How Rust Projects Use Macros: A Large-Scale Empirical Study

Leopoldo Teixeira
Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil
lmt@cin.ufpe.br

Nilo Drumond
Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil
nbmcd@cin.ufpe.br

Abstract

Macros are a central feature of the Rust programming language, enabling metaprogramming and compile-time code generation. However, there is limited empirical understanding into how they are used in practice. Using a descriptive survey design, we analyze macro usage across the 99 most popular open-source Rust repositories on GitHub, comprising 4,001 crates and 2,215,517 total invocations, with a custom Tree-sitter-based static analysis pipeline. We address three research questions: (RQ1) how are invocations distributed across macro categories (declarative, derive, attribute, function-like). (RQ2) how concentrated is macro usage around a small set of widely-used ecosystem crates. (RQ3) how does invocation density vary across project domains. We find that declarative macros dominate macro definitions while function-like and declarative call sites jointly dominate macro invocations. Derive and attribute macros are concentrated among a small set of widely-used ecosystem crates: Three crates alone account for 31.9% of user-defined attribute invocations. Finally, educational and AI/LLM-oriented projects exhibit higher density than general-purpose application code.

Keywords

Empirical Study, Open Source, Static Analysis, Software Reuse

1 Introduction

Rust is a modern systems programming language recognized for its performance, concurrency support, and robust memory safety guarantees achieved without the overhead of a garbage collector [7, 10]. Among Rust's key distinguishing features is its powerful macro system, enabling metaprogramming and compile-time code generation. Macros enable abstraction, reduce boilerplate code, and allow developers to express domain-specific logic [9]. Due to their versatility, macros are often used not only within Rust's standard library and compiler internals but also across third-party crates (the Rust ecosystem's unit of compilation and distribution) and applications, supporting tasks such as automatic trait derivation, syntactic transformations, and custom compile-time behaviors.

Despite their widespread adoption, there remains limited empirical evidence regarding how macros are used in Rust, particularly at a large scale. Ernst et al. [4] and Dietrich [3] analyze C macro usage. Pappas and Gazzillo [16] address macro portability within C, while Peng et al. [17] address portability to Rust. The absence of studies on macro usage in Rust poses challenges for language designers, tool developers, and practitioners aiming to understand maintainability, complexity, and productivity implications.

In this paper, using a descriptive survey design, we conduct the first large-scale empirical characterization of macro usage patterns in Rust, to the best of our knowledge. It covers the 99 most popular open-source Rust repositories on GitHub, comprising 4,001 crates and more than 2.2 million macro invocations. We aim to systematically identify trends in macro usage, categorize prevalent macro types, and reveal which projects rely most heavily on macros.

To achieve these goals, we developed Ferrometer,[1] a static analysis tool using GitHub's GraphQL API and the Tree-sitter framework [2]. Our tool classifies macro usage into four categories: declarative macros, derive macros, attribute macros, and function-like macros, enabling a detailed quantitative assessment of macro usage patterns. Our results show that declarative macros dominate macro definitions while function-like and declarative call sites jointly dominate macro invocations, reflecting their ease of use and versatility. In particular, derive and attribute macros from prominent crates such as `serde` and `tokio` demonstrate broad adoption across diverse project contexts. Macro usage intensity also varies across domains, with educational and AI/LLM-oriented projects exhibiting the highest density, indicating that domain context meaningfully shapes macro adoption.

We believe that this study provides guidance for both Rust developers and researchers. Understanding macro adoption practices can inform future macro-aware tooling developments, improve maintainability and readability in macro-intensive projects, and contribute to better guidelines and documentation.

More specifically, from a software-reuse perspective, macros are themselves a reuse mechanism: Rather than duplicating boilerplate across projects, developers reuse macro definitions published by ecosystem crates, invoking them uniformly across otherwise unrelated codebases. Our RQ2 findings characterize this reuse pattern concretely: A small number of crates supply the macros that a large share of the ecosystem imports and invokes. This is analogous to how reusable architectural frameworks standardize cross-cutting concerns in component-based systems.

The remainder of this paper is structured as follows. Section 2 reviews Rust's macro system and motivates the need for empirical investigation. Section 3 details our methodology for data collection and analysis. Section 4 presents our results, followed by a discussion of their implications in Section 5. Section 6 addresses threats to validity, and we review related work in Section 7. Finally, Section 8 concludes the paper and proposes future research directions.

2 Background and Motivation

Rust is a statically typed systems programming language designed for performance-critical applications, providing safety and concurrency guarantees without relying on a garbage collector [7, 10]. The

[1] From *ferro-*, evoking iron and rust, and *-meter*, a measuring instrument

macro system is a core feature, allowing sophisticated metaprogramming and compile-time code generation. Using macros, developers can eliminate boilerplate code, achieve higher levels of abstraction, and express domain-specific constructs, all without sacrificing runtime efficiency or static safety guarantees.

Unlike traditional macro systems found in languages like C or C++, which rely on textual substitution performed by preprocessors, Rust macros operate directly on tokens rather than raw text. C preprocessor macros suffer from several well-documented pitfalls: argument expressions are substituted verbatim, so a macro like `#define SQUARE(x) x*x` evaluates `SQUARE(1+2)` to `1+2*1+2 = 5` rather than 9. Macros can silently capture variables from the surrounding scope, causing subtle name-clash bugs, and there is no type checking on arguments, so type errors are only caught (if at all) after expansion. Rust avoids these pitfalls: declarative `macro_rules!` macros are token-based and hygienic, meaning their internal bindings cannot capture or be captured by identifiers at the call site. Procedural macros are also token-based but are unhygienic, and are best understood as compile-time token transformations whose output is subsequently type-checked by the compiler. Rust's macro system is categorized into declarative and procedural macros [9], each tailored to distinct use cases and complexity levels.

Declarative macros, defined using the `macro_rules!` syntax, are pattern-matching macros that transform code snippets according to predefined rules. They are widely used due to their simplicity, readability, and hygienic expansion behavior, making them accessible even to developers new to metaprogramming. Listing 1 defines a `vec!` macro: the pattern `$($x:expr),*` matches a comma-separated sequence of expressions, and the expansion pushes each matched expression onto a freshly allocated `Vec`. A call such as `vec![1, 2, 3]` expands to three `push` calls, code that would otherwise be written by hand for every collection initialization. Without `vec!`, initializing a three-element vector requires declaring a `Vec`, calling `push` three times, and returning it, four lines of code instead of one. Declarative macros eliminate this class of boilerplate.

Listing 1: Declarative macro for vector initialization

```
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $( temp_vec.push($x); )*
            temp_vec
        }
    };
}
```

Procedural macros, in contrast, offer deeper customization by running at compile time and transforming token streams representing Rust syntax. They can be understood as functions from a `TokenStream` to another `TokenStream`, coming in three main varieties: derive macros, attribute macros, and function-like macros. Derive macros automate trait implementations, reducing boilerplate in scenarios such as serialization or debugging. For instance, Listing 2 shows an example from the `serde` crate that uses derive macros to automatically generate serialization code, simplifying data interchange across different formats. This use of `#[derive(...)]` spares developers from implementing boilerplate code for data interchange, making it useful for data-centric applications.

Listing 2: Derive macro for automatic serialization

```
#[derive(Serialize, Deserialize)]
struct User {
    name: String,
    age: u32,
}
```

Attribute macros enable annotations that transform code items, altering their behavior or generating auxiliary code. An example is the `tokio` runtime macro, which conveniently encapsulates setup for asynchronous programming, as seen in Listing 3.

Listing 3: Attribute macro from the tokio crate

```
#[tokio::main]
async fn main() {
    // asynchronous code here
}
```

Function-like procedural macros share call-site syntax with declarative macros (i.e., `name!(...)`) but are implemented as Rust programs that receive a token stream and return a transformed token stream. They are used when pattern-matching is insufficient. For example, `quote!` from the `quote` crate constructs Rust token trees programmatically, and `sqlx`'s `query!` macro verifies SQL statements against a live database at compile time. As they are compiled as separate crates, function-like macros have access to the full Rust language and crate dependencies, making them more powerful than declarative macros, at the cost of higher complexity.

The Rust macro system is *hygienic*, meaning that it guarantees that identifiers introduced by a macro expansion do not accidentally capture or are captured by identifiers in the surrounding code. This is also called referential transparency of bindings [19]. If a declarative macro internally uses a variable named `temp`, that binding is invisible to the caller's scope, and vice versa. This prevents typical bugs from C macros, where a helper variable inside a macro could clash with a same-named variable at the call site, and makes macro expansions compositional: two independently authored macros can be used together without risk of identifier interference [6].

Macros are integral to Rust's rich and growing ecosystem. Popular libraries such as `serde`, `tokio`, and `thiserror` heavily rely on macros to offer concise, expressive APIs for common development scenarios: Serialization, asynchronous programming, and error handling. Despite their extensive use and practical importance, detailed empirical analyses of macro adoption and usage patterns remain sparse, leaving open questions about how developers use macros in large, real-world projects.

3 Methodology

We designed and implemented a static analysis pipeline to analyze macro usage patterns within Rust's open-source ecosystem. Figure 1 shows the two-phase pipeline implemented in Ferrometer. Phase 1 (corpus collection) queries GitHub's GraphQL API for the top-99 starred Rust repositories, clones them, and walks the filesystem to discover crates, producing a corpus lockfile that pins the

exact repository state for reproducibility. Phase 2 (static analysis) passes each `.rs` file through Tree-sitter, classifies every macro invocation into one of four categories, normalizes counts by lines of code, and emits the per-repository `results.csv` and an interactive HTML report. The subsections that follow detail each step. Phase 1 constitutes the survey's sampling frame, the population from which the corpus is drawn. Phase 2 implements the measurement instrument applied uniformly to every sampled repository.

Study design. This study follows a descriptive survey design [20]. We formulate three research questions, each targeting a distinct aspect of macro usage that cannot be answered by inspection of individual repositories:

- **RQ1** *How are macro invocations distributed across the four macro categories (declarative, derive, attribute, and function-like)?* We expected function-like and declarative macros to dominate invocation counts, since utility macros like `println!` and `vec!` are invoked repeatedly per function, whereas derive and attribute macros apply once per declaration.
- **RQ2** *How concentrated is macro usage around a small set of widely-used ecosystem crates?* We expected usage to mirror crates.io's well-known long-tail concentration, since macros are typically distributed by crates solving cross-cutting problems (serialization, async execution, error handling), concentrating invocations in a few providers.
- **RQ3** *How does invocation density vary across project domains?* We expected variation by domain: educational projects' pedagogical code samples may over-represent idiomatic macro use, while projects that build infrastructure code and rely on serialization and async I/O should elevate derive/attribute usage relative to application code, which has more hand-written logic.

The methodology that follows describes how Ferrometer was designed to answer these questions at scale.

3.1 Corpus Construction

We began by using GitHub's GraphQL API to select the top 99 public Rust repositories on GitHub based on their star count as of data collection. This popularity-based selection ensures a focus on mature, widely adopted, and actively maintained projects. We chose $n = 100$[2] as a corpus size large enough to expose ecosystem-level concentration patterns (RQ2) while keeping per-repository statistics (RQ3) interpretable and the manual domain-labeling task tractable. The exact threshold is not critical to the qualitative findings, as our sensitivity analysis (Section 6) confirms.

To assess domain coverage, we assigned each of the 99 repositories to one of six categories: systems and CLI tools (29 repos), applications (20), AI and LLM tooling (17), libraries (17), developer tools (11), and educational projects (5) (two repositories are excluded from the RQ3 density analysis due to anomalous *ts_cloc_ratio* values, see Section 6). The labeling proceeded in two steps. First, we used a large language model as an initial-pass classifier following the LLM-as-judge methodology [14], providing each repository's name, GitHub description, README, and primary topic tags as input and prompting it to assign a single category from the six-class scheme. Second, one author of this paper reviewed every LLM-proposed label against the repository's README and source, adjusting 13 of 99 labels (13.1%). The human-adjusted labels are the ground truth used in all downstream analyses. The LLM step served as a labor-saving first pass, not as an independent rater.

We acknowledge that LLM-assisted labeling carries risks of prompt sensitivity and category bias that human review only partially mitigates [14], and that the resulting labels have not been validated against an independent second human rater. We discuss the implications in Section 6.

Crate and workspace identification. Rust projects are structured by *crates*, identified by the presence of a `Cargo.toml` file. However, repositories often define workspaces aggregating multiple crates. To correctly identify analyzable crates, we recursively traversed each repository's filesystem, distinguishing between workspace configurations (with a `[workspace]` section) and individual crate configurations (with a `[package]` section). Only directories defining crates were analyzed further.

Filtering and file selection. Instead of performing traditional upfront filtering, we adopted an incremental parsing strategy for file selection. Each `.rs` source file found was immediately parsed using Tree-sitter. Files failing initial parsing, often test fixtures or intentionally malformed code, were excluded immediately, minimizing unnecessary overhead. This strategy proved practical and robust, filtering irrelevant or unusable files without requiring additional manual configuration.

3.2 Parsing and Macro Classification

Syntax tree analysis with Tree-sitter. We chose Tree-sitter[3] due to its speed, scalability, and incremental parsing capabilities, well-suited for analyzing large-scale codebases across diverse projects. Tree-sitter produces structured Abstract Syntax Trees (ASTs) with precise positional information, enabling detailed extraction of macro definitions and invocations.

Specifically, we extracted `macro_invocation` nodes, macro definitions (`macro_definition` nodes, or procedural macro attributes), and derive lists and attribute annotations. As a simplified illustration, consider the following declarative macro definition presented in Listing 4. The `#[macro_export]` attribute exposes the macro at the crate root, making it visible to importers.

Listing 4: Simple declarative macro definition

```
#[macro_export]
macro_rules! say_hello {
    () => {
        println!("Hello, world!");
    };
}
```

Tree-sitter captures such definitions, distinguishing identifiers, patterns, and expansions, enabling precise downstream classification and analysis. Our tool parses the code into the following high-level structure.

[2]We queried the top-100 at the beginning of June 2026, but GitHub's GraphQL API included a repository in which the majority of code is not in Rust, so we discarded it.

[3]https://tree-sitter.github.io

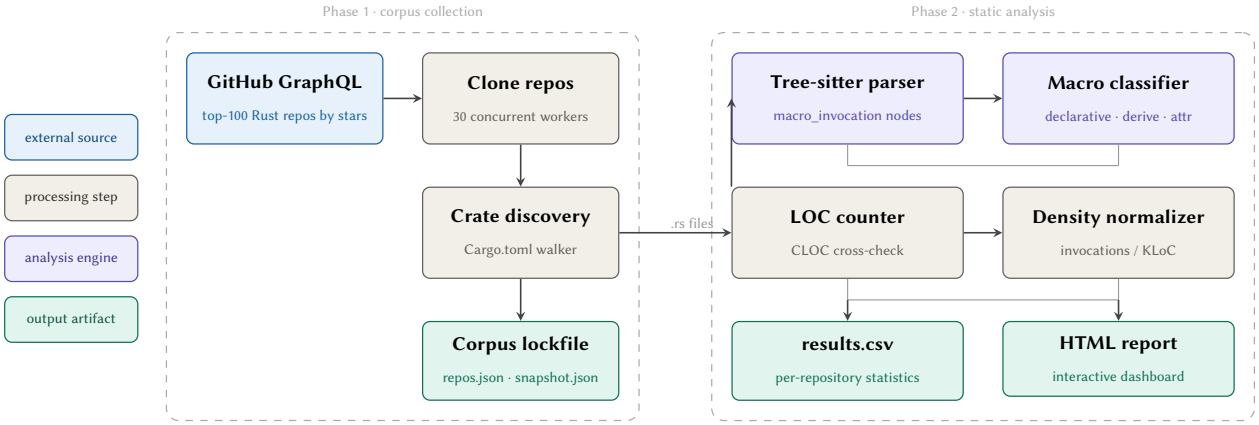

Figure 1: End-to-end macro analysis pipeline.

```
source_file
 +-- macro_definition
     +-- identifier: say_hello
     +-- rules
         +-- pattern
         +-- expansion
```

This structure enables rule-based detection of where macros are declared and used, independent of stylistic differences.

Macro type classification and heuristics. Macros were classified into four categories: declarative macros, derive macros, attribute macros, and function-like macros, as shown in Table 1. However, due to Tree-sitter's purely syntactic parsing limitations, ambiguities occasionally arose, particularly in distinguishing function-like procedural macros from declarative macros. To mitigate these ambiguities, we relied on heuristics informed by syntactic context and known macro sets from standard libraries and popular crates. While this introduced a minor classification approximation, it was consistent across all analyzed projects and sufficiently accurate for large-scale pattern analysis.

Table 1: Macro categories and characteristics

Type	Description
Declarative	Defined using `macro_rules!`; match-based; user-defined.
Derive	Used via `#[derive(...)]`; automatically implements traits.
Attribute	Custom annotations like `#[tokio::main]`; may change item behavior.
Function-like	Invoked like macros but implemented as procedural transformations.

Declarative/function-like conflation. Declarative (`macro_rules!`) and function-like procedural macros share identical call syntax, so Tree-sitter parses both as the same AST node type. Separating them would require macro expansion, which we did not perform. The two result columns therefore report the same counter. We quantify the impact in what follows.

3.3 Classifier Validation

Before reporting results, we present an empirical validation of the classification approach described above. This validation quantifies how reliably the syntactic heuristics in Section 3.2 correspond to human judgment, and motivates the merged declarative/function-like category used in all quantitative claims that follow.

To assess the reliability of our classification heuristics, we conducted an inter-rater validation study. We drew a stratified random sample with target $n = 50$ per category to achieve adequate per-category precision estimates. The allocation is therefore non-proportional to corpus distribution. The *built-in attribute* and *user-defined attribute* strata received smaller allocations (34 and 16 respectively) because the population of user-defined attribute paths in the corpus was smaller than the declarative or derive populations. For each instance we extracted the surrounding source context.

Two researchers labeled all 200 items independently and blind to each other: one author of this paper, with extensive previous experience with macros, preprocessors, and software reuse, and one independent rater who is a professional software developer with experience using Rust. For practical reasons, we sampled candidate items by text-pattern matching across the corpus rather than by re-invoking the AST-based analyzer. Nine items were subsequently identified as text-pattern false positives (macro names appearing inside string literals or comments) and excluded from validation as not representative of the AST-based pipeline's output, leaving 191 comparable pairs.

Inter-rater reliability. Cohen's kappa between the two labelers was $\kappa = 0.917$ (Almost perfect on the Landis–Koch scale [12]), with 93.7% raw agreement, indicating strong agreement between independent raters.

Per-category precision. Using Author 1's labels as the resolved label and the tool's predicted category as the hypothesis, we computed per-category precision (Table 2). Derive macros achieved

perfect precision (100.0%), confirming that `#[derive(...)]` syntax is unambiguous. User-defined attributes reached 75.0% and built-in attributes 67.6%. The apparent precision of the combined declarative/function-like category was 34.1%: Of the 91 instances the tool labeled as *declarative*, 60 were judged to be function-like procedural macros and only 31 to be genuine `macro_rules!` invocations. This directly quantifies the known limitation described above and motivates treating declarative and function-like as a single merged category in all quantitative claims.

Table 2: Per-category classifier precision from inter-rater validation (n = 191, 9 unsure items excluded). Resolved label = Author 1.

Category	Prec.	TP / pred.
Derive	100.0%	50 / 50
User-defined attr	75.0%	12 / 16
Built-in attr	67.6%	23 / 34
Decl.+Fn-like†	34.1%	31 / 91

†Conflation artifact; see text.

To enable fair comparison across repositories of varying sizes, we normalized all invocation counts by lines of code, reporting macro invocation density as invocations per thousand lines of code (KLoC). Collected data were serialized to JSON and aggregated into a per-repository CSV for statistical analysis; an interactive HTML report was generated automatically from the same JSON and is included in the replication package. Ferrometer integrates four modules: a GitHub GraphQL client for corpus selection, a filesystem crawler for crate discovery, Tree-sitter bindings for parsing, and a macro walker that collects invocation and definition statistics.

4 Results

Our dataset comprises 4,001 crates drawn from 99 repositories, yielding 2,215,517 total macro invocations. In what follows, we answer each of the three research questions.

4.1 RQ1: Invocation Distribution Across Macro Categories

How are macro invocations distributed across the four macro categories (declarative, derive, attribute, and function-like)?

Figure 2 shows the distribution of macro invocations across the four categories. The combined function-like and declarative category accounts for the majority of invocations (54.0%), followed by built-in attributes (31.1%, with `#[derive]` alone representing 6.1% of all invocations) and user-defined attribute macros (14.9%). Exact counts appear in Tables 3–9.

Built-in attributes. Table 3 lists the seven most frequently invoked built-in attributes. `#[test]` leads, reflecting the Rust community's strong testing culture, followed by `#[derive]` (trait automation) and `#[cfg]` (conditional compilation).

User-defined attribute macros. Table 4 shows the seven most common user-defined attribute macros. The top three originate from `serde` (`#[serde]`), `tokio` (`#[tokio::test]`), and `thiserror` (`#[error]`). Ranks four and six (`#[stable]` and `#[unstable]`) are

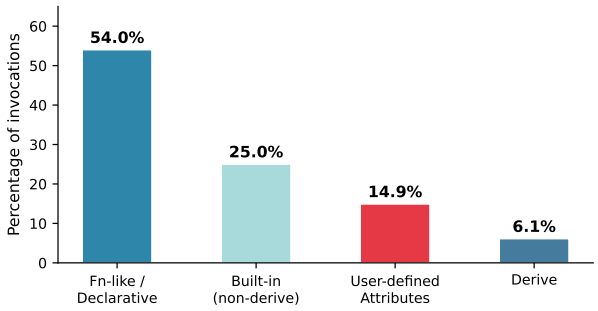

Figure 2: Distribution of macro invocations by type.

Table 3: Top 7 built-in attributes by invocation count (% of all built-in attr invocations)

Attribute	Invocations	%
`#[test]`	152,194	22.1%
`#[derive]`	135,216	19.6%
`#[cfg]`	101,987	14.8%
`#[inline]`	95,978	13.9%
`#[doc]`	54,237	7.9%
`#[cfg_attr]`	50,200	7.3%
`#[allow]`	32,777	4.8%

compiler-internal stability annotations present because the Rust compiler repository is part of the corpus.

Table 4: Top 7 user-defined attribute macros by invocation count (% of all user-defined attr invocations)

Macro	Invocations	%
`#[serde]`	54,292	16.5%
`#[tokio::test]`	32,678	9.9%
`#[error]`	18,026	5.5%
`#[stable]`	17,029	5.2%
`#[rstest]`	16,554	5.0%
`#[unstable]`	14,203	4.3%
`#[prost]`	11,237	3.4%

Function-like and declarative macros. Table 5 lists the seven most invoked function-like and declarative macros, with the share of its category total. Testing assertions (`assert_eq!`, `assert!`) lead by a wide margin, reflecting the volume of test code in popular repositories. Six of the seven entries are standard-library macros; the exception is `json!` from `serde_json` (rank 6). Table 6 shows the top seven after excluding standard-library macros (qualified and unqualified aliases merged, e.g., `log::debug!` and `debug!` combined). The non-std list is dominated by logging macros from the `log` and `tracing` crates (`debug!`, `warn!`, `info!`, `error!`), JSON construction (`json!` from `serde_json`), error propagation (`anyhow!` from `anyhow`), and a token-matching macro (`T!`) defined locally in parser-heavy projects (surrealdb, biome).

Note that Tree-sitter cannot reliably distinguish declarative (e.g., `macro_rules!`) from function-like procedural macro call sites at the syntactic level. Both share the `macro_invocation` AST node kind and are reported under a single counter (see Section 6).

Table 5: Top 7 function-like and declarative macros by invocation count (% of category total)

Macro	Invocations	%
`assert_eq!`	306,747	25.6%
`assert!`	180,009	15.0%
`format!`	106,970	8.9%
`vec!`	100,630	8.4%
`matches!`	28,121	2.3%
`json!`	25,078	2.1%
`write!`	23,416	2.0%

Table 6: Top 7 non-std function-like macros by invocation count (% of non-std fn-like total; standard-library macros excluded; qualified/unqualified aliases merged)

Macro	Invocations	%
`json!`	32,921	9.2%
`debug!`	18,434	5.2%
`warn!`	15,109	4.2%
`info!`	14,888	4.2%
`error!`	11,541	3.2%
`anyhow!`	10,864	3.1%
`T!`	10,279	2.9%

Derive macros. The corpus contains 135,216 derive annotations generating 482,069 individual trait derivations (median: 3 traits per annotation). Table 7 shows the seven most derived traits, with their share of all trait derivations; `Debug` and `Clone` dominate, reflecting near-universal adoption for debugging and cloning. Table 8 shows the top seven after excluding core-language traits (`Debug`, `Clone`, `Copy`, `PartialEq`, `Eq`, `Hash`, etc.), revealing that `serde`'s serialization traits (`Serialize`, `Deserialize`) account for nearly half of all non-std derivations.

Table 7: Top 7 derived traits by total derivation count (% of all trait derivations)

Trait	Derivations	%
`Debug`	93,938	19.5%
`Clone`	86,545	18.0%
`PartialEq`	49,301	10.2%
`Serialize`	37,489	7.8%
`Deserialize`	35,386	7.3%
`Eq`	34,262	7.1%
`Default`	24,768	5.1%

Table 8: Top 7 non-std derived traits by count (% of non-std derivations; core-trait derivations excluded: `Debug`, `Clone`, `Copy`, etc.)

Trait	Derivations	%
`Serialize`	37,489	25.2%
`Deserialize`	35,386	23.8%
`serde`	12,013	8.1%
`prost`	4,577	3.1%
`Message`	4,337	2.9%
`JsonSchema`	4,010	2.7%
`Error`	3,762	2.5%

Macro definitions. Table 9 breaks down the 7,577 macro definitions found across the corpus. Declarative definitions (7,112) outnumber procedural ones by more than 17-to-1, confirming that `macro_rules!` remains the primary vehicle for macro authoring despite the power of procedural macros.

Table 9: Macro definitions by type

Type	Definitions
Declarative (`macro_rules!`)	7,112
Derive (proc macro)	208
Attribute (proc macro)	154
Function-like (proc macro)	103
Total	**7,577**

Answer to RQ1. Function-like and declarative macros dominate usage at 54.0%. This combined figure should be read with the classifier's validated limitation in mind (Section 3.3), since our current tooling cannot reliably disaggregate the two categories from syntactic data alone. Built-in attributes (31.1%) reflect core language idioms: Testing, trait automation, and conditional compilation. User-defined attribute macros (14.9%) are led by a small set of ecosystem crates. At the definition level, `macro_rules!` greatly outnumbers procedural definitions, indicating that macro *authoring* is less common than macro *consumption*.

4.2 RQ2: Concentration of Usage Around Ecosystem Crates

How concentrated is macro usage around a small set of widely-used ecosystem crates?

Among user-defined attribute macros, the three most frequent entries in Table 4 account for 31.9% of all user-defined attribute invocations; the top seven cover 49.8%. This concentration is driven chiefly by `serde`, `tokio`, and `thiserror`, whose combined invocations account for nearly a third (31.9%) of all user-defined attribute usage.

A parallel pattern appears in derive macros: The seven most derived traits (Table 7) account for 75.0% of all 482,069 individual derivations. `Serialize` and `Deserialize` from `serde` together represent 15.1% of all derivations, confirming `serde`'s outsized role in the ecosystem. The same concentration holds for non-std derives: as

shown in Table 8, `Serialize` and `Deserialize` together account for nearly half of all non-standard trait derivations.

The function-like/declarative category shows the same pattern at the ecosystem level (Table 6): after excluding standard-library macros, logging infrastructure (`log`/`tracing`) and error handling (`anyhow`) drive the majority of non-std function-like invocations.

At the definition level, `rust-lang/rust` accounts for the largest share of macro definitions (7,112 declarative), since it hosts the compiler internals and standard-library macros consumed by all other projects.

Answer to RQ2. Macro usage is strongly concentrated. Three ecosystem crates (`serde`, `tokio`, `thiserror`) account for 31.9% (nearly a third) of all user-defined attribute usage, with the top seven covering 49.8%; `serde` alone shapes 15.1% of all trait derivations. The same concentration pattern holds for non-std function-like macros, where logging and error-handling crates dominate.

The concentration pattern observed here is qualitatively similar to what Zoghbi et al. [21] report for potentially dangerous effects in the Rust ecosystem (roughly 3% of top crates account for the bulk of flagged effects), and to broader findings on ecosystem dependency networks: A small number of upstream providers shape behavior across a large number of downstream consumers. The implication for our study is that macro adoption is not uniformly distributed across the corpus but flows through a small number of influential providers whose API decisions propagate to virtually every project that depends on them.

4.3 RQ3: Invocation Density Across Project Domains

How does invocation density vary across project domains?

Macro invocation density (invocations per KLoC) varies substantially across repositories. The corpus-wide median is 78.4 inv/KLoC, with valid-data repositories ranging from 23.1 to 147.3 inv/KLoC (repositories with *ts_cloc_ratio* < 0.5 excluded; see Section 6).

Table 10 lists the ten highest-density repositories. Two patterns are visible: Educational projects (`rust-course`, `Rust`) anchor the top, while the remainder mixes command-line utilities (`coreutils`, `starship`, `fnm`) with AI/LLM- related projects (`openhuman`, `fhevm`). The latter cluster is consistent with the elevated median density we observe for the *ai-llm* domain overall (Table 11).

To quantify per-domain differences, we manually assigned each of the 99 repositories to one of six project-domain categories (n counts reflect the 97 valid-ratio repositories): *systems-cli* ($n = 29$), *application* ($n = 20$), *ai-llm* ($n = 16$), *library* ($n = 17$), *devtools* ($n = 11$), and *educational* ($n = 4$). Table 11 reports median density per domain for the 97 valid-ratio repositories.

A Kruskal-Wallis H-test confirmed that densities differ significantly across domains ($H(5) = 25.39$, $p = 0.00012$). Pairwise Mann–Whitney U tests with Bonferroni–Holm correction identified 3 of 15 pairs as significant at the family-wise $\alpha = 0.05$ level. All three significant pairs involve the *application* domain and have large effect sizes ($|r| \geq 0.53$): *ai-llm* vs. *application* ($p_{\text{adj}} = 0.0019$, $|r| = 0.756$), *application* vs. *educational* ($p_{\text{adj}} = 0.034$, $|r| = 0.900$), and *application* vs. *systems-cli* ($p_{\text{adj}} = 0.025$, $|r| = 0.534$). In all three cases the *application* domain has the lowest median density (61.3 inv/KLoC), significantly below *ai-llm* (95.1 inv/KLoC), *educational* (104.0 inv/KLoC), and *systems-cli* (80.1 inv/KLoC).

Table 10: Top 10 repositories by macro invocation density (inv/KLoC). Repositories with *ts_cloc_ratio* < 0.5 are excluded (Section 6).

Repository	Density
`sunface/rust-course`	147.3
`rtk-ai/rtk`	138.3
`uutils/coreutils`	128.8
`googleworkspace/cli`	127.1
`TheAlgorithms/Rust`	117.6
`zama-ai/fhevm`	117.4
`tinyhumansai/openhuman`	113.7
`starship/starship`	110.2
`Schniz/fnm`	110.0
`zeroclaw-labs/zeroclaw`	109.7

Table 11: Median macro invocation density (inv/KLoC) by project domain. Kruskal–Wallis $H = 25.39$, $df = 5$, $p = 0.00012$. Repositories with *ts_cloc_ratio* < 0.5 excluded.

Domain	Repos	Median inv/KLoC
educational	4	104.0
ai-llm	16	95.1
systems-cli	29	80.1
library	17	77.2
devtools	11	67.5
application	20	61.3

Answer to RQ3. Density varies substantially across the corpus (23.1–147.3 inv/KLoC; median 78.4). Domain is a statistically significant moderator ($H(5) = 25.39$, $p = 0.00012$). General-purpose *application* projects exhibit significantly lower density than AI/LLM, educational, and systems-CLI projects (all $p_{\text{adj}} < 0.05$, large effect sizes), indicating that macro adoption is meaningfully shaped by the role a project plays in the ecosystem.

5 Discussion

With 2.2 million invocations across 99 repositories, macros permeate virtually every aspect of Rust development: Testing, serialization, async runtimes, error propagation, and platform abstraction. The 54.0% share attributable to declarative and function-like call sites (RQ1) confirms their role as the primary workhorse for code abstraction. Derive and attribute macros account for the remainder, concentrated on a small set of ecosystem crates (RQ2): The top three user-defined attribute providers supply nearly a third of all user-attribute invocations, while `Serialize`/`Deserialize` alone represent 15.1% of trait derivations. This concentration reflects macros functioning as a de facto reuse mechanism at ecosystem scale. Invocation density (RQ3) varies by more than 6× across the corpus, with educational and AI/LLM-oriented projects at the high

end and general-purpose applications at the low end, suggesting that domain context is a meaningful moderator of macro adoption.

Macro definitions paint a complementary picture. `macro_rules!` accounts for 7,112 of the 7,577 definitions, dwarfing procedural variants. Repos at the top of the definition count do not always top the invocation count, confirming that macro *authorship* and macro *consumption* are distinct activities. Most projects consume macros from ecosystem crates without defining many of their own. `rust-lang/rust` is the single outlier: Its density is modest relative to its size, yet it contributes the largest share of macro definitions (stdlib built-ins used everywhere else in the corpus).

Our findings characterize *what* macros are used and *how much* they are used. They do not reveal *why*. We observe that AI/LLM-oriented projects have unusually high macro density, but cannot determine whether this reflects deliberate architectural choices, a particular demographic of contributors, or selection effects in which LLM-related tools tend to be built rapidly on macro-heavy frameworks. Similarly, we observe that application-domain projects have the lowest density, but cannot infer whether this reflects a deliberate preference for runtime flexibility over compile-time generation, or simply that application developers encounter fewer use cases where macros are the natural tool. These are empirical questions that longitudinal or interview-based studies could address.

5.1 Implications for Practitioners

The obtained data sketch a clear authoring discipline. Declarative `macro_rules!` accounts for 7,112 of the 7,577 macro definitions in the corpus, and has by far the lowest authorship barrier of any macro form: A project that repeats a multi-arm pattern can introduce a local declarative macro with negligible maintenance cost and immediate consistency benefit. Procedural macros, in contrast, demand more deliberate treatment. They must be compiled as separate crates and interact with the compiler's `proc_macro` API, so mixing them into application crates risks circular dependencies and slower incremental builds. Our data are consistent with this practice: the corpus's 103 function-like and 208 derive proc-macro definitions concentrate in a small number of library crates rather than spreading thinly across applications.

On the consumption side, RQ2 suggests a strong default: prefer established ecosystem macros over custom ones. Three providers (`serde`, `tokio`, `thiserror`) supply 31.9% of all user-defined attribute invocations corpus-wide. In the case of `serde`, `Serialize` and `Deserialize` alone account for 15.1% of trait derivations. Thus, rolling a custom attribute or derive macro for functionality already covered by a well-maintained crate increases maintenance burden without proportionate benefit. Macro density itself, finally, deserves attention as a maintainability signal. RQ3 shows densities varying by more than 6× across the corpus, with the highest-density repositories clustering at the educational and AI/LLM ends of the spectrum. We hypothesize, though do not directly test, that in production codebases sustained high density could indicate over-reliance on compile-time generation, with attendant costs in build time and debuggability. Validating this and establishing whether invocation density is a useful complement to conventional metrics like test coverage and cyclomatic complexity is a natural direction for future work.

This authorship-consumption asymmetry has a practical consequence that runs through the rest of this discussion: The Rust developer's primary relationship with macros is as a *consumer*, not an *author*. The macro system's visible complexity, with four categories, procedural compilation model, hygiene rules, is largely encapsulated behind the APIs of a handful of ecosystem crates. Understanding that asymmetry shapes the guidance we offer: Advice to practitioners focuses on consumption discipline, while advice to tool builders and researchers targets the gap between what syntactic analysis can see and what semantic analysis would need to see.

5.2 Implications for Tool Builders and Researchers

For the tooling community, the headline implication is that macro-awareness is not optional in Rust static analysis. With 2.2 million invocations across 99 repositories, a tool that cannot see through macro expansions is blind to a substantial fraction of the code under analysis. IDEs, linters, refactoring engines, and semantic-search tools must either integrate with `rust-analyzer`'s macro expansion infrastructure or invoke `cargo expand`; pure syntactic heuristics are insufficient even for basic distinctions. Our classifier validation (Section 3.3) makes this concrete: 60 of the 91 call sites our tool labeled as *declarative* turn out to be function-like procedural macros, a distinction that matters for any tool reasoning about where code is generated and what dependencies it crosses.

Beyond expansion, the density figures from RQ3 argue for exposing macro density as a first-class project metric, visible in repository dashboards and CI quality gates alongside coverage and complexity, so maintainers can detect macro-heavy growth as it happens rather than after build times become a problem.

For language designers and the research community, the concentration patterns in RQ2 suggest that derive and attribute macros are natural targets for ecosystem standardization. Nearly a third (31.9%) of all user-defined attribute usage flows through three crates, and a comparable share of trait derivations through one. Language-level initiatives, such as stabilizing additional built-in derives, or providing standard interfaces for common derive patterns, could reduce dependence on third-party proc-macro crates and improve interoperability across the ecosystem.

More broadly, the dataset we release is intended to support follow-up empirical work: longitudinal studies of how macro usage evolves with the language, semantic studies of what expansion actually generates, and cross-language comparisons with the macro systems of Scheme, Racket, Nim, and Elixir are all natural extensions that the present snapshot enables but does not itself perform.

6 Threats to Validity

Threats to validity are organized following the four categories proposed by Wohlin et al. [20]: *construct* (whether our measurements capture what they claim to capture), *internal* (whether observed patterns arise from the phenomenon rather than from artifacts of our method), *external* (the generalizability of our findings beyond the studied corpus), and *conclusion* (the statistical strength of our inferences). For each category we report the main threats and the mitigations we applied.

6.1 Construct Validity

Tooling, parsing, and categorization. Our analysis relies on Tree-sitter's syntactic parsing, which cannot distinguish declarative from function-like procedural macros at the call site and does not evaluate conditional compilation, uniformly over-counting macros in inactive paths. Our four-category classifier embeds heuristic choices accordingly: curated standard/ecosystem macro lists, syntactic context disambiguation, and the merged declarative/function-like category. The inter-rater validation (Section 3.3, $\kappa = 0.917$) quantifies their impact: Derive and user-defined attribute macros achieve high precision ($\geq$ 75%), while the declarative/function-like conflation remains the one exception, treated by merging the categories. Future work could refine these heuristics by integrating semantic resolution from rust-analyzer or compiler-internal tooling.

False positives in syntactic identification. Some repositories contain intentionally malformed Rust, test scaffolding, or benchmark fixtures that do not reflect production usage, and macros embedded in documentation examples, comments, or string literals may occasionally be misidentified. We mitigate these by excluding known test and fixture directories and skipping files that fail initial Tree-sitter parsing.

LOC reliability and density exclusions. Our density metric (invocations per KLoC) relies on Tree-sitter line counts, cross-validated against CLOC via the *ts_cloc_ratio* column. Two repositories[4] show anomalously low ratios (0.006, 0.13) from non-standard directory layouts causing incomplete crate discovery. We exclude repositories with *ts_cloc_ratio* < 0.5 from RQ3 density claims (2 excluded, 97 retained), while RQ1/RQ2 counts include every repository.

Domain labeling. The six-category domain assignment used in RQ3 was produced by an LLM first-pass classifier (13 of 99 labels subsequently adjusted by one author), not multiple independent human raters. This introduces three limitations: First, the LLM's choices may favor categories overrepresented in its training distribution. Second, single-rater adjustment cannot detect bias the LLM and human share. Finally, no inter-rater agreement statistic is available, unlike for the macro classifier (Section 3.3). We mitigate these by releasing the LLM prompt, model version, and both raw and human-adjusted labels, enabling external auditing.

Semantic granularity. Our analysis is purely syntactic and does not examine macros' runtime behavior, generated code size, or maintainability impact. A simple identifier replacement and a complex code-generating expansion are counted equally; our findings therefore reflect usage frequency rather than semantic weight.

6.2 Internal Validity

Because our study is descriptive and observational rather than experimental, internal validity threats are limited: We make no causal claims about what *produces* the observed patterns. The main threat is researcher bias in qualitative interpretation. Our RQ3 narrative (Section 4.3) and discussion (Section 5) reflect one reading of which patterns are noteworthy. We mitigate this by releasing the full per-repository dataset for external re-analysis.

[4] `rust-lang/rustlings` and `ruvnet/RuView`

6.3 External Validity

Selection bias and representativeness. We analyzed the 99 most popular Rust repositories on GitHub by star count, assuming popularity correlates with maturity, community relevance, and best practices. This choice may omit less visible repositories, private industry code, or experimental research tools; stratified sampling from curated registries (e.g., `lib.rs`) is a complementary strategy for better coverage of less-starred, production-grade crates. Educational repositories in particular, frequent in the top-starred set, document macro use pedagogically and may overrepresent density relative to production code.

To assess whether the 4 educational repositories distort our density findings, we re-ran the Kruskal–Wallis and pairwise Mann–Whitney tests after removing them. The results remain consistent: $H = 20.26$ ($df = 4$, $p = 0.00044$), still highly significant, with 2 of 10 pairwise comparisons surviving Bonferroni correction (versus 3/15 in the full analysis). The two surviving significant pairs both involve application-domain repositories, confirming that the key density contrast is not an artifact of including educational projects.

Temporal scope. Our analysis is a snapshot of macro usage at a specific point in time, and the rapid evolution of Rust's macro features and ecosystem conventions means that usage trends may shift as new features stabilize and legacy patterns deprecate.

6.4 Conclusion Validity

Several factors affect the statistical strength of RQ3. First, the *educational* ($n = 4$) and *devtools* ($n = 11$) groups have small sample sizes, limiting the power of pairwise tests involving them; of the three significant pairs reported in Section 4.3, only *ai-llm* vs. *application* and *application* vs. *systems-cli* involve groups with $n \geq 10$ on both sides. Second, we use Kruskal–Wallis and Mann–Whitney U tests because per-repository density is right-skewed and not normally distributed, trading statistical power for robustness. Third, we applied a Bonferroni–Holm correction across the 15 pairwise comparisons; reported p_{adj} values are adjusted accordingly.

Finally, the corpus-level percentages reported in RQ1 and RQ2 (Section 4.1–4.2) are descriptive rather than inferential. Beyond these statistical considerations, our measurement instrument's own reliability is a conclusion validity concern: The inter-rater validation (Section 3.3, $\kappa = 0.917$, 93.7% raw agreement) confirms reliable category assignments for derive, built-in attribute, and user-defined attribute invocations, with the declarative/function-like conflation (Construct Validity, above) as the one exception.

7 Related Work

Hygienic, structured macro systems trace to Kohlbecker et al.'s original hygiene algorithm [11] and Weise and Crew's programmable syntax macros [19], both developed to avoid the identifier-capture and context-dependent-expansion pitfalls of textual substitution systems like the C preprocessor (Section 2). Flatt's composability and phase-separation principles [6] d irectly inform Rust's macro design. Ernst et al. [4] present a large-scale empirical analysis of macro usage in C, categorizing definitions by form (object-like or function-like) and flagging complexity and maintainability concerns from heavy usage. Our work is similar, but benefits from

Rust's structured macro system and syntax trees for finer-grained classification and density analysis.

Pappas and Gazzillo [16] present Maki, a semantic-analysis tool that classifies C macros by their portability to safer languages like Rust, finding many are more portable than previously thought. Unlike Maki's focus on C-to-Rust portability and transformation, our work characterizes usage *within* the Rust ecosystem itself; Maki's semantic categorization could nonetheless inform future maintainability-oriented analyses of Rust macros.

Dietrich [3] introduces CppSig, which extracts type signatures from C macro invocations to reconstruct functional equivalents, complementing Mennie and Clarke's [15] work on transforming parameterless macros into constants. Unlike this reconstruction-oriented goal, our work characterizes macro usage at scale across Rust projects, where invocations are explicit and AST-aligned and signature inference is less critical than in C.

Static analysis in the presence of macros remains challenging across ecosystems: Kästner et al.'s TypeChef [8] handles macros via variability-aware parsing for C, showing that accurate analysis requires resolving macros' syntactic and semantic intricacies. Rust's structured macro system eases but does not eliminate this. Procedural macros and conditional compilation still introduce parsing complexity, which we address via Tree-sitter's syntactic parsing supplemented by classification heuristics.

Evans et al. [5] surveyed the prevalence of `unsafe` code across open-source Rust projects, finding it selective and typically confined to a small fraction of each codebase. Astrauskas et al. [1] extended this line of inquiry to ask *why* programmers reach for `unsafe`, identifying FFI, performance-critical operations, and low-level data structure implementation as primary drivers. Qin et al. [18] conducted a systematic study of memory and thread safety bugs in real-world Rust programs, cataloging root causes and suggesting tooling improvements. Li et al. [13] analyzed unstable compiler feature usage across crates.io, characterizing which features are adopted in production and the migration patterns when they stabilize. These studies characterize how Rust's safety guarantees and ownership model play out in practice. Zoghbi et al. [21] introduce Cargo Scan, a side-effects analysis for auditing third-party Rust crates, finding that potentially dangerous effects are concentrated in roughly 3% of the top crates on crates.io. This concentration pattern resonates with our finding that macro consumption is dominated by a small set of ecosystem crates (RQ2). Our work is complementary: Rather than safety features, we study macros, an orthogonal but equally central part of Rust, providing the first systematic, large-scale empirical characterization of macro usage patterns across diverse open-source projects, to the best of our knowledge.

8 Conclusion and Future Work

This work presents the first large-scale empirical characterization of macro usage patterns in Rust to our knowledge, analyzing 99 popular open-source repositories (4,001 crates, 2.2 million invocations) with the custom Ferrometer pipeline. Our three main findings are: (i) declarative macros dominate macro *definitions* while function-like and declarative call sites jointly dominate macro *invocations*; (ii) usage is strongly concentrated around a small set of ecosystem crates that function as de facto reuse providers, led by `serde`, `tokio`, and `thiserror`; and (iii) macro invocation density varies significantly across project domains, with educational and AI/LLM projects at the high end and general-purpose application code at the low end.

The findings open several directions for future work. On the empirical side, our analysis is a snapshot of macro usage at a specific point in time. Longitudinal studies tracking the same repositories across major Rust editions would help. Such studies could reveal whether the ecosystem's heavy reliance on a small set of macro providers is stable, or whether it shifts as new language features reduce the need for third-party crates. Cross-language comparisons with the macro systems of Scheme, Racket, Nim, and Elixir would situate Rust's patterns within the broader scenario of hygienic macro adoption. On the tooling side, purely syntactic analysis cannot distinguish declarative from function-like invocations, a limitation that affects any tool reasoning about this distinction, including refactoring engines, semantic search, and dependency analyzers. Such tools must integrate macro expansion via `rust-analyzer` or `cargo expand`. The corpus and validation set we release provide a concrete benchmark for evaluating such approaches. Finally, the density metric introduced here (macro invocations per KLoC) could become a standard component of Rust project health dashboards and CI pipelines, surfacing macro-heavy growth before it affects build times or maintainability; the Ferrometer pipeline is designed to support exactly that kind of continuous monitoring.

Artifact Availability

Ferrometer, the analysis pipeline, and the replication dataset are archived on Zenodo under the MIT license: https://doi.org/10.5281/zenodo.21866345. The corresponding source repository is available on GitHub at https://github.com/STAR-RG/ferrometer. The archive includes Ferrometer's Rust implementation, the Python analysis and table-generation scripts, the corpus lockfile of the repositories used in this submission (`corpus/sbcars-2026/`), the per-repository CSV (`results.csv`), the validation sample and scoring results, and the generated interactive HTML report. See the `README.md` file at the root of the repository for details on how to reproduce the results.

Acknowledgments

We used Claude Sonnet 4.6 (Anthropic) in three ways during this research. First, it produced first-pass domain labels for the corpus repositories (Section 3). All labels were subsequently reviewed and adjusted by the authors. Second, accessed via Claude Code, it assisted in the development and review of the Python scripts used for data analysis, in particular the ones that generate the LaTeX macros and tables from the analysis data. Third, it generated the TikZ source for Figure 1.

This work was partially supported by the National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence (INES.IA),[5] funded by CNPq grant 408817/2024-0. Leopoldo Teixeira is partially supported by CNPq grant 310405/2025-4.

[5] www.ines.org.br

References

[1] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How do programmers use unsafe rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (Nov. 2020), 27 pages. doi:10.1145/3428204

[2] Max Brunsfeld. 2020. Tree-sitter: A parser generator tool and incremental parsing library. https://tree-sitter.github.io/tree-sitter/.

[3] Christian Dietrich. 2021. CppSig: Extracting Type Information for C-Preprocessor Macro Expansions. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems* (Virtual Event, Germany) *(PLOS '21)*. Association for Computing Machinery, New York, NY, USA, 62–68. doi:10.1145/3477113.3487268

[4] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng.* 28, 12 (Dec. 2002), 1146–1170. doi:10.1109/TSE.2002.1158288

[5] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is rust used safely by software developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 246–257. doi:10.1145/3377811.3380413

[6] Matthew Flatt. 2002. Composable and compilable macros: you want it when?. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming* (Pittsburgh, PA, USA) *(ICFP '02)*. Association for Computing Machinery, New York, NY, USA, 72–83. doi:10.1145/581478.581486

[7] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2018), 34 pages. doi:10.1145/3158154

[8] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 805–824. doi:10.1145/2048066.2048128

[9] Daniel Keep. 2016. The Little Book of Rust Macros. https://veykril.github.io/tlborm/.

[10] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press. https://doc.rust-lang.org/book/

[11] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) *(LFP '86)*. Association for Computing Machinery, New York, NY, USA, 151–161. doi:10.1145/319838.319859

[12] J. Richard Landis and Gary G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33, 1 (1977), 159–174. http://www.jstor.org/stable/2529310

[13] Chenghao Li, Yifei Wu, Wenbo Shen, Zichen Zhao, Rui Chang, Chengwei Liu, Yang Liu, and Kui Ren. 2024. Demystifying Compiler Unstable Feature Usage and Impacts in the Rust Ecosystem. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 24, 13 pages. doi:10.1145/3597503.3623352

[14] Dawei Li, Bohan Jiang, Liangjie Huang, Alimohammad Beigi, Chengshuai Zhao, Zhen Tan, Amrita Bhattacharjee, Yuxuan Jiang, Canyu Chen, Tianhao Wu, Kai Shu, Lu Cheng, and Huan Liu. 2025. From Generation to Judgment: Opportunities and Challenges of LLM-as-a-judge. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, EMNLP 2025, Suzhou, China, November 4-9, 2025*, Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (Eds.). Association for Computational Linguistics, 2757–2791. doi:10.18653/V1/2025.EMNLP-MAIN.138

[15] C.A. Mennie and C.L.A. Clarke. 2004. Giving meaning to macros. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE, New York, NY, USA, 79–85. doi:10.1109/WPC.2004.1311050

[16] Brent Pappas and Paul Gazzillo. 2024. Semantic Analysis of Macro Usage for Portability. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 21, 12 pages. doi:10.1145/3597503.3623323

[17] Haoran Peng, Baris Kasikci, Gilbert Louis Bernstein, and Michael D. Ernst. 2026. Hayroll: A Modular Wrapper for Translating C Macros and Conditional Compilation to Rust. *Proc. ACM Program. Lang.* 10, PLDI, Article 198 (June 2026), 24 pages. doi:10.1145/3808276

[18] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 763–779. doi:10.1145/3385412.3386036

[19] Daniel Weise and Roger Crew. 1993. Programmable syntax macros. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 156–165. doi:10.1145/155090.155105

[20] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2024. *Experimentation in Software Engineering, Second Edition*. Springer, Berlin, Germany. doi:10.1007/978-3-662-69306-3

[21] Lydia Zoghbi, David Thien, Ranjit Jhala, Deian Stefan, and Caleb Stanford. 2026. Auditing Rust Crates Effectively. In *Programming Languages and Systems: 35th European Symposium on Programming, ESOP 2026, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2026, Turin, Italy, April 11–16, 2026, Proceedings, Part II* (Turin, Italy). Springer-Verlag, Berlin, Heidelberg, 424–444. doi:10.1007/978-3-032-22723-2_15