the limited time available to conduct the study. A total of 127 projects are analyzed in this study, which is considerably more than what has been analyzed in previous research [5].

There is essentially an unlimited amount of combinations of compiler warning flags. As such, common warning flags are grouped and generalized. For example, flags that enable an array of warnings, e.g., `-Wall` or `-Wextra`, are very common on GCC and Clang. Whereas flags that control specific warnings are less common. To enable analysis of the collected data, the study focuses on the aforementioned grouped warning flags.

2
Background

This chapter contains some necessary background information that could help with understanding the rest of the study.

2.1 Compiler Warnings and Errors

Warnings issued by compilers, *compiler warnings*, are emitted by C++ compilers for encountered suspicious code constructs. However, special arguments are often required to enable these kinds of warnings by providing various *warning flags* to the compiler upon compilation. While some warnings are outlined in the C++ standard, compiler vendors generally provide compiler warnings as they see fit. In contrast, *compiler errors* occur due to ill-formed code, i.e., illegal code that does not satisfy the C++ grammar. While the error messages vary in presentation, compiler implementations should, in theory, accept and reject code identically. Compiler warnings are not only emitted by C++ compilers. For example, Rust [6], a modern language similar to C++, makes use of extensive compiler warnings.

Linters and static code analysis tools are able to inspect source code and accordingly provide warnings. However, these warnings differ from those emitted from compilers during compilation. Such tools are typically used to obtain insights into quality, style, and security aspects. Examples of popular linters and static code analysis tools for C++ code include *clang-tidy* [7] and SonarQube [8].

2.2 Code Quality

Software quality, referred to as *code quality* in this report, and related metrics are defined in the ISO 25010:2011 standard [9], which defines software quality as the "*degree to which a software product satisfies stated and implied needs when used under specified conditions*". Furthermore, ISO divides code quality characteristics into distinct categories, such as *internal quality*, *external quality*, and *quality in use*. Internal quality is concerned with the static source code and associated artifacts. Examples of internal quality characteristics are maintainability and portability. External quality relates to the runtime characteristics of the software, such as reliability and performance efficiency. Lastly, quality in use relates to the ability of users to utilize software to achieve their goals. An example of a quality-in-use characteristic is effectiveness, i.e., the level of accuracy for users in accomplishing their tasks.

This study focuses on product quality characteristics, especially those related to internal quality. Relevant characteristics include maintainability, reliability, and security. Metrics such as the number of bugs, code smells, and security vulnerabilities are used to assess these characteristics. Estimates for these metrics are obtained using the static code analysis tool SonarCloud by SonarSource, which uses the same analysis engine as SonarQube [10]. SonarCloud is used instead of SonarQube since it provides free analysis support for public projects hosted on GitHub. The SonarCloud analysis was invoked using GitHub Actions [11]. All build scripts are publicly available in the sample projects, provided as forks of a GitHub organization associated with this report (see Appendix A).

2.3 Bayesian Data Analysis

Bayesian data analysis (BDA) is an alternative approach to statistical analysis instead of the traditional Frequentist approach. While Frequentist methods are more common in scientific research, Bayesian methods are growing more popular [12]. Both approaches can be used to achieve similar results but are derived from different fundamental views on statistics. Bayesian analysis is essentially based on counting possibilities and updating existing beliefs based on observed data, while Frequentist methods are based on the supposed frequency of events given sufficiently large samples [12]. The process of updating existing beliefs through observed data is referred to as Bayesian updating [13]. The process involves determining *priors* before any data is collected, which encodes prior beliefs and assumptions regarding the considered population. These beliefs are subsequently updated by collecting and observing samples from the population which gradually improves the beliefs to match the data. The mechanism used to shift prior beliefs is Bayes' theorem, hence the term *Bayesian* data analysis. Some of the advantages of using BDA include the explicitness regarding assumptions and that posterior distributions are easier to understand than the notoriously unintuitive p-values typically used in Frequentist methods [14].

2.4 Causal Inference

Causal analysis is the practice of making sense of data [15]. More specifically, it is about determining the *cause* for some effect. Causation is difficult to account for. Failure to properly handle causation can lead to contradictory results, such as in Simpson's paradox, where the addition of conditioned parameters changes the outcome completely. An important aspect of causal inference is that correlation on its own does not imply causation. For example, the number of Waffle House diners per person in US states is correlated with the divorce rate [12], but few would argue that it is the Waffle House diners that cause the elevated divorce rate. One approach to explicitly state assumptions regarding causation is to utilize directed acyclic graphs (DAGs) to indicate causal relationships between variables under consideration [15]. This study subsequently proposes a causal DAG as a basis for all causal reasoning regarding the results.

2.5 Related Work

This section discusses related research in the area of compiler warnings. The relationship between compiler warnings and code quality has seen limited attention in research, but a decent amount of research has been conducted on closely related topics, e.g., compiler errors, code quality, and static code analysis.

2.5.1 Compiler Warnings

According to Kudrjavets et al. [4], there is a need for further research into the effectiveness of compiler warnings, and their impact on code quality in particular. A general view held by many software developers is that compiler warnings are useful for detecting suspicious code constructs, so their use should subsequently improve code quality. However, this view is surprisingly unsubstantiated in research. The authors stress a lack of empirical evidence as one of the primary reasons for compiler warnings being somewhat neglected by some developers, unconvinced of their utility. Notably, the authors themselves mention their industry experience as a basis for their feeling that compiler warnings are indeed useful. Furthermore, the perception that *if warnings were an issue, they would be errors*, is highlighted as a cause for the reluctance of some developers to care about compiler warnings. The paper additionally calls for further research into the relationship between compiler warnings and code quality.

A study published by Moser et al. [5] in 2006 found a positive correlation between the density of compiler warnings and likely code defects. Five closed-source projects from the telecommunication sector were evaluated, in which files with frequent changes are assumed to be subject to a greater risk of code defects. The authors conclude that source files with a large number of compiler warnings should be prioritized for testing, along with the However, the limited sample size limits the generalizability of the findings, necessitating further research. Interestingly, the authors remark on the limited research into the efficacy of compiler warnings, similar to Kudrjavets et al. [4] in their paper from 2022, 16 years later. Measuring quality scores as density metrics, e.g., concerning lines of code, is mentioned as an important aspect to consider in future research. Additionally, the correlation between specific compiler warnings and different code defect types is underlined as of interest for future research.

Danilova et al. [3] conducted a Grounded Theory study to investigate developer preferences regarding the presentation of various kinds of warnings and errors. Professional developers, students, and researchers were included in a focus group amounting to 33 people. Warning markers, i.e., especially highlighted code to signify potential problems, were found to be the most preferred presentation form. While compiler warnings were found to be preferred over pop-ups and warning-on-commit schemes. However, the authors underline that there was no clear winner out of the evaluated warning types. One notable finding is that 56% and 64% of the participants answered (to a multiple choice question) that they preferred to see warnings

before running the program, or before committing their changes, respectively. Additionally, 60% indicated that they would like to see warnings on demand. Another finding was that more experienced developers generally seem more predisposed to positively rate all warning types, except for pop-ups. In general, the authors conclude that there are clear differences in the preferred method of presenting warnings and that configurability is very important for developers when it comes to the presentation of warnings.

Compilers are software and, just like all other software, contain bugs. The topic of erroneous compiler warnings has been explored by Tang et al. [16], who designed a tool for detecting defective compiler warnings. The group found 8 compiler warning defects in GCC and Clang after running their tool for 2 months. Similarly, Sun et al. [17] proposed a randomized differential testing technique designed to reveal erroneous compiler warnings when using the C programming language. Their technique was able to detect a total of 99 bugs in GCC and Clang over 6 months. Sun et al. emphasize the historical lack of attention paid to verifying the accuracy of compiler diagnostics. Both Tang et al. and Sun et al. denote the importance of compiler warnings in aiding programmers to write better code, but warnings must be accurate so that development time is not wasted. The fact that compilers emit spurious warnings occasionally could hurt the trust of programmers that have to deal with them. Especially given the aforementioned lack of empirical evidence on the correlation between compiler warnings and code quality, as outlined by Kudrjavets et al. [4].

Melo et al. [18] studied the variability of emitted compiler warnings in different configurations of the Linux operating system (which is written in C). The authors used GCC to compile 21,030 distinct configurations of Linux, using the `-Wall` flag. Interestingly, the authors explain that `-Wall` was used since it enables *all* warnings. However, this is inaccurate, there are far more warnings than those enabled by `-Wall`, but it is not surprising that the authors were confused by the misleading name of the flag. Nevertheless, the authors determined that the most common warnings involved dead code and uninitialized memory. Another rather surprising finding was that no warnings were found to be configuration independent, i.e., no warning types were found in every evaluated configuration. Since C++ features the same preprocessor functionality as C (such as `#ifdef` and `#define`), the same configuration variability likely applies to C++ projects. The study did not include any form of code quality analysis, opting instead to use compiler warnings as a proxy for code with suspected quality issues.

2.5.2 Compiler Errors

Barik et al. [19] investigated whether developers even read error messages from compilers. The authors conducted an eye-tracking study in which 56 university students solved different errors in a Java code base. The results showed that the participants spent 13–25% of their "fixations", i.e., gazes at a specific location, reading the compiler error messages. The authors determined that developers try to read compiler error messages but find reading the error messages more challenging than

the source code. Therefore, Barik et al. conclude that there is substantial room for improvement in the presentation of compiler error messages. It is reasonable to suspect that the same applies to compiler warning messages.

A problem with compiler warnings and error messages is that they are sometimes perceived as cryptic and too generic, limiting their usefulness. Barik et al. [20] conducted a comparative evaluation study with professional developers to assess the understandability of compiler messages, focusing on Java (OpenJDK) compiler error messages. The authors found that, in general, developers prefer compiler error messages phrased as natural explanations over terse and technical messages without any explanations. However, the study also found that developers will prefer a less explanatory error message if it includes a solution to the problem that caused the message in the first place. Notably, the compiler warning example in Section 1.1 does provide a resolution to the cause of the problem, indicating that this is a clear and understandable message. Great care must be applied when formulating compiler warning messages to guarantee clarity and usefulness for software developers.

2.5.3 Static Code Analysis

Marcilio et al. [21] investigated whether issues reported by static analysis tools are addressed. A survey was conducted to determine the perception of static code analysis tools amongst developers. In total, 18 team leaders from the sampled projects answered the survey. A clear majority, 80%, of the respondents indicated that they feel that static code analysis tools are useful for improving code quality in their projects. The authors subsequently analyzed a total of 421,976 issues from 246 projects, obtained using the static code analysis tool SonarQube. Some of the most notable findings include that the average resolution rate of issues in the sampled projects was merely 13%, and the median time taken for a fix of a single issue was 19 days. Additionally, the authors found that the industrial projects took longer to implement fixes to issues compared to the open-source projects in their sample. The authors believe that only a subset of the default rules included in static code analysis tools is indicative of actual issues in code. It is concluded by the authors that static code analysis tools, such as SonarQube, are beneficial for the code quality of projects, given that they are configured in such a way that the included rules are considered relevant.

Ziang et al. [22] studied the effectiveness of static code analyzers for the Scala programming language. The authors devised a method for evaluating Scala static code analysis warnings. The study included six open-source projects, on which 115 checker rules were evaluated. A total of 191,000 warnings were emitted, of which the study concludes that 154,000 are false positives. In other words, only a fifth of the emitted warnings were deemed accurate. While Scala and C++ are drastically different languages, a non-negligible amount of warnings emitted from static code analysis tools, unfortunately, appear likely to be false positives, as indicated by Marcilio et al. [21].

Joshy et al. [23] devised a tool that automatically creates reproducible test cases based on code snippets included in warnings from static analysis tools to validate their credibility. The aim was to provide an easy way for developers to recognize which warnings from their static analysis tools are worth addressing. The authors reported a successful compilation rate of 68.5% for their automatically generated test cases. The tool was able to detect both confirmed and likely false positives. While warnings from static analysis tools and compilers differ, it would be interesting to see similar tools for warnings emitted by compilers since developers often find such warnings hard to decipher [20].

Lenarduzzi et al. [24] conducted an empirical study including 21 Java projects hosted on GitHub to evaluate the accuracy of SonarQube measures. The authors found that issues listed as "bugs" by SonarQube were rarely indicative of faulty code, while issues marked as code smells were much more likely to be indicative of such code. The authors argue that violations issued by SonarQube are effective predictors of suspicious code if evaluated collectively. Additionally, the reported violation severity was not found to be correlated to fault-proneness. That is, critical, major, and minor violations did not differ in this regard. The authors concluded that projects should customize the analysis rules according to their needs to reduce false positives, akin to the conclusion of Marcilio et al. [21].

2.5.4 Summary

The presence of emitted compiler warnings is generally assumed to correlate with code of lower quality [4, 5, 16, 18]. However, very little research has been conducted to support the soundness of this assumption [4, 5]. The lack of research forms the basis for the motivation behind this study, aiming to answer this question (see RQ2). Furthermore, the study aims to collect enough evidence to provide a sound foundation for decisions made by developers regarding their potential utilization of compiler warnings in C++ projects. The absence of such a foundation has been problematic when arguing for compiler warning usage [4]. This study uses metrics scaled according to the size of sampled projects, as outlined by Moser et al. [5].

While developers have differing opinions on the presentation of warnings [3], most developers appear interested in being notified of potential code issues before committing their code. However, compiler warnings are imperfect and can be misleading [23] and poorly formulated [20]. Despite these considerations, compiler warnings are still generally regarded as a positive influence on code quality. Therefore, any correlation between compiler warning usage and code quality could likely be enhanced through relatively simple measures.

This study uses SonarQube to estimate the code quality of C++ projects. Static code analysis tools such as SonarQube are imperfect, but developers generally perceive such tools as good indicators for problematic code [21]. While only a subset of issues highlighted by tools such as SonarQube seems worthy of developer consideration [24,

21, 23], the tools can detect authentic code quality issues in code bases. These tools are subsequently practical for use in real projects. As such, the usefulness of the obtained code quality metric estimates outweighs the risk of occasional false positives, forming the basis of code quality analysis in this study.

3

Method

This chapter discusses the research methodology utilized in the study. An overview of the overarching research steps is provided in Figure 3.1. The individual steps are discussed in further detail in the following sections.

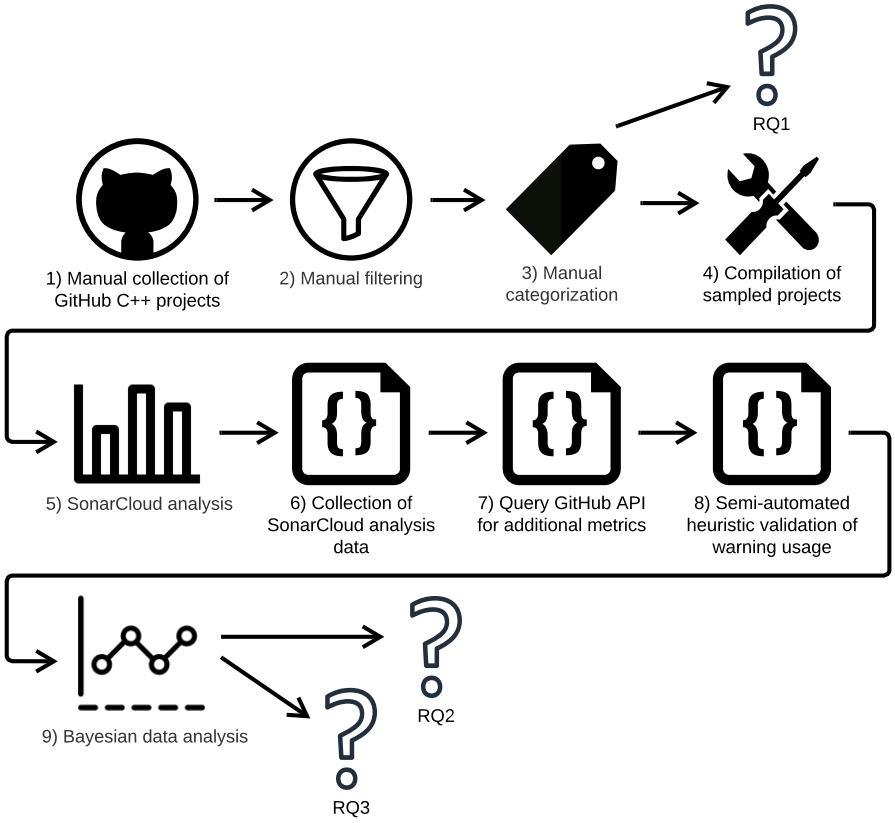

Figure 3.1: Overview of the methodology applied in the study.

3.1 Data Collection

This section discusses the overall data collection strategy enforced to assemble the sample of C++ projects. In total, 127 projects were collected.

3.1.1 Sampling Strategy

This study is a sample study that was conducted using a *purposive sampling* [25] approach to collect C++ projects. This means that the sample projects are explicitly selected according to some predefined criteria and not randomly. The population considered in this study is all C++ projects. While this approach means that the sampling process will be inherently subjective to some degree, it also offers greater opportunity for the selection of projects that are considered representative of *useful* C++ projects, A definition of a useful C++ project is naturally difficult to provide. Essentially, the study is interested in C++ projects that are used to some extent to solve some problem and provides some tangible value, i.e., projects that have working functionality.

A probabilistic sampling strategy, such as randomly selecting N C++ projects hosted on GitHub, is possible. Recent research by Pina et al. [26] has aimed to automate the process of mining GitHub projects and subsequently analyze them using SonarQube, demonstrating the plausibility. However, a stochastic sampling strategy does not necessarily imply a more representative sample [25], but it is undoubtedly easier to argue that the samples should be representative (given sufficient sample size). The primary obstacle to using a stochastic sampling strategy in this study is the difficulty in ensuring that the collected projects are of sufficient maturity. For example, an abandoned "Hello, world" project of a few lines of code is hardly representative. Additionally, the categorization and exclusion of samples is a manual process, meaning that collecting a sufficient sample size required for a random sampling approach is infeasible.

Another issue with an automated sampling and categorization approach for C++ projects is the fact that there is no standard C++ project structure. This makes it very difficult to algorithmically process projects reliably. Additionally, CMake uses a custom scripting language, so extracting compiler warnings from CMake scripts dependably would essentially require writing a tool capable of parsing CMake script code. This necessitates the suboptimal process of evaluating and categorizing C++ projects by hand. For other, more opinionated languages with more regular organization schemes, this kind of process might prove significantly more feasible.

Sample studies are inherently unobtrusive, i.e., the process of conducting the study itself does not affect the observed variables, although the precision of any measurements is subsequently somewhat limited [27]. Furthermore, sample studies allow for maximum generalizability for the target population. The chosen approach prevents conclusions regarding any causal relationships [27], so this study is primarily interested in the *correlation* between the usage of compiler warnings and code quality. The aim is that the results shall apply to most C++ projects.

The sampling strategy consisted of going through C++ projects on GitHub, by navigating to the C++ topic and specifying C++ as the language. Subsequently, the

projects were sorted by descending amount of stargazers[1]. A manual approach was then applied, where projects in the list were checked for validity one-by-one, in a top-down fashion, enforcing predefined exclusion criteria which are described in a later section. This process corresponds to steps 1 and 2 in Figure 3.1. The deciding factor for the sample size was the available time allocated for the data collection phase, consisting of approximately three weeks. The data collection process was conducted in February 2023.

3.1.2 Exclusion Criteria

During the data collection, projects were excluded based on the following exclusion criteria. The exclusion criteria aimed to ensure that the included projects are representative of authentic C++ projects. However, some criteria, such as the requirement to use English documentation or CMake as the build system, stem from the languages spoken by the authors and the used analysis tools, respectively. This is important since all sampled projects had to be built manually, whereas some projects had non-trivial build requirements. All excluded projects are listed in Appendix C.

- Must have documentation in English.
- Must use CMake.
- Must compile on a recent version of Ubuntu.
- Must produce an analyzable binary when compiled.
- Must not be a collection of other projects.
- Must not be a course or exercise material.
- Must not be a demonstration project.
- Must not be a blog.
- Must not be a component of another sampled project.

3.2 Categorization of Compiler Warning Usage

The usage of compiler warning flags differs greatly from project to project. Additionally, compilers have slightly different names and meanings for their warning flags. Flags from the three most popular C++ compilers were considered: MSVC, GCC, and Clang. The classification process has therefore required generalization of used flags and their intention. In general, the study makes use of the compiler warning flag names used by the GCC and Clang compilers. This section discusses the approach taken to define compiler warning categories, and measures taken to accurately reflect the intention behind compiler-specific warnings. The categorization process described in this section corresponds to step 3 in Figure 3.1.

3.2.1 Classification of Warnings Across Compilers

Some projects only specify compiler warning flags for specific compilers or platforms. Therefore, the study makes general assumptions regarding comparable warning lev-

[1]The C++ project list can be viewed here: `https://github.com/topics/cpp?l=c%2B%2B`

els for different platforms. GCC and Clang mostly use compatible warning flag names, while MSVC [28] uses a separate naming convention altogether. The assumptions regarding corresponding warning flags are shown in Table 3.1. For instance, a project using `-Wall` is considered to be in the same category as a project using `/W3`.

GCC/Clang	MSVC
`-Wall`	`/W3`
`-Wextra`	`/W4`
`-Werror`	`/WX`

Table 3.1: Overview of assumptions regarding equivalent warning flags in terms of severity for the GCC, Clang, and MSVC compilers.

Furthermore, some compiler-specific flags have been translated into more general warnings. An overview of these translations is provided in Table 3.2. The motivation for translating these warnings is to avoid excluding information in a manner that would bias the analysis results. For example, a project using `-Weverything` clearly makes use of a lot of compiler warnings. It would be quite misleading to categorize such a project as using no warnings because it does not use any of the common group warning flags. As such, the translations have been designed to preserve the intention behind the use of the original flags as well as possible.

Compiler	Flag	Translation
Clang	`-Weverything`	`-Wall, -Wextra, -Wpedantic, -Werror`
GCC	`-pedantic-errors`	`-Wpedantic`

Table 3.2: Overview of translation strategies for compiler-specific warning flags.

3.2.2 Compiler Warning Categories

Most compilers support individual control of separate compiler warnings. The amount of possible combinations of compiler warning flags is subsequently essentially unlimited. However, some warning flags represent a "family" of warnings, and control the usage of several individual warnings. Common examples include `-Wall` and `-Wextra` for the GCC and Clang compilers. As such, sampled projects have been categorized per their use of such grouped warning flags. Categories were assembled after the data collection was completed, to accurately describe the collected projects. All categories are provided in Table 3.3.

Index	ID	Description
1	N	Uses none of the warning group flags, or no warnings at all.
2	A	Uses `-Wall`.
3	AE	Uses `-Wall, -Wextra`.
4	AP	Uses `-Wall, -Wpedantic`.
5	AEP	Uses `-Wall, -Wextra, -Wpedantic`.
6	AE+	Uses `-Wall, -Wextra, -Werror`.
7	AEP+	Uses `-Wall, -Wextra, -Wpedantic, -Werror`.
8	AP+	Uses `-Wall, -Wpedantic, -Werror`.
9	E+	Uses `-Wextra, -Werror`.
10	A+	Uses `-Wall, -Werror`.
11	E	Uses `-Wextra`.

Table 3.3: List of warning categories established from the samples. The index column contains the numerical identifiers associated with each category. The indices are used extensively in the source code for the statistical analysis.

Warnings enabled by `-Wall` are generally easy to avoid through good practices and cover cases of clearly suspicious code constructs that can be quickly addressed [29]. Examples of code constructs that are discouraged when using `-Wall` include unused functions, tautological comparisons, and infinite recursion [29, 30]. Subsequently, `-Wextra` includes additional warnings not enabled by `-Wall`. These warnings are slightly more cautious and may not be as obvious to resolve. Examples of targeted code constructs include comparisons of integers of different signs, redundant move operations, and unused parameters. Furthermore, the `-Wpedantic` flag enables warnings that generally enforce strict adherence to C++ standards [29]. These warnings are relatively easy to trigger, e.g., superfluous semicolons, use of non-standard extensions, and C compatibility issues are aspects that are targeted. Lastly, `-Werror` promotes all warnings to errors, preventing program compilation unless no warnings are emitted.

In this study, categories N, A, AE, AEP, and AEP+ are hereby referred to as the *major* categories. These categories have decent sample sizes since their warning flag combinations are common. Therefore, these categories provide a more reliable basis for analysis and subsequent conclusions. An overview of the collected data is provided in Appendix D.

3.3 Code Quality Analysis

Analysis of the code quality of sampled projects was conducted using the static analysis tool SonarCloud [31], which formed the foundation for the code quality metric estimations. Examples of important code quality metrics include the number of bugs, code smells, and security vulnerabilities.

In order to be able to analyze the C++ projects, the sampled projects first had to be successfully compiled. This was done using GitHub Actions, with virtual

machines running Ubuntu Linux, following the instructions provided in each sampled project. The build scripts used for the sampled projects can be found in the `.github/workflows` directory in the associated repositories. After successful compilation, SonarCloud analysis was conducted on the resulting binary, and the analysis results were recorded. The process of compiling the sampled projects and running the SonarCloud analysis corresponds to steps 4 and 5 in Figure 3.1.

Collection of the SonarCloud analysis data was performed using the SonarCloud API [32], using the `/api/projects/search` and `/api/measures/component` endpoints. The measures that were included consisted of `bugs`, `code_smells`, `vulnerabilities`, `security_hotspots`, `duplicated_lines`, `complexity`, `cognitive_complexity`, `files`, `major_violations`, `minor_violations`, `ncloc`, and `sqale_debt_ratio`. This resulted in JSON files filled with the code quality analysis data. Step 6 in Figure 3.1 refers to this process.

Additional metrics, such as the number of contributors, stargazers, and project age, were obtained using the GitHub API [33]. This corresponds to step 7 in Figure 3.1. Endpoints that were used included `/repos/USER/REPO`, `/repos/USER/REPO/stargazers`, and `/repos/USER/REPO/contributors`, where `USER` and `REPO` corresponds to the names of a GitHub user (or organization) and repository, respectively. This information was subsequently merged with the SonarCloud analysis data. The replication package contains a CSV file with the corresponding assembled data (see Appendix A).

3.3.1 Code Quality Metrics

The metrics used to assess the code quality of sampled projects cover reliability, maintainability, and security. There are three issue categories, *bugs*, *code smells*, and *vulnerabilities* [34], which correspond to the aforementioned quality characteristics. Bugs are considered code issues that are very likely to lead to erroneous behavior, and as such require immediate attention. Code smells are issues that make the code hard to read or understand, which negatively affects maintainability. The last category, vulnerabilities, are issued in the code that constitutes possible attack vectors for nefarious users.

In addition to vulnerabilities, issues may be marked as *security hotspots*. These are issues that might constitute actual security issues but with less certainty compared to detected vulnerabilities [35]. Developers are expected to review security hotspots manually, and subsequently determine whether the issue is an actual vulnerability that needs fixing.

Furthermore, detected issues are classified according to their estimated severity [34]. The *critical violations*, *major violations*, and *minor violations* metrics measure the number of issues with the corresponding severities. Critical issues are deemed to have a possibility of impacting the observable behavior of a project and must be fixed as soon as possible. Major issues are considered likely to affect overall pro-

ductivity but are not as likely to have externally observable effects. Lastly, minor issues cover more pedantic code problems, which are unlikely to negatively affect the behavior of the program.

While some of the previous metrics are inherently subjective, such as code smells, there are some additional metrics that are slightly easier to define. These are cyclomatic and cognitive complexity, and the amount of duplicated lines. The cyclomatic complexity is a measure of the amount of linearly independent paths in the source code, i.e., it is a measure of the complexity of control flow structures. Subsequently, the cognitive complexity metric is a measure that tries to estimate the difficulty for humans to understand control flow structures, e.g., by taking indentation into account [36].

Additionally, a metric provided by SonarCloud referred to as the "technical debt ratio" (called `sqale_debt_ratio` in the SonarCloud API) is used as an indicator of the combined code quality in projects. It is defined as the technical debt (in minutes) of a project divided by the estimated total project development time [36]. This essentially makes the technical debt ratio an aggregate metric, making it particularly useful for measuring the overall code quality and the effects of using different compiler warnings on the quality of the sampled projects.

3.4 Handling of Recent Warning Usage Changes

The usage of compiler warning flags is not static, and projects will occasionally make changes to their build configurations. As such, it is important to account for possible recent changes to the used compiler warning flags in sampled projects. This is done to accommodate for a delay in the effects of changing the used compiler warning flags, since enabling stricter compiler warnings is unlikely to significantly affect the code quality immediately. For this study, a somewhat generous threshold of 183 days, i.e., half a year, has been used. This means that sample projects that have made changes to their compiler warning flags within 183 days have been categorized according to their previously used flags. An overview of the projects with such recent changes is provided in Table 3.4.

Project	Old Category	New Category	Git Commit
`endless-sky/endless-sky`	AE	AP+	`f5e4503`
`yuzu-emu/yuzu`	AE	AE+	`4966956`
`symforce-org/symforce`	N	AE+	`6d1c918`
`paceholder/nodeeditor`	AE	AE+	`a3969b9`
`KDAB/hotspot`	N	AEP	`432bb14`

Table 3.4: Overview of recent compiler warning flag changes in sampled projects, including the Git commits that introduced the changes.

The process of checking the sampled projects for recent changes to their compiler warning usage was partially automated. A Python script was devised that performed

a regex-based recursive search for compiler warning flags of interest in all CMake scripts in the sampled projects. All files named `CMakeLists.txt`, or those with the `cmake` file extension were considered CMake scripts. Subsequently, the date of the last change to each CMake script was obtained using Git[2]. This information was used to compile a list of projects with changes within the aforementioned threshold period. The recently changed CMake files in these projects were manually checked to determine the appropriate category. Step 8 in Figure 3.1 refers to this process.

3.5 Causal Analysis

A causal DAG has been derived for this study that captures the causal assumptions made regarding relevant parameters, see Figure 3.2. Culture refers to the general engineering practices utilized by developers of sampled projects, which is assumed to affect the likelihood of refactoring existing code with the aim of improved code quality. Subsequently, code defects are assumed to cause refactoring activities when detected. The presence of code defects is also assumed to cause the emission of compiler warnings, which in turn is presumed to lead to further refactoring activities. Lastly, code size is assumed to influence the number of code defects in projects where larger projects are deemed more likely to exhibit code defects. Note, this study does not aim to prove the existence of a causal relationship between the usage of compiler warnings and code quality. This is because the utilized methodology prevents causal conclusions [27]. However, the choice of methodology does not prohibit discussions regarding possible causal relationships.

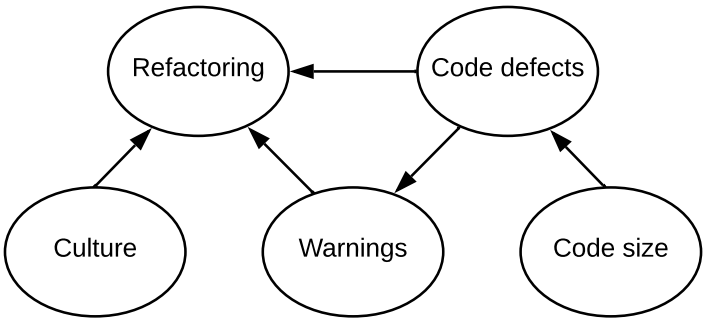

Figure 3.2: The DAG that encodes the causal assumptions regarding parameters considered in this study.

3.6 Model Design

This section discusses the design of all statistical models developed in this study. In this section, a parameter P is *standardized* if it features a "hat" symbol, i.e., $\hat{P}$. A

[2]Obtained using `git log -1 --format=%ai -- FILE`

standardized variable has a mean and standard deviation of 0 and 1, respectively. In each model, $a_{[i]}$ represents an index intercept variable for each warning flag category.

3.6.1 Prior Selection

All candidate models in this study feature standardized parameters along with a standardized outcome. This is due to the difficulty in formulating reasonable ontological and epistemological arguments for priors regarding the variables under consideration. Furthermore, basing priors on the sample is poor practice [12]. With standardized variables, prior selection becomes trivial: mean of 0, standard deviation of 1. This is the primary advantage of standardized variables.

The practice of standardizing parameters has inherent advantages and disadvantages. Some of the advantages include better performance when computing the posterior distributions, due to the simple distribution space for the MCMC engine to explore. This applies to both the time spent and the distribution coverage, leading to better effective sample sizes [12]. One of the main disadvantages of standardized parameters is the fact that they are no longer in outcome space, i.e., they lose their unit. However, the estimates can be converted back into their natural outcome space. This is done in the results chapter for all obtained estimates in order to make the estimates more intuitive.

3.6.2 Model Template

All statistical models used in the study have similar definitions, due to the aforementioned practice of standardizing parameters. The model template is shown in Figure 3.3. All models in this study feature the $\alpha_{[i]}$ intercept, but the β parameters may be omitted. For brevity, subsequent model specifications are provided in terms of the conditioned parameters.

$$
\begin{aligned}
\hat{N}_Y &\sim \text{Normal}(\mu, \sigma) \\
\mu &= \alpha_{[i]} + \beta_{X_0}\hat{X}_0 + ... + \beta_{X_n}\hat{X}_n \\
a_{[i]} &\sim \text{Normal}(0, 1) \\
\beta_{X_n} &\sim \text{Normal}(0, 1) \\
\sigma &\sim \text{Exponential}(1)
\end{aligned}
$$

Figure 3.3: Template for all statistical model definitions. $\hat{N}_Y$ is the standardized outcome, and $\hat{X}_n$ refers to a standardized parameter. $\alpha_{[i]}$ represents an intercept for each of the warning categories.

Due to the limited insight into the procedures that produce the code quality metrics used in the study, Normal distributions are used in the model template for the outcome variable and parameters. This is because the Normal distribution is the

maximum entropy distribution [12] for outcomes that are not counts, which is the case for the standardized outcomes and parameters used in the study. It should be noted that the standardization does not cause the underlying data to be normally distributed. However, the Normal distribution is a reasonable assumption, especially since opting for any other distribution would constitute a stronger assumption, which would be difficult to motivate. The sampling should still work quite well, given the centered values produced by the standardization. It is possible to base the choice of distributions and priors on the observed data, but that is poor practice [12], which is why this is not done in the study.

4
Results

This chapter presents the collected data, model comparisons, and the results from the statistical analysis.

4.1 Collected Data

An overview of the collected data is shown in Table 4.1. All values have been rounded upwards to the closest integer. A list of all sampled projects is provided in Appendix B. Furthermore, distribution plots for the various metrics across the major categories are provided in Appendix D.

Metric	Mean	Median	Minimum	Maximum
Stars	4,252	2,193	986	33,580
Contributors	83	46	1	368
Age (in days)	2,357	2,363	126	5,101
Lines of code	98,350	36,084	1,174	696,194
Files	591	211	5	6,686
Bugs (B)	56	13	0	1,330
Code smells (CS)	5,586	1,962	17	64,256
Critical violations (CV)	1,587	461	5	16,049
Major violations (MaV)	2,129	683	5	36,023
Minor violations (MiV)	1,701	559	1	26,523
Security hotspots (SH)	43	12	0	753
Vulnerabilities (V)	1	0	0	23
Cyclomatic complexity (Cyc)	12,678	4,482	47	90,453
Cognitive complexity (Cog)	10,159	3,191	28	83,575
Duplicated lines (DL)	13,011	2,059	0	202,359
Technical debt ratio (TDR)	1.81	1.6	0	10.8

Table 4.1: Summary of available metrics for the samples. The acronyms assigned to the different code quality metrics are provided in parentheses.

To account for the difference in code size amongst the sampled projects, all metrics are scaled per 1,000 lines of code (KLOC). This makes comparisons among projects fair since the metrics become independent of code size. See Table 4.2 for a scaled version of the earlier metric table with raw metric values. The metrics listed in the table are the ones used when discussing the obtained results in subsequent analysis sections.

Metric	Mean	Median	Minimum	Maximum
B/KLOC	0.67	0.36	0.00	9.17
CS/KLOC	66.37	54.85	0.10	628.43
CV/KLOC	17.02	14.09	0.04	108.90
MaV/KLOC	26.81	20.49	0.05	585.62
MiV/KLOC	20.13	16.54	0.01	101.91
SH/KLOC	0.69	0.28	0.00	5.57
V/KLOC	0.01	0.00	0.00	0.29
Cyc/KLOC	156.30	148.64	0.28	1,187.92
Cog/KLOC	109.52	105.34	0.17	372.87
DL/KLOC	94.59	52.45	0.00	596.87

Table 4.2: Summary of code quality metrics scaled according to the code size of project samples. The TDR metric is not included since it is already adjusted according to code size.

The distribution of samples for each category is presented in Figure 4.1. All of the major categories consist of more than 15 samples. The remaining categories are not deemed large enough for reliable analysis.

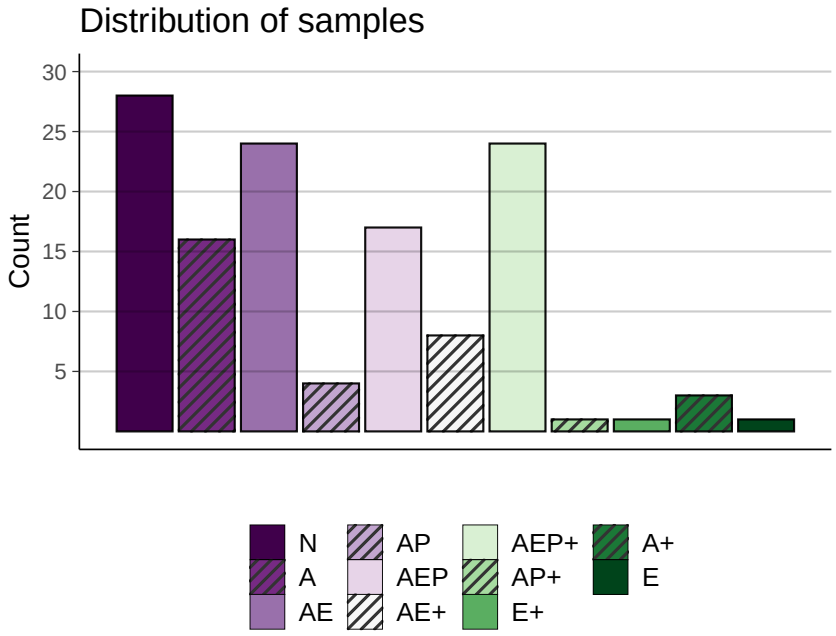

Figure 4.1: Distribution of samples across all warning categories. Some categories are too small to be representative, so a subset of the categories is used to make statistical deductions. These are referred to as the "major" categories, featuring N, A, AE, AEP, and AEP+.

The mean code size across the major categories is presented in Figure 4.2. The distribution exhibits the distinctive bell shape exhibited by Normal distributions. It is interesting that both projects that use no warnings and projects that use the most strict compiler warnings tend to be smaller. This could perhaps be explained

by a perceived notion that AE constitutes a sweet spot regarding maintainability as projects grow larger.

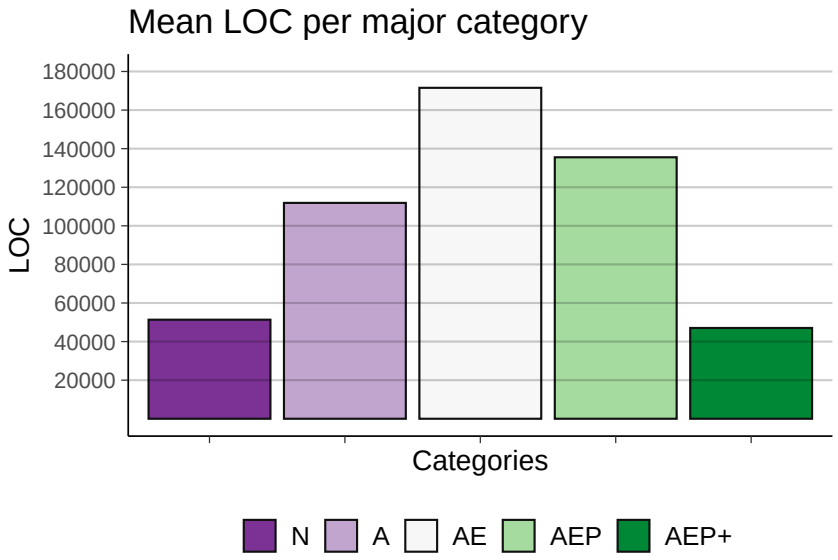

Figure 4.2: Mean lines of code per project for each major category.

An overview of the mean age of projects in the major categories is provided in Figure 4.3. In general, it appears that projects tend to become slightly more likely to opt for stricter compiler warnings as they become older. The difference between AEP+ and N is approximately 1,050 days, i.e., 2.8 years.

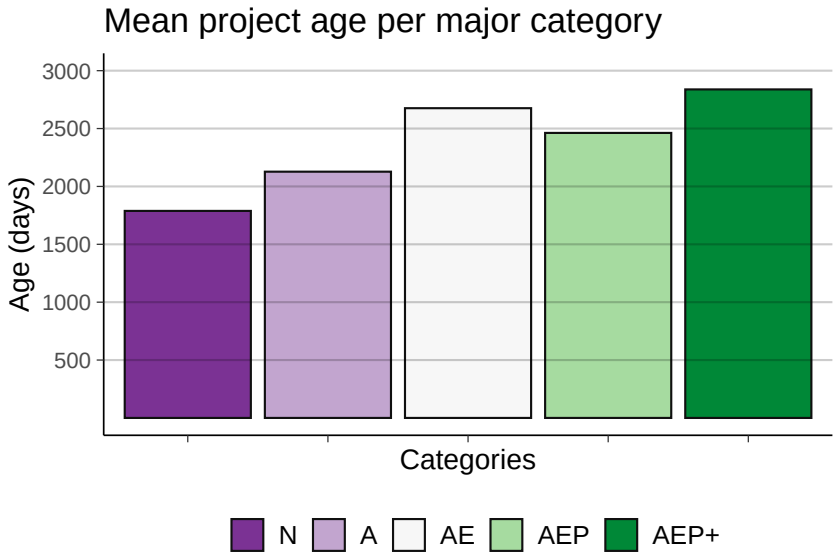

Figure 4.3: Mean age in days of projects in each major category. Projects in stricter warning categories tend to be older.

As previously discussed, the number of stargazers was used to select samples so it is interesting to observe the number of stargazers across the major categories.

An overview of the mean number of stargazers per major category is included in Figure 4.4. Projects in AEP+ have the most stargazers on average while N has the fewest, the mean difference between these categories is roughly 2,175 stargazers. For some reason, projects in AE and AEP+ tend to be more "popular", as in they have more stargazers on average.

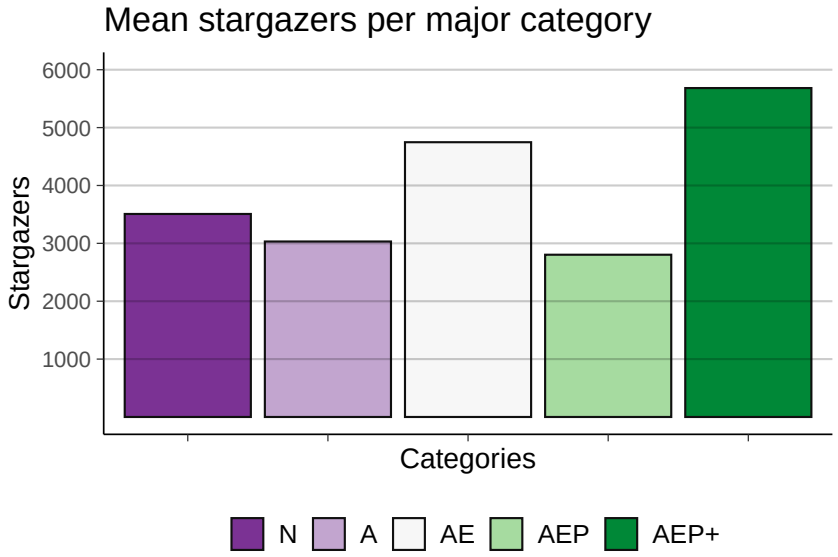

Figure 4.4: Mean number of stargazers of projects in each major category.

4.2 Analysis

This section presents the developed statistical models and analysis results of said models. A total of eleven code quality metrics are considered, corresponding to various code quality characteristics. The various parameters used in the model specifications are listed below.

C: number of contributors.
S: number of stargazers.
T: project age, in days.
Z: physical lines of code.
F: number of files.

4.2.1 Bugs

Models related to the bug metric have the standardized number of bugs per KLOC as their outcome variable (lower is better). The candidate models that were considered are listed in Table 4.3, where they are sorted according to their predictive power. As such, B1 and B6 have the best and worst predictive accuracy, respectively. Therefore, B1 is used for further analysis of the bug metric.

Model	Parameters	ELPD (LOO)	Δ ELPD
B1	None	-192.0	0.0
B4	T	-196.0	-4.0
B2	C	-196.0	-4.0
B3	S	-196.6	-4.6
B5	C, S	-197.1	-5.1
B6	C, S, T	-197.9	-5.9

Table 4.3: Overview of candidate models for the bug metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. B1, the model with no parameters, is the model that produces the best predictions.

An overview of the results for the major categories using model B1 is provided in Table 4.4. These results were obtained using the complete sample. From the table, it is apparent that categories N and A have the worst mean scores, at 0.97 and 0.94 B/KLOC, respectively. Category AEP has the best overall mean at 0.37 B/KLOC, amounting to a difference of 0.60 B/KLOC compared to category N.

Category	Mean	SD	0.05 CI	0.95 CI
N	0.97	0.88	0.64	1.31
A	0.94	0.94	0.50	1.37
AE	0.65	0.89	0.29	1.00
AEP	0.37	0.92	-0.05	0.78
AEP+	0.42	0.89	0.07	0.78

Table 4.4: Summary of results for major categories, from model B1 using the complete sample. Category N has the worst mean at 0.97 B/KLOC, and category AEP has the best mean at 0.37 B/KLOC.

The credible intervals from Table 4.4 are plotted in Figure 4.5. It is notable that the difference between the best categories, AEP and AEP+, is minor. This could indicate that the addition of the `-Werror` flag (or equivalent) does not significantly improve the code quality regarding the bug metric. There is a discernable, somewhat linear, gradient from the stricter categories to the ones using fewer warnings. As such, it appears that `-Wextra` and `-Wpedantic` are both effective at reducing the overall number of bugs. Furthermore, the similar scores of categories A and N suggest that `-Wall` is insufficient to improve the number of bugs on its own. Overall, stricter warning usage tends to result in lower amounts of bugs in the sampled projects.

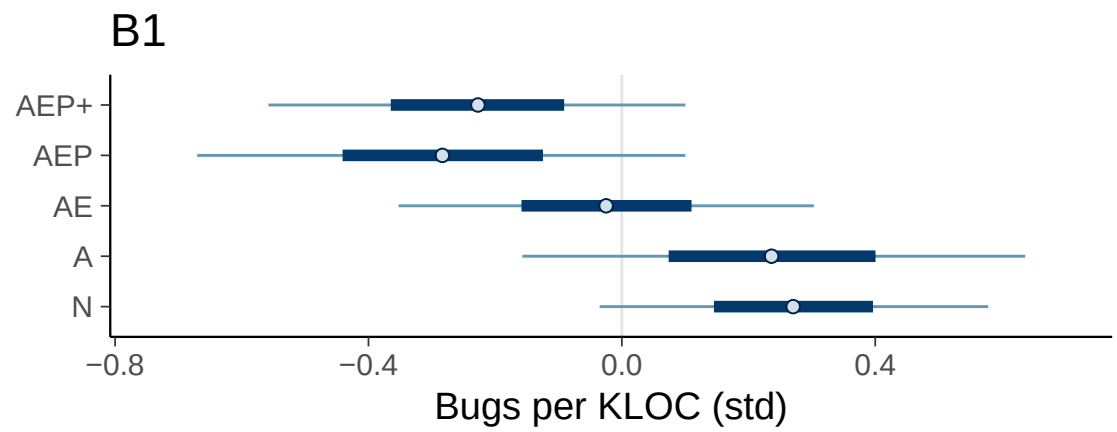

Figure 4.5: Credible intervals of B/KLOC (standardized) for each major category, from model B1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

Credible intervals for projects that do use warnings and those that do not are presented in Figure 4.6. The model has greater uncertainty for the samples that do not use warnings in this case, due to the small sample size concerning the aggregate samples that use warnings to some extent. Nevertheless, it seems that the usage of warnings to any extent results in lower bug amounts compared to when no warnings are used.

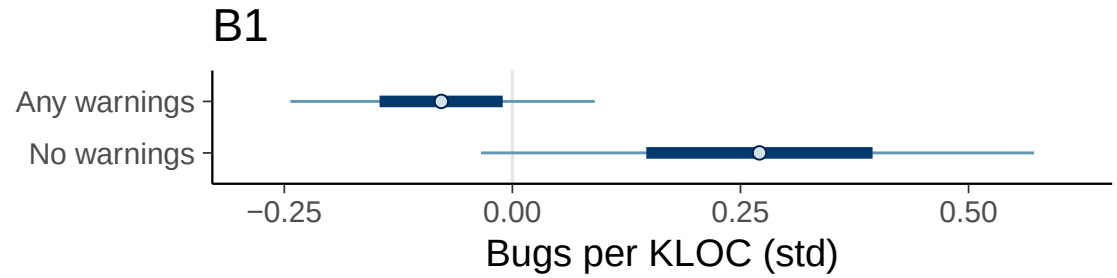

Figure 4.6: Credible intervals of B/KLOC (standardized) for projects that use any warnings and projects that use none, from model B1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. The projects that use warnings generally feature lower bug scores than projects that use no warnings.

4.2.2 Code Smells

The models designed for the code smell metric have the standardized number of code smells per KLOC as their outcome variable (lower is better). All such models that were devised are provided in Table 4.5. Model CS6 has the best ELPD score, and thus provides the best predictions, while model CS2 produces the worst predictions. Therefore, model CS6 is used for further analysis of the code smell metric.

Model	Parameters	ELPD (LOO)	Δ ELPD
CS6	C, T	-197.1	0.0
CS3	S	-197.8	-0.6
CS5	C, S	-197.8	-0.7
CS4	T	-198.5	-1.4
CS1	None	-198.6	-1.5
CS7	C, S, T	-198.9	-1.8
CS2	C	-199.1	-2.0

Table 4.5: Overview of candidate models for the code smell metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. CS6 is the model that produces the best predictions and is therefore utilized in further analysis steps.

The results from model CS6 for the major categories using the complete sample are presented in Table 4.6. Surprisingly, category AEP has the highest mean out of any category, at 87.95 CS/KLOC, while the similar category AEP+ has the lowest mean at 54.31 CS/KLOC. Taken at face value, the difference between these categories amounts to 33.64 CS/KLOC. Excluding AEP, categories N and A have the highest means, at 71.45 and 72.72 CS/KLOC, respectively. The peculiar contrast of AEP and AEP+, which use the same set of warnings, can be explained by outliers in the sample, see Figure D.2. There are two considerable outliers, one in category AEP ("clipp") and another in category AEP+ ("hana"). These two outliers are likely biasing the estimates for AEP and AEP+, given that the rest of the samples in these categories have substantially lower scores, even if the effects appear to be less noticeable for category AEP+.

Category	Mean	SD	0.05 CI	0.95 CI
N	71.45	79.07	50.50	91.76
A	72.72	82.24	47.33	98.11
AE	60.66	79.70	39.71	82.24
AEP	87.95	81.60	63.20	112.71
AEP+	54.31	79.70	32.72	75.89

Table 4.6: Summary of results for major categories, from model CS6 using the complete sample. Category AEP has the worst mean at 87.95 CS/KLOC, and category AEP+ has the best mean at 54.31 CS/KLOC.

Credible intervals from Table 4.6 are presented in Figure 4.7. Looking at the figure, the discrepancy between AEP and AEP+ is obvious. Excluding category AEP, the categories exhibit a general pattern of lower code smell scores when using stricter warnings.

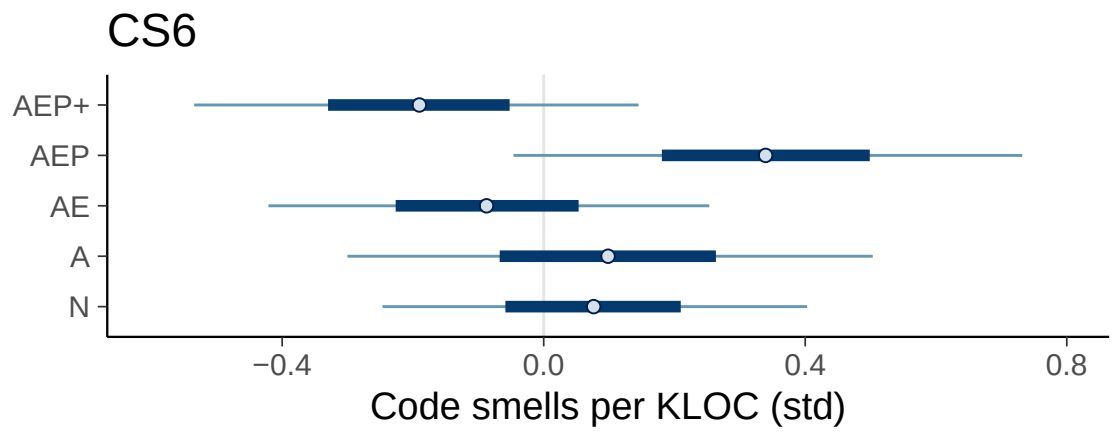

Figure 4.7: Credible intervals of CS/KLOC (standardized) for each major category, from model CS6 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

To determine whether the aforementioned outliers bias the results, another round of analysis is conducted using model CS6 excluding said outliers. The two outliers, `clipp`, and `hana`, consist of 9,000 and 59,000 LOC, respectively. A significant majority of the reported code smells for `clipp` come from test files, only 251 out of 5,681 code smells come from the main library sources. The code smells from `hana` are not limited to just test files, but are isolated to a few source files. In fact, more than half of the code smells (9,809 out of 16,248) come from just four source files, `basic_tuple.hpp`, `detail/ebo.hpp`, `functional/partial.hpp`, and `test/_include/laws/base.hpp`. Given the complex nature of `hana`, being a compile-time regular expression library, some files can be assumed to contain non-trivial code that produces an unreasonable amount of warnings.

The results from running model CS6 using the culled sample are provided in Table 4.7. AEP+ is still the best category, at 44.20 CS/KLOC, and AEP is dramatically improved to be much closer to AEP+, at 55.44 CS/KLOC. This makes AEP the second-best category, despite being the worst category using the complete sample. Categories N and A remain similar, at 73.83 and 73.15 CS/KLOC, respectively, making N the worst category. The difference between the best and worst categories is subsequently 29.63 CS/KLOC. Notably, the difference between N and AEP+ went from 17.14 CS/KLOC to 29.63 CS/KLOC by using the culled sample. It is apparent that the model becomes much more confident in the results using the culled sample, given the tighter credible intervals and more distinct means.

Category	Mean	SD	0.05 CI	0.95 CI
N	73.83	66.68	63.27	84.04
A	73.15	68.38	59.87	86.77
AE	58.51	67.02	47.61	69.74
AEP	55.44	68.04	42.50	68.38
AEP+	44.20	67.02	32.63	55.78

Table 4.7: Summary of results for major categories, from model CS6 using a sample that excludes `clipp` and `hana`. Category N has the worst mean at 73.83 CS/KLOC, and category AEP+ has the best mean at 44.20 CS/KLOC.

A plot of the credible intervals for the major categories excluding the outliers is provided in Figure 4.8. Note the gradual increase in standardized CS/KLOC, going from category AEP+ to category N, i.e., going from strict warnings to no warnings at all. Additionally, the regions are more distinctly separated using the culled sample. Given the similarity of the means for AE and AEP, it appears that `-Wpedantic` results in a minor but not significant improvement regarding code smells. There is a significant step from N and A to the other major categories, so `-Wextra` might be particularly impactful for preventing code smells. The non-negligible difference between AEP+ and AEP is interesting, one possible explanation is that developers ignore `-Wpedantic` warnings until they are forced to deal with them, such as when using `-Werror`.

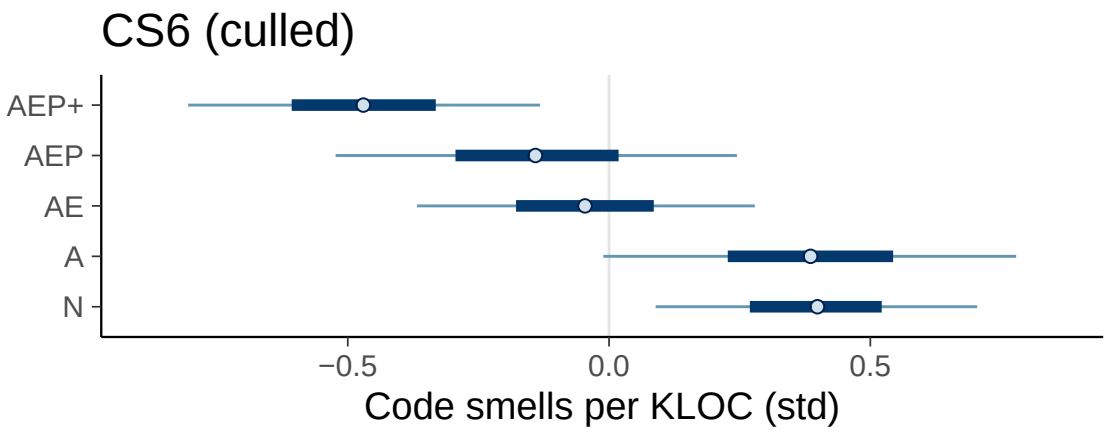

Figure 4.8: Credible intervals of CS/KLOC (standardized) for each major category, from model CS6 using a sample that excludes `clipp` and `hana`. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

The effect of excluding the outliers is very noticeable when comparing the projects that use any warnings and projects that use none. See Figure 4.9 and Figure 4.10. The two groups are clearly distinct when `hana` and `clipp` are excluded, with projects that use warnings to some extent performing significantly better overall.

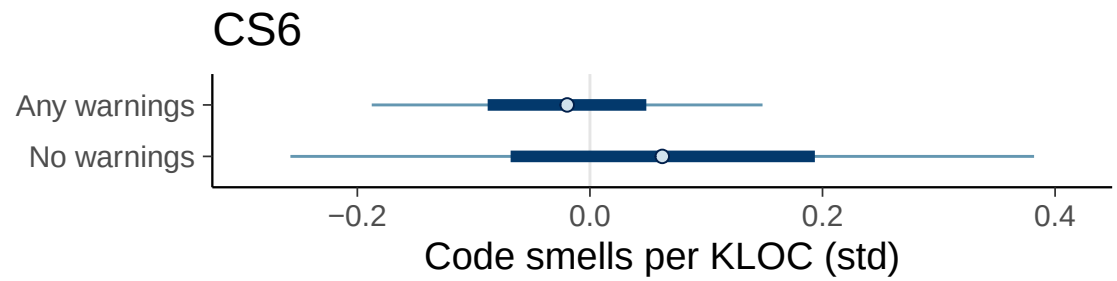

Figure 4.9: Credible intervals of CS/KLOC (standardized) for projects that use any warnings and projects that use none, from model CS6 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

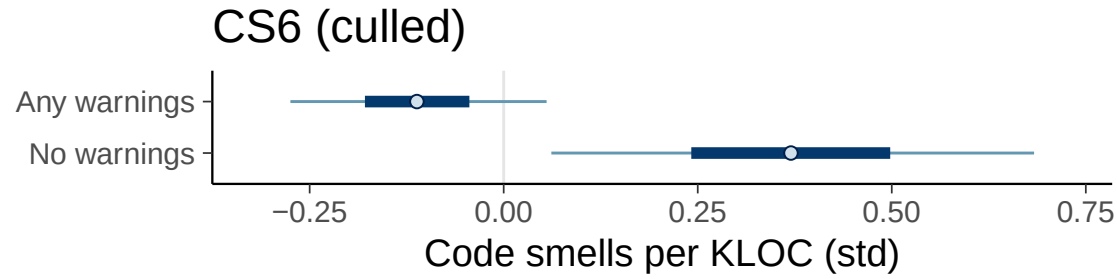

Figure 4.10: Credible intervals of CS/KLOC (standardized) for projects that use any warnings and projects that use none, from model CS6 using a sample that excludes `clipp` and `hana`. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

4.2.3 Critical Violations

The models developed for the critical violation metric all have the standardized number of critical violations per KLOC as their outcome variable. All candidate models are provided in Table 4.8. The simplest model, CV1, offers the best predictions out of the candidate models and is therefore used in further analysis.

Model	Parameters	ELPD (LOO)	Δ ELPD
CV1	None	-185.0	0.0
CV4	F	-185.4	-0.4
CV3	S	-185.5	-0.5
CV2	C	-185.6	-0.6
CV6	Z	-185.8	-0.8
CV5	T	-185.8	-0.8
CV8	C, T	-186.2	-1.2
CV7	C, S	-186.2	-1.2

Table 4.8: Overview of candidate models for the critical violation metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. CV1 is the model that produces the best predictions and is subsequently used for further analysis.

The results from model CV1 for the major categories using the complete sample are displayed in Table 4.9. Category AEP+ is the best category by quite some margin, at 11.89 CV/KLOC, and category N is the worst category at 23.87 CV/KLOC. The remaining categories A, AE, and AEP all feature closer means, ranging from around 19 to 16 CV/KLOC. This metric displays the familiar gradual decrease in CV/KLOC as the warnings used get stricter. There are no obvious outliers in the sample, see Appendix D.3, so there is no reason to perform additional analysis with a culled sample.

Category	Mean	SD	0.05 CI	0.95 CI
N	23.97	19.74	19.44	28.50
A	19.29	20.65	13.40	25.18
AE	17.63	20.04	12.64	22.61
AEP	16.12	20.50	10.38	21.86
AEP+	11.89	20.04	7.05	16.72

Table 4.9: Summary of results for major categories, from model CV1 using the complete sample. Category N has the worst mean at 23.97 CV/KLOC, and category AEP+ has the best mean at 11.89 CV/KLOC.

The credible intervals from Table 4.9 are plotted in Figure 4.11. The aforementioned gradual decrease is evident in the figure, going from category N to AEP+ in an approximately linear fashion. Note how the credible intervals for categories N and AEP+ have no overlap whatsoever, which is a testament to the confidence of the statistical model in the difference between these categories. In other words, the statistical model is extremely confident that AEP+ projects have fewer CV/KLOC than projects in category N. There is a sizable difference between AEP+ and AEP, but the improvement when increasing the number of warnings used appears quite uniform in general.

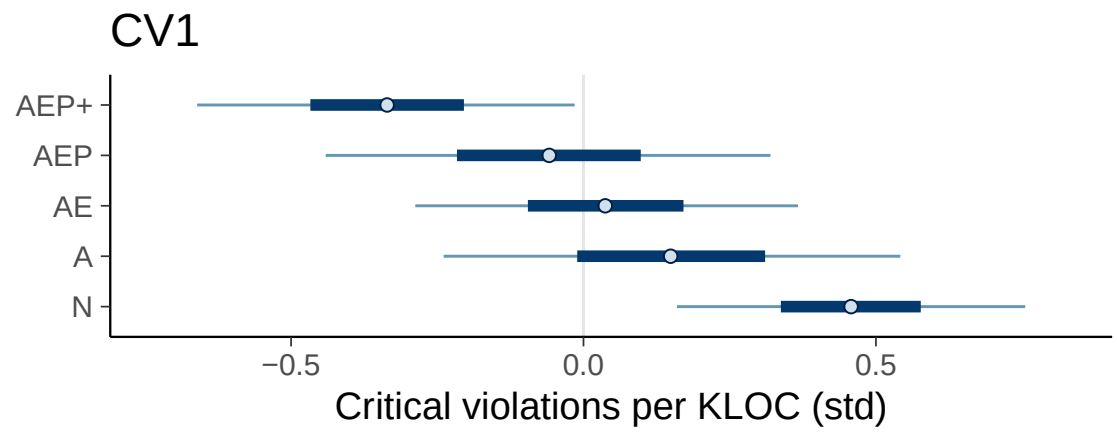

Figure 4.11: Credible intervals of CV/KLOC (standardized) for each major category, from model CV1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

The same confidence is evident when comparing the projects that use warnings and those that do not. A plot of such credible intervals is presented in Figure 4.6. Projects that do make use of warnings obviously feature fewer CV/KLOC on average. Compiler warnings appear quite effective at combating this family of code quality issues.

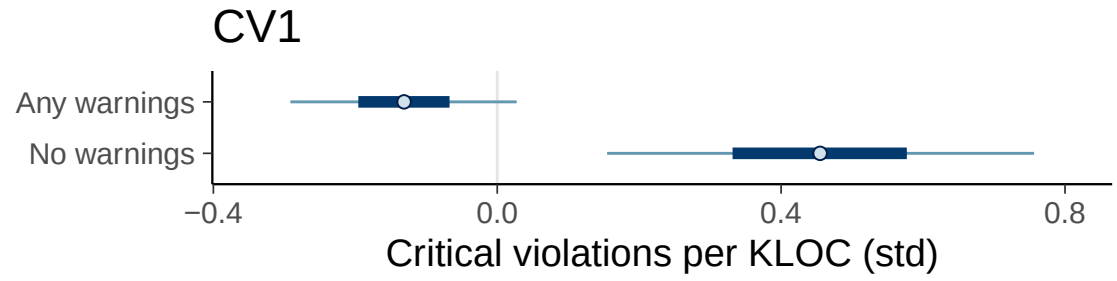

Figure 4.12: Credible intervals of CV/KLOC (standardized) for projects that use any warnings and projects that use none, from model CV1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively. Projects that do use warnings feature fewer CV/KLOC.

4.2.4 Major Violations

The models developed for the major violation metric feature the standardized number of major violations per KLOC as their outcome variable (lower is better). These models are presented in Table 4.10. The model that conditions on the age of projects, MAV5, produces the best predictions and is therefore chosen for further use.

Model	Parameters	ELPD (LOO)	Δ ELPD
MAV5	T	-205.9	0.0
MAV3	S	-205.9	-0.1
MAV8	S, T	-206.0	-0.1
MAV4	F	-206.0	-0.2
MAV7	C, T	-207.1	-1.2
MAV1	None	-208.1	-2.2
MAV2	C	-208.1	-2.2
MAV6	C, S	-208.9	-3.0

Table 4.10: Overview of candidate models for the major violation metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. MAV5 produces the best predictions and is used in subsequent analysis steps.

A summary of the results from model MAV5 using the complete sample is provided in Table 4.11. The best category is AEP+, with 17.90 MaV/KLOC. Interestingly, category AEP is by far the worst performing category, at 52.49 MaV/KLOC, which is considerably higher than the second worst category, A, at 29.43 MaV/KLOC. However, there is a tremendous outlier in category AEP for this metric, see Figure D.4. The outlier in question is `clipp`, which has previously been discussed in the results section for code smells. Again, to determine the impact of this outlier, the analysis is conducted once again with a culled sample.

Category	Mean	SD	0.05 CI	0.95 CI
N	24.72	37.29	7.42	41.48
A	29.43	39.91	7.95	50.91
AE	22.62	37.81	4.80	40.43
AEP	52.49	39.39	31.53	72.92
AEP+	17.90	37.81	-0.43	35.72

Table 4.11: Summary of results for major categories, from model MAV5 using the complete sample. Category AEP has the worst mean at 52.49 MaV/KLOC, and category AEP+ has the best mean at 17.90 MaV/KLOC. An outlier is likely causing the poor results for category AEP.

The credible intervals from Table 4.11 are visualized in Figure 4.13. Category AEP is clearly displaced compared to the other major categories, while the other categories appear relatively similar. There is no clear gradient from stricter warning categories to more relaxed ones as seen for some earlier metrics. The model appears quite uncertain about the intervals of the categories, due to the considerable overlap of the different credible intervals.

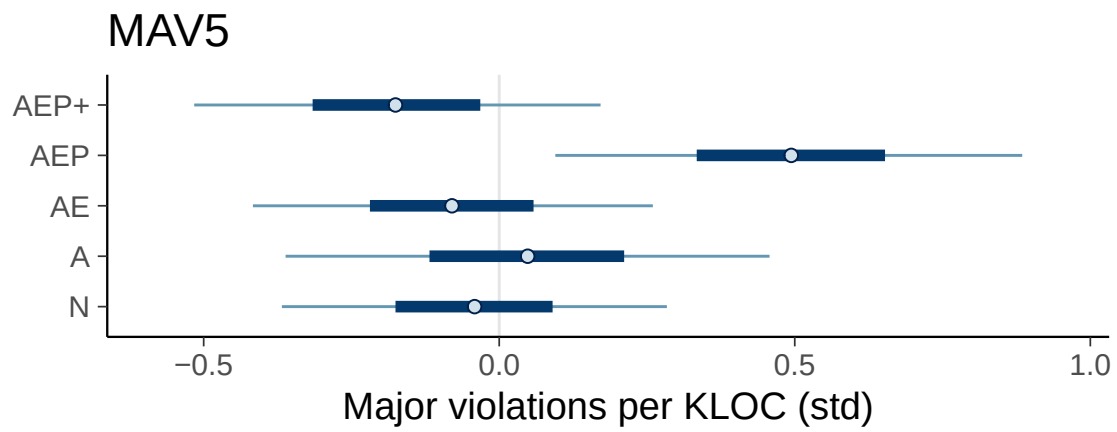

Figure 4.13: Credible intervals of MaV/KLOC (standardized) for each major category, from model MAV5 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

A comparison of projects that use warnings and those that do not are provided in Figure 4.14. The interval for projects without warnings is very large, but the model seems to believe that projects that use warnings are worse on average when the outlier is included in the sample.

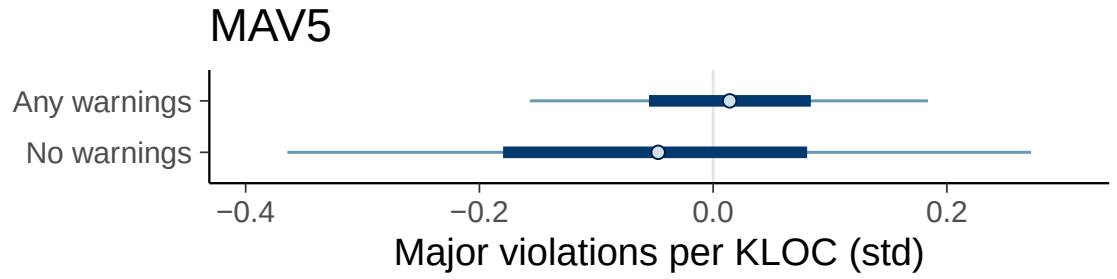

Figure 4.14: Credible intervals of MaV/KLOC (standardized) for projects that use any warnings and projects that use none, from model MAV5 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

A result summary when running model MAV5 without `clipp` is provided in Table 4.12. Category AEP, which was previously the worst category by a clear margin, becomes the second best category when `clipp` is excluded. The other categories have means relatively similar to the ones obtained when using the complete sample. It is clear that `clipp` biases the estimations quite heavily when included.

Category	Mean	SD	0.05 CI	0.95 CI
N	25.53	25.38	20.64	30.59
A	29.48	26.17	23.17	35.80
AE	21.74	25.53	16.38	26.80
AEP	20.80	26.17	14.64	27.11
AEP+	16.53	25.69	11.16	21.90

Table 4.12: Summary of results for major categories, from model MAV5 using a sample excluding `clipp`. Category A now has the worst mean at 29.48 MaV/KLOC, and category AEP+ still has the best mean of 16.53 MaV/KLOC.

A plot of the credible intervals with the culled sample is included in Figure 4.15. The intervals are now laid out much more linearly, although A is notably performing worse than N. This might be caused by an ineffectiveness in `-Wall` warnings to highlight issues marked as major violations by SonarScanner. Overall, it still appears that stricter warnings correlate with fewer MaV/KLOC in general.

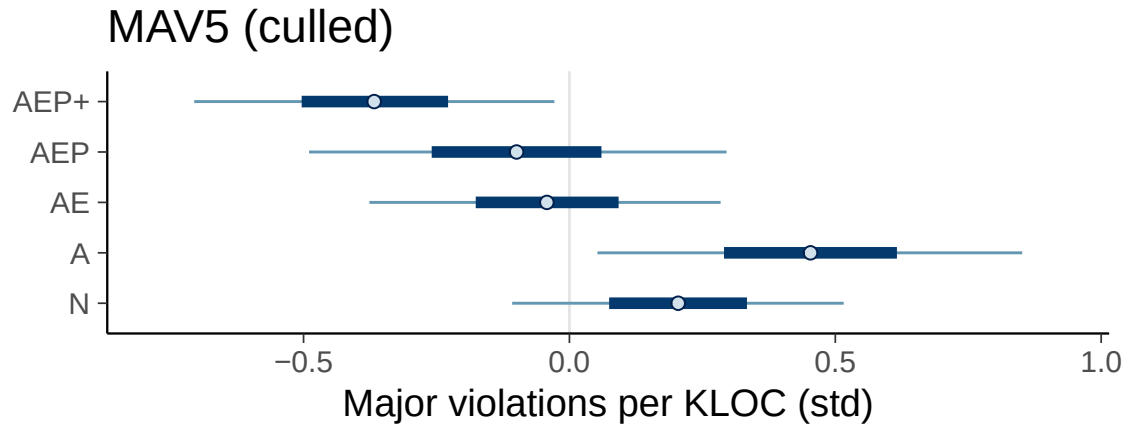

Figure 4.15: Credible intervals of MaV/KLOC (standardized) for each major category, obtained from model MAV5 using a sample that excludes `clipp`. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

When comparing the effects of any warning usage compared to using no warnings, the model appears relatively certain that projects that use warnings to some extent have fewer MaV/KLOC, see Figure 4.16. However, the credible interval for projects that do not use any warnings is quite wide, which is likely caused by the fact that projects in category A have the greatest amount of MaV/KLOC, in addition to the smaller sample size of projects without any warnings.

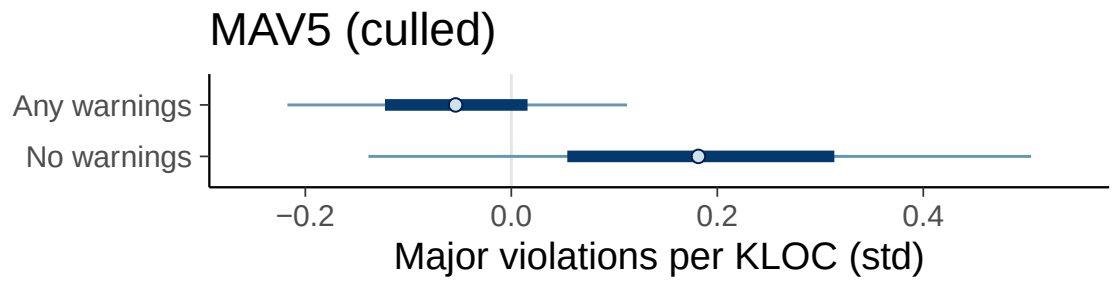

Figure 4.16: Credible intervals of MaV/KLOC (standardized) for projects that use any warnings and projects that use none, obtained from model MAV5 using a sample that excludes `clipp`. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

4.2.5 Minor Violations

The minor violation models have the standardized number of minor violations per KLOC as their outcome variable. All models that were considered are presented in Table 4.13. Several candidate models perform very similarly, with MIV1 and MIV2 at the top. Due to the similar performance, MIV1 is used in further analysis, since it is the simplest model.

Model	Parameters	ELPD (LOO)	Δ ELPD
MIV2	C	-186.9	0.0
MIV1	None	-187.1	-0.2
MIV5	Z	-187.2	-0.4
MIV3	S	-187.4	-0.5
MIV6	T	-187.5	-0.7
MIV4	F	-187.9	-1.0
MIV7	C, S	-188.0	-1.1
MIV8	C, T	-188.0	-1.1

Table 4.13: Overview of candidate models for the minor violation metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. MIV2 and MIV1 perform very similarly, so MIV1 is used in further analysis despite not having the highest ELPD score.

Analysis results from model MIV1 with the complete sample are presented in Table 4.14. Out of the major categories, AEP+ and N have the best and worst scores, respectively. The difference between the mean MiV/KLOC for these two categories amounts to 7.36 MiV/KLOC. There are no obvious outliers in the sample in regards to the minor violation metric (see Figure D.5), so there is little reason to perform further analysis with MIV1 using a culled sample.

Category	Mean	SD	0.05 CI	0.95 CI
N	24.10	22.92	19.54	28.51
A	23.36	23.66	17.48	29.25
AE	17.92	23.07	13.07	22.78
AEP	17.92	23.51	12.33	23.51
AEP+	16.74	23.07	11.89	21.60

Table 4.14: Summary of results for major categories, from model MIV1 using the complete sample. Category N has the worst mean at 24.10 MiV/KLOC, and category AEP+ has the best mean at 16.74 MiV/KLOC.

Credible intervals for the major categories are presented in Figure 4.17. There seem to be two groupings among the major categories. Category N and A have means around 24 MiV/KLOC, while AE, AEP, and AEP+ have means around 17-18 MiV/KLOC. This suggests that the `-Wextra` flag could be especially influential for this metric since it is shared by the best major categories. Additionally, the `-Wpedantic` flag does not seem overly significant for this metric, since AE and AEP have very similar scores. However, AEP+ is better than both AE and AEP, but uses the same set of warnings as AEP (but treated as errors). This could potentially be caused by programmers ignoring `-Wpedantic` warnings unless they are required to address them, which is the case when using `-Werror`.

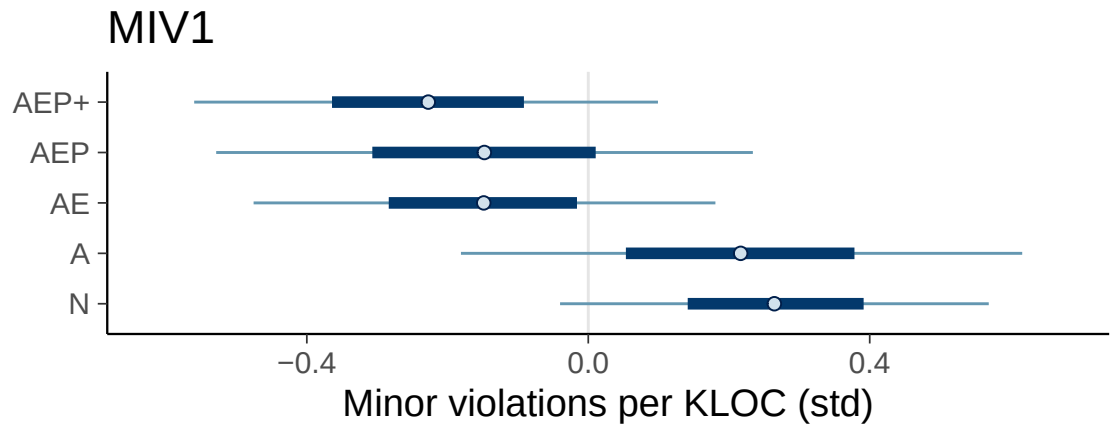

Figure 4.17: Credible intervals of MiV/KLOC (standardized) for each major category, from model MIV1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

A comparison of projects that use warnings and those without warnings is provided in Figure 4.18. The tendency for projects that make use of compiler warnings to perform better appears to hold for the minor violation metric.

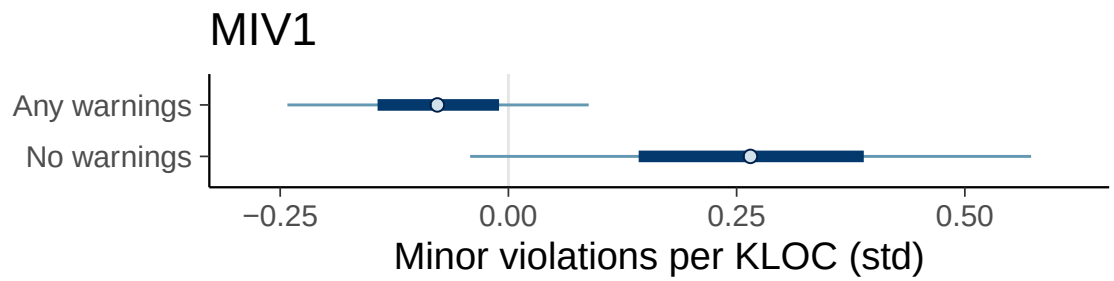

Figure 4.18: Credible intervals of MiV/KLOC (standardized) for projects that use any warnings and those that use none, from model MIV1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

4.2.6 Security Hotspots

Models proposed for analyzing the security hotspot metric are included in Table 4.15. These models have the standardized number of security hotspots per KLOC as their outcome variable. Model S2, which conditions on the number of contributors, leads to the best predictions and is therefore used in the subsequent analysis.

Model	Parameters	ELPD (LOO)	Δ ELPD
S2	C	-184.2	0.0
S7	C, S	-184.6	-0.4
S8	C, Z	-184.7	-0.5
S3	S	-184.7	-0.5
S5	F	-184.9	-0.8
S6	Z	-184.9	-0.8
S1	None	-185.1	-0.9
S4	T	-185.8	-1.6

Table 4.15: Overview of candidate models for the security hotspot metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. S2 is the model that produces the best predictions and is used in further analysis.

An overview of the results from model S2 when using the complete sample is provided in Table 4.16. Unsurprisingly, the worst category is N, at 0.97 SH/KLOC. It is perhaps slightly more surprising to see the best category is AE, at 0.48 SH/KLOC. There is no clear correlation in this case between stricter warning usage and a reduction in SH/KLOC. In fact, the amount of SH/KLOC increases when going from category AE to AEP to AEP+. There are no obvious outliers in the sample for the security hotspot metric (see Figure D.6), so further analysis using a culled sample is not conducted.

Category	Mean	SD	0.05 CI	0.95 CI
N	0.97	0.89	0.64	1.30
A	0.89	0.94	0.48	1.31
AE	0.48	0.90	0.13	0.83
AEP	0.58	0.93	0.18	1.00
AEP+	0.70	0.90	0.35	1.04

Table 4.16: Summary of results for major categories, from model S2 using the complete sample. Category N has the worst mean at 0.97 SH/KLOC, while category AE has the best mean at 0.48 SH/KLOC.

The data in Table 4.16 is plotted as credible intervals in Figure 4.19. Note the aforementioned reverse gradient when going from AE to AEP+. Categories N and A have similar intervals, indicating that the `-Wall` warnings have little to no effect in regards to the SH/KLOC metric. Furthermore, the unintuitive estimates for AE, AEP, and AEP+ might hint that `-Wextra` provides some utility in decreasing SH/KLOC, but there are likely other external factors that contribute more to this metric. After all, it is unlikely that the introduction of stricter warnings, such as `-Wpedantic` and `-Werror`, increases the number of security hotspots.

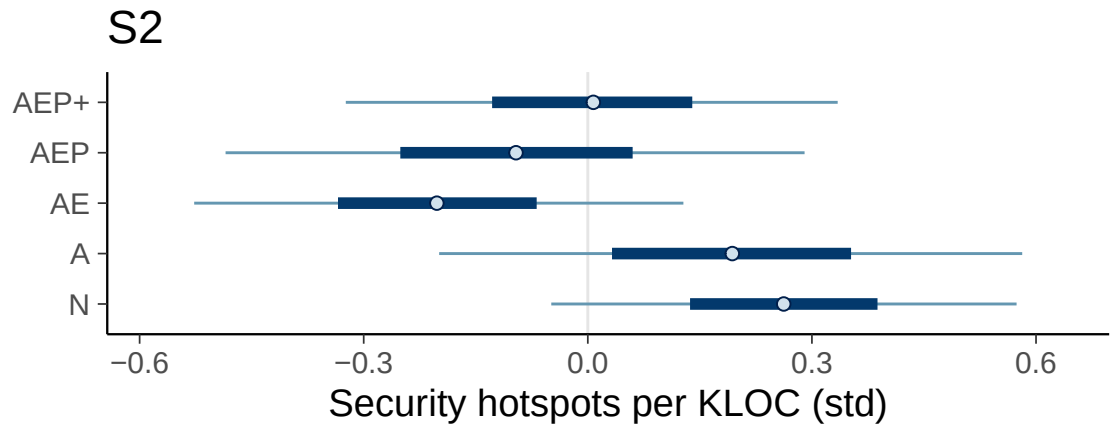

Figure 4.19: Credible intervals of SH/KLOC (standardized) for each major category, from model S2 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

While stricter warnings appear ineffective at reducing the SH/KLOC metric, projects that use warnings generally appear to exhibit slightly fewer SH/KLOc overall, see Figure 4.20. There are quite a bit of overlapping intervals for this metric, indicative of greater variance within the different categories, increasing the uncertainty of the model estimates.

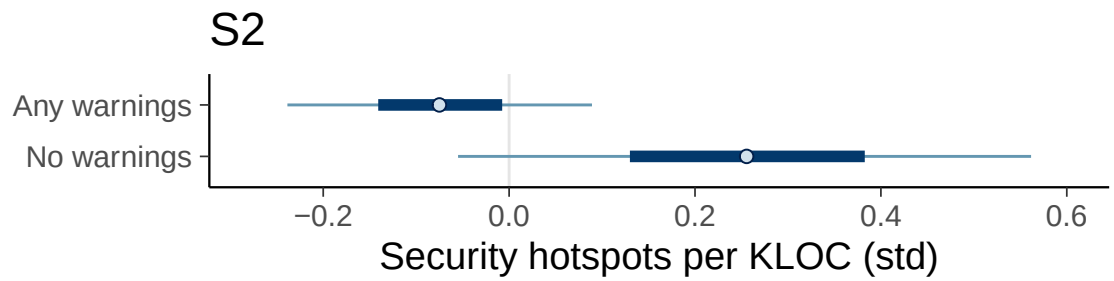

Figure 4.20: Credible intervals of SH/KLOC (standardized) for projects that use warnings and those that do not, from model S2 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

4.2.7 Vulnerabilities

These statistical models developed for the vulnerability metric are listed in Table 4.17. The outcome variable of these models is the standardized amount of vulnerabilities per KLOC. In this case, the simplest model with no additional parameters resulted in the best predictions, V1. Therefore, V1 is used to obtain estimates for the vulnerability metric.

Model	Parameters	ELPD (LOO)	Δ ELPD
V1	None	-203.7	0.0
V6	F	-203.7	0.0
V4	T	-203.9	-0.2
V5	Z	-204.2	-0.5
V3	S	-204.3	-0.7
V2	C	-204.4	-0.7
V7	C, T	-204.6	-0.9
V8	C, Z	-204.7	-1.0

Table 4.17: Overview of candidate models for the vulnerability metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. V1 has the greatest predictive accuracy and is used in additional analysis.

Estimates obtained from model V1 are presented in Table 4.18. Vulnerabilities represent serious security issues in code bases, they are as such quite scarce in comparison to other code quality metrics. As for many of the investigated code quality metrics, AEP+ has the best estimate at 0.000951 V/KLOC. Category A has the worst estimate at 0.022476 V/KLOC, approximately one order of magnitude higher than the estimate for AEP+. There is a notable decrease in V/KLOC as warnings get stricter, although category N performs better than two other categories that use warnings (A and AE). This is suspect if compiler warnings do indeed prevent vulnerabilities. The similarity between the estimates for AEP+ and N indicates that external factors other than compiler warning usage are more influential when it comes to the number of vulnerabilities. The are no obvious outliers in the sample in

regards to the vulnerability metric (see Figure D.7), so there is no need for a round of analysis using a modified sample.

Category	Mean	SD	0.05 CI	0.95 CI
N	0.006849	0.014115	-0.003162	0.016899
A	0.022476	0.016060	0.009269	0.035717
AE	0.012868	0.014587	0.002055	0.023660
AEP	0.004848	0.015799	-0.008017	0.017773
AEP+	0.000951	0.014693	-0.010096	0.011965

Table 4.18: Summary of results for all categories, from model V1 using the complete sample. Category A has the highest mean at 0.022476 V/KLOC, and category AEP+ has the lowest mean at 0.000951 V/KLOC.

The anomaly that is category N for the vulnerability metric is apparent in Figure 4.21, which includes the credible intervals from Table 4.18. There is a notable decrease in V/KLOC when going from A to AEP+, but N clearly does not perform according to this gradual increase in V/KLOC. While the strictest warning categories seem better than N, it is safe to say that the difference is not as emphasized as for other code quality metrics that have been considered.

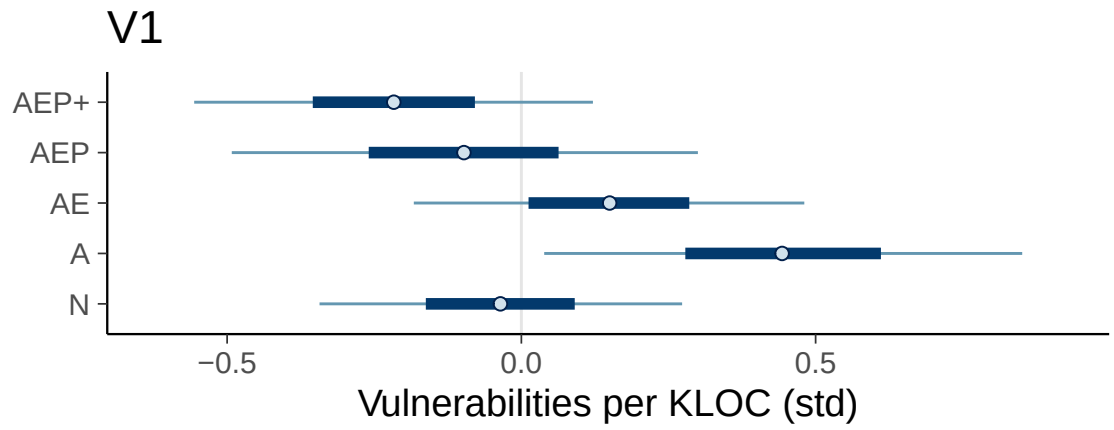

Figure 4.21: Credible intervals of V/KLOC (standardized) for each major category, from model V1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

The overall difference between projects that use some warnings and those with no warnings is highlighted in Figure 4.22. The intervals are quite similar, where the interval for projects with no warnings begins very wide due to the small sample size concerning the number of projects that use warnings. Subsequently, it has to be concluded that compiler warnings have little impact on the overall amount of vulnerabilities. Other factors, such as culture or best practices, are likely more influential in this case.

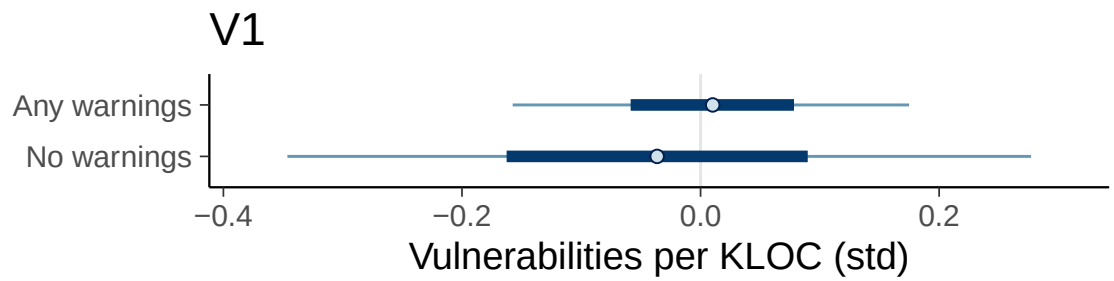

Figure 4.22: Credible intervals of V/KLOC (standardized) for projects that use any warnings and projects that use none, from model V1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

4.2.8 Cyclomatic Complexity

It is unlikely that compiler warnings directly affect cyclomatic complexity since none of the considered compilers feature warnings for cyclomatic complexity. While this metric is included for completeness, the results in this section have been condensed, see the replication package for the complete set of plots and analysis steps. Cyclomatic complexity was analyzed using the models listed in Table 4.19. The outcome variable of these models is the standardized amount of cyclomatic complexity per KLOC. The model that includes the number of files and code size, CYC7, provides the best predictions. Therefore, model CYC7 is used in further analysis.

Model	Parameters	ELPD (LOO)	Δ ELPD
CYC7	Z, F	-198.7	0.0
CYC4	F	-199.0	-0.2
CYC3	Z	-199.8	-1.1
CYC8	Z, T	-199.9	-1.1
CYC5	T	-200.0	-1.3
CYC6	S	-201.2	-2.4
CYC2	C	-201.5	-2.8
CYC1	None	-201.9	-3.1

Table 4.19: Overview of candidate models for the cyclomatic complexity metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. CYC7 marginally outperforms CYC4 and is therefore used in subsequent analysis.

The results from analyzing the sample using CYC7 are summarized in Table 4.20. Curiously, cyclomatic complexity appears to increase as the utilized compiler warnings become stricter. Category N turns out to feature the best mean at 137.50 Cyc/KLOC while A is the worst category at 178.42, with AEP+ not far of at 174.00 Cyc/KLOC. However, the sample features a substantial outlier AEP+, see Figure D.8. The outlier is `better-enums`, a relatively small project at only 1,921

LOC. Therefore, another round of analysis is warranted using a sample that excludes `better-enums`.

Category	Mean	SD	0.05 CI	0.95 CI
N	137.50	177.31	103.21	172.89
A	178.42	183.95	133.07	223.77
AE	151.87	179.53	115.38	189.48
AEP	160.72	182.84	116.48	204.96
AEP+	174.00	178.42	136.39	211.60

Table 4.20: Summary of results for major categories, from model CYC7 using the complete sample. N has the best mean at 137.50 Cyc/KLOC, while A has the worst estimated mean at 178.42 Cyc/KLOC.

The analysis results when the `better-enums` sample is excluded are provided in Table 4.21. AEP+ now features the best mean estimate at 132.80 Cyc/KLOC and category A retains the worst mean at 177.50 Cyc/KLOC. Estimates still appear sporadic with no apparent tendencies among the major categories. The fact that the strictest and most relaxed compiler warnings categories feature such similar estimates is evidence of the lack of influence that compiler warning usage has on the overall cyclomatic complexity.

Category	Mean	SD	0.05 CI	0.95 CI
N	138.31	159.13	119.94	156.68
A	177.50	162.81	153.62	201.38
AE	150.56	160.36	130.35	170.15
AEP	159.75	162.19	136.48	183.01
AEP+	132.80	160.36	111.98	153.01

Table 4.21: Summary of results for major categories, from model CYC7 using a sample that excludes `better-enums`. Category A remains the worst category at 177.50 Cyc/KLOC, while AEP+ becomes the best category at 132.80 Cyc/KLOC, marginally better than N with 138.31 Cyc/KLOC.

Credible intervals for the major categories are plotted in Figure 4.23. The sporadic nature of the categories is clear in the figure, with no real discernable tendencies. This is not particularly surprising considering the aforementioned absence of compiler warnings dedicated to decreasing cyclomatic complexity.

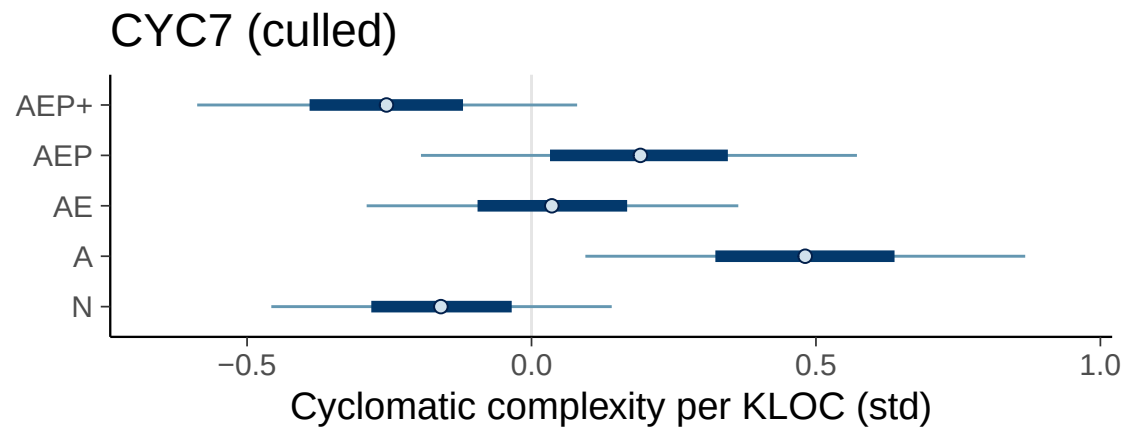

Figure 4.23: Credible intervals of Cyc/KLOC (standardized) for each major category, from model CYC7 using a sample that excludes `better-enums`. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

A comparison of the intervals for projects that use warnings as a whole versus those with no warning usage is provided in Figure 4.24. The model is quite uncertain regarding the estimates for projects that use no compiler warnings, as indicated by the very wide interval. Nevertheless, the model is still somewhat confident that projects that use compiler warnings to some extent have higher cyclomatic complexity on average. The main takeaway from this analysis is that cyclomatic complexity is likely influenced by external factors distinct from compiler warning usage.

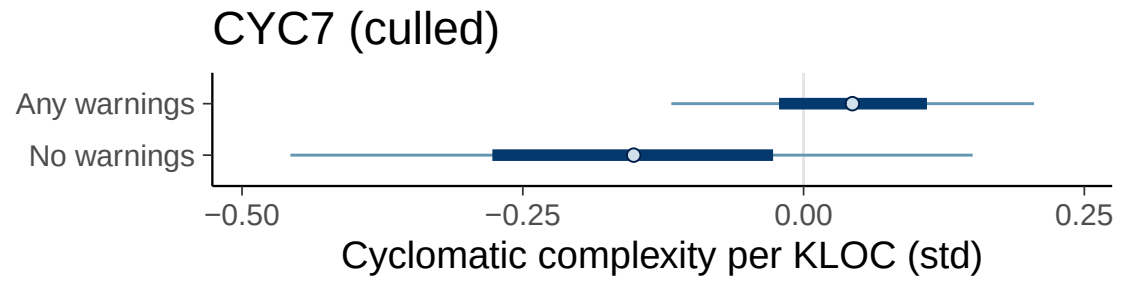

Figure 4.24: Credible intervals of Cyc/KLOC (standardized) for projects that use any warnings and projects that use none, obtained from model CYC7 using a sample that excludes `better-enums`. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

4.2.9 Cognitive Complexity

As discussed in Section 4.2.8, cognitive complexity is not expected to be significantly affected by the choice of enabled compiler warnings. However, some linters, such as `clang-tidy` (a part of the LLVM project [37]), do provide rules for analyzing cognitive complexity [38]. It is plausible that projects that utilize stricter compiler warnings are more likely to make use of such linters. Regardless, any correlation is likely to be caused by other factors, such as engineering culture or similar. The

results presented in this section are somewhat condensed for this reason, akin to the cyclomatic complexity results.

The models considered for the cognitive complexity metric are listed in Table 4.22, all of which feature the standardized amount of cognitive complexity per KLOC as their outcome variable. Model COG5 has the best predictive ability and is therefore used in subsequent analysis.

Model	Parameters	ELPD (LOO)	Δ ELPD
COG5	F	-178.7	0.0
COG8	Z, C	-179.7	-0.9
COG4	Z	-181.2	-2.4
COG3	S	-181.6	-2.9
COG1	None	-182.0	-3.2
COG7	Z, T	-182.1	-3.4
COG2	C	-182.6	-3.8
COG6	T	-183.1	-4.4

Table 4.22: Overview of candidate models for the cognitive complexity metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. COG5 provides the best predictions and is therefore chosen for use in subsequent analysis.

A summary of the results from analyzing the complete sample with COG5 is provided in Table 4.23. A, AE, and AEP all appear very closely grouped at approximately 130–140 Cog/KLOC. Furthermore, AEP+ and N are quite close at 75.95 Cog/KLOC and 94.75 Cog/KLOC, respectively. As for the cyclomatic complexity results, it is difficult to discern any clear correlation between compiler warning usage and cognitive complexity from these results. However, the recurring prowess of AEP+ is worth underlining, since it still performs very well compared to all other categories.

Category	Mean	SD	0.05 CI	0.95 CI
N	94.75	121.61	75.28	114.22
A	139.07	124.97	114.22	164.58
AE	131.68	122.28	110.19	153.17
AEP	141.08	124.29	116.24	165.25
AEP+	75.95	122.28	55.13	97.44

Table 4.23: Summary of results for major categories, from model COG5 using the complete sample. AEP has the worst estimate at 141.08 Cog/KLOC, closely followed by A at 139.07 Cog/KLOC. AEP+ has the best mean at 75.95 Cog/KLOC.

The results from COG5 are visualized in Figure 4.25, in which the aforementioned groupings are obvious. All of the intervals are quite narrow, indicating that the model is confident in the plausible values for each category. There is no reasonable

explanation involving specific compiler warnings for these results, there are most probably other causes for variations in cognitive complexity.

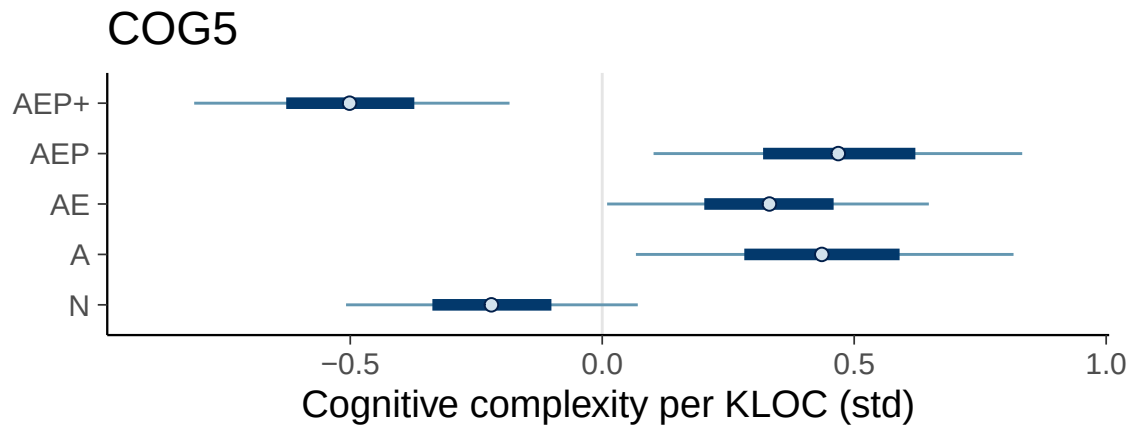

Figure 4.25: Credible intervals of Cog/KLOC (standardized) for each major category, from model COG5 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

The difference between the sampled projects that use warnings and those that do not is illustrated in Figure 4.26. These intervals are quite reminiscent of those in Figure 4.24 from the cyclomatic complexity analysis. It appears that projects that do use warnings generally feature higher cognitive complexity compared to projects with no compiler warnings enabled.

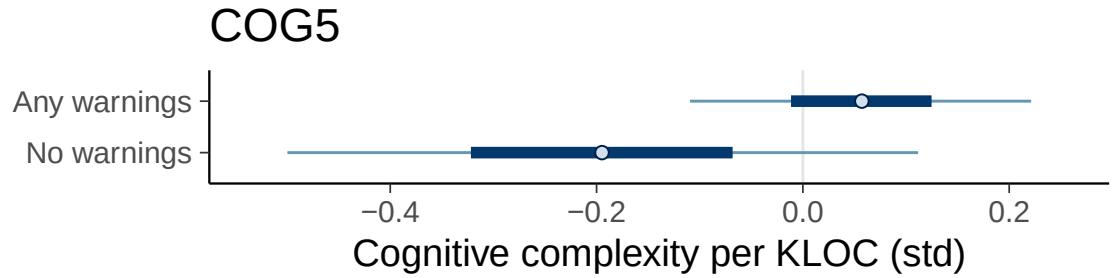

Figure 4.26: Credible intervals of Cog/KLOC (standardized) for projects that use any warnings and those that use none, from model COG5 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

4.2.10 Duplicated Lines

The models considered for analyzing the amount of duplicated lines are provided in Table 4.13. The outcome variable in this case is the standardized amount of duplicated lines per KLOC. Conditioning on the number of files appears to result in the best predictions, as such model D6 is used in the following analysis steps.

Model	Parameters	ELPD (LOO)	Δ ELPD
D6	F	-179.8	0.0
D5	Z	-180.2	-0.4
D1	None	-180.9	-1.1
D8	C, Z	-181.2	-1.4
D4	T	-181.3	-1.5
D2	C	-181.8	-2.0
D7	C, T	-182.3	-2.5
D3	S	-182.7	-2.9

Table 4.24: Overview of candidate models for duplicated lines. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy. D6 has the best ELPD score and is selected for further analysis.

An overview of the simulated results using D6 is provided in Table 4.25. While AEP+ and N are the best and worst categories respectively, there is no apparent pattern among the categories. For instance, AEP is almost as bad as the worst category N, and A is almost as good as AEP+. There is no sign of gradual improvement as the compiler warnings used become stricter. Additionally, the observed difference in estimates between AEP and AEP+ is curious. While it appears that compiler warnings have little influence on the amount of duplicated lines, it is notable that N and AEP+ still perform so differently, hinting at an external factor shared by projects that opt for very strict compiler warning configurations.

Category	Mean	SD	0.05 CI	0.95 CI
N	116.38	116.38	81.28	151.48
A	63.12	122.43	18.33	109.11
AE	81.28	118.80	42.54	120.01
AEP	112.75	121.22	67.96	157.53
AEP+	58.28	117.59	20.75	97.01

Table 4.25: Summary of results for all categories, from model D6 using the complete sample. Category N has the highest estimated mean at 116.38 DL/KLOC, while category AEP+ has the best estimate at 58.28 DL/KLOC.

The results are illustrated in Figure 4.27, in which the lack of a clear trend is evident. It seems unlikely that compiler warnings significantly contribute to the reduction in the amount of duplicated lines in projects.

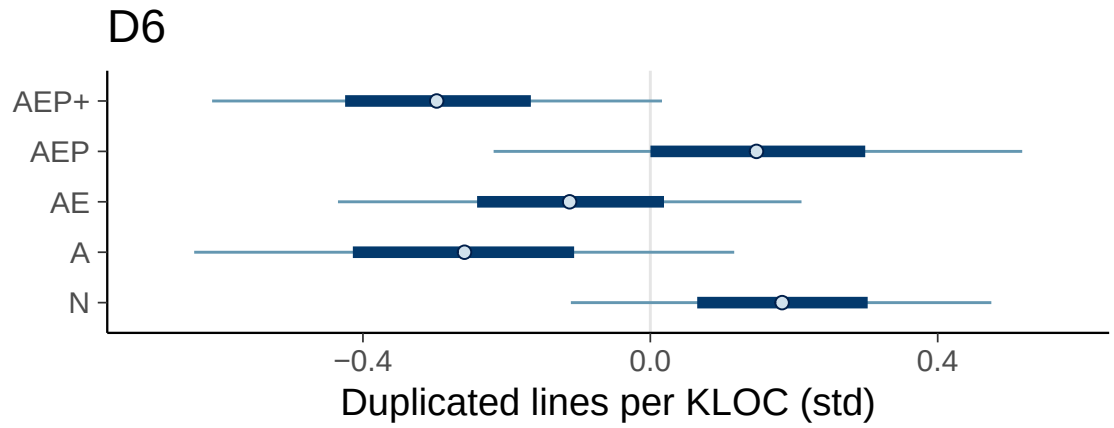

Figure 4.27: Credible intervals of DL/KLOC (standardized) for each major category, from model D6 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

While the results amongst the categories that do use warnings are inconclusive, it still appears that projects that use warnings fare better than those that opt for no compiler warnings, as shown in Figure 4.28. However, this is likely caused by something else than the compiler warnings themselves, as previously discussed.

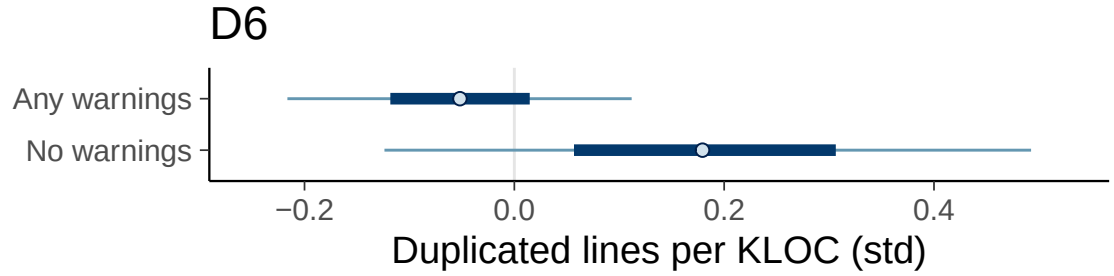

Figure 4.28: Credible intervals of DL/KLOC (standardized) for projects using warnings and those that do not, obtained from model D6 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

4.2.11 Technical Debt Ratio

All statistical models designed for the technical debt ratio metric are presented in Table 4.26. Unlike most other metrics used in the study, the outcome variable for these models is not scaled according to the code size of the projects. Instead, the outcome variable is simply the standardized technical debt ratio (TDR), introduced in Section 3.3.1 (lower values are better). Due to the similarity in predictive performance for several models, TD1 is chosen due to its simplicity, even if TD3 and TD2 provide slightly better predictions.

Model	Parameters	ELPD (LOO)	Δ ELPD
TD3	C	-190.2	0.0
TD2	F	-190.2	0.0
TD1	None	-190.4	-0.2
TD4	Z	-190.4	-0.2
TD7	C, F	-190.6	-0.4
TD6	S	-190.8	-0.6
TD8	C, S	-190.8	-0.7
TD5	T	-191.3	-1.2

Table 4.26: Overview of candidate models for the TDR metric. The models are ordered by their ELPD scores (higher is better), indicating their predictive accuracy.

Using TD1 with the complete sample results in the estimates listed in Table 4.27. Assuming that compiler warnings do positively impact code quality when used, it is reasonable to expect a clear gradual improvement in the TDR scores given stricter warning sets, given the aggregate nature of the TDR metric. Categories N, A, and AEP all have quite similar TDR scores of around 2, while AE and AEP+ distinguish themselves as the best categories. The difference between AEP and AEP+ is strange but can be explained by an outlier in AEP, see Figure D.11. Nevertheless, the difference between N and AEP+ is clear, at 0.76.

Category	Mean	SD	0.05 CI	0.95 CI
N	2.16	2.06	1.74	2.57
A	2.05	2.13	1.51	2.59
AE	1.70	2.09	1.24	2.16
AEP	2.12	2.13	1.59	2.65
AEP+	1.40	2.08	0.94	1.85

Table 4.27: Summary of results for major categories, from model TD1 using the complete sample. Categories N and AEP+ have the worst and best means, with TDR scores of 2.16 and 1.40, respectively.

The intervals for the major categories when using the complete sample are visualized in Figure 4.29. There is a clear gradient amongst the major categories, but the aforementioned outlier in AEP appears to skew the interval of AEP to the right. The outlier, in this case, is `clipp`, which is a recurring outlier for several metrics. Another round of analysis is conducted without `clipp` to determine the influence of said outlier.

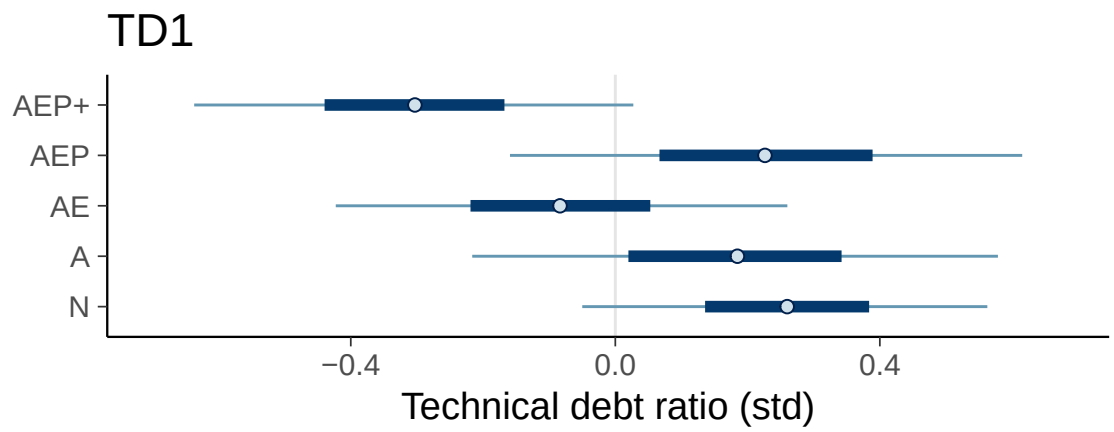

Figure 4.29: Credible intervals of TDR for each major category, from model TD1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

Regardless of the outlier in AEP, the model appears confident that projects that use warnings in some capacity have lower TDR than projects that do not use warnings, see Figure 4.30.

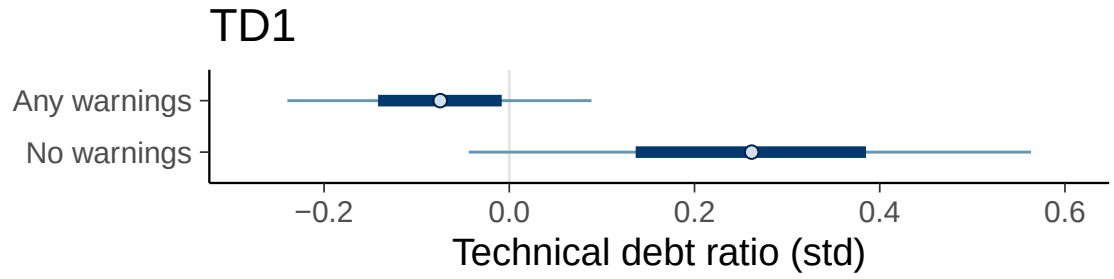

Figure 4.30: Credible intervals of TDR for projects that use warnings and those that do not, obtained from model TD1 using the complete sample. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

Results from applying TD1 on a sample excluding `clipp` are provided in Table 4.28. AEP now has a TDR score significantly closer to that of AEP+, going from 2.12 to 1.60. The other categories retain remarkably similar estimates, with N and AEP+ still the worst and best categories, respectively.

Category	Mean	SD	0.05 CI	0.95 CI
N	2.15	1.94	1.82	2.49
A	2.05	2.00	1.61	2.48
AE	1.69	1.95	1.33	2.05
AEP	1.60	2.00	1.16	2.04
AEP+	1.39	1.95	1.02	1.76

Table 4.28: Summary of results for major categories, from model TD1 using a sample that excludes `clipp`. AEP+ and N are respectively the best and worst categories.

The gradient from N to AEP+ becomes very obvious when `clipp` is removed, as shown in Figure 4.31. The fact that this gradual improvement is clearly visible is a positive sign for the hypothesis that compiler warnings usage correlates with improved code quality. Note that the intervals for AEP+ and N are completely separated, i.e., there is no overlap in the intervals. As such, the model is very confident that projects in AEP+ have better TDR than projects that do not use warnings.

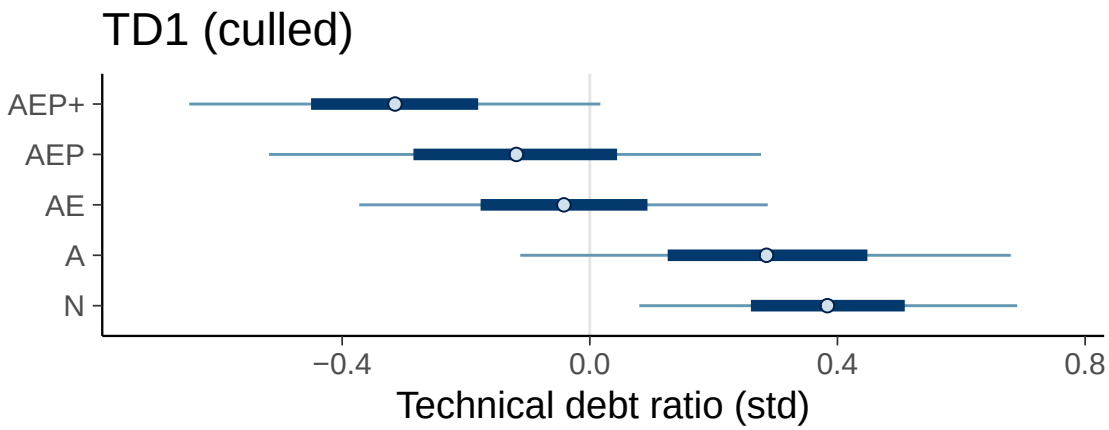

Figure 4.31: Credible intervals of TDR for each major category, from model TD1 using a sample that excludes `clipp`. The inner and outer regions represent 50% and 90% of the probability mass, respectively.

The model still expects projects that use warnings to have lower TDR compared to projects that do not use them when the culled sample is used, see Figure 4.32. It is worth noting that the intervals are completely separated when using the culled sample, but the trend is clear regardless.

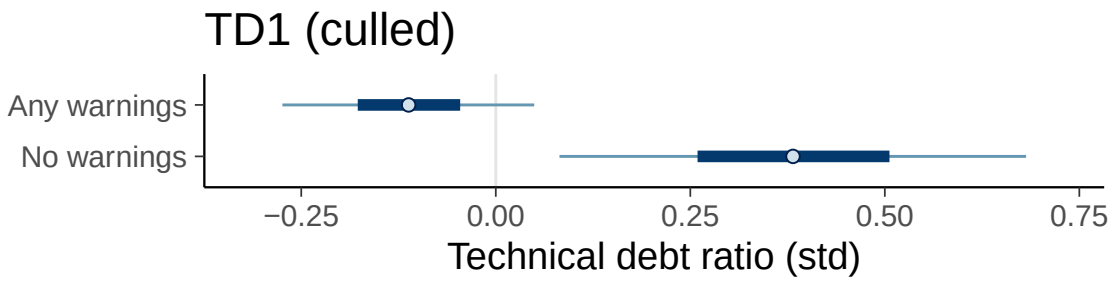

Figure 4.32: Credible intervals of TDR for projects that use warnings and those that do not, obtained from model TD1 using a sample that excludes `clipp`. The inner and outer regions represent 50% and 90% of the probability mass, respectively..

4.3 Summary

An overview of all obtained results is provided in Table 4.29, which lists the best and worst categories for each investigated code quality metric. AEP+ is the best category for most metrics, while N is usually the worst category. At the very least, this implies that there is some correlation between the usage of compiler warnings and improved code quality metrics.

Metric	Worst Category	Best Category	Difference
B/KLOC	N (0.97)	AEP (0.37)	0.60
CS/KLOC	AEP (87.95)	AEP+ (54.31)	33.64
CS/KLOC (culled)	N (73.83)	AEP+ (44.20)	29.63
CV/KLOC	N (23.97)	AEP+ (11.89)	12.10
MaV/KLOC	AEP (52.49)	AEP+ (17.90)	34.59
MaV/KLOC (culled)	A (29.48)	AEP+ (16.53)	12.95
MiV/KLOC	N (24.10)	AEP+ (16.74)	7.36
SH/KLOC	N (0.97)	AE (0.48)	0.49
V/KLOC	A (0.022476)	AEP+ (0.000951)	0.021525
Cyc/KLOC	A (178.42)	N (137.50)	40.92
Cyc/KLOC (culled)	A (177.50)	AEP+ (132.80)	44.70
Cog/KLOC	AEP (141.08)	AEP+ (75.95)	65.13
DL/KLOC	N (116.38)	AEP+ (58.28)	58.1
TDR	N (2.16)	AEP+ (1.40)	0.76
TDR (culled)	N (2.15)	AEP+ (1.39)	0.76

Table 4.29: Summary of mean metric results, listing the best and worst out of the major categories. Metrics that have been analyzed using both the full and a culled sample have both cases are provided in the table. In general, AEP+ and N are the best and worst categories, respectively.

5
Discussion

Given the results from the study, answers to the research questions can be summarized as follows.

To what extent are compiler warnings used in C++ projects?
78% (99/127) of sampled projects use compiler warnings to some extent, leaving 22% (28/127) of the sampled projects with no compiler warnings enabled.

Does compiler warning usage correlate with C++ code quality?
The usage of compiler warnings correlates with improved measures concerning bugs, code smells, and technical debt. Projects with any compiler warnings enabled generally exhibit improved code quality characteristics.

What is the optimal level of compiler warnings in C++ projects?
The optimal compiler warnings appear to be the strictest ones possible, preferably treating warnings as errors through flags such as `-Werror` to ensure that warnings cannot be ignored.

In general, the usage of stricter compiler warnings in C++ projects correlates with improved code quality. Observed projects that do not make use of compiler warnings generally perform worse on average than projects using compiler warnings, especially regarding metrics such as bugs, code smells, and technical debt. As a result, the existing practice of using the presence of compiler warnings as a way to detect code quality issues appears reasonable, as in Melo et al. [18]. Interestingly, projects using compiler warnings perform better when it comes to metrics that are unlikely to be directly affected by compiler warnings, such as security hotspots and duplicated lines. However, there are metrics for which projects with no compiler warnings prevail, such as vulnerabilities, cognitive complexity, and cyclomatic complexity. The most convincing piece of evidence for a correlation between stricter compiler warning usage and improved code quality is the technical debt ratio results (due to the aggregate nature of this metric), which indicate a gradual decrease in technical debt as the compiler warnings used become stricter.

The results of the study are applicable to most C++ projects and, to some extent, C projects. Naturally, the results should be generalizable to other C++ projects, not just those hosted on GitHub. In addition to C++ projects, it is reasonable to assume that the findings would at least partially apply transfer to C projects as well. While there are clear differences between C and C++, the compilers considered in this study (MSVC, Clang, and GCC) all provide C front ends with very similar compiler warning interfaces. This assumption also builds on the fact that

C++ is essentially a superset of C, i.e., most C code is valid C++ code. For example, the program included in the motivating example is also a C program. However, it is difficult to argue for the applicability of the results to other programming languages without additional research.

External factors such as engineering culture could explain the superior performance of AEP+ compared to AEP, especially for metrics that are unlikely to be influenced by compiler warnings. Projects that use compiler warnings generally fare better in terms of code quality, even when compiler warnings themselves are unlikely to be helpful, such as for the number of duplicated lines. This could be explained by varying engineering cultures among sampled projects. Development teams with rigorous quality-minded workflows could be more inclined to utilize tools such as linters or static code analysis software like SonarCloud or SonarQube, in addition to opting for stricter compiler warnings. This would of course influence the results for projects that actively work to improve their measured code quality characteristics. Furthermore, the *perceived* quality of projects by contributors can influence the overall quality of contributions [39]. This means that developers might settle for lower-quality additions if they feel that the existing code is already of poor quality, and vice versa. It is possible that projects that utilize strict compiler warnings appear more quality conscious and therefore influence the efforts of contributors.

Treating compiler warnings as errors appear to significantly improve measured code quality characteristics. The difference in results between AEP and AEP+ is surprising given the fact that these categories use the same set of warnings, which may be explained by software developers ignoring compiler warnings unless they have to fix them. When using `-Werror`, as in AEP+, software developers simply have no choice regarding this matter. As a result, it might be the case that more relaxed warnings, as in categories A or AE, could be nearly as effective as AEP+ if they would include `-Werror`. However, these configurations appear quite rare in practice, as only 2% (3/127) and 6% (8/127) of sampled projects belong to A+ and AE+, respectively. Developers might not be inclined to fix warnings when there already exist multiple emitted warnings. One additional warning might not feel as important when you already have hundreds lined up [39]. The efficacy of `-Werror` regarding code quality is worthy of additional research, and will likely require a larger sample. Additionally, the amount of emitted compiler warnings across various configurations is a potential topic of future research to provide evidence for the hypothesis that developers ignore compiler warnings.

The cost of migrating to treating warnings as errors should be explored. While projects that treat warnings as errors were found to exhibit improved code quality, it is not trivial to make such a change given a project with thousands of unresolved warnings. Committing to treating all warnings as errors is inherently inflexible and might put too much focus on compiler warnings themselves, creating a false sense of security. This could be especially problematic due to the differences in opinions among developers regarding warning types, as outlined by Danilova et

al. [3]. After all, the use of compiler warnings is just one approach to improving code quality. Furthermore, few projects are likely to accept an extended period of development where the code base does not compile due to compiler warnings. Possible solutions could be to treat a gradually increasing subset of warnings as errors in order to reduce the risk of excessive errors due to pedantic warnings. Further research should investigate the effectiveness of treating warnings as errors for more relaxed warning configurations, such as `-Wall`.

Developers should enable compiler warnings in their C++ projects. The results of this study indicate that projects that opt to enable compiler warnings generally feature better code quality. Additionally, existing C++ projects that already use some compiler warnings should consider enabling stricter warnings if possible. This recommendation applies to C projects as well, due to the aforementioned similarities in the available compiler warnings and language foundations. While this study focused on C++ projects, it would not be surprising to see similar correlations between compiler warning usage and improved code quality for other programming languages. However, unlike some previous research, such correlations should not be assumed to hold without evidence. Researchers may continue to use compiler warnings as a heuristic for problematic code for C++ projects, but additional research is required to clarify causal relationships.

5.1 Threats to Validity

This section discusses potential threats to the validity of the study. Four different types of validity threats are discussed, as per the recommendations of Wohlin et al. [40].

5.1.1 Conclusion Validity

> Conclusion validity concerns the correctness of the statistical analysis applied in a study.

The Bayesian statistical models and analysis methods that have been used in the study are well-tested and proven. Reasonable assumptions were made concerning prior selection and choice of distributions, i.e., the observed data did not influence the choice of these aspects. The study has made use of good practices such as prior predictive checks and model comparisons, generally avoiding too many conditioned parameters. A causal DAG has been utilized to explicitly state causal assumptions. Furthermore, statistical diagnostic tools such as R-hat and MCMC chain convergence were validated. Conclusions from the data were deduced from the sample as a whole, not by looking at individual sampled projects. Analysis results from using a culled sample were always presented in addition to the results from the complete sample. The uncertainty of measurements is built into Bayesian statistical models, illustrated in the provided credible intervals. Given these points, it is reasonable to argue for sound conclusion validity of the study results.

5.1.2 Internal Validity

> Internal validity concerns factors that could affect independent variables without researchers knowing about them.

This study is a sample study, not a randomized controlled trial, so control of all variables was not possible. One of the primary suspected confounders was engineering culture since this was deemed as a likely explanation for differences in measured code quality for metrics where compiler warnings are not expected to be effective. The found correlation between the usage of stricter compiler warnings and improved code quality is valid, but it is likely affected by additional factors such as engineering culture. In general, this study could not make any causal claims regarding compiler warning usage and code quality, this could be the focus of additional research.

The manual aspects of certain steps in the applied methodology, such as categorization and sample selection, are inherently error-prone. For example, some projects in the sample could have been placed in the wrong warning usage category. Additionally, projects that were excluded according to the exclusion criteria might have been valid. It is near impossible to guarantee that no such errors were made. However, scripts were devised in an attempt to provide automatic heuristics regarding potential compiler warning usage in order to limit such errors. Furthermore, it is unlikely that projects that should have been placed in AEP+ were mistaken as using no warnings. Instead, it is more likely that projects were mistakenly placed in similar categories, e.g., AEP instead of AEP+. This aspect of the study would heavily benefit from an automated tool capable of reliably extracting compiler warning usage information from C++ projects.

The fact that the algorithms concerning the collection of the code quality metrics are not open-source is an issue. The correspondence between the code quality metric estimates reported by SonarCloud and the "true" code quality level is subsequently difficult to validate. While tools such as SonarQube and SonarCloud are recurring in existing research, the lack of insight is problematic due to the inability to judge the soundness of their classification schemes. The use of an open-source alternative in addition to SonarCloud would increase the level of transparency and therefore the level of trust that the code quality metrics are representative and reliable. Furthermore, the amount of reported code quality issues that are false positives could be an issue. It could be argued that it is in the interest of tools such as SonarCloud to report as many issues as possible, regardless of their accuracy. After all, SonarCloud is a paid service for private projects. It is plausible that reporting more issues leads to a greater sense of urgency among organizations to buy their services. The noise caused by false positives can hopefully be assumed to be somewhat constant across compiler warning usage categories. Therefore, the results of the study should be unaffected since the relative measures would remain similar regardless of the noise.

Outliers in the sample were found to have a substantial influence on the obtained results if included. Generally, these outliers consisted of samples with smaller code

sizes, which increases the sensitivity to outliers since most metrics were scaled by code size in the study. Outlier samples also seemed to have their offending source code concentrated in a handful of files, with most files exhibiting average metrics. The overall results of the study are unlikely to have been influenced by these outliers due to the two analysis passes with and without said outliers. Further research into the underlying causes for some projects to exhibit extreme code quality metrics is warranted. Notably, a small set of sampled projects constituted the outliers for several metrics. In other words, anomalous projects generally exhibited extraordinary measures for more than one code quality metric.

5.1.3 Construct Validity

> Construct validity concerns the ability of a study to answer its research questions.

The study design used has been well-tested, especially in the domain of repository mining studies. One possible risk factor in this study was the measurement of code quality, due to the difficulty of reliably estimating it. There are several possible approaches to code quality estimation, static code analysis tools such as SonarCloud provide one such approach. These kinds of tools have been used extensively in similar studies and have generally been regarded as a viable way to obtain code quality estimates. Other approaches might have yielded different code quality estimates, which could have affected the general findings. Even if the code quality estimates used in this study are not perfect, they are regarded as reliable enough to make general conclusions concerning detected correlations.

5.1.4 External Validity

> External validity concerns the generalizability of results from a study.

The purposive sampling strategy utilized in this study inherently limits the statistical generalizability of the findings due to its subjective nature. At the same time, the choice of sampling strategy led to a sample consisting of serious and realistic C++ projects. The manual categorization and code compilation steps necessitated a method of collecting representative samples quickly due to time restraints, which the chosen sampling approach enabled. Greater sample sizes would require automated warning usage categorization and compilation, which is a difficult process to automate due to the irregular and unpredictable structure of C++ projects. Therefore, it was deemed that a purposive sampling approach would lead to more generalizable results than a purely random sampling approach. The exclusion criteria applied when sampling projects were designed with practicality in mind, e.g., non-English projects and projects that could not be compiled were excluded. These requirements could have affected the generalizability of the results.

Out of the total sample of 127 projects, 28 were determined to not make use of

compiler warnings. The remaining 99 projects used compiler warnings to varying extents. This size disparity increased the uncertainty of comparisons between these two groups as a whole. However, these comparisons were only included in addition to the per-category results, which were much more evenly distributed concerning sample size. Again, a greater sample size would have improved the confidence in the findings for projects that do not use warnings, even if the relative sample sizes would likely remain similar.

The choice to only consider GitHub projects could hurt the generalizability of the findings. While GitHub is a popular platform for hosting Git repositories, there are popular alternatives such as Bitbucket and GitLab. The usage of GitHub Actions to compile and analyze sample projects was the primary factor that limited the possibility to consider projects hosted on other platforms. Overall, it was deemed unlikely that projects hosted on GitHub would differ drastically from projects hosted on other platforms. Nevertheless, further research should consider sampling projects from multiple platforms.

The results often indicated better code quality for AEP+ compared to AEP. This sparked the hypothesis that treating warnings as errors is beneficial for overall code quality. However, categories using `-Werror` (other than AEP+) contained a limited amount of samples. For example, A+ consisted of merely three samples. The relative scarcity of projects that treat warnings as errors make it difficult to conclude whether treating warnings as errors is always effective. Ideally, the sample would contain categories of similar sizes where each category had a corresponding category that treats warnings as errors.

Another aspect of the study that was susceptible to mistakes was the estimation of changes to compiler warning usage among sampled projects. An arbitrary threshold of six months was enforced, where any changes to the warning configuration of projects that fell within this threshold were ignored. The aforementioned scripts used Git to extract the dates of changes to configuration files as a heuristic for likely changes to the used compiler warnings. Regardless, the process of verifying recent changes to compiler warnings was inherently error-prone. Five projects were identified as having recently changed their warning configurations, which at least proved the ability of this process to detect projects with such changes. A different choice of threshold could have affected the analysis results.

5.2 Future Research

Given the scarce research into the effects of compiler warning usage, there are several possible research topics to consider for future research. One such topic could be the causal relationships between the usage of compiler warnings and code quality. An experiment could be devised where a system is developed with and without compiler warnings enabled to be able to compare the resulting code quality in the two versions. Such an experiment could provide strong indications regarding the true effects of enabling compiler warnings which would make it easier to argue for

(or against) the soundness of compiler warnings.

Another interesting approach could be to investigate the rate at which developers address emitted compiler warnings. This would tie in with the results of this study where projects that treat warnings as error fare substantially better than similar projects that do not treat warnings as errors. Developers may become more prone to address warnings if they are convinced that code quality will improve by doing so. Similarly, the effects of emitted compiler warnings on developer productivity could be an interesting topic of future research. That is, emitted warnings might be deemed cryptic and difficult to understand, possibly slowing down development. The general expectations and attitudes of developers towards compiler warnings are potentially interesting targets for qualitative research. It would have been interesting to include interviews with developers from sampled projects to get additional insights into the thought process behind their approach to compiler warnings and code quality in general. This could provide some insight into potential causal relationships.

The impact of changes to existing compiler warning configurations is another interesting aspect. For example, the time it takes for compiler warning configuration changes to affect code quality and the magnitude of those changes are aspects that could be investigated. This could provide better guidelines for thresholds regarding compiler warning usage categorization, whereas this study utilized an arbitrary threshold of six months for changes to warning configurations. Furthermore, such research could provide useful evidence regarding the advantages of making adjustments to their current warning configurations. Especially if it turns out that small adjustments could have a significant impact on code quality.

6
Conclusion

The usage of compiler warnings has widely been assumed to have a positive effect on code quality, but this notion has little evidence in research. The purpose of this study was to investigate the relationship between the usage of compiler warnings and overall code quality in C++ projects and to provide empirical evidence.

127 C++ projects were sampled using a *purposive* sampling approach, and categorized according to the warnings flags configured for the project. The projects were selected from the projects with the most stargazers on GitHub according to the specified criteria for the study. Code quality was then measured from the project using SonarCloud, resulting in different metrics that were used to approximate the code quality in different aspects of the projects. Finally, these metrics were analyzed, utilizing Bayesian data analysis, to enquire results about the relationship between the warning flags and the code quality.

The analysis showed a correlation between stricter compiler warning usage and improved code quality, mainly looking at reported bugs, code smells, and technical debt. Projects with any compiler warnings enabled generally had improved code quality compared to projects without warnings enabled. When it comes to the usage of compiler warnings it was concluded that 78% of the sampled projects used compiler warnings and 22% had no compiler warnings enabled. Looking at the optimal configuration of compiler warnings, it was concluded that the analysis indicates that the strictest warning setting with `-Werror` or equivalent flag appears to be the most optimal warning configuration.

Further research into the effects of compiler warnings on different aspects of code quality is needed to make better recommendations regarding specific compiler warning flags. Additionally, further investigations into the relationship between compiler warning usage and engineering culture are also needed. This study aims to act as a first step in exploring the otherwise unexplored terrain of compiler warnings.

Bibliography

[1] GCC, "GCC, the GNU Compiler Collection," 2023, [Online]. Available: https://gcc.gnu.org/ (accessed on 2023-03-08).

[2] LLVM, "Clang," 2023, [Online]. Available: https://clang.llvm.org/ (accessed on 2023-03-08).

[3] A. Danilova, A. Naiakshina, and M. Smith, "One Size Does Not Fit All: A Grounded Theory and Online Survey Study of Developer Preferences for Security Warning Types," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 136–148.

[4] G. Kudrjavets, A. Kumar, N. Nagappan, and A. Rastogi, "The Unexplored Terrain of Compiler Warnings," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022, pp. 283–284.

[5] R. Moser, B. Russo, and G. Succi, "Empirical analysis on the correlation between GCC compiler warnings and revision numbers of source files in five Industrial Software Projects," *Empirical Software Engineering*, vol. 12, no. 3, pp. 295–310, 2006.

[6] T. R. Team, "Rust," 2023, [Online]. Available: https://www.rust-lang.org (accessed on 2023-03-17).

[7] The Clang Team, "clang-tidy," 2023, [Online]. Available: https://clang.llvm.org/extra/clang-tidy (accessed on 2023-03-17).

[8] SonarQube, "SonarQube Documentation," 2023, [Online]. Available: https://docs.sonarqube.org/latest/ (accessed on 2023-03-17).

[9] ISO, "Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model," International Organization for Standardization, Standard, March 2011.

[10] C. Cameron, "Sonarcloud or sonarqube? - guidance on choosing one for your team," Apr 2020. [Online]. Available: https://www.sonarsource.com/blog/sq-sc_guidance/

[11] GitHub, "GitHub Actions," 2023, [Online]. Available: https://docs.github.com/en/actions (accessed on 2023-03-10).

[12] R. McElreath, *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*. Chapman & Hall/CRC, 2020.

[13] J. K. Kruschke and T. M. Liddell, "Bayesian data analysis for newcomers," *Psychonomic Bulletin & Review*, vol. 25, no. 1, pp. 155–177, 2017.

[14] D. B. Dunson, "Commentary: Practical Advantages of Bayesian Analysis of Epidemiologic Data," *American Journal of Epidemiology*, vol. 153, no. 12, pp.

1222–1226, 06 2001. [Online]. Available: https://doi.org/10.1093/aje/153.12.1222

[15] J. Pearl, *Causal Inference in Statistics : A Primer.*, ser. New York Academy of Sciences Ser. John Wiley and Sons, Incorporated, 2016. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=cat07472a&AN=clec.EBC7104473&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds

[16] Y. Tang, H. Jiang, Z. Zhou, X. Li, Z. Ren, and W. Kong, "Detecting Compiler Warning Defects Via Diversity-Guided Program Mutation," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4411–4432, 2022.

[17] C. Sun, V. Le, and Z. Su, "Finding and Analyzing Compiler Warning Defects," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 203–213.

[18] J. Melo, E. Flesborg, C. Brabrand, and A. Wąsowski, "A Quantitative Analysis of Variability Warnings in Linux," in *Proceedings of the 10th International Workshop on Variability Modelling of Software-Intensive Systems*, ser. VaMoS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 3–8. [Online]. Available: https://doi.org/10.1145/2866614.2866615

[19] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin, "Do Developers Read Compiler Error Messages?" in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 575–585.

[20] T. Barik, D. Ford, E. Murphy-Hill, and C. Parnin, "How Should Compilers Explain Problems to Developers?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 633–643. [Online]. Available: https://doi.org/10.1145/3236024.3236040

[21] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "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)*, 2019, pp. 209–219.

[22] X. Zhang, J. Yan, B. Cui, J. Yan, and J. Zhang, "Are the Scala Checks Effective? Evaluating Checks with Real-world Projects," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021, pp. 978–989.

[23] A. Kallingal Joshy, X. Chen, B. Steenhoek, and W. Le, "Validating Static Warnings via Testing Code Fragments," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, pp. 540–552. [Online]. Available: https://doi.org/10.1145/3460319.3464832

[24] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are SonarQube Rules Inducing Bugs?" in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 501–511.

[25] S. Baltes and P. Ralph, "Sampling in software engineering research: a critical review and guidelines," *Empirical Software Engineering*, vol. 27, no. 4, p. 94, 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10072-8

[26] D. Pina, A. Goldman, and C. Seaman, "Sonarlizer Xplorer: a tool to mine Github projects and identify technical debt items using SonarQube," in *2022 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2022, pp. 71–75.

[27] K.-J. Stol and B. Fitzgerald, "The ABC of Software Engineering Research," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, pp. 1–51, 2018.

[28] Microsoft, "Microsoft C++, C, and Assembler documentation," 2023, [Online]. Available: https://learn.microsoft.com/en-us/cpp/ (accessed on 2023-03-08).

[29] GCC, "Warning Options," 2023, [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html (accessed on 2023-05-15).

[30] LLVM, "Diagnostic flags in Clang," 2023, [Online]. Available: https://clang.llvm.org/docs/DiagnosticsReference.html (accessed on 2023-05-15).

[31] Sonar, "SonarCloud," 2023, [Online]. Available: https://www.sonarsource.com/products/sonarcloud/ (accessed on 2023-03-08).

[32] SonarCloud, "Web API," 2023, [Online]. Available: https://sonarcloud.io/web_api (accessed on 2023-03-07).

[33] GitHub, "GitHub API," 2023, [Online]. Available: https://docs.github.com/en/rest?apiVersion=2022-11-28 (accessed on 2023-03-07).

[34] SonarCloud, "Issues," 2023, [Online]. Available: https://docs.sonarcloud.io/digging-deeper/issues (accessed on 2023-03-01).

[35] ——, "Security Hotspots," 2023, [Online]. Available: https://docs.sonarcloud.io/digging-deeper/security-hotspots (accessed on 2023-03-01).

[36] ——, "Metric Definitions," 2023, [Online]. Available: https://docs.sonarcloud.io/digging-deeper/metric-definitions (accessed on 2023-03-01).

[37] LLVM, "The LLVM Compiler Infrastructure," 2023, [Online]. Available: https://llvm.org/ (accessed on 2023-05-04).

[38] The Clang Team, "clang-tidy - readability-function-cognitive-complexity," 2023, [Online]. Available: https://clang.llvm.org/extra/clang-tidy/checks/readability/function-cognitive-complexity.html (accessed on 2023-03-07).

[39] W. Levén, H. Broman, T. Besker, and R. Torkar, "The Broken Windows Theory Applies to Technical Debt," 2022. [Online]. Available: https://arxiv.org/abs/2209.01549

[40] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*, 1st ed. Berlin, Heidelberg : Springer Berlin Heidelberg : Imprint: Springer, 2012., 2012.

A

Resources

This appendix lists publicly available resources related to the thesis.

- The replication package can be accessed online[1], hosted on GitHub Pages.
- Complete R source code for the statistical analysis can be found in the `impact-of-compiler-warnings-thesis/replication`[2] GitHub repository.
- The complete sample of C++ projects are additionally provided as "forks" of the `impact-of-compiler-warnings-thesis`[3] GitHub organization.

[1] `https://impact-of-compiler-warnings-thesis.github.io/replication`
[2] `https://github.com/impact-of-compiler-warnings-thesis/replication`
[3] `https://github.com/impact-of-compiler-warnings-thesis`

B

Included Projects

This appendix lists all GitHub projects that were included in the sample.

B.1 Category N

1. dragonflydb/dragonfly
2. CoatiSoftware/Sourcetrail
3. hoffstadt/DearPyGui
4. idea4good/GuiLite
5. oatpp/oatpp
6. flashlight/flashlight
7. cemu-project/Cemu
8. amrayn/easyloggingpp
9. KDAB/hotspot
10. NVIDIA/cutlass
11. Vita3K/Vita3K
12. p-ranav/indicators
13. johnBuffer/AntSimulator
14. RuntimeCompiledCPlusPlus/RuntimeCompiledCPlusPlus
15. patr0nus/DeskGap
16. hosseinmoein/DataFrame
17. jpd002/Play-
18. TianZerL/Anime4KCPP
19. verilator/verilator
20. bloomberg/bde
21. mandreyel/mio
22. cpeditor/cpeditor
23. David-Haim/concurrencpp
24. onlytailei/CppRobotics
25. symforce-org/symforce
26. matt-42/lithium
27. mutouyun/cpp-ipc
28. githubuser0xFFFF/Qt-Advanced-Docking-System

B.2 Category A

1. dolphin-emu/dolphin
2. citra-emu/citra
3. arrayfire/arrayfire
4. MaskRay/ccls
5. LizardByte/Sunshine
6. LibreSprite/LibreSprite
7. xelatihy/yocto-gl
8. manticoresearch/manticoresearch
9. OpenXRay/xray-16
10. plibither8/2048.cpp
11. jlblancoc/nanoflann
12. skypjack/uvw
13. magiblot/tvision
14. CVCUDA/CV-CUDA
15. ArthurSonzogni/Diagon
16. FEX-Emu/FEX

B.3 Category AE

1. yuzu-emu/yuzu
2. catchorg/Catch2
3. musescore/musescore
4. yhirose/cpp-httplib
5. PointCloudLibrary/pcl
6. Alinshans/MyTinySTL
7. PCSX2/pcsx2
8. organicmaps/organicmaps
9. asmjit/asmjit
10. OGRECave/ogre
11. eidheim/Simple-Web-Server
12. paceholder/nodeeditor
13. NVIDIA/libcudacxx
14. dfeneyrou/palanteer
15. fnc12/sqlite_orm
16. syoyo/tinygltf
17. martinus/robin-hood-hashing
18. moneymanagerex/moneymanagerex
19. decaf-emu/decaf-emu
20. blend2d/blend2d
21. InsightSoftwareConsortium/ITK
22. mhx/dwarfs
23. Cylix/cpp_redis
24. paullouisageneau/libdatachannel

B.4 Category AP

1. skypjack/entt
2. sharkdp/dbg-macro
3. PolyMC/PolyMC
4. VcDevel/Vc

B.5 Category AEP

1. VowpalWabbit/vowpal_wabbit
2. olive-editor/olive
3. juce-framework/JUCE
4. danmar/cppcheck
5. PrismLauncher/PrismLauncher
6. dpilger26/NumCpp
7. cycfi/elements
8. nextcloud/desktop
9. ihhub/fheroes2
10. Corvusoft/restbed
11. wxFormBuilder/wxFormBuilder
12. CrowCpp/Crow
13. eProsima/Fast-DDS
14. epasveer/seer
15. variar/klogg
16. godlikepanos/anki-3d-engine
17. muellan/clipp

B.6 Category AE+

1. nlohmann/json
2. OpenRCT2/OpenRCT2
3. isl-org/Open3D
4. microsoft/cpprestsdk
5. Neargye/nameof
6. ETLCPP/etl
7. nfrechette/acl
8. sewenew/redis-plus-plus

B.7 Category AEP+

1. CMU-Perceptual-Computing-Lab/openpose
2. ethereum/solidity
3. gabime/spdlog
4. polybar/polybar

5. nasa/fprime
6. g-truc/glm
7. Project-OSRM/osrm-backend
8. libcpr/cpr
9. doctest/doctest
10. arvidn/libtorrent
11. ArthurSonzogni/FTXUI
12. tinyobjloader/tinyobjloader
13. ChaiScript/ChaiScript
14. KhronosGroup/Vulkan-Hpp
15. ccache/ccache
16. Dobiasd/FunctionalPlus
17. taocpp/PEGTL
18. boostorg/hana
19. rpclib/rpclib
20. Nelarious/imnodes
21. aantron/better-enums
22. tfussell/xlnt
23. KomputeProject/kompute
24. gulrak/filesystem

B.8 Category AP+

1. boostorg/compute

B.9 Category E+

1. andreasfertig/cppinsights

B.10 Category A+

1. rapidsai/cudf
2. endless-sky/endless-sky
3. jrouwe/JoltPhysics

B.11 Category E

1. xtensor-stack/xsimd

C

Excluded projects

This appendix lists all GitHub projects that were excluded from the sample.

C.1 Non-CMake projects

1. carbon-language/carbon-lang
2. microsoft/calculator
3. xenia-project/xenia
4. tensorflow/serving
5. includeos/IncludeOS
6. FlaxEngine/FlaxEngine
7. microsoft/WindowsAppSDK
8. mrousavy/react-native-mmkv
9. lewisbaker/cppcoro
10. ARM-software/ComputeLibrary
11. seladb/PcapPlusPlus
12. kth-competitive-programming/kactl
13. skift-org/skift
14. unicode-org/icu
15. Tencent/TscanCode
16. PanosK92/SpartanEngine
17. wichtounet/thor-os
18. dyanikoglu/ALS-Community
19. FarGroup/FarManager
20. stefanhaustein/TerminalImageViewer
21. eventql/eventql
22. treefrogframework/treefrog-framework
23. bshoshany/thread-pool
24. boostorg/pfr
25. felixguendling/cista
26. JustasMasiulis/lazy_importer
27. Tw1ddle/geometrize
28. adriengivry/Overload
29. novak-99/MLPP
30. marzer/tomlplusplus
31. fossephate/JoyCon-Driver

C.2 Non-compilable projects

Projects in the following list were excluded on the basis that they were not compilable.

1. x64dbg/x64dbg (only windows)
2. typesense/typesense
3. vesoft-inc/nebula
4. redpanda-data/redpanda
5. ethereum/aleth
6. ydb-platform/ydb
7. jacobdufault/cquery
8. oneapi-src/oneDNN
9. f1xpl/openauto
10. AshampooSystems/boden
11. appleseedhq/appleseed
12. steemit/steem
13. kyleneideck/BackgroundMusic
14. skyline-emu/skyline
15. microsoft/EdgeML
16. dhapyshev/aspia
17. securesocketfunneling/ssf
18. AshampooSystems/boden
19. eclipse-iceoryx/iceoryx
20. agauniyal/rang
21. superpoweredSDK/Low-Latency-Android-iOS-Linux-Windows-tvOS-macOS-Interactive-Audio-Platform
22. Cxbx-Reloaded/Cxbx-Reloaded
23. microsoft/DirectXTex
24. Artikash/Textractor
25. deepmodeling/deepmd-kit
26. ethz-adrl/control-toolbox
27. zeldaret/botw
28. danielkrupinski/Osiris

C.3 Projects with non-English documentation

These projects featured no English documentation and as such were excluded.

1. Qv2ray/Qv2ray
2. qinguoyi/TinyWebServer
3. oceanbase/oceanbase
4. szad670401/HyperLPR
5. urho3d/urho3d
6. ZachL1/Bilibili-Plus
7. 0voice/cpp_new_features
8. sylar-yin/sylar

9. parallel101/course
10. tongtzeho/PKUCourse
11. scarsty/kys-cpp
12. liu-jianhao/Cpp-Design-Patterns
13. ThisisGame/cpp-game-engine-book
14. sanhuohq/AwesomeCpp
15. harvestlamb/Cpp_houjie
16. wondertrader/wondertrader
17. sakura-editor/sakura
18. huangmingchuan/Cpp_Primer_Answers
19. yuesong-feng/30dayMakeCppServer

C.4 Exercises, tutorials, demos, courses

1. huihut/interview
2. TheAlgorithms/C-Plus-Plus
3. changkun/modern-cpp-tutorial
4. ssloy/tinyrenderer
5. microsoft/IoT-For-Beginners
6. wisdompeak/LeetCode
7. BoomingTech/Piccolo
8. kamyu104/LeetCode-Solutions
9. CGAL/cgal
10. JakubVojvoda/design-patterns-cpp
11. KhronosGroup/Vulkan-Samples
12. tomlooman/EpicSurvivalGame
13. Overv/VulkanTutorial
14. rachitiitr/DataStructures-Algorithms
15. mortennobel/cpp-cheatsheet
16. tomloonman/ActionRoguelike
17. ssloy/tinykaboom
18. ssloy/tinyraycaster
19. ssloy/tinyraytracer
20. CppCon/CppCon2020
21. luliyucoordinate/Leetcode
22. GameTechDev/IntroductionToVulkan
23. manishbisht/Competitive-Programming
24. LeadCoding/3-weeks-Google-prep
25. smv1999/CompetitiveProgrammingQuestionBank
26. pptacher/probabilistic_robotics
27. mandliya/algorithms_and_data_structures

C.5 Miscellaneous

1. flashlight/wav2letter

2. ANYbotics/grid_map
3. deepmind/open_spiel
4. facebookresearch/ELF
5. grame-cncm/faust
6. maxbachmann/RapidFuzz
7. NVIDIA/thrust

D

Data Summary

D.1 Bugs

The scores for the bug metric in all sampled projects belonging to the major categories are presented in Figure D.1. Most values generally range from approximately 0.25 to 9.0 bugs per KLOC. Category N has a clear outlier, and category AE has another less obvious outlier, but still quite a bit larger than the rest of the samples in that category. The remaining categories have more evenly distributed values.

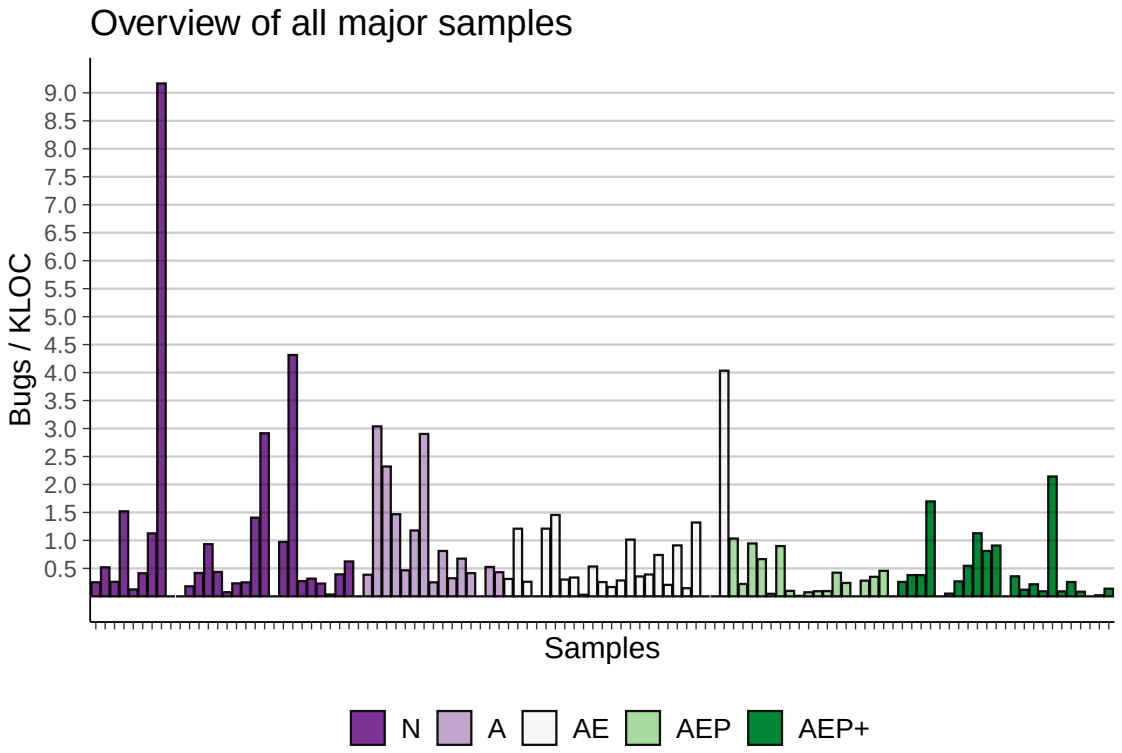

Figure D.1: Overview of bugs per KLOC for each sample in the major categories.

D.2 Code Smells

The amount of code smells per KLOC for each sample in the major categories is shown in Figure D.2. Typical values range from around 20 to 170 code smells per KLOC. Category AEP has a considerable outlier, reaching 628 code smells per

KLOC, about six times larger than the second-largest sample in the same category. Additionally, category AEP+ also has a clear outlier, although not quite as dramatic.

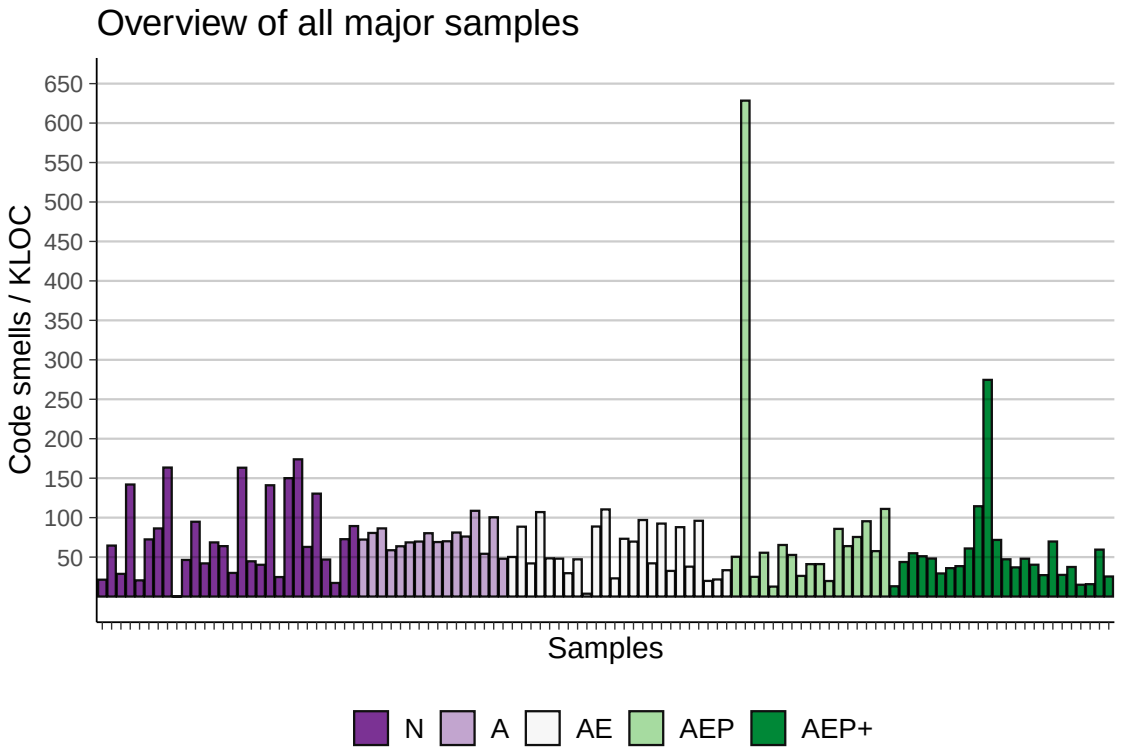

Figure D.2: Overview of code smells per KLOC for each sample in the major categories.

D.3 Critical Violations

The data for the critical violation metric is quite well-behaved, with no clear-cut outliers in the samples belonging to the major categories. Scores for the critical violation metric amongst the major categories are plotted in Figure D.3.

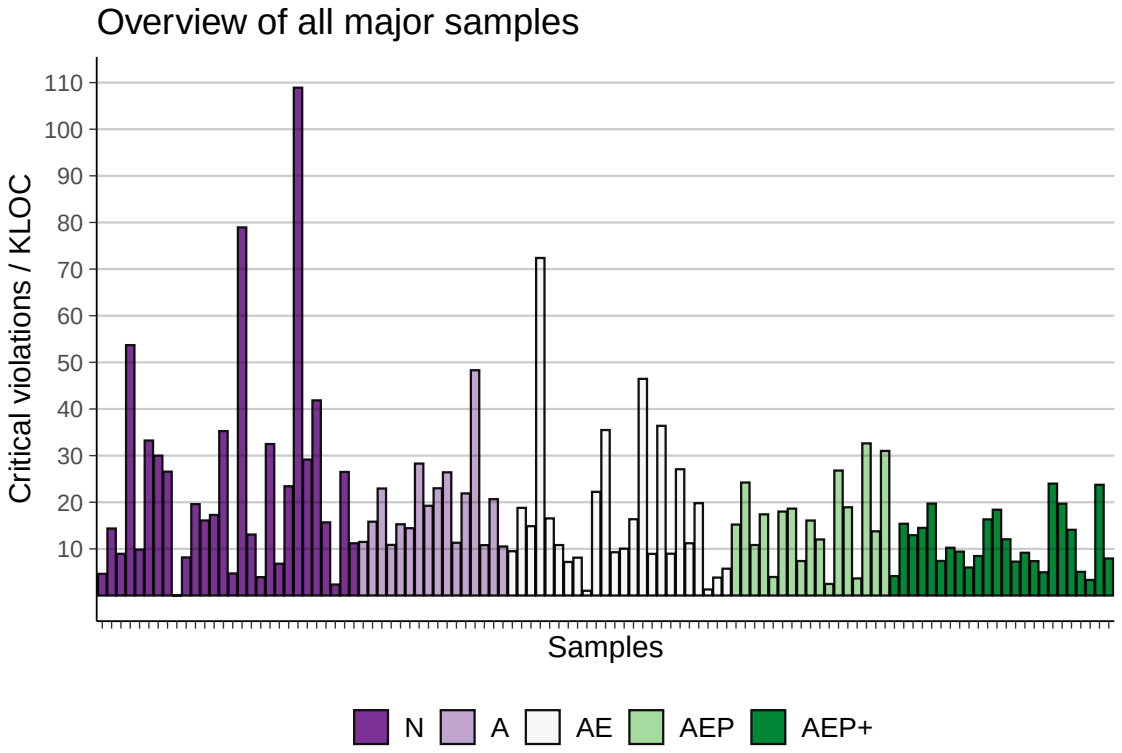

Figure D.3: Overview of critical violations per KLOC for each sample in the major categories.

D.4 Major Violations

Scores for the major violation metric are presented in Figure D.4. There are two clear outliers in the data, which can be observed in categories N and AEP. The outlier in category AEP is quite extreme, compared to the rest of the notably similar samples, at least concerning the outlier in AEP.

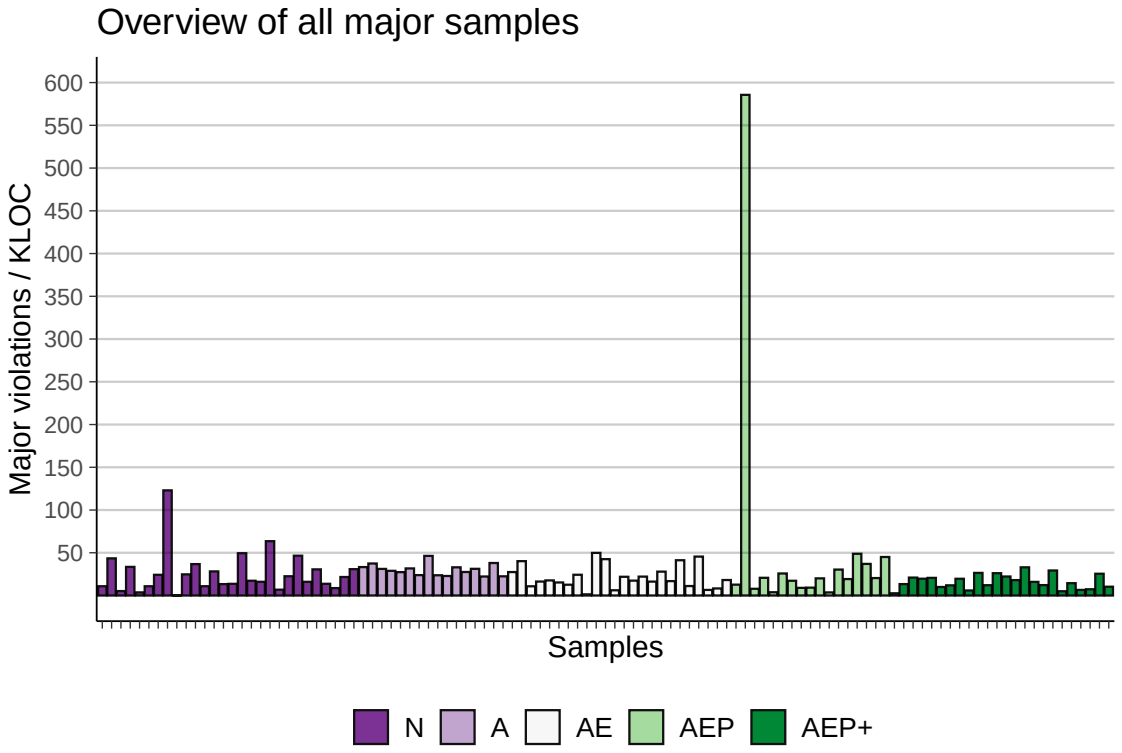

Figure D.4: Overview of major violations per KLOC for each sample in the major categories.

D.5 Minor Violations

An overview of the scores for the minor violation metric is provided in Figure D.5. There are some notable samples in categories N and AEP+ with greater scores, but nothing too extreme.

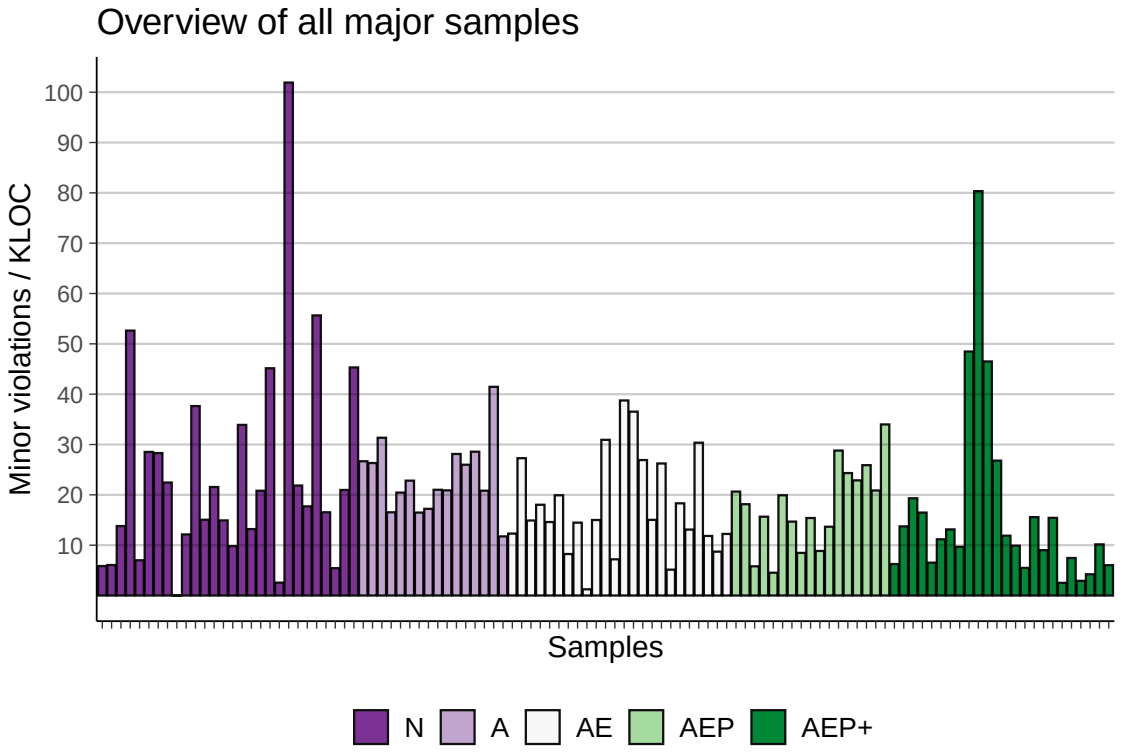

Figure D.5: Overview of minor violations per KLOC for each sample in the major categories.

D.6 Security Hotspots

The scores for the security hotspot metric seem to be a little more "sporadic" than for other metrics, as shown in Figure D.6. There are no clear outliers, but the variation within each category is noteworthy.

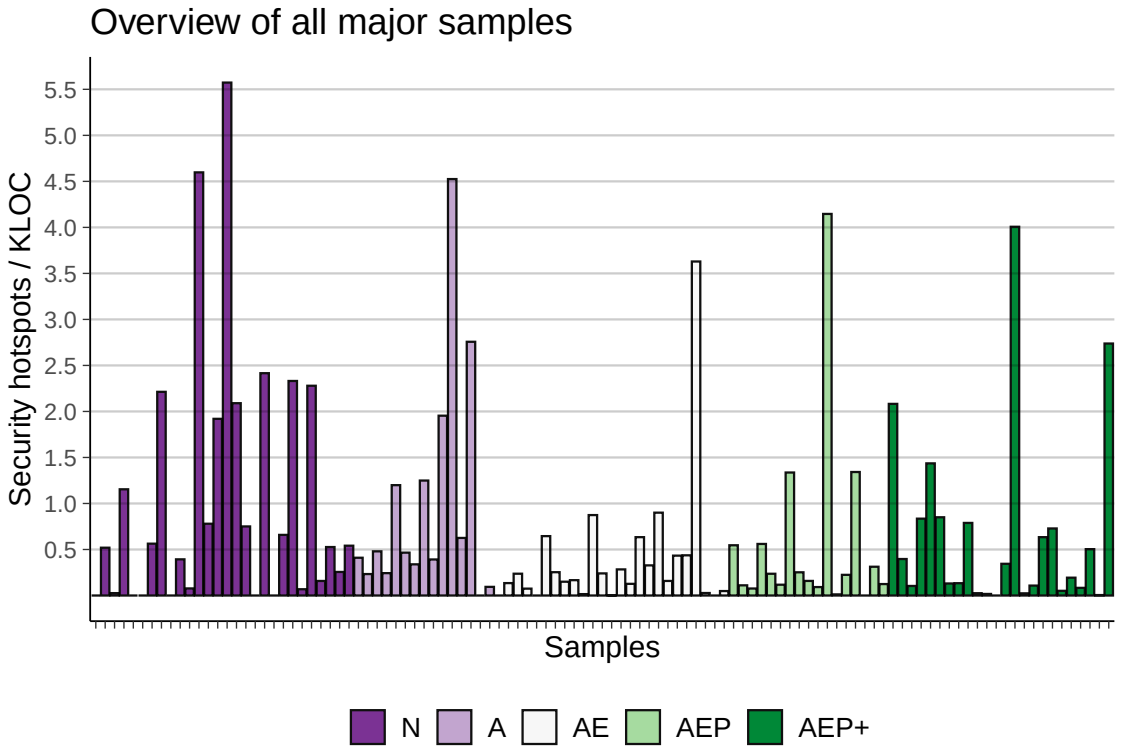

Figure D.6: Overview of security hotspots per KLOC for each sample in the major categories.

D.7 Vulnerabilities

The vulnerability metric is by far the most scarce, with most samples having no detected vulnerabilities at all, see Figure D.7. It is difficult to make any conclusions regarding outliers for this particular metric.

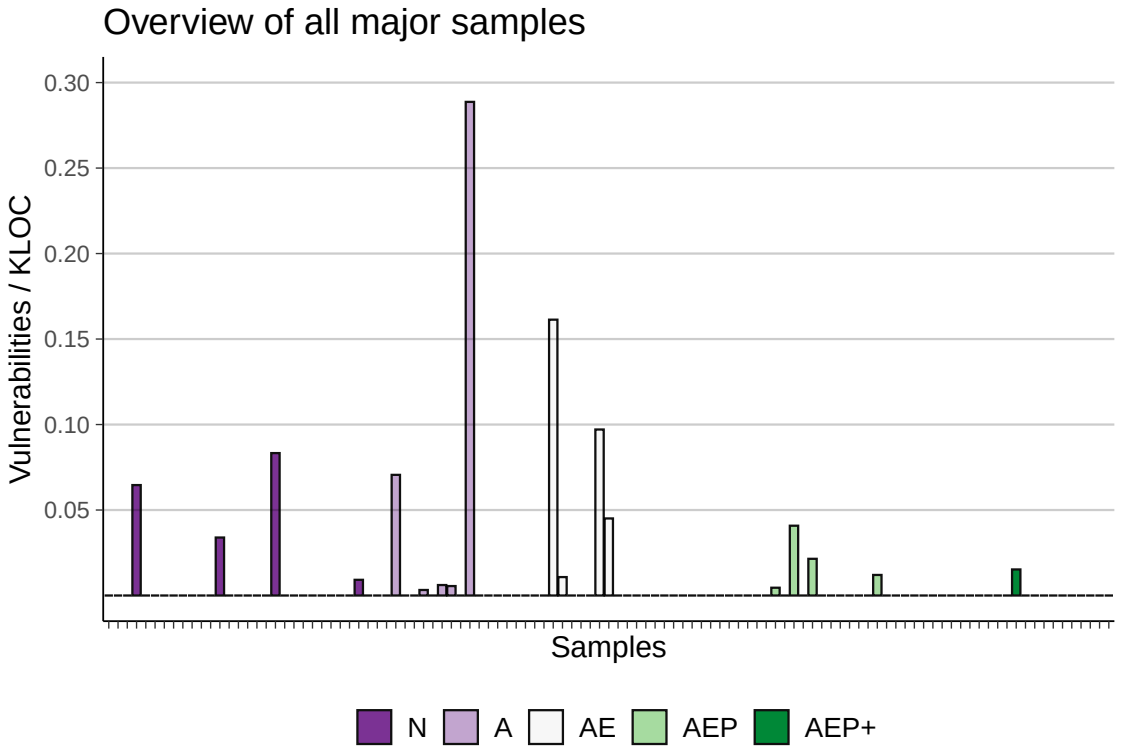

Figure D.7: Overview of vulnerabilities per KLOC for each sample in the major categories.

D.8 Cyclomatic Complexity

Scores for the cyclomatic complexity metric are presented in Figure D.8. The values are remarkably uniform across all categories, except for a clear outlier in category AEP+.

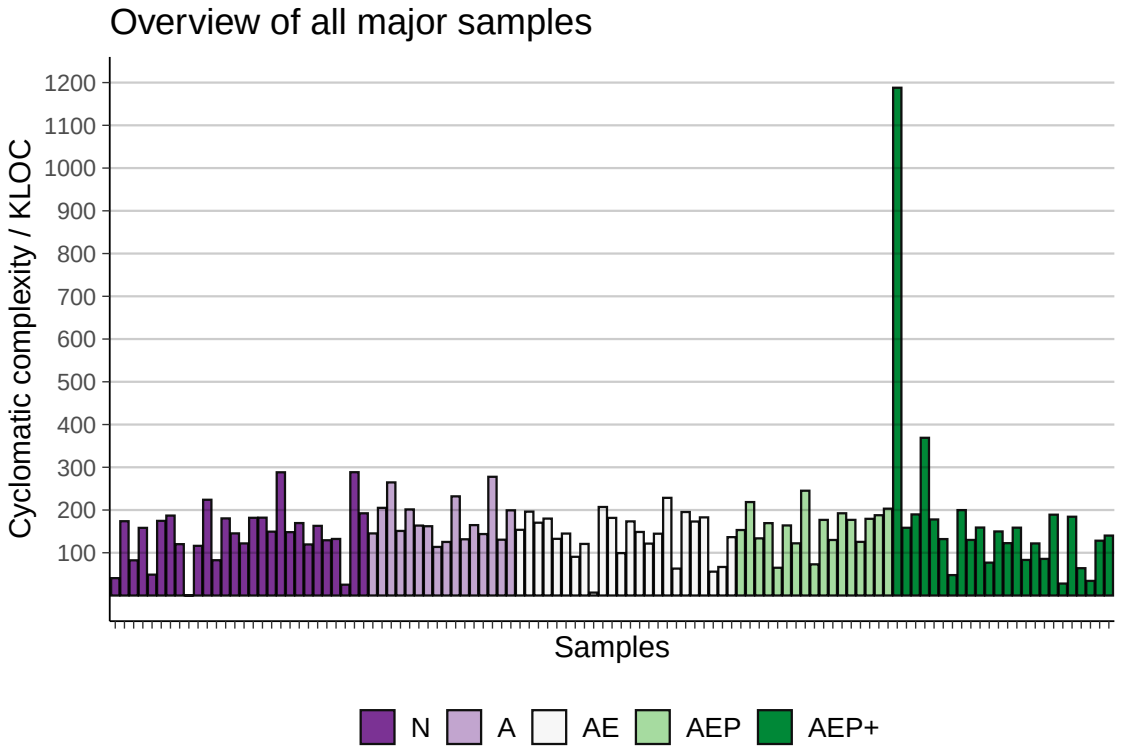

Figure D.8: Overview of cyclomatic complexity per KLOC for each sample in the major categories.

D.9 Cognitive Complexity

Scores for the cognitive complexity metric are presented in Figure D.9. Similar to the case with the cyclomatic complexity metric, samples across the various categories have similar scores. However, the scores are a little bit more varied within each category. Additionally, there are no clear outliers for this metric.

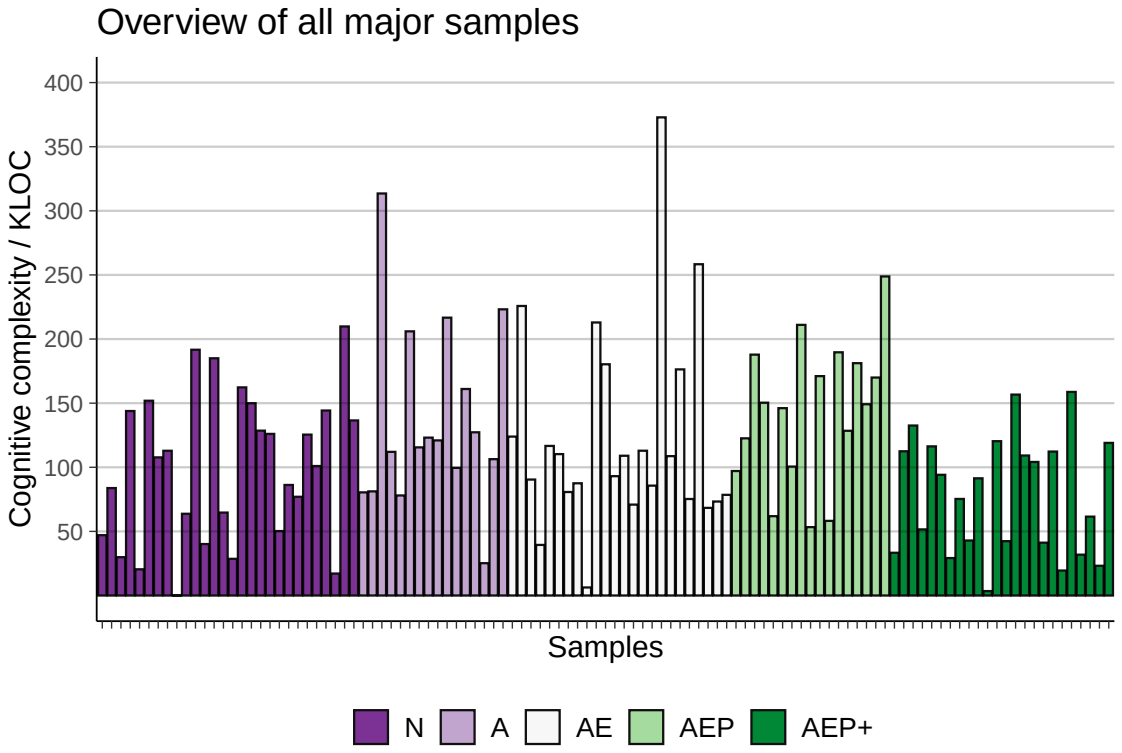

Figure D.9: Overview of cognitive complexity per KLOC for each sample in the major categories.

D.10 Duplicated Lines

Scores for the line duplication metric are presented in Figure D.10. There are no obvious outliers across the different categories, even if the values are quite varied within each category.

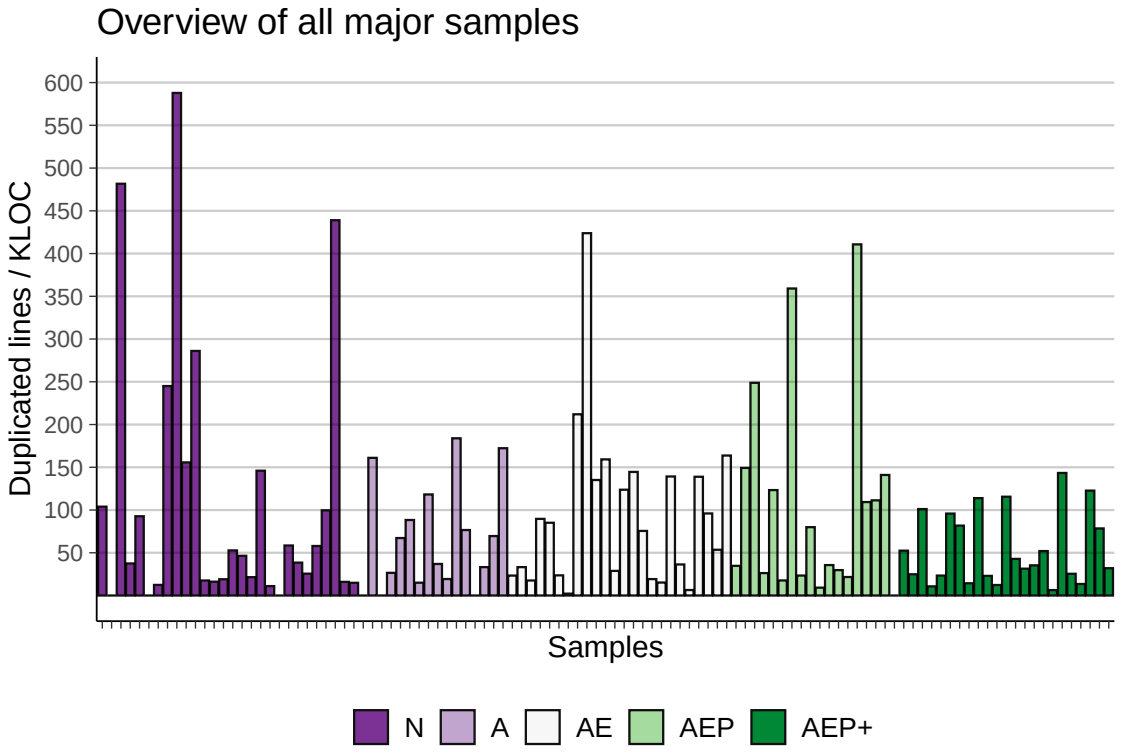

Figure D.10: Overview of duplicated lines per KLOC for each sample in the major categories.

D.11 Technical Debt Ratio

The technical debt ratio for each major sample is presented in Figure D.11. Except for a suspicious outlier in AEP, the overall sample appears reasonable.

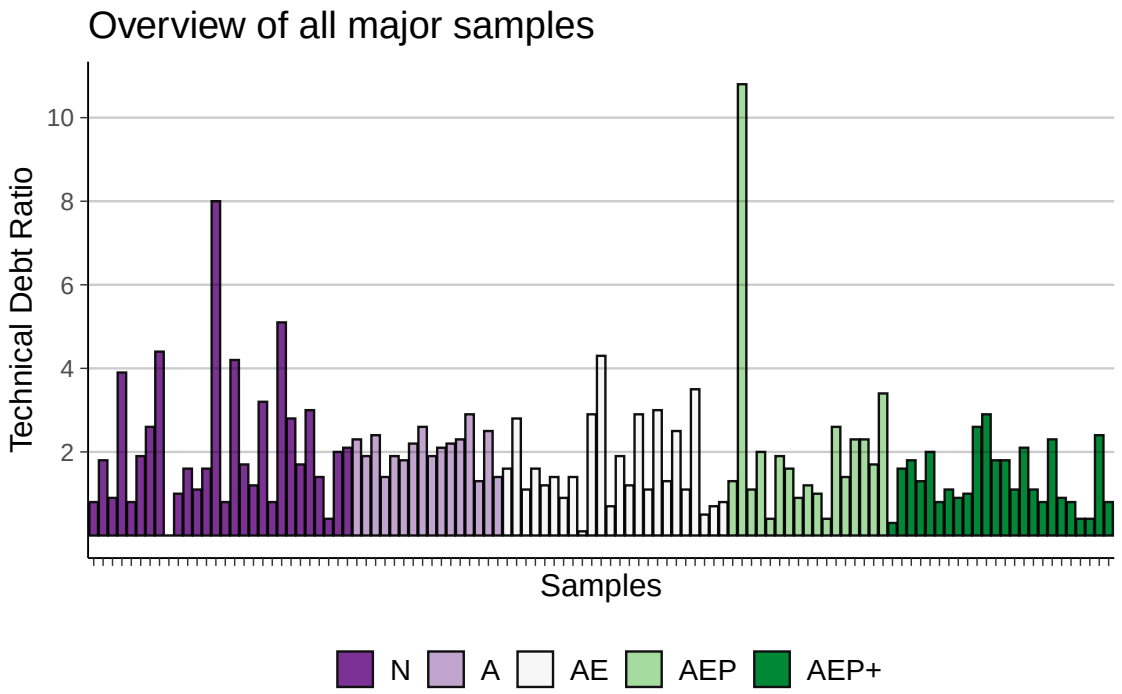

Figure D.11: Overview of the technical debt ratio for each sample in the major categories.