Programmiersprachen im Vergleich

Artur Papoyan

January 31, 2026

Abstract

Rust is a modern systems programming language that was designed to address long-standing safety and reliability issues in low-level software development. Traditional systems languages such as C and C++ provide fine-grained control over memory and hardware, but they also allow many classes of errors, especially memory safety violations and data races, which are difficult to detect and debug. Rust approaches this problem by enforcing strict safety guarantees at compile time while avoiding a garbage collector. Its core mechanisms, such as ownership, borrowing, and a strong static type system, aim to prevent undefined behavior without sacrificing performance. In recent years, Rust has gained significant adoption in both industry and academia, and its claims of memory safety and "fearless concurrency" have been the subject of formal and empirical analysis. This paper presents an overview of Rust's design principles, language features, and ecosystem, and discusses how these aspects support safe and efficient systems programming[4, 3, 2].

1 Introduction

Motivation Systems software forms the foundation of modern computing, including operating systems, browsers, network services, and embedded systems. Errors in such software can have severe consequences, ranging from security vulnerabilities to system crashes. A large portion of these problems is caused by memory safety issues such as use-after-free errors, buffer overflows, and data races. Languages like C and C++ offer high performance and direct control over system resources, but they provide only limited built-in mechanisms to prevent these errors. As a result, correctness heavily depends on developer discipline and extensive testing.

Rust was created to address this situation by combining low-level control with strong compile-time safety guarantees. Its design is based on the idea that many common programming errors can be ruled out statically, before the program is executed. In particular, Rust focuses on memory safety without relying on a garbage collector and on preventing data races in concurrent programs. These goals are achieved through a combination of language features, most notably ownership, borrowing, and a strict type system[4, 3]. Over time, these ideas have been formalized and evaluated, for example through the RustBelt project, which provides a soundness argument for Rust's core safety guarantees[2]. More recent work has examined whether Rust's promise of safe and efficient concurrency holds in practice, especially for parallel programs[1].

Overview of the paper structure The remainder of this paper is structured as follows. The first part introduces the origins of Rust and its design philosophy, including its historical background and the role of its governing organizations. This is followed by a discussion of Rust's ownership and borrowing model, which forms the foundation of its memory safety guarantees. The next part explains how Rust manages memory without the use of a garbage collector and contrasts this approach with garbage-collected languages.

Afterwards, the paper focuses on concurrency and synchronization in Rust, covering concepts such as fearless concurrency, zero-cost abstractions, and the role of traits like `Send` and `Sync`. The subsequent part presents important characteristics of Rust's type system and its mechanisms for error handling. The tooling and ecosystem surrounding Rust, including the package manager and documentation infrastructure, are then outlined.

Finally, the paper will present the practical part of this work and conclude with a summary of the main findings and limitations[3, 1].

2 Rust Highlights

Rust is designed as a systems programming language that combines low-level control with strong safety guarantees. Its main distinguishing features are not individual syntax elements, but rather a set of design principles that influence the entire language. These principles aim to reduce common programming errors while preserving performance comparable to traditional systems languages. Many of Rust's highlights are closely connected and build upon each other, especially in the areas of memory management, concurrency, and type safety[4, 3].

2.1 Origin and Design Philosophy

History and Background Rust was originally developed as a personal project by Graydon Hoare in 2006. The language was later adopted and officially supported by Mozilla, where it evolved into a larger open-source project. The initial motivation behind Rust was to address recurring problems observed in

large systems software, particularly memory safety errors and concurrency bugs, which were common in existing low-level languages such as C and C++[4, 5].

From its early stages, Rust was designed with a strong focus on safety enforced at compile time rather than at runtime. This approach was intended to prevent entire classes of errors before a program is executed. Over several years of development, Rust's design was refined through community feedback and practical experience. The first stable release, Rust 1.0, marked a point where the language was considered mature enough for production use, while still continuing to evolve in later versions[3, 5].

An important aspect of Rust's history is its transition from a research-driven project to a widely used industrial language. While Mozilla played a central role in its early development, Rust gradually became governed by a broader community and formal organizational structures. This evolution reflects Rust's growing relevance beyond experimental settings and highlights its ambition to serve as a general-purpose systems programming language with strong safety guarantees[3, 5].

Goal: Memory Safety without Garbage Collector One of the central goals of Rust's design is to provide strong memory safety guarantees without relying on a garbage collector. Traditional garbage-collected languages simplify memory management by automatically reclaiming unused memory at runtime, but this often introduces runtime overhead and unpredictable pauses. Rust deliberately avoids this approach in order to remain suitable for systems programming, where predictable performance and low-level control are essential[4, 3].

Instead of a garbage collector, Rust enforces memory safety through compile-time rules based on ownership and borrowing. These rules ensure that each value has a clear owner and that memory is released deterministically when this owner goes out of scope. By shifting memory management decisions to compile time, Rust aims to eliminate common errors such as use-after-free and double-free bugs while maintaining performance comparable to languages like C and C++[4, 2]. This design choice reflects Rust's broader philosophy of preventing errors early, rather than detecting them at runtime.

Role of the Rust Foundation As Rust grew from an experimental project into a widely used programming language, its governance structure also evolved. Today, the development of Rust is supported by the Rust Foundation, which provides organizational and legal backing for the language and its ecosystem. The foundation's role is to ensure the long-term sustainability of Rust by supporting its open-source development and coordinating contributions from industry and the community[3, 5].

The Rust Foundation does not control the technical direction of the language in a centralized manner. Instead, Rust continues to be developed through open processes involving working groups and community discussion. This governance model reflects Rust's origins as a community-driven project and helps balance stability with continued innovation. By separating technical decision-making from legal and organizational responsibilities, the Rust Foundation supports the ongoing evolution of the language while preserving its core design principles[3, 5].

2.2 Ownership and Borrowing

Ownership model: who owns what? The ownership model is a central concept in Rust and forms the basis of its memory safety guarantees. In Rust, every value has exactly one owner at any point in time. The owner is responsible for the value and determines when the associated memory is released. When the owner goes out of scope, the value is automatically dropped and its memory is deallocated. This rule ensures that memory is freed exactly once and prevents errors such as double frees and use-after-free bugs[4, 3].

Ownership can be transferred, for example when a value is passed to a function or assigned to another variable. After such a move, the previous owner can no longer access the value. By enforcing these rules at compile time, Rust guarantees that there is always a clear and unambiguous responsibility for each piece of memory. This approach replaces manual memory management patterns commonly found in C and C++ with a statically checked discipline[4, 2].

Borrowing: shared and exclusive references While strict ownership rules provide safety, they would be too restrictive for practical programming without additional mechanisms. Rust therefore introduces borrowing, which allows temporary access to a value without transferring ownership. Borrowing is expressed through references, which can be either shared or exclusive. Shared references allow multiple readers to access a value at the same time, but they do not permit mutation. Exclusive references allow mutation, but only one such reference may exist at a time[3].

This distinction between shared and exclusive references enforces the principle that aliasing and mutation cannot occur simultaneously. By preventing mutable aliasing, Rust eliminates data races in safe code. These rules are checked entirely at compile time and do not introduce runtime overhead. Borrowing therefore enables flexible access to data while preserving the guarantees required for memory and thread safety[4, 2].

Borrow Checker and lifetimes The enforcement of ownership and borrowing rules is performed by the Rust compiler component known as the borrow checker. The borrow checker analyzes the program to ensure that all references are valid and that they do not outlive the data they refer to. To achieve this, Rust uses the concept of lifetimes, which describe the scope during which a reference is valid[3].

Lifetimes are not inferred in general. Instead, Rust applies a set of lifetime elision rules that allow lifetime annotations to be omitted in simple function signatures. These rules cover many common cases in practice, which allows programmers to write code without explicitly specifying lifetimes most of the time. However, when functions involve multiple references and return one of them, explicit lifetime annotations are required to make the relationship between references clear and verifiable.

Through this analysis, the borrow checker guarantees that references never become dangling and that memory safety is preserved. The soundness of this approach, even in the presence of certain unsafe abstractions, has been formally studied and justified in prior work [2].

Example: Ownership, Borrowing, and Lifetimes in Practice To better illustrate Rust's programming style and its approach to memory safety, this section presents a small but representative code example. The example focuses on borrowing and lifetimes, which are often considered one of Rust's most distinctive features.

Consider the following function:

Listing 1: Simple Rust function

```
fn longest<'a>(a: &'a str, b: &'a str) → &'a str {
    if a.len() > b.len() {
        a
    } else {
        b
    }
}
```

This function takes two string slices and returns the longer one. At first glance, the lifetime annotation $'a$ may seem unnecessary or confusing, especially for programmers coming from languages such as C or Java. However, this annotation plays a crucial role in ensuring memory safety. In Rust, references must always be valid. The compiler must be able to guarantee that the returned reference does not outlive the data it points to. In this example, both input references a and b have the same lifetime $'a$. By also assigning the same lifetime to the return type, the function explicitly states that the returned reference is valid only as long as both input references are valid.

Without this information, the compiler would not know which input reference the return value is related to. As a result, Rust requires the programmer to specify this relationship explicitly when it cannot be derived automatically.

This becomes clearer when looking at an incorrect version:

Listing 2: Simple Rust function

```
fn longest(a: &str, b: &str) → &str {
    if a.len() > b.len() {
        a
    } else {
        b
    }
}
```

This code does not compile. The compiler reports that it cannot infer the lifetime of the returned reference. Even though the logic of the function is correct, Rust rejects it because the lifetime relationship between the inputs and the output is ambiguous.

The explicit lifetime annotation does not change how the program runs at runtime. Lifetimes are a purely compile-time concept. They exist only to help the compiler reason about borrowing and to prevent common memory errors such as use-after-free.

This example also highlights a general design principle of Rust: safety guarantees are enforced statically, before the program is executed. While this may require additional annotations in some cases, it avoids entire classes of runtime errors that are common in low-level languages.

In practice, many functions do not require explicit lifetime annotations because Rust supports lifetime elision rules. These rules allow the compiler to omit lifetimes in simple and common cases. However, when a function takes multiple references and returns one of them, as shown here, explicit lifetimes are necessary to express the intended relationship clearly.

Overall, this example demonstrates how Rust combines explicit annotations with strong static analysis to achieve memory safety without garbage collection. While the syntax may appear unusual at first, it enables precise reasoning about resource usage and helps prevent subtle bugs in complex systems.

2.3 Memory Management Without Garbage Collector

Automatic deallocation via scope-based drop Rust manages memory without using a garbage collector by relying on deterministic, scope-based deallocation. This mechanism is closely tied to the ownership model. Each value in Rust has a single owner, and when this owner goes out of scope, the value is automatically dropped. During this drop operation, Rust releases the associated resources, including heap-allocated memory[3, 4].

This approach allows memory to be reclaimed at well-defined points in the program, without the need for runtime tracing or background cleanup. Because deallocation is determined statically by the program structure, Rust avoids unpredictable pauses that are common in garbage-collected systems. At the same time, the compiler ensures that values are dropped exactly once, which prevents memory errors such as double frees and dangling pointers[2]. As a result, Rust achieves memory safety while keeping memory management explicit and predictable.

Comparison to Garbage Collector In garbage-collected languages, memory management is typically

handled by a runtime system that periodically identifies and frees unused objects. While this simplifies programming, it introduces additional runtime overhead and can make performance behavior less predictable. Rust deliberately avoids this model in order to remain suitable for systems programming, where control over performance and resource usage is important[3, 4].

By enforcing memory safety through compile-time checks instead of runtime garbage collection, Rust shifts responsibility from the runtime system to the compiler. This design reduces runtime costs and allows Rust programs to achieve performance comparable to traditional systems languages. At the same time, formal analyses have shown that Rust's approach can provide strong safety guarantees, even in the presence of low-level abstractions, as long as the language's rules are respected[2].

2.4 Concurrency and Synchronization

Fearless Concurrency Concurrency is one of the main areas where Rust aims to improve over traditional systems programming languages. The Rust documentation introduces this goal under the term "fearless concurrency," which describes the idea that many concurrency errors should be detected at compile time rather than at runtime. In Rust, the same ownership and borrowing rules that ensure memory safety are also used to prevent data races in concurrent programs[3].

In safe Rust code, it is not possible for multiple threads to have unsynchronized mutable access to the same data. This eliminates data races by construction. As a result, programmers can write concurrent code with stronger guarantees that certain classes of errors cannot occur. However, recent empirical studies have shown that while Rust effectively prevents data races, it does not eliminate all concurrency-related challenges, especially in more complex parallel patterns[1].

Safety + Performance Rust's approach to concurrency is closely tied to its performance goals. Instead of relying on runtime checks or a virtual machine, Rust enforces safety properties at compile time. This design avoids runtime overhead while still providing strong guarantees. As a result, Rust programs can achieve performance comparable to C and C++ while offering stronger safety properties[4, 3].

Formal and empirical analyses have shown that Rust's safety mechanisms do not inherently require performance trade-offs. In many cases, safe abstractions compile down to efficient machine code. However, when programmers need to express more irregular parallelism, they may have to rely on synchronization primitives or unsafe code, which can reintroduce complexity and potential performance costs[1, 2].

Zero-Cost Abstractions A key principle in Rust's design is the use of zero-cost abstractions. This means that high-level language features should not impose additional runtime overhead compared to equivalent low-level code written manually. In the context of concurrency, this principle allows developers to use safe abstractions for threading and synchronization without paying a performance penalty[3].

Zero-cost abstractions rely on the compiler's ability to optimize away abstraction layers during compilation. Rust's type system and ownership model provide the necessary information for these optimizations. As a result, concurrency constructs in Rust can be both expressive and efficient, as long as they remain within the guarantees enforced by the type system[4].

Send/Sync Rust uses marker traits to express thread-safety properties at the type level. The two most important traits in this context are `Send` and `Sync`. A type that implements `Send` can be safely transferred between threads, while a type that implements `Sync` can be safely shared between threads through references. These traits are checked at compile time and form the basis of Rust's concurrency guarantees[3].

Most primitive types in Rust implement these traits automatically, while more complex types may not. This ensures that only data structures that are safe to use in concurrent contexts can be shared or transferred between threads. By encoding these properties in the type system, Rust prevents many common concurrency errors before the program is executed[2].

Threads, Channels Rust provides basic concurrency primitives such as threads and message-passing channels through its standard library. Threads allow multiple units of execution to run in parallel, while channels provide a safe way to communicate between threads without shared mutable state. Communication through channels follows the ownership model, as values are transferred from sender to receiver[3].

This message-passing approach reduces the need for shared mutable data and simplifies reasoning about concurrent programs. By combining threads with channels and ownership-based transfers, Rust encourages designs that avoid data races and make synchronization explicit and verifiable at compile time[4].

async/await and runtime ecosystem In addition to thread-based concurrency, Rust supports asynchronous programming through the `async` and `await` keywords. These features allow programmers to write non-blocking code in a sequential style. Asynchronous tasks are managed by runtime systems that schedule and execute them efficiently[3].

Rust's asynchronous model integrates with its ownership and type system, ensuring that memory safety and data race freedom are preserved even in asynchronous contexts. While the language itself provides the core syntax, the execution model is implemented by external runtime libraries. This separation allows flexibility in choosing different runtime implementations while maintaining the same safety guarantees[3, 1].

2.5 Type System and Language Characteristics

Static typing Rust is a statically typed programming language. This means that the types of all variables and expressions are known at compile time. Static typing allows the compiler to detect many classes of errors early, before the program is executed. In Rust, type checking works together with the ownership and borrowing rules to ensure memory safety and prevent invalid operations[3, 4].

Although Rust is statically typed, it reduces verbosity through type inference. In many cases, the compiler can infer types automatically, which keeps code readable while still benefiting from strong static guarantees. This balance allows Rust to combine safety with practical usability in large systems programs[3].

Traits and Generics Rust uses traits to define shared behavior across different types. A trait specifies a set of methods that a type must implement. This mechanism is similar to interfaces in other languages, but it is deeply integrated into Rust's type system. Traits are widely used to express abstraction and to enable code reuse without relying on inheritance[3].

Generics allow functions, structs, and enums to operate on multiple types while preserving static type safety. When generics are used together with traits, Rust can enforce constraints on type parameters at compile time. This design enables flexible and reusable code without introducing runtime overhead, which is consistent with Rust's zero-cost abstraction principle[4].

Pattern Matching Pattern matching is an important language feature in Rust that allows values to be compared against structured patterns. It is commonly used with enums and other composite types to handle different cases explicitly. Pattern matching helps ensure that all possible cases are considered, which improves program correctness[3].

The compiler checks pattern matches for exhaustiveness, meaning that missing cases are detected at compile time. This behavior reduces the likelihood of unhandled states and makes control flow more explicit. As a result, pattern matching supports Rust's overall goal of preventing errors through static analysis[3].

Error handling: Result, Option Rust uses explicit types for error handling instead of exceptions. The `Option` type represents the presence or absence of a value, while the `Result` type represents either a successful outcome or an error. By encoding these possibilities in the type system, Rust forces programmers to handle error cases explicitly[3].

This approach makes error handling visible in function signatures and prevents errors from being silently ignored. Combined with pattern matching, `Option` and `Result` encourage robust and predictable error handling strategies. This design aligns with Rust's emphasis on safety and explicitness, especially in systems-level code[3].

2.6 Tooling and Ecosystem

Cargo (Package Manager and Build System) Rust provides an integrated build system and package manager called Cargo. Cargo is used to manage dependencies, compile projects, run tests, and build documentation. It standardizes common development tasks and reduces the need for external tools or custom build scripts. By providing a single, unified interface, Cargo simplifies project setup and improves reproducibility across different systems[3].

Cargo also enforces a conventional project structure, which makes Rust projects easier to understand and maintain. This standardization supports collaboration and lowers the entry barrier for new developers. The tight integration between the language and its tooling is a key aspect of Rust's ecosystem[3].

Crates.io (Ecosystem) Crates.io is Rust's official package registry and hosts reusable libraries, known as crates. These crates can be easily added as dependencies using Cargo. The registry plays a central role in Rust's ecosystem by enabling code reuse and sharing across projects[3].

The ecosystem includes libraries for a wide range of tasks, including networking, concurrency, and data processing. Many of these libraries rely on Rust's safety guarantees and use interior abstractions to provide safe interfaces. This ecosystem supports Rust's adoption in both experimental and production settings[3].

Rustup and Toolchains Rustup is a tool for managing Rust toolchains and compiler versions. It allows developers to install, update, and switch between different versions of the Rust compiler. This is particularly useful for testing code against multiple compiler releases or using nightly features when required[3].

By separating the compiler from the system environment, rustup supports consistent development workflows across platforms. It also simplifies access to additional tools provided by the Rust project, such as formatting and linting utilities[3].

Rustdoc, Testing, Benchmarking Rust includes built-in support for documentation generation through rustdoc. Documentation comments written in the source code are converted into structured HTML documentation. This encourages developers to document APIs as part of the development process[3].

Testing is also integrated into the language and tooling. Cargo provides commands to run unit tests and integration tests, which are written using standard Rust syntax. This integration supports test-driven development and helps ensure correctness. Benchmarking support is available through dedicated tools, allowing developers to measure performance in a controlled way[3].

2.7 Current Developments

Rust continues to evolve through an open and community driven development process. New language features and improvements are introduced carefully, with a strong focus on backward compatibility and stability. This approach allows Rust to grow while preserving the guarantees that existing code relies on[3, 5].

Current development efforts focus on improving usability, expanding the standard library, and refining existing features. At the same time, research continues to analyze and formalize Rust's safety guarantees, particularly in areas such as concurrency and unsafe abstractions[2, 1].

2.8 Comparison with Other Languages

Rust vs. C++ Rust and C++ are both used for systems programming and provide low-level control over memory and performance. However, they differ significantly in how safety is enforced. In C++, memory safety and data race freedom depend largely on programmer discipline and runtime testing. Rust, in contrast, enforces strict safety rules at compile time through its ownership and type system[4, 2].

While both languages can achieve similar performance, Rust aims to prevent entire classes of errors before execution. Studies have shown that Rust's approach can significantly reduce memory-related bugs, although complex concurrency patterns may still require careful design and, in some cases, unsafe code[1]. As a result, Rust represents a different trade-off between control, safety, and complexity compared to C++[3].

3 Practical Implementation

The practical part of this work is the implementation of a simplified text search tool that recursively scans files and directories for a given regular expression. The program supports features such as contextual output, optional colored highlighting, and configurable handling of hidden files. The implementation serves as a concrete example to evaluate how Rust's language design and ecosystem support the development of a small but non-trivial systems program.

One of the most noticeable aspects during implementation was the strong support provided by Rust's standard library. Core functionality such as file access, buffered input, directory traversal, and command-line argument handling is available without relying on platform-specific APIs. These abstractions are designed to be safe by default, which reduces the likelihood of common programming errors. At the same time, they remain close enough to the underlying system interfaces to allow fine-grained control when needed[3].

Rust's type system influenced the structure of the program in a practical way. Many operations that may fail, such as opening files or reading directory entries, are represented explicitly in the types returned by library functions. This forces error handling decisions to be made deliberately and locally, rather than being deferred implicitly. As a result, failure cases become part of the program's control flow and are easier to reason about. This explicit style aligns with Rust's general design goal of making potentially unsafe situations visible at compile time[3].

The ownership model also affected the organization of the code, particularly in functions that process file contents and context lines. Data is passed between functions in a way that clearly defines which part of the program is responsible for it at any moment. Temporary buffers and intermediate results are dropped automatically when they are no longer needed. This allows deterministic resource management without manual memory handling, while still avoiding runtime garbage collection[4]. In practice, this made it easier to reason about resource lifetimes, especially when working with nested directory structures and multiple files.

External libraries were integrated through Rust's package management infrastructure. The regular expression functionality used in the program is provided by a reusable library obtained via the official package registry. Cargo handles dependency resolution, compilation, and integration automatically. This tight coupling between language and tooling reduces setup effort and helps maintain a clear project structure. From a practical perspective, this makes experimenting with additional functionality straightforward, while keeping builds reproducible[3].

Although the program itself is sequential, Rust's concurrency model still influenced design decisions. Shared mutable state was avoided by structuring the code around local processing and clearly scoped data. This matches Rust's philosophy of preventing data races by construction. Even without explicit parallelism, the guarantees enforced by the compiler make it easier to reason about potential future extensions, such as parallel file processing, without fundamentally restructuring the code[1].

Overall, the practical implementation confirms several observations made in the theoretical part of this work. Rust provides strong guidance through its type system and standard library, which shapes program structure toward safe and explicit designs. While this requires the developer to engage with the language's constraints, it also reduces ambiguity and hidden behavior. These characteristics are consistent with formal and empirical analyses of Rust's safety goals and demonstrate how they manifest in everyday systems programming tasks[2, 3].

4 Conclusion

This paper examined the Rust programming language from both a theoretical and a practical perspective. The goal was to understand how Rust addresses common problems in systems programming and how its design choices affect real-world software development.

By combining a discussion of Rust's core concepts with a practical implementation, this work provides a structured overview of Rust's strengths, limitations, and intended use cases.

At the theoretical level, Rust was shown to be strongly centered around memory safety without the use of a garbage collector. Its ownership and borrowing model enforces strict rules about how data can be accessed and modified. These rules are checked at compile time and prevent many classes of memory errors, such as use-after-free and double-free bugs. Formal work such as RustBelt demonstrates that these guarantees are not only intuitive design goals, but can also be justified rigorously, even in the presence of low-level abstractions[2]. This places Rust in a unique position among systems programming languages.

Another central theme of this work was concurrency. Rust promotes the idea of "fearless concurrency" by using the same ownership and type-based rules to prevent data races. In safe Rust code, unsynchronized shared mutable access is not possible, which eliminates an important source of concurrency bugs. However, recent empirical studies show that while Rust effectively prevents data races, it does not remove all challenges related to parallel programming. More complex access patterns and irregular parallelism still require careful design and, in some cases, the use of synchronization primitives or unsafe code[1]. This highlights that Rust improves safety, but does not eliminate the inherent complexity of concurrent systems.

The discussion of Rust's type system further showed how safety and expressiveness are combined. Static typing, traits, generics, and pattern matching contribute to making program behavior explicit and verifiable. Error handling through types such as `Result` and `Option` encourages developers to deal with failure cases explicitly. These mechanisms support Rust's broader philosophy of making potential errors visible early, rather than hiding them behind implicit runtime behavior[3].

The practical implementation of a search tool served as a concrete example of how these concepts influence everyday programming. Rather than focusing on implementation details, the practical part demonstrated how Rust's language features guide program structure. Ownership and explicit error handling shaped how data and failures were handled, while the standard library and tooling ecosystem simplified interaction with the file system and external libraries. The use of Cargo and crates from the official registry showed how Rust integrates language and tooling in a coherent way, reducing setup complexity and supporting reproducible builds[3].

One important observation from the practical work is that Rust encourages deliberate design decisions. The compiler enforces constraints that may initially slow down development, especially for programmers new to the language. However, these constraints also reduce ambiguity and hidden behavior. As a result, the final program benefits from stronger guarantees about correctness and resource usage. This confirms Rust's goal of shifting effort from debugging runtime errors to reasoning at compile time[4]. At the same time, this work also highlights limitations. Rust's strict rules can make certain programming patterns harder to express, particularly in advanced concurrent scenarios. While unsafe code and synchronization primitives provide escape hatches, they reintroduce some of the risks that Rust aims to avoid. This aligns with existing research, which shows that Rust improves safety but cannot fully remove the challenges of complex parallel systems[1]. Therefore, Rust should be seen as a language that reduces, but does not eliminate, the need for careful system design.

In summary, Rust represents a significant step forward in systems programming language design. Its combination of compile-time safety guarantees, predictable performance, and a strong ecosystem makes it a compelling alternative to traditional languages such as C and C++. The results of this work support the view that Rust's design principles are not only theoretically sound, but also practically relevant. At the same time, Rust remains a language that requires expertise and careful thinking, especially in advanced use cases. These trade-offs define Rust's role as a modern systems programming language that prioritizes safety without abandoning control[3, 2].

References

[1] Javad Abdi et al. "When Is Parallelism Fearless and Zero-Cost with Rust?" In: *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2024, pp. 1–14. DOI: 10.1145/3626183.3659966.

[2] Ralf Jung et al. "RustBelt: Securing the Foundations of the Rust Programming Language". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2018), 66:1–66:34. DOI: 10.1145/3158154.

[3] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. Official Rust documentation. No Starch Press, 2019.

[4] Nicholas Matsakis and Felix Klock II. "The Rust Language". In: *ACM SIGAda Ada Letters* 34.3 (2014), pp. 103–104. DOI: 10.1145/2692956.2663188.

[5] Wikipedia contributors. *Rust (programming language)*. Accessed: 2026-01-08. 2025. URL: https://de.wikipedia.org/wiki/Rust_(Programmiersprache).