Unveiling Confusion in Android Data Persistence Libraries

Paulo Guilherme Medeiros Marques
Federal University of Ceara
Fortaleza, Ceara, Brazil
pgmededeiros@gmail.com

Lincoln Souza Rocha
Federal University of Ceara
Fortaleza, Ceara, Brazil
lincoln@dc.ufc.br

Windson Viana de Carvalho
Federal University of Ceara
Fortaleza, Ceara, Brazil
windson@virtual.ufc.br

Abstract

Data persistence libraries play a central role in the Android ecosystem, where maintainability and performance are critical concerns. However, their internal complexity can hinder both. In this work, we investigate the prevalence and evolution of "Atoms of Confusion" (ACs), the smallest pieces of code that can induce misinterpretation by developers, in SQLite, Realm, ObjectBox, and Room. We also explore their performance impact to determine whether they are conscious optimizations or merely coding habits. Our findings reveal a clear dichotomy. Modern Java-based implementations exhibit lower prevalence rates (20%) and tend to encapsulate confusing constructs, whereas SQLite's native core shows widespread contamination, affecting approximately 70% of files, predominantly due to Omitted Curly Braces. An analysis of SQLite's evolution over a decade (2015–2025) aligns with Lehman's Law of Increasing Complexity, showing a strong correlation between software growth and the presence of ACs. Finally, our performance analysis in C++ indicates that most patterns provide only marginal gains, with Post-Increment being the only case that consistently outperforms its refactored counterpart.

Keywords

atoms of confusion, android apps, data persistence, code quality

1 Introduction

In recent years, mobile devices have become an integral aspect of everyday life. The widespread use of smartphones, tablets, and other connected devices has transformed how individuals, businesses, and governments produce and consume data. In 2025, Brazil surpassed 502 million active digital devices, averaging more than two devices per inhabitant [5]. In this scenario, Android stands out as the dominant mobile platform, accounting for over 75% of the global market [17]. Its open-source nature and broad adoption have established it not only as a platform for application consumption but also as a key environment for technological innovation.

The ubiquity of mobile applications demands robust and maintainable software infrastructure, particularly in data persistence libraries that support the Android ecosystem. However, the internal complexity of such libraries poses challenges to a fundamental activity in software maintenance: code comprehension. Large-scale studies with professionals indicate that developers spend about 58% of their time reading and understanding existing code rather than implementing new functionality [21]. In this context, the concept of "Atoms of Confusion" (ACs) becomes particularly relevant.

Originally proposed by Gopstein et al. [7], an AC is a syntactically valid code pattern that, while functionally correct, can induce misinterpretations due to its unintuitive structure, introducing difficulties in understanding the code. The concept emerged from a systematic analysis of intentionally obfuscated code, specifically the winning entries from the International Obfuscated C Code Contest (IOCCC)[1]. In their seminal study [7], the authors broke down these programs until they isolated the smallest fragments responsible for causing confusion. Those that could be rewritten into clearer, semantically equivalent forms were classified as ACs.

Listings 1 and 2 present two equivalent code snippets: one containing an AC, extracted from Realm [2]; and another in a refactored, clearer form. The AC is located in the conditional expression of Listing 1, where the use of the Conditional Operator combines a null check with a negated equality comparison in a single statement. This structure influences comprehension by forcing the reader to simultaneously evaluate multiple conditions, namely whether `name` is `null`, whether it is equal to `simple.name`, and the effect of the negation. In the clean snippet, shown in Listing 2, the logic is decomposed into an explicit `if/else` structure. This eliminates the risk of confusion by separating each condition into distinct branches.

Listing 1: Confusing code containing an AC.

```
name != null ? !name.equals(simple.name) :
    simple.name != null
```

Listing 2: Equivalent code without an AC.

```
if (name != null) {
    return !name.equals(simple.name);
} else {
    return simple.name != null;
}
```

Although both versions are syntactically correct and semantically equivalent, their difference lies in the likelihood of human misinterpretation. While such subtle differences may seem insignificant at first glance, they caused bugs in the past [15, 20], such as the Apple "goto fail", an error in which *Omitted Curly Braces* and a mismatched indentation compromised network connections in iOS [2]. In addition to their impact on human cognition, a recent study showed how Large Language Models (LLMs) respond to these patterns [1]. Evidence suggests that code regions containing ACs exhibit higher model uncertainty during processing, which correlates with human neurophysiological responses associated with confusion in code comprehension tasks. Results suggest an alignment between difficulties faced by humans and LLMs.

Several studies have proposed taxonomies and tools capable of locating ACs in large-scale codebases, linking them to software

[1] https://www.ioccc.org/
[2] https://github.com/realm/realm-java

quality metrics and the occurrence of defects [10–12, 20]. In the mobile ecosystem, previous research used these kinds of tools to examine ACs occurrence and their impact within general application logic, frequently limiting the analysis to a single programming language (e.g., Java, Swift) [3, 4, 18, 19]. In this work, we address these limitations by focusing on a more specialized and critical aspect, i.e., **data persistence**, and by adopting a multilingual perspective that includes code written in Java and C/C++.

Within this domain, data persistence mechanisms, especially those based on embedded databases and Object Relational Mappers (ORMs), introduce additional layers of abstraction and performance-oriented design decisions. These characteristics may encourage the use of compact or non-intuitive constructs, potentially shaping how ACs appear and evolve over time. This context motivates the following research question: **What is the phenomenon of prevalence and persistence of ACs in the data persistence domain in Android code written in Java and C/C++?**

With this question in mind, we carried out three complementary studies. First, Study 1 characterizes the prevalence of ACs across four Android data persistence libraries through automated analysis. Subsequently, Study 2 examines the evolution of these constructs over a decade within a representative library from the initial set. Finally, Study 3 assesses the performance implications of the most frequent patterns identified in Study 2.

The remainder of this paper is organized as follows. Section 2 presents the methodology, including the research design, data collection, and evaluation metrics. Section 3 describes Study 1, which analyzes the prevalence and distribution of ACs across the selected libraries. Section 4 investigates the evolution of these patterns over time, focusing on SQLite. Section 5 evaluates their impact on performance through controlled micro-benchmarks. Section 6 discusses the results in relation to the research questions. Finally, Section 7 concludes the paper and highlights directions for future work.

2 Methodology

Our work goal is to analyze the prevalence, evolution, and performance implications of ACs in data persistence libraries used in Android: Room[3], Realm[4], ObjectBox[5], and SQLite[6]. We seek to identify trends, patterns, and assess their impact on code maintainability and efficiency. For that, we defined three research questions:

- **RQ1** - What is the prevalence and distribution of atoms of confusion in major Android data persistence libraries?
- **RQ2** - How does the occurrence of ACs evolve over time in one of these libraries?
- **RQ3** - What is the impact of ACs on software performance?

2.1 Library Selection

The selection of persistence libraries followed criteria commonly adopted in empirical studies on software repositories. As discussed by Kalliamvakou et al. [9], explicit selection criteria are necessary when mining platforms such as GitHub to avoid biases related to inactive, experimental, or non-representative projects, thereby preserving the external validity of the results.

Based on these guidelines, the following criteria were applied: (i) widespread adoption in Android projects, indicated by metrics such as stars, downloads, and usage in public repositories; (ii) project maturity, reflected by a history of releases and active maintenance; (iii) availability of open-source code and documentation, ensuring reproducibility; (iv) relevance in academic or technical contexts; (v) coverage of different persistence paradigms, including relational and object-oriented approaches; and (vi) predominance of Java and/or C/C++ code.

Four libraries were selected, representing established solutions for data persistence in the Android ecosystem: **(I) Room** is the official persistence library provided by Google as part of Android Jetpack. It offers an abstraction layer over SQLite, including annotation-based mapping, compile-time query validation, and integration with modern Android architectures [6]. Its widespread adoption and active maintenance make it representative of current development practices; **(II) Realm** is an object-oriented database designed for mobile. It abstracts persistence without requiring direct SQL usage and has a long development history, making it suitable for analyzing code evolution over time [13]; **(III) ObjectBox** is a high-performance database optimized for embedded and mobile applications. Its design emphasizes efficiency and simplicity, providing an opportunity to observe how ACs appear in performance-oriented codebases [14]; and **(IV) SQLite**, although not exclusive to Android, is the underlying persistence mechanism of the platform [16]. Its inclusion allows the analysis of ACs in a mature, low-level system where performance constraints are critical.

2.2 Data Collection

Data collection was conducted in two main stages: (i) ACs automated detection in the selected libraries and (ii) mining of their development history. This process supports both the prevalence (RQ1) and evolution analysis (RQ2).

2.2.1 Tools. ACs were identified using two complementary tools. For Java, we used **BOHR**[7], proposed by Mendes et al. [12], which detects instances of ACs based on the taxonomy introduced by Langhout et al. [10]. BOHR has been empirically validated and adopted in Java studies, providing adequate precision and consistency for large-scale analyses. For components implemented in C/C++, particularly in SQLite, we used **AtomFinder**[8], a tool designed to detect ACs in languages of the C family [8]. Since the original definition of ACs was derived from C-based programs, AtomFinder ensures consistency when analyzing low-level code not covered by BOHR.

Both tools have known limitations. BOHR is restricted to Java and does not analyze Kotlin or native code. AtomFinder may miss patterns in complex macro expansions or edge cases related to parsing. Additionally, both approaches are subject to false positives and false negatives inherent to automated detection. These limitations are considered when interpreting the results.

[3] https://github.com/androidx/androidx
[4] https://github.com/realm/realm-java
[5] https://github.com/objectbox/objectbox-java
[6] https://github.com/sqlite/sqlite
[7] https://github.com/wendellmfm/bohr
[8] https://github.com/dgopstein/atom-finder

2.2.2 Repository Mining. The development history of each data persistence library was obtained from its official Git repository. The repositories were cloned locally, enabling access to their full version history, including commits, tags, and directory structures.

For the prevalence analysis (RQ1), we considered the latest version available in the main branch of each project. For the evolution analysis (RQ2), we extracted all official releases identified through version tags. Following the recommendations of [9], only tagged and stable versions were included to avoid biases related to incomplete or experimental states on RQ2.

Relevant source files were identified based on language (Java and C/C++) and project structure. To evaluate the holistic cognitive load of the repositories' maintenance perimeter, all analyzable source code was included in the dataset. For each selected version, the codebase was processed by the appropriate detection tool (BOHR or AtomFinder), and the resulting data were stored for analysis. Additional metrics, such as LOC, were extracted to support normalization and comparative analysis across projects and versions.

2.3 Performance Benchmark

To investigate the ACs impact on software performance (RQ3), we designed a controlled experiment based on micro-benchmarks. The goal was to compare the execution behavior of code snippets containing ACs with their semantically equivalent, refactored versions.

A benchmark suite was implemented in C/C++, consisting of pairs of functions representing two variants of the same logic: (i) a version containing a specific AC and (ii) a clean version without it. The ACs selected correspond to the five most frequent patterns identified in RQ2, ensuring that the experiment focuses on realistic and representative cases. Each benchmark executes the same workload under both variants, isolating the structural differences introduced by the AC. The code was compiled using `g++` with optimization disabled (`-O0`), as recommended by Valgrind, preventing the compiler from rewriting confusing constructs. This evaluation is limited to C/C++ code. Evaluating Java was considered out of scope, as it would require a different experimental design to control for JIT compilation, warm-up cycles and garbage collection overheads. Consequently, to maintain consistency and isolation, we focused solely on the most prevalent patterns found in SQLite.

Performance data were collected using two complementary tools. The metrics were obtained using the Nanobench[9] library, which provides high-resolution timing and statistical summaries. Memory-related metrics were obtained using Valgrind[10]. An automated script was used to run the benchmarks repeatedly and consolidate the results into structured datasets for analysis.

2.4 Metrics

2.4.1 Prevalence Metrics (RQ1). To measure the ACs presence, we use the *absolute count* (N_{ac}), defined as the total number of detected ACs in a given project. This metric provides a direct view of how frequently these patterns occur. To account for differences in code size, we compute the number of *lines of code* (LOC), considering only the files analyzed by the detection tools. Based on these values, we derive the *density of ACs*, expressed as the number of ACs per thousand lines of code (KLOC):

$$D_{ac} = \frac{N_{ac}}{LOC} \times 1000 \tag{1}$$

This enables comparisons across libraries of different sizes. To capture how ACs are distributed within the system, we also compute the *average number of ACs per file*:

$$\mu_{file} = \frac{N_{ac}}{N_{file}} \tag{2}$$

where N_{file} is the total number of analyzed files. Additionally, we measure the *propagation rate*, defined as the percentage of files containing at least one AC:

$$R_{prop} = \frac{N_{aff}}{N_{file}} \times 100 \tag{3}$$

where N_{aff} is the number of affected files. This metric indicates whether ACs are concentrated in specific components or spread throughout the system.

Finally, we analyze the *distribution by AC type*, capturing the relative frequency of each category.

2.4.2 Evolution Metrics (RQ2). To analyze the ACs evolution, we track changes across consecutive project versions. The *variation in density* is computed as:

$$\Delta D_{ac} = D_{ac}(v_i) - D_{ac}(v_{i-1}) \tag{4}$$

and the *absolute variation* in the number of ACs is defined as:

$$\Delta N_{ac} = N_{ac}(v_i) - N_{ac}(v_{i-1}) \tag{5}$$

These metrics allow us to identify whether ACs are being more introduced, removed, or remain stable over time. In addition, we monitor the evolution of μ_{file} and R_{prop} to assess how the ACs distribution changes across versions. Complementarily, we identify *hotspots*, defined as files with the highest ACs concentration.

2.4.3 Performance Metrics (RQ3). To evaluate the ACs impact on performance, we used metrics derived from micro-benchmark. Execution time is measured in *nanoseconds per operation* (ns/op) using the Nanobench framework. This metric captures the average time required to execute each code variant.

Furthermore, we analyzed *instructions per cycle* (IPC) to assess how efficiently the CPU parallelizes instructions under each variant.

The impact on the memory was evaluated using Valgrind, with CPU analyses isolated to avoid interference. We measured *memory reads* (Dr) and *memory writes* (Dw) to verify whether an AC introduces unwanted stack or register overhead. Additionally, the occurrence of first-level cache misses (D1mr and D1mw) was observed to identify possible data access overheads or variations in stack usage induced by the refactoring of atoms.

All performance metrics are collected over multiple independent runs for each variant (e.g., 30 executions). To ensure data integrity, executions presenting high measurement noise (e.g., error margins > 5%) are automatically rejected, and the aggregated values (e.g., mean) of the stable runs are used for comparison between implementations with and without ACs.

[9] https://github.com/martinus/nanobench
[10] https://valgrind.org/

3 Study 1: Prevalence

This study investigates the prevalence and ACs distribution in the selected data persistence libraries, addressing RQ1. The analysis considers both Java and C/C++ components, allowing a comprehensive assessment of the libraries. While Java code predominates in Room, Realm, and ObjectBox, native components play a central role in SQLite and partially in Realm.

3.1 Study Design and Dataset

The dataset consists of the latest available versions of the four data persistence libraries. They differ in size, programming language and architecture, which impacts the interpretation of the results.

Table 1: Code Distribution by Language

Library	Java		C/C++		Kotlin	
	LOC	Files	LOC	Files	LOC	Files
SQLite	12.096	58	391.780	351	0	0
ObjectBox	37.775	299	0	0	753	13
Realm	140.353	676	16.118	88	43.764	212
Room	3.462	95	0	0	161.399	1.105

Table 1 summarizes these aspects. Realm has the largest Java codebase, with $\approx 140K$ lines of code, while ObjectBox also presents a considerable Java implementation. In contrast, Room contains a relatively small amount of analyzable Java code. This is due to the predominance of Kotlin in its modern implementation, which is not supported by the detection tools used in this study. As a result, the analysis of Room is limited to components written in Java.

Although SQLite includes a small Java interface layer, its core is implemented in C, totaling $\approx 400K$ lines of code. The Java portion serves primarily as an access interface for Android applications. These differences highlight the importance of considering language distribution and system architecture when interpreting the ACs prevalence across projects.

3.2 Results

3.2.1 Prevalence and Density. The total number of ACs detected varies across projects and is naturally influenced by code size. To enable comparison, results are normalized using ACs density (AC/K-LOC). Table 2 summarizes these values.

Table 2: ACs Density by Language

Library	Java			C/C++		
	LOC	ACs	Density	LOC	ACs	Density
SQLite	12.096	100	8,26	391.780	18.002	45,94
Realm	140.353	1.018	7,25	12.848	130	10,11
ObjectBox	37.775	614	16,25	—	—	—
Room	3.462	37	10,68	—	—	—

The results reveal substantial variation in density across the analyzed libraries. ObjectBox exhibits the highest density, indicating a higher concentration of potentially confusing constructs relative to its code size. Realm, despite having a larger codebase, presents a lower density, suggesting a more consistent use of clearer constructs. Room shows comparatively low values; however, this result must be interpreted with caution due to the limited portion of analyzable code. SQLite stands out with the highest density in the study, indicating a strong ACs concentration in its codebase.

3.2.2 Granularity and Propagation. Beyond global density, file-level metrics provide insight into how confusion is distributed within each project. Table 3 presents the average number of ACs per file and their propagation rate.

Table 3: Prevalence and Propagation of ACs by Language

Library	Atoms	Files	Affected	Mean	Propagation
Java					
SQLite	100	58	12	1,72	20,69%
Realm	1.018	676	147	1,51	21,75%
ObjectBox	614	299	76	2,05	25,42%
Room	37	95	11	0,39	11,58%
C/C++					
SQLite	18.002	351	245	51,29	69,80%
Realm	130	64	25	2,03	39,06%

SQLite combines a high propagation rate with a high average number of ACs per file, indicating that confusion is both widespread and dense. A large proportion of its files contain at least one AC, and the number of ACs per affected file is higher than in the other libraries. This suggests a systemic pattern, where ACs are pervasive across the codebase rather than confined to specific components.

In contrast, ObjectBox presents a different distribution. Although its overall density is relatively high, its propagation rate is considerably lower, indicating that ACs are concentrated in a smaller subset of files. Most files in the project do not contain ACs, while a limited number of components accumulate a higher concentration. This pattern suggests localized confusion. Realm follows a similar trend to ObjectBox but with lower overall intensity. Both the propagation rate and the average number of ACs per file are comparatively moderate, indicating a more balanced distribution. ACs are present, but neither highly concentrated nor broadly pervasive.

Table 4: Files with the Highest AC Concentration per Library

Library	1st Place		2nd Place		3rd Place	
	File	*#*	*File*	*#*	*File*	*#*
JAVA						
SQLite	`CApi.java`	26	`Tester1.java`	19	`Sqlite.java`	16
Realm	`AllTypesRealm Proxy-pre-dictionary.java`	57	`AllTypesRealm Proxy.java`	56	`RealmConfig-uration.java`	43
Room	`Song.java`	12	`AutoValue_ Person.java`	5	`Artist.java`	4
ObjectBox	`Utf8Safe.java`	73	`ArrayRead WriteBuf.java`	60	`Abstract Object.java`	47
C/C++						
SQLite	`jimsh0.c`	1.635	`sqlite3-wasm.c`	601	`btree.c`	561
Realm	`utf8.hpp`	34	`java_ accessor`	25	`java_object _accessor.hpp`	22

Hotspot analysis reinforces these observations (Table 4). In ObjectBox, the most affected files are utility classes associated with

low-level data handling, such as buffer manipulation and encoding. In Realm, the highest concentrations appear in proxy and generated classes, indicating that part of the confusion originates from code generation mechanisms. In SQLite, hotspots include core components responsible for crucial operations, showing that highly confusing constructs are embedded in critical parts of the system.

3.2.3 AC Type Distribution. To complement the quantitative analysis, the distribution of ACs types was examined. Figures 1, 2 and 3 present the frequency of the most common patterns.

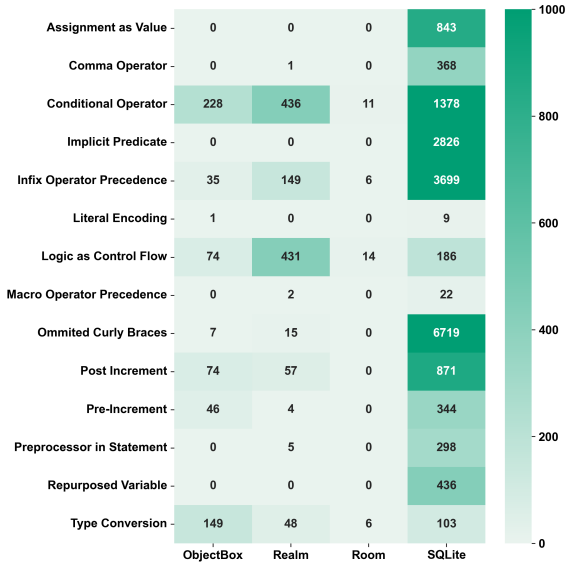

Figure 1: ACs Occurrence by Library

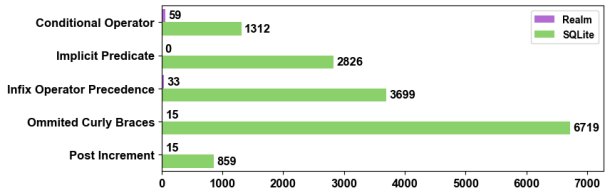

Figure 2: Top 5 ACs in C/C++

ObjectBox and Realm exhibit similar typological profiles, dominated by ACs related to control flow, particularly the use of *Conditional Operators* and logical expressions as control structures. These constructs are used extensively and account for a large portion of the detected ACs, suggesting a preference for concise implementations that may increase cognitive load.

ObjectBox also shows the presence of *Type Conversion* snippets, which align with its use of low-level data manipulation mechanisms. Realm, in turn, shows a concentration of *Logic as Control Flow*, especially in proxy and generated components.

SQLite presents a different distribution, with dominance of patterns associated with omitted structural elements, particularly the *Omitted Curly Braces*. These ACs appear extensively throughout the codebase and contribute to the overall density and propagation.

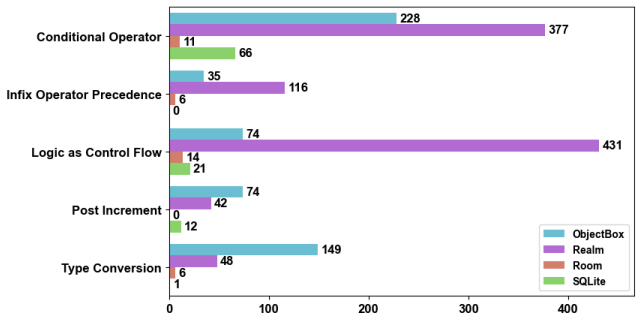

Figure 3: Top 5 ACs in Java

4 Study 2: Evolution

This study investigates how ACs evolve over time, addressing RQ2. The goal is to analyze whether these patterns tend to increase, decrease, or stabilize throughout the software lifecycle.

The analysis focuses exclusively on SQLite. This choice is motivated by three factors. First, its implementation allows the use of AtomFinder, enabling consistent detection of ACs across all analyzed versions. Second, SQLite provides a long and stable version history, which supports the observation of trends over an extended period. Third, its role as a widely used persistence engine since it's available on over 4 billion devices, which increases the practical relevance of these findings.

4.1 Study Design

The study is based on the analysis of official stable versions of SQLite, obtained from its versioning history. Each version was processed independently, allowing the extraction of metrics related to the ACs presence.

The analysis considers the evolution between consecutive versions, allowing the identification of patterns such as accumulation, reduction, or stabilization of ACs. In addition, changes in density are examined to account for variations in code size, ensuring that growth in the codebase does not distort the interpretation of results.

4.2 Results

4.2.1 Global Evolution. We analyzed the absolute evolution of ACs in SQLite across stable releases from version 3.10.0 (2015) to 3.50.0 (2025), as well as the current state of the main branch. The selected versions represent major release milestones, spaced approximately two to three years apart, allowing the identification of long-term trends while reducing noise from minor revisions.

Table 5 summarizes the evolution of lines of code (LOC) and the total number of ACs detected.

The results show an increase in the ACs number over time. The system evolves from 11,072 ACs in version 3.10.0 to 19,072 in version 3.50.0, representing a growth of 72.25%. This increase occurs without any reduction phase across stable releases.

Figure 4 illustrates this trend, showing an increase in the ACs number throughout the analyzed period. A minor decrease is observed in the current main branch compared to v3.50.0, but this variation is small and does not alter the overall upward trajectory.

Table 5: Historical Evolution and Variations (SQLite)

Version	LOC	Atoms	Previous Var.	Cumulative Var.
		C/C++		
master	391.780	18.002	-1.070	+6.930
v3.50.0	412.404	19.072	+2.385	+8.000
v3.40.0	373.912	16.687	+1.586	+5.615
v3.30.0	343.927	15.101	+1.269	+4.029
v3.20.0	312.783	13.832	+2.760	+2.760
v3.10.0	250.129	11.072	**Ref.**	**Ref.**
		Java		
master	12.096	100	0	+100
v3.50.0	12.075	100	+100	+100
v3.40.0	0	0	0	0
v3.30.0	0	0	0	0
v3.20.0	0	0	0	0
v3.10.0	0	0	**Ref.**	**Ref.**

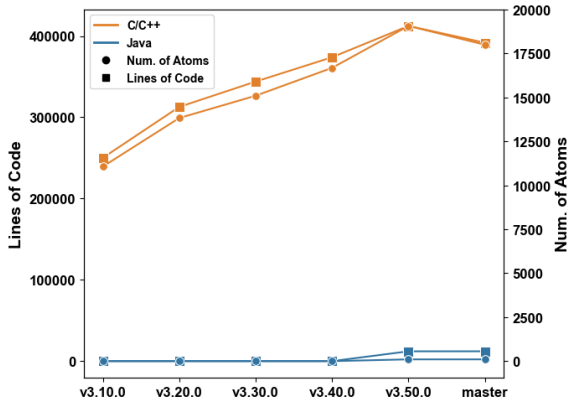

Figure 4: Relation Between Number of ACs and LOC in SQLite

When compared with the growth of the codebase, a strong positive correlation emerges (r = 0,95). Figure 4 also shows that the increase in LOC is closely followed by an increase in the ACs number. Over the analyzed period, the codebase expands by approximately 162,000 lines, while the ACs number increases by around 8,000.

This corresponds to an average introduction rate of approximately 49 atoms per 1,000 lines of code added. This value is slightly higher than the historical average density of the project, indicating that newly introduced code tends to include atoms at a higher rate than the existing baseline.

4.2.2 Density Evolution. To control for the effect of system size, the analysis considers the evolution of atom density (AC/KLOC). Table 6 presents the density values across versions.

The results indicate that density does not follow a linear trend. Instead, two distinct phases can be observed, as shown in Figure 5.

Between versions 3.10.0 and 3.30.0, density remains stable despite substantial growth in LOC. During this period, density slightly decreases from 44.27 to 43.91 AC/KLOC, suggesting that code additions were proportionally aligned with existing patterns.

Table 6: Density Evolution and Variations (SQLite)

Version	LOC	Atoms	Dens.	Previous Var.	Cumulative Var.
			C/C++		
master	391.780	18.002	45,94	-0,30	+1,68
v3.50.0	412.404	19.072	46,24	+1,62	+1,98
v3.40.0	373.912	16.687	44,62	+0,72	+0,36
v3.30.0	343.927	15.101	43,90	-0,32	-0,36
v3.20.0	312.783	13.832	44,22	-0,04	-0,04
v3.10.0	250.129	11.072	44,26	**Ref.**	**Ref.**
			Java		
master	12.096	100	8,26	-0,02	—
v3.50.0	12.075	100	8,28	+8,28	—
v3.40.0	0	0	0,00	0,00	—
v3.30.0	0	0	0,00	0,00	—
v3.20.0	0	0	0,00	0,00	—
v3.10.0	0	0	0,00	**Ref.**	—

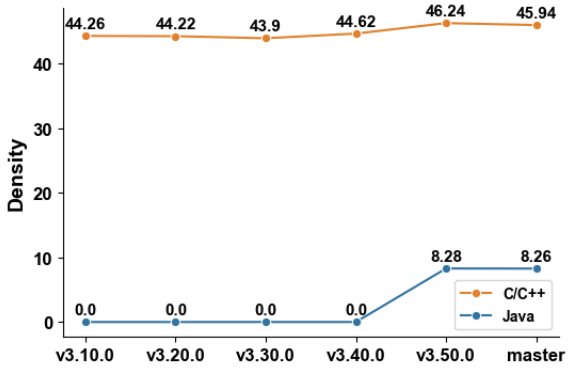

Figure 5: Density Evolution Between SQLite Versions

From version 3.40.0 onward, the trend changes: AC density starts to increase and reaches a peak of 46.25 AC/KLOC in version 3.50.0. This breaks the previous stability and suggests that in that version, new and refactored code added more ACs than removing them. The current main branch shows a small reduction to 45.95 AC/KLOC, but this density is higher than the first one we analyzed.

4.2.3 Propagation and Granularity Evolution. We analyzed how ACs are distributed across files over time, using propagation rate and average ACs per file. Table 7 presents these metrics.

Propagation remains consistently high across all versions. The percentage of affected files never falls below 65% and stabilizes around 70% in recent versions. This indicates that ACs are broadly distributed across the codebase rather than confined to isolated modules. The average number of ACs per file (μ_{file}) reveals a change in behavior over time. From version 3.10.0 to 3.40.0, the system exhibits a dilution pattern. The number of files increases from 272 to 409, while the average ACs per file remains stable, about 40. This suggests that newly introduced ACs are distributed across new files.

In version 3.50.0 and the current main branch, this pattern changes. The number of files decreases from 409 to 351, while the total number of ACs continues to grow. As a result, the average increases

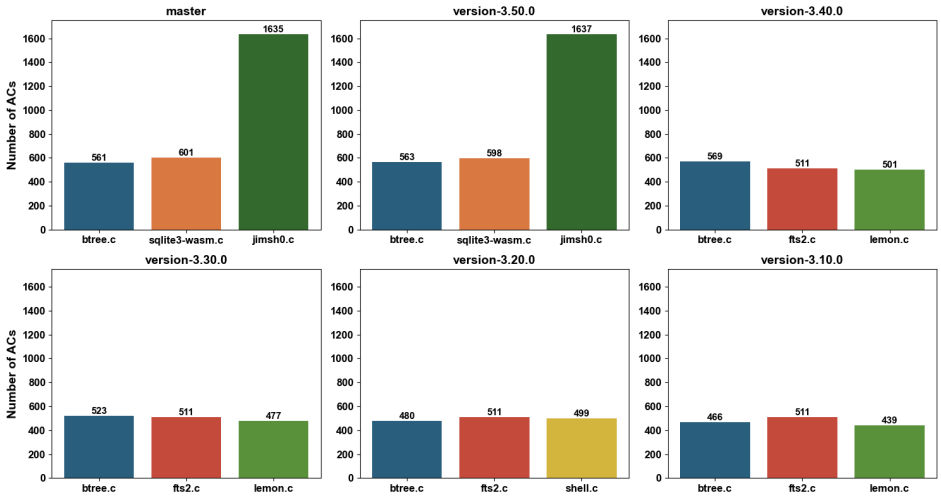

Figure 6: Three Files with the Most ACs Across SQLite Versions

Table 7: Evolution of AC Propagation by File

Version	Files	Atoms	Mean	Affected Files	Propagation
C/C++					
master	351	18.002	51,28	245	69,80%
v3.50.0	390	19.072	48,90	276	70,77%
v3.40.0	409	16.687	40,79	281	68,70%
v3.30.0	385	15.101	39,22	262	68,05%
v3.20.0	345	13.832	40,09	230	66,67%
v3.10.0	272	11.072	40,70	178	65,44%
Java					
master	58	100	1,72	12	20,69%
v3.50.0	58	100	1,72	12	20,69%
v3.40.0	0	0	0,00	0	0,00%
v3.30.0	0	0	0,00	0	0,00%
v3.20.0	0	0	0,00	0	0,00%
v3.10.0	0	0	0,00	0	0,00%

to 51.29 ACs per file, indicating a concentration effect, where ACs accumulate in fewer files. This likely occurred due to the Java inclusion in the codebase. Hotspot analysis provides more details. Figure 6 shows the evolution of files with the highest number of ACs.

Two distinct behaviors can be identified in the evolution of hotspots. The first is a pattern of chronic accumulation, exemplified by the file `btree.c`, which consistently appears among the most affected files across all analyzed versions. Its ACs number increases from 466 in version 3.10.0 to 561 in the current version, indicating continuous growth without reduction over time. The second behavior corresponds to the acute introduction of highly concentrated files in recent versions. Notably, files such as `jimsh0.c` and `sqlite3-wasm.c` emerge with disproportionately high ACs counts. In particular, `jimsh0.c` contains more than 1,630 ACs, exceeding other files and contributing substantially to the observed increase in average ACs per file.

4.2.4 Evolution by Atom Type. To understand which patterns drive the observed growth, the analysis examines changes in ACs types between version 3.10.0 and the current version. Table 8 summarizes the largest increases.

Table 8: Most Common ACs Types (2015–2025)

Atom Type	2015 (v3.10.0)	2025 (*Master*)	Growth	Var. (%)
Omitted Curly Braces	3.954	6.719	+2.765	+69,9%
Implicit Predicate	1.436	2.826	+1.390	+96,8%
Infix Operator Precedence	2.715	3.699	+984	+36,2%
Conditional Operator	752	1.312	+560	+74,5%

The growth is not uniform across types. A small subset of ACs accounts for most of the increase. The most significant increase is observed in the *Omitted Curly Braces* pattern. This AC grew from 3,954 occurrences in 2015, with an addition of 2,765 new instances. Other relevant increases include *Implicit Predicate* (from 1,436 up to 2,826) and *Infix Operator Precedence* (an increase of 984 occurrences). In contrast, other ACs such as *Comma Operator* and *Type Conversion*, remain stable, contributing minimally to overall growth.

5 Study 3: Performance

Study 3 evaluates the impact of ACs on computational performance and resource usage. A common assumption in software development is that more concise code leads to better performance. This study explicitly examines this assumption by focusing on the most frequent ACs identified in the previous studies. We assess whether these patterns provide measurable performance benefits or whether clearer refactorings introduce significant overhead.

5.1 Study Design and Experimental Subjects

Five AC types were selected based on their frequency in SQLite: (i) *Post-increment*, (ii) *Conditional Operator*, (iii) *Infix Operator Precedence*, (iv) *Implicit Predicate*, and (v) *Omitted Curly Braces*.

For each atom, controlled micro-benchmarks were constructed. Each benchmark consists of two semantically equivalent implementations: a confusing version and a refactored version. Each benchmark executes **30 times** the same workload under both variants. All experiments were conducted on a machine equipped with an AMD Ryzen 7 5700X 8-Core Processor (3.40 GHz) and 32 GB of RAM, running Windows 11. To minimize external interference, the benchmarks were executed within the Windows Subsystem for Linux (WSL), using an Ubuntu distribution.

5.2 Results

5.2.1 Memory Analysis. The memory analysis shows no observable differences between confusing and clean implementations across all evaluated patterns. The number of memory reads (Dr) and writes (Dw) remains identical in both versions, indicating that refactoring ACs into clearer constructs does not introduce additional allocation or data access overhead. This equivalence also holds for ACs that alter control flow structure, such as the *Conditional Operator*. No variations are observed in cache behavior, including L1 data cache misses (D1mr and D1mw), nor in stack usage. These results indicate that, from a memory perspective, ACs does not incur measurable memory cost.

5.2.2 Execution Analysis. Table 9 summarizes execution time, while Figures 7 and 8 compare code versions.

Table 9: Performance Comparison: Confusing vs. Clean

Atom Type	Confusing (*Mean* ± *SD*)	Clean (*Mean* ± *SD*)	*p*-Value	Cohen's *d*
Execution Time (ns/op)				
Omitted Curly Braces	4.75 ± 0.15	4.86 ± 0.10	**0.003**	**0.81**
Implicit Predicate	4.12 ± 0.51	3.92 ± 0.07	**0.045**	0.53
Infix Precedence	7.55 ± 0.11	7.62 ± 0.08	**0.006**	0.74
Conditional Operator	7.06 ± 0.08	7.08 ± 0.12	0.321	**0.25**
Post Increment	3.84 ± 0.08	4.00 ± 0.07	**0.001**	**2.15**
Instructions Per Cycle (IPC)				
Omitted Curly Braces	2.95 ± 0.07	2.90 ± 0.05	**0.001**	0.87
Implicit Predicate	2.89 ± 0.20	3.05 ± 0.01	**0.001**	1.13
Infix Precedence	3.30 ± 0.02	3.30 ± 0.02	0.456	0.19
Conditional Operator	3.21 ± 0.02	3.23 ± 0.01	**0.001**	1.27
Post Increment	2.95 ± 0.06	2.78 ± 0.04	**0.001**	3.29

We carried out a two-tailed Student's t-test for independent samples, testing the null hypothesis (H_0) that there is no difference between the confusing and refactored implementations. For the *Post-Increment*, *Omitted Curly Braces*, and *Implicit Predicate* patterns, the null hypothesis was rejected ($p < 0.05$), indicating statistically significant differences in execution time.

Post Increment shows the most consistent advantage for the confusing version, with an average of 3.84 ns/op compared to 4.00 ns/op for the refactored version ($p < 0.001$, $d = 2.15$). This difference is supported by a higher instruction throughput, with IPC values

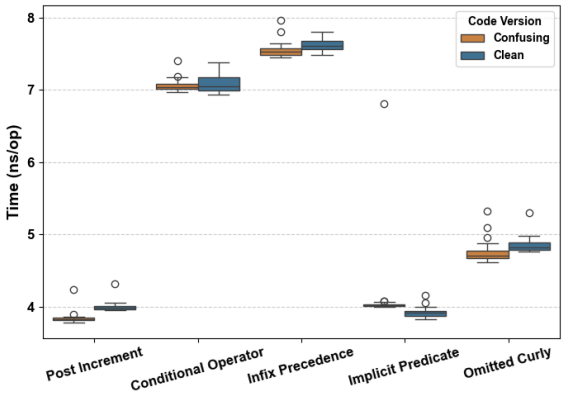

Figure 7: Ns/Op Comparison: Confusing vs. Clean

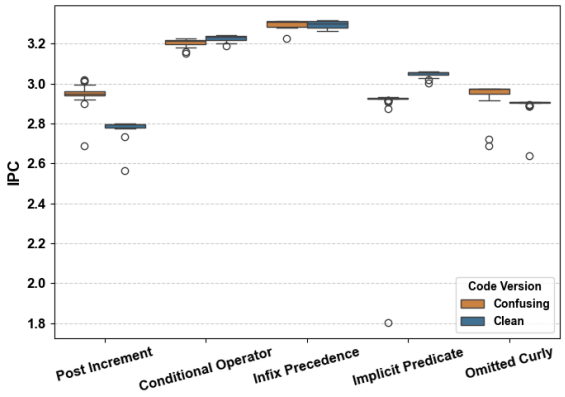

Figure 8: IPC Comparison: Confusing vs. Clean

of 2.94 for the confusing version and 2.78 for the refactored one. A similar effect is observed for *Omitted Curly Braces*, where the confusing version achieves 4.75 ns/op, while the refactored version reaches 4.85 ns/op ($p = 0.003$, $d = 0.81$), an overhead of about 2.15%.

In contrast, the *Implicit Predicate* pattern shows the opposite behavior. The refactored version performs better, with 3.92 ns/op versus 4.11 ns/op for the confusing version, and a higher IPC of 3.04. For *Infix Operator Precedence* and *Conditional Operator*, the differences are minimal. In particular, *Conditional Operator* shows no statistically significant difference ($p = 0.32$, $d = 0.25$). In this case, the choice between a conditional operator and an equivalent if/else structure does not affect performance.

6 Discussion

6.1 Answer to RQ1

Main result: The analysis reveals that ACs are present in all analyzed libraries, but with distinct profiles. Systems like ObjectBox and Realm concentrate confusion in specific components, with propagation around 20-25%, while SQLite exhibits a more pervasive distribution, with a propagation around 70%.

The results show that the prevalence and ACs density vary across the analyzed libraries. This variation is not explained solely by code size, but is probably influenced by implementation strategies and code styles adopted by their main developers.

ObjectBox and Realm exhibit moderate densities, with a clear dominance of specific ACs types, particularly *Conditional Operator* and *Logic as Control Flow*. These patterns indicate a preference for compact expressions that reduce verbosity but increase cognitive effort during code comprehension. Despite this, both libraries present relatively low propagation rates, showing that confusion is not widespread. Instead, it is concentrated in specific components, which limits its impact on the overall maintainability of the system.

SQLite presents a different profile. Its density is substantially higher, and its propagation rate indicates that ACs are present in most files. In addition, the average number of ACs per file is larger than in the other libraries. This combination characterizes a systemic distribution of confusion, where confusing constructs are embedded throughout the codebase. SQLite's analysis may reflect stylistic choices that favor minimal syntax at the cost of code clarity.

The predominance of patterns such as *Omitted Curly Braces* reinforces this observation, reflecting a coding style that prioritizes conciseness and low-level control. While such choices may be associated with performance considerations, they increase the risk of developer misinterpretation and maintenance errors.

In ObjectBox, the high incidence of *Type Conversion* is directly related to its internal design. Its reliance on low-level data manipulation leads to frequent explicit conversions, which are detected as atoms. Hotspot analysis confirms that these patterns are concentrated in utility classes responsible for buffer handling and encoding. This indicates that ACs are localized in infrastructure components rather than distributed across the entire library's codebase.

Room presents low values in both absolute and relative terms. However, this result is constrained by the limited portion of analyzable code, since most of its implementation is written in Kotlin.

Typological results also confirm previously reported patterns. In Java libraries, the dominance of *Conditional Operator* and *Logic as Control Flow* aligns with the findings of Tabosa et al. [19]. And SQLite exhibits a predominance of *Omitted Curly Braces*, consistent with the profile identified by Gopstein et al. [8] for C/C+.

These results also reflect the nature of the analyzed subjects. All libraries operate at the infrastructure level, where performance and memory efficiency are primary concerns. In this context, the use of compact or non-intuitive constructs may be a deliberate choice rather than an incidental artifact. This suggests that ACs may, in some cases, be associated with optimization strategies.

6.2 Answer to RQ2

Main result: The analysis of SQLite reveals that ACs accumulate proportionally to codebase growth, with an average introduction of 49 atoms per 1,000 LOC. While earlier versions exhibited a dilution effect, recent versions exhibit higher density and concentration, increasing to 51 atoms per file.

The results show that the evolution of ACs in SQLite follows a pattern of accumulation combined with a recent increase in density and concentration. This answers RQ2 by indicating that confusion tends to grow over time and is not reduced as the system matures.

A strong positive correlation is observed between code size (LOC) and the number of ACs (Pearson's r = 0,95). As the codebase grows, ACs increase proportionally, with an average of 49 new ACs per 1,000 lines added. This suggests that ACs are introduced as a byproduct of feature growth, rather than being controlled or removed.

More relevant than the absolute growth is the change in density and distribution. Between versions 3.10.0 and 3.30.0, density remains stable despite its expansion, indicating controlled growth. From version 3.40.0 onward, this behavior changes: density increases and reaches its peak in version 3.50.0. This indicates that recent additions introduce atoms at a higher rate than earlier code.

At the same time, the distribution of atoms across files shifts. Earlier versions show a dilution effect, where new atoms are spread across an increasing number of files, keeping the average per file stable. In recent versions, this pattern is replaced by concentration. The reduction in the number of files, combined with continued growth in atoms, results in 51 atoms per file, indicating the formation of larger and more confusing modules.

Propagation remains consistently high across all versions, $\approx$ 70% of files. This shows that atoms are present throughout the system. Both long-standing files, such as `btree.c`, and recently introduced modules contribute to this pattern. The analysis by AC type shows that growth is driven by a subset of patterns, particularly those related to control flow and logical expressions. The increase in *Omitted Curly Braces* is the most significant, next *Implicit Predicate* and *Infix Operator Precedence*. These patterns prioritize concise syntax but reduce clarity, indicating a style that favors compactness.

6.3 Answer to RQ3

Main result: Not all Atoms of Confusion provide performance gains, challenging the common assumption that more concise code is always more efficient. For *Post-Increment* and *Omitted Curly Braces*, the confusing versions were slightly faster. In contrast, for the *Implicit Predicate* pattern, the refactored version performed better, indicating that improved clarity can come without cost or even with gains. Finally, for *Infix Operator Precedence* and *Conditional Operator*, the differences were negligible.

The results of Study 3 do not support the assumption that confusing code is inherently more efficient. The memory analysis shows identical behavior across all evaluated patterns, indicating that modern compilers generate equivalent memory access regardless of syntactic form. Therefore, the findings indicate that the presence of these atoms cannot be explained by memory optimization.

Execution results are mixed. *Post Increment* and *Omitted Curly Braces* show small, but statistically significant, advantages for the confusing versions. Although the differences are minor, they may accumulate in performance-critical contexts like SQLite. In contrast, the Implicit Predicate pattern shows the opposite effect: the refactored version is both faster and more stable. For *Conditional Operator* and *Infix Operator Precedence*, no significant differences were observed, indicating no measurable performance benefits.

Overall, the impact of ACs on performance is inconsistent. Some patterns yield marginal gains, while others degrade performance or have no effect. This indicates that efficiency concerns do not justify ACs, as clarity can improve without reliable performance loss.

6.4 Implications

The findings suggest that ACs should be studied as recurring and evolving characteristics of software systems, rather than as isolated syntactic issues. For researchers, the contrast between localized ACs in Java-based libraries and the widespread presence of ACs in SQLite highlights the importance of analyzing confusing code in domain-specific and language-specific contexts. The longitudinal results also indicate that AC density, propagation, and hotspot concentration may serve as useful indicators of maintainability degradation, architectural pressure, and future comprehension difficulties in long-lived systems.

For practitioners, the results show that confusing constructs should be monitored during development and code review, especially in infrastructure and performance-critical components. Static analysis tools and coding guidelines can help teams identify patterns such as *Omitted Curly Braces*, *Conditional Operator*, *Logic as Control Flow*, and *Implicit Predicate*. The performance analysis further shows that confusing code should not be preserved based on assumed efficiency, since most ACs provide negligible or inconsistent gains, and clearer versions may even perform better. Therefore, refactoring decisions should be guided by empirical evidence, code clarity, and the specific distribution of ACs within each project.

6.5 Threats to Validity

Construct Validity The identification of ACs relies on two automated tools that implement predefined taxonomies and detection heuristics. As a result, not all ACs may have been captured or correctly identified. Nevertheless, these tools have been previously used in other studies to measure the prevalence and occurrence of ACs. In addition, the code snippets analyzed in this study were manually inspected to ensure correct identification. Furthermore, our dataset includes all shipped analyzable source code, such as generated proxy classes and vendored third-party files (e.g., `jimsh0.c`). While intentionally reflecting the holistic maintenance load, distinguishing authored ACs from inherited code is crucial, as the latter can inflate density metrics.

Internal Validity The performance evaluation is based on controlled micro-benchmarks using isolated code snippets. These were executed in an x86-64 WSL environment with disabled compiler optimizations (`-O0`) to ensure repeatability and to prevent the compiler from erasing the source-level structural distinctions under study. Although this design allows precise measurement of execution characteristics, it does not capture interactions present in real-world systems. Consequently, the observed performance differences may not fully reflect behavior in production environments.

External Validity The study focuses on four data persistence libraries, which, although representative and widely used, do not cover the full diversity of the Android ecosystem. In particular, the limited support for Kotlin in the detection tools restricts the analysis of modern Android codebases, especially in the case of Room. Therefore, the findings may not generalize to systems implemented in Kotlin or to other application domains beyond data persistence. Additionally, our performance findings reflect baseline x86-64 execution. These results cannot be directly generalized to other architectures or production environments, where different compiler behaviors and optimizations may occur.

Conclusion Validity The statistical analysis of performance results is based on repeated executions and hypothesis testing; however, measurement noise and other factors may still influence the results. Although high-variability executions were filtered, residual variance may affect the detection of small performance differences.

7 Final Considerations and Future Work

This study investigates the prevalence, evolution, and performance impact of Atoms of Confusion in Android data persistence libraries through static analysis, longitudinal study, and micro-benchmarking, providing a unified view of their effects on maintainability and execution. Results show two distribution profiles: in Java libraries, confusion is localized with $\approx 20\%$ propagation, whereas SQLite's native core exhibits a systemic pattern with $\approx 70\%$ propagation. Longitudinal analysis reveals that, in SQLite, ACs are persistent and cumulative, with a structural shift from dilution to concentration starting at version 3.50.0, leading to higher density and reduced modularity, consistent with Lehman's Law of Increasing Complexity. Performance results indicate no memory-related benefits and inconsistent execution gains: some patterns slightly improve throughput, while others degrade performance and increase variability, suggesting that their persistence is often based on perceived rather than actual performance advantages.

The main limitation of this study is the current tool-gap for Kotlin, which restricts the analysis of modern Android codebases. Additionally, the differences in AC catalogs between C and Java preclude a direct quantitative comparison between languages. Also, the performance evaluation considers only a limited subset of ACs, restricted to a single programming language, and implemented as isolated micro-benchmarks. These implementations do not fully capture the complexity, interactions, and contextual constraints present in real-world systems. As a result, the findings reflect controlled experimental behavior rather than the full performance implications of these patterns in production environments.

Future research should focus on developing detection tools for Kotlin to provide a comprehensive view of modern mobile development. Also, the performance evaluation should be expanded to cover a broader set of ACs in C, as well as patterns from other languages and architectures. Future benchmarks should also incorporate ACs extracted from real-world systems, allowing the analysis to better reflect practical usage scenarios found in production code.

Lastly, controlled studies involving human developers are needed to empirically assess whether higher AC density per file leads to increased comprehension time and higher error rates during maintenance. Comparative analyses across ecosystems and refactoring strategies could further support sustainable coding practices.

Artifact Availability

The datasets from this study are available at: https://github.com/httpaulie/AoC_Android_Persistence_Dataset; and the scripts used for the performance benchmark are available at: https://github.com/httpaulie/AoC_Benchmark_C.

Acknowledgements

This research was partially supported by CNPq, under grant number 307831/2025-6. ChatGPT was used to revise and improve the text.

References

[1] Youssef Abdelsalam, Norman Peitek, Anna-Maria Maurer, Mariya Toneva, and Sven Apel. 2026. Are Humans and LLMs Confused by the Same Code? An Empirical Study on Fixation-Related Potentials and LLM Perplexity. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM. https://www.se.cs.uni-saarland.de/publications/docs/APM+26.pdf

[2] H. A. Boyes, P. Norris, I. Bryant, and T. Watson. 2014. Trustworthy software: lessons from 'goto fail' & heartbleed bugs. In *9th IET International Conference on System Safety and Cyber Security (2014)*. IET, London, UK, 1–7. doi:10.1049/cp.2014.0970

[3] Fernando Castor. 2018. Identifying Confusing Code in Swift Programs. In *Proceedings of the VI Workshop on Software Visualization, Evolution and Maintenance (VEM 2018)*. SBC, Porto Alegre, RS, Brasil, 41–48. doi:10.5753/vem.2018.14442

[4] Diego N. Feijó, Carlos D. A. de Almeida, and Lincoln S. Rocha. 2024. Studying the Impact of CI/CD Adoption on Atoms of Confusion Distribution and Prevalence in Open-Source Projects. *Journal of Software Engineering Research and Development* 12, 1 (Nov. 2024), 17:1 – 17:15. doi:10.5753/jserd.2024.4118

[5] FGV. 2025. Brasil tem mais dispositivos digitais em uso do que habitantes, revela pesquisa da FGV. https://portal.fgv.br/noticias/brasil-tem-mais-dispositivos-digitais-em-uso-do-que-habitantes-revela-pesquisa-da-fgv

[6] Google. 2025. Save data in a local database using Room. Google Developers. https://developer.android.com/training/data-storage/room

[7] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. 2017. Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 129–139. doi:10.1145/3106237.3106264

[8] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. 2018. Prevalence of confusing code in software projects: atoms of confusion in the wild. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 281–291. doi:10.1145/3196398.3196432

[9] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 92–101. doi:10.1145/2597073.2597074

[10] Chris Langhout and Maurício Aniche. 2021. Atoms of Confusion in Java. arXiv:2103.05424 [cs.SE] https://arxiv.org/abs/2103.05424

[11] F. Medeiros, G. Lima, G. Amaral, S. Apel, C. Kästner, M. Ribeiro, and R. Gheyi. 2019. An investigation of misunderstanding code patterns in C open-source software projects. *Empirical Software Engineering* 24 (2019), 1693–1726. doi:10.1007/s10664-018-9666-x

[12] Wendell Mendes, Oton Pinheiro, Emanuele Santos, Lincoln Rocha, and Windson Viana. 2022. Dazed and Confused: Studying the Prevalence of Atoms of Confusion in Long-Lived Java Libraries. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, New York, NY, USA, 106–116. doi:10.1109/ICSME55016.2022.00018

[13] MongoDB. 2025. Atlas Device SDK para Java. https://www.mongodb.com/pt-br/docs/atlas/device-sdks/sdk/java/

[14] ObjectBox. 2024. ObjectBox Documentation. https://docs.objectbox.io/

[15] Oton Pinheiro, Lincoln Rocha, and Windson Viana. 2023. How They Relate and Leave: Understanding Atoms of Confusion in Open-Source Java Projects . In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, Los Alamitos, CA, USA, 119–130. doi:10.1109/SCAM59687.2023.00022

[16] SQLite. 2025. SQLite Documentation. https://sqlite.org/docs.html

[17] Statcounter. 2025. Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide

[18] Davi Tabosa, Oton Pinheiro, Lincoln Rocha, and Windson Viana. 2024. A Dataset of Atoms of Confusion in the Android Open Source Project. In *Proceedings of the 21st International Conference on Mining Software Repositories* (Lisbon, Portugal) *(MSR '24)*. Association for Computing Machinery, New York, NY, USA, 520–524. doi:10.1145/3643991.3644874

[19] Davi Tabosa, Windson Viana, and Lincoln Rocha. 2024. Atoms of Confusion in the Android Open Source Project: A Prevalence Study. In *Anais do XII Workshop de Visualização, Evolução e Manutenção de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 35–44. doi:10.5753/vem.2024.3840

[20] Adriano Torres, Caio Oliveira, Márcio Okimoto, Diego Marcílio, Pedro Queiroga, Fernando Castor, Rodrigo Bonifácio, Edna Dias Canedo, Márcio Ribeiro, and Eduardo Monteiro. 2023. An Investigation of confusing code patterns in JavaScript. *Journal of Systems and Software* 203 (2023), 111731. doi:10.1016/j.jss.2023.111731

[21] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering* 44, 10 (2018), 951–976. doi:10.1109/TSE.2017.2734091