Put The "Code" Back In "Code Comprehension Study"

Kyle D. Chin
Computer Science Department
University of British Columbia
Vancouver, Canada

Reid Holmes
Computer Science Department
University of British Columbia
Vancouver, Canada

ABSTRACT

Despite numerous code comprehension studies, there is no consensus on what code features affect comprehensibility for software engineers. Results in these studies often contradict each other, even using similar methodologies or datasets. These contradictions make it difficult for both researchers and practitioners to derive replicable findings into code comprehension. Most prior studies employ controlled experiments with synthetic code, student subjects, single languages, short methods, and small sample sizes; these design decisions threaten their authenticity and generalizability. In this work, we approach code comprehension with a different set of restrictions: we examine 604k methods, across all method lengths, from real open open-source code systems across five programming languages, *but do not control nor observe the environment in which the developers actually comprehended the code.* We argue that we can still derive insight into comprehensibility since developers wrote, evolved, and reviewed the source code we examine, and that this means *someone* found the code comprehensible. Our methodology provides insight into reasons for prior inconsistencies and demonstrates the influence of experimental design decisions on results while providing actionable insights for software engineers who are trying to write comprehensible code. We find that *Length*—the number of lines of code in a method—heavily correlates with prior comprehension metrics and may be a confounder of prior results. We also find that comment presence serves as a useful comprehension metric: across all languages and projects, comments co-locate with large, complex, and hard-to-read code.

CCS CONCEPTS

• **Software and its engineering → Patterns**; • **General and reference → Empirical studies**.

KEYWORDS

Software Engineering, Code Comprehension, Code Comments

ACM Reference Format:
Kyle D. Chin and Reid Holmes. 2026. Put The "Code" Back In "Code Comprehension Study". In *34th IEEE/ACM International Conference on Program Comprehension (ICPC '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3794763.3794806

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
ICPC '26, April 12–13, 2026, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2482-4/2026/04
https://doi.org/10.1145/3794763.3794806

1 INTRODUCTION

Code is difficult to understand. It seems like it is even more difficult to understand why code is difficult to understand. Many studies have shared a similar approach to code comprehension: they present subjects with code snippets with slight variations in composition (e.g., naming conventions, decomposition, comment usage, etc.) and measure the subject's comprehension using various methods, such as asking the subject to make changes or fixes to the code [21, 23, 38], timing how long it took them to comprehend it [12, 23, 38, 42, 52, 53], or even measuring their physical stimuli [21, 24, 37, 42]. Despite numerous studies in this space, there are conflicting results on what structural aspects of code actually affect its comprehensibility. Aspects such as naming conventions, method length, and source code comments have been found to be of conflicting or inconclusive value: some studies claim they are helpful for comprehension [44, 59], others find they are harmful [33, 42], and others yet find that they have little effect at all [10, 53]. These conflicting results highlight the difficulty in effectively measuring code comprehension. In order for researchers to design studies that appropriately measure code comprehension going forward, it is essential to understand: *why are results from code comprehension studies so conflicting?*

A close examination of prior studies suggests that authenticity and scale are two primary factors that threaten their ecological validity and may help explain their inconsistent results. Study designers have made design choices that may have compromised the authenticity of their experiments compared to the kind of code comprehension that developers undergo in practice. For example, most studies primarily used students as subjects [10, 12, 13, 21, 23, 46, 51–53, 57, 59], studied only Java code [10, 12, 33, 38, 42, 51–53], synthetic code [12, 38, 46, 51, 57, 59], and/or only small snippets of code [10, 12, 19, 33, 38, 42, 44, 46]. Studies that found line width as a key metric for comprehensibility had subjects evaluate methods *no longer than 12 lines of code* [10, 46]. Subsequent studies found that these results do not generalize to longer snippets of code [15, 44] and conflict with other findings despite similar methodologies [19, 53, 61] or even the same exact dataset [36, 44]. When code comprehension takes place in practice, software developers are both familiar with and invested in the code they are attempting to comprehend. Subjects in each of these studies presented neither, furthering the authenticity gap between research and practice.

To understand how ecological design choices may cause disparate study results, we conduct an empirical study that inverts the tradeoffs of prior studies: we examine a large breadth and quantity of code that developers have comprehended within their normal workflows, but do not run a controlled user study. Our empirical methodology analyzes code comprehension metrics in over 604k methods across 49 large, professionally-managed software projects in 5 programming languages, but notably *does not involve subjects*

making perceived comprehensibility judgments in a controlled environment. We argue that code review processes in large, professionally managed projects assures that the code has been comprehended by someone.[1] Even if we can no longer directly query comprehensibility as a controlled study can, the quantity, length, and breadth of code in our study leads to findings that more directly generalize to the kinds of software being written in practice. Importantly, these findings provide novel reasoning into prior inconsistent results and highlight how study design directly influences findings. We find the following primary results:

(1) Previously-studied comprehension metrics exhibit large variance in real code, even within methods that have similar *Length*
(lines of code). This leads to high standard error, especially in small samples, which may explain prior inconsistencies.
(2) These metrics also provide discriminatory power in a *Length*-unconstrained environment. However, they are highly correlated with *Length*, which is usually controlled for in controlled studies. *Length*-aware studies are necessary to better understand code comprehension.
(3) Methods with comments are typically longer, more complex, and less comprehensible than methods without comments. This may indicate that comment-presence is a useful comprehension metric since comments may be used to explain harder-to-comprehend code. This reflects a common software engineering belief that has yet to be consistently verified by replicable, evidence-based research.

2 MOTIVATION & RELATED WORK

The fundamental difficulty in code comprehension research is that comprehension, as a concept, is a *latent variable*—we cannot directly measure it. As a result, studies usually attempt to approximate comprehension using metrics as a proxy for how well a subject comprehended code and/or about the features that impacted comprehension of the code itself. Finding a consistent global measure of comprehensibility has proven elusive due to variation across subjects and their experience levels, systems and their programming languages, and the diversity of tasks developers need accurate comprehension to perform effectively. For example, prior studies often measure both *Readability* (R), or how easy it is to *read* code, and *Comprehensibility* (C), or how easy it is to *read and understand* code. Intuitively, one would expect these to align: generally, if code is more readable, then it should also be more comprehensible (and vice versa). However, examining the **Findings** columns in Table 1 this is not often the case (R and C differ in most studies that measure both metrics). Scalabrino et al., studied some of these inconsistencies in two separate studies with similar methodologies and conclusions found that *none* of their 121 static code metrics strongly correlated with comprehension [52, 53]. They note that the experience level of the subject was a better predictor than any features of the code itself. These inconsistencies hint that our existing metrics and/or study methods need to be improved to effectively measure code comprehension.

[1] We validate this approach in Section 5.

2.1 Inconsistencies in Metrics

We categorize metrics used to measure comprehensibility into three general dimensions and discuss their results from prior studies:

(1) **Volume:** Metrics that measure the quantity of code.
(2) **Complexity:** Metrics that measure how dense or complicated code is.
(3) **Expressivity:** Metrics that measure the quality of the code and how it was written.

Metric definitions have slight variations from study-to-study, so for this section we refer to them conceptually as opposed to formally (our formal metric definitions are in Section 4.4).

2.1.1 Inconsistencies with Volume metrics. Intuitively, code with larger volume should be harder to comprehend. While prior studies generally agree with this intuition, they largely disagree on which volume metrics actually drive this impact. These results are displayed in the **Findings→Volume** column in Table 1. Buse et al. found that average line width and average number of identifiers per line of code were the most detrimental to readability [10]. However, Posnett, et al. re-examined the same dataset with different statistical techniques and found that a better model for predicting the subjective readability scores can use just *Entropy*, *Length* (lines of code), and *Halstead* volume—and that Buse' model does not generalize well past code snippets longer than 7 LOC [44]. Mi, et al. also re-examined the same dataset using causal methods and found that the average number of assignments and identifiers are the only metrics that have a strong negative impact on readability [36]. Dorn et al. found that average line width is the only feature that predicts readability across languages, and that less readable code is 2.2x more likely to contain defects [19]. Several studies have investigated *Length*'s effect on code comprehension with mixed results: some found larger *Length* negatively impacted comprehension [12, 42], others found no effect [23, 46], and others yet found it improved comprehension [44].

2.1.2 Inconsistencies in Complexity metrics. Intuitively, code that is more complex should be harder to comprehend. However, as shown in the **Findings→Complex.** column in Table 1, studies run by Börstler et al, Feigenspan et al, and Peitek et al have shown that *McCabe* (cyclomatic complexity) correlates with cognitive load but not actual code comprehension [12, 23, 42, 49] Others find it impacts neither readability nor comprehension [52, 53].

2.1.3 Inconsistencies in Expressivity metrics. Intuitively, given code with similar volume and complexity, more expressive code should be easier to comprehend. Most prior studies use *Entropy*, identifier naming, or comments. *Entropy* measures the disorder of tokens and is generally found to negatively impact readability [33, 44]. Variants in identifier names have sometimes been found to impact readability [4, 5, 11, 32], though several studies also find no impact [3, 38, 57]. While it would intuitively follow from popular software-engineering doctrine that source code comments would only help comprehension as found in several studies [10, 36, 46, 51, 57, 59], many other studies find they have either no effect [12, 19, 38, 44, 61] or even a negative effect on comprehension [33]. Studies and even their replications find conflicting results on the impact of comments on comprehension. Börstler et al. found that comments positively

Table 1: Summary of code comprehension study designs and findings. In terms of study design, most prior work examines one language, leverages students' subjective opinions, and only investigates small fragments of often synthetic code. In terms of findings, the results are broadly inconsistent. Readability (R) and Comprehensibility (C) do not consistently align. Volume, Complexity, and Expressivity measuring metrics do not find similar results between studies. Legend: Lines of Code (LOC); † represents synthetic code, ∗ toy code; Negative Impact (⇓); No Impact (∼); Positive Impact (⇑); Blank indicates not evaluated.

	Study Scope					Findings		
Study	Language(s)	Subjects	Evals. Per	LOC Per Eval.	Demographic	Volume	Complex.	Express.
Scalabrino [52, 53]	Java	109	8	30-70	Students & Devs	R∼/C∼	R∼/C∼	R∼/C∼
Börstler [12]	Java	104	5	22-54†	Students	R∼/C⇓	R∼/C∼	R∼/C∼
Sampaio [51]	Java	106	6	22-54†	Students			R∼/C⇑
Nielebock [38]	Java	277	6	7-27†	Students & Devs			R⇑/C∼
Peitek [42]	Java	19	3	≤ 30	Students	C⇓	C∼	
Feigenspan [23]	Java, AspectJ	21	1	Project	Students	C∼	C∼	
Ribeiro [46]	Python	66	16	2-12∗†	Students	R∼/C∼		R⇑/C⇑
Buse [10]	Java	120	10	4-11	Students	R⇓		R⇑
Posnett [44]	Java	120	10	4-11	Students	R⇓/R⇑		R∼
Mi [36]	Java	5,638	6-10	2-70	Students & Devs	R⇓		R⇑⇑
Dorn [19]	Java, Python, CUDA	5,468	6	10-50	Students & Devs	R⇓		R∼
Trockman [61]	Java	46	8	30-70†	Students & Devs	R⇓		R∼
Beniamini [3]	C	56	1	10∗†	Students			R∼/C∼
Tenny [59]	PL/I	148	1	73∗†	Students			C⇑
Takang [57]	Modula-2	89	1	252∗†	Students			R∼/C⇑
Lee [33]	Java	N/A	120k files	46	Devs			R⇓
This Study	**C++, TypeScript, Java, Rust, Python**	**N/A**	**604k**	**3-200+**	**Devs**	**C⇓**	**C⇓**	**C⇑**

affected readability but not comprehension [12]. Sampaio et al. replicated this study to find that comments positively affected comprehension but not readability [51]. Nielebock, et al. also partially replicated the original study and found that participants perceived comments to be helpful even if they did not make any difference on their actual readability ratings nor comprehension [38].

2.1.4 Metrics are also inconsistent in practice. Some studies have applied metric-based readability models in practice. Fakhoury et al. scraped commits where developers explicitly stated that they were improving readability, but found that the four prior readability models [10, 19, 44, 53] did not capture this sentiment in the changes that were actually included in the commit [22]. Pantiuchina et al. found a similar conclusion, noting that code quality metrics were unable to capture the quality improvements expressed by the developers [40].

2.1.5 Summary. Many studies have investigated Volume, Complexity, and Expressivity-measuring code metrics' impact on code comprehension. Although one might expect studies with similar methodologies or using the same dataset to produce similar results, these studies do not. Some studies find that some of the metrics are useful while others do not. Existing metrics do not appear to apply to real developers working on real code.

2.2 Implications of Study Design

The precise design decisions and controls in controlled experiments often dictate their results [29]. With any study that employs statistics to draw conlusions from observational data, it is important to state the premise of the study and justify design decisions and controls. For example, study designers may ask themselves questions like: *'Who is in the study?' 'What kind of code is in the study?' 'Are our measurements accurate proxies for what we want to measure?' 'Is our study large enough to draw meaningful and general insights?'* When examining prior literature, we found that few studies discussed or justified the implications of these premises. In this section, we discuss why these design decisions may explain some of their inconsistent findings.

2.2.1 Threats to Construct Validity. Code comprehension studies are fundamentally impacted by the difficulty of measurement: what metric should a study use that actually measures comprehension? In a survey of recent code comprehension studies, Oliveira, et al. found that 37% of studies only asked participants to use a single cognitive skill and 17% measured only personal opinion (perceived readability) [39]. Using perceived readability (e.g., by asking a participant for their subjective opinion on a code snippet's readability), as done in many studies [10, 12, 19, 38, 46, 52, 53], does not directly measure and has been shown to be inconsistent with a participant's actual comprehension of the code [12, 38, 57].

Using the time taken until a participant states they have comprehended the code is used in many studies [12, 23, 38, 42, 52, 53], but some of these studies find that time depends more on the participant than the code: professional developers took longer to comprehend the code than students, but also performed better [52, 53]. Other studies also find that time does not correlate with correctness nor performance [23, 38, 42].

Using tests, such as multiple choice tests [42, 52, 53, 57, 59] or Cloze tests [12, 51]—where subjects study a piece of code, some lines of code are removed, and then they are asked to fill in the missing lines—does not reflect how developers comprehend code

in practice. Developers do not comprehend code to pass an assessment; they comprehend code to complete specific tasks like a code review, a bug fix, or to implement a new feature. While some studies *do* have subjects complete realistic tasks like bug fixes or new features [3, 21, 23, 38], it is difficult to precisely identify which part of the task is related to comprehension.
Summary: Measuring comprehension is difficult even in a controlled setting. Existing comprehension measurement tasks are different than comprehension in practice and thus form an imperfect measure of true comprehension.

2.2.2 Threats to Internal Validity. It is difficult to design a study that isolates the measurement of a human's comprehension of the code *and* controls feature(s) of the code to analyze. For example, trying to isolate the use of code comments in comprehension studies on decomposition has proven to be surprisingly difficult. Nielebock et al. provide two versions of code in their study: one with comments and intentionally obfuscated identifier names, and one with high quality identifier names only [38]. The obfuscation was done to avoid an imbalance of information in the versions, but may compromise its validity since neither version reflects how developers write code in practice. Similarly, Tempero, et al. provide their subjects two versions of the same code: one that is decomposed into well-named helper methods, and one that contains comments describing the blocks of code that had been extracted into helper methods in the first version [58]. This was also done to avoid an imbalance of information, but may also help explain why they found negative results on the impact of decomposition (the comments and the method names served the same semantic purpose to explain the code).

Additionally, code is complex and feature rich. Code features may confound each other and make it difficult to isolate which features are truly influencing comprehension and may vary heavily depending on the sample of code used. For example, Buse, et al. extract 27 features from the code snippets under study even though each snippet was between 4 and 11 LOC [10]. This may help explain why their initial results differ from those in subsequent studies depending on the statistical technique used to analyze the dataset [36, 44]. This phenomenon reappears in the Scalabrino, et al. studies—which extract 121 unique features [52, 53]. Trockman, et al. re-examine these same features using multiple component analysis and find different results [61].

Finally, studies suffer from demographic variability. Most studies exclusively [3, 4, 10, 12, 13, 21, 23, 57, 59], primarily [46, 51–53], or partially [19, 38] use students as subjects. However, several comprehension studies have concluded that students and developers comprehend code differently [3, 4, 38, 50, 61] or that a subject's experience level was more impactful on comprehension results than any code feature [52, 53].
Summary: It is difficult to construct a controlled experiment that isolates the impact of specific code features on comprehension. Using students as subjects threatens the applicability of prior results.

2.2.3 Threats to External Validity. Studies may lack generality due to study design (**Study Scope** columns in Table 1).
Language Uniformity: Most recent studies use only Java code [10, 12, 33, 38, 42, 51–53]. It is not clear whether or how any of the findings apply to other programming languages.
Small Scale: Studies show each participant a limited number of code samples (no more than 16) due to time constraints. This limits what studies can glean about each participant's view of comprehension since they will not interact with many code variations. Studies also control for *Length*, the LOC of the code examined by participants. Some studies offer larger code (50+ LOC) but others examine small code only (≤ 12 LOC). Controlling for *Length* means that it cannot be properly studied as a factor for code comprehension.
Synthetic Code: Some studies use manually constructed or altered code instead of real, developer-written code [42, 46, 57, 59]. Crucially, even with open source code, subjects lack the context a developer on the project would have while comprehending the code. Most studies have subjects evaluate a method's comprehensibility without putting it in the context of its project or a real task. These deviate from how software engineers comprehend code in practice.

When examining scale, it is important to justify how the sample size impacts results. It is entirely possible that small sample sizes preclude studies from being able to find meaningful correlations at all. For example, study designers may consider conducting a *power analysis* to ensure that their sample size is large enough to result in a powerful enough correlation between code metrics and comprehension measures that has sufficient effect size (e.g. to say that line width has a medium impact on comprehension) [30].
Summary: While prior studies presumably aim to derive insights into the comprehensibility of real code, they often have a narrow focus on synthetic code. Prior studies do not analyze if their code samples are large enough or representative of real code.

2.2.4 Overall Summary. Designing a controlled study to measure code comprehension is difficult. The strength of prior studies is their attempts to directly measure comprehension by having subjects comprehend code in a controlled setting. However, these studies also present tradeoffs in exchange for this control: they often use code that is unrepresentative of real code, subjects who are unfamiliar with the code and are often students, and do not assign comprehension within the context of performing realistic tasks.

3 APPROACH

Our goal is not to disregard the results or methods of prior studies. Instead, we view them as a collection of partial windows into the community's understanding of code comprehension that we can complement with an alternate viewpoint. To explore how prior controls and design decisions might have affected prior results, we invert the methodological tradeoffs of prior work in our study. First, we address *authenticity* by analyzing real code in production software, produced by professional developers in actual development tasks. This aligns with our target demographic of professional developers comprehending code while completing a concrete task. Second, we investigate many (604k) code samples, including code with many lines of code. This gives our study enough *scale* to derive meaningful and generalizable insights. In exchange for these tradeoffs, we cede control of our subjects making comprehension judgments in a controlled environment. This means we cannot directly observe nor attempt to measure the degree of comprehensibility of the code under study (although as noted previously, existing measurements are also imperfect).

A key question under this methodology is: **How can a code comprehension study measure code comprehension without explicitly asking humans to comprehend code?** To assert that the developers working on the open source software that we examine are in fact comprehending the code, we rely on two primary rationale: first, the code in these projects must be comprehensible enough for developers to successfully evolve—these projects are being actively maintained and deployed. Second, each code change must undergo code review. Modern open source teams practice code review, in which a peer must review code before it is accepted into the code base [47]. Teams and projects have specific code review guidelines to assist both authors and reviewers align on what type of code should be accepted. For example, developers value functionality and comprehensibility when examining code [2, 6, 7]. Underlying these guidelines is an unspoken one: *the reviewer must be able to comprehend the code under review—how else would they be able to properly review it?* We validate this premise—that humans *are* performing comprehension-judgments even if they are outside of our supervision—in Section 5.

3.1 Research Questions

Within this empirical approach, we investigate the following:

RQ1: Why do prior code comprehension metrics work for some studies but not others?
For this research question, we investigate how metrics vary within methods of similar *Length* (as is the case in those studies) and reason about how sample size may impact results given this variance.

RQ2: Do prior code comprehension metrics generalize?
Prior studies examined metrics only on small code—methods with few LOC. In this question, we evaluate if these metrics apply to more representative code, which is often much larger.

RQ3: How does code expressivity relate to comprehension?
The prior two research questions focus on the structure of code: how *much* code is written and how *complex* it is. How code is written also affects its comprehensibility. Prior work measuring expressivity through comments has yielded inconsistent findings; in this question, we reconcile these inconsistencies and provide clearer insights into the role of comments for code comprehension.

4 METHODOLOGY

Our approach presents several tradeoffs with respect to traditional, code-first code comprehension user studies. These existing methodologies are imperfect—and our approach is too, but in different ways that impact the kind of claims we can make. Following guidance from Robillard et al. [48] we present our experimental design decisions and implications here instead of as a separate threats-to-validity section. The primary implications of our selection of an empirical study are that:

- It **increases generality** since the analysis is conducted on a larger scale, language, and project breadth.
- It **mitigates ecological constraints during comprehension** since it examines the code developers actually read, write, and approve in authentic environments. Because developers are actioning upon their code, we argue this represents true comprehension.
- It **cannot directly examine, query, nor measure the nuances of individual developers’ code comprehension**, since we do not use explicit human judgments. As noted by prior studies, each developer comprehends code differently [52, 53]. We did not analyze differences between developers, and chose to examine the differences in code instead. Qualitative studies would excel at investigating these subjective differences.

4.1 Language Selection

We examined a mix of popular and emerging programming languages to ensure a balance of maturity and relevance. TypeScript, Python, Java, and C++ are all among the most popular languages on GitHub. Rust, with its security features, is one of the fastest growing languages. These languages are imperative and support object-oriented programming.
Implication: Our findings span more languages than prior studies but may not generalize to languages that differ in fundamental ways (e.g., functional languages).

4.2 Project Selection

We chose the top starred projects on GitHub that were at least two years old and under active development and extracted the most-used language in each. We chose this method to focus on projects that were mature, large, and actively developed. We selected the first ten projects in each language that had enough code in that language, to give our analysis in each language and as a whole a sufficiently large scale. The projects ranged across many domains including databases, computer vision, and web frameworks. A full list of projects is in our supplemental materials.[2]
Implication: Our findings may not generalize to projects that are smaller, less mature, or lack rigorous code review.

4.3 Method Selection

Several related mining/comprehensibility research analyzed code at the method-level (instead of file, class, or line-level) [15, 16]. This also reflects how developers conduct code review (method-by-method). We mined 604k total methods and function declarations from the selected projects using a custom AST parser. Many languages allow both methods and functions which we will refer to collectively as “methods". In order to mitigate the influence of outlier methods, we excluded excessively long methods (the top 1% in length in each project) and short methods (≤ 3 LOC).
Implication: Nuanced findings from the excluded short or long methods are not captured in this study.

4.4 Code Metric Extraction

We use only metrics that were also used in prior comprehension studies. For each category of metric, we justify which metrics we chose in the following subsections.

4.4.1 Volume. These metrics measure the amount of code. Intuitively, more code means more reading and harder comprehension.
***Length*:** The number of lines of code in a method, excluding blank lines and lines that are entirely comments. *Length* is the most

[2] https://github.com/kdchin/empirical-comprehension

widely-used and easy to calculate metric in software engineering. Practitioners have used *Length* as a measure for code quality since it has been shown to correlate with underlying quality measures (like bug-proneness and maintainability) that are much more difficult to directly compute [15, 16]. *Length* correlates strongly with volume and complexity metrics like *McCabe*, *FanOut*, and *Halstead* [1, 17, 27, 28], leading some to conclude *Length* is the only code quality metric [20, 25, 55, 56]. However, as noted in Section 2.2.3, studies incidentally control for *Length* by choosing code of within certain LOC bounds which may affect results.
***Identifiers*:** The number of variable, method, class, or other names used or referenced. Prior studies have investigated *Identifiers*' impact on comprehensibility with mixed results: some say it negatively affects it [10, 44] and others say it has little effect [19, 36, 52, 53, 61].
***Characters*:** The total number of characters in a method body, starting from the first non-whitespace character. Prior studies have generally found that *Characters* negatively impacts comprehension [10, 19, 44, 61], though many studies have also found it to have little impact [36, 52, 53].

Both *Identifiers* and *Characters* were treated by prior studies as averages or maxes based on the *Length* (i.e. average identifiers/characters per line of code). In our *Length*-unconstrained environment, per-line averages make less sense, since that would imply that a method with 10 lines of code and 100 lines of code are similarly comprehensible so long as they have the same amount of characters per line. We examine both averages and raw counts for *Identifiers* and *Characters* as a part of **RQ1** and **RQ2**.

4.4.2 Complexity. These measure the degree of complication. Intuitively, more complex code should be harder to comprehend.
***McCabe*:** The number of linearly independent paths through the code (`1 + #predicates`). *McCabe* correlates with real defects and maintainability issues (and also with *Length*) [20, 25, 55, 56].
***FanOut*:** The number of calls to other classes and methods. *FanOut* also correlates with actual defects but has not been studied in depth in code comprehension studies [12, 15].
***Halstead*:** The amount of operators ($op1$) and operands used in a method ($op2$) used: $N_{op1} + N_{op2} * log_2(\text{unique } (N_{op1} + N_{op1}))$. Halstead volume has been shown to negatively impact comprehension but also has not been widely studied [12, 44].

4.4.3 Expressivity. These metrics measure how expressive or detailed the code is. They are more stylistic and subjective than Volume and Complexity metrics and capture *how* the developer wrote the code (not just *what* they wrote).
Identifier Naming: How the developer names their identifiers (variables, method names, etc). Prior studies find mixed results on the quality of identifier naming with respect to comprehensibility [3, 4, 32]. We chose **not** to evaluate *Identifier Naming* as a metric because it is inherently too subjective to be consistently useful in a multi-project, multi-language empirical study. For example, heuristic models like *Identifier Length* may not generalize because longer is better in some cases but shorter is better in others [3].
***CommentPresence*:** Counting only inline and block comments, a method may either have no comments (NC), one single comment (SC), or multiple comments (MC). Source code comments are important to developers [39], especially in code review [6]. Code comments do not comprise the executable part of a system; they exist primarily to help developers manage complexity [8] or communicate design rationale or how it should be used [14, 62]. By adding comments to their code, developers are signaling that their code requires additional context that is *necessary* for effective code comprehension; in this way, comments can be thought of as a proxy signal for code that requires additional detail to be comprehended.
***Entropy*:** The degree of disorder in the distribution of tokens used. *Entropy* captures textual variance and has been shown to negatively impact comprehension, but has not been widely studied [33, 44].

5 VALIDATION

In this section we investigate if our premises hold in the projects under study. First, we validate that developers are comprehending iterating on their code in the projects under study. This accounts for our lack of control of their exact environments in which they are comprehending code. Next, because source code comments are so important to developers, we specifically focus our evaluation of expressivity features on *CommentPresence*. In a large-scale empirical study, this means it is essential to evaluate our design decisions in our definition of *CommentPresence* to ensure it is well defined with respect to our intent. Many artifacts from this validation do not fit in this paper but are included in the supplemental materials.

5.1 Comprehend, Collaborate & Iterate

We argue that developers in mature software projects are continually comprehending code through their code review processes. This is validated by their ability to be maintained and repeatedly updated by many collaborators. While this assumption may not hold for every single line of code, prior work has supported its general validity: analyzing the history of 25 open source software systems, Piantadosi, et al. find that only a small amount of code begins unreadable and stays unreadable—the vast majority of code is created or becomes readable over time [43]. This is supported by Lehman's Law, part of which states that successful software is definitionaly amenable to evolution [34]. Production software systems are subject to survivor bias as well—the code that we see is most often the most readable or highest-quality code—compared to the drafted or previous versions.

5.1.1 Project Processes & Activity. To validate that each project under study is a mature project that uses collaborative processes, we manually examined their code contribution guidelines, static analysis rules, and project metadata. Two authors examined the code contribution guidelines of each project and checked each for a section giving guidance on the following: (1) team practices/code review before acceptance (2) code formatting/style; (3) testing; (4) adding documentation. Projects frequently contained all four guidelines, with each project containing at least one. We also examined each project's static analysis rules for at least one rule relating to code formatting (which existed in all-but-two projects). Almost all projects had a rule limiting the maximum line width. This affects the correlation between *Length* and *Characters* of our dataset, since it forces additional characters to appear in a new line. Despite this, we decided not to apply consistent formatting because we wanted to preserve authenticity by analyzing the code in the form its developers and reviewers comprehended it. Apart from this, few projects had rules that would directly affect our reported metrics (`core`

restricted *McCabe* to 25 per method; hadoop restricted *Length* to 150 LOC per method, and a few others). Finally, to validate the maintainability and collaboration of each project, we retrieved the number of commits and number of human pull request review comments in the two years prior using Github Archive [26]. The median project (grpc), is professionally managed by Google and contained 7.1k commits (almost 300 per month) by 229 unique authors. Collaborators of the grpc project produced 13.7k pull request review comments from 381 unique reviewers. More mature projects such as cassandra naturally exhibit less development (547 commits from 40 authors and 1.4k review comments from 69 reviewers) while surging projects such as rust have been iterated on much more (44.6k commits from 1.7k authors and 261k review comments from 1.9k reviewers). These indicate that the projects in this study are actively worked on by many authors and reviewers, who must come to a shared agreement on what acceptable code means.

5.2 *CommentPresence* Design Decisions

In this section we validate three design decisions we made in defining *CommentPresence* as a metric.

Table 2: Comment usage in each language and relationship with mean *Length* per method. Methods with one or multiple comments are likely to be longer than those with no comments. Methods with only one docstring comment are similarly long to those with no comments. Java methods are shorter and less frequently commented.

Lang	%CM	NC *Len.*	Doc. *Len.*	OC *Len.*	MC *Len.*
C++	32%	10.8	11.9	18.6	38.0
TypeScript	29%	12.5	11.7	17.5	33.3
Python	29%	12.1	9.4	18.3	34.4
Rust	24%	14.1	13.1	23.2	40.7
Java	17%	9.5	9.9	17.2	27.2

5.2.1 All comments are semantically meaningful. **Rationale:** "Junk comments" such as commented-out code and TODOs, make up a small portion (1-4%) of total comments [41, 60]. State-of-the-art quality measures for comments are highly subjective. Textual or lexical features of comments (such as identifier length or vocabulary choice) have mixed findings in prior code comprehension studies (e.g., [4, 11, 31, 32, 54]), and may be more appropriate to study in a closer study that can capture the nuances of individuals' perspectives (see [45] for an index of studies that do).
Validation: To validate this decision, two authors independently examined a random sample of 500 comments and their surrounding code (100 from each language). For each comment, an examiner decided if the comment could have been intended to be helpful for comprehension or if it was "junk" (e.g. a TODO or commented-out code). The authors then reviewed their independent decisions together, and agreed that 12 out of the 500 (2.4%) were "junk comments" which is within the 1-4% range of prior work [41, 60].
Implication: We assume that all comments in our dataset *could* have been intended to be helpful for comprehension in some way.

5.2.2 No, one, or multiple comments is a useful granularity. **Rationale:** More detailed metrics like characters per comment, lines of comment code, scalar number of comments may be too granular to be useful to developers. More verbose comments, more total comments, and more lines of comment code are not necessarily desirable since they may actually hinder comprehension through excessive reading (as argued by Agile practitioners [18]). We count consecutive lines of comment code as one comment regardless of syntax because they likely represent the same sentiment.
Validation: We evaluated two variations of comment metrics: unconstrained comment count, and comment lines of code (which were used in prior studies) [10, 12, 19, 33, 36, 38, 44, 46, 51, 53, 57, 59, 61]. For each metric, we calculated their association with the same suite of comprehension metrics as we did *CommentPresence*. We then measured their correlation coefficients and visualized their relationships. Both show that their propensity to grow together at scale is limited (Pearson's r^2 is low (~0.2), and the data do not appear linear). Since this aligns with the practical intuition that (unconstrained) comment counts or lines does not necessarily aid comprehension, we proceed with the more simple, constrained metric (*CommentPresence*) which delineates just three categories: methods with no comments (NC), single comment (SC), or multiple comments (MC). As shown in Table 2 and Figure 2, even this granularity provides strong discriminatory power, with metrics drastically increasing with more comments.
Implication: We do not provide insight into how the content nor size of comments affects comprehension. We present *CommentPresence* as a comprehensibility metric that is easy to compute, natural to developers, but also limited in detail.

5.2.3 Docstrings are not comments. **Rationale:** Docstrings often convey different information than typical in-line source code comments (e.g., API or usage information).
Validation: At least according to *Length*, *methods with docstrings are more similar to methods without comments than those with them.* The average *Length* of no comment (NC) methods (9.5 LOC Java; 14.1 LOC Rust) are very similar to those with only a docstring and no other comment (9.9 LOC Java; 13.1 LOC Rust) (Table 2). However, when considering docstrings as comments, usage numbers would differ drastically: Python would increase from 29% to 54% commented methods (Python docstrings often contain type information) while Java would rise from 17% to 38%. Analyzing the differences of docstrings and comments on comprehension is interesting future work but out of scope for this study.
Implication: We presume that docstrings and comments are circumstantially different. Projects that use docstrings and comments similarly may find different results.

6 EVALUATION

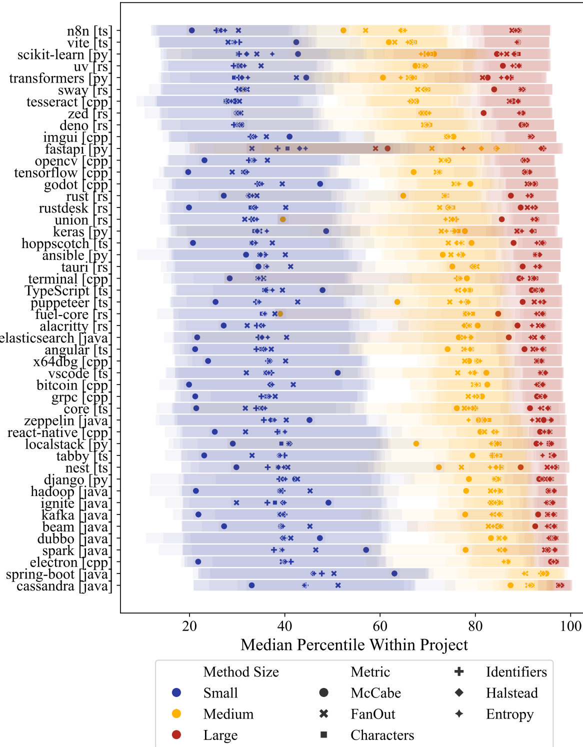
In this section we empirically evaluate our research questions. See our supplemental materials for full figures and datasets.

6.1 *Length*-local Variations Exist in Real Code

To investigate why code comprehension metrics have inconsistent results in prior work, we investigated their varying values in methods with similar *Lengths.* Figure 1 plots the medians of each of these metrics in three *Length*-groupings of methods in each project: Small (≤ 15 LOC), Medium (16-38 LOC), and Large (39+) LOC methods.[3] For example, the blue dot in the n8n row represents the median

[3] These limits were chosen to capture equal thirds of the total LOC in our dataset.

Figure 1: Measuring comprehension metrics against Small (≤ 15 LOC), Medium (16-38 LOC), and Large (39+ LOC) methods. The bars represent the inter-quartile range for each metric and show their *Length*-local variation. This may explain why they are valuable in comprehension studies with fixed LOC. The close grouping of each metric within method size groupings shows that they are highly correlated with *Length*.



McCabe value for methods under 16 LOC in the `n8n` project, and its blue shaded region captures its inter-quartile range (*McCabe* values between the 25th and 75th percentile of Small methods). This plot captures three key observations.

First, most methods in real open source projects are Small or Medium, measured by any of the metrics. This is represented by the blue and yellow shaded bars having a large area, since they account for a higher percentage of each project's total methods. Second, examining the project and language differences, the projects lowest on the Y-axis are primarily Java projects. Most prior studies investigated small-to-medium Java code.

Finally, across all projects, the comprehension metric values display moderate variation within each *Length* groupings (the shaded IQR bars). This is especially present in the Small and Medium method groups. In other words, controlling *Length* to similarly-lengthed methods still results in meaningful *Length-local* differences in comprehension metric values. For example, Small methods in the IQR of `tensorflow` range from 10 to 29 *Identifiers*. Examining these metrics as they were used in prior studies (as averages), the Small method IQR range results in a range of 2.0 to 3.3 *Identifiers* and 31 to 47 *Characters* per line.

We then calculated the standard deviation of each metric, which measures the degree of spread a metric has from its mean (colloquially, its variance). In Java, a mean of 235 (Small methods)/450 (All methods) *Characters* per method with a standard deviation of 171 (Small)/537 (All) indicates a high standard deviation. This pattern repeats itself for Small methods in other languages and shows that methods in real code vary greatly according to metrics.

We also calculated the standard error to estimate how much the mean of each metric is likely vary from sample to sample. Unsurprisingly, given how large the sample sizes are, Java methods in our dataset had low standard errors of 0.4 (Small methods) and 1.1 (all methods). However, using our calculated standard deviation on the maximum code sample size in prior study (16), the standard error jumps to 43 and 134, respectively. In other words, taking different samples of 16 methods may result in large differences in the average amount of *Characters* in the sample. This pattern is similar across other metrics and languages and highlights that *code metrics used in prior comprehension studies are likely to vary from sample to sample, especially when using small samples.*

RQ1: Why are prior studies inconsistent?

Summary: Examining real code that is similar in length to those used in studies, prior comprehension metrics vary in code with similar *Length*. *Length*-local variations like these may be an additional source of sample variability and may contribute to inconsistent prior results.

6.2 *Length* Correlates With Other Comprehension Metrics

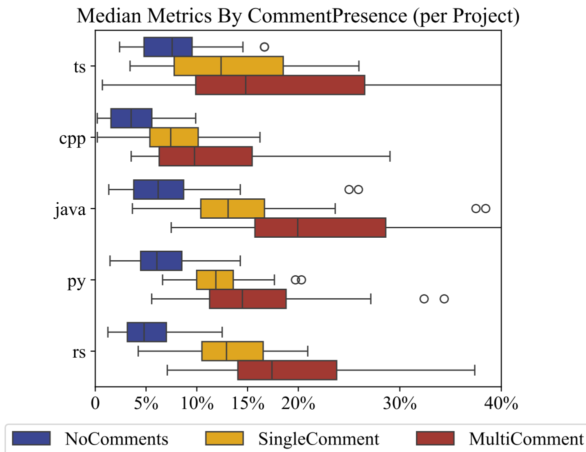
To understand how prior comprehensibility metrics relate to *Length*, we again examine Figure 1. When we zoom out and look at the differences between the groupings, we see that the median comprehension metric values are very different. Each metric is much larger when moving up a grouping. In other words, the *Length-non-local* differences in comprehension metric values is large, meaning *Length* likely also impacts other comprehension metrics (and comprehension as a whole). For example, Small methods in `tensorflow` have a median of 18 *Identifiers*, compared against 74 (Medium) and 94 (Large) methods. This visual intuition is supported by the Pearson r^2 correlation coefficients, which indicate that *Length* correlates strongly with *Characters* (0.91), *Identifiers* (0.86), *Halstead* (0.83), and also with *FanOut* (0.72), *McCabe* (0.71) and *Entropy* (0.58), all with minimal standard error (< 0.002). This demonstrates that in a large, *Length*-unbounded dataset, *Length* strongly correlates with existing comprehension metrics. This is also backed up by the standard deviations, where Small Java methods have a standard deviation of 171 *Characters* while all Java methods, 537. In other words, metric values vary more widely when they are not constrained by *Length*.

Finally, examining per-line *average* metric values as used in prior studies, these metrics provide much less discriminatory power. For example, in `tensorflow`, the median value in each group for average *Identifiers* is 2.6 (Small), 2.8 (Medium) and 2.9 (Large). Average *Characters* varies from 39 (Small), 44 (Medium) and 47 (Large).

RQ2: How does *Length* interact with other metrics?

Summary: Prior metrics also have strong *Length-non*-local impact—they scale linearly with *Length*—indicating that *Length* may impact comprehension. Average metrics (per LOC) do not discriminate as well on larger code.

Figure 2: Comprehension metrics between NC, SC, and MC groups across languages. Each language/comment-group boxplot is generated from the metric × project medians, normalized by the maximum value. For example, the NC Java boxplot consists of 6 metrics × 10 projects = 60 data points. NC methods are relatively low complexity while SC and MC methods are several times larger.



6.3 Comments Co-locate With Complexity

We evaluate how *CommentPresence* relates to other measures of code comprehension. In, Figure 2, we divided the dataset into methods with no comments (NC), methods with one comment (SC), and methods with multiple comments (MC) and calculated the median values for each comprehension metric within each group and project. This plot demonstrates that, generally speaking, methods with comments are much more complex than those without comments, since the blue NC boxes are lower than the yellow SC and red MC boxes. This effect is especially noticeable in Java and Rust, with SC methods being almost twice as large and complex as NC methods. For example, the median *Length*/*Identifiers*/*Halstead* of a NC method in `vscode` is 8/15/94, while that of a OC method is 13/27/212. We also see that methods with multiple comments (MC) are generally even more complex (23/53/480 in `vscode`). We confirmed this relationship between the three levels of *CommentPresence* using two statistical tests:

1. **One-Way ANOVA**, which is the multi-group version of a t-test and measures the Type-1 error (concluding the groups differ when they do not).
2. **Cliff's Delta**, which measures the effect size, or how often values in one distribution are larger than those in another.

We conducted an ANOVA test for each language, measuring the effect of *CommentPresence* on each metric. Each ANOVA test resulted in a p-value of 0.00.[4] This is due to the mean metric values being vastly different for each group, and indicate that *CommentPresence* has a statistically significant effect on each metric. We computed Cliff's Delta between NC and SC, and SC and MC groups. This resulted in a median effect size of 0.36 and 0.45, respectively, indicating medium-to-large effect which backs up the approximate size differences portrayed in Figure 2 and Table 2.

A natural follow-up question is if these distinct differences can be explained because commented methods tend to be much larger in *Length* (and hence larger in the other metrics as well, as demonstrated in **RQ2**). In Figure 3 we calculate each metric across all projects in a language with a fixed *Length*, meaning that each comparison contains the computed metrics for all methods at that exact *Length* only. For example, a dot for 15 LOC compares all 15 LOC commented methods with non-commented methods in a metric (as noted in Section 4 LOC does not count comment lines). Still, almost universally for each *Length* and metric, Cliff's Delta values are typically small-to-medium toward commented-methods being more complex. Python exhibits the largest effect sizes, with some bordering on large. Even within the same amount of lines of code, commented methods tend to be larger and more complex. *In other words, commented methods having a longer Length does not alone explain the increase in other metric values, and this pattern generalizes by language and by project.*

RQ3: How is expressivity related to comprehension?

Summary: Methods with comments, especially those with multiple comments, are larger and more complex than methods without comments. This observation holds even when accounting for methods' *Lengths*.

7 DISCUSSION

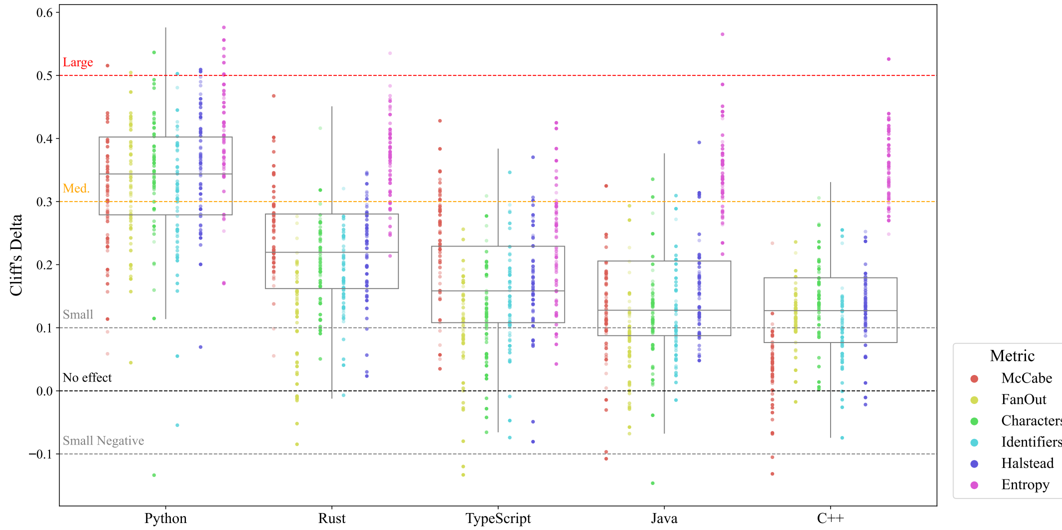
Fundamentally, we sought to understand the influence that design decisions and controls may have had on the ecological validity and results of prior code comprehension studies. Prior studies were able to directly observe their subjects' comprehension of code, but had to compromise on other aspects, for example by using students, examining small code, and/or assigning unrealistic tasks. In our empirical methodology, we no longer directly observed our "subjects"' comprehension of code but this allowed us to examine a large amount of real code that professional developers collaboratively wrote and evolved. We believe that this preserves the ecological validity of the code under study. This allows us to derive meaningful insights into the general population of code relevant to code comprehension research and practice at the expense of not being able to investigate the subjective, developer-specific nature of code comprehension. In this section, we discuss the implications of our findings.

7.1 Sample Variability Affects Results

In **RQ1**, we sought to understand why prior studies found inconsistent results in features of code that we would intuitively think to *always* impact comprehension. Higher values in volume and complexity metrics (e.g., more code, more complex code) should negatively impact comprehensibility but have not always been found to. In Table 1 we noted that studies usually examined at most 16 different code snippets per participant. We posit that this is not a large enough sample size to consistently represent of the population of code that exists in practice. Our evaluation showed that when looking at population of code that is similar to those used in studies, volume and complexity metrics vary greatly and exhibit high standard error even in our large dataset. This implies that samples of code vary greatly in nature and may affect results. The implication for study designers is that it is crucial to calculate,

[4] Rounded to four decimals.

Figure 3: Cliff's Delta between NC methods and commented methods for each metric and across languages when *Length* is fixed from 3-60 lines of code. Each dot indicates a Cliff's Delta value for a fixed line of code across all methods in that language. A darker dot indicates a higher count for lines of code. Metrics on the legend are left-to-right on each box plot. Even when fixing for *Length*, *CommentMethods* demonstrate a small-to-medium increase in metrics in all languages and metrics.



report, and discuss the effect of sample size and metric variations on results, especially since it is not always realistic to simply increase the amount of code variation and/or subjects. Standard error is inversely proportional to $\sqrt{samplesize}$ (meaning 9x increase in sample size is needed to achieve 1/3 of the standard error). Studies may benefit from performing *a priori* power analysis to determine how feasible it is to even achieve results given the sample size [9, 30]. For example, a sample size of 16 may not even be sufficient to show a strong enough correlation (e.g., effect size of 0.5) between a code metric and a comprehension measure and within a confidence interval (e.g., 0.2). This may be one reason that prior studies found inconsistent results: they might not even have been able to find statistically significant and noticeably large effects because they were too small to begin with.

This raises the meta question about the overall goals of comprehension studies given this context variability: are we seeking to build a consensus on comprehensibility ratings? Given the methodology of prior studies, the answer appears to be yes—most studies have multiple subjects analyze multiple different pieces of code and try to derive objective, general results. However, prior research has shown that subjects do not always consistently assess comprehensibility when compared against other subjects or even with themselves, even when they have a rubric [35].

Instead, given the difficulty of running a user study that is large enough to bring statistically significant and general results, it might be more fruitful to pivot the goals of the study. Instead of trying to build a consensus in these studies, it may be more valuable to analyze the *intra*-participant-code differences (instead of the *inter*-participant-code ones)—by investigating how each participant comprehended each code snippet differently, each in their own, subjective way. The nature of code is highly influenced by the context in which it is produced and by the personal, subjective qualities of the human comprehending it. Smaller scale qualitative studies excel at exploring each individuals process of comprehension, and may bring a deeper understanding of comprehension even if they cannot necessarily be generalized.

Implication: Practitioners of smaller scale user studies should consider and state how strong, precise, and general they seek for their results, and if their sample size and standard deviation are large enough to find it. Qualitative, human-first comprehension studies are valuable in their ability to deeply explore the inherently subjective interpretations of code. A code-first comprehension study like ours presented here cannot capture the subjective nuances of comprehension.

7.2 Context Matters

Meaningful differences exist in code between projects and languages. Figure 3 shows that, particularly in Python and less so in C++, commented methods tend to be more complex than their non-commented counterparts of the same *Length*. Additionally, the distributions between the top and bottom rows in Figure 1 (`n8n` and `cassandra`) and the overall variation row-to-row show that *Length* and other metric distributions vary greatly among projects. Particularly notable was `spring-boot`, in which *every method was 27 lines of code or fewer* despite having no static analysis rules that would automatically enforce this (`spring-boot` also has an anomalous rows in Figure 1). We speculate that strong and strict code review dictate this pattern. Given these language variations, project and domain differences, the natural variations in code (**RQ1**), and variations in how individual developers comprehend the same code fragment, we believe that it is near-impossible to produce a single truly generalizable result from a small-scale, controlled user study. We believe that this may be a contributing cause to the inconsistencies in prior work, and emphasize the importance of future studies to analyze the impact of these differences (especially those by developers) on comprehension.

Additionally, in **RQ2** we examined the impact of using per-line averaged metrics (namely in *Characters* and *Identifiers*). We showed that across non-local LOC bounds (from Small to Medium or Medium to Large methods), these metrics exhibit large variations. In other words, an averaged metric value has significance primarily

when examining local methods with similar *Lengths*. This may not hold when when comparing methods with larger differences in LOC.

Implication: Studies may have more earnest results by analyzing the baseline metric values within their code samples to understand if those generalize to the greater population of code or to isolate specific metrics that may be more salient within the sample.

7.3 Comments Also Matter

Despite their only purpose being to provide additional context to code that cannot be expressed in the code itself, comments have not been consistently found to improve comprehensibility [12, 19, 53, 61]. Our results provide an explanation for why: throughout each language and project, comments appear in more complex and harder to read code (even when we control for *Length*).[5] In other words, the natural complexity of the code may counteract any comprehensibility benefits that comments provide. This follows the natural intuition of developers that comments aid comprehension because they give additional context or rationale that could not be expressed in the code itself, and would appear more in code where developers and reviewers felt the extra detail was necessary—longer, more complex or harder to understand code. Revisiting Figure 3, we can see this effect visually: the additional volume and/or complexity of code that may negatively affect comprehensibility (i.e. the effect size) is counteracted by comments. This implies that comprehensibility cannot be captured by any one metric.

Implication: If we hope to use metrics to effectively measure comprehensibility (as something that is impossible to directly quantify), we must do so using a combination of metrics. Our definition of *CommentPresence* may be useful in this regard.

7.4 Helping Developers in Code Review

Our insights on the value of *Length* and comments on code comprehension are valuable because they give the first consistent, large-scale research-backed credence to intuitions that all software developers have. We already knew that developers reported valuing these metrics in comprehension and code review [2, 6, 7]—now we know that these features actually, empirically, make a difference. So, what should developers take away from this insight?

Modern software projects often use static analysis tools to prevent bugs or enforce consistent formatting [47]. We argue that projects could benefit from adding static analysis rules to manage the comprehensible complexity of methods, much as many already do for line width (e.g., `angular`, `scikit-learn`) and/or cyclomatic complexity (e.g., `core`). Just one of the projects under study had a static analysis rule for number of lines of code per method (`hadoop`: $Length \leq 150$), and popular static analysis tools in Python, C++, and Rust *do not even support this as an option* despite popular style guides (e.g., Google's C++ Guide) explicitly recommending writing short methods. *Length* and *CommentPresence* are easy to compute and, importantly, easy for developers to amend, meaning they lend well to automated static analysis. For example, inducing developers to either write a comment or reduce the LOC in a method to satisfy a static rule benefits comprehension by resulting in either a new comment or decomposition, which also improves maintainability [16]. These are typical parts of code review that can (and we argue, should) be semi-automated, allowing developers and code-writing AI agents to focus on other important aspects of development.

Implication: Static analysis rules, e.g. where *Length* thresholds increase based on *CommentPresence*, may be an effective way to automatically manage *Length* and comments to aid code review.

[5] Lee et al had speculated this as the reason they found that comments negatively impacted readability [33].

8 CONCLUSION

Code comprehension studies have struggled to build a consensus around what actually affects code comprehension. In our study, we attempt to bring generality to inconsistent prior results using a *code-first* methodology that prioritizes authentic context and large scale. We analyze comprehensibility features in 604k methods across 49 professionally-managed software projects in 5 different programming languages. We demonstrate that metrics used in prior comprehension studies exhibit high variance in real code and especially with small sample sizes. This may help explain why prior studies find conflicting results. We find that *Length*—the amount of lines of code in a method—is highly correlated with previous comprehension metrics and posit that future code comprehension studies should be *Length*-aware. We then find that comments and *Length* are useful code comprehension features that warrant further exploration in human-first studies. We believe these results provide evidence-based, actionable ways for developers and researchers to improve the understanding, implementation, and study of code comprehension.

ACKNOWLEDGMENTS

To Ronald Garcia and Caroline Lemieux for their invaluable feedback on prior versions of this paper.

REFERENCES

[1] Vard Antinyan, Miroslaw Staron, Jörgen Hansson, Wilhelm Meding, Per Österström, and Anders Henriksson. 2014. Monitoring evolution of code complexity and magnitude of changes. *Acta Cybernetica* 21, 3 (Aug. 2014), 367–382. https://doi.org/10.14232/actacyb.21.3.2014.6

[2] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *International Conference on Software Engineering (ICSE)*. 712–721. https://doi.org/10.1109/ICSE.2013.6606617

[3] Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach, and Dror G. Feitelson. 2017. Meaningful Identifier Names: The Case of Single-Letter Variables. In *International Conference on Program Comprehension (ICPC)*. 45–54. https://doi.org/10.1109/ICPC.2017.18

[4] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Empirical Software Engineering (ESE)* 18, 2 (April 2013), 219–276. https://doi.org/10.1007/s10664-012-9201-4

[5] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Empirical Software Engineering (ESE)* 18, 2 (April 2013), 219–276. https://doi.org/10.1007/s10664-012-9201-4

[6] Jürgen Börstler, Kwabena E. Bennin, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, Bonnie MacKellar, Rodrigo Duran, Harald Störrle, Daniel Toll, and Jelle van Assema. 2023. Developers talking about code quality. *Empirical Software Engineering (ESE)* 28, 1 (2023), 128. https://doi.org/10.1007/s10664-023-10381-0

[7] Amiangshu Bosu, Michaela Greiler, and Christian Bird. 2015. Characteristics of Useful Code Reviews: An Empirical Study at Microsoft. In *Working Conference on Mining Software Repositories (MSR)*. 146–156. https://doi.org/10.1109/MSR.2015.21

[8] Frederick P. Brooks. 1995. *The Mythical Man-Month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[9] Mohamad A. Bujang. 2024. An elaboration on sample size determination for correlations based on effect sizes and confidence interval width: A guide for researchers. *Restorative Dentistry & Endodontics* 49, 2 (2024), e21. https://doi.org/10.5395/rde.2024.49.e21
[10] Raymond P.L. Buse and Westley R. Weimer. 2010. Learning a Metric for Code Readability. *Transactions on Software Engineering (TSE)* 36, 4 (2010), 546–558. https://doi.org/10.1109/TSE.2009.70
[11] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2009. Relating Identifier Naming Flaws and Code Quality: An Empirical Study. In *Working Conference on Reverse Engineering (WCRE)*. 31 – 35. https://doi.org/10.1109/WCRE.2009.50
[12] Jürgen Börstler and Barbara Paech. 2016. The Role of Method Chains and Comments in Software Readability and Comprehension—An Experiment. *Transactions on Software Engineering (TSE)* 42, 9 (2016), 886–898. https://doi.org/10.1109/TSE.2016.2527791
[13] Celia Chen, Reem Alfayez, Kamonphop Srisopha, Lin Shi, and Barry Boehm. 2016. Evaluating Human-Assessed Software Maintainability Metrics. In *Software Engineering and Methodology for Emerging Domains*, Lu Zhang and Chang Xu (Eds.). Springer Singapore, Singapore, 120–132. https://doi.org/10.1007/978-981-10-3482-4_9
[14] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. 2021. Why My Code Summarization Model Does Not Work: Code Comment Improvement with Category Prediction. *Transactions on Software Engineering Methodolgy (TOSEM)* 30, 2, Article 25 (Feb. 2021), 29 pages. https://doi.org/10.1145/3434280
[15] Shaiful Chowdhury, Reid Holmes, Andy Zaidman, and Rick Kazman. 2022. Revisiting the debate: Are code metrics useful for measuring maintenance effort? *Empirical Software Engineering (ESE)* 27, 6 (Nov. 2022), 31 pages. https://doi.org/10.1007/s10664-022-10193-8
[16] Shaiful Alam Chowdhury, Gias Uddin, and Reid Holmes. 2022. An empirical study on maintainable method size in Java. In *International Conference on Mining Software Repositories (MSR)*. 252–264. https://doi.org/10.1145/3524842.3527975
[17] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. 1979. Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. *Transactions on Software Engineering (TSE)* 5, 2 (March 1979), 96–104. https://doi.org/10.1109/TSE.1979.234165
[18] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. 2005. A study of the documentation essential to software maintenance. In *International Conference on Design of Communication: Documenting & Designing for Pervasive Information*. 68–75. https://doi.org/10.1145/1085313.1085331
[19] Jonathan Dorn. 2012. A General Software Readability Model. https://www.cs.virginia.edu/~weimer/students/dorn-mcs-paper.pdf. https://www.cs.virginia.edu/~weimer/students/dorn-mcs-paper.pdf Master's thesis, University of Virginia.
[20] Kalhed El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. 2001. The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *Transactions on Software Engineering (TSE)* 27, 7 (July 2001), 630–650. https://doi.org/10.1109/32.935855
[21] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2018. The effect of poor source code lexicon and readability on developers' cognitive load. In *International Conference on Program Comprehension (ICPC)*. 286–296. https://doi.org/10.1145/3196321.3196347
[22] Sarah Fakhoury, Devjeet Roy, Adnan Hassan, and Vernera Arnaoudova. 2019. Improving Source Code Readability: Theory and Practice. In *International Conference on Program Comprehension (ICPC)*. 2–12. https://doi.org/10.1109/ICPC.2019.00014
[23] Janet Feigenspan, Sven Apel, Jorg Liebig, and Christian Kastner. 2011. Exploring Software Measures to Assess Program Comprehension. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 127–136. https://doi.org/10.1109/ESEM.2011.21
[24] Thomas Fritz, Andrew Begel, Sebastian C. Müller, Serap Yigit-Elliott, and Manuela Züger. 2014. Using psycho-physiological measures to assess task difficulty in software development. In *International Conference on Software Engineering (ICSE)*. 402–413. https://doi.org/10.1145/2568225.2568266
[25] Yossi Gil and Gal Lalouche. 2017. On the correlation between size and metric validity. *Empirical Software Engineering* 22, 6 (2017), 2585–2611. https://doi.org/10.1007/s10664-017-9513-5
[26] Igor Grigorik. 2012. GH Archive: Record the public GitHub timeline, archive it, and make it easily accessible. https://www.gharchive.org/. Accessed: 2025-10-23.
[27] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. 2008. Reading Beside the Lines: Indentation as a Proxy for Complexity Metric. In *Proceedings International Conference on Program Comprehension (ICPC)*. 133–142. https://doi.org/10.1109/ICPC.2008.13
[28] D. Kafura and G.R. Reddy. 1987. The Use of Software Complexity Metrics in Software Maintenance. *Transactions on Software Engineering (TSE)* SE-13, 3 (1987), 335–343. https://doi.org/10.1109/TSE.1987.233164
[29] Amy J. Ko, Thomas D. LaToza, and Margaret M. Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (2015), 110–141. https://doi.org/10.1007/s10664-013-9279-3
[30] Daniël Lakens. 2022. 8 Sample Size Justification – Improving Your Statistical Inferences. https://lakens.github.io/statistical_inferences/08-samplesizejustification.html#planprecision.
[31] Dawn Lawrie, Henry Feild, and David Binkley. 2006. Leveraged Quality Assessment using Information Retrieval Techniques. *International Conference on Program Comprehension (ICPC)* 2006, 149– 158. https://doi.org/10.1109/ICPC.2006.34
[32] Dawn Lawrie, Henry Feild, and David Binkley. 2007. Quantifying identifier quality: An analysis of trends. *Empirical Software Engineering* 12 (07 2007), 359–388. https://doi.org/10.1007/s10664-006-9032-2
[33] Taek Lee, Jung-Been Lee, and Hoh In. 2015. Effect Analysis of Coding Convention Violations on Readability of Post-Delivered Code. *Transactions on Information and Systems (TIS)* E98D (07 2015), 1–11. https://doi.org/10.1587/transinf.2014EDP7327
[34] M.M. Lehman. 1980. Programs, life cycles, and laws of software evolution. *Proc. IEEE* 68, 9 (1980), 1060–1076. https://doi.org/10.1109/PROC.1980.11805
[35] Marcus Messer, Neilnbsp;C. C. Brown, Michael Kölling, and Miaojing Shi. 2025. How Consistent Are Humans When Grading Programming Assignments? *Transactions on Computing Education (TCE)* 25, 4, Article 49 (Sept. 2025), 37 pages. https://doi.org/10.1145/3759256
[36] Qing Mi, Mingjie Chen, Zhi Cai, and Xibin Jia. 2023. What Makes a Readable Code? A Causal Analysis Method. *Software: Practice and Experience* 53, 6 (2023), 1391–1409. https://doi.org/10.1002/spe.3192
[37] Takao Nakagawa, Yasutaka Kamei, Hidetake Uwano, Akito Monden, Kenichi Matsumoto, and Daniel M. German. 2014. Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: a controlled experiment. In *International Conference on Software Engineering (ICSE)*. 448–451. https://doi.org/10.1145/2591062.2591098
[38] Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. 2019. Commenting source code: is it worth it for small programming tasks? *Empirical Software Engineering (ESE)* 24, 3 (June 2019), 1418–1457. https://doi.org/10.1007/s10664-018-9664-z
[39] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. 2020. Evaluating Code Readability and Legibility: An Examination of Human-centric Studies. In *International Conference on Software Maintenance and Evolution (ICSME)*. 348–359. https://doi.org/10.1109/ICSME46990.2020.00041
[40] Jevgenija Pantiuchina, Michele Lanza, and Gabriele Bavota. 2018. Improving Code: The (Mis) Perception of Quality Metrics. In *International Conference on Software Maintenance and Evolution (ICSME)*. 80–91. https://doi.org/10.1109/ICSME.2018.00017
[41] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. 2019. Classifying code comments in Java software systems. *Empirical Software Engineering (ESE)* 24, 3 (June 2019), 1499–1537. https://doi.org/10.1007/s10664-019-09694-w
[42] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program Comprehension and Code Complexity Metrics: An fMRI Study. In *International Conference on Software Engineering (ICSE)*. 524–536. https://doi.org/10.1109/ICSE43902.2021.00056
[43] Valentina Piantadosi, Fabiana Fierro, Simone Scalabrino, Alexander Serebrenik, and Rocco Oliveto. 2020. How does code readability change during software evolution? *Empirical Software Engineering (ESE)* 25, 6 (Nov. 2020), 5374–5412. https://doi.org/10.1007/s10664-020-09886-9
[44] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. 2011. A simpler model of software readability. In *Working Conference on Mining Software Repositories (MSR)*. 73–82. https://doi.org/10.1145/1985441.1985454
[45] Pooja Rani, Arianna Blasi, Nataliia Stulova, Sebastiano Panichella, Alessandra Gorla, and Oscar Nierstrasz. 2023. A decade of code comment quality assessment: A systematic literature review. *Journal of Systems and Software (JSS)* 195, C (Jan. 2023), 22 pages. https://doi.org/10.1016/j.jss.2022.111515
[46] Talita Vieira Ribeiro, Paulo Sérgio Medeiros dos Santos, and Guilherme Horta Travassos. 2023. On the Investigation of Empirical Contradictions - Aggregated Results of Local Studies on Readability and Comprehensibility of Source Code. *Empirical Software Engineering (ESE)* 28, 6 (Nov. 2023), 41 pages. https://doi.org/10.1007/s10664-023-10360-5
[47] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. 2012. Contemporary Peer Review in Action: Lessons from Open Source Development. *IEEE Software* 29, 6 (2012), 56–61. https://doi.org/10.1109/MS.2012.24
[48] Martin P. Robillard, Deeksha M. Arya, Neil A. Ernst, Jin L. C. Guo, Maxime Lamothe, Mathieu Nassif, Nicole Novielli, Alexander Serebrenik, Igor Steinmacher, and Klaas-Jan Stol. 2024. Communicating Study Design Trade-offs in Software Engineering. *Transactions on Software Engineering Methodolgy (TOSEM)* 33, 5, Article 112 (June 2024), 10 pages. https://doi.org/10.1145/3649598
[49] Devjeet Roy, Sarah Fakhoury, John Lee, and Venera Arnaoudova. 2020. A Model to Detect Readability Improvements in Incremental Changes. In *International Conference on Program Comprehension (ICPC)*. 25–36. https://doi.org/10.1145/3387904.3389255
[50] Felice Salviulo and Giuseppe Scanniello. 2014. Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals. *ACM International Conference Proceeding Series* (05 2014). https://doi.org/10.1145/2601248.

2601251

[51] Isabel Braga Sampaio and Alberto Sampaio. 2024. Replication of a Study about the Impact of Method Chaining and Comments on Readability and Comprehension. In *2024 4th International Conference on Code Quality (ICCQ)*. 35–52. https://doi.org/10.1109/ICCQ60895.2024.10576941

[52] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2017. Automatically assessing code understandability: how far are we?. In *International Conference on Automated Software Engineering (ASE)*. 417–427.

[53] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2021. Automatically Assessing Code Understandability. *Transactions on Software Engineering (TSE)* 47, 3 (2021), 595–613. https://doi.org/10.1109/TSE.2019.2901468

[54] Bonita Sharif and Jonathan I. Maletic. 2010. An Eye Tracking Study on camelCase and under_score Identifier Styles. In *International Conference on Program Comprehension (ICPC)*. 196–205. https://doi.org/10.1109/ICPC.2010.41

[55] Martin Shepperd. 1988. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal (SEJ)* 3, 2 (March 1988), 30–36. https://doi.org/10.1049/sej.1988.0003

[56] Dag I.K. Sjøberg, Aiko Yamashita, Bente C.D. Anda, Audris Mockus, and Tore Dybå. 2013. Quantifying the Effect of Code Smells on Maintenance Effort. *Transactions on Software Engineering (TSE)* 39, 8 (2013), 1144–1156. https://doi.org/10.1109/TSE.2012.89

[57] Armstrong Takang, Penny Grubb, and Robert Macredie. 1996. The effects of comments and identifier names on program comprehensibility: An experimental investigation. *Journal of Programming Languages* 4 (09 1996), 143–167.

[58] Ewan Tempero, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, Diana Kirk, Juho Leinonen, Asma Shakil, Robert Sheehan, James Tizard, Yu-Cheng Tu, and Burkhard Wuensche. 2024. On the comprehensibility of functional decomposition: An empirical study. In *International Conference on Program Comprehension (ICPC)*. 214–224. https://doi.org/10.1145/3643916.3644432

[59] T. Tenny. 1988. Program readability: Procedures versus comments. *Transactions on Software Engineering (TSE)* 14, 9 (1988), 1271–1279. https://doi.org/10.1109/32.6171

[60] Tri Minh Triet Pham and Jinqiu Yang. 2020. The Secret Life of Commented-Out Source Code. In *International Conference on Program Comprehension (ICPC)*. 308–318. https://doi.org/10.1145/3387904.3389259

[61] Asher Trockman, Keenen Cates, Mark Mozina, Tuan Nguyen, Christian Kästner, and Bogdan Vasilescu. 2018. "Automatically assessing code understandability" reanalyzed: combined metrics matter. In *International Conference on Mining Software Repositories (MSR)*. 314–318. https://doi.org/10.1145/3196398.3196441

[62] Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically Classifying and Propagating Natural Language Comments via Program Analysis. In *International Conference on Software Engineering (ICSE)*. 1359–1371.