Universidade Federal de Pernambuco
Centro de Informática
Graduação em Ciência da Computação

Sofia Melo de Lucena

Análise Estática da Propagação de Erros em Rust: Um Estudo Expandido

2025

Sofia Melo de Lucena

Análise Estática da Propagação de Erros em Rust: Um Estudo Expandido

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Área de Concentração: Engenharia de Software

Orientador: Kiev Santos da Gama

Coorientador: Fernando José Castor de Lima Filho

2025

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

Lucena, Sofia Melo de.
Análise Estática da Propagação de Erros em Rust: Um Estudo Expandido / Sofia Melo de Lucena. - Recife, 2025.
49 p. : il., tab.

Orientador(a): Kiev Santos da Gama
Cooorientador(a): Fernando José Castor de Lima Filho
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2025.
Inclui referências.

1. Engenharia de Software. 2. Rust. 3. Tratamento de Erros. 4. Propagação de Erros. 5. Qualidade de Software. I. Gama, Kiev Santos da . (Orientação). II. Lima Filho, Fernando José Castor de . (Coorientação). IV. Título.

000 CDD (22.ed.)

SOFIA MELO DE LUCENA

Análise Estática da Propagação de Erros em Rust: Um Estudo Expandido

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: 28/03/2025

BANCA EXAMINADORA

_____

Prof. Dr. Kiev Santos da Gama (Orientador)

Universidade Federal de Pernambuco

_____

Profa. Dra. Fernanda Madeiral Delfim (Examinador Interno)

Universidade Federal de Pernambuco

AGRADECIMENTOS

A conclusão deste trabalho representa um marco importante em minha trajetória acadêmica e profissional, e não poderia deixar de expressar minha gratidão a todos que contribuíram para este momento.

Aos meus pais, meu irmão e Felipo, pelo apoio incondicional, pelos ensinamentos, pela paciência e por acreditarem em mim em cada etapa dessa jornada. Seu incentivo foi essencial para que eu pudesse superar desafios e seguir em frente com determinação.

Aos meus amigos do curso, que compartilharam madrugadas de estudo e trabalho, sempre oferecendo apoio de alguma forma. Em especial, a Alex, cuja presença e incentivo foram fundamentais para que eu me mantivesse firme diante das dificuldades.

Aos meus professores, pelo conhecimento transmitido, pela orientação e pelos desafios propostos, que me permitiram crescer não apenas academicamente, mas também como profissional e como pessoa. Meu agradecimento especial ao meu orientador, Kiev, e ao coorientador, Castor, por toda a ajuda e dedicação nesse processo de finalização do trabalho de graduação.

Ao CIn e à UFPE, pelas oportunidades oferecidas, pelo ambiente de aprendizado e pelas experiências enriquecedoras que contribuíram significativamente para minha formação.

A todos que, de alguma forma, fizeram parte dessa jornada, meu sincero e profundo agradecimento.

”A imaginação é a faculdade da descoberta, predominantemente.
É ela que penetra nos mundos invisíveis que nos rodeiam, nos mundos da ciência.”
- ***ADA LOVELACE***.

RESUMO

Esse trabalho investiga como erros em projetos Rust são propagados e tratados, analisando repositórios de código aberto para identificar padrões e correlações entre métricas. Em particular, analisamos quão comum é a existência de funções que propagam erros e a distância entre a sinalização e o tratamento desses erros. Além disso, examinamos se o nível de maturidade de um projeto influencia a complexidade da propagação e tratamento de erros. A metodologia adotada envolve a extração automatizada de métricas, como a proporção de propagação de erros por função, a profundidade das cadeias de tratamento e a quantidade de erros ignorados, complementada por técnicas de visualização de dados. Os resultados indicam que, em média, 17% das funções no código dos projetos analisados envolvem propagação ou tratamento de erros. A maioria dos erros na amostra de dados é propagada por até quatro chamadas de função antes de ser tratada, com esse número aumentando conforme a complexidade das cadeias, atingindo um máximo de 12 chamadas nas amostras analisadas. Projetos menos maduros não apresentam diferenças significativas no tratamento de erros quando comparados a projetos mais maduros. Por fim, a investigação revelou padrões relevantes sobre o tratamento de erros em Rust, incluindo práticas comuns no uso das funções **expect** e **unwrap**, a substituição de mensagens de erro que compromete a sua rastreabilidade e o uso de erros dinâmicos em executáveis CLI. A principal contribuição deste trabalho é fornecer uma análise do tratamento de erros na linguagem, auxiliando na compreensão das práticas adotadas pela comunidade e oferecendo insights para futuras pesquisas sobre qualidade e segurança do software em Rust.

Palavras-chaves: Rust, Tratamento de Erros, Propagação de Erros, Qualidade de Software

ABSTRACT

This study investigates how errors in Rust projects are propagated and handled by analyzing open-source repositories to identify patterns and correlations among metrics. Specifically, we examine how frequently functions propagate errors and the distance between error signaling and handling. Additionally, we explore whether a project's maturity level influences the complexity of error propagation and handling. The adopted methodology involves the automated extraction of metrics such as the proportion of error propagation per function, the depth of error-handling chains, and the number of ignored errors, complemented by data visualization techniques. The results indicate that, on average, 17% of functions in the analyzed code involve error propagation or handling. Most errors in the dataset are propagated through up to four function calls before being handled, with this number increasing as chain complexity grows, reaching a maximum of 12 calls in the analyzed samples. Less mature projects do not show significant differences in error handling compared to more mature ones. Finally, the investigation revealed relevant patterns in Rust's error-handling practices, including the frequent use of the expect and unwrap functions, the replacement of error messages that compromises traceability, and the use of dynamic errors in CLI executables. The primary contribution of this study is to provide an in-depth analysis of error handling in Rust, helping to understand the community's practices and offering insights for future research on software quality and security in the language.

Keywords: Rust, Error Handling, Error Propagation, Software Quality

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AST	Abstract syntax tree
CVEs	Common Vulnerabilities and Exposures
HIR	High-Level Intermediate Representation
LLVM IR	Low Level Virtual Machine Intermediate Representation
MIR	Mid-level Intermediate Representation
THIR	Typed High-Level Intermediate Representation

SUMÁRIO

1 INTRODUÇÃO

Entender como os erros são tratados em diferentes linguagens de programação é fundamental para compreender como elas são utilizadas na prática e como os desenvolvedores estruturam aplicações no mundo real [1]. Estudos têm sido realizados para mapear o comportamento dos sistemas de erros em diferentes linguagens, pois são aspectos frequentemente negligenciados do código. Há pesquisa que explora como é realizado o tratamento de erros em Swift [1], como é usado o gerenciamento de exceções em Java [2] e que investiga a adoção do mecanismo de tratamento de exceções em C++ [3].

No caso de Rust, poucos estudos investigam diretamente o comportamento de sistemas de erro da linguagem. Muitos deles abordam erros em contextos específicos, como Xu et al. (2021), que analisa todos os Common Vulnerabilities and Exposures (CVEs) relacionados à segurança de memória em Rust [4], e Chen et al. (2023), que propõe uma abordagem alternativa para lidar com erros de alocação de memória em tempo de execução [5]. Nesses casos os erros são estudados pontualmente, e não como elementos que percorrem um fluxo dentro do sistema. Há também trabalhos mais abrangentes, como Qin et al. (2024), que investiga *panics* e o uso inadequado de tipos de erro em projetos *open source* [6]. No entanto, esse estudo também não analisa a organização estrutural do tratamento de erros, demonstrando uma lacuna na investigação da arquitetura de propagação de erros em Rust. Essa lacuna se torna ainda mais relevante diante do crescimento recente de Rust.

Nos últimos anos, Rust tem se consolidado como uma das linguagens de programação mais admiradas, conforme indicado pela pesquisa anual do Stack Overflow [7]. Esse reconhecimento acompanha um crescimento significativo no uso da linguagem, que atualmente ocupa a 19ª posição no *ranking* das mais adotadas [8], evidenciando sua expansão e aceitação na comunidade de desenvolvedores. Seu modelo de ownership e borrowing é uma de suas principais características, garantindo segurança no acesso e manipulação de memória sem a necessidade de um garbage collector. Essa abordagem impacta diretamente o tratamento de erros na linguagem. Diferente de linguagens que utilizam mecanismos de tratamento de exceções, os quais podem interromper o fluxo de execução de forma inesperada, Rust implementa um gerenciamento explícito de erros, incentivando, ou mesmo forçando, os desenvolvedores a lidar com as informações dos erros [9].

Diante disso, Rust é considerada uma linguagem mais segura. Para aprofundar essa percep-

ção, esta pesquisa analisará a propagação de erros em projetos que utilizam a linguagem. Esse tipo de análise examina o caminho percorrido pelos erros no código, desde sua origem até o ponto em que são tratados ou registrados. Abordagens semelhantes já foram aplicadas a outras linguagens, como no estudo de Fu & Ryder (2007), que desenvolveram uma análise estática para identificar cadeias completas de propagação de exceções em Java [10], e no trabalho de Cacho et al. (2014) no qual analisaram como mudanças no código C# afetam a robustez do tratamento de exceções [11]. Essas pesquisas destacam a importância de compreender o fluxo dos erros dentro de um programa, e este estudo busca expandir essa abordagem para Rust.

A análise estática foi a escolhida, pois oferece insights sobre os fluxos de erro e potenciais problemas antes mesmo da execução [12]. Esses são fatores essenciais, já que a arquitetura de propagação de erros raramente é detalhada pelos programadores, e a identificação manual de problemas pode ser extremamente desafiadora. Erros podem atravessar múltiplos componentes e camadas de um sistema antes de serem registrados para análise, tornando difícil localizar a causa raiz [10]. Esse desafio é ainda maior quando o comportamento de recuperação de erros e as interações entre os componentes não estão bem documentados.

Dado o cenário apresentado, este trabalho tem como objetivo proporcionar uma compreensão aprofundada das práticas de tratamento de erros em Rust, buscando responder às seguintes perguntas:

- **P1:** *Com que frequência funções propagam erros em projetos Rust open source?*
- **P2:** *Quão longe dos locais onde erros são sinalizados programas em Rust tratam erros?*
- **P3:** *Maturidade influencia a complexidade da propagação e tratamento de erros?*

Para responder a essas perguntas, este estudo utilizou a ferramenta *static-result-analyzer* [13], desenvolvida por Thomas Kas. Essa ferramenta foi criada para analisar a propagação de erros em Rust de forma estática, permitindo mapear o fluxo de erros dentro do código por meio da extração de métricas estruturais. A pesquisa original de Kas, no entanto, teve um escopo limitado, sendo aplicada a um número reduzido de projetos [14]. Assim, este estudo amplia essa abordagem, aplicando a ferramenta a um conjunto maior e mais diversificado de repositórios Rust, considerando diferentes níveis de maturidade e contextos de desenvolvimento. Com essa expansão, espera-se identificar padrões no tratamento e propagação de erros, avaliando se projetos mais maduros adotam melhores práticas.

2 FUNDAMENTOS

Para compreender os conceitos e análises apresentados neste trabalho, é fundamental estabelecer uma base conceitual sobre erros, Rust, seu modelo de tratamento de erros e o funcionamento de seu compilador. Dessa forma, esta seção apresenta uma visão geral desses elementos, fornecendo as informações essenciais para a contextualização e compreensão dos tópicos abordados nas seções seguintes.

2.1 CONCEITO DE ERRO

Na engenharia de software, um erro é definido como uma ação humana que produz um resultado incorreto. Esse erro leva a uma falha no software que, quando executada, resulta em uma falha observável, como um resultado incorreto exibido na tela, o travamento ou fechamento inesperado do programa, ou até mesmo a corrupção de arquivos [15]. No contexto da linguagem Rust, os erros podem ser classificados em dois tipos: recuperáveis e não recuperáveis. Um erro não recuperável provoca a interrupção da execução do programa, enquanto um erro recuperável permite que o programa adote medidas apropriadas para lidar com a situação e continuar sua execução [6].

Ao longo deste trabalho, o conceito de erro será frequentemente utilizado no sentido da informação gerada por um erro recuperável, com o objetivo de tornar a leitura mais fluida e evitar repetições desnecessárias. Já os erros não recuperáveis serão tratados com base no conceito de pânico (*panic*), conforme definido pela linguagem.

Para compreender melhor como esses dois tipos de erro são tratados na prática, é necessário conhecer os principais mecanismos internos da linguagem Rust, cuja estrutura foi concebida justamente para prevenir falhas em tempo de execução e incentivar o tratamento seguro e explícito de erros.

2.2 RUST

Rust é uma linguagem de programação estaticamente tipada, sem *garbage collector*, que se destaca por combinar alta performance com um forte compromisso com a segurança. Como descreve os autores Blandy e Orendorff, Rust é *"uma linguagem concorrente segura com a per-*

formance de C e C++" [9]. Seu grande diferencial reside na forma como gerencia a memória, utilizando um sistema de *ownership* e *borrowing*, que aplica regras rigorosas verificadas em tempo de compilação.

No sistema de *ownership*, cada valor em Rust tem um proprietário exclusivo durante sua existência no programa. Esse proprietário, chamado de "dono"[1], pode permitir que outras partes do código acessem temporariamente o valor por meio do conceito de *borrowing*. Contudo, essa liberdade é limitada por regras que previnem acessos inválidos à memória. As três principais regras do sistema de *ownership* são [16]:

- **Cada valor em Rust tem um único dono:** Apenas uma parte do programa detém o controle total sobre um valor em qualquer momento;
- **Só pode existir um dono por vez:** Quando um valor é movido para outra parte do código, o dono anterior perde o acesso;
- **Se o valor sair de escopo, ele é descartado:** Quando um valor deixa o escopo onde foi definido (por exemplo, ao sair de uma função), sua memória é automaticamente liberada, eliminando a necessidade de um *garbage collector*.

Esse sistema reduz significativamente erros relacionados à memória, como uso de memória não inicializada, ponteiros nulos ou duplicados, e acessos a referências inválidas. Ele garante que cada valor é inicializado antes de ser usado e que múltiplos donos não criam inconsistências, estabelecendo uma base sólida para o modelo de tratamento de erros da linguagem.

Código Fonte 1 – Exemplo de onwership e borrowing

```
fn main() {
    let s = String::from(""); // s é o dono da String
    print_emprestimo(&s); // Empréstimo imutável (não move a posse)
    println!("{}", s); // Isso não causaria erro, pois s ainda está no escopo
    print_posse(s); // Transfere a posse para a função
    println!("{}", s); // Isso causaria erro, pois s foi movido
}

fn print_emprestimo(s: &String) {
    println!("Empréstimo: {}", s);
} // s sai do escopo, mas não há problema, pois é um empréstimo.

fn print_posse(s: String) {
```

[1] refere-se à variável ou estrutura responsável por gerenciar o ciclo de vida de um valor

```
    println!("Posse: {}", s);
} // s sai do escopo e a memória é liberada.
```

Fonte: Elaborado pela autora (2025).

Outro aspecto crucial de Rust, que impacta diretamente sua abordagem ao tratamento de erros, é sua característica de *Type Safety* (segurança de tipos). Essa propriedade impede operações inválidas ou inconsistentes com os tipos de dados definidos no programa. Por exemplo, o compilador detecta e impede operações como somar um inteiro com um ponto flutuante sem conversão explícita, capturando esses erros em tempo de compilação [9].

Combinando *Type Safety* e o sistema de *ownership*, Rust não apenas previne erros em tempo de execução, mas também promove um código mais seguro e robusto, forçando os desenvolvedores a lidar de forma explícita com possíveis falhas. Esse rigor, como será detalhado a seguir, é especialmente relevante no contexto do tratamento de erros, onde a linguagem exige que eles sejam tratados de maneira clara e intencional.

2.3 TRATAMENTO DE ERROS EM RUST

No geral, muitas linguagens de programação tratam erros com o uso de exceções, que podem interromper o fluxo normal do programa e transferir o controle para um bloco de tratamento específico. Essas exceções podem introduzir imprevisibilidade ao código, dificultando a rastreabilidade e aumentando o risco de erros silenciosos ou não tratados [17]. Por esses motivos, Rust adota uma abordagem diferente e mais explícita para o tratamento de erros. A linguagem substitui o modelo de exceções por um sistema baseado em tipos, no qual os erros devem ser tratados diretamente pelo desenvolvedor, sem que interrupções inesperadas ocorram no fluxo do programa. Rust possui o tipo **Result<T, E>** para erros recuperáveis e o sistema de *panic*, que interrompe a execução quando o programa encontra um erro irrecuperável [9].

2.3.1 Results

Funções que podem falhar em Rust utilizam o tipo enum **Result<T, E>**, no qual "T" é um parâmetro genérico que representa o tipo do valor retornado em caso de sucesso, encapsulado na variante *Ok*, e "E" é um parâmetro genérico que representa o tipo do erro retornado em caso de falha, encapsulado na variante *Err* [16]. A definição do enum é a seguinte:

Código Fonte 2 – Definição do Result

```
enum Result<T, E> {
    Ok(T), // Representa sucesso com um valor de tipo T.
    Err(E),// Representa falha com um valor de tipo E.
}
```

Fonte: Adaptado de The Rust Programming Language (Rust Team, 2024).

A forma mais comum e eficiente de lidar com o tipo *Result* em Rust é utilizando a estrutura *match*. Essa estrutura garante que todas as variantes possíveis do tipo sejam tratadas de forma explícita, o que se assemelha ao uso de *try/catch* para exceções em outras linguagens. No caso do *Result*, isso significa tratar explicitamente os cenários de sucesso e de falha [16]. Essa abordagem evita que erros sejam ignorados ou deixados sem tratamento, promovendo segurança no código e incentivando boas práticas de desenvolvimento.

Abaixo, segue um exemplo de uso do *match*, no qual a função "divide" retorna um *Result* que pode conter um valor inteiro de 32 bits (i32) em caso de sucesso ou uma mensagem de erro do tipo *String* em caso de falha:

Código Fonte 3 – Uso de *match*

```
fn divide(a: i32, b: i32) -> Result<i32, String> {
    if b == 0 {
        Err(String::from("Divisão por zero não é permitida"))
    } else {
        Ok(a / b)
    }
}

fn main() {
    let resultado = divide(10, 2);

    match resultado {
        Ok(valor) => println!("Resultado: {}", valor),
        Err(erro) => println!("Erro: {}", erro),
    }
}
```

Fonte: Elaborado pela autora (2025).

Além do uso do match, Rust oferece métodos utilitários que podem ser aplicados a casos específicos, como **is_ok()**. Este método retorna um valor booleano indicando *true* se o Result contiver um valor de sucesso ou false se contiver um erro [16]. Esses métodos, assim como outros disponíveis para *Result* e *Option*, são ferramentas que facilitam o tratamento de erros em

Rust.

Para simplificar a propagação de erros, Rust tem o operador "?". Este operador pode ser aplicado a valores do tipo *Result* (ou *Option*) e possui duas funções principais.

Código Fonte 4 – Funcionamento do operador ?

```
let resultado = divide(10, 2);
match resultado {
    Ok(valor) => Ok(valor), // Propaga o valor
    Err(erro) => Err(erro),  // Propaga o erro
}
```

Fonte: Elaborado pela autora (2025).

Se o *Result* contiver um valor de sucesso , o operador realiza um *unwrap* e retorna o valor encapsulado. Se o *Result* contiver um erro, ele interrompe a execução da função atual e retorna o erro ao chamador [9]. Essa abordagem torna o código mais limpo e legível, como mostra abaixo.

Código Fonte 5 – Exemplo com operador ?

```
let resultado = divide(10, 2)?; // Propaga o erro automaticamente se for Err
```

Fonte: Elaborado pela autora (2025).

Todas características comentadas anteriormente trazem maior liberdade e segurança para os desenvolvedores, onde a única forma explícita de ignorar erros sem causar panic seria o _.

Código Fonte 6 – Uso de _

```
match divide(10.0, 0.0) {
    Ok(valor) => println!("Resultado da divisão: {}", valor),
    _ => (), // Ignora qualquer erro, sem tratá-lo ou reportá-lo
}
```

Fonte: Elaborado pela autora (2025).

A obrigatoriedade de tratar erros explicitamente resulta em código onde o fluxo de controle para tratamento de erros se mistura com a lógica principal da aplicação, dificultando a separação entre esses dois aspectos durante uma análise estática. Além disso, a flexibilidade fornecida pelo *match* e pelos métodos associados ao *Result* permite que os desenvolvedores manipulem erros de diversas maneiras, o que complica a tarefa de prever ou rastrear o comportamento do código de forma automática.

2.3.2 Panic

Durante a execução de um programa em Rust, pode ocorrer um *panic*, que representa uma falha em tempo de execução, normalmente causada por situações inesperadas ou inválidas, como índices fora dos limites de um vetor ou o uso de métodos como **unwrap()** em valores de erro. Quando um *panic* acontece, o programa pode seguir um de dois comportamentos, configuráveis pelo usuário. O primeiro é *Unwind*, onde pilha de chamadas é "desmontada", liberando recursos alocados à medida que o programa tenta sair de forma controlada. O segundo é *Abort* quando, o programa encerra imediatamente sem desmontar a pilha, liberando recursos apenas no nível do sistema operacional [9].

Embora o Rust incentive um tratamento explícito e seguro de erros com tipos como Result, métodos como ***unwrap()*** e ***expect()*** podem levar a *panic* se forem usados de forma inadequada. Por esse motivo, nesse trabalho a contagem de *panics* leva em consideração *unwraps* e *expects* que chamam *Results* ou *Options*. Esses métodos, quando aplicados a um valor *Err* ou *None*, forçam o programa a entrar em estado de pânico [16]. Por exemplo:

Código Fonte 7 – Exemplo de panic

```
let valor = Some(10);
let resultado = valor.unwrap(); // Funciona porque `valor` é `Some`.
let nenhum_valor: Option<i32> = None;
let erro = nenhum_valor.unwrap(); // Causa panic porque é `None`.
```

Fonte: Elaborado pela autora (2025).

2.4 RUSTC

O compilador de Rust, chamado rustc, verifica rigorosamente as propriedades de segurança da linguagem, como *ownership*, *borrowing* e tipos, durante o tempo de compilação [18]. Ele é extremamente robusto, fornecendo mensagens de erro detalhadas que ajudam o desenvolvedor a identificar e corrigir problemas de maneira eficiente. Isso evita problemas comuns em linguagens como C e C++, como *use-after-free*, vazamentos de memória e acessos concorrentes inseguros [19]. Essas características fazem do compilador uma ferramenta fundamental para análises estáticas. Considere o seguinte código:

Código Fonte 8 – Exemplo código com erro de compilação

```
fn main() {
```

```
    let mut x = 10;
    let y = &x; // Empréstimo imutável de 'x'
    x = 20;     // Tentativa de modificar 'x' enquanto está emprestado
    println!("y: {}", y);
}
```

Fonte: Elaborado pela autora (2025).

O compilador de Rust geraria um erro detalhado ao tentar compilar o código acima, explicando por que ele ocorre e apontando as linhas específicas envolvidas no problema. A resposta do compilador 9 mostra onde o empréstimo começa ***(let y = &x;)***, onde o erro acontece ***(x = 20;)*** e como o empréstimo ainda está em uso ***(println!("y: ", y);)***.

Código Fonte 9 – Exemplo de erro do compilador

```
error[E0506]: cannot assign to `x` because it is borrowed
 --> src/main.rs:4:5
  |
3 |     let y = &x; // Empréstimo imutável de 'x'
  |             -- borrow of `x` occurs here
4 |     x = 20;     // Modificação inválida enquanto `x` está emprestado
  |     ^^^^^^^^^^ assignment to `x` occurs here
5 |     println!("y: {}", y);
  |                       - borrow later used here
```

Fonte: Elaborado pelo compilador rustc (2025).

Esses detalhes ajudam o desenvolvedor a compreender a causa raiz do problema e oferecem uma base sólida para a correção do erro. A capacidade do rustc de fornecer mensagens de erro tão detalhadas e úteis está diretamente relacionada à forma como o compilador é projetado, especialmente no uso de etapas intermediárias.

O processo de compilação de Rust segue uma sequência bem definida [18]:

1. ***Lexing* e *Parsing*:** O compilador começa com o *lexing*, onde o código-fonte é transformado em uma sequência de *tokens*. Em seguida, o *parsing* converte esses *tokens* em uma Árvore de Sintaxe Abstrata (Abstract syntax tree (AST)).

2. **High-Level Intermediate Representation (HIR):** A AST é convertida em uma Representação Intermediária de Alto Nível, onde são realizadas verificações de regras de tipo, *ownership*, *lifetimes* e *borrowing*. O HIR é mais abstrato que o código original, mas ainda preserva muitas semelhanças.

3. **Typed High-Level Intermediate Representation (THIR):** Uma versão reduzida do HIR, onde todos os tipos são preenchidos após a verificação de tipos.

4. **Mid-level Intermediate Representation (MIR):** Baseado em um gráfico de fluxo de controle, elimina expressões aninhadas e torna todos os tipos explícitos.

5. **Low Level Virtual Machine Intermediate Representation (LLVM IR):** O MIR é então convertido para a Representação Intermediária do LLVM (LLVM IR), que é uma forma de código de baixo nível. Este código passa por diversas otimizações para melhorar a eficiência do programa.

6. **Geração de código de máquina:** Por fim, o LLVM IR é traduzido para o código de máquina, que pode ser executado diretamente pelo processador.

Esse processo garante que o código gerado seja eficiente e seguro, aproveitando ao máximo as vantagens de análise estática e otimização oferecidas pelo compilador rustc.

2.5 ANÁLISE ESTÁTICA

A análise estática, conceito central deste trabalho, foi escolhida como abordagem para a avaliação sistemática dos códigos. Esse processo consiste em inspecionar o código-fonte sem executá-lo, permitindo a identificação de potenciais problemas de forma independente das entradas e saídas do programa e abrangendo todos os possíveis caminhos de execução [20].

Essa técnica possibilita a detecção de uma ampla gama de problemas que podem comprometer a segurança, o desempenho e a qualidade do projeto. Em geral, é muito utilizada em ferramentas externas às linguagens, para identificar erros lógicos como código inatingível, falhas em fluxos de execução e uso ineficiente de recursos. Além disso, verifica a conformidade com boas práticas de desenvolvimento, garantindo a aderência a padrões de estilo e qualidade [21].

No caso de Rust, o próprio compilador da linguagem realiza uma análise estática rigorosa. Devido ao seu sistema de borrowing, o compilador valida, antes da execução, se todas as referências a dados seguem regras rígidas de propriedade, mutabilidade e tempo de vida[18]. Isso previne falhas comuns em linguagens sem coletor de lixo, como acessos inválidos à memória, condições de corrida e *use-after-free*. Esse nível de análise estática torna Rust mais robusto do que a maioria das linguagens tradicionais, nas quais verificações similares geralmente dependem de ferramentas externas ou ocorrem apenas em tempo de execução [21].

Por fim, no que diz respeito à propagação de erros, a análise estática se destaca como o principal meio de examinar todos os fluxos possíveis, capturando falhas e oferecendo uma visão abrangente da cadeia de erros, sem a omissão de nenhum cenário relevante.

2.6 MATURIDADE DE CÓDIGO

A maturidade do código, no contexto deste trabalho, é definida por um conjunto de fatores que refletem a evolução e estabilidade de projetos de software de código aberto, os quais foram utilizados como base para a análise neste estudo. Para definir essas métricas, são utilizados critérios inspirados no projeto de código aberto InnerSource [22], que discute boas práticas em projetos de código aberto e colaborativo. Neste trabalho são considerados cinco aspectos principais: frequência de atualização, popularidade, engajamento da comunidade, tempo de vida do projeto [22] e qualidade do código (avaliada por testes automatizados, ferramentas de linting e documentação).

A frequência de atualização de um software pode ser um indicativo de maturidade, pois reflete a adoção de práticas alinhadas à entrega contínua, como atualizações frequentes e incrementais. Essa abordagem facilita a detecção precoce de conflitos e inconsistências, promovendo melhorias constantes e contribuindo para a estabilidade do sistema ao longo do tempo [23].

Para Projetos de Software Livre e Código Aberto, popularidade e o engajamento da comunidade são muito importantes. Projetos frequentemente se tornam populares devido à transparência e possibilidade de colaboração aberta [22]. Esse modelo facilita a contribuição de diversos desenvolvedores, o que acelera a identificação e correção de bugs, além de aprimorar funcionalidades. Além disso, projetos populares atraem mais contribuidores, o que pode levar a uma melhor revisão do código e, consequentemente, a um aumento na qualidade do software [24].

Outro fator analisado é o tempo de vida do projeto, que pode indicar sua estabilidade e maturação ao longo do tempo. Projetos que permanecem ativos por longos períodos geralmente acumulam refinamentos estruturais e boas práticas, o que pode resultar em um código mais robusto [23]. No entanto, esse aspecto deve ser avaliado em conjunto com a frequência de atualizações, pois projetos antigos, mas pouco mantidos, podem não refletir boas práticas.

Por fim, a qualidade do código é considerada por meio da adoção de práticas como testes automatizados, linting e documentação. A presença de testes abrangentes melhora a confiabilidade do software, enquanto ferramentas de linting ajudam a garantir um código mais padronizado e legível [15]. Além disso, a documentação bem estruturada facilita a manutenção e colaboração

no projeto, tornando-o mais acessível para novos contribuidores.

Dessa forma, a maturidade do código, neste trabalho, é analisada a partir desses fatores, permitindo investigar sua influência no tratamento de erros em projetos Rust de código aberto. A correlação entre maturidade e boas práticas no gerenciamento de erros pode fornecer insights sobre como projetos mais desenvolvidos lidam com falhas e se há um impacto significativo na propagação e resolução de erros no código.

3 METODOLOGIA

Para analisar como os erros são tratados em projetos Rust, esta pesquisa adota uma abordagem empírica, utilizando a ferramenta *static-error-analyzer* [13] para realizar uma análise estática de código em repositórios públicos no GitHub. O objetivo é investigar a frequência em que erros ocorrem, a relação entre a sinalização e o tratamento de erros e as diferenças no tratamento de erros entre projetos de diferentes níveis de maturidade.

3.1 FERRAMENTA

A ferramenta *static-error-analyzer* analisa programas Rust interceptando o processo de compilação para extrair informações estruturais do código. Ela utiliza a representação intermediária HIR para construir um grafo de chamadas, no qual os nós representam funções e as arestas indicam as chamadas entre elas. Esse grafo é enriquecido com informações extraídas do MIR, detalhando a propagação de valores e tipos de retorno. A partir desse grafo, a ferramenta identifica cadeias de propagação de erros, rastreando os caminhos que um erro pode percorrer até ser tratado. Para essa análise, são adotadas algumas definições fundamentais:

- Um erro é considerado qualquer tipo utilizado como segundo argumento de um **Result<T, E>**.
- A propagação de erro ocorre quando uma função retorna um Result ou utiliza o operador "?" sobre um *Result*.
- O erro é considerado tratado quando sua propagação cessa, ou seja, quando ele é capturado e processado explicitamente no código.

O resultado final da análise são dois grafos: um que ilustra a propagação e o tratamento de erros no programa e outro é o grafo de chamadas, permitindo um estudo detalhado do fluxo de erros. Além disso, a ferramenta coleta quatro métricas específicas: Número total de cadeias de propagação de erro; Tamanho total da maior cadeia de propagação de erro; Maior sequência de chamadas consecutivas dentro de uma cadeia de propagação; Tamanho médio das cadeias de propagação de erro.

3.1.1 Limitações da Ferramenta

Embora a ferramenta *static-error-analyzer* forneça uma base sólida para a análise da propagação de erros em Rust, ela apresenta algumas limitações que precisam ser consideradas na interpretação dos resultados. De acordo com o estudo original da ferramenta [14], seus principais pontos de falha incluem:

- A ferramenta considera um erro como tratado sempre que ele não é propagado, o que pode levar a classificações incorretas. Esse critério pode incluir erros que são ignorados intencionalmente, como aqueles descartados com "_", sem qualquer registro ou manipulação no código. Nesse caso, é possível que o erro seja simplesmente descartado, sem impacto no fluxo do programa.
- O modelo original da ferramenta não inclui um mapeamento detalhado dos pontos onde o sistema de *panic* é utilizado dentro dos programas analisados. Como esse mecanismo está relacionado a falhas irrecuperáveis, sua ausência pode limitar a compreensão sobre como os erros são gerenciados em projetos Rust.

Além dessas limitações, há restrições técnicas relacionadas à execução da ferramenta, como a necessidade de os projetos serem compiláveis via cargo build, a falta de disponibilidade de MIR em alguns casos (o que impacta a extração de informações de tipo), e a impossibilidade de construir um grafo de chamadas completo para bibliotecas externas. No entanto, como o foco desta pesquisa é realizar uma análise quantitativa dos padrões de propagação e tratamento de erros, priorizou-se a mitigação das duas primeiras limitações.

3.1.2 Mudanças na ferramenta

Dado o foco desta pesquisa na análise de dados de projetos *Open Source*, algumas modificações foram implementadas na ferramenta para aprimorar a coleta de métricas. Para lidar com o caso em que o erro é tratado sempre que ele não é propagado, foi adicionada uma lógica que captura os casos em que erros são ignorados intencionalmente pelo desenvolvedor:

Código Fonte 10 – Como erros são ignorados em RUST

```
1 // Caso 1
  if let Err(_) = foo() {}
3
```

```
// Caso 2
match foo() {
    Ok(val) => println!("{}", val),
    Err(_) => {},
}
```

Fonte: Elaborado pela autora (2025).

Além disso, a análise foi expandida para identificar possíveis pontos de *panic* no código, uma vez que certos padrões de código podem gerar ou trazer riscos de pânicos em tempo de execução. Para detectar essas ocorrências, foi realizada uma análise do grafo de chamadas de função, a fim de mapear quais funções podem causar *panic*.

Essa análise levou em consideração o funcionamento interno do compilador Rust, identificando as principais funções e macros que podem levar a um *panic*. Foi constatado que, no Rust, nem todo pânico ocorre da mesma forma: enquanto macros como **assert!()** e **assert_eq!()** chamam internamente **core::panicking::assert_failed**, outras falhas se propagam através de chamadas internas até resultar em **core::panicking::panic**.

Com base nisso, foram selecionadas as principais funções associadas ao *panic*. A seleção dessas funções foi feita de forma mutuamente excludente, garantindo que cada ocorrência fosse contabilizada uma única vez, evitando a duplicação de contagem devido a chamadas indiretas. As funções analisadas incluem:

- **std::option::Option::<T>::unwrap** – Causa *panic* se o valor for None.
- **std::option::Option::<T>::expect** – Causa *panic* se o valor for None.
- **std::result::Result::<T, E>::unwrap** – Causa *panic* se o *Result* for Err(_).
- **std::result::Result::<T, E>::expect** – Causa *panic* se o *Result* for Err(_).
- **core::panicking::AssertKind::Eq e core::panicking::AssertKind::Ne** – Representam falhas de **assert_eq!()** e **assert_ne!()**.
- **core::panicking::panic** – Função interna que lida com *panic* diretamente.
- **std::rt::panic_fmt** – Chamada quando o macro **panic!()** é usado diretamente.

Após identificar as funções que podem causar *panic*, foi realizada a contagem das arestas que chegam a cada uma delas no grafo de chamadas, permitindo determinar quantas vezes esses pânicos podem ocorrer no fluxo de execução. Essa abordagem quantifica a propagação dos *panics*, fornecendo uma visão objetiva da frequência de possíveis falhas no código.

Com essas modificações, novas métricas foram adicionadas à análise, totalizando 13 indicadores para avaliar o código:

1. **Total de Cadeias de Propagação de Erro** - Contagem de fluxos distintos onde erros são passados de função para função. Mostra quantos caminhos diferentes um erro pode percorrer dentro do programa, ajudando a avaliar a complexidade do fluxo de erro.

2. **Maior Cadeia de Propagação** - Número máximo de funções conectadas por propagação de erro. Cadeias muito grandes podem dificultar o diagnóstico e tratamento de falhas.

3. **Maior Caminho de Erro** - Maior trecho ininterrupto de propagação dentro de uma única cadeia. Indica trechos do código onde erros se propagam sem interrupção, o que pode sugerir falta de tratamento intermediário.

4. **Tamanho Médio das Cadeias de Propagação** - Média do número de funções por cadeia de propagação. Ajuda a entender se no geral os erros são resolvidos rapidamente ou se percorrem muitos níveis antes de serem tratados.

5. **Possíveis Panics** – Contagem de funções que chamam a função interna de pânico de *core::panicking::panic* e que chamam os macros *unwrap()* ou *expect()* com *Result* ou *Option*. Essa métrica ajuda a identificar potenciais pontos críticos no sistema, uma vez que panics podem comprometer a execução do programa.

6. **Taxa de Possíveis Panics** – Proporção de funções que contêm chamadas a *panics* em relação ao total de funções do código. Projetos com uma taxa alta podem estar mais suscetíveis a falhas inesperadas.

7. **Erros Ignorados** – Número de erros descartados sem qualquer tratamento explícito, como quando _ é utilizado. Essa métrica é relevante, pois erros ignorados silenciosamente podem comprometer a confiabilidade do código.

8. **Taxa de Erros Ignorados** – Percentual de erros descartados em relação ao total de erros detectados. Uma taxa alta pode indicar que o projeto está evitando lidar com falhas explicitamente.

9. **Máximo de Erros Recebidos por Função** – Número máximo de erros recebidos por uma única função dentro do código. Se uma função recebe muitos erros, pode ser um ponto crítico na propagação e tratamento de falhas.

10. **Média de Erros Recebidos por Função** – Média do número de erros recebidos por todas as funções do projeto. Ajuda a entender se os erros estão distribuídos uniformemente ou se poucas funções concentram a maioria dos erros.

11. **Taxa de Erros** – Percentual de funções que lidam com erros em relação ao total de funções do código. Uma taxa alta pode indicar que o projeto se preocupa com o tratamento de erros, enquanto uma taxa baixa pode sugerir que muitos casos de erro são ignorados ou tratados de maneira implícita.

12. **Taxa de Propagação de Erros** – Proporção de funções que propagam erros em relação ao total de funções que lidam com erros. Permite analisar se os erros são tratados adequadamente ou apenas propagados sem um controle efetivo.

13. **Total de Funções** – Número total de funções no código. Essa métrica serve como base para normalizar outras métricas e entender a escala geral do projeto.

Essas métricas adicionais foram escolhidas para aprofundar a análise da forma como os erros são tratados nos projetos Rust. Esses aprimoramentos permitem uma visão mais abrangente do comportamento dos erros em projetos, contribuindo para a compreensão das práticas adotadas na comunidade.

3.2 COLETA DE DADOS

A coleta dos dados foi realizada por meio de um *script* em Python, desenvolvido para extrair repositórios do GitHub [25] relacionados à linguagem Rust. O processo iniciou-se com a seleção de repositórios que continham o tópico "rust" e possuíam mais de 500 estrelas, além dos listados no *awesome-rust*, uma curadoria de projetos considerados relevantes pela comunidade [26]. Ao todo, foram selecionados aproximadamente 3.500 repositórios.

Um dos principais desafios enfrentados foi a compatibilidade entre versões. Como a versão *nightly* do Rust é atualizada diariamente, projetos mais antigos ou mais novos podem apresentar incompatibilidades [16]. Para mitigar esse problema, o script priorizou a execução de testes em *commits* realizados até três dias antes da versão utilizada, garantindo maior estabilidade.

Após a coleta inicial, os repositórios foram filtrados para incluir apenas aqueles que puderam ser compilados com sucesso usando ***cargo build***. Essa restrição representou uma limitação, uma

vez que os projetos apresentam diversidade em suas configurações, frequentemente integrando outras linguagens e ferramentas.

Além disso, durante a execução dos projetos, observou-se que repositórios com menos de 1.000 estrelas frequentemente apresentavam poucas ou nenhuma cadeia de erro. Como resultado, após a etapa de compilação, aproximadamente 71% desses projetos foram descartados, pois sua inclusão poderia desequilibrar o conjunto de dados.

Ao final do processo, com todas essas limitações, 112 repositórios foram selecionados para análise estatística. A partir desses projetos, foram extraídas as métricas necessárias para avaliar a maturidade dos repositórios, considerando critérios como popularidade, frequência de atualização e documentação.

3.3 MÉTRICA DE MATURIDADE

A maturidade dos repositórios foi avaliada com base em um conjunto de métricas extraídas automaticamente por um script Python. Para definir essas métricas, foram utilizados critérios inspirados no projeto de código aberto InnerSource [22], que discute boas práticas em projetos de código aberto e colaborativo. Para o código gerar as métricas, os cinco aspectos considerados são: frequência de atualização, popularidade, engajamento da comunidade, tempo de vida do projeto [22] e qualidade do código (avaliada por testes automatizados, ferramentas de linting e documentação).

A popularidade e o engajamento da comunidade em relação ao projeto foram considerados por meio do número de estrelas, forks, issues abertas e seguidores. Projetos com maior interação tendem a apresentar um desenvolvimento mais ativo e contínuo, sendo mais propensos a melhorias e correções ao longo do tempo. Para refletir esse impacto na maturidade, cada métrica foi ponderada na pontuação final, conforme a equação [22]:

$$S_p = (forks \times 5) + (subscribers) + \left(\frac{stars}{3}\right) + \left(\frac{open\_issues}{5}\right)$$

Além da popularidade, a frequência de atualização foi um fator determinante. Repositórios recentemente atualizados indicam continuidade no desenvolvimento, enquanto projetos inativos por longos períodos tendem a se tornar obsoletos [22]. Para incorporar essa variável, o tempo desde a última atualização foi utilizado como um fator de ajuste multiplicativo:

$$F_a = \left(1 + \frac{100 - \min(days\_since\_last\_update, 100)}{100}\right)$$

Dessa forma, repositórios atualizados recentemente sofrem um aumento proporcional na pontuação, enquanto projetos sem atualizações há mais de 100 dias têm impacto reduzido. Outro critério analisado foi a idade do repositório e sua atividade recente. Projetos mais antigos tendem a acumular mais contribuições e refinamentos ao longo do tempo, o que pode indicar maior estabilidade. No entanto, repositórios criados recentemente, mas com alta atividade, também podem demonstrar maturidade crescente [22]. Para evitar um viés que favoreça excessivamente projetos novos, foi aplicada uma bonificação proporcional ao tempo desde a criação e à frequência de atualização, definida por:

$$B = (1000 - \min(days\_since\_last\_update, 365) \times 2.74) \times \left(\frac{365 - \min(days\_since\_creation, 365)}{365} \right)$$

Para evitar distorções, esse bônus foi limitado a um máximo de 500 pontos, garantindo equilíbrio na avaliação. A presença de testes automatizados e ferramentas de qualidade também foi considerada. Repositórios que contêm diretórios tests/, test/ ou arquivos nomeados com _test.rs foram valorizados, pois a existência de testes indica um maior cuidado com a confiabilidade do código. Da mesma forma, a presença de arquivos de configuração do Clippy (clippy.toml) sugere a utilização de boas práticas de linting e melhoria da qualidade do código [15]. Essas características foram incorporadas à pontuação final com valores fixos, 100 caso contenha testes e 50 caso utilize *linting*. Por fim, repositórios com descrições contendo mais de 30 caracteres receberam um acréscimo de 50 na pontuação.

Para garantir que repositórios com pontuações muito altas não distorcessem a análise, foi aplicada uma normalização nos casos em que a pontuação final ultrapassava um limite predefinido [22]. Essa normalização foi feita por meio de um ajuste logarítmico, garantindo que projetos altamente bem avaliados ainda fossem destacados, mas sem criar discrepâncias excessivas:

$$S_{final} = 3000 + \log(S) \times 100, \quad \text{se } S > 3000$$

Por fim, se divide tudo por 10 para diminuir a granularidade da pontuação. Após o cálculo de todas essas métricas, os resultados foram armazenados em um arquivo CSV, permitindo a análise posterior dos repositórios avaliados.

3.4 CONSTRUÇÃO DA ANÁLISE

A análise das métricas coletadas foi conduzida por meio de visualização de dados, com o objetivo de identificar padrões no tratamento de erros em projetos Rust. Os dados provenientes da ferramenta *static-result-analyzer* e do cálculo de maturidade foram processados utilizando a biblioteca Pandas [27]. O uso dessa ferramenta possibilitou a aplicação de técnicas de estatística descritiva, permitindo uma visualização clara e facilitando, na maioria dos casos, a compreensão imediata dos fenômenos analisados [28].

Com foco nas questões "P1: Com que frequência funções propagam erros em projetos Rust open source?"e "P2: Quão longe dos locais onde erros são sinalizados programas em Rust tratam erros?", foi realizada uma análise exploratória, com o cálculo de estatísticas descritivas, como a média geral dos valores analisados, proporcionando uma visão preliminar sobre esses aspectos. Além disso, foi necessária uma análise mais interpretativa para a segunda questão, então foram escolhidos três repositórios com maiores cadeias de propagação e outros três com cadeias intermediárias (entre 2 e 4 níveis) para analisar os caminhos seguidos na propagação dos erros e identificar padrões estruturais.

Para responder à pergunta mais complexa, "P3: A maturidade influencia a complexidade da propagação e tratamento de erros?", foram empregadas diferentes visualizações gráficas para compreender a distribuição das variáveis e identificar padrões relevantes.

Histogramas foram gerados para todas as métricas analisadas, permitindo observar a distribuição e a frequência de fenômenos relacionados ao tratamento de erros nos projetos estudados. A análise da distribuição auxilia na identificação de tendências e possíveis desvios no comportamento dos projetos.

Além disso, foi elaborada uma matriz de correlação utilizando o coeficiente de Kendall, uma vez que os dados não seguem uma distribuição normal. O objetivo dessa matriz é analisar a relação entre o tamanho das cadeias de propagação de erro, a média de propagação de erro, a taxa de panics, a taxa de erros ignorados e a nota de maturidade dos projetos. Essa abordagem possibilitou verificar se a maturidade de um repositório influencia o número e a extensão das cadeias de propagação de erro.

Gráficos de dispersão foram utilizados para explorar possíveis correlações entre a maturidade do projeto e diferentes métricas de tratamento de erro, fornecendo uma visão comparativa entre repositórios com diferentes níveis de maturidade. Adicionalmente, gráficos de boxplot foram gerados para destacar a variação e a presença de outliers nas métricas de propagação de

erro, fornecendo um panorama sobre a consistência e a variabilidade do tratamento de erros nos projetos analisados.

Cada uma dessas visualizações contribuiu para a análise ao sintetizar os dados coletados e evidenciar padrões, auxiliando na formulação de interpretações sobre as práticas adotadas no tratamento de erros em projetos Rust.

3.5 ANÁLISES QUALITATIVAS

Durante a realização das análises dos dados, algumas lacunas foram identificadas, exigindo uma investigação mais aprofundada para compreendê-las em um nível detalhado. Para isso, foi adotada uma abordagem qualitativa, analisando diretamente o código dos repositórios selecionados. Essa etapa visa complementar os achados quantitativos e oferecer uma visão mais interpretativa dos padrões observados. Foram definidas três frentes de análise qualitativa:

- **Uso de *unwrap* e *expect* em possíveis *panics*:** Os três repositórios que apresentam o maior número de potenciais *panics* foram examinados para entender como as funções **unwrap** e **expect** são utilizadas e em quais cenários elas podem representar um risco para a robustez do código.

- **Erros ignorados:** Foram analisados os cinco repositórios com o maior número de erros ignorados para compreender como essa prática ocorre na prática e quais padrões podem ser observados. Em seguida, a mesma análise foi aplicada aos cinco repositórios com maior pontuação de maturidade, buscando avaliar quantos erros ignorados eles possuem e em que contextos ocorrem.

- **Ausência de propagação de erros:** Entre os repositórios com mais de 500 estrelas e menos de 1000, foram selecionados 10 que não apresentam cadeias de propagação de erro para entender os motivos dessa ausência. A análise buscou identificar se há padrões alternativos de tratamento de erros ou outros fatores que justifiquem essa característica.

Essas análises qualitativas permitiram um aprofundamento nos padrões observados estatisticamente, possibilitando a identificação de boas práticas, potenciais fragilidades e o impacto das decisões adotadas na escrita do código.

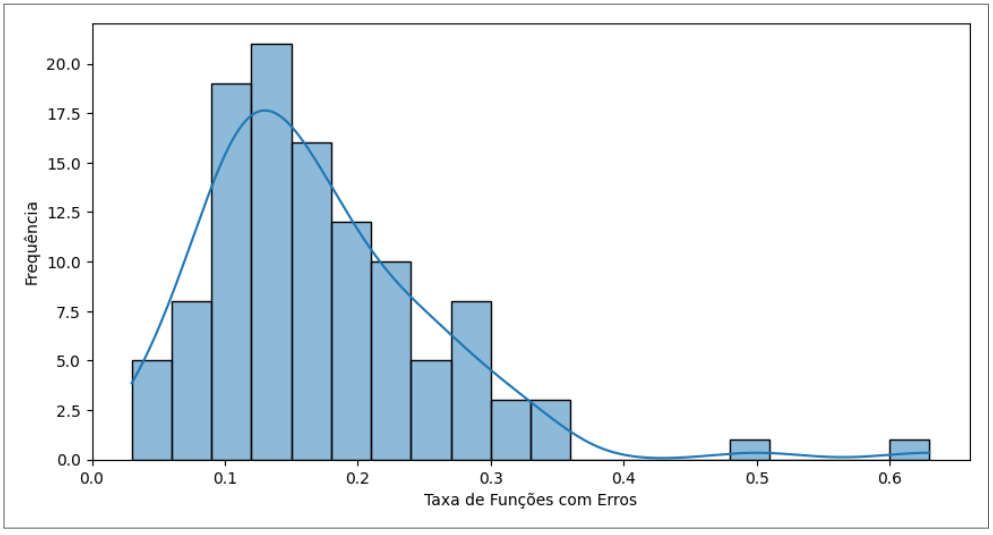
4 RESULTADOS

Nesta seção, apresentamos os resultados da análise da propagação e do tratamento de erros nos projetos Rust investigados. Os dados foram extraídos utilizando a ferramenta *static-error-analyzer* e combinados com métricas de maturidade dos repositórios, possibilitando uma avaliação mais detalhada das questões levantadas neste estudo.

4.1 AVALIAÇÃO DA FREQUÊNCIA DE ERROS

De acordo com a Tabela 1 em média, 17% das funções em um projeto estão envolvidas na propagação ou tratamento de erros, com uma variação entre os projetos de aproximadamente ±8,99%. Considerando que a mediana das funções por projeto é 188, isso indica que, em um projeto típico, cerca de 32 funções lidam com a propagação de erros, enquanto a maior parte do código permanece focada em outras operações. No caso de Linhas de Código, que tem uma mediana de 5912, a média por projeto é de 1005 linhas envolvidas na propagação ou tratamento de erro.

Figura 1 – Distribuição da Taxa de Erros



Fonte: Elaborado pela autora (2025)

Na Figura 1, observa-se que a maioria dos projetos tem entre 10% e 25% de suas funções envolvidas na propagação de erros. Além disso, a Figura 2 mostra que, mesmo em projetos com um número maior de cadeias de propagação de erros, a taxa de erros no código não apresenta um padrão de crescimento ou redução significativa, mostrando que a taxa de erro permanece

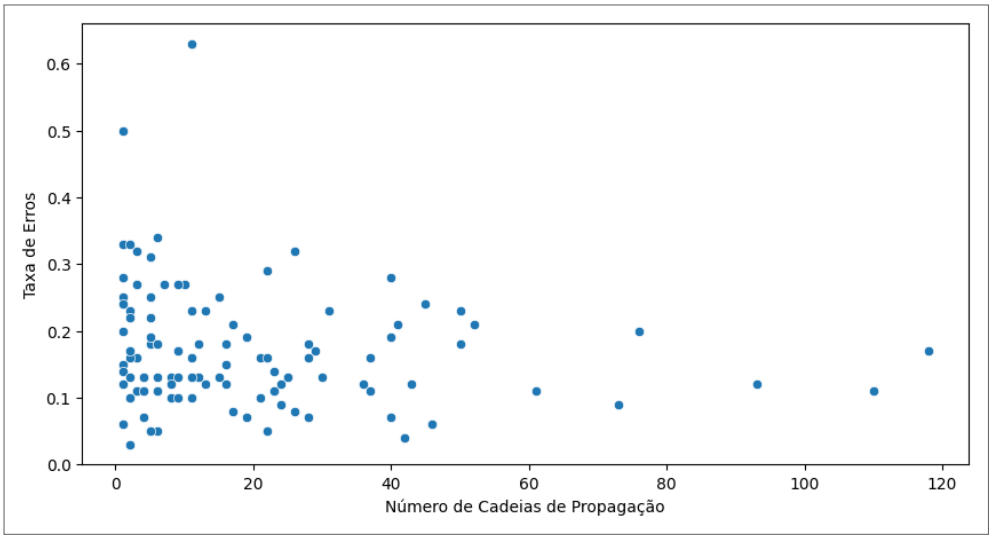
Tabela 1 – Métricas Estatísticas das Funções

Métrica	Média	Máximo	Mínimo	Desvio Padrão	Mediana
Cadeias de Propagação de Erros	18.66	118.00	1.00	21.99	11.00
Maior Cadeia de Chamadas de Função	44.98	1608.00	1.00	166.23	6.00
Maior Caminho de Propagação de Erro	2.57	12.00	1.00	1.96	2.00
Comprimento Médio da Cadeia	5.20	72.00	1.00	10.58	1.64
Possíveis Panics	22.88	301.00	0.00	43.24	8.00
Erros Ignorados	1.12	22.00	0.00	2.89	0.0
Máximo de Erros Recebidos por uma Função	40.71	1026.00	1.00	103.53	17.50
Média de Erros Recebidos por Função	2.87	11.95	1.00	1.59	2.70
Taxa de Erros	0.17	0.63	0.03	0.09	0.16
Taxa de Propagação de Erros	0.74	0.99	0.00	0.28	0.85
Taxa de Possíveis Panics	0.08	0.69	0.00	0.10	0.05
Taxa de Erros Ignorados	0.004	0.08	0.00	0.0096	0.00
Total de Funções	245.84	1109.00	2.00	210.70	188.00
Linhas de Código	14676	309903	395	40733	5912
Métrica de Maturidade	136.98	406.00	0.00	128.89	83.50

Fonte: Elaborada pela autora (2025)

relativamente estável na maioria dos projetos.

Figura 2 – Relação entre Taxa de Erros e Cadeias de Propagação



Fonte: Elaborado pela autora (2025)

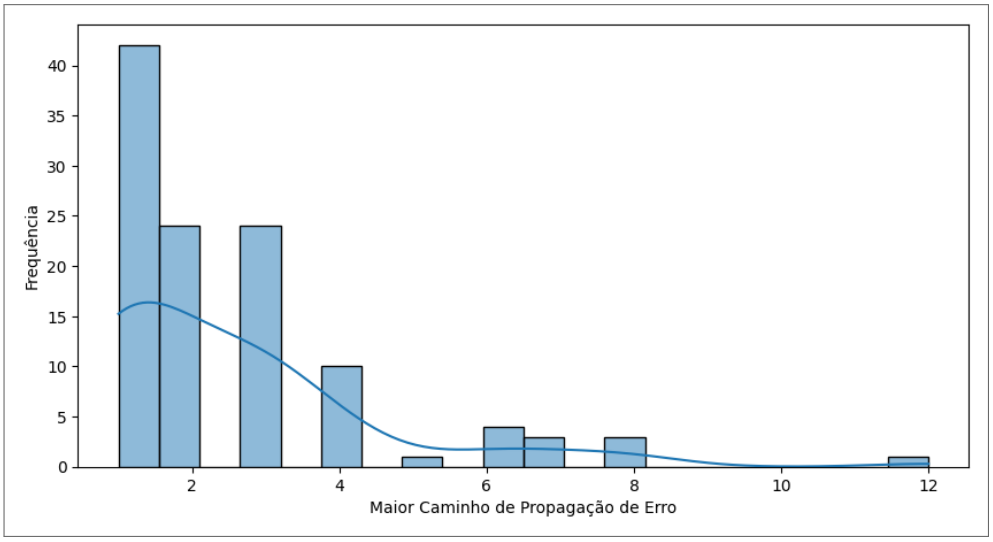
4.2 DISTÂNCIA ENTRE A SINALIZAÇÃO E O TRATAMENTO DE ERROS

A análise dos dados da Tabela 1 revela que, os erros percorrem aproximadamente 2,57 chamadas de função antes de serem tratados, com uma variação de ±1,96 chamadas entre os projetos. Isso indica que, na maioria dos casos, o tratamento de erros ocorre relativamente próximo ao ponto onde são sinalizados. No entanto, alguns projetos apresentam distâncias significativamente maiores, chegando a até 12 chamadas de função no caso mais extremo.

A taxa média de propagação de erros foi de 73,85%, indicando que quase três quartos dos erros são propagados antes de serem tratados. Isso sugere que o tratamento imediato não é a abordagem predominante na maioria dos projetos analisados. Além disso, o desvio padrão de 28,16% revela uma considerável variação entre os projetos, evidenciando que a distância entre a sinalização e o tratamento dos erros não segue um padrão rígido e pode variar significativamente dependendo do contexto do código.

A análise da distribuição no gráfico 3 revela que uma parte significativa dos maiores caminhos está entre 0 e 4 chamadas. Isso sugere que, na maioria dos casos, o tratamento de erros ocorre dentro de um número reduzido de chamadas. No entanto, essa interpretação pode ser influenciada pela forte correlação dessa métrica com a métrica Maior Cadeia de Chamadas. Ao observar o gráfico de distribuição das maiores cadeias de chamadas (4), percebe-se uma semelhança com o de maiores caminhos, indicando que a predominância de caminhos curtos pode ser consequência da própria distribuição dos dados.

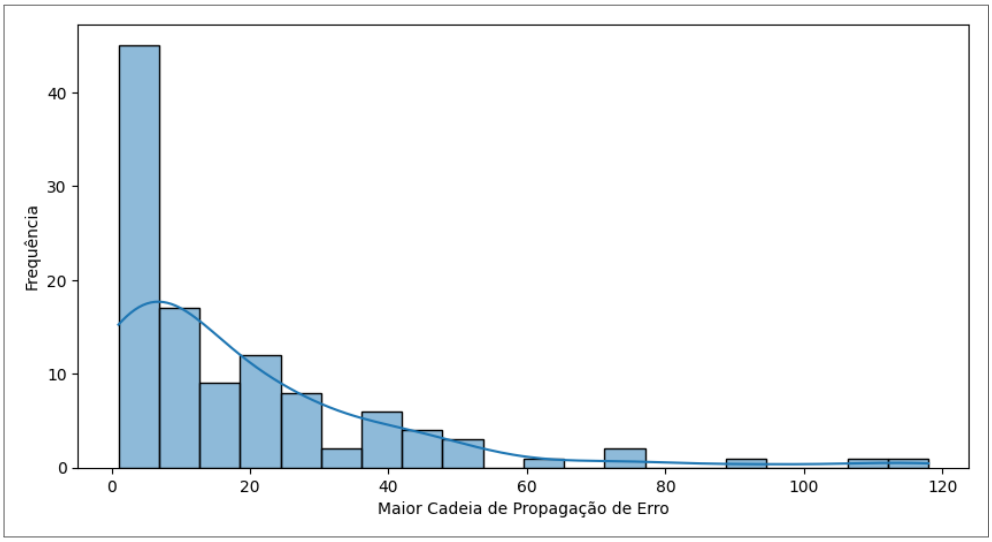
Figura 3 – Distribuição de Maior Caminho de Propagação de Erro



Fonte: Elaborado pela autora (2025)

A Figura 6 revelou uma forte correlação entre a Maior das Cadeias de Chamadas e o Maior Caminho de Erro. Os coeficientes de correlação entre essas métricas atingiram valores acima de 90%, indicando que projetos com maior profundidade apresentam cadeias de propagação de erros mais extensas. Esse comportamento pode ser atribuído à forma como as funções interagem dentro do código: quando um erro ocorre em um nível mais baixo da hierarquia, ele tende a se propagar para funções superiores até encontrar um ponto de tratamento adequado.

Figura 4 – Distribuição de Maior Cadeia de Propagação de Erro



Fonte: Elaborado pela autora (2025)

Para um exame mais detalhado, foram selecionados três repositórios com as maiores cadeias de propagação e outros três com cadeias intermediárias (entre 2 e 4 níveis). A análise foi realizada a partir dos grafos de propagação de erro gerados pela ferramenta *static-error-analyzer*.

A inspeção desses grafos revelou padrões distintos entre projetos com cadeias longas e curtas. Em dois dos três projetos com maior propagação de erros, observou-se que o fluxo geral continha cadeias mais longas, tornando a visualização dos grafos altamente complexa e sugerindo a falta de uma organização clara na propagação dos erros. Nesses projetos, grandes cadeias interconectadas resultavam em uma arquitetura acoplada e difícil de compreender. Nos casos mais extremos, cadeias com seis ou mais níveis já apresentavam um nível significativo de complexidade.

Além disso, esses projetos frequentemente continham múltiplos nós de alta conectividade, ou seja, funções críticas que recebiam um grande volume de erros. Esses nós apareciam em diferentes níveis da cadeia, indicando que cada caminho adotava padrões distintos para lidar com falhas. No final, essas funções acabavam convergindo para outras, tornando o fluxo de

propagação ainda mais intricado. Esse cenário pode dificultar significativamente a manutenção e o entendimento do código, aumentando a chance de erros se perderem no meio de cadeias excessivamente longas.

Por outro lado, no outro projeto de cadeia longa, cuja cadeia mais longa possuía oito funções, a propagação dos erros apresentou uma estrutura mais organizada. A propagação ocorreu predominantemente dentro de uma única cadeia principal, enquanto as outras variavam entre um e quatro níveis. Essa cadeia tinha em média 4 nós de alta conectividade que estavam no mesmo nível hierarquico. Esse modelo organizacional possibilitou uma visualização mais clara do fluxo de erros, mesmo diante de um alto número de chamadas.

Todos os repositórios analisados possuíam um nível de maturidade entre 100 e 250, indicando um histórico de popularidade significativa e envolvimento ativo da comunidade. Demonstrando que, mesmo com múltiplos desenvolvedores contribuindo, a propagação de erros pode sair do controle caso não seja gerida adequadamente. O repositório com a maior cadeia de propagação (12 níveis) encontra-se atualmente desatualizado, sem commits nos últimos três anos e com diversas issues abertas.

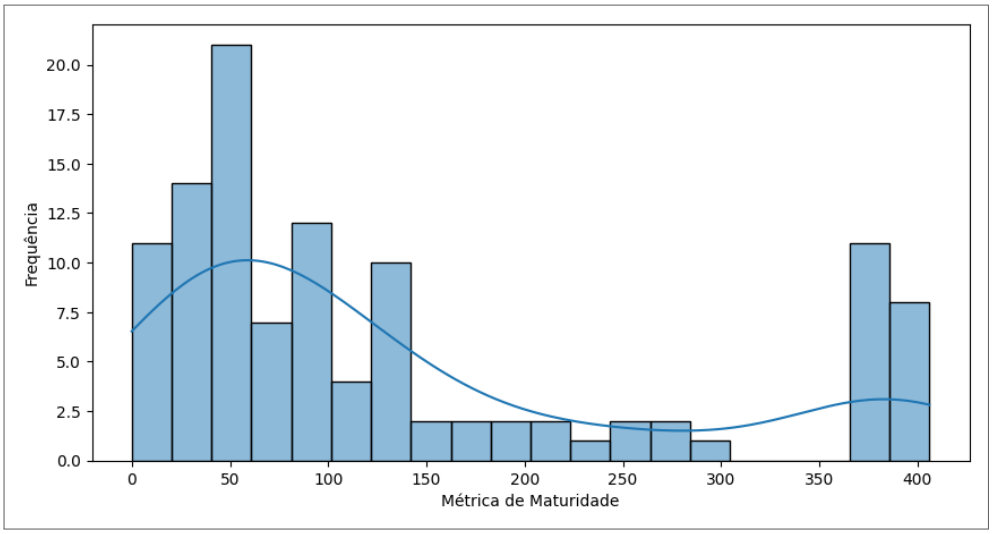
Outra observação relevante é que, dada a alta correlação entre a quantidade de cadeias e seu comprimento, é possível que projetos Rust, à medida que crescem, desenvolvam fluxos de erro cada vez mais difíceis de compreender. As análises realizadas indicam que, sem um controle sobre a propagação, diferentes partes do código podem adotar abordagens distintas para lidar com erros, resultando em maior complexidade e dificultando a rastreabilidade. Esse cenário reforça a importância de definir estratégias eficazes para o tratamento de erros, evitando que a arquitetura do software se torne caótica e de difícil manutenção.

4.3 MATURIDADE X PROPAGAÇÃO DE ERROS

No contexto da maturidade dos projetos, algumas análises foram realizadas. A Figura 5 ilustra a distribuição da métrica de maturidade dos projetos analisados. Observa-se que a maioria dos repositórios se concentra em faixas de menor pontuação de maturidade, enquanto um número reduzido de projetos apresenta valores significativamente mais altos. Isso significa que os dados estão desbalanceados, característica que deve ser levada em consideração para evitar vieses na interpretação dos resultados.

Observando o gráfico de correlação entre as métricas, na última linha da Figura 6, percebe-se que a Pontuação da Maturidade não apresenta uma correlação forte com nenhuma variável.

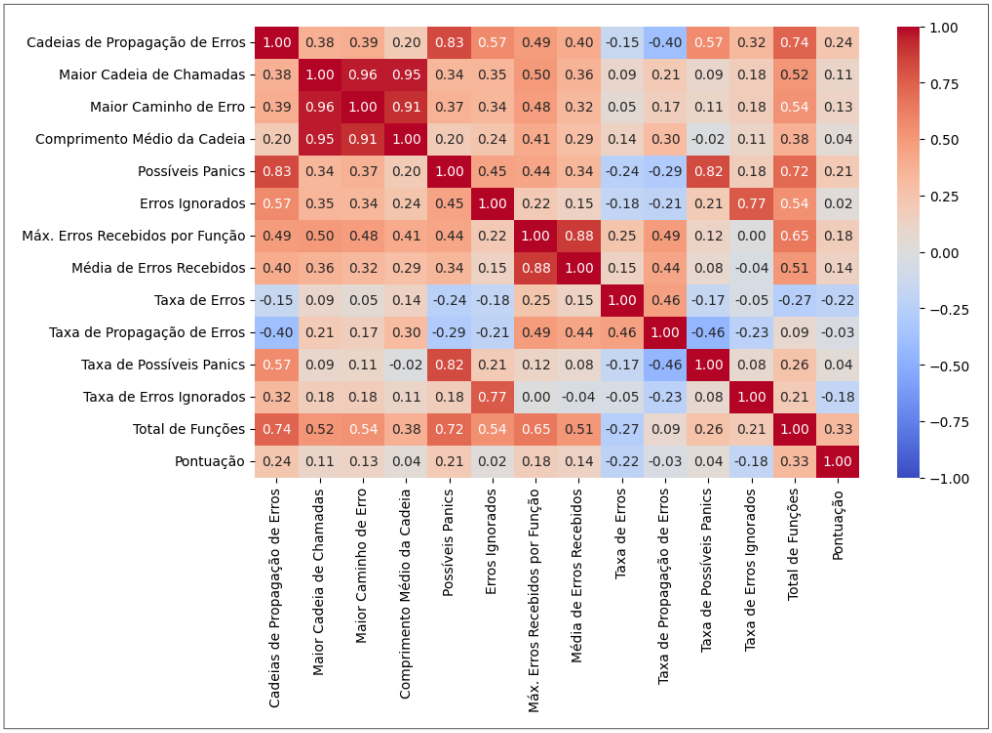
Figura 5 – Distribuição de Métrica de Maturidade



Fonte: Elaborado pela autora (2025)

Isso sugere que a maturidade do projeto não tem uma relação direta com as métricas de erro. Assim, a análise de outros gráficos pode fornecer uma compreensão mais detalhada desse comportamento.

Figura 6 – Correlação entre as Métricas



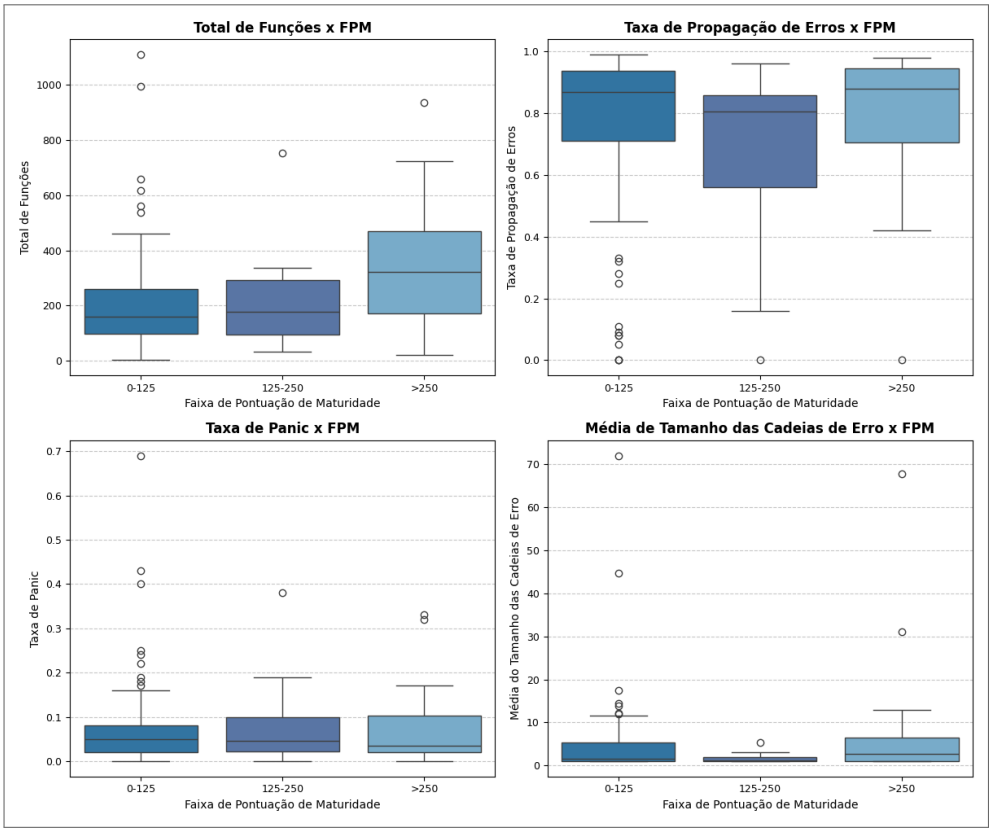
Fonte: Elaborado pela autora (2025)

Ao analisar os gráficos de distribuição, Figura 7, nota-se que as taxas de propagação de erro, *panic* e outras métricas se mantêm relativamente estáveis entre as diferentes faixas de

pontuação de maturidade. Isso indica que, no geral, a maturidade não altera significativamente essas proporções. No caso dos outliers altos nas pontuações de maturidade entre 0-125, é notável pelo gráfico Total de Funções x FPM 7, que projetos nessa faixa tendem a ter um menor número total de funções, o que pode explicar a maior presença de outliers nas métricas dessa faixa de pntuação de maturidade. Como esses projetos são menores, variações nas métricas têm um impacto proporcionalmente maior, resultando em maior dispersão nos dados.

Apesar da diferença no tamanho dos projetos, observa-se que o tamanho médio das cadeias de erro se mantém relativamente constante conforme as faixas de pontuação de maturidade mudam. Isso sugere que os projetos com pontuação >250 evitam cadeias proporcionalmente maiores.

Figura 7 – Gráficos de Distribuições de Dados x Faixa de Pontuação de Maturidade (FPM)



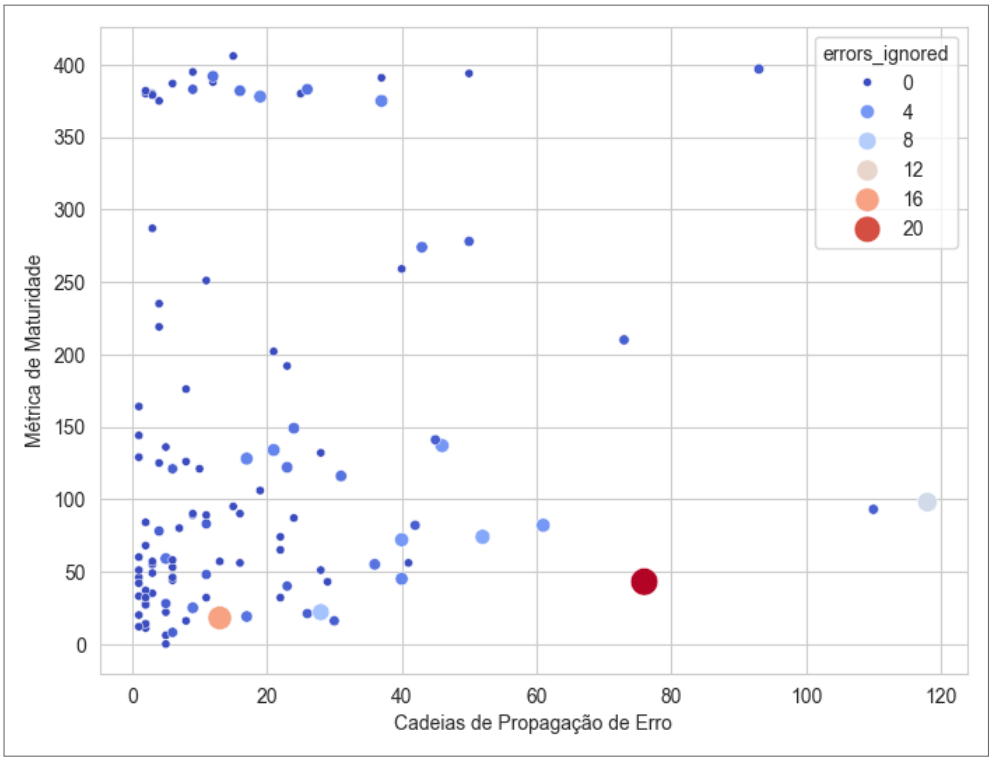
Fonte: Elaborado pela autora (2025)

Olhando a coorelação da Pontuação de Maturidade com Taxa de Erros Ignorados 6, é possível notar que projetos com pontuações menores podem acabar tendo menos erros ignorados. A partir da Figura 8, observa-se que não há uma relação de fato direta entre a maturidade do projeto e a quantidade de erros ignorados. No entanto, projetos com menor pontuação de maturidade tendem a apresentar uma maior variação nessa métrica, incluindo alguns casos onde

um número significativo de erros é ignorado. Esse comportamento é mais evidente em projetos com um maior número de cadeias de propagação de erro.

Por outro lado, em projetos com maior pontuação de maturidade, o número de erros ignorados permanece consistentemente baixo, geralmente variando entre 0 e 4. Isso pode indicar uma abordagem mais estruturada para o tratamento de erros, onde falhas são gerenciadas de forma mais eficiente ao longo do fluxo do programa, evitando que erros sejam descartados.

Figura 8 – Relação entre Cadeias de Propagação de Erro, Erros Ignorados e Score



Fonte: Elaborado pela autora (2025)

4.4 ANÁLISES COMPLEMENTARES

Além das análises voltadas para responder às perguntas principais deste trabalho, alguns resultados adicionais emergiram da correlação entre diferentes métricas de erro e novas dúvidas levantadas ao longo da pesquisa.

4.4.1 Relação entre Possíveis Panics e Cadeias de Propagação

A análise revelou uma forte correlação entre o número de possíveis *panics* e a quantidade de cadeias de propagação de erro, atingindo 83%. Esse valor indica que, nos repositórios analisados, a ocorrência de panics está diretamente relacionada à propagação de erros. Além disso, tanto os *panics* quanto as cadeias de erro demonstraram uma correlação significativa com o número total de funções no projeto:

- Cadeias de propagação de erro × Total de funções: 74%
- Possíveis *panics* × Total de funções: 72%

Isso sugere que, à medida que o código cresce e o número de funções aumenta, tanto a propagação de erros quanto os *panics* se tornam mais frequentes. No entanto, ao analisar taxas em vez de números absolutos, observa-se que essa relação não é puramente proporcional. A taxa de possíveis *panics* teve uma correlação de apenas 23% com o número total de funções, enquanto sua relação com a taxa de cadeias de erro foi 57%, um valor significativamente maior.

Esse resultado indica que o crescimento do código por si só não é o principal fator que determina o aumento dos *panics*. Em vez disso, eles parecem estar fortemente ligados à maneira como os erros se propagam dentro do código.

Outro ponto relevante é a correlação negativa de -46% entre a taxa de propagação de erros e a taxa de possíveis *panics*. Esse resultado sugere que quanto maior a taxa de possíveis panics em um código, menor é a taxa de propagação de erros. Isso pode indicar que as funções responsáveis por causar panics são as mesmas que tratam os erros, eliminando a necessidade de propagação. Esse comportamento sugere que muitos desenvolvedores podem estar utilizando funções como **expect()**, **unwrap()** e **assert!()** para lidar com erros, mesmo que essas funções apresentem riscos significativos de falha em tempo de execução.

Esses achados levantam uma questão fundamental: o aumento do número de funções que podem causar panic está associado a quais padrões específicos de implementação e práticas de desenvolvimento? Entender essa dinâmica pode ajudar a diferenciar entre o uso consciente de funções que podem causar panics e um possível padrão problemático de tratamento de erros em código Rust.

Para aprofundar essa observação, foi realizada uma análise qualitativa nos três projetos com maior número de panics, o que permitiu identificar alguns padrões recorrentes no uso de **expect()** e **unwrap()**. O **expect()** foi frequentemente utilizado em trechos de código que deveriam

ser inacessíveis em condições normais de execução, servindo como uma garantia de que a execução falharia caso alguma invariável fosse violada. Além disso, um uso comum do **expect()** foi identificado em códigos que deveriam falhar propositalmente, como em testes automatizados, ou em cenários com variáveis estáticas, como em possíveis falhas na leitura de configurações que deveriam interromper a execução do programa.

Já o **unwrap()** apareceu em diversas situações, muitas vezes acompanhado de verificações prévias para garantir que o valor não fosse *Err* ou *None*. No entanto, também foram encontrados casos onde o uso era inadequado, incluindo trechos de código marcados com **// TODO: cleanup**, indicando que os desenvolvedores tinham intenção de substituir essa abordagem no futuro. Um caso notável foi a presença de **unwrap()** em *locks* de concorrência, onde um erro pode levar a *panic* em situações como *deadlocks*.

Esses padrões ajudam a contextualizar por que repositórios com cadeias de propagação de erro mais extensas apresentam um número maior de possíveis *panics*. Esses projetos geralmente envolvem uma estrutura mais complexa, exigindo mais testes, uso intensivo de configurações e variáveis de ambiente e gerenciamento de concorrência, fatores que naturalmente elevam a necessidade de mecanismos explícitos para lidar com falhas. Entretanto, esses achados também sugerem que, em muitos casos, há oportunidades para melhorar a forma como os panics são utilizados, substituindo **unwrap()** por métodos mais seguros e garantindo que falhas críticas sejam devidamente registradas.

4.4.2 Erros Ignorados

A presença de repositórios com um número significativamente elevado de erros ignorados motivou uma análise mais detalhada dos cinco projetos com maior incidência desse fenômeno. Um aspecto positivo identificado foi que poucas eram as situações nas quais projetos analisados utilizava **Err(_)** para descartar completamente os erros, o que contrariou uma das hipóteses iniciais da pesquisa.

Ainda assim, observou-se que a forma como as mensagens eram tratadas apresentava algumas fragilidades. Entre os principais problemas identificados, destacam-se: erros que eram registrados em logs com mensagens totalmente novas, propagação de erros sem qualquer mensagem explicativa e erros tratados como condições comuns. A prática mais recorrente foi a substituição do erro original por uma string genérica, comprometendo a rastreabilidade ao longo do fluxo de propagação. Esse tipo de abordagem dificulta a depuração, pois torna difícil entender

onde e como cada erro ocorreu.

Todos os cinco projetos analisados apresentavam pontuações de maturidade entre 0 e 100. Para obter uma nova perspectiva, foram examinados os cinco projetos com maiores índices de maturidade. Nesses casos, apenas dois apresentavam erros ignorados, e mesmo assim, em menor quantidade. Além disso, quando erros eram descartados, isso ocorria de maneira mais controlada e documentada. Em alguns projetos, os erros com mensagens ignoradas eram propagados dentro de estruturas mais adequadas ao contexto, enquanto outros adotavam valores de fallback em situações esperadas. Nos casos em que None era retornado, a prática seguia um padrão previsível e documentado, sendo aplicada apenas quando a ausência do valor era uma possibilidade prevista no fluxo do programa. Essas abordagens reduzem a perda de informação e garantem maior previsibilidade ao comportamento do código. Assim, embora ignorar mensagens de erro nem sempre represente uma falha, é essencial que essa prática seja realizada com critério, garantindo que a estrutura da linguagem seja utilizada de forma coerente e segura.

4.4.3 Detecção de Erros Dinâmicos em Executáveis CLI

Durante a fase de coleta de dados a ferramenta retornou alguns projetos com 500 a 1000 estrelas que não apresentavam cadeias de propagação de erro. Para entender melhor esse fenômeno, foram selecionados 10 projetos nessa categoria para uma investigação mais detalhada. Desses, 9 eram executáveis CLI, e, ao contrário do que a ferramenta indicava, esses projetos de fato possuíam propagação de erro. No entanto, os erros eram tratados de maneira dinâmica, o que impediu sua detecção pela ferramenta.

Os erros dinâmicos são aqueles que não seguem um tipo estático e estruturado (**Result<T, E>** com um enum de erro), mas sim são encapsulados em tipos genéricos, como **Box<dyn Error>** ou **anyhow::Error**[16]. Esse tipo de tratamento é comum em executáveis CLI, pois prioriza a simplicidade e a flexibilidade. Como esses programas são executados apenas uma vez e não precisam manter estado, o tratamento de erro não precisa ser altamente estruturado. Nesses casos, basta que o erro seja capturado e exibido para o usuário antes da finalização do programa. Assim, a ausência de propagação explícita não compromete a robustez do sistema.

Essa característica explica por que a ferramenta de análise não identificou corretamente a propagação de erro nesses projetos, uma vez que foi projetada para detectar propagação explícita baseada em tipos de erro estruturados.

5 CONCLUSÃO

Este trabalho investigou a propagação e o tratamento de erros na linguagem Rust, analisando repositórios de código aberto por meio de análise estática. O objetivo foi compreender como os erros são tratados pelos desenvolvedores e identificar possíveis padrões que possam influenciar a qualidade do software. Isso foi feito focando na frequência dos erros, a distância entre sinalização e tratamento de erro e coorelações entre a métrica de maturidade e as de erro.

Os resultados indicam que a propagação de erros é um fenômeno comum nos projetos analisados, mas ocorre de maneira relativamente controlada, com a maioria dos erros sendo tratados após 2-4 chamadas de função. Porém, à medida que as cadeias crescem, a propagação de erros pode se tornar desorganizada e mais longa. Além disso, não foi encontrada uma correlação forte entre a maturidade do projeto e nenhuma das métricas de erro. Isso sugere que as restrições impostas pelo modelo obrigatório de gerenciamento de erros em Rust desempenham um papel fundamental na padronização das abordagens, auxiliando desenvolvedores menos experientes a manterem boas práticas na propagação e tratamento de erro.

Outros achados relevantes incluem o uso frequente das funções **unwrap()** e **expect()** em testes, variáveis estáticas e concorrência, demonstrando que mesmo sendo funções que podem causar *panic* em todo sistema, podem ser muito úteis em muitos casos específicos. Além disso, a investigação sobre mensagens de erro ignoradas revelou que, embora ainda existam casos de uso não ideal, raramente os erros são totalmente descartados. O mais comum é a substituição de mensagens por strings genéricas, o que compromete a rastreabilidade do erro. Por fim, observou-se que, em executáveis CLI, os erros dinâmicos são amplamente utilizados, indicando uma abordagem mais simples, mas ainda válida nesse contexto. Esses achados destacam a diversidade de estratégias de tratamento de erro e a necessidade de métodos mais abrangentes para analisá-las, já que foram identificados tanto pela análise direta do código quanto pela correlação com os dados levantados.

Entre as limitações deste estudo, destaca-se a dependência da ferramenta *static-error-analyzer*. Embora eficiente, foi mostrado que ainda existem maneiras de propagar erro que ela não captura, além da sua utilização ser restringida pelo uso do *rust-nightly* e pela necessidade de compatibilidade com *Cargo build*. Como consequência, apenas um subconjunto específico de projetos pôde ser analisado, o que pode ter gerado um viés na seleção da amostra. Futuras pesquisas podem explorar análises semelhantes com uma amostra normalizada.

Como trabalhos futuros, sugere-se a ampliação da análise para um conjunto maior de repositórios, combinando abordagens de análise estática e dinâmica para capturar um panorama mais abrangente da propagação de erros. Além disso, seria interessante um estudo detalhado das dificuldades enfrentadas na execução da ferramenta para aprimorá-la e reduzir suas restrições.

Dessa forma, este estudo contribui para o entendimento das práticas adotadas na comunidade Rust, fornecendo insights que podem auxiliar pesquisadores e desenvolvedores na melhoria da confiabilidade dos sistemas escritos nesta linguagem. Além disso, reforça a importância de ferramentas de análise estática para a engenharia de software, abrindo caminho para estudos futuros que busquem aprimorar a detecção e prevenção de falhas em linguagens seguras como Rust.

REFERÊNCIAS

[1] CASSEE, N. et al. How swift developers handle errors. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, 2018. (MSR '18), p. 292–302. ISBN 9781450357166. Disponível em: <https://doi.org/10.1145/3196398.3196428>.

[2] EBERT, F.; CASTOR, F.; SEREBRENIK, A. An exploratory study on exception handling bugs in java programs. *Journal of Systems and Software*, v. 106, p. 82–101, 2015. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121215000862>.

[3] BONIFÁCIO, R. et al. The use of c++ exception handling constructs: A comprehensive study. *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, p. 21–30, 2015. Disponível em: <https://api.semanticscholar.org/CorpusID:25083856>.

[4] XU, H. et al. Memory-safety challenge considered solved? an in-depth study with all rust cves. *ACM Trans. Softw. Eng. Methodol.*, Association for Computing Machinery, New York, NY, USA, v. 31, n. 1, set. 2021. ISSN 1049-331X. Disponível em: <https://doi.org/10.1145/3466642>.

[5] CHEN, C. et al. Oom-guard: Towards improving the ergonomics of rust oom handling via a reservation-based approach. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2023. (ESEC/FSE 2023), p. 733–744. ISBN 9798400703270. Disponível em: <https://doi.org/10.1145/3611643.3616303>.

[6] QIN, B. et al. Understanding and detecting real-world safety issues in rust. *IEEE Transactions on Software Engineering*, v. 50, n. 6, p. 1306–1324, 2024.

[7] Stack Overflow. *Stack Overflow Developer Survey 2024: Technology*. 2024. Acessado em: 4 fev. 2025. Disponível em: <https://survey.stackoverflow.co/2024/technology#2-programming-scripting-and-markup-languages>.

[8] O'GRADY, S. The redmonk programming language rankings: June 2024. *RedMonk*, September 2024. Acesso em: 12 mar. 2025. Disponível em: <https://redmonk.com/sogrady/2024/09/12/language-rankings-6-24/>.

[9] BLANDY, J.; ORENDORFF, J. *Programming Rust: Fast, Safe Systems Development*. 1st. ed. [S.l.]: O'Reilly Media, 2017. ISBN 978-1491927281.

[10] FU, C.; RYDER, B. G. Exception-chain analysis: Revealing exception handling architecture in java server applications. In: *Proceedings of the 29th International Conference on Software Engineering*. USA: IEEE Computer Society, 2007. (ICSE '07), p. 230–239. ISBN 0769528287. Disponível em: <https://doi.org/10.1109/ICSE.2007.35>.

[11] CACHO, N. et al. Trading robustness for maintainability: an empirical study of evolving c# programs. In: *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014), p. 584–595. ISBN 9781450327565. Disponível em: <https://doi.org/10.1145/2568225.2568308>.

[12] EDWARDS, S. H.; KANDRU, N.; RAJAGOPAL, M. B. Investigating static analysis errors in student java programs. In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*. New York, NY, USA: Association for Computing Machinery, 2017. (ICER '17), p. 65–73. ISBN 9781450349680. Disponível em: <https://doi.org/10.1145/3105726.3106182>.

[13] EGIETJE. *Rust Static Error Analyzer*. 2025. Acessado em: 4 fev. 2025. Disponível em: <https://github.com/Egietje/rust-static-error-analyzer>.

[14] Kas, T. *Static Analysis of Rust Error Propagation*. 2024. Disponível em: <http://essay.utwente.nl/100758/>.

[15] SOMMERVILLE, I. *Software Engineering*. 9. ed. Boston, MA: Addison-Wesley, 2011. ISBN 978-0-13-703515-1.

[16] KLABNIK, S.; NICHOLS, C.; COMMUNITY, T. R. *The Rust Programming Language*. Rust Project Developers, 2025. Acessado em: 4 fev. 2025. Disponível em: <https://doc.rust-lang.org/book/>.

[17] PÁDUA, G. B. de; SHANG, W. Studying the prevalence of exception handling anti-patterns. In: *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 2017. (ICPC '17), p. 328–331. ISBN 9781538605356. Disponível em: <https://doi.org/10.1109/ICPC.2017.1>.

[18] Rust Community. *What is rustc?* 2025. Acesso em: 4 fev. 2025. Disponível em: <https://doc.rust-lang.org/rustc/what-is-rustc.html>.

[19] GIRNUS, P. *Rust vs. C/C++: Ensuring Memory Safety & Security*. 2025. Accessed: 2025-03-03. Disponível em: <https://www.petergirnus.com/blog/rust-vs-cc-ensuring-memory-safety-and-security>.

[20] GARCÍA-FERREIRA, I. et al. Static analysis: a brief survey. *Logic Journal of the IGPL*, v. 24, n. 6, p. 871–882, 2016.

[21] THOMSON, P. Static analysis: An introduction: The fundamental challenge of software engineering is one of complexity. *Queue*, Association for Computing Machinery, New York, NY, USA, v. 19, n. 4, p. 29–41, set. 2021. ISSN 1542-7730. Disponível em: <https://doi.org/10.1145/3487019.3487021>.

[22] InnerSource Commons. *Padrões de InnerSource*. 2025. Acessado em: 07 jul. 2025. Disponível em: <https://patterns.innersourcecommons.org/pt-br>.

[23] HUMBLE, J.; FARLEY, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. [S.l.]: Addison-Wesley, 2010. (Addison-Wesley Signature Series (Fowler)). ISBN 978-0321601919.

[24] RAYMOND, E. S. *A Catedral e o Bazar: Reflexões sobre Linux e a Revolução do Código Aberto*. [S.l.]: O'Reilly Media, 1999. ISBN 978-0596001087.

[25] GitHub. *GitHub: Where the world builds software*. 2025. Acessado em: 2 mar. 2025. Disponível em: <https://github.com>.

[26] UNOFFICIAL, R. *Awesome Rust*. 2025. GitHub repository. Disponível em: <https://github.com/rust-unofficial/awesome-rust>.

[27] REBACK, J. et al. *pandas-dev/pandas: Pandas*. 2020. Disponível em: <https://pandas.pydata.org/>.

[28] SAMPAIO, N. A. d. S.; ASSUMPÇÃO, A. R. P. d.; FONSECA, B. B. d. *Estatística Descritiva*. Belo Horizonte: Editora Poisson, 2018. Disponível em: <https://www.poisson.com.br/livros/estatistica/volume1/Estatistica_Descritiva.pdf>.