Rust Usage in Data Science

*

Universität Siegen
Siegen, Germany

ABSTRACT

This paper examines the growing role of Rust in the data science ecosystem. Rust's unique blend of performance, memory safety, and concurrency features makes it an attractive alternative to traditional data science languages like R, Python, and C++. This paper explores Rust's advantages in handling large datasets, performing complex computations, and ensuring reliable concurrent processing. The paper compares Rust with R, highlighting Rust's superior performance characteristics due to its compiled nature, efficient memory management, and optimized data structures. The paper also compares Rust with C/C++, emphasizing Rust's memory safety guarantees, zero-cost abstractions, and improved developer productivity. While Rust's statistical and visualization libraries are still under development, its rapid growth and the emergence of powerful crates like *ndarray* suggest a promising future for Rust in data science. As the ecosystem matures, Rust is poised to become an increasingly compelling choice for data scientists seeking performance, reliability, and control.

KEYWORDS

Rust, Data Science, Performance, Concurrency, Memory Safety, Rust Python, R, C/C++

1 INTRODUCTION RUST IN DATA SCIENCE

This paper explores Rust's strengths and weaknesses in the context of data science, examining its performance benefits. Afterward, it discovers its usage in real-world projects, compares it with other popular languages, and discusses its challenges and limitations.

2 ADVANTAGES OF RUST IN DATA SCIENCE

One of the key advantages of Rust in data science is its strong focus on safety and performance. Rust was designed to be a "safe" language, meaning it enforces memory safety without the need for a garbage collector, reducing the risk of common programming errors like null pointer dereferences and data races [Bugden and Alahmar 2022]. This makes Rust a particularly suitable choice for developing high-performance, concurrent data processing pipelines.

Moreover, Rust's performance characteristics are impressive, often outperforming well-established languages like C, C++, Java, and Python in various benchmarks [Bugden and Alahmar 2022]. This makes Rust an attractive option for computationally-intensive data analysis tasks, where performance is critical.

Rust's ecosystem of third-party packages, or "crates," is another factor contributing to its growing popularity in data science. The Rust community has developed a wide range of data-related crates, covering areas such as numerical computing, machine learning, and data visualization. This abundance of high-quality libraries and tools in the Rust ecosystem can significantly enhance the productivity and capabilities of data scientists working with the language.

Author's address:, Universität Siegen, Siegen, Germany.

3 DATA HANDLING AND PROCESSING WITH RUST

Rust's performance and safety features make it exceptionally well-suited for handling and processing large datasets in data science. Its approach to data handling and processing distinguishes itself through a focus on efficiency, safety, and control, enabled by its core concepts of ownership, borrowing, and zero-cost abstractions.

Theorem: Rust's ownership system, borrowing mechanisms, and zero-cost abstractions enable efficient and safe data handling and processing.
Proof:

Ownership and Borrowing: Rust's ownership system ensures memory safety by preventing data races and dangling pointers [Saligrama et al. [n. d.]]. Every value has a single owner at any given time. When the owner goes out of scope, the value is dropped. Borrowing allows temporary access to data without transferring ownership, enabling efficient data sharing [Saligrama et al. [n. d.]]. Mutable borrows provide exclusive access for modification, while immutable borrows allow shared access. This system guarantees data consistency and prevents common concurrency issues.

Zero-Cost Abstractions: Rust empowers developers to write high-level, expressive code without sacrificing performance through zero-cost abstractions. These abstractions compile down to efficient machine code, providing the benefits of both high-level programming and low-level control [Lyu and Rzeznik 2023]. Iterators, for example, provide a convenient way to process collections of data without the overhead of explicit loops.

Memory Management: Rust's ownership and borrowing system eliminates the need for garbage collection [Perkel 2020]. This results in predictable performance, essential for data processing tasks where consistent execution times are crucial. Manual memory management, guided by the ownership system, allows for fine-grained control over memory allocation and deallocation.

Conclusion: Rust's features combine to create a powerful environment for data handling and processing. The ownership system and borrowing mechanisms guarantee memory safety and prevent data races [Qin et al. 2020], while manual memory management and optimized data structures contribute to high performance. Zero-cost abstractions enable expressive and efficient code, making Rust a compelling choice for data-intensive tasks. The growing ecosystem for AI and scientific computing further strengthens Rust's position in the data science landscape [Xu 2024].

4 RUST'S PERFORMANCE BENEFITS

Rust's performance advantages stem from its meticulous design, combining low-level control and sophisticated optimizations. Its ability to achieve near-native performance without compromising safety and ergonomics is a significant draw for data scientists seeking efficient tools for computationally demanding tasks [?].

4.1 Fast Parsing and Data Manipulation

Rust's efficient parsing and data manipulation capabilities have proven valuable in various domains, including computational biology. In one benchmark involving parsing 5.7 million sequence records, Rust demonstrated superior performance, even outperforming C, a language traditionally favored for high-performance computing [?]. This efficiency is partly due to Rust's zero-cost abstractions, which minimize runtime overhead compared to languages like Python [?].

4.2 Concurrency and Parallelism

Rust's robust concurrency and parallelism features are particularly advantageous in data-intensive applications. Its ownership and borrowing system, combined with thread-safe primitive types, enables the development of scalable and reliable concurrent data processing pipelines [Saligrama et al. [n. d.]]. This system prevents data races at compile time, eliminating a common source of bugs and instability in concurrent programs written in languages like C++ [Qin et al. 2020] [Reed 2015]. This allows developers to confidently exploit multi-core processors and distributed computing resources, maximizing performance for large-scale data analysis.

4.3 Data Structures and Libraries

Rust's commitment to zero-cost abstractions shines through in its data structure libraries. ndarray [Xu 2024], for instance, provides an N-dimensional array type analogous to NumPy in Python, but with the performance benefits of Rust's ownership and borrowing system. This allows for efficient manipulation of large datasets without the overhead of garbage collection or dynamic typing. Other crates like nalgebra offer optimized linear algebra routines, further enhancing Rust's capabilities for numerical computation.

4.4 Compiler Optimizations

Rust leverages the LLVM compiler infrastructure, which performs a wide range of optimizations, including inlining, loop unrolling, and vectorization. These optimizations, combined with Rust's static typing and memory management, result in highly efficient machine code. The ability to compile to WebAssembly also broadens Rust's reach, allowing for performant code execution in web browsers [Lyu and Rzeznik 2023].

4.5 Interoperability with other languages

Rust's Foreign Function Interface enables seamless integration with code written in other languages, particularly C/C++. This allows Rust programs to leverage existing high-performance libraries and tools written in C/C++, such as optimized numerical routines or specialized hardware interfaces. This interoperability extends Rust's capabilities and allows for gradual adoption within existing data science workflows.

4.6 Profiling and Benchmarking Tools

Rust provides tools such as cargo bench and perf for profiling and benchmarking code. These tools enable developers to identify performance bottlenecks and optimize their code effectively. By measuring execution times and resource usage, developers can fine-tune their Rust code for optimal performance in specific data science applications.

5 CHALLENGES AND LIMITATIONS

While Rust presents a compelling case for data science, it's essential to acknowledge its challenges and limitations:

Steep Learning Curve: Rust's ownership and borrowing system, while crucial for memory safety and concurrency, introduces a significant learning curve for developers accustomed to garbage-collected languages like Python or R [Qin et al. 2020] . This initial hurdle can require a substantial time investment before developers become proficient.

Ecosystem Maturity: The Rust data science ecosystem, while burgeoning, is still relatively nascent compared to the mature and extensive libraries and tools available in Python and R [Xu 2024]. This can sometimes limit the availability of specialized packages for specific data science tasks, potentially requiring developers to implement custom solutions or rely on less mature crates.

Limited Library Availability: Although the Rust ecosystem is expanding rapidly, there may be instances where a specific library or functionality isn't readily available. This can necessitate using foreign function interfaces to bridge with existing C/C++ libraries or developing workarounds, potentially impacting development speed and code complexity [Xu 2024].

6 RUST IN REAL-WORLD PROJECTS

Despite these challenges, Rust is gaining traction in real-world data science projects, demonstrating its growing maturity and applicability. For example, the Dana-Farber Cancer Institute in Boston, Massachusetts, has successfully leveraged Rust's performance advantages in computational biology tasks, showcasing its potential in the field of bioinformatics [Perkel 2020].

In the broader context, Rust's adoption by major tech companies, such as Microsoft's exploration of using Rust to mitigate security vulnerabilities in system software, underscores its increasing relevance and credibility as a systems programming language [?].

While detailed public information about specific real-world data science projects using Rust is scarce, here are some sample types of projects where Rust's strengths can be leveraged:

Bioinformatics and Genomics: Rust's speed and memory efficiency are highly beneficial in bioinformatics, handling massive genomic datasets. Tasks like DNA sequence analysis, variant calling, and phylogenetic studies are computationally demanding, benefiting significantly from Rust's performance. [Perkel 2020] mentions *Varlociraptor* as a Rust-based bioinformatics tool, highlighting the need for speed and efficient memory usage in such applications.

High-Performance Computing in Scientific Simulations: Rust's performance and fine-grained control over memory management make it well-suited for HPC in scientific domains. Simulating complex physical phenomena, such as climate modeling or fluid dynamics, requires maximizing computational resources. Rust's ability to minimize overhead and deliver predictable performance aligns perfectly with these requirements.

Machine Learning Infrastructure Development: Rust is increasingly employed to build the underlying infrastructure and tools that support machine learning workflows. This includes data pre-processing, feature engineering, and model serving. Rust's reliability and performance enhance the efficiency and robustness of these systems, especially in production environments [Lyu and Rzeznik 2023]. Explores Rust's applications in machine learning, although specific project details are often confidential due to competitive reasons.

Web Services and Cloud Infrastructure: Rust's efficiency and low-level control make it well-suited for building high-performance web servers and cloud infrastructure. Companies like Cloudflare, Discord, and AWS utilize Rust for various components of their infrastructure, benefiting from improved performance, reduced resource consumption, and enhanced security [Logunova 2024].

Real-time Data Processing and Analysis: Rust's predictable performance and lack of garbage collection make it suitable for real-time data processing. Analyzing sensor data, processing financial transactions, or monitoring network traffic demands consistent and timely responses. Rust's ability to deliver predictable performance is crucial in these scenarios [Mittal 2023].

Embedded Systems and Edge Computing: Rust's minimal runtime and efficient resource utilization make it a strong choice for data analysis on embedded systems and edge devices. Analyzing data closer to the source, without the overhead of a full operating system or garbage collection, is advantageous in resource-constrained environments [Mittal 2023].

Game Development: A particularly exciting area of Rust's application is in game development. The gaming industry has traditionally pushed the boundaries of performance and real-time processing, and Rust is emerging as a powerful tool in this domain. Game developers appreciate that Rust provides the low-level control necessary for fine-tuning performance while also offering modern language features that reduce the likelihood of memory leaks, data races, or other bugs that can plague game engines written in more traditional languages. Several innovative game engines written in Rust, such as Bevy and Amethyst, are already making waves in the developer community. These engines take advantage of Rust's "fearless concurrency" to handle complex game logic and render tasks across multiple threads efficiently [Wolverson 2021]. This concurrency model is particularly valuable in the context of modern games, which require the simultaneous processing of physics, AI, audio, and graphics without compromising performance. Moreover, Rust's ability to interface with established game engines via Foreign Function Interfaces (FFI) means that teams can integrate Rust modules into larger, pre-existing projects [Logunova 2024]. This allows developers to optimize performance-critical sections of code—such as rendering pipelines or physics simulations—while maintaining the overall structure of their game development workflow.

6.1 Volvo's Exploration of Rust in Automotive Software

Volvo's adoption of Rust highlights the language's growing relevance in safety-critical and performance-sensitive industries like automotive. Rust's strict memory safety and ownership model directly addresses common pain points encountered in embedded C/C++ development, such as memory corruption bugs and race conditions [Foufas 2022]. These safeguards are crucial in automotive software, where even minor errors can have significant implications for safety and reliability.

A key factor driving Volvo's adoption is Rust's "quality up front" philosophy, which aligns perfectly with the stringent requirements of automotive software. As noted by Volvo engineers, Rust's compile-time checks for lifetimes and mutability ensure that code, once compiled, tends to "just work," reducing the risk of runtime errors. Volvo's initial deployment of Rust targeted the low-power node of the vehicle's core computer, a relatively less complex but critical component, demonstrating a practical starting point for integrating Rust into existing systems [Editors 2024].

Volvo's approach emphasizes coexistence with legacy C/C++ code rather than complete replacement. This incremental strategy allows for a manageable transition, focusing on incorporating Rust in areas where its benefits are most pronounced, such as security-critical modules or newly developed subsystems. Although initial challenges included gaps in hardware abstractions (e.g., missing CAN bus drivers), Volvo actively collaborated with the open-source community to address these limitations and enhance Rust's embedded ecosystem [Foufas 2022]. Volvo anticipates further compiler advancements to support a wider range of hardware targets, potentially expanding Rust's role in safety-critical automotive applications. Volvo's experience serves as a compelling example of how Rust can be effectively integrated into large-scale embedded systems, balancing performance, safety, and a pragmatic approach to adoption.

7 COMPARISION WITH OTHER PROGRAMMING LANGUAGES

7.1 Rust vs. Python

Rust and Python represent fundamentally different approaches to data science, each with its own strengths and weaknesses. While Python prioritizes ease of use and rapid prototyping with its dynamic typing and extensive libraries, Rust emphasizes performance, memory safety, and control. Let's compare Rust and Python based on six key criteria: *Execution Model (Compiled vs. Interpreted)*, *Performance*, *Memory Management*, *Concurrency*, *Ecosystem*, and *Learning Curve*. Table 1 provides a concise overview using a simplified "-/+/++" rating scheme.

Compiled vs. Interpreted Languages. A *compiled language* (e.g. Rust, C, C++) translates source code into machine code before execution. This often yields faster runtime performance because the CPU directly executes the optimized binary. Moreover, compile-time checks (such as Rust's ownership checks) can catch errors

Criteria	Rust	Python
Execution Model	Compiled (++)	Interpreted (+)
Performance	++	+
Memory Management	++	-
Concurrency	++	-
Ecosystem	+	++
Learning Curve	-	++

Table 1: High-level Comparison of Rust and Python

early in the development process, resulting in more predictable performance and improved safety guarantees.

On the other hand, an *interpreted language* (e.g. Python, R) processes source code at runtime. An interpreter reads and executes the code line-by-line, which typically allows for quicker prototyping, dynamic typing, and easier experimentation. However, interpreted languages may incur additional overhead during execution, leading to slower performance in computationally intensive scenarios.

Performance. Rust leverages its status as a compiled language to optimize code at compile time, leading to substantial performance gains in *computationally intensive tasks* [Perkel 2020; **?**]. Python, being interpreted, is known for its ease of use and rapid development but often trails in raw speed compared to optimized Rust binaries.

Memory Management. Rust's *ownership and borrowing system* eliminates the need for garbage collection [Qin et al. 2020], ensuring predictable memory usage and reducing runtime overhead. Python's automatic garbage collection greatly simplifies development but introduces periodic collection cycles, potentially causing variable performance and memory usage.

Concurrency. Rust's strict ownership rules detect and prevent data races at compile time [Saligrama et al. [n. d.]], facilitating safe and efficient concurrency. Python, while capable of concurrency, relies on mechanisms like multi-processing or asynchronous I/O to circumvent the Global Interpreter Lock (GIL), which can restrict parallel performance in CPU-bound tasks.

Ecosystem. Python boasts a mature, expansive ecosystem for data science (`NumPy`, `Pandas`, `Scikit-learn`, etc.), making it a top choice for rapid prototyping and diverse analytics [Xu 2024]. Rust's data science ecosystem, though growing, is comparatively smaller; developers often either implement custom solutions or utilize emerging community crates.

Learning Curve. Python's highly readable syntax and dynamic typing yield a gentle learning curve, especially for scripting and exploratory data analysis. In contrast, Rust demands a deeper initial investment due to its *ownership* model [Qin et al. 2020], but rewards developers with enhanced safety and performance once the concepts are mastered.

Summary. Rust's advantages in safety, speed, and memory management make it particularly attractive for *computationally heavy* or *performance-critical* applications. Meanwhile, Python excels where rapid prototyping and rich library support are top priorities. Ultimately, the choice between Rust (compiled) and Python (interpreted) depends on balancing development speed, library ecosystem, and the performance or safety constraints of a given project.

7.2 Rust vs. R

R holds a dominant position in the world of statistical computing and data visualization, favored for its extensive libraries and tools tailored for statistical modeling and exploratory data analysis. However, R's performance can become a bottleneck, especially when dealing with large datasets or computationally demanding operations. Rust, designed with high-performance computing in mind, excels in these scenarios, offering a compelling alternative. Here is a concise overview using a "-/+/++" rating scheme of the previous key criteria.

Criteria	Rust	R
Execution Model	++	+
Performance	++	+
Memory Management	++	-
Concurrency	++	-
Ecosystem	+	++
Learning Curve	-	+

Table 2: High-level Comparison of Rust and R

Execution Model (Compiled vs. Interpreted). Rust is a *compiled language*. R, an *interpreted language*, processes code at runtime, facilitating rapid experimentation but potentially incurring additional overhead.

Performance. Rust's compiled nature and strong type system enable efficient execution [**?**]. R can handle many analyses with ease, but complex or large-scale computations may run slower in pure R, prompting the use of underlying C/C++ extensions for performance-critical sections.

Memory Management. Rust's ownership and borrowing system ensures *predictable* and *deterministic* memory usage [**?**]. R simplifies memory handling for users via garbage collection, but that same GC can introduce unpredictable pauses and overhead, earning it a lower rating here.

Concurrency. Rust's compile-time checks detect data races early, making safe concurrency is both possible and relatively straightforward [Saligrama et al. [n. d.]]. In R, parallel or concurrent computing is available through specialized packages (e.g. `parallel`, `future`), but the language itself was not originally designed with robust concurrency in mind.

Ecosystem. R has a *massive* ecosystem of libraries (CRAN, Bioconductor, etc.) for statistical computing, machine learning, and visualization, which is ideal for data exploration and domain-specific analyses. While Rust's ecosystem is growing rapidly, it still lags behind R in terms of sheer breadth and specialized libraries.

Learning Curve. R's syntax can be quirky, but for statisticians and data analysts, it is generally considered accessible. Rust demands a steeper learning curve making it more challenging, especially for those accustomed to garbage-collected languages.

Summary. R is an excellent choice for statisticians and data scientists who value its rich library support, specialized statistical functionality, and powerful visualization tools. As Rust's data science ecosystem continues to mature, it becomes an increasingly compelling option for large-scale or high-performance data processing where R's runtime overhead may be limiting.

7.3 Rust vs. C/C++

C and C++ have long served as workhorses for high-performance computing, frequently employed in data science when low-level control and optimizations are essential. However, Rust offers several compelling advantages over these established languages.

Criteria	Rust	C/C++
Execution Model	++	++
Performance	++	++
Memory Management	++	-
Concurrency	++	+
Ecosystem	+	++
Learning Curve	-	+

Table 3: High-level Comparison of Rust and C/C++

Execution Model (All Compiled). Both Rust and C/C++ are compiled. This compilation step enables powerful optimizations and fast runtimes.

Performance. All languages (C, C++, and Rust) are known for *high performance*. In many computationally intensive scenarios, Rust can match or even surpass C/C++ due to advanced compiler optimizations [**?**]. C/C++ remains a strong contender for low-level optimizations in performance-critical applications.

Memory Management. Rust's ownership and borrowing system enforces memory safety at compile time, drastically reducing vulnerabilities like *dangling pointers* and *buffer overflows* [**?**]. In C/C++, developers manually manage memory, which provides maximum flexibility at the cost of increased risk for memory leaks and undefined behavior.

Concurrency. Rust's type system ensures data-race freedom by design, making concurrent programming less error-prone [Saligrama et al. [n. d.]]. C/C++ offer fine-grained control via threading libraries and atomics, but concurrency issues such as data races and deadlocks are easier to introduce if not carefully managed.

Ecosystem. C/C++ has extensive, mature ecosystems for scientific computing, systems programming, and application development—decades of libraries are available across all major platforms. Rust's ecosystem is newer and thus less extensive.

Learning Curve. Rust introduces new concepts that can initially feel restrictive but significantly reduce common bugs. C++ also has a steep learning curve (particularly modern C++ features). Ultimately, the relative difficulty depends on individual experience and willingness to adopt Rust's safety-first approach.

Summary. Rust and C/C++ are all well-suited to performance-critical and low-level systems tasks. C/C++ remains indispensable for legacy systems and offer a vast ecosystem of existing code. Rust's compile-time safety, automatic memory management via ownership, and robust concurrency model make it appealing for new projects where reliability and maintainability are key. As Rust continues to prove itself, it stands as a modern alternative when developers seek both C-like performance and stronger safety guarantees.

8 CONCLUSION

As Rust continues to gain recognition in the field of data science, its widespread adoption will depend on significant advancements in its ecosystem, usability, and industry recognition. While Rust offers substantial performance benefits, its adoption remains constrained by a relatively small library ecosystem, a steeper learning curve, and limited academic and industry integration compared to established languages like Python and R.

8.1 Expanding Rust's Data Science Ecosystem

A key challenge for Rust in data science is the **limited availability of specialized libraries**. While foundational libraries such as `Polars` for data processing and `ndarray` for numerical computing provide essential functionality, Rust still lacks mature frameworks for deep learning, advanced statistical analysis, and large-scale data processing.

To close this gap, future development should focus on expanding Rust's machine-learning ecosystem by creating libraries. That can rival established frameworks like TensorFlow and PyTorch. Additionally, improving interoperability with Python and R would enable seamless integration with existing data science tools, making Rust more accessible for practitioners. Another important aspect is the development of domain-specific libraries tailored for fields such as finance, bioinformatics, and real-time analytics. Strengthening these areas will be crucial for Rust's broader adoption in data science.

If Rust's ecosystem continues to evolve at its current trajectory, it has the potential to become a viable alternative for performance-intensive applications in data science.

8.2 Enhancing Accessibility and Usability

Despite its advantages, Rust's **learning curve remains a barrier** for many data scientists, particularly those accustomed to high-level programming languages. Unlike Python, which prioritizes ease of use, Rust's memory safety model and strict type system require a more extensive understanding of low-level programming concepts. To facilitate wider adoption, future developments should focus on:

- Designing **higher-level APIs** to abstract Rust's complexity while maintaining its performance benefits.

- Improving **data visualization tools**, offering more intuitive alternatives to Matplotlib and ggplot2.
- Enhancing **Jupyter Notebook support**, enabling a more interactive and user-friendly experience.

By improving usability and lowering the entry barrier, Rust can attract a broader audience, including data scientists with limited experience in systems programming.

8.3 Optimizing Performance for Large-Scale Computing

Rust is already recognized for its **high-performance computing capabilities**, but further enhancements could make it even more efficient for data-intensive applications. Future research should explore:

- Implementing **Just-In-Time (JIT) compilation** techniques to optimize execution speed.
- Expanding **GPU acceleration** and CUDA integration to support Rust-based deep learning models.
- Refining **multi-threading and parallel computing strategies** for large-scale distributed computing.

Advancements in these areas could position Rust as a leading choice for computationally demanding data science applications.

8.4 Promoting Adoption in Academia and Industry

For Rust to gain mainstream acceptance in data science, it must achieve broader adoption in both academic and industrial settings. While Rust has gained traction in fields such as systems programming and blockchain, its presence in data science education and research remains limited. To accelerate adoption, initiatives should include:

- Encouraging **universities to incorporate Rust** into data science curricula alongside Python and R.
- Publishing **case studies and benchmark comparisons** to highlight Rust's advantages in real-world applications.
- Facilitating **collaborations between Rust developers and data scientists**, fostering the development of community-driven tools and best practices.

By increasing exposure to academic research and industry projects, Rust could strengthen its position as a viable alternative in data science.

8.5 Addressing Ethical and Security Considerations

As Rust expands into machine learning and artificial intelligence, ensuring **ethical standards and security** becomes increasingly important. Rust's memory safety and concurrency features provide a solid foundation for reliable AI systems, but additional measures are needed to enhance transparency, fairness, and robustness.

Developing **explainability frameworks** for Rust-based AI models will improve interpretability, while methodologies to **detect and mitigate bias** can help ensure fairness. Additionally, strengthening **security frameworks** will safeguard Rust-based AI systems from vulnerabilities, making them more reliable for applications in finance, healthcare, and autonomous systems.

By integrating these considerations, Rust can position itself as a secure and responsible choice for AI-driven data science.

8.6 Conclusion and Future Directions

Rust holds significant promise for data science, particularly in performance-critical applications such as **financial modeling, scientific computing, and real-time analytics**. However, to transition from a niche alternative to a widely adopted standard, Rust must continue to improve its **ecosystem, usability, and industry adoption**.

The next few years will be crucial to determine Rust's role in data science. If the language continues to evolve through enhanced libraries, accessibility, and stronger community engagement, it has the potential to become a leading choice for performance-driven data science applications. Competing directly with established languages such as Python and R.

ACKNOWLEDGMENTS

The authors thank the open-source Rust community for providing valuable libraries, documentation, papers, and examples that inspired this preliminary work. We also appreciate the feedback from colleagues and peers who reviewed early drafts of this paper.

REFERENCES

William Bugden and Ayman Alahmar. 2022. Rust: The Programming Language for Safety and Performance. *Cornell University* (01 2022). https://doi.org/10.48550/arxiv.2206.05503

Rustaceans Editors. 2024. *Volvo Bets Big on Rust.* Retrieved February 7, 2025 from https://medium.com/rustaceans/rust-bytes-2d81355617c3

Johannes Foufas. 2022. *Why Rust is actually good for your car.* Retrieved February 7, 2025 from https://medium.com/volvo-cars-engineering/why-volvo-thinks-you-should-have-rust-in-your-car-4320bd639e09

Inna Logunova. 2024. *Top 12 Real-World Business Use Cases for Rust.* Retrieved February 7, 2025 from https://serokell.io/blog/best-rust-in-use-cases

Shing Lyu and Andrew Rzeznik. 2023. *Practical Rust Projects: Build Serverless, AI, Machine Learning, Embedded, Game, and Web Applications.* Apress.

Chetan Mittal. 2023. *10-best-use-cases-of-rust-programming-language-in-2023.* Retrieved February 07, 2025 from https://medium.com/@chetanmittaldev/10-best-use-cases-of-rust-programming-language-in-2023-def4e2081e44

Jeffrey M. Perkel. 2020. Why scientists are turning to Rust. *Nature Portfolio* 588, 7836 (12 2020), 185–186. https://doi.org/10.1038/d41586-020-03382-2

Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. (06 2020). https://doi.org/10.1145/3385412.3386036

Eric Reed. 2015. Patina: A Formalization of the Rust Programming Language.

Aditya Saligrama, Andrew Shen, and Jon Gjengset. [n. d.]. A Practical Analysis of Rust's Concurrency Story.

Herbert Wolverson. 2021. *Game Development with Rust.* Retrieved February 7, 2025 from https://medium.com/pragmatic-programmers/game-development-with-rust-31147f7b6096

Bo Xu. 2024. Towards Understanding Rust in the Era of AI for Science at an Ecosystem Scale. (05 2024), 1450–1455. https://doi.org/10.1109/cisce62493.2024.10653388

Received 23 February 2025