UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
CIÊNCIA DA COMPUTAÇÃO

NILO BEMFICA MINEIRO CAMPOS DRUMOND

MACRO USAGE IN THE RUST PROGRAMMING LANGUAGE IN OPEN SOURCE REPOSITORIES

Recife
2025

NILO BEMFICA MINEIRO CAMPOS DRUMOND

MACRO USAGE IN THE RUST PROGRAMMING LANGUAGE IN OPEN SOURCE REPOSITORIES

Undergraduate thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science.

Advisor: Leopoldo Motta Teixeira, PhD.

Recife

2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Drumond, Nilo Bemfica Mineiro Campos.
Macro usage in the Rust programming language in open source repositories / Nilo Bemfica Mineiro Campos Drumond. - Recife, 2025.
43 p. : il.

Orientador(a): Leopoldo Motta Teixeira
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2025.

1. Metaprogramação. 2. Macros. 3. Rust. 4. Compiladores. I. Teixeira, Leopoldo Motta. (Orientação). II. Título.

000 CDD (22.ed.)

NILO BEMFICA MINEIRO CAMPOS DRUMOND

MACRO USAGE IN THE RUST PROGRAMMING LANGUAGE IN OPEN SOURCE REPOSITORIES

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Ciência da Computação.

Aprovado em: _03__/_04__/__2025____

BANCA EXAMINADORA

_____

Prof. Dr. Leopoldo Motta Teixeira (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Leopoldo Motta Teixeira (Examinador Interno)

Universidade Federal de Pernambuco

_____

Prof. Dr. Breno Alexandro Ferreira de Miranda (Examinador Interno)

Universidade Federal de Pernambuco

ABSTRACT

This work presents an analysis of macro usage in the 100 most popular open-source Rust projects hosted on GitHub. The study aims to uncover patterns in the use of macros, which play a critical role in the Rust ecosystem by providing metaprogramming capabilities and improving code efficiency. By scraping project repositories, we identify the types of macros most frequently employed, analyze which projects rely heavily on macros, and delve into specific categories to determine the most commonly used macros. The results offer valuable insights into macro adoption trends in the Rust community and may assist developers in understanding how and why macros are applied in large-scale projects.

TABLE OF CONTENTS

1. INTRODUCTION

Rust is a modern programming language focused on memory safety, performance, and concurrency. One of its most powerful features is macros, which provide metaprogramming capabilities that allow developers to generate code, reduce redundancy, and improve maintainability. Macros are widely used in Rust projects for code abstraction, automatic trait implementation, and compile-time transformations.

This thesis examines the usage of macros in the 100 most popular open-source Rust repositories hosted on GitHub. By analyzing these widely used projects, the study aims to uncover trends in macro usage and provide insights into their role in large-scale Rust development. Key questions explored include:

- Which types of macros are most commonly used?
- Which projects make extensive use of macros?
- What are the most frequently used macros within specific categories?

To conduct this analysis, a custom tool was developed in Rust, utilizing GitHub's GraphQL API for data collection and TreeSitter for syntax analysis. The latter tool enables detailed tracking of macro definitions and invocations, allowing for a comprehensive analysis of how macros are employed across different projects.

The project tool is hosted on the following repository: https://github.com/STAR-RG/rust-macro-analyzer

1.1 Goals

The primary goal of this thesis is to analyze macro usage in Rust projects, with the following objectives:

- **Identify the most commonly used types of macros**: This will help categorize macros by their usage frequency and reveal which types predominate in popular projects.
- **Determine which projects rely heavily on macros**: By comparing macro usage across projects, we aim to uncover patterns of reliance on macros in

different contexts. Understanding what types of projects make more use of macros, and what types of projects define more macros, and if these two groups are the same or not.

- **Examine the most frequently used macros within each category**: Highlighting the most used macros will offer insights into how Rust developers apply metaprogramming in practice.

Through this investigation, the study seeks to offer valuable information for developers, whether they are experienced in Rust or transitioning from other languages, providing guidance on the practical use of macros in large projects.

1.2 Structure

This thesis is organized as follows:

- **Chapter 2** overviews the Rust programming language, focusing on macros and metaprogramming.
- **Chapter 3** details the methodology used to collect and analyze the data from Rust repositories, including the tools and libraries employed.
- **Chapter 4** presents the results, discussing macro usage trends, the most commonly used macros, and how they are distributed across projects.
- **Chapter 5** concludes the study by summarizing the key findings, discussing their relevance to Rust developers, and proposing suggestions for future research.

2. RUST PROGRAMMING LANGUAGE

This chapter introduces key aspects of Rust that set it apart from other programming languages, providing essential context for understanding the analysis of macros in later chapters.

2.1 Overview of the Rust Programming Language

Rust is a modern systems programming language known for its focus on safety, performance, and concurrency. Initially developed by Mozilla, Rust has quickly gained popularity in areas such as systems development, web applications, and embedded systems. Its unique features, particularly around memory safety, make it a powerful tool for building reliable and efficient software without sacrificing performance.

2.1.1 Memory Safety and Ownership

One of Rust's most distinguishing features is its ownership system, which ensures memory safety without needing a garbage collector. Rust enforces strict rules about how memory is accessed and managed, preventing common errors like null pointer dereferencing or data races in concurrent code.

The key concepts of Rust's memory model are:

- **Ownership**: Every value in Rust has a single owner, and when the owner goes out of scope, the value is automatically deallocated. This ownership rule ensures that memory is cleaned up safely and efficiently.
- **Borrowing**: Rust allows references to a value through borrowing, either mutably (one mutable reference) or immutably (multiple immutable references). These rules prevent data races at compile time, making concurrency safer and easier to reason about.
- **Lifetimes**: Lifetimes are used to ensure that references remain valid as long as they are needed, further guaranteeing memory safety. Rust's compiler checks lifetimes to prevent dangling references, where a value is dropped while it's still being referenced elsewhere.

Rust's approach to memory management allows it to achieve performance comparable to languages like C or C++, while providing strong guarantees against memory safety issues.

2.1.2 Core Constructs of Rust: Structs, Enums, Impl, and Traits

Rust provides several key constructs to define and manage data and behavior. They will be briefly explained in the following sections, as macros often are associated with them. These constructs are: Structs, Enums, Impl Blocks and Traits.

2.1.2.1 Structs

Structs are used to define custom data types. They allow grouping related fields together into a single entity. Although similar to *Objects* in other languages, structs carry no behavior.

```
struct Person {
    name: String,
    age: u8,
}
```

2.1.2.2 Enums

Enums allow defining a type that can have multiple variants. This is particularly useful for modeling different states or conditions in a type-safe manner. Like **struct**s, they carry no behavior. Rust's **enum**s can store data in their variants, making them more versatile than those found in languages like C or Java.

```
enum Message {
    Text(String),
    Signal(i32, i32),
    Leave,
}
```

2.1.2.3 Impl Blocks

The **impl** keyword is used to implement methods for structs and enums, allowing the definition of functions associated with a type. This enables behavior encapsulation, making Rust's type system more expressive.

```
impl Person {
    fn salute(&self) {
        println!("Hello, my name is {}", self.name);
    }
}
```

2.1.2.4 Traits

Traits in Rust define shared behavior across different types. They are similar to interfaces in other languages but provide more flexibility. A trait defines a set of methods that types can implement. Rust's trait system also enables polymorphism, allowing functions to accept parameters of any type that implements a given trait.

```
trait Salutations {
    fn salute(&self);
}

impl Salutations for Pessoa {
    fn salute(&self) {
        println!("Hello, I'm {}!", self.nome);
    }
}
```

2.2 Macros in Rust

Macros in Rust provide a mechanism for metaprogramming, allowing developers to generate and manipulate code at compile time. They are extensively used to eliminate redundancy, implement automatic code transformations, and enable domain-specific abstractions. Rust provides two main categories of macros:

- **Declarative Macros**: These macros use pattern matching to transform input syntax into generated code. They are primarily used to reduce repetition and simplify common patterns.

- **Procedural Macros**: These operate by manipulating Rust's **Abstract Syntax Tree (AST)**, allowing more complex transformations. They are further divided into:

    - **Derive Macros**: Automatically implement predefined traits for structs and enums.
    - **Attribute Macros**: Modify code components such as functions, modules, structs, or enums.
    - **Function-like Macros**: Generate arbitrary Rust code from their inputs.

2.2.1 Differences Between Macros and Functions

In Rust, although macros and functions both aim to promote code reuse and abstraction, they have fundamental differences:

- **Compile-Time Expansion**: Macros are expanded during compilation, enabling them to generate code structures that functions cannot, such as implementing traits or generating new modules.
- **Variable Number of Arguments**: Macros can accept a variable number of parameters, providing greater flexibility in their usage.
- **Code Manipulation**: Macros can manipulate code syntax and structure, whereas functions operate on values at runtime.

These differences make macros particularly suitable for tasks that require code generation or transformations that are not possible with regular functions.

2.2.2 Declarative Macros

Declarative macros use pattern matching to transform input tokens into output code. Defined using a specific construct, they are similar to pattern-matching logic but apply it to code syntax. Each macro consists of one or more rules, with a pattern to match and a transformation that generates the corresponding output.

These macros are especially useful for repetitive tasks, allowing developers to specify patterns for code generation based on various inputs. They are also generally considered the easiest type of macro to write in Rust.

Here is what a declarative macro might look like. We will create a declarative macro called **vec** that will instantiate a Vector with some initial items already in it.

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

It always starts with a **macro_rules!**, followed by its name and then the body of the macro. They can them be invoked by their name followed by a bang (!) and the parameters like this such:

```
vec![1,2,3];
```

The resulting code after the compilation for this invocation in specific would be:

```
{
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
}
```

2.2.3 Procedural Macros

Procedural macros provide more flexibility by allowing direct manipulation of Rust's Abstract Syntax Tree (AST). They are functions that take a stream of tokens as input and produce a transformed stream of tokens as output.

This approach of manipulating the Abstract Syntax Tree (AST) at compile-time aligns with concepts from Lisp-like macro systems, as discussed in “Programmable Syntax Macros” (Weise & Cew, 1993). Unlike simple token substitution in languages like C, Rust's macros operate at the syntactic level, enabling safe and powerful code transformations in syntactically rich languages without requiring a separate macro language or manual code templates.

Procedural macros operate on streams of tokens, which represent the basic elements of code, such as keywords, identifiers, and punctuation. These tokens are parsed from the code and passed to the macros for manipulation. The macro then generates a new set of tokens that replaces or extends the original code.

This process allows macros to interact with the underlying structure of the code at a very low level, enabling complex transformations and code generation.

Procedural macros are categorized into three types: Derive macros, Attribute macros and Function-like macros. The following sections will describe each of them.

2.2.3.1 Derive Macros

Derive macros automate the implementation of traits for data types such as structs or enums. They are particularly useful for standard traits like **Clone** or **Debug**, where the implementation follows predictable patterns.

As an example of a derive macro, we will create one that automatically implements **Debug** and **Serialize** (from the crate **serde**) for a struct, allowing it to be printed in both regular and JSON formats.

To implement a derive macro, we need to use the **proc_macro_derive** macro.

```
#[proc_macro_derive(JsonDebug)]
```

```
pub fn json_debug_derive(input: TokenStream) -> TokenStream {
    let ast = parse_macro_input!(input as DeriveInput);
    let name = &ast.ident;

    let expanded = quote! {
        impl std::fmt::Debug for #name {
            fn fmt(&self, f: &mut std::fmt::Formatter) ->
std::fmt::Result {
                write!(f, "{}",
serde_json::to_string_pretty(self).unwrap_or_else(|_| "Error
serializing to JSON".to_string()))
            }
        }

        impl #name {
            pub fn to_json(&self) -> String {

serde_json::to_string_pretty(self).unwrap_or_else(|_|
"{}".to_string())
            }
        }
    };

    TokenStream::from(expanded)
}
```

We can then use this macro in the **#[derive(...)]** of a struct.

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

fn main() {
    let user = User {
        name: "Alice".to_string(),
        age: 30,
    };

    println!("{:?}", user); // Prints JSON format using Debug
```

```
    println!("{}", user.to_json()); // Explicitly prints JSON
format
}
```

2.2.3.2 Attribute Macros

Attribute macros define custom attributes that can be attached to various items in the code, such as functions, structs, or modules. These macros can modify the behavior of the items or generate additional code based on the provided attributes.

A great example of how powerful these procedural macros can be is manifested in an attribute macro called **tokio::main**, from the **tokio** crate. This one line of macro changes the entire runtime to one built for multi-threading.

```
#[tokio::main]
async fn main() {
  // ...
}
```

Attribute macros are defined with a **#[proc_macro_attribute]**. Here is an example of a macro that automatically logs when a function starts and ends execution, along with its runtime.

```
#[proc_macro_attribute]
pub fn log_execution(_attr: TokenStream, item: TokenStream) ->
TokenStream {
    let input = parse_macro_input!(item as ItemFn);
    let func_name = &input.sig.ident;
    let block = &input.block;
    let inputs = &input.sig.inputs;
    let output = &input.sig.output;

    let expanded = quote! {
        fn #func_name(#inputs) #output {
            use std::time::Instant;
           println!("Starting function: {}",
```

```
stringify!(#func_name));
            let start = Instant::now();

            let result = (|| #block)();

            let duration = start.elapsed();
            println!("Finished function: {} (Execution time: {:?})",
stringify!(#func_name), duration);

            result
          }
    };

    TokenStream::from(expanded)
}
```

It essentially adds some code before and after the function block, without caring what the function does at all, which makes it highly reusable. All we need to do is attach it to a function using it as an attribute.

```
#[log_execution]
fn compute_sum(n: u32) -> u32 {
    (1..=n).sum()
}
```

Now whenever **compute_sum** is called, we will have a log similar to this:

```
Starting function: compute_sum
Finished function: compute_sum (Execution time: 2.3ms)
```

2.2.3.3 Function-like Macros

Function-like macros resemble function calls but operate at the syntax level, taking a series of tokens as input and generating corresponding output. They are often used to create custom syntax constructs or to simplify repetitive code patterns.

We will later see that function-like macros are the ones least used. That is because it can be replaced by a declarative macro – which is easier to write – most of the time. Still, it has its niches.

As an example of a function-like macro, we will create a macro that wraps an expression and measures its execution time, printing the duration. To define it we use **#[proc_macro]**.

```
#[proc_macro]
pub fn measure_time(input: TokenStream) -> TokenStream {
    let expr = parse_macro_input!(input as syn::Expr);

    let expanded = quote! {
        {
            use std::time::Instant;
           let start = Instant::now();
           let result = { #expr };
           let duration = start.elapsed();
           println!("Execution time: {:?}", duration);
           result
        }
    };

    TokenStream::from(expanded)
}
```

It can then be invoked similarly to a declarative macro:

```
fn main() {
    let sum = measure_time!({
        let mut total = 0;
        for i in 0..1_000_000 {
            total += i;
        }
        total
    });

    println!("Result: {}", sum);
}
```

2.2.4 Hygiene in Macros

Macro hygiene refers to the scoping rules that ensure macros do not unintentionally interfere with the surrounding code. In Rust:

- **Declarative Macros**: Automatically manage scope and avoid naming conflicts by keeping track of where identifiers are introduced. However, when a macro needs to refer to items within the defining crate, the **$crate** metavariable can be used to ensure proper resolution of those items.
- **Procedural Macros**: Provide more flexibility but require careful handling of scope. Although procedural macros have more control over naming and scope management, Rust's **proc_macro** API allows developers to manage hygiene explicitly.

Although declarative macros manage conflicts automatically, the **$crate** metavariable is an important tool to ensure correct references to external items, especially when used across crates. Rust's hygiene system helps generate unique names for internal variables or functions, preventing unintentional naming conflicts.

Rust's macro system also benefits from clear phase separation between compile-time and runtime, similar to the approach taken by MzScheme as detailed in "Composable and compilable macros: you want it when?" (Flat, 2002). This design choice helps prevent the mingling of compile-time and runtime values, enhancing the safety and tooling support of the language.

2.2.5 Advantages of Using Macros in Rust

- **Reduced Code Duplication**: Macros can automatically generate repetitive code patterns, minimizing manual repetition.
- **Enhanced Abstraction**: They enable developers to create higher-level abstractions that are difficult to achieve with functions alone.
- **Compile-Time Optimization**: Macros allow for code generation and transformation at compile time, potentially improving runtime performance.
- **Conditional Compilation**: Macros can selectively include or exclude code based on compile-time conditions, which is useful for platform-specific or feature-dependent code.

2.2.6 Comparison with Macros in Other Languages

Macros are present in several programming languages, but their capabilities and limitations vary significantly. In most cases, macros provide some level of code generation, but they differ in terms of safety and expressiveness. Below is a comparison of how macros work in different languages:

- **C/C++ Preprocessor Macros (#define)**: Expanded at preprocessing time, they lack scope awareness and type safety. They perform simple textual substitution, which can lead to unintended side effects and debugging difficulties.
- **Rust Macros (Declarative Macros, Procedural Macros)**: Rust macros are expanded at **compile-time** while maintaining full type safety. Unlike C macros, they operate on structured syntax (AST), preventing common pitfalls like unintentional variable substitution or lack of scoping.

Compared to C/C++, Rust's macro system is more robust and secure, allowing complex code transformations while enforcing compile-time correctness.

2.2.7 Practical Use Cases

Rust macros are employed in a wide range of applications, including:

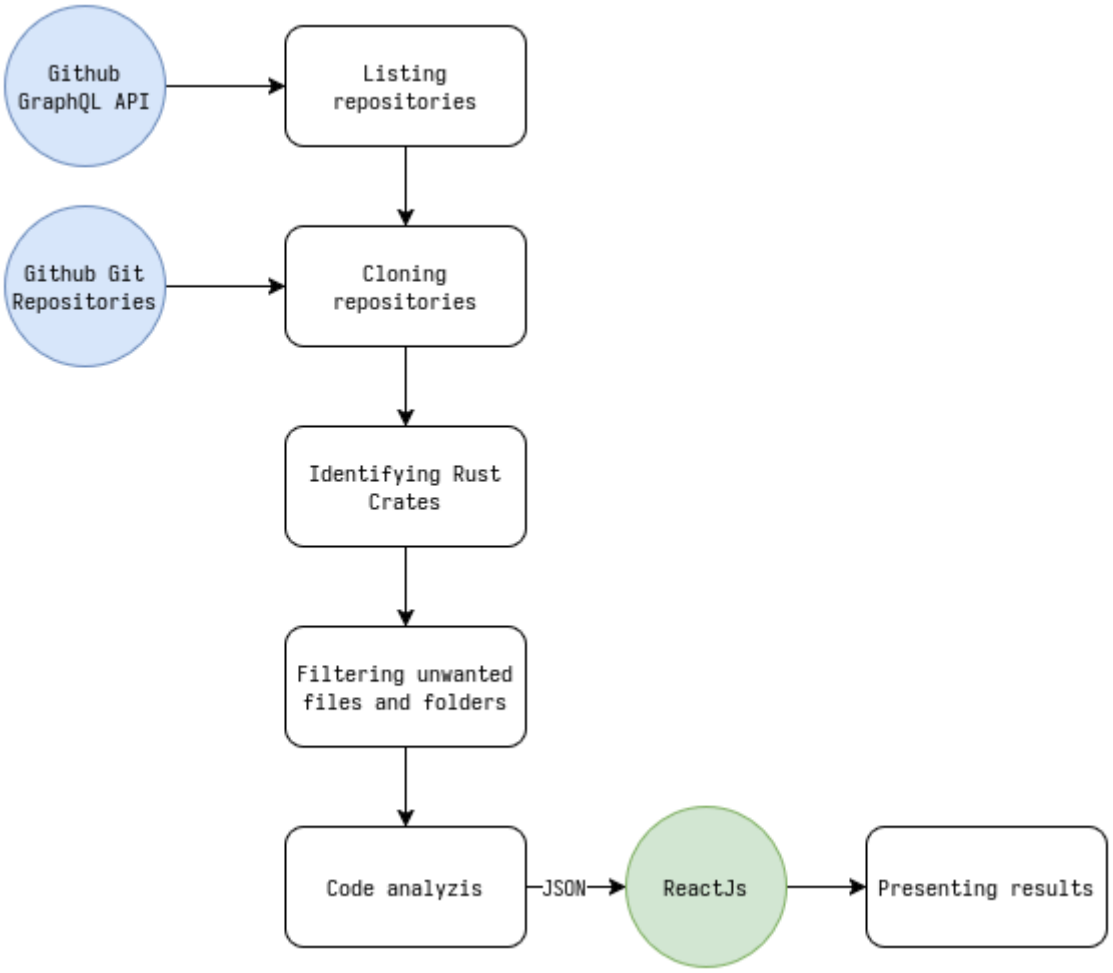
- **Automatic Trait Implementations**: Macros are commonly used to implement traits like **Debug** or **Clone** for custom types, saving developers the effort of writing repetitive code.
- **Domain-Specific Languages (DSLs)**: Macros are frequently used to create custom syntax or DSLs for specific purposes, such as testing frameworks or concurrency abstractions.
- **Boilerplate Code Generation**: Macros can generate common code patterns, such as error handling or logging, reducing the amount of code developers need to write manually.

By understanding the key features and advantages of Rust's macro system, we gain insights into how they are used across real-world projects. In the following chapters, we will explore how macros are applied in practice within the Rust community, examining usage patterns and best practices among popular GitHub repositories.

3. METHODOLOGY

This chapter outlines the methodology used to scrape and analyze macro usage in the 100 most popular Rust repositories on GitHub. The process was divided into six key stages: listing repositories, cloning them locally, identifying Rust projects (crates), filtering unwanted files, analyzing the code, and finally presenting the results in a visual format. All the code used for the tool is available in a github repository: https://github.com/STAR-RG/rust-macro-analyzer

Here is a diagram of the steps taken by the tool.



3.1 Listing Repositories

The first step involved identifying the most popular Rust repositories on GitHub. This was achieved using the GitHub GraphQL API with specific filters to ensure relevance. The filter criteria were:

- Language: The primary programming language of the repository must be Rust.
- Sorting: Repositories were sorted in descending order based on the number of stars to prioritize those with the highest popularity.
- First 100: We limited this study to the 100 most popular repositories only.

This approach provided a curated list of popular Rust projects, ensuring that the analysis focused on widely used, real-world codebases.

3.2 Cloning Repositories

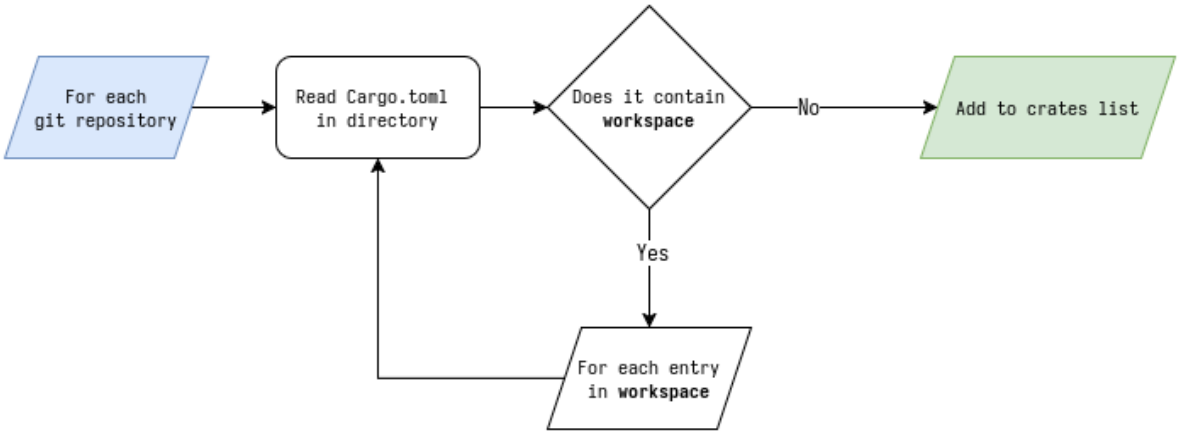
Once the repositories were listed, the next step was to clone them locally. This was done using the **git clone** command, along with the **--recurse-submodules** flag to ensure that any submodules within the repositories were also included. Submodules are often used in large-scale projects to include dependencies, so capturing them was essential for a comprehensive analysis.

3.3 Identifying Rust Crates

In Rust, a project that can be compiled is referred to as a *crate*. A crate is defined by the presence of a **Cargo.toml** file, which contains metadata about the project, including dependencies and compilation instructions.

Since a GitHub repository can contain multiple crates (sub-projects), the next step was to identify all crates within the cloned repositories. Although a crate is defined by the presence of a **Cargo.toml**, a **Cargo.toml** doesn't necessarily imply a crate. This is because a **Cargo.toml** with the field **workspace** indicates that the folder is not of a crate, but of a workspace (collection) of crates.

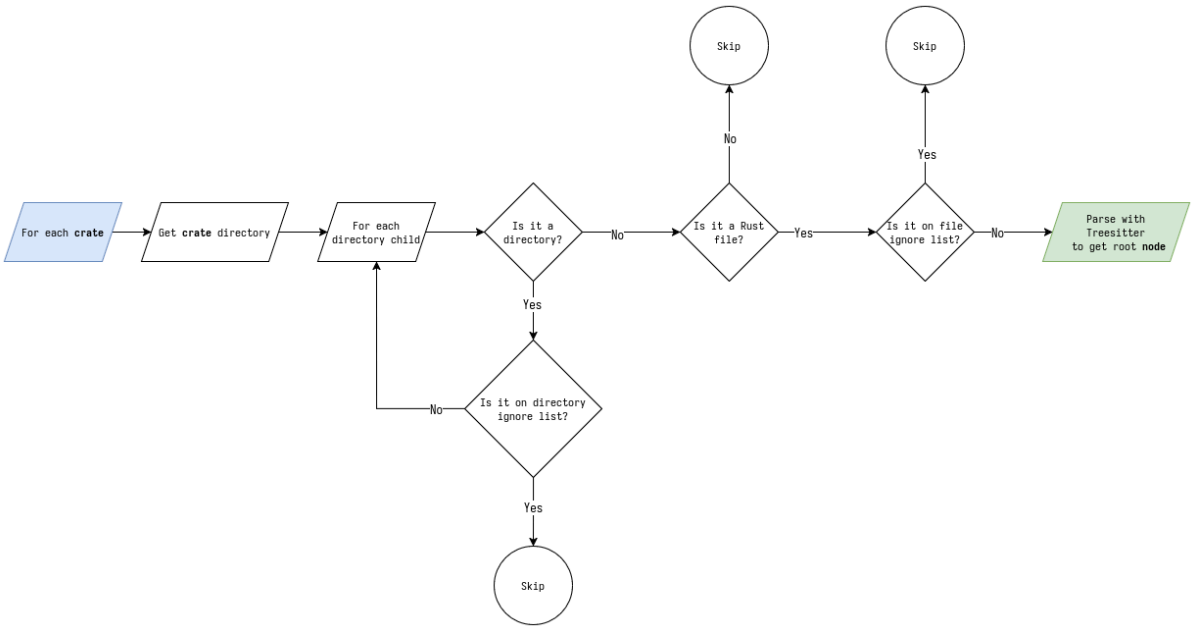
To properly identify all the crates, we followed the following flow. With it, we were able to properly list all crates.



3.4 Filtering Unwanted Files

Before beginning the core analysis, we needed to filter which files to parse with Treesitter. We aimed to include every Rust file in each crate, excluding those listed in an ignore list. This ignore list was created to bypass intentionally malformed Rust code, which some projects use for showcasing or testing purposes.

Instead of identifying all the correct Rust files before starting the analysis, we adopted a streamlined approach: as soon as a file was identified, we immediately parsed its code with Treesitter.



3.5 Code Analysis

For each Rust file that passed the previous stage's filtering, we parse it with Treesitter, a tool that constructs a syntactic tree from source code files. Here is an example of Rust code and it's corresponding syntactic tree parsed by Treesitter:

```rust
fn main() {
    println!("Hello, world!");
}
```

Corresponding syntactic tree:

```
source_file [0, 0] - [4, 0]
 function_item [0, 0] - [2, 1]
   name: identifier [0, 3] - [0, 7]
   parameters: parameters [0, 7] - [0, 9]
   body: block [0, 10] - [2, 1]
     expression_statement [1, 4] - [1, 28]
       macro_invocation [1, 4] - [1, 27]
         macro: identifier [1, 4] - [1, 11]
         token_tree [1, 12] - [1, 27]
           string_literal [1, 13] - [1, 26]
             string_content [1, 14] - [1, 25]
```

For each file, a tree is generated. We then iterated through the tree from the root node downward, identifying macro invocations, macro definitions, and registering their occurrences.

Treesitter enabled the extraction of specific information regarding macro usage, including:

- **Macro Definitions**: The presence and usage of custom-defined macros within the codebase.
- **Macro Invocations**: The frequency and type of macros invoked within the code.
    - **Derive Macros**: For derive macro invocations, additional details, such as the number of traits passed to the macro, were also collected.

- **Attribute Macros**: For attribute macro invocations, we were able to immediately identify if they were a built-in macro or an user defined on.e

This structured approach provided a granular view of macro usage across different types and projects.

This need for parsing without preprocessing parallels the challenges faced in languages like C, where conditional compilation and lexical macros complicate automatic analysis. The “Variability-aware parsing in the presence of lexical macros and conditional compilation” (Flat, 2002) paper discusses these issues and introduces strategies, such as those in the TypeChef project, to parse code variability soundly and efficiently—issues that Rust also faces in macro-heavy codebases.

3.6 Presenting Results

After collecting the data, the results were serialized into a JSON format file. A web interface that was developed in React reads from this file to show the data. The front-end used Recharts and ApexCharts libraries to render graphs, offering an interactive representation of the macro usage statistics. These visualizations allowed for easy interpretation of patterns, trends, and outliers in macro usage across the analyzed Rust repositories.

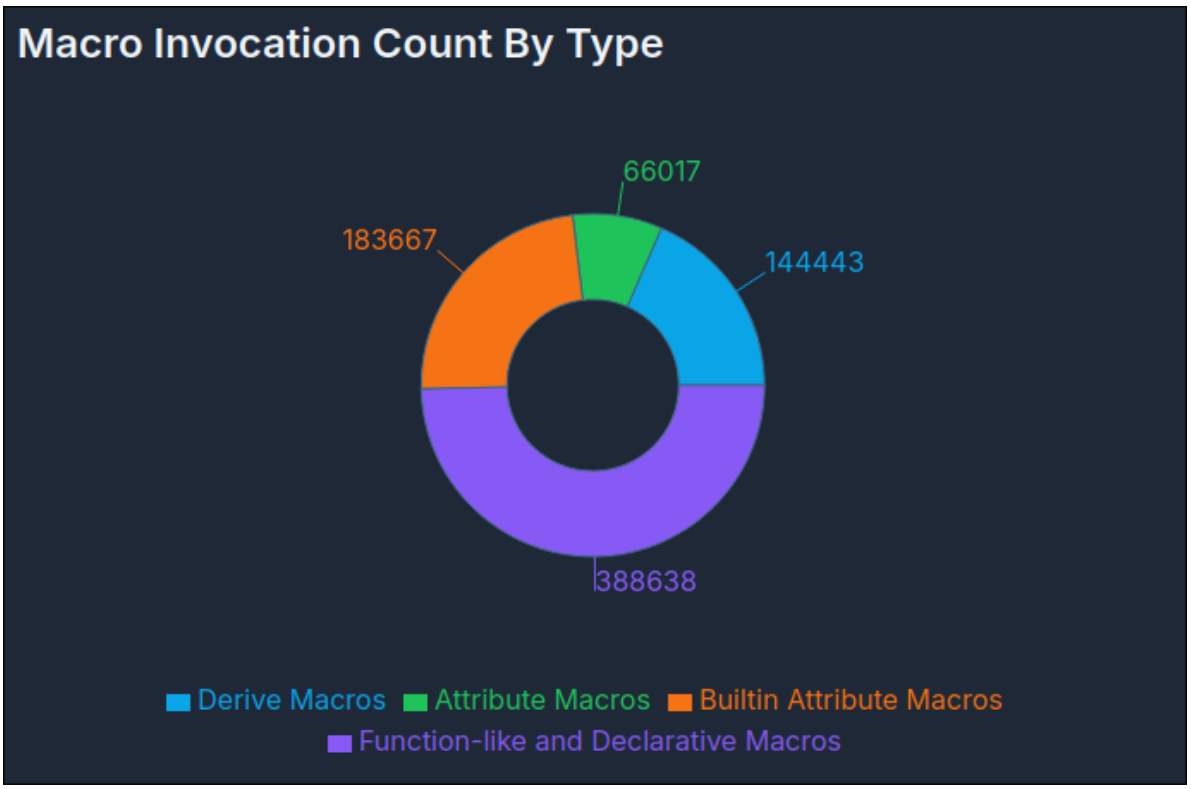
This structured methodology ensured that the analysis was comprehensive, capturing the intricacies of macro usage in large-scale Rust projects. The following chapters will discuss the findings in detail, presenting the results of the macro usage analysis across different types of Rust projects.

4. RESULTS AND ANALYSIS

This chapter presents the results of the analysis on macro usage in the 100 most popular Rust repositories hosted on GitHub. The study examined a total of 1893 crates within these repositories, uncovering patterns in macro invocations and definitions, categorized by macro type and their frequency across projects.

4.1 Macro Invocation Patterns

Across the analyzed repositories, a total of 782,765 macro invocations were recorded, broken down into several categories, including derive macros, attribute macros (both user-made and built-in), and function-like or declarative macros. The distribution of macro invocation counts by type can be seen in Macro Invocation Count by Type chart.



The majority of macro invocations belong to function-like and declarative macros, with a total of 388,638 invocations, making up approximately 49.6% of the total. Unfortunately, Treesitter cannot differentiate between the two, as their invocation method is identical. Moving on, Built-in attribute macros account for 183,667

invocations, or about 23.5%, followed by derive macros with 144,443 invocations (18.4%) and user-made attribute macros with 66,017 invocations (8.4%).

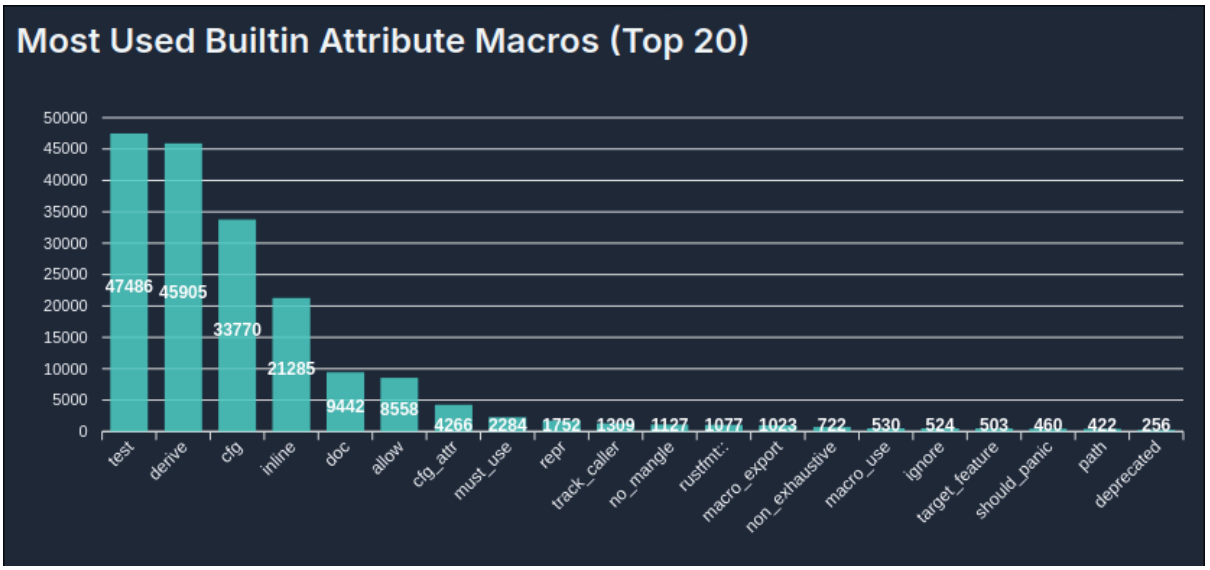
The prevalence of function-like and declarative macros suggests that developers in large Rust projects heavily rely on macros that perform more complex, reusable code operations. Built-in attribute macros are also significant, and we will see why in the following section.

4.1.1 Most Used Built-In Attribute Macros

Among the built-in attribute macros, the top six were as follows:

- **test**: 47,486 invocations
- **derive**: 45,905 invocations
- **cfg**: 33,770 invocations
- **inline**: 21,285 invocations
- **doc**: 9,442 invocations
- **allow**: 8,558 invocations

The **test** macro's prominence indicates the widespread use of Rust's built-in unit testing framework. The **derive** macro being so heavily used is not a surprise, as it is essential for type derivation and polymorphism in Rust. Meanwhile, **cfg** allows for platform-specific code and conditional compilation, both of which are essential for cross-platform development. Not too far off, the **inline** macro is used for performance optimization, which reinforces the usage of Rust for performance critical applications.

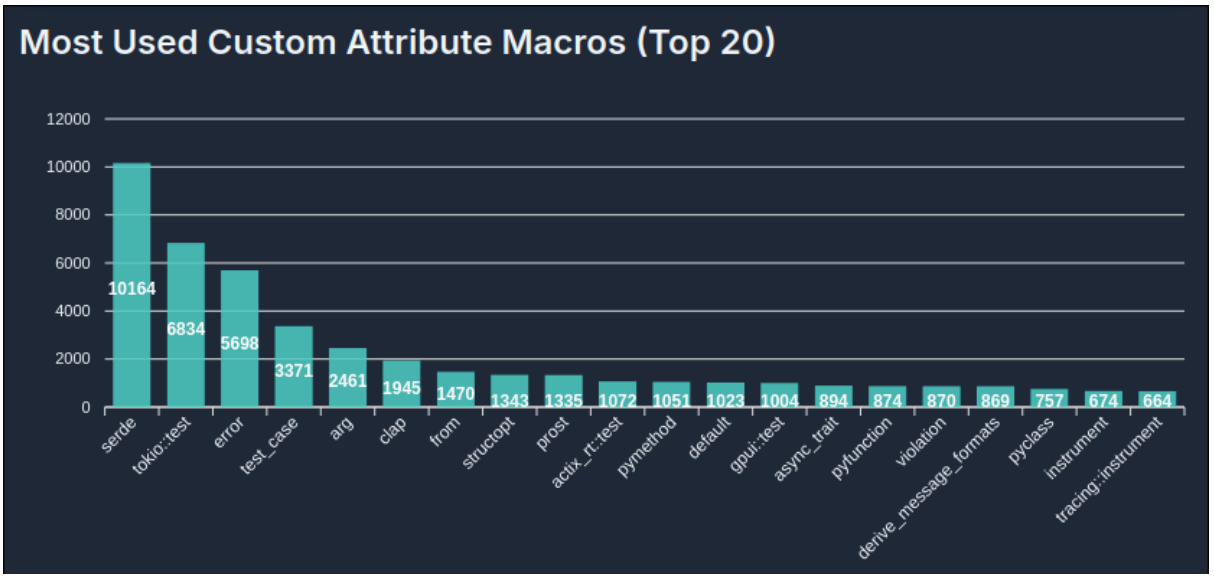


4.1.2 Most Used User-Made Attribute Macros

In addition to built-in macros, user-made attribute macros were analyzed. The following user-made macros appeared most frequently:

- **serde**: 10,164 invocations
- **tokio::test**: 6,834 invocations
- **error**: 5,698 invocations
- **test_case**: 3,371 invocations
- **arg**: 2,461 invocations
- **clap**: 1,945 invocations

The dominance of the **serde** macro indicates the high reliance on serialization and deserialization tasks, a critical part of many Rust projects, especially those involving data exchange and APIs. Similarly, the frequent use of **tokio::test** reinforces the widespread use of unit testing, but also reflects the increasing popularity of asynchronous programming in Rust, where **tokio** is a widely adopted runtime. The **error** macro is a utility helper for creating error types from one of the most popular Rust crates called **thiserror**.



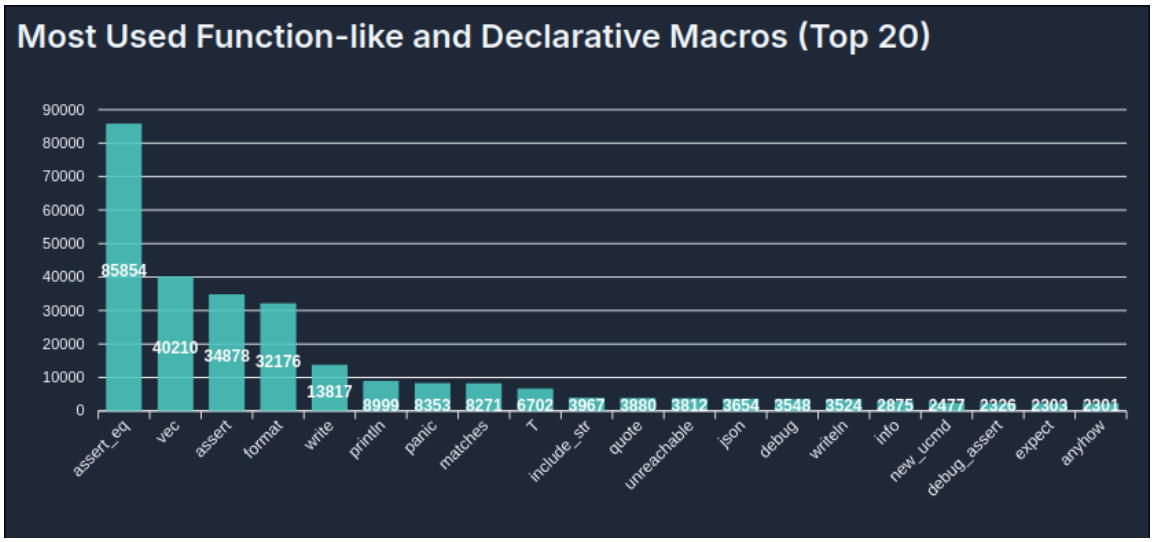
4.2 Function-Like and Declarative Macros

Function-like and declarative macros, as noted earlier, represent the largest category of macro invocations in the dataset. Among them, the seven most commonly used were built in macros:

- **assert_eq**: 85,854 invocations
- **vec**: 40,210 invocations
- **assert**: 34,878 invocations
- **format**: 32,176 invocations
- **write**: 13,817 invocations
- **println:** 8,999 invocations
- **panic**: 8,353 invocations

The **assert_eq** and **assert** macros are frequently used for debugging and test cases, suggesting that they are an integral part of code validation processes. The high usage of **format** and **write** come as no surprise as they are key in manipulating strings in Rust. Similarly, the **vec** macro, although technically a simple macro for initializing **Vectors** (dynamic arrays), being so common indicates how common the usage of **Vectors** is.

The **T** macro is used inside the compiler, and it's an utilitary for token. One of the most used custom macros was **json!**, which is used to write json inside Rust, and is provided by **serde**. Another group of frequently used custom macros is the ones used for logging, such as **debug!**, **info!**, and **warn!**.

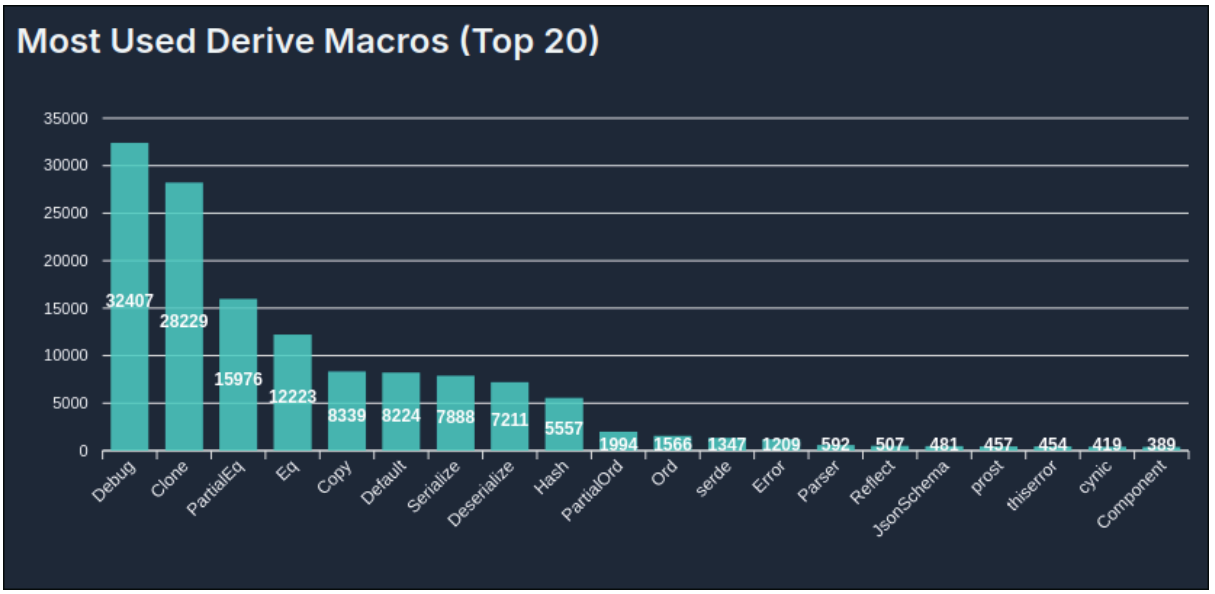


4.3 Derive Macros

Derive macros, which simplify the implementation of common traits, were invoked a total of 144,443 times, in a total of 45,905 different **#[derive(...)]** calls. The most used derive macros are:

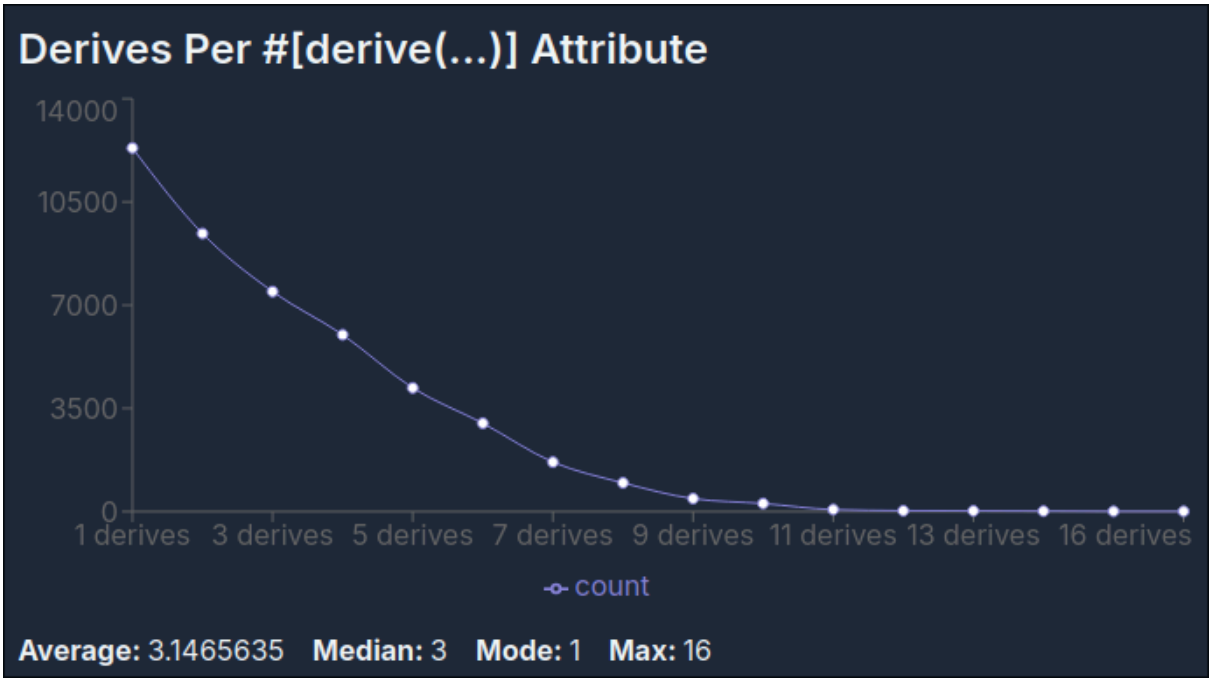
- **Debug**: 32,407 invocations
- **Clone**: 28,229 invocations
- **PartialEq**: 15,976 invocations
- **Eq**: 12,223 invocations
- **Copy**: 8,339 invocations
- **Default**: 8,224 invocations
- **Serialize**: 7,888 invocations
- **Deserialize**: 7,211 invocations

The first six are built in derive macros, used to implement common traits in Rust, such as **Clone** and **Copy**, which are crucial traits for dealing with Rust's ownership system. **Debug** being the most used is natural, as it is standard for debugging in Rust. Additionally, the **Serialize** and **Deserialize** macros from **serde** indicate that Rust is often used from projects that communicate with other projects such as server API and need to handle data serialization and deserialization.



4.3.1 Derives per derive macro

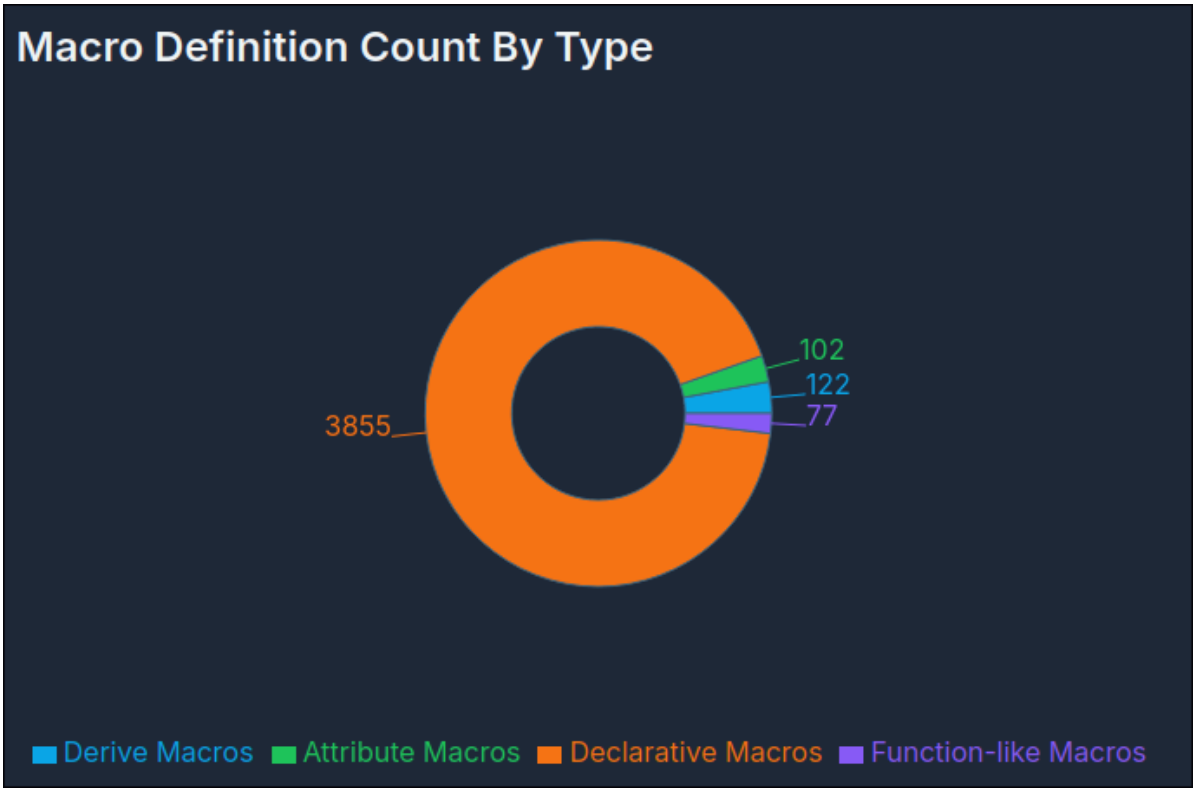
An interesting piece of information we were able to extract from derive macros is the amount of derives used inside a **#[derive(...)]** macro. The most common amount of derives is a single derive, only decreasing from there. The median is 3 derives, which is close to the average, which is 3.14.



4.4 Macro Definition Patterns

In total, 4,156 macro definitions were identified across the 1893 crates analyzed. The distribution of macro definitions by type is as follows:

- **Declarative macros**: 3,855 definitions
- **Attribute macros**: 102 definitions
- **Derive macros**: 122 definitions
- **Function-like macros**: 77 definitions



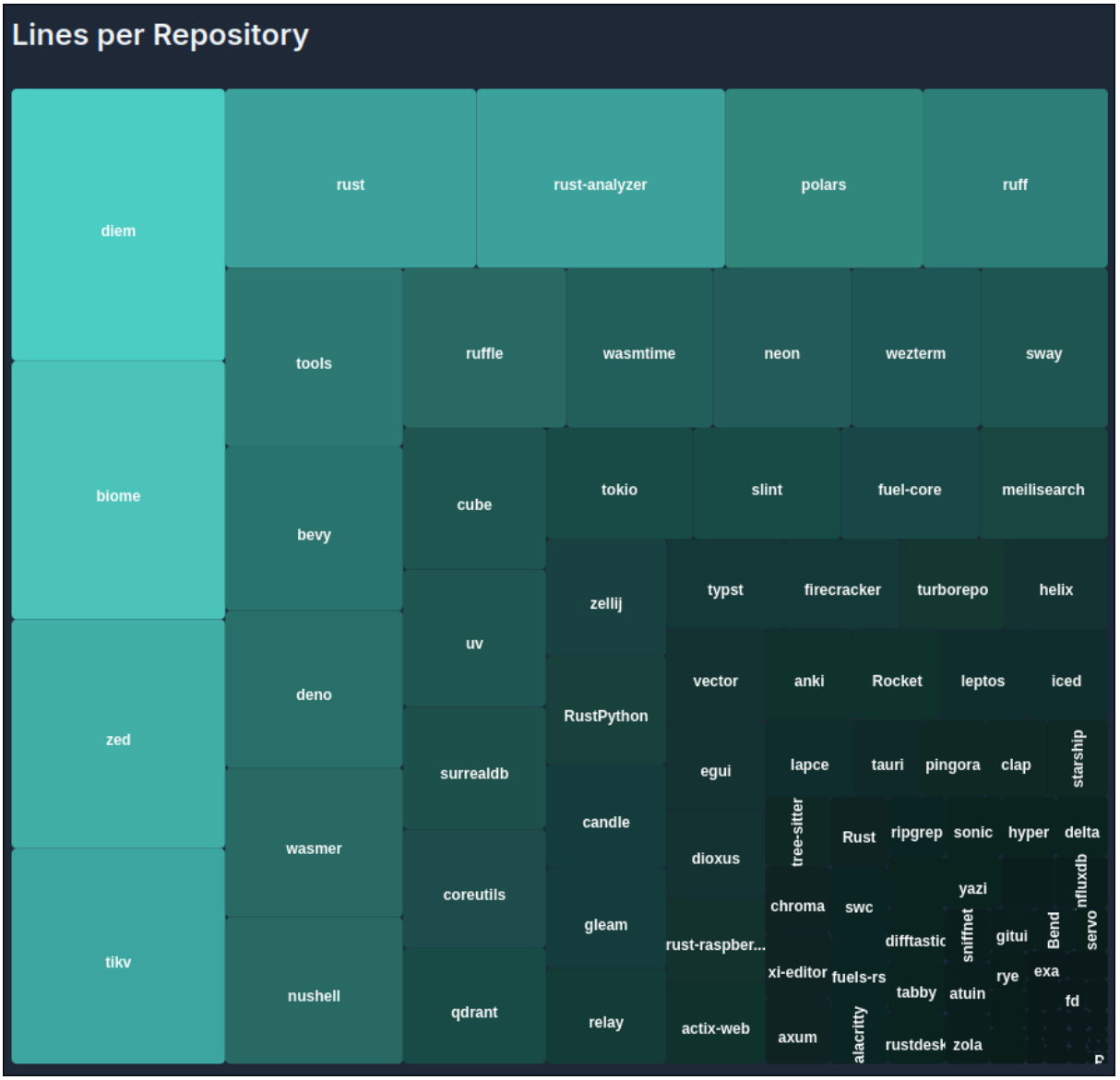
Declarative macros account for the vast majority of definitions, suggesting that the main use for projects is to create their own specialized macros to automate repetitive tasks. For some use cases you could technically use other types of macros, but this overwhelming dominance highlights the ease of use of Declarative macros that rely on easy to understand match-like syntax, in contrast with Procedural macros that require manipulating the AST.

Procedural macros, although more powerful are harder to work with and are likely reserved for complex cases that usually become their own crate and shared, so each user doesn't have to come up with it again on their own.

4.5 Macro Usage by Repository

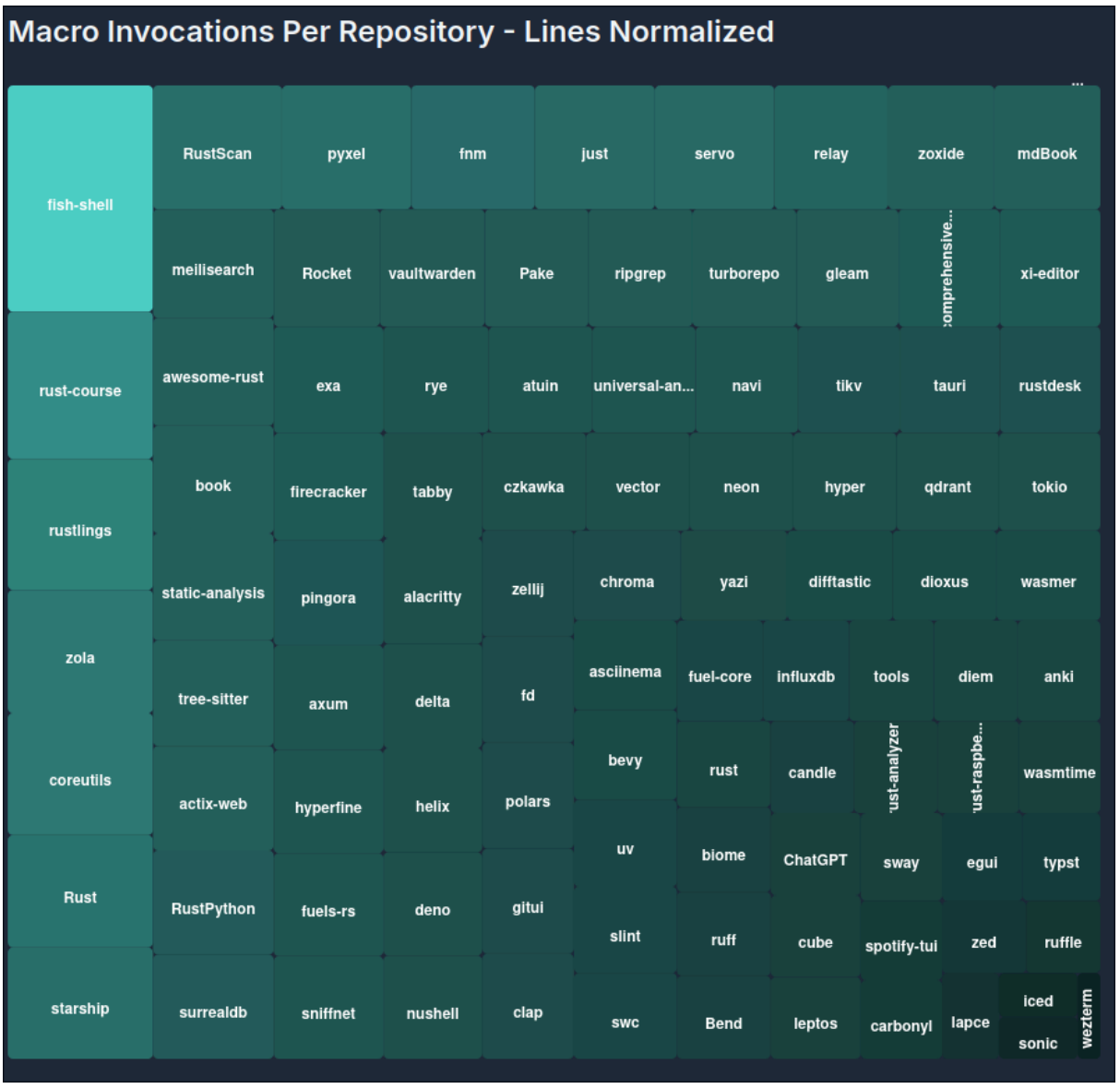
The distribution of macro invocations and definitions per repository on its own doesn't reveal much as the size of the repository affects the numbers a lot. To counteract that, we have divided the invocations and definitions by lines of code, which we called "Lines Normalized".

As we can see from the Lines per Repository chart, the sizes of these repositories vary a lot. The biggest one being **diem/diem** with 549904 lines, while the smallest, **casey/just**, is only 293 lines long.



4.5.1 Macro Invocations per Repository

Starting with invocations we can see 2 categories of projects on the top.



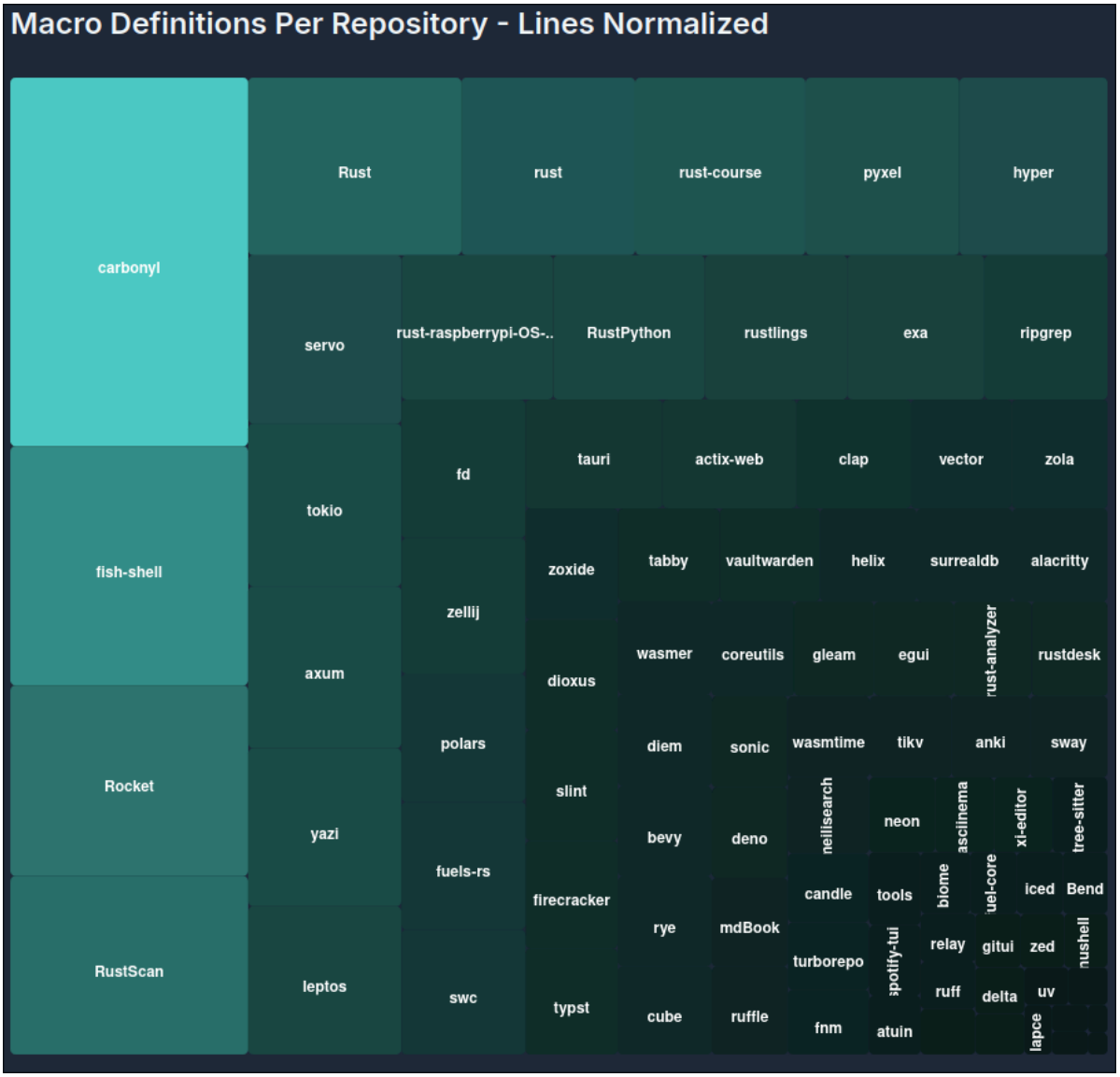
The first category is courses and tutorials. We have **sunface/rust-course**, **rust-lang/rustling**, and **TheAlgorithms/Rust** which are repositories dedicated for studying materia, occupying second, third and sixth place respectively. With the normalization by lines, it makes sense that they would appear here, as their purpose is to highlight Rust's features, meaning both that they feature a variety of macros but also with a low amount of lines of code, as the code is mainly composed of example codes.

The second category is composed of projects where performance is crucial, that generally deal with lower-level integrations and most of them are extensible. On the top of this category we have:

- **fish-shell**: a complete and extensible command-line shell.
- **zola**: a static site generator that uses “fast” as its main word for advertisement.
- **coreutils**: a cross-platform implementation of GNU utils.
- **Starship:** a highly extensible cross-shell prompt.

4.5.2 Macro Definitions per Repository

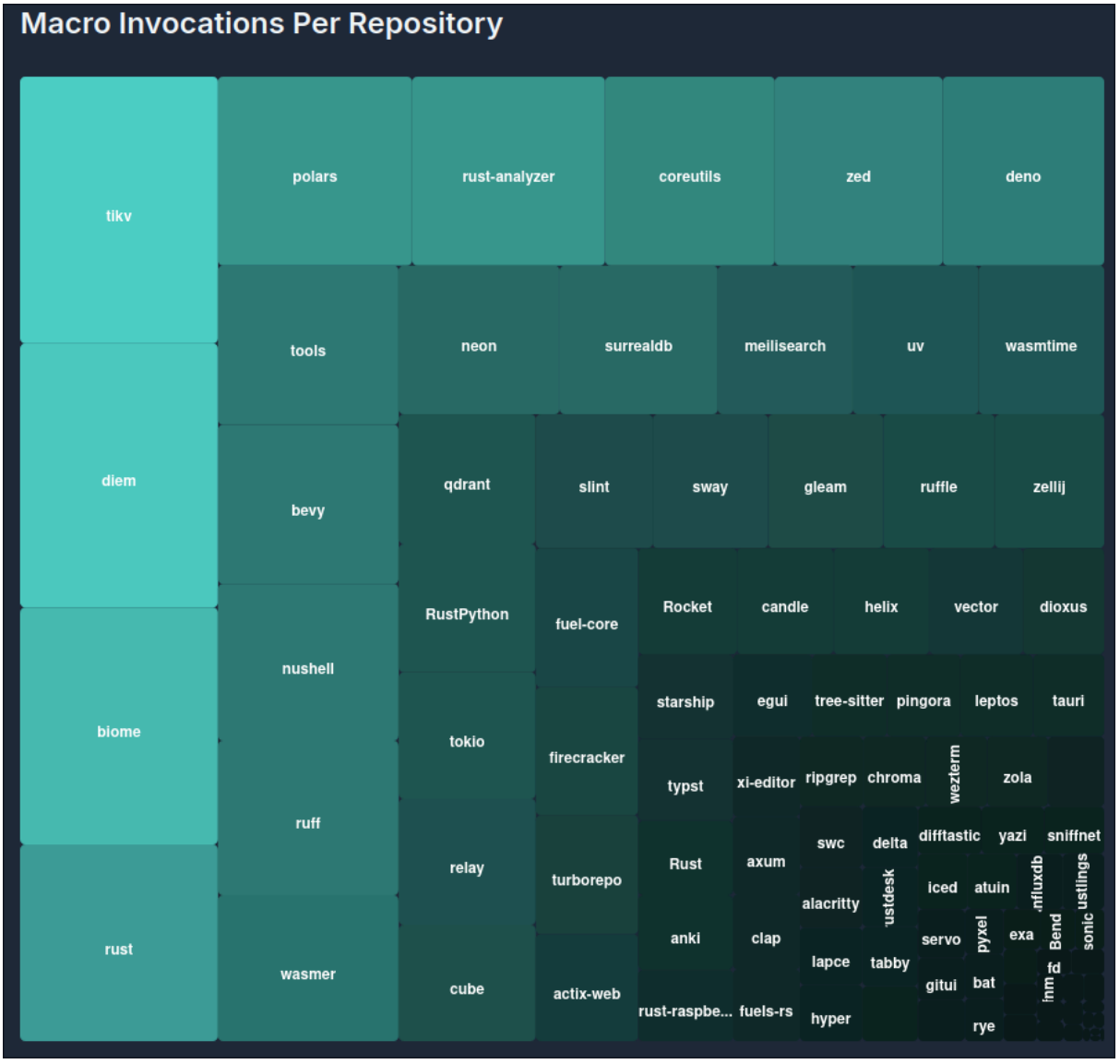
When it comes to macro definitions the results are not too unlike invocations. We can still see the courses **sunface/rust-course** and **TheAlgorithms/Rust**, as well as **fish-shell**, **pyxel** and **RustScan** that were also on the top of invocations. In first place we have **fathyb**/**cabonyl**, an ambitious project of a Chromium based browser running inside the terminal.



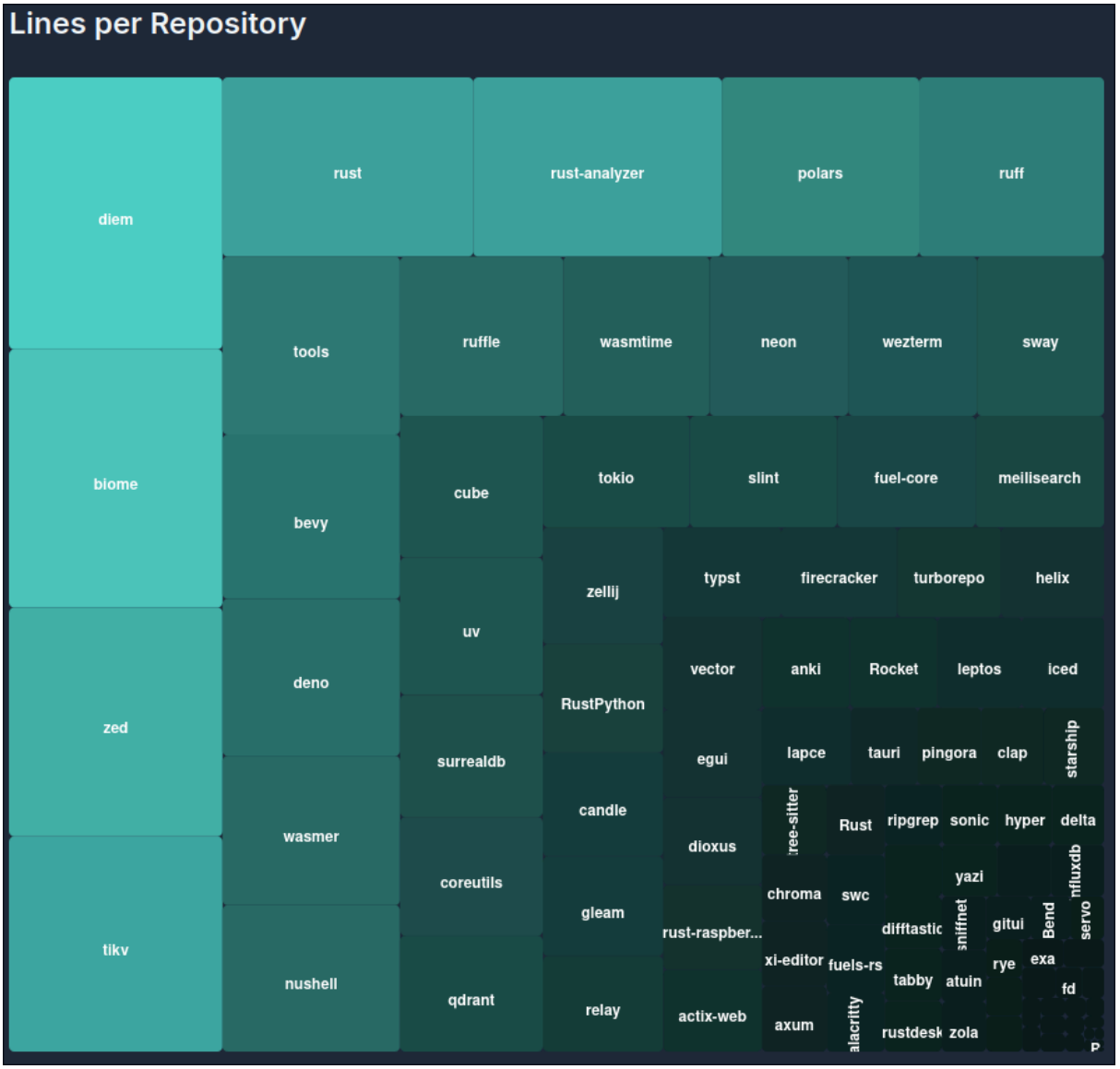
Not too far we have the repository for the Rust language itself. Which is interesting because it didn't appear on the top invocations per repository when normalized due to its size. We can see that even with its size it managed to get sixth place on

definitions normalized by virtue of hosting Rust's builtin macros. This can be explained by looking at the raw data, that is, without normalizing by lines:

The Rust language repository comes in first with 598 different macro definitions, followed by **polars** with only 242. The same discrepancy does not appear on invocations per repository:



Which seems to be way more proportional to the size of the repository itself:



Which suggests that invoking macros is standard practice for large projects, while defining macros vary a lot according to the project context and goals.

5. CONCLUSION AND FUTURE RESEARCH

This thesis set out to analyze the usage of macros in the 100 most popular open-source Rust projects on GitHub, focusing on uncovering trends and identifying the most commonly used types of macros, projects that heavily rely on macros, and specific macro categories. Through an extensive analysis of 1893 crates and over 780,000 macro invocations, several key findings were revealed, shedding light on how macros are integrated into large-scale Rust development.

5.1 Summary of Findings

The analysis provided insights into the diverse ways macros are utilized in the Rust ecosystem. Several important trends were identified:

- The widespread use of testing related macros, such as **#[test]**, further indicates the Rust community's focus on code quality and testing, making the testing framework a cornerstone in most Rust projects.
- Declarative macros stood out not only in usage but in sheer definition count, vastly outnumbering procedural macros. This suggests that developers prioritize ease of implementation when creating project-specific tooling. While procedural macros offer greater flexibility, their complexity appears to limit their use to more sophisticated or reusable libraries
- The frequent use of attribute macros from popular crates like **tokio** and **serde** points to a strong ecosystem-wide preference for macro-based extensibility in both asynchronous and data-centric applications. the areas of web services, asynchronous programming, and data processing
- Although macros in Rust are broadly used, some types of projects appear to require more of them than others. Notably, performance sensitive applications for optimization and reusable applications for abstractions.

Overall, the analysis demonstrated that macros are not only fundamental to Rust's metaprogramming features but also widely adopted across a variety of project types, even more on high-performance system libraries.

5.2 Implications for Developers

The insights gained from this study suggest several actionable takeaways:

- **Strategic Use of Declarative Macros**: Developers can confidently adopt declarative macros for internal project tooling and abstraction, especially where performance and maintainability intersect.
- **Adopting Ecosystem Standards**: The popularity of macros from libraries like **serde**, **tokio**, and **thiserror** reflects community consensus around certain tasks—serialization, async handling, and error management. New projects benefit from aligning with these conventions.
- **Awareness of Complexity Trade-offs**: While procedural macros unlock powerful capabilities, their lower adoption suggests they should be used judiciously—ideally when general-purpose solutions are intended to be shared across projects or crates.
- **Domain-Specific Macro Strategies**: Developers building educational tools, compilers, or high-performance libraries may find distinct value in different macro styles. Understanding the macro landscape can inform architectural choices.

5.3 Limitations of the Study

While this thesis provides valuable insights into macro usage trends, it is important to acknowledge its limitations:

- **Sample Bias**: The focus on the most starred projects may skew results toward well-documented or community-driven codebases, underrepresenting niche, experimental, or private repositories.
- **Parsing Constraints**: The use of TreeSitter enabled robust syntax analysis but imposed restrictions. Notably, it cannot distinguish declarative from procedural macros at the invocation level, and it does not resolve macro paths, potentially conflating similarly named macros from different sources.
- **Lack of Semantic Context**: While macro invocations were tracked, this analysis does not account for the semantic role or impact of those macros, which would require deeper integration with Rust's type system or compiler internals.

5.4 Directions for Future Research

While this study provides a broad overview of macro usage in popular Rust projects, several areas merit further exploration:

1. **Macro Impact Analysis**: Future work could quantify how macro usage affects compilation speed, runtime performance, and binary size—key factors in systems programming.
2. **Temporal Trends**: A longitudinal study could explore how macro usage evolves over time within projects, offering insight into adoption curves, refactoring trends, or deprecation patterns.
3. **Cross-Language Comparison**: Comparing macro systems in Rust with those in C++, Zig, or Nim could reveal differences in philosophy, ergonomics, and safety, benefiting developers transitioning between ecosystems.
4. **Readability and Maintenance**: Macros often obscure control flow or logic. Research into how they affect code comprehension, onboarding, and bug rates would be valuable for teams adopting macro-heavy codebases.

BIBLIOGRAPHY

FLAT, M. Composable and compilable macros: you want it when. *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, *37*(9), 72-83, 2002.

KASTNER, C. Variability-aware parsing in the presence of lexical macros and conditional compilation. *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, *46*(10), 805-824, 2011.

KEEP, D. *The Little Book of Rust Macros*. 2016. https://veykril.github.io/tlborm/ (retrieved October 3, 202).

KLABNIK, S.; NICHOLS, C.; KRYCHO, C. *The Rust Programming Language*. 2018. https://doc.rust-lang.org/book (retrieved September 17, 2024)

THE RUST TEAM. *The Rust Reference*. (s.d.). https://doc.rust-lang.org/reference/ (retrieved September 19, 2024).

WEISE, D.; CREW, R. Programmable syntax macros. *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, *28*(6), 156-165, 1993.