

Evaluation of Rust code verbosity, understandability and complexity

Original

Evaluation of Rust code verbosity, understandability and complexity / Ardito, Luca; Barbato, Luca; Coppola, Riccardo; Valsesia, Michele. - In: PEERJ. COMPUTER SCIENCE.. - ISSN 2376-5992. - (2021), pp. 1-33. [10.7717/peerj-cs.406]

Availability:

This version is available at: 11583/2869605 since: 2021-02-26T09:44:17Z

Publisher:

PeerJ

Published

DOI:10.7717/peerj-cs.406

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

1 Evaluation of Rust code verbosity, 2 understandability and complexity

3 Luca Ardito¹, Luca Barbato², Riccardo Coppola¹, and Michele Valsesia¹

4 ¹Politecnico di Torino

5 ²Luminem

6 Corresponding author:

7 Luca Ardito¹

8 Email address: luca.ardito@polito.it

9 ABSTRACT

10 Rust is an innovative programming language initially implemented by Mozilla, developed to ensure high
11 performance, reliability, and productivity.

12 The final purpose of this study consists of applying a set of common static software metrics to pro-
13 grams written in Rust to assess the verbosity, understandability, organization, complexity, and maintain-
14 ability of the language.

15 To that extent, nine different implementations of algorithms available in different languages were se-
16 lected. We computed a set of metrics for Rust, comparing them with the ones obtained from C and a
17 set of object-oriented languages: C++, Python, JavaScript, TypeScript. To parse the software artifacts
18 and compute the metrics, it was leveraged a tool called *rust-code-analysis* that was extended with a
19 software module, written in Python, with the aim of uniforming and comparing the results.

20 The Rust code had an average verbosity in terms of the raw size of the code. It exposed the most
21 structured source organization in terms of the number of methods. Rust code had a better Cyclomatic
22 Complexity, Halstead Metrics, and Maintainability Indexes than C and C++ but performed worse than
23 the other considered object-oriented languages. Lastly, the Rust code exhibited the lowest COGNI-
24 TIVE complexity of all languages.

25 The collected measures prove that the Rust language has average complexity and maintainability com-
26 pared to a set of popular languages. It is more easily maintainable and less complex than the C and
27 C++ languages, which can be considered syntactically similar. These results, paired with the memory
28 safety and safe concurrency characteristics of the language, can encourage wider adoption of the lan-
29 guage of Rust in substitution of the C language in both the open-source and industrial environments.

30 1 INTRODUCTION

31 Software maintainability is defined as the ease of maintaining software during the delivery of its re-
32 leases. Maintainability is defined by the ISO 9126 standard as *"The ability to identify and fix a fault
33 within a software component"* [17], and by the ISO/IEC 25010:2011 standard as *"degree of effective-
34 ness and efficiency with which a product or system can be modified by the intended maintainers"* [18].
35 Maintainability is an integrated software measure that encompasses some code characteristics, such as
36 readability, documentation quality, simplicity, and understandability of source code [1].

37 Maintainability is a crucial factor in the economic success of software products. It is commonly ac-
38 cepted in the literature that the most considerable cost associated with any software product over its
39 lifetime is the maintenance cost [50]. The maintenance cost is influenced by many different factors,
40 e.g., the necessity for code fixing, code enhancements, the addition of new features, poor code quality,
41 and subsequent need for refactoring operations [35].

42 Hence, many methodologies have consolidated in software engineering research and practice to en-
43 hance this property. Many metrics have been defined to provide a quantifiable and comparable measure-
44 ment for it [37]. Many metrics measure lower-level properties of code (e.g., related to the number of
45 lines of code and code organization) as proxies for maintainability. Several comprehensive categoriza-
46 tions and classifications of the maintainability metrics presented in the literature during the last decades

47 have been provided, e.g., the one by Frantz et al. provides a categorization of 25 different software
48 metrics under the categories of *Size*, *Coupling*, *Complexity*, and *Inheritance* [13].
49 The academic and industrial practice has also provided multiple examples of tools that can automati-
50 cally compute software metrics on source code artifacts developed in many different languages [34].
51 Several frameworks have also been described in the literature that leverage combinations of software
52 code metrics to predict or infer the maintainability of a project [22, 3, 33]. The most recent work in the
53 field of metric computation is aiming at applying machine learning-based approaches to the prediction
54 of maintainability by leveraging the measurements provided by static analysis tools [44].
55 However, the benefit of the massive availability of metrics and tooling for their computation is con-
56 trasted by the constant emergence of novel programming languages in the software development com-
57 munity. In most cases, the metrics have to be readapted to take into account newly defined syntaxes,
58 and existing metric-computing tools cannot work on new languages due to the unavailability of parsers
59 and metric extraction modules. For recently developed languages, the unavailability of appropriate
60 tooling represents an obstacle for empirical evaluations on the maintainability of the code developed
61 using them.
62 This work provides a first evaluation of verbosity, code organization, understandability, and complexity
63 of Rust, a newly emerged programming language similar in characteristics to C++, developed with the
64 premises of providing better maintainability, memory safety, and performance [29]. To this purpose, we
65 (i) adopted and extended a tool to compute maintainability metrics that support this language; (ii) de-
66 veloped a set of scripts to arrange the computed metrics into a comparable JSON format; (iii) executed
67 a small-scale experiment by computing static metrics for a set of programming languages, including
68 Rust, analyzing and comparing the final results. To the best of our knowledge, no existing study in the
69 literature has provided computations of such metrics for the Rust language and the relative comparisons
70 with other languages.
71 The remainder of the manuscript is structured as follows: Section 2 provides background information
72 about the Rust language and presents a brief review of state-of-the-art tools available in the literature
73 for the computation of metrics related to maintainability; Section 3 describes the methodology used to
74 conduct our experiment, along with a description of the developed tools and scripts, the experimental
75 subjects used for the evaluation, and the threats to the validity of the study; Section 4 presents and
76 discusses the collected metrics; Section 5 concludes the paper by listing its main findings and providing
77 possible future directions of this study.

78 **2 BACKGROUND AND RELATED WORK**

79 This section provides background information about the Rust language characteristics, studies in the
80 literature that analyzes its advantages, and the list of available tools present in the literature to measure
81 metrics used as a proxy to quantify software projects' maintainability.

82 **2.1 The Rust programming language**

83 Rust is an innovative programming language initially developed by Mozilla and is currently maintained
84 and improved by the Rust Foundation¹.

85 The main goals of the Rust programming language are: memory-efficiency, with the abolition of
86 garbage collection, with the final aim of empowering performance-critical services running on em-
87 bedded devices, and easy integration with other languages; reliability, with a rich type system and own-
88 ership model to guarantee memory-safety and thread-safety; productivity, with an integrated package
89 manager and build tools.

90 Rust is compatible with multiple architectures and is quite pervasive in the industrial world. Many
91 companies are currently using Rust in production today for fast, low-resource, cross-platform solutions:
92 for example, software like Firefox, Dropbox, and Cloudflare use Rust [41].

93 The Rust language has been analyzed and adopted in many recent studies from academic literature.
94 Uzlu et al. pointed out the appropriateness of using Rust in the Internet of Things domain, mentioning
95 its memory safety and compile-time abstraction as crucial peculiarities for the usage in such domain
96 [48]. Balasubramanian et al. show that Rust enables system programmers to implement robust security
97 and reliability mechanisms more efficiently than other conventional languages [7]. Astrauskas et al.

¹<https://www.rust-lang.org/>

Table 1. Languages supported by the metrics tools

Language	CBR Insight	CCFinderX	CKJM	CodeAnalyzers	Halstead Metrics Tool	Metrics Reloaded	Squale
C	x	x		x		x	x
C++	x	x		x	x		x
C#	x	x		x			
Cobol	x	x		x			x
Java	x	x	x	x	x	x	
Rust							
Others	x			x			

Table 2. Case study definition template [40]

Objective	Evaluation of code verbosity, understandability and complexity
The case	Development with the Rust programming language
Theory	Static measures for software artifacts
Research questions	What is the verbosity, organization, complexity and maintainability of Rust?
Methods	Comparison of Rust static measurements with other programming languages
Selection strategy	Open-source multi-language repositories

98 leveraged Rust’s type system to create a tool to specify and validate system software written in Rust [6].
 99 Koster mentioned the speed and high-level syntax as the principal reasons for writing in the Rust lan-
 100 guage the Rust-Bio library, a set of safe bioinformatic algorithms [24]. Levy et al. reported the process
 101 of developing an entire kernel in Rust, with a focus on resource efficiency [26]. These common usages
 102 of Rust in such low-level applications encourage thorough analyses of the quality and complexity of a
 103 code with Rust.

104 2.2 Tools for measuring static code quality metrics

105 Several tools have been presented in academic works or are commonly used by practitioners to measure
 106 quality metrics related to maintainability for software written in different languages.

107 In our previous works, we conducted a systematic literature review that led us to identify fourteen dif-
 108 ferent open-source tools that can be used to compute a large set of different static metrics [5]. In the
 109 review, it is found that the following set of open-source tools can cover most of quality metrics de-
 110 fined in the literature, for the most common programming languages: *CBR Insight*, a tool based on the
 111 closed-source metrics computation Understand framework, that aims at computing reliability and main-
 112 tainability metrics [27]; *CCFinderX*, a tool tailored for finding duplicate code fragments [30]; *CKJM*, a
 113 tool to compute the C&K metrics suite and method-related metrics for Java code [21]; *CodeAnalyzers*,
 114 a tool supporting more than 25 software maintainability metrics, that covers the highest number of
 115 programming languages along with CBR Insight [43]; *Halstead Metrics Tool*, a tool specifically devel-
 116 oped for the computation of the Halstead Suite [16]; *Metrics Reloaded*, able to compute many software
 117 metrics for C and Java code either in a plug-in for IntelliJ IDEA or through command line [42]; *Squale*,
 118 a tool to measure high-level quality factors for software and measuring a set of code-level metrics to
 119 predict economic aspects of software quality [28].

120 Table 1 reports the principal programming languages supported by the described tools. For the sake of
 121 conciseness, only the languages that were supported by at least two of the tools are reported. With this
 122 comparison, it can be found that none of the considered tools is capable of providing metric computa-
 123 tion facilities for the Rust language.

Table 3. List of metrics used in this study

RQ	Acronym	Name	Description
RQ1	SLOC	Source Lines of Code	It returns the total number of lines in a file
	PLOC	Physical Lines of Code	It returns the total number of instructions and comment lines in a file
	LLOC	Logical Lines of Code	It returns the number of logical lines (statements) in a file
	CLOC	Comment Lines of Code	It returns the number of comment lines in a file
	BLANK	Blank Lines of Code	Number of blank statements in a file
RQ2	NOM	Number of Methods	It returns the number of methods in a source file
	NARGS	Number of Arguments	It counts the number of arguments for each method in a file
	NEXITS	Number of Exit Points	It counts the number of exit points of each method in a file
RQ3	CC	McCabe's Cyclomatic Complexity	It calculates the code complexity examining the control flow of a program; the original McCabe's definition of cyclomatic complexity is the the maximum number of linearly independent circuits in a program control graph [14]
	COGNITIVE	Cognitive Complexity	It is a measure of how difficult a unit of code is to intuitively understand, by examining the cognitive weights of basic software control structures [20]
	Halstead	Halstead suite	A suite of quantitative intermediate measures that are translated to estimations of software tangible properties, e.g. volume, difficulty and effort (see Table 4 for details)
RQ4	MI	Maintainability Index	A composite metric that incorporates a number of traditional source code metrics into a single number that indicates relative maintainability (see Table 5 for details about the considered variants) [49]

124 As additional limitations of the identified set of tools, it can be seen that the tools do not provide complete coverage of the most common metrics for all the tools (e.g., the Halstead Metric suite is computed only by the Halstead Metrics tool), and in some cases, (e.g., CodeAnalyzer), the number of metrics is limited by the type of acquired license. Also, some of the tools (e.g., Squale) appear to have been discontinued by the time of the writing of this article.

129 3 STUDY DESIGN

130 This section reports the goal, research questions, metrics, and procedures adopted for the conducted study.

132 To report the plan for the experiment, the template defined by Robson was adopted [40]. The purpose of the research, according to Robson's classification, is *Exploratory*, i.e., to find out what is happening, seeking new insights, and generating ideas and hypotheses for future research. The main concepts of the definition of the study are reported in table 2.

136 In the following subsections, the best practices for case study research provided by Runeson and Host are adopted to organize the presentation of the study [19]. More specifically, the following elements are reported: goals, research questions, and variables; objects; instrumentation; data collection and analysis procedure; evaluation of validity.

140 3.1 Goals, Research Questions and Variables

141 The high-level goal of the study can be expressed as:

142 *Analyze and evaluate the characteristics of the Rust programming language, focusing on verbosity, understandability and complexity measurements, measured in the context of open-source code, and interpreting the results from developers and researchers' standpoint.*

145 Based on the goal, the research questions that guided the definition of the experiment are obtained. Four different aspects that deserve to be analyzed for code written in Rust programming language were identified, and a distinct Research Question was formulated for each of them. In the following, the research questions are listed, along with a brief description of the metrics adopted to answer them. Table 3 reports a summary of all the metrics.

150 The comparisons between different programming languages were made through the use of static metrics. A static metric (opposed to dynamic or runtime metrics) is obtained by parsing and extracting information from a source file without depending on any information deduced at runtime.

- 153 • **RQ1:** What is the verbosity of Rust code with respect to code written in other programming languages?

155 To answer RQ1, the size of code artifacts written in Rust was measured in terms of the number of code lines in a source file. Four different metrics have been defined to differentiate between the nature of the inspected lines of code:

- 158 • *SLOC*, i.e., Source lines of code;
- 159 • *CLOC*, Comment Lines of Code;
- 160 • *PLOC*, Physical Lines of Code, including both the previous ones;
- 161 • *LLOC*, Logical Lines of Code, returning the count of the statements in a file;
- 162 • *BLANK*, Blank Lines of Code, returning the number of blank lines in a code.

163 The rationale behind using multiple measurements for the lines of code can be motivated by the need for measuring different facets of the size of code artifacts and of the relevance and content of the lines of code. The measurement of physical lines of code (*PLOC*) does not take into consideration blank lines or comments; the count, however, depends on the physical format of the statements and programming style since multiple *PLOC* can concur to form a single logical statement of the source code. *PLOC* are sensitive to logically irrelevant formatting and style conventions, while *LLOC* are less sensitive to these aspects [36]. In addition to that, the *CLOC* and *BLANK* measurements allow a finer analysis of the amount of documentation (in terms of used APIs and explanation of complex parts of algorithms) and formatting of a source file.

Table 4. The Halstead Metrics Suite

Measure	Symbol	Formula
Base measures	$\eta 1$	Number of distinct operators
	$\eta 2$	Number of distinct operands
	N1	Total number of occurrences of operators
	N2	Total number of occurrences of operands
Program length	N	$N = N1 + N2$
Program vocabulary	η	$\eta = \eta 1 + \eta 2$
Volume	V	$V = N * \log_2(\eta)$
Difficulty	D	$D = \eta 1 / 2 * N2 / \eta 2$
Program Level	L	$L = 1 / D$
Effort	E	$E = D * V$
Estimated Program Length	H	$H = \eta 1 * \log_2(\eta 1) + \eta 2 * \log_2(\eta 2)$
Time required to program (in seconds)	T	$T = E / 18$
Number of delivered bugs	B	$B = E^{2/3} / 3000$
Purity Ratio	PR	$PR = H / N$

172 • **RQ2:** How is Rust code organized with respect to code written in other programming languages?

173 To answer RQ2, the source code structure was analyzed in terms of the properties and functions of
 174 source files. To that end, three metrics were adopted: *NOM*, Number of Methods; *NARGS*, Number
 175 of Arguments; *NEXITS*, Number of exits. *NARGS* and *NEXITS* are two software metrics defined by
 176 Mozilla and have no equivalent in the literature about source code organization and quality metrics.
 177 The two metrics are intuitively linked with the easiness in reading and interpreting source code: a
 178 function with a high number of arguments can be more complex to analyze because of a higher number
 179 of possible paths; a function with many exits may include higher complexity in reading the code for
 180 performing maintenance efforts.

181 • **RQ3:** What is the complexity of Rust code with respect to code written in other programming
 182 languages?

183 To answer RQ3, three metrics were adopted: *CC*, McCabe's Cyclomatic Complexity; *COGNITIVE*,
 184 Cognitive Complexity; and the *Halstead suite*. The Halstead Suite, a set of quantitative complexity
 185 measures originally defined by Maurice Halstead, is one of the most popular static code metrics avail-
 186 able in the literature [16]. Table 4 reports the details about the computation of all operands and oper-
 187 ators. The metrics in this category are more high-level than the previous ones and are based on the
 188 computation of previously defined metrics as operands.

189 • **RQ4:** What are the composite maintainability indexes for Rust code with respect to code written
 190 in other programming languages?

191 To answer RQ4, the Maintainability Index was adopted, i.e., a composite metric originally defined by
 192 Oman et al. to provide a single index of maintainability for software [38]. Three different versions
 193 of the Maintainability Index are considered. First, the original version by Oman et al.. Secondly, the
 194 version defined by the Software Engineering Institute (SEI), originally promoted in the C4 Software
 195 Technology Reference Guide [9]; the SEI adds to the original formula a specific treatment for the
 196 comments in the source code (i.e., the CLOC metric), and it is deemed by research as more appropriate
 197 given that the comments in the source code can be considered correct and appropriate [49]. Finally, the
 198 version of the MI metric implemented in the Visual Studio IDE [31]; this formula resettles the MI value
 199 in the 0-100 range, without taking into account the distinction between CLOC and SLOC operated by
 200 the SEI formula [32].

201 The respective formulas are reported in Table 5. The interpretation of the measured MI varies accord-
 202 ing to the adopted formula to compute it: the ranges for each of them are reported in Table 6. For the

Table 5. Considered variants of the MI metric

Acronym	Meaning	Formula
MI_O	Original Maintainability Index	$171.0 - 5.2 * \ln(V) - 0.23 * CC - 16.2 * \ln(SLOC)$
MI_{SEI}	MI by Software Engineering Institute	$171.0 - 5.2 * \log_2(V) - 0.23 * CC - 16.2 * \log_2(SLOC) + 50.0 * \sin(\sqrt{2.4 * (CLOC/SLOC)})$
MI_{VS}	MI implemented in Visual Studio	$\max(0, (171 - 5.2 * \ln(V) - 0.23 * CC - 16.2 * \ln(SLOC)) * 100 / 171)$

Table 6. Maintainability ranges of source code according to different formulas for the MI metric

Variant	Low maintainability	Medium maintainability	High maintainability
Original	$MI < 65$	$65 < MI < 85$	$MI > 85$
SEI	$MI < 65$	$65 < MI < 85$	$MI > 85$
VS	$MI < 10$	$10 < MI < 20$	$MI > 20$

203 traditional and the SEI formulas of the MI, a value over 85 indicates easily maintainable code; a value
 204 between 65 and 85 indicates average maintainability for the analyzed code; a value under 65 indicates
 205 hardly maintainable code. With the original and SEI formulas, the MI value can also be negative. With
 206 the Visual Studio formula, the thresholds for medium and high maintainability are moved respectively
 207 to 10 and 20.

208 The Maintainability Index is the highest-level metric considered in this study, as it includes an interme-
 209 diate computation of one of the Halstead suite metrics.

210 3.2 Objects

211 For the study, it was necessary to gather a set of simple code artifacts to analyze the Rust source code
 212 properties and compare them with other programming languages.

213 To that end, a set of nine simple algorithms was collected. In the set, each algorithm was implemented
 214 in 5 different languages: C, C++, JavaScript, Python, Rust, and TypeScript. All implementations of
 215 the code artifacts have been taken from the Energy-Languages repository². The rationale behind the
 216 repository selection is its continuous and active maintenance and the fact that these code artifacts are
 217 adopted by various other projects for tests and benchmarking purposes, especially for evaluations of the
 218 execution speed of code written in a given programming language after compilation.

219 The number of different programming languages for the comparison was restricted to 5 because those
 220 languages (additional details are provided in the next section) were the common ones for the Energy-
 221 Languages repository and the set of languages that are correctly parsed by the tooling employed in the
 222 experiment conduction.

223 Table 7 lists the code artifacts used (sorted out alphabetically) and provides a brief description of each
 224 of them.

225 3.3 Instruments

226 This section provides details about the framework that was developed to compare the selected metrics
 227 and the existing tools that were employed for code parsing and metric computation.

228 A graphic overview of the framework is provided in Figure 1. The diagram only represents the logical
 229 flow of the data in the framework since the actual flow of operations is reversed, being the *compare.py*
 230 script the entry point of the whole computation.

231 The rust-code-analysis tool is used to compute static metrics and save them in the JSON format. The
 232 *analyzer.py* script receives as input the results in JSON format provided by the rust-code-analysis tool
 233 and formats them in a common notation that is more focused on academic facets of the computed met-
 234 rics rather than the production ones used by the rust-code-analysis default formatting. The *compare.py*

²<https://github.com/greensoftwarelab/Energy-Languages>

Table 7. Selected source code artifacts for the study

Name	Description
binarytrees	Allocate and deallocate binary trees
fannkuchredux	Indexed-access to tiny integer-sequence
fasta	Generate and write random DNA sequences
knucleotide	Hashtable update and k-nucleotide strings
mandelbrot	Generate Mandelbrot set portable bitmap file
nbody	Double-precision N-body simulation
regexredux	Match DNA 8-mers and substitute magic patterns
revcomp	Read DNA sequences - write their reverse-complement
spectralnorm	Eigenvalue using the power method

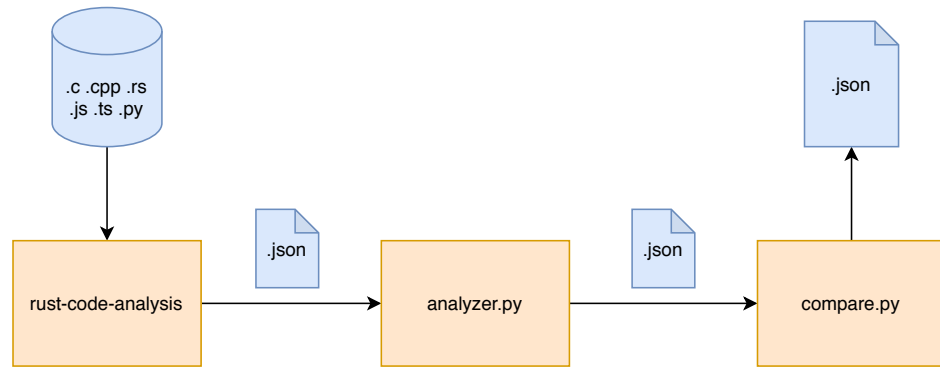


Figure 1. Representation of the data flow of the framework

235 has been developed to call the *analyzer.py* script and to use its results to perform pair-by-pair compar-
236 isons between the JSON files obtained for source files written in different programming languages.
237 These comparison files allow us to immediately assess the differences in the metrics computed by the
238 different programming languages on the same software artifacts. The stack of commands that are called
239 in the described evaluation framework is shown in Figure 2.
240 The evaluation framework has been made available as an open-source repository on GitHub³.

241 **3.3.1 The Rust Code Analysis tool**

242 All considered metrics have been computed by adopting and extending a tool developed in the Rust
243 language, and able to compute metrics for many different ones, called *rust-code-analysis*. We have
244 forked version 0.0.18 of the tool to fix a few minor defects in metric computation and to uniform the
245 presentation of the results, and we have made it available on a GitHub repository⁴.
246 We have decided to adopt and personally extend a project written in Rust because of the advantages
247 guaranteed by this language, such as memory and thread safety, memory efficiency, good performance,
248 and easy integration with other programming languages.
249 *rust-code-analysis* builds, through the use of an open-source library called *tree-sitter*⁵, builds an Ab-
250 stract Syntax Tree (AST) to represent the syntactic structure of a source file. An AST differs from a
251 Concrete Syntax Tree because it does not include information about the source code less important
252 details, like punctuation and parentheses. On top of the generated AST, *rust-code-analysis* performs
253 a division of the source code in *spaces*, i.e., any structure that can incorporate a function. It contains
254 a series of fields such as the name of the structure, the relative line start, line end, kind, and a *metric*
255 object, which is composed of the values of the available metrics computed by *rust-code-analysis* on
256 the functions contained in that space. All metrics computed at the function level are then merged at
257 the parent space level, and this procedure continues until the space representing the entire source file is

³<https://github.com/SoftengPoliTo/SoftwareMetrics>

⁴<https://github.com/SoftengPoliTo/rust-code-analysis>

⁵<https://tree-sitter.github.io/>

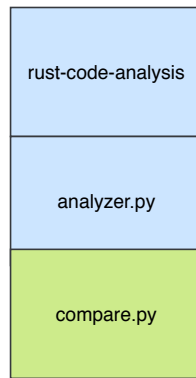


Figure 2. Representation of the process stack of the framework

258 reached.

259 The tool is provided with parser modules that are able to construct the AST (and then to compute the
 260 metrics) for a set of languages: C, C++, C#, Go, JavaScript, Python, Rust, Typescript. The program-
 261 ming languages currently implemented in rust-code-analysis have been chosen because they are the
 262 ones that compose the Mozilla-central repository, which contains the code of the Firefox browser. The
 263 metrics can be computed for each language of this repository with the exception of Java, which does
 264 not have an implementation yet, and HTML and CSS, which are excluded because they are formatting
 265 languages.

266 *rust-code-analysis* can receive either single files or entire directories, detect whether they contain
 267 any code written in one of its supported languages, and output the resultant static metrics in various
 268 formats: textual, JSON, YAML, toml, cbor. [4].

269 Concerning the original implementation of the rust-code-analysis tool, the project was forked and
 270 modified by adding metrics computations (e.g., the COGNITIVE metric). Also, the possible output
 271 format provided by the tool was changed.

272 Listing 1, reported in the annex of the manuscript, reports an excerpt of the JSON file produced as
 273 output by rust-code-analysis.

274 **3.3.2 Analyzer**

275 A Python script named *analyzer.py* was developed to analyze the metrics computed from rust-code-
 276 analysis. This script can launch different software libraries to compute metrics and adapt their results to
 277 a common format.

278 In this experiment, the *analyzer.py* script was used only with the Rust-code-analysis tool, but in a future
 279 extension of this study – or other empirical assessments – the script can be used to launch different
 280 tools simultaneously on the same source code.

281 The *analyzer.py* script performs the following operations:

- 282 • The arguments are parsed to verify their correctness. For instance, *analyzer.py* receives as argu-
 283 ments the list of tools to be executed, the path of the source code to analyze, and the path to the
 284 directory where to save the results;
- 285 • The selected metric computation tool(s) is (are) launched, to start the computation of the soft-
 286 ware metrics on the source files passed as arguments to the analyzer script;
- 287 • The output of the execution of the tool(s) is converted in JSON and formatted in order to have a
 288 common standard to compare the measured software metrics;
- 289 • The newly formatted JSON files are saved in the directory previously passed as an argument to
 290 *analyzer.py*.

291 The output produced by rust-code-analysis through *analyzer.py* was modified for the following reasons:

- 292 • The names of the metrics computed by the tool are not coherent with the ones selected from the
 293 scientific literature about software static quality metrics;

- 294 • The types of data representing the metrics are floating-point values instead of integers since
295 rust-code-analysis aims at being as versatile as possible;
- 296 • The missing aggregation of each source file metrics contained in a directory within a single
297 JSON-object, which is composed of global metrics and the respective metrics for each file. This
298 additional aggregate data allows obtaining a more general prospect on the quality of a project
299 written in a determined programming language.

300 Listing 2 reports an excerpt of the JSON file produced as output by the Analyzer script. As further
301 documentation of the procedure, the full JSON files generated in the evaluation can be found in the
302 Results folder of the project⁶.

303 **3.3.3 Comparison**

304 A second Python script, *Compare.py*, was finally developed to perform the comparisons over the JSON
305 result files generated by the *Analyzer.py* script. The *Compare.py* script executes the comparisons be-
306 tween different language configurations, given an analyzed source code artifact and a metric.

307 The script receives a *Configuration* as a parameter, a pair of versions of the same code, written in two
308 different programming languages.

309 The script performs the following operations for each received *Configuration*:

- 310 • Computes the metrics for the two files of a configuration by calling the analyzer.py script;
- 311 • Loads the two JSON files from the Results directory and compares them, producing a JSON file
312 of differences;
- 313 • Deletes all local metrics (the ones computed by rust-code-analysis for each subspace) from the
314 JSON file of differences;
- 315 • Saves the JSON file of differences, now containing only global file metrics, in a defined destina-
316 tion directory.

317 The JSON differences file is produced using a JavaScript program called JSON-diff⁷.

318 Listing 3 reports an excerpt of the JSON file produced as output by the Comparison script. As further
319 documentation of the procedure, the full JSON files generated in the evaluation can be found in the
320 Compare folder of the project⁸.

321 **3.4 Data collection and Analysis procedure**

322 To collect the data to analyze, the described instruments were applied on each of the selected software
323 objects for all the languages studied (i.e., for a total of 45 software artifacts).

324 The collected data was formatted in a single .csv file containing all the measurements.

325 To analyze the results, comparative analyses of the average and median of each of the measured metrics
326 were performed to provide a preliminary discussion.

327 A non-parametric Kruskal-Wallis test was later applied to identify statistically significant differences
328 among the different sets of metrics for each language.

329 For significantly different distributions, post-hoc comparisons with Wilcoxon signed rank-sum test,
330 with Benjamini-Hochberg correction [11], were applied to analyze the difference between the metrics
331 measured for Rust and the other five languages in the set.

332 Descriptive and statistical analyses and graph generation were performed in R. The data and scripts
333 have been made available in an online repository⁹.

⁶<https://github.com/SoftengPoliTo/SoftwareMetrics/tree/master/Results>

⁷<https://www.npmjs.com/package/json-diff>

⁸<https://github.com/SoftengPoliTo/SoftwareMetrics/tree/master/Compare>

⁹<https://github.com/SoftengPoliTo/rust-analysis>

334 3.5 Threats to Validity

335 *Threats to Internal Validity.* The study results may be influenced by the specific selection of the tool
336 with which the software metrics were computed, namely the *rust-code-analysis* tool. The values mea-
337 sured for the individual metrics (and, by consequence, the reasoning based upon them) can be heavily
338 influenced by the exact formula used for the metric computation.

339 In the Halstead suite, the formulas depend on two coefficients defined explicitly in the literature for
340 every software language, namely the denominators for the T and B metrics. Since no previous result
341 in the literature has provided Halstead coefficients specific to Rust, the C coefficients were used for
342 the computation of Rust Halstead metrics. More specifically, 18 was used as the denominator of the
343 T metric. This value, called Stoud number (S), is measured in moments, i.e., the time required by the
344 human brain to carry out the most elementary decision. In general, S is comprised between 5 and 20. In
345 the original Halstead metrics suite for the C language, a value of 18 is used. This value was empirically
346 defined after psychological studies of the mental effort required by coding. 3000 was selected as the
347 denominator of the Number of delivered Bugs metric; this value, again, is the original value defined for
348 the Halstead suite and represents the number of mental discriminations required to produce an error in
349 any language. The 3000 value was originally computed for the English language and then mutated
350 for programming languages [39]. The choice of the Halstead parameters may significantly influence
351 the values obtained for the T and B metrics. The definition of the specific parameters for a new pro-
352 gramming language, however, implies the need for a thorough empirical evaluation of such parameters.
353 Future extensions of this work may include studies to infer the optimal Halstead parameters for Rust
354 source code.

355 Finally, two metrics, NARGS and NEXITS, were adopted for the evaluation of readability and organi-
356 zation of code. Albeit extensively used in production (they are used in the Mozilla-central open-source
357 codebase), these metrics still miss empirical validation on large repositories, and hence their capacity of
358 predicting code readability and complexity cannot be ensured.

359 *Threats to External Validity.* The results presented in this research have been measured on a limited
360 number of source artifacts (namely, nine different code artifacts per programming language). Therefore,
361 we acknowledge that the results cannot be generalized to all software written with one of the analyzed
362 programming languages. Another bias can be introduced in the results by the characteristics of the
363 considered code artifacts. All considered source files were small programs collected from a single
364 software repository. The said software repository itself was implemented for a specific purpose, namely
365 the evaluation of the performance of different programming languages at runtime. Therefore, it is
366 still unsure whether our measurements can scale up to bigger software repositories and real-world
367 applications written in the evaluated languages. As well, the results of the present manuscript may
368 inherit possible biases that the authors of the code had in writing the source artifacts employed for
369 our evaluation. Future extensions of the current work should include the computation of the selected
370 metrics on more extensive and more diverse sets of software artifacts to increase the generalizability of
371 the present results.

372 *Threats to Conclusion Validity.* The conclusions detailed in this work are only based on the analysis of
373 quantitative metrics and do not consider other possible characteristics of the analyzed source artifacts
374 (e.g., the developers' coding style who produced the code). Like the generalizability of the results, this
375 bias can be reduced in future extensions of the study using a broader and more heterogeneous set of
376 source artifacts [45].

377 In this work, we make assumptions on verbosity, complexity, understandability, and maintainability of
378 source code based on quantitative static metrics. It is not ensured that our assumptions are reflected by
379 maintenance and code understanding effort in real-world development scenarios. It is worth mentioning
380 that there is no unanimous opinion about the ability of more complex metrics (like MI) to capture the
381 maintainability of software programs more than simpler metrics like lines of code and Cyclomatic
382 Complexity.

383 Researcher bias is a final theoretical threat to the validity of this study since it involved a comparison in
384 terms of different metrics of different programming languages. However, the authors have no reason to
385 favor any particular approach, neither inclined to demonstrate any specific result.

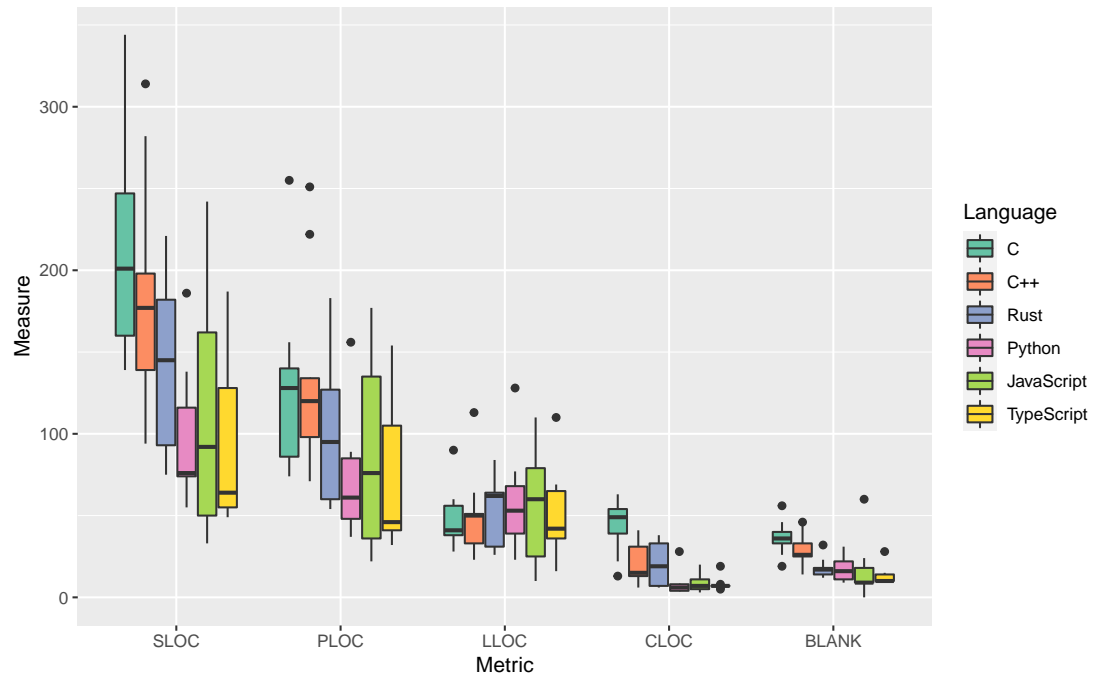


Figure 3. Distribution of the metrics about lines of code for all the considered programming languages

Table 8. Mean (Median) values of the metrics about lines of code for all the considered programming languages

Language	SLOC	PLOC	LLOC	CLOC	BLANK
C	209 (201)	129 (128)	48 (41)	43 (49)	37 (36)
C++	186 (177)	137 (120)	51 (50)	20 (15)	28 (26)
Rust	144 (145)	105 (95)	52 (62)	21 (19)	18 (17)
Python	99 (76)	73 (61)	59 (53)	8 (6)	18 (16)
JavaScript	107 (92)	83 (76)	58 (60)	9 (7)	16 (9)
TypeScript	95 (64)	74 (46)	51 (42)	8 (7)	13 (10)

386 4 RESULTS AND DISCUSSION

387 This section reports the results gathered by applying the methodology described in the previous section,
 388 subdivided according to the research question they answer.

389 4.1 RQ1 - Code verbosity

390 The boxplots in Figure 3 and Table 8 report the measures for the metrics adopted to answer RQ1.
 391 The mean and median values of the Source Lines of Code (SLOC) metric (i.e., total lines of code in the
 392 source files) are largely higher for the C, C++, and Rust language: the highest mean SLOC was for C
 393 (209 average LOCs per source file), followed by C++ (186) and Rust (144). The mean values are way
 394 smaller for Python, TypeScript, and JavaScript (respectively, 98, 107, and 95).
 395 A similar trend is assumed by the Physical Lines of Code (PLOC) metric, i.e., the total number of in-
 396 structions and comment lines in the source files. In the examined set, 74 average PLOCs per file were
 397 measured for the Rust language. The highest and smallest values were again measured respectively for
 398 C and TypeScript, with 129 and 74 average PLOCs per file. The values measured for the CLOC and
 399 BLANK metrics showed that a higher number of empty lines of code and comments were measured
 400 for C than for all other languages. In the CLOC metric, the Rust language exhibited the second-highest
 401 mean of all languages, suggesting a higher predisposition of Rust developers at providing documenta-
 402 tion in the developed source code.

Table 9. Null hypotheses and p-values for RQ1 metrics obtained by applying Kruskal-Wallis chi-squared test¹⁰

Name	Description	p-value	Decision	Significance
$H0_{SLOC}$	No significant difference in SLOC for the sw artifacts	0.001706	Reject	**
$H0_{PLOC}$	No significant difference in PLOC for the artifacts	0.03617	Reject	*
$H0_{LLOC}$	No significant difference in LLOC for the artifacts	0.9495	Not Reject	-
$H0_{CLOC}$	No significant difference in CLOC for the artifacts	7.07e-05	Reject	***
$H0_{BLANK}$	No significant difference in BLANK for the artifacts	0.0001281	Reject	***

Table 10. p-values for post-hoc Wilcoxon Signed Rank test for RQ1 metrics between Rust and the other languages

Metric	C	C++	JavaScript	Python	TypeScript
SLOC	0.0519	0.3309	0.2505	0.0420	0.0519
PLOC	0.3770	0.3081	0.3607	0.2790	0.2790
CLOC	0.0399	0.8242	0.0620	0.0620	0.097
BLANK	0.0053	0.0618	0.1944	0.7234	0.0467

403 A slightly different trend is assumed by the Logical Lines of Code (LLOC) metric (i.e., the number of
 404 instructions or statements in a file). In this case, the mean number of statements for Rust code is higher
 405 than the ones measured for C, C++ and TypeScript, while the SLOC and PLOC metrics are lower. The
 406 Rust scripts also had the highest median LLOC. This result may be influenced with the different num-
 407 ber of types of statements that are offered by the language. For instance, the Rust language provides
 408 19 types of statements while C offers just 14 types (e.g., the Rust statements *If let* and *While let* are not
 409 present in C). The higher amount of logical statements may indeed hint at a higher decomposition of
 410 the instructions of the source code into more statements, i.e., more specialized statements covering less
 411 operations.

412 Albeit many higher-level measures and metrics have been derived in the latest years by related litera-
 413 ture to evaluate the understandability and maintainability of software, the analysis of code verbosity
 414 can be considered a primary proxy for these evaluations. In fact, several studies have linked the intrin-
 415 sic verbosity of a language to lower readability of the software code, which translates to higher effort
 416 when the code has to be maintained. For instance, Flauzino et al. state that verbosity can cause higher
 417 mental energy in coders working on implementing an algorithm and can be correlated to many smells
 418 in software code [12]. Toomim et al. highlight that redundancy and verbosity can obscure meaningful
 419 information in the code, thereby making it difficult to understand [47].

420 The metrics for RQ1 were mostly evenly distributed among different source code artifacts. Two out-
 421 liers were identified for the PLOC metric in C and C++ (namely, *fasta.c* and *fasta.cpp*), mostly due to
 422 the fact that they have the highest SLOC value, so the results are coherent. More marked outliers were
 423 found for the BLANK metric, but such measure is strongly influenced by the developer’s coding style
 424 and the used code formatters; thereby, no valuable insight can be found by analyzing the individual
 425 code artifacts.

426 Table 9 reports the results of applying the Kruskal-Wallis non-parametric test on the set of measures for
 427 RQ1. The difference for SLOC, PLOC, CLOC, and BLANK were statistically significant (with strong
 428 significance for the last two metrics). Post-hoc statistical tests focused on the comparison between Rust,
 429 and the other languages (table 10) led to the evidence that Rust had a significantly lower CLOC than C
 430 and a significantly lower BLANK than C and TypeScript.

Answer to RQ1: The examined source files written in Rust exhibited an average verbosity (144 mean SLOCs per file and 74 mean PLOCs per file). Such values are lower than C and C++ and higher than the other considered object-oriented languages. Rust exhibited the third-highest average (and highest median) LLOC among all considered languages. Significantly lower values were measured for CLOC against C and BLANK against C and TypeScript.

¹⁰Signific. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 '.' 1

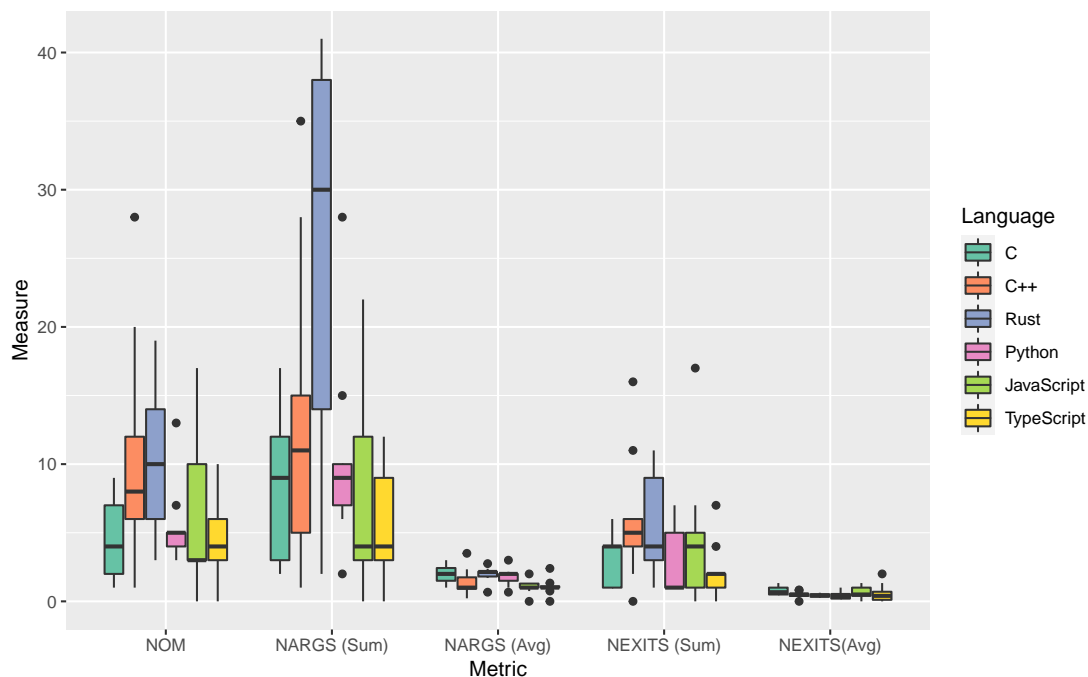


Figure 4. Distribution of the metrics about organization of code for all the considered programming languages

Table 11. Mean (Median) values of the metrics about code organization for all the considered programming languages

Language	NOM	NARGS (Sum)	NARGS (Avg)	NEXITS (Sum)	NEXITS (Avg)
C	4.4 (4)	8.6 (9)	2.0 (2)	3.1 (4)	0.75 (0.67)
C++	10.6 (8)	13.4 (11)	1.4 (1)	6.0 (5)	0.48 (0.5)
Rust	10.3 (10)	25.1 (30)	2.0 (2)	5.7 (4)	0.44 (0.43)
Python	5.7 (5)	10.6 (9)	1.8 (2)	2.8 (1)	0.45 (0.33)
JavaScript	5.9 (3)	7.4 (4)	1.1 (1)	4.6 (4)	0.63 (0.5)
TypeScript	4.7 (4)	5.7 (4)	1.1 (1)	2.1 (2)	0.58 (0.4)

433 4.2 RQ2 - Code organization

434 The boxplots in Figure 4 and Table 11 report the measures for the metrics adopted to answer RQ2.
 435 For each source file, two different measures were collected for the Number of Arguments (NARGS)
 436 metric: the sum at file level of all the methods arguments and the average at file level of the number of
 437 arguments per method (i.e., $NARGS/NOM$).

438 The Rust language had the highest median value for the Number of Methods (NOM) metric, with a
 439 median of 10 methods per source file. The average NOM value was only lower than the one measured
 440 for C++ sources. However, this value was strongly influenced by the presence of one outlier in the set
 441 of analyzed sources (namely, the C++ implementation of *fasta* having a NOM equal to 20). While the
 442 NOM values were similar for C++ and Rust, all other languages exhibited much lower distributions,
 443 with the lowest median value for JavaScript (3). This high number of Rust methods can be seen as
 444 evidence of higher modularity than the other languages considered.

445 Regarding the number of arguments, it can be noticed that the Rust language exhibited the highest av-
 446 erage and median cumulative number of arguments (Sum of Arguments) of all languages. The already
 447 discussed high NOM value influences this result. The highest NOM (and, by consequence, of the total
 448 cumulative number of arguments) can be caused by the missing possibility of having default values in
 449 the Rust language. This characteristic may lead to multiple variations of the same method to take into
 450 account changes in the parameter, thereby leading to a higher NARGS.

Table 12. Null hypotheses and p-values for RQ2 metrics obtained by applying Kruskal-Wallis chi-squared test

Name	Description	p-value	Decision	Significance
$H0_{NOM}$	No significant difference in NOM for the artifacts	0.04372	Reject	*
$H0_{NARGSUM}$	No significant difference in $NARGSUM$ for the artifacts	0.02357	Reject	*
$H0_{NARGSAVG}$	No significant difference in $NARGSAVG$ for the artifacts	0.008224	Reject	**
$H0_{NEXITSUM}$	No significant difference in $NEXITSUM$ for the artifacts	0.142	Not Reject	-
$H0_{NEXITSAVG}$	No significant difference in $NEXITSAVG$ for the artifacts	0.2485	Not Reject	-

Table 13. p-values for post-hoc Wilcoxon Signed Rank test for RQ2 metrics between Rust and the other languages

Metric	C	C++	JavaScript	Python	TypeScript
NOM	0.0534	0.7560	0.1037	0.0546	0.0533
$NARGSUM$	0.0239	0.0633	0.0199	0.0318	0.0177
$NARGSAVG$	0.5658	0.1862	0.0451	0.4392	0.0662

451 The lowest average measures for NOM and NARGSUM metrics were obtained for the C language.
 452 This result can be justified by the lower modularity of the C language. By examining the C source
 453 files, it could be verified that the code presented fewer functions and more frequent usage of nested
 454 loops, while the Rust sources were using more often data structures and ad-hoc methods. In general,
 455 the results gathered for these metrics suggest a more structured Rust code organization with respect to
 456 the C language.

457 The NOM metric has an influence on the verbosity of the code, and therefore it can be considered as a
 458 proxy of the readability and maintainability of the code.

459 Regarding the Number of Exits (NEXITS) metric, the values were close for most languages, except
 460 Python and TypeScript, which respectively contain more methods without exit points and fewer func-
 461 tions. The obtained NEXITS value for Rust shows many exit points distributed among many functions,
 462 as demonstrated by the NOM value, making the code much more comfortable to follow.

463 An analysis of the outliers of the distributions of the measurements for RQ2 was performed. For C++,
 464 the highest value of NOM was exhibited by the *revcomp.cpp* source artifact. This high value was
 465 caused by the extensive use of classes methods to handle chunks of DNA sequences. *knucleotide.py*
 466 and *spectralnorm.py* had a higher number of functions than the other considered source artifacts.
 467 *fasta.cpp* uses lots of small functions with many arguments, resulting in an outlier value for the $NARGSUM$
 468 metric. *pidigits.py* had 0 values for NOM and NARGSUM, since it used zero functions. Regarding NEX-
 469 ITS, very high values were measured for *fasta.cpp* and *revcomp.cpp*, which had many functions with
 470 return statements. Lower values were measured for *regexredup.cpp*, which has a single main function
 471 without any return, and *pidigits.cpp*, which has a single return. A final outlier was the NEXITS value
 472 for *fasta.js*, which features a very high number of function with return statements.

473 Table 12 reports the results of the application of the Kruskal-Wallis non-parametric test on the set of
 474 measures for RQ2. The difference for NOM, $NARGSUM$ and $NARGSAVG$ was statistically significant,
 475 while no significance was measured for the metrics related to the NEXITS. Post-hoc statistical tests
 476 focused on the comparison between Rust, and the other languages (table 13) highlighted that Rust had
 477 a significantly higher $NARGSUM$ than C, JavaScript, Python, and TypeScript, and a significantly higher
 478 $NARGSAVG$ than JavaScript.

Answer to RQ2: The examined source files written in Rust exhibited the most structured organi-
 479 zation of the considered set of languages (with a mean 10.3 NOM per file, with a mean of 2 argu-
 480 ments for each method). The Rust language had a significantly higher number of arguments than C,
 JavaScript, Python, and TypeScript.

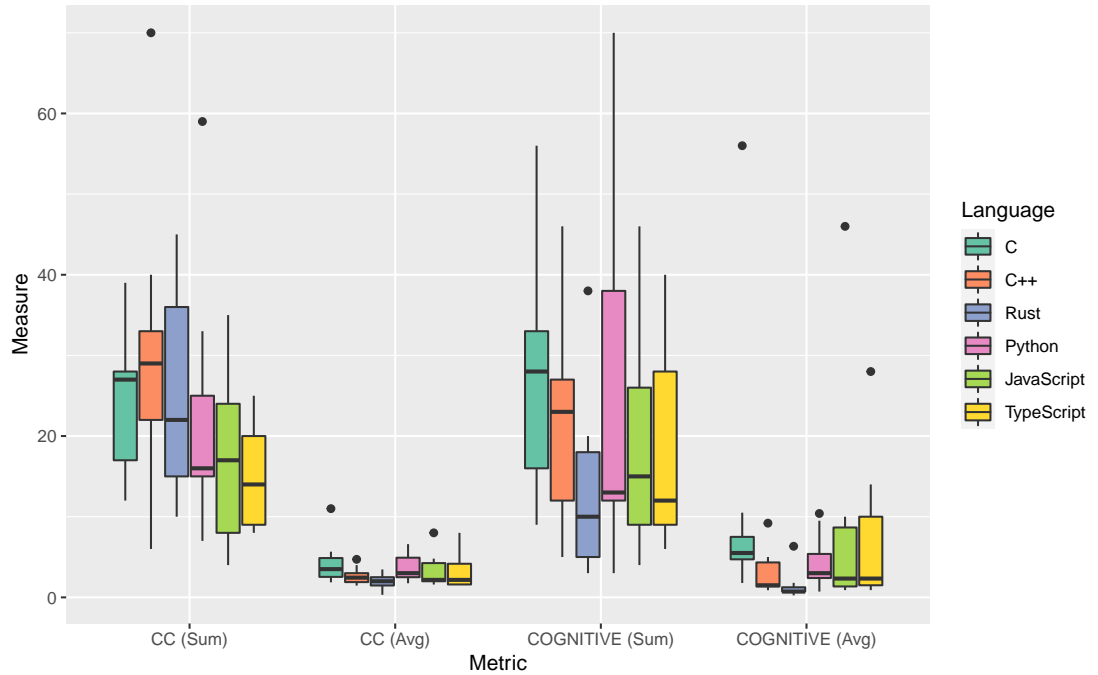


Figure 5. Distribution of complexity metrics for all the considered programming languages

Table 14. Mean (Median) values of the complexity metrics for all the considered programming languages

Language	CC_{Sum}	CC_{Avg}	$COGNITIVE_{Sum}$	$COGNITIVE_{Avg}$
C	27.3 (28)	4.3 (3.5)	24.3 (21.0)	11.2 (5.5)
C++	31.1 (29)	2.7 (2.4)	22.4 (23.0)	3.2 (1.5)
Rust	25.3 (22)	2.0 (2.0)	13.1 (10.0)	1.5 (0.7)
Python	23.0 (16)	3.6 (3.0)	25.4 (13.0)	4.4 (3.0)
JavaScript	17.6 (17)	3.4 (2.2)	19.9 (15.0)	8.5 (2.3)
TypeScript	15.2 (14)	3.4 (2.2)	17.0 (12.0)	7.2 (2.3)

481 4.3 RQ3 - Code complexity

482 The boxplots in Figure 5 and Table 14 report the measures for the metrics adopted to answer RQ3. For
 483 the Computational Complexity, two metrics were computed: the sum of the Cyclomatic Complexity
 484 (CC) of all *spaces* in a source file (CC_{Sum}), and the averaged value of CC over the number of spaces in
 485 a file (CC_{Avg}). A space is defined in *rust-code-analysis* as any structure that incorporates a function. For
 486 what concerns the COGNITIVE complexity, two metrics were computed: the sum of the COGNITIVE
 487 complexity associated to each function and closure present in a source file, ($COGNITIVE_{Sum}$), and the
 488 average value of COGNITIVE complexity, ($COGNITIVE_{Avg}$), always computed over the number of
 489 functions and closures. Table 14 reports the mean and median values over the set of different source
 490 files selected for each language, of the sum and average metrics computed at the file level.

491 As commonly accepted in the literature and practice, a low cyclomatic complexity generally indicates
 492 a method that is easy to understand, test, and maintain. The reported measures showed that the Rust
 493 language had a lower median CC_{Sum} (22) than C and C++ and the second-highest average value (25.3).
 494 The lowest average and median CC_{Sum} was measured for the TypeScript language. By considering
 495 the average of the Cyclomatic Complexity, CC_{Avg} , at the function level, the highest average and mean
 496 values are instead obtained for the Rust language. It is worth mentioning that the average CC values for
 497 all the languages were rather low, hinting at an inherent simplicity of the software functionality under
 498 examination. So an analysis based on different codebases may result in more pronounced differences
 499 between the programming languages.

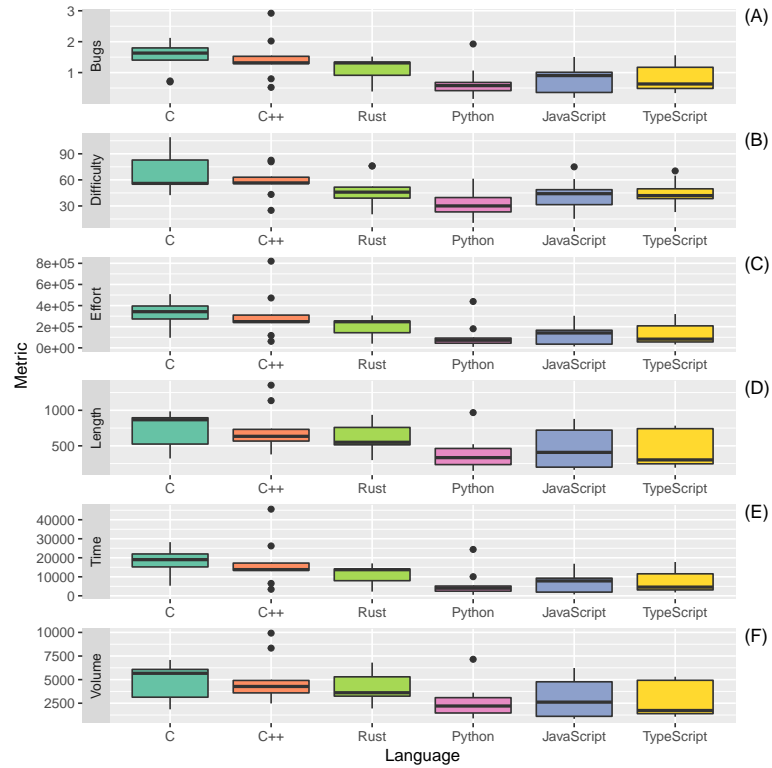


Figure 6. Distribution of Halstead metrics (A: Bugs; B: Difficulty; C: Effort; D: Length; E: Time; F: Volume) for all the considered programming languages

Table 15. Mean (Median) values of Halstead metrics for all the considered programming languages

Language	Bugs	Difficulty	Effort	Length	Programming Time	Volume
C	1.52 (1.6)	66.7 (55.9)	322,313 (342,335)	726.0 (867.0)	17,906 (19,018)	4,819 (5,669)
C++	1.46 (1.3)	57.8 (56.4)	311,415 (248,153)	728.1 (634.0)	17,300 (13,786)	4,994 (4,274)
Rust	1.1 (1.3)	48.6 (45.9)	199,152 (246,959)	602.2 (550.0)	11,064 (13,719)	4,032 (3610)
Python	0.7 (0.6)	33.7 (30.0)	111,103 (72,110)	393.8 (334.0)	6,172 (4,006)	2,680 (2204)
JavaScript	0.8 (0.9)	43.1 (44.1)	139,590 (140,951)	458.6 (408.0)	7,755 (7,830)	2,963 (2615)
TypeScript	0.8 (0.6)	45.2 (41.9)	132,644 (82,369)	435.7 (302.0)	7,369 (4,576)	2,734 (1730)

500 COGNITIVE complexity is a software metric that assesses the complexity of code starting from hu-
501 man judgment and is a measure for source code comprehension by the developers and maintainers [8].
502 Moreover, empirical results have also proved the correlation between COGNITIVE complexity and
503 defects [2]. For both the average COGNITIVE complexity and the sum of COGNITIVE complexity
504 at the file level, Rust provided the lowest mean and median values. Specifically, Rust guaranteed a
505 COGNITIVE complexity of 0.7 per method, which is less than half the second-lowest value for C++
506 (1.5). The highest average COGNITIVE complexity per class was measured for C code (5.5). This
507 very low value of the COGNITIVE complexity per method for Rust is related to the highest number
508 of methods for Rust code (described in the analysis of RQ2 results). By considering the sum of the
509 COGNITIVE complexity metric at the file level, Rust had a mean $COGNITIVE_{Sum}$ of 13.1 over the 9
510 analyzed source files. The highest mean value for this metric was measured for Python (25.4), and the
511 highest median for C++ (23). Such lower values for the Rust language can suggest a more accessible,
512 less costly, and less prone to bug injection maintenance for source code written in Rust. This lowest
513 value for the COGNITIVE metric counters some measurements (e.g., for the LLOC and NOM metrics)
514 by hinting that the higher verbosity of the Rust language has not a visible influence on the readability
515 and comprehensibility of the Rust code.

516 The boxplots in Figure 6 and Table 15 report the distributions, mean, and median of the Halstead met-

Table 16. Null hypotheses and p-values for RQ3 metrics obtained by applying Kruskal-Wallis chi-squared test

Name	Description	p-value	Decision	Significance
$H_{CC\ SUM}$	No significant difference in CC_{SUM} for the artifacts	0.113	Not reject	-
$H_{CC\ AVG}$	No significant difference in CC_{AVG} for the artifacts	0.1309	Not Reject	-
$H_{COGNITIVE\ SUM}$	No significant difference in $COGNITIVE_{SUM}$ for the artifacts	0.4554	Not Reject	-
$H_{COGNITIVE\ AVG}$	No significant difference in $COGNITIVE_{AVG}$ for the artifacts	0.009287	Reject	**
$H_{Halstead\ Vocabulary}$	No significant difference in Halstead Vocabulary for the artifacts	0.07718	Not Reject	.
$H_{Halstead\ Difficulty}$	No significant difference in Halstead Difficulty for the artifacts	0.01531	Reject	*
$H_{Halstead\ Prog.time}$	No significant difference in Halstead Prog. time for the artifacts	0.005966	Reject	**
$H_{Halstead\ Effort}$	No significant difference in Halstead Effort for the artifacts	0.005966	Reject	**
$H_{Halstead\ Volume}$	No significant difference in Halstead Volume for the artifacts	0.03729	Reject	*
$H_{Halstead\ Bugs}$	No significant difference in Halstead Bugs for the artifacts	0.005966	Reject	**

Table 17. p-values for post-hoc Wilcoxon Signed Rank test for RQ3 metrics between Rust and the other languages

Metric	C	C	JavaScript	Python	TypeScript
$COGNITIVE_{AVG}$	0.0062	0.0244	0.0222	0.0240	0.0222
$HALSTEAD_{Difficulty}$	0.2597	0.2621	0.5328	0.2621	0.6587
$HALSTEAD_{ProgrammingTime}$	0.1698	0.3767	0.3081	0.1930	0.3134
$HALSTEAD_{Effort}$	0.1698	0.3767	0.3081	0.1930	0.3134
$HALSTEAD_{Volume}$	0.5960	0.5328	0.2621	0.2330	0.2330
$HALSTEAD_{Bugs}$	0.1698	0.3767	0.3081	0.1930	0.3134

517 rics computed for the six different programming languages.

518 The Halstead Difficulty (D) is an estimation of the difficulty of writing a program that is statically
 519 analyzed. The Difficulty is the inverse of the program level metric. Hence, as the volume of the imple-
 520 mentation of code increases, the difficulty increases as well. The usage of redundancy hence influences
 521 the Difficulty. It is correlated to the number of operators and operands used in the code implementa-
 522 tion. The results suggest that the Rust programming language has an average Difficulty (median of
 523 45.9) on the set of considered languages. The most difficult code to interpret, according to Halstead
 524 metrics, was C (median of 55.9), while the easiest to interpret was Python (median of 30.0). A similar
 525 hierarchy between the different languages is obtained for the Halstead Effort (E), which estimates the
 526 mental activity needed to translate an algorithm into code written in a specific language. The Effort is
 527 linearly proportional to both Difficulty and Volume. The unit of measure of the metric is the number of
 528 elementary mental discriminations [15].

529 The Halstead Length (L) metric is given by the total number of operator occurrences and the total
 530 number of operand occurrences. The Halstead Volume (V) metric is the information content of the
 531 program, linearly dependent on its vocabulary. Rust code had the third-highest mean and median Hal-
 532 stead Length (602.2 mean, 550.0 median) and Halstead Volume (4,032 mean, 3,610 median), again
 533 below those measured for C and C++. The results measured for all considered source files were in line
 534 with existing programming guidelines (Halstead Volume lower than 8000). The reported results about
 535 Length and Volume were, to some extent, expectable since these metrics are largely correlated to the
 536 number of lines of code present in a source file [46].

537 The Halstead Time metric (T) is computed as the Halstead Effort divided by 18. It estimates the time
 538 in seconds that it should take a programmer to implement the code. A mean and median T of 11,064
 539 and 13,719 seconds were measured, respectively, for the Rust programming language. These values
 540 are significantly distant from those measured for Python and TypeScript (the lowest) and from those
 541 measured for C and C++ (the highest).

542 Finally, the Halstead Bugs Metric estimates the number of bugs that are likely to be found in the soft-
 543 ware program. It is given by a division of the Volume metric by 3000. We estimated a mean value of
 544 1.1 (median 1.3) bugs per file with the Rust programming language on the considered set of source
 545 artifacts.

546 An analysis of the outliers of the distributions of measurements regarding RQ3 was performed. A

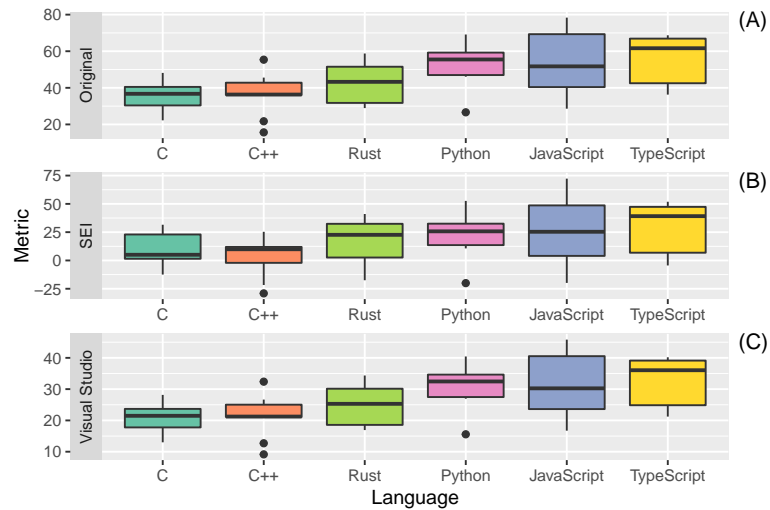


Figure 7. Distribution of Maintainability Indexes (A: Original; B: SEI; C: Visual Studio) for all the considered programming languages

Table 18. Mean (Median) values of Maintainability Indexes for all the considered programming languages

Language	Original	SEI	Visual Studio
C	35.9 (36.7)	10.5 (5.0)	21.0 (21.5)
C++	36.5 (36.3)	3.6 (9.9)	21.3 (21.2)
Rust	43.0 (43.3)	15.8 (22.6)	25.1 (25.3)
Python	52.5 (55.5)	23.3 (25.7)	30.7 (32.5)
JavaScript	54.2 (51.7)	27.7 (25.3)	31.7 (30.3)
TypeScript	55.9 (61.6)	29.4 (39.2)	32.7 (36.0)

547 relevant outlier for the CC metric was *revcomp.cpp*, in which the usage of many nested loops and
 548 conditional statements inside class methods significantly increased the computed complexity. For the
 549 set of Python source files, *knucleoutide.py* had the highest CC due to the usage of nested code; the
 550 same effect occurred for *fannchuckredux.rs* which had the highest CC and COGNITIVE complexity
 551 for the Rust language. The JavaScript and TypeScript versions of *fannchuckredux* both presented a
 552 high usage of nested code, but the lower level of COGNITIVE complexity for the TypeScript version
 553 suggests a better-written source code artifact. The few outliers that were found for the Halstead metrics
 554 measurements were principally for C++ source artifacts and mostly related to the higher PLOC and
 555 number of operands of the C++ source codes.

556 Table 16 reports the results of the application of the Kruskal-Wallis non-parametric test on the set of
 557 measures for RQ3. No statistical significance was measured for the differences in the measurements
 558 of the two metrics related to CC. A statistically significant difference was measured for the averaged
 559 COGNITIVE complexity. Regarding the Halstead metrics, all differences were statistically significant
 560 with the exception of those for the *Difficulty* metric. Post-hoc statistical tests focused on the compar-
 561 ison between Rust and the other languages (table 17) highlighted that Rust had a significantly lower
 562 average COGNITIVE complexity than all the other considered languages.

Answer to RQ3: The Rust software artifacts exhibited an average Cyclomatic Complexity (mean 2.0 per function) and a significantly lower COGNITIVE complexity (mean 1.5 per function) than all other languages. Rust was the third-highest performing language, after C and C++, for the Halstead metric values.

Table 19. Null hypotheses and p-values for RQ4 metrics obtained by applying Kruskal-Wallis chi-squared test¹¹

Name	Description	p-value	Decision	Significance
$H_{MI\ Original}$	No significant difference in MI Original for the artifacts	0.006002	Reject	**
$H_{MI\ SEI}$	No significant difference in MI SEI for the artifacts	0.1334	Not Reject	.
$H_{MI\ Visual\ Studio}$	No significant difference in MI Visual Studio for the artifacts	0.006002	Reject	**

Table 20. p-values for post-hoc Wilcoxon Signed Rank test for RQ4 metrics between Rust and the other languages

Metric	C	C	JavaScript	Python	TypeScript
$MI_{Original}$	0.2624	0.3308	0.2698	0.2624	0.2624
$MI_{VisualStudio}$	0.2624	0.3308	0.2698	0.2624	0.2624

565 4.4 RQ4 - Code maintainability

566 The boxplots in Figure 7 and Table 18 report the distributions, mean, and median of the Maintainability
567 Indexes computed for the six different programming languages.

568 The Maintainability Index is a composite metric aiming to give an estimate of software maintainability
569 over time. The Metric has correlations with the Halstead Volume (V), the Cyclomatic Complexity (CC),
570 and the number of lines of code of the source under examination.

571 The source files written in Rust had an average MI that placed the fourth among all considered pro-
572 gramming languages, regardless of the specific formula used for the calculation of the MI. Minor dif-
573 ferences in the placement of other languages occurred, e.g., the median MI for C is higher than for
574 C++ with the original formula for the Maintainability Index and lower with the SEI formula. Regard-
575 less of the formula used to compute MI, the highest maintainability was achieved by the TypeScript
576 language, followed by Python and JavaScript. These results were expectable in light of the previous
577 metrics measured, given the said strong dependency of the MI on the raw size of source code.

578 It is interesting to underline that, in accordance with the original guidelines for the MI computation, all
579 the values measured for the software artifacts under study would suggest hard to maintain code, being
580 the threshold for easily maintainable code set to 80. On the other hand, according to the documentation
581 of the Visual Studio MI metric, all source artifacts under test can be considered as easy to maintain
582 (MI_{VS20}).

583 Outliers in the distributions of MI values were mostly found for C++ sources and were likely related to
584 higher values of SLOC, CC, and Halstead Volume, all leading to very low MI values.

585 Table 19 reports the results of the application of the Kruskal-Wallis non-parametric test on the set of
586 measures for RQ4. The measured differences were statistically significant for the original MI metric
587 and for the version employed by Visual Studio. Post-hoc statistical tests focused on the comparison be-
588 tween Rust, and the other languages (table 20) highlighted that difference was statistically significant.

589 **Answer to RQ4:** Rust exhibited an average Maintainability Index, regardless of the specific formula
590 used (median values of 43.3 for MI_O , 22.6 for MI_{SEI} , 25.3 for MI_{VS}). Highest Maintainability index
were obtained for Python, JavaScript and TypeScript.

591 However, it is worth mentioning that several works in the literature from the latest years have high-
592 lighted the intrinsic limitations of the MI metric. A study by T. Kuipers underlines how the MI metric
593 exposes limitations, particularly for systems built using object-oriented languages since it is based
594 on the CC metric that will be largely influenced by small methods with small complexity; hence both
595 will inevitably be low [25]. Counsell et al. as well warn against the usage of MI for Object-Oriented
596 software, highlighting the class size as a primary confounding factor for the interpretation of the MI
597 metric [10]. Several works have tackled the issue of adapting the MI to object-oriented code: Kaur et
598 al., for instance, propose the utilization of package-level metrics [23]. Kaur et al. have evaluated the
599 correlation between the traditional MI metrics and the more recent maintainability metrics provided
600 by the literature, like the CHANGE metric. They found that a very scarce correlation can be measured
601 between MI and CHANGE [21]. Lastly, many white and grey literature sources underline how different

602 metrics for the MI can provide different estimations of the maintainability for the same code. This issue
603 is reflected by our results. While the comparisons between different languages are mostly maintained
604 by all three MI variations, it can be seen that all average values for original and SEI MI suggest very
605 low code maintainability, while the average values for the Visual Studio MI would suggest high code
606 maintainability for the same code artifacts.

607 5 CONCLUSION AND FUTURE WORK

608 In this paper, we have evaluated the complexity and maintainability of Rust code by using static metrics
609 and compared the results on equivalent software artifacts written in C, C++, JavaScript, Python, and
610 TypeScript. The main findings of our evaluation study are the following:

- 611 • The Rust language exhibited average verbosity between all considered languages, with lower
612 verbosity than C and C++;
- 613 • The Rust language exhibited the most structured code organization of all considered languages.
614 More specifically, the examined source code artifacts in Rust had a significantly higher number
615 of arguments than most of the other languages;
- 616 • The Rust language exhibited average CC and values for Halstead metrics. Rust had a signifi-
617 cantly lower COGNITIVE complexity with respect to all other considered languages;
- 618 • The Rust language exhibited average compound maintainability indexes. Comparative analyses
619 showed that the maintainability indexes were slightly higher (hinting at better maintainability)
620 than C and C++.

621 All the evidence collected in this paper suggests that the Rust language can produce less verbose, more
622 organized, and readable code than C and C++, the languages to which it is more similar in terms of
623 code structure and syntax. The difference in maintainability with these two languages was not signifi-
624 cant. On the other hand, the Rust language provided lower maintainability than that measured for more
625 sophisticated and high-level object-oriented languages.

626 It is worth underlining that the source artifacts written in the Rust language exhibited the lowest COG-
627 NITIVE complexity, meaning that the language can guarantee the highest understandability of source
628 code compared to all others. Understandability is a fundamental feature of code during its evolution
629 since it may significantly impact the required effort for maintaining and fixing it.

630 This work contributes to the existing literature of the field as a first, preliminary evaluation of static
631 qualities related to maintainability for the Rust language and a first comparison with a set of other pop-
632 ular programming languages. As the prosecution of this work, we plan to perform further developments
633 on the *rust-code-analysis* tool such that it can provide more metric computation features. At the present
634 time, for instance, the tool is not capable of computing class-level metrics. However, it can only be
635 employed to compute metrics only on function and class methods.

636 We also plan to implement parsers for more programming languages (e.g., Java) to enable additional
637 comparisons. We also plan to extend our analysis to real projects composed of a significantly higher
638 amount of code lines that embed different programming paradigms, such as the functional and concur-
639 rent ones. To this extent, we plan to mine software projects from open source libraries, e.g., GitHub.

640 REFERENCES

- 641 [1] Aggarwal, K. K., Singh, Y., and Chhabra, J. K. (2002). An integrated measure of software main-
642 tainability. In *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.*
643 *02CH37318)*, pages 235–241. IEEE.
- 644 [2] Alqadi, B. S. and Maletic, J. I. (2020). Slice-based cognitive complexity metrics for defect prediction.
645 In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*
646 *(SANER)*, pages 411–422. IEEE.
- 647 [3] Amara, D. and Rabai, L. B. A. (2017). Towards a new framework of software reliability measurement
648 based on software metrics. *Procedia Computer Science*, 109:725–730.

- 649 [4] Ardito, L., Barbato, L., Castelluccio, M., Coppola, R., Denizet, C., Ledru, S., and Valsesia, M.
650 (2020a). rust-code-analysis: A rust library to analyze and extract maintainability information from
651 source codes. *SoftwareX*, 12:100635.
- 652 [5] Ardito, L., Coppola, R., Barbato, L., and Verga, D. (2020b). A tool-based perspective on software
653 code maintainability metrics: A systematic literature review. *Scientific Programming*, 2020.
- 654 [6] Astrauskas, V., Müller, P., Poli, F., and Summers, A. J. (2019). Leveraging rust types for modular
655 specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–
656 30.
- 657 [7] Balasubramanian, A., Baranowski, M. S., Burtsev, A., Panda, A., Rakamarić, Z., and Ryzhyk, L.
658 (2017). System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot
659 Topics in Operating Systems*, pages 156–161.
- 660 [8] Barón, M. M. n., Wyrich, M., and Wagner, S. (2020). An empirical validation of cognitive complexity
661 as a measure of source code understandability. In *Proceedings of the 14th ACM / IEEE International
662 Symposium on Empirical Software Engineering and Measurement (ESEM), ESEM '20*, New York,
663 NY, USA. Association for Computing Machinery.
- 664 [9] Bray, M., Brune, K., Fisher, D. A., Foreman, J., and Gerken, M. (1997). C4 software technology refer-
665 ence guide—a prototype. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering
666 Inst.
- 667 [10] Counsell, S., Liu, X., Eldh, S., Tonelli, R., Marchesi, M., Concas, G., and Murgia, A. (2015). Re-
668 visiting the ‘maintainability index’ metric from an object-oriented perspective. In *2015 41st Euromicro
669 Conference on Software Engineering and Advanced Applications*, pages 84–87. IEEE.
- 670 [11] Ferreira, J., Zwinderman, A., et al. (2006). On the benjamini–hochberg method. *Annals of Statistics*,
671 34(4):1827–1849.
- 672 [12] Flauzino, M., Veríssimo, J., Terra, R., Cirilo, E., Durelli, V. H., and Durelli, R. S. (2018). Are you
673 still smelling it? a comparative study between java and kotlin language. In *Proceedings of the VII
674 Brazilian symposium on software components, architectures, and reuse*, pages 23–32.
- 675 [13] Frantz, R. Z., Rehbein, M. H., Berlezi, R., and Roos-Frantz, F. (2019). Ranking open source
676 application integration frameworks based on maintainability metrics: A review of five-year evolution.
677 *Software: Practice and Experience*, 49(10):1531–1549.
- 678 [14] Gill, G. K. and Kemerer, C. F. (1991). Cyclomatic complexity density and software maintenance
679 productivity. *IEEE transactions on software engineering*, 17(12):1284.
- 680 [15] Halstead, M. H. (1977). *Elements of software science*, volume 7. Elsevier New York.
- 681 [16] Hariprasad, T., Vidhyagarán, G., Seenu, K., and Thirumalai, C. (2017). Software complexity
682 analysis using halstead metrics. In *2017 International Conference on Trends in Electronics and
683 Informatics (ICEI)*, pages 1109–1113. IEEE.
- 684 [17] ISO (1991). Iso 9126 software quality characteristics. [http://www.sqa.net/iso9126.](http://www.sqa.net/iso9126.html)
685 [html](http://www.sqa.net/iso9126.html). Online; accessed 08/12/2020.
- 686 [18] ISO/IEC (2011). Iso/iec 25010:2011 systems and software engineering — systems and software
687 quality requirements and evaluation (square) — system and software quality models. <https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en>. Online; accessed
688 08/12/2020.
- 689 [19] Jedlitschka, A. and Pfahl, D. (2005). Reporting guidelines for controlled experiments in software
690 engineering. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages
691 10–pp. IEEE.
- 692 [20] Jingqiu Shao and Yingxu Wang (2003). A new measure of software complexity based on cognitive
693 weights. *Canadian Journal of Electrical and Computer Engineering*, 28(2):69–74.
- 694 [21] Kaur, A., Kaur, K., and Pathak, K. (2014a). A proposed new model for maintainability index of open
695 source software. In *Proceedings of 3rd International Conference on Reliability, Infocom Technologies
696 and Optimization*, pages 1–6. IEEE.
- 697 [22] Kaur, A., Kaur, K., and Pathak, K. (2014b). Software maintainability prediction by data mining of
698 software code metrics. In *2014 International Conference on Data Mining and Intelligent Computing
699 (ICDMIC)*, pages 1–6. IEEE.
- 700 [23] Kaur, K. and Singh, H. (2011). Determination of maintainability index for object oriented systems.
701 *ACM SIGSOFT Software Engineering Notes*, 36(2):1–6.
- 702 [24] Köster, J. (2016). Rust-bio: a fast and safe bioinformatics library. *Bioinformatics*, 32(3):444–446.
- 703

- 704 [25] Kuipers, T. and Visser, J. (2007). Maintainability index revisited—position paper. In *Special session*
705 *on system quality and maintainability (SQM 2007) of the 11th European conference on software*
706 *maintenance and reengineering (CSMR 2007)*. Citeseer.
- 707 [26] Levy, A., Campbell, B., Ghena, B., Pannuto, P., Dutta, P., and Levis, P. (2017). The case for writing
708 a kernel in rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–7.
- 709 [27] Ludwig, J. and Cline, D. (2019). Cbr insight: measure and visualize source code quality. In *2019*
710 *IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 57–58. IEEE.
- 711 [28] Ludwig, J., Xu, S., and Webber, F. (2017). Compiling static software metrics for reliability and
712 maintainability from github repositories. In *2017 IEEE International Conference on Systems, Man,*
713 *and Cybernetics (SMC)*, pages 5–9. IEEE.
- 714 [29] Matsakis, N. D. and Klock, F. S. (2014). The rust language. *ACM SIGAda Ada Letters*, 34(3):103–
715 104.
- 716 [30] Matsushita, T. and Sasano, I. (2017). Detecting code clones with gaps by function applications. In
717 *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*,
718 pages 12–22.
- 719 [31] Microsoft (2011). Code Metrics – Maintainability Index. [https://docs.microsoft.com/
720 en-gb/archive/blogs/zainnab/code-metrics-maintainability-index](https://docs.microsoft.com/en-gb/archive/blogs/zainnab/code-metrics-maintainability-index). On-
721 line; accessed 08/12/2020.
- 722 [32] Molnar, A. and Motogna, S. (2017). Discovering maintainability changes in large software systems.
723 In *Proceedings of the 27th International Workshop on Software Measurement and 12th International*
724 *Conference on Software Process and Product Measurement*, pages 88–93.
- 725 [33] Mshelia, Y. U. and Apeh, S. T. (2019). Can software metrics be unified? In *International Conference*
726 *on Computational Science and Its Applications*, pages 329–339. Springer.
- 727 [34] Mshelia, Y. U., Apeh, S. T., and Edoghogho, O. (2017). A comparative assessment of software
728 metrics tools. In *2017 International Conference on Computing Networking and Informatics (ICCNI)*,
729 pages 1–9. IEEE.
- 730 [35] Nair, L. S. and Swaminathan, J. (2020). Towards reduction of software maintenance cost through
731 assignment of critical functionality scores. In *2020 5th International Conference on Communication*
732 *and Electronics Systems (ICCES)*, pages 199–204. IEEE.
- 733 [36] Nguyen, V., Deeds-Rubin, S., Tan, T., and Boehm, B. (2007). A sloc counting standard. In *Cocomo*
734 *ii forum*, volume 2007, pages 1–16. Citeseer.
- 735 [37] Nuñez-Varela, A. S., Pérez-Gonzalez, H. G., Martínez-Perez, F. E., and Soubervielle-Montalvo, C.
736 (2017). Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164 –
737 197.
- 738 [38] Oman, P. and Hagemester, J. (1992). Metrics for assessing a software system’s maintainability. In
739 *Proceedings Conference on Software Maintenance 1992*, pages 337–338. IEEE Computer Society.
- 740 [39] Ottenstein, L. M., Schneider, V. B., and Halstead, M. H. (1976). Predicting the number of bugs
741 expected in a program module.
- 742 [40] Robson, C. and McCartan, K. (2016). *Real world research*. John Wiley & Sons.
- 743 [41] Rust (2020). Rust in production. <https://www.rust-lang.org/>. Online; accessed
744 07/12/2020.
- 745 [42] Saifan, A. A., Alsghaier, H., and Alkhateeb, K. (2018). Evaluating the understandability of android
746 applications. *International Journal of Software Innovation (IJSI)*, 6(1):44–57.
- 747 [43] Sarwar, M. I., Tanveer, W., Sarwar, I., and Mahmood, W. (2008). A comparative study of mi tools:
748 Defining the roadmap to mi tools standardization. In *2008 IEEE International Multitopic Conference*,
749 pages 379–385. IEEE.
- 750 [44] Schnappinger, M., Osman, M. H., Pretschner, A., and Fietzke, A. (2019). Learning a classifier for
751 prediction of maintainability based on static analysis tools. In *2019 IEEE/ACM 27th International*
752 *Conference on Program Comprehension (ICPC)*, pages 243–248. IEEE.
- 753 [45] Sjøberg, D. I., Anda, B., and Mockus, A. (2012). Questioning software maintenance metrics: a
754 comparative case study. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical*
755 *Software Engineering and Measurement*, pages 107–110. IEEE.
- 756 [46] Tashtoush, Y., Al-Maolegi, M., and Arkok, B. (2014). The correlation among software complexity
757 metrics with case study. *arXiv preprint arXiv:1408.4523*.
- 758 [47] Toomim, M., Begel, A., and Graham, S. L. (2004). Managing duplicated code with linked editing.

- 759 In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*, pages 173–180. IEEE.
- 760 [48] Uzlu, T. and Şaykol, E. (2017). On utilizing rust programming language for internet of things. In
- 761 *2017 9th International Conference on Computational Intelligence and Communication Networks*
- 762 *(CICN)*, pages 93–96. IEEE.
- 763 [49] Welker, K. D. (2001). The software maintainability index revisited. *CrossTalk*, 14:18–21.
- 764 [50] Zhou, Y. and Leung, H. (2007). Predicting object-oriented software maintainability using multivari-
- 765 ate adaptive regression splines. *Journal of systems and software*, 80(8):1349–1361.

766 **Listing 1.** Sample output of the *rust-code-analysis* tool for the Rust version of the binarytrees
 767 algorithm.

```

76b {
76c   "name": "Assets/Rust/binarytrees.rs",
77   "start_line": 1,
77a  "end_line": 75,
77b  "kind": "unit",
77c  "metrics": {
77d    "nargs": {
77e      "sum": 14.0,
77f      "average": 2.0
78   },
78a   "nexits": {
78b     "sum": 3.0,
78c     "average": 0.42857142857142855
78d   },
78e   "cognitive": {
78f     "sum": 5.0,
78g     "average": 0.7142857142857143
78h   },
78i   "cyclomatic": {
78j     "sum": 12.0,
78k     "average": 1.5
78l   },
78m   "halstead": {
78n     "n1": 22.0,
78o     "N1": 193.0,
78p     "n2": 43.0,
78q     "N2": 140.0,
78r     "length": 333.0,
78s     "estimated_program_length": 331.4368800622107,
390   "purity_ratio": 0.9953059461327649,
391   "vocabulary": 65.0,
392   "volume": 2005.4484817384753,
393   "difficulty": 35.81395348837209,
394   "level": 0.02792207792207792,
395   "effort": 71823.03864830818,
396   "time": 3990.168813794899,
397   "bugs": 0.5759541722145377
398   },
399   "loc": {
400     "sloc": 75.0,
401     "ploc": 56.0,
402     "lloc": 31.0,
403     "cloc": 7.0,
404     "blank": 12.0
405   },
406   "nom": {
407     "functions": 4.0,
408     "closures": 3.0,
409     "total": 7.0
410   },
411   "mi": {
412     "mi_original": 58.75785297946959,
413     "mi_sei": 33.08134287773029,
414     "mi_visual_studio": 34.36131753185356
415   }
416 }

```

825 **Listing 2.** Sample output of the *analyzer.py* script for the Rust version of the binarytrees algorithm.

```

82b {
82c   "SLOC": 75,
82d   "PLOC": 56,
82e   "LLOC": 31,
82f   "CLOC": 7,
83   "BLANK": 12,
83a  "CC.SUM": 12,
83b  "CC.AVG": 1.5,
83c  "COGNITIVE.SUM": 5,
83d  "COGNITIVE.AVG": 0.7142857142857143,
83e  "NARGS.SUM": 14,
83f  "NARGS.AVG": 2.0,
83g  "NEXITS": 3,
83h  "NEXITS.AVG": 0.42857142857142855,
83i  "NOM": {
83j    "functions": 4,
83k    "closures": 3,
83l    "total": 7
83m  },
83n  "HALSTEAD": {
83o    "n1": 22,
242   "n2": 43,
243   "N1": 193,
244   "N2": 140,
245   "Vocabulary": 65,
246   "Length": 333,
247   "Volume": 2005.4484817384753,
248   "Difficulty": 35.81395348837209,
249   "Level": 0.02792207792207792,
250   "Effort": 71823.03864830818,
251   "Programming_time": 3990.168813794899,
252   "Bugs": 0.5759541722145377,
253   "Estimated_program_length": 331.4368800622107,
254   "Purity_ratio": 0.9953059461327649
255   },
256   "MI": {
257     "Original": 58.75785297946959,
258     "Sei": 33.08134287773029,
259     "Visual_Studio": 34.36131753185356
260   }
261 }

```

867
868

Listing 3. Sample output of the *compare.py* script for the C++/Rust comparisons of the binarytrees algorithm. The *__old* label identifies C++ metric values, while *__new* the Rust ones.

```
86b {
87b   "SLOC": {
87c     "__old": 139,
87d     "__new": 75
87e   },
87f   "PLOC": {
87g     "__old": 98,
87h     "__new": 56
87i   },
87j   "LLOC": {
87k     "__old": 25,
87l     "__new": 31
87m   },
87n   "CLOC": {
87o     "__old": 15,
87p     "__new": 7
87q   },
87r   "BLANK": {
87s     "__old": 26,
87t     "__new": 12
87u   },
87v   "CC.SUM": {
87w     "__old": 19,
87x     "__new": 12
87y   },
87z   "CC.AVG": {
87aa     "__old": 1.4615384615384615,
87ab     "__new": 1.5
87ac   },
87ad   "COGNITIVE.SUM": {
87ae     "__old": 8,
87af     "__new": 5
87ag   },
87ah   "COGNITIVE.AVG": {
87ai     "__old": 0.8888888888888888,
87aj     "__new": 0.7142857142857143
87ak   },
87al   "NARGS.SUM": {
87am     "__old": 2,
87an     "__new": 14
87ao   },
87ap   "NARGS.AVG": {
87aq     "__old": 0.2222222222222222,
87ar     "__new": 2
87as   },
87at   "NEXITS": {
87au     "__old": 5,
87av     "__new": 3
87aw   },
87ax   "NEXITS.AVG": {
87ay     "__old": 0.5555555555555556,
87az     "__new": 0.42857142857142855
87ba   },
87bb   "NOM": {
87bc     "functions": {
87bd       "__old": 9,
87be       "__new": 4
87bf     },
87bg     "closures": {
87bh       "__old": 0,
87bi       "__new": 3
87bj     },
87bk     "total": {
87bl       "__old": 9,
87bm       "__new": 7
87bn     }
87bo   },
87bp   "HALSTEAD": {
87bq     "n1": {
87br       "__old": 28,
```

```
71b     "__new": 22
71c   },
71d   "n2": {
71e     "__old": 56,
71f     "__new": 43
71g   },
71h   "N1": {
71i     "__old": 251,
71j     "__new": 193
71k   },
71l   "N2": {
71m     "__old": 173,
71n     "__new": 140
71o   },
71p   "Vocabulary": {
71q     "__old": 84,
71r     "__new": 65
71s   },
71t   "Length": {
71u     "__old": 424,
71v     "__new": 333
71w   },
71x   "Volume": {
71y     "__old": 2710.3425872581947,
71za     "__new": 2005.4484817384753
71zb   },
71zc   "Difficulty": {
71zd     "__old": 43.25,
71ze     "__new": 35.81395348837209
71zf   },
71zg   "Level": {
71zh     "__old": 0.023121387283236993,
71zha     "__new": 0.02792207792207792
71zib   },
71zic   "Effort": {
71zid     "__old": 117222.31689891692,
71zie     "__new": 71823.03864830818
71zif   },
71zif   "Programming_time": {
71zih     "__old": 6512.3509388287175,
71zih     "__new": 3990.168813794899
71zij   },
71zij   "Bugs": {
71zjk     "__old": 0.7983970910222301,
71zjk     "__new": 0.5759541722145377
71zkl   },
71zkl   "Estimated_program_length": {
71zkm     "__old": 459.81781345283866,
71zkm     "__new": 331.4368800622107
71zkn   },
71zkn   "Purity_ratio": {
71zko     "__old": 1.0844759751246196,
71zko     "__new": 0.9953059461327649
71zkp   }
71zqp   },
71zqp   "MI": {
71zqr     "Original": {
71zqs       "__old": 45.586404609681736,
71zqs       "__new": 58.75785297946959
71zqt     },
71zqt     "Sei": {
71zqu       "__old": 16.3624350913677,
71zqu       "__new": 33.08134287773029
71zqu     },
71zqu     "Visual_Studio": {
71zqv       "__old": 26.658716146012715,
71zqv       "__new": 34.36131753185356
71zqv     }
71zqr   }
71zrp   }
71zrp }
```