**Should You Use Rust in the Backend?**

[[](https://medium.com/@patrickkoss?source=post_page-----e5590563a842--------------------------------)](https://medium.com/@patrickkoss?source=post_page-----e5590563a842--------------------------------)

[Patrick Koss](https://medium.com/@patrickkoss?source=post_page-----e5590563a842--------------------------------)

·

Follow

10 min read

·

May 1

88



As a backend developer, I frequently utilize Golang to develop RESTful or gRPC microservices that interact with external services such as databases, object storage, or message queues. However, I recently discovered Rust and was captivated by its robust features, which led me to ponder: Is it worthwhile to adopt Rust for backend development? Notable advantages include performance, thread safety, and error handling, while the primary drawbacks are a steep learning curve and extended compile times.

The suitability of Rust depends on the current stage of your product and the values you prioritize. If rapid iteration cycles are your primary concern, then JavaScript/TypeScript or Python may be your best bet. Python, in particular, is renowned for its capacity for swift prototyping. However, these advantages come at a cost, as higher resource consumption and the absence of static type checking can have significant consequences.

If you prefer a combination of low resource usage, simplicity, and static type checking, Golang might be the ideal choice for your backend development needs.

On the other hand, if you highly value performance and correctness, Rust stands out as an excellent option. With its borrow checker and error handling mechanisms, Rust enables you to create applications that are highly unlikely to crash unexpectedly.

In this blog post, I will delve deeper into the benefits of utilizing Rust for backend development and further explore its potential advantages.

**The Significance of Performance**

Is performance truly significant? This inquiry arises frequently. I believe it’s shortsighted to presume that it isn’t important. Here are a few reasons why performance does matter:

a. Improved User Experience: Better performance leads to faster response times, which in turn improves the overall user experience. A responsive application is more likely to retain users and increase user satisfaction.

b. Scalability: As the number of users grows, it’s essential for the backend to handle increased load efficiently. Rust’s performance characteristics make it easier to scale applications with minimal resource usage, allowing your application to handle more users without compromising performance.

c. Energy Efficiency: High-performance applications consume less energy, which not only reduces operational costs but also helps minimize the environmental impact of running software. Rust’s ability to optimize CPU and memory usage contributes to a greener computing environment.

d. Competitive Advantage: In today’s fast-paced world, an application that performs better than its competitors can gain a significant edge in the market. By using Rust in the backend, you can ensure that your application remains competitive and stands out in terms of performance.

e. Lower Latency: Latency refers to the time it takes for a client’s request to be sent to a server and for a response to be received. Ensuring low latency is crucial, as users anticipate seamless, real-time interactions with technology devoid of any delays. Problems arising from high latency or prolonged delays can lead users to discontinue their engagement on a platform and permanently shift to an alternative application.

f. Easier Maintenance: Efficient code often results in cleaner and more maintainable codebases. Rust’s emphasis on performance and safety can lead to a more organized and easier-to-maintain codebase, making it simpler to update, refactor, and troubleshoot your application as it evolves.

g. Cost Reduction: Operating hardware incurs tangible expenses. By utilizing resources more efficiently and reducing consumption, cost savings can be achieved. This enables us to reserve smaller virtual machines or execute functions-as-a-service, like Lambda, more swiftly. Ultimately, the duration of FaaS operations impacts the final bill — faster execution translates to cost savings.

**Error Handling**

Error handling is a fundamental aspect of developing reliable and secure applications. The ability to catch and address all errors paves the way for a robust application that never succumbs to unexpected crashes. Efficient error handling not only improves the overall user experience, but also aids in identifying, diagnosing, and resolving issues during the development process. By incorporating comprehensive error handling techniques, developers can create resilient applications that can withstand various challenges and maintain their functionality under diverse conditions. Rust’s robust error handling system offers several advantages over other languages:

a. Explicit Error Handling: Rust’s Result type makes error handling explicit and straightforward, forcing developers to address errors directly. This reduces the chance of overlooking potential issues and leads to more reliable and fault-tolerant applications.  
Take, for instance, the following JavaScript example:

JSON.parse(requestData)

In the event that the requestData is invalid, can you determine the outcome? An exception will be thrown, which may not have been caught. Now, let’s examine the Rust equivalent:

let res: Data = serde\_json::from\_slice(&requestData)?;

In this case, it’s evident that the deserialization process can result in an error due to the “?” operator. Consequently, you are obligated to handle the result, making it impossible to accidentally overlook errors.

b. Type Safety: Rust’s strong type system helps catch many errors at compile time, reducing the number of runtime errors. This contributes to improved software stability and reduces the likelihood of crashes and unexpected behavior.

c. Error Chaining: Rust makes it easy to chain errors, allowing developers to propagate errors up the call stack without losing context. This simplifies error management and makes it easier to diagnose and fix issues when they occur.

fn read\_file\_contents<P: AsRef<Path>>(path: P) -> Result<String, io::Error> {  
 let mut file = File::open(path)?;  
 let mut contents = String::new();  
 file.read\_to\_string(&mut contents)?;  
 Ok(contents)  
}

d. Custom Error Types: Rust enables developers to create custom error types, making it possible to represent application-specific errors more accurately. This leads to better error reporting and improved understanding of the root causes of issues.

e. Pattern Matching: Rust’s pattern matching capabilities make it easy to handle different types of errors in a concise and readable manner. This allows developers to focus on the logic of their application rather than getting bogged down by error handling boilerplate.

f. No Exceptions: Rust does not use exceptions, which means there’s no need to worry about handling exceptions or dealing with unexpected side effects. This results in cleaner code and more predictable control flow.

**Thread Safety**

Thread safety is a crucial consideration in backend server development, as these servers typically employ multi-threading, which can introduce potential concurrency issues. Additionally, optimizing runtime performance often involves leveraging multiple threads to execute tasks efficiently. However, developers must remain vigilant to prevent concurrency anomalies that can occur in such scenarios. Rust offers a significant advantage in ensuring thread safety, thanks to its ownership model. By adhering to this principle, developers can create highly concurrent applications with minimized risks of data races or other concurrency-related problems, leading to more reliable and performant systems. Here are additional arguments that showcase the importance of Rust’s approach to concurrency:

a. Fearless Concurrency: Rust’s ownership system and borrow checker ensure that data access is controlled and synchronized, allowing developers to write concurrent code without worrying about data races or other concurrency-related bugs. This leads to more efficient and safer multi-threaded applications.  
Consider the following Golang example:

func main() {  
 var data int  
 var wg sync.WaitGroup  
  
 wg.Add(2)  
  
 go func() {  
 defer wg.Done()  
 data += 1  
 }()  
  
 go func() {  
 defer wg.Done()  
 data += 1  
 }()  
  
 wg.Wait()  
}

Is anything amiss here? In this straightforward example, the problem is relatively simple to identify. We are attempting to modify the same memory location, potentially leading to unpredictable behavior.  
Now, examine the Rust example:

fn main() {  
 let mut data = 0;  
  
 let handle1 = thread::spawn(move || {  
 data += 1;  
 });  
  
 let handle2 = thread::spawn(move || {  
 data += 1;  
 });  
  
 handle1.join().unwrap();  
 handle2.join().unwrap();  
}

Despite the ease of identifying the concurrency issue in this example, more complex cases may exist. The advantage of Rust is that it produces a compile-time error:

error[E0382]: use of moved value: `data`  
 --> src/main.rs:8:9  
 |  
4 | let mut data = 0;  
 | -------- move occurs because `data` has type `i32`, which does not implement the `Copy` trait  
5 |   
6 | let handle1 = thread::spawn(move || {  
7 | data += 1;  
 | ---- value moved here  
8 | });  
 | - value used here after move

b. Efficient Parallelism: Rust enables developers to leverage parallelism, allowing applications to take full advantage of modern multi-core processors. This can lead to substantial performance improvements, especially in compute-bound tasks or applications with high levels of concurrency.

c. Easier Debugging: Since Rust guarantees thread safety at compile time, there’s no need to spend time debugging tricky concurrency-related issues, such as deadlocks or race conditions. This allows developers to focus on the core logic of their application, resulting in a more efficient development process.

d. Crossbeam and Rayon Libraries: Rust offers powerful concurrency libraries like Crossbeam and Rayon, which provide high-level abstractions and parallel data structures for writing safe, concurrent code. These libraries make it even easier to harness the full potential of multi-core processors in Rust applications.

e. Improved Application Stability: Applications with robust thread safety are less likely to crash or exhibit unexpected behavior due to concurrency-related bugs. Rust’s focus on thread safety contributes to the overall stability and reliability of your applications, providing a better experience for end-users.

f. Future-Proofing: As hardware continues to evolve and multi-core processors become even more common, the importance of efficient concurrency and parallelism will only grow. Rust’s focus on thread safety positions it well for the future, ensuring that your applications can take advantage of emerging hardware trends.

**Fast Startup Times**

Quick startup times are essential in various aspects of backend development, as they facilitate efficient horizontal scaling and compatibility with FaaS platforms, among other benefits. Rapid startup times contribute to the overall performance of an application by minimizing latency when responding to incoming requests. This results in a seamless user experience, ensuring that customers remain satisfied and engaged with the application. Moreover, applications with swift startup times can better handle dynamic workloads or rapidly scale to accommodate fluctuating user demands. This adaptability is crucial for maintaining application stability and reliability under a wide range of conditions. Rust’s fast startup times offer several benefits in backend development, extending beyond horizontal scaling and suitability for FaaS platforms:

a. Improved Responsiveness: Applications with fast startup times are more responsive, as they can quickly begin processing requests without lengthy initialization processes. This contributes to a better user experience, especially during peak traffic periods when new instances need to be quickly spun up.

b. Efficient Resource Management: Fast startup times enable more efficient resource management, as applications can quickly release resources when they are no longer needed and reclaim them when demand increases. This allows for better utilization of available resources, resulting in cost savings and reduced environmental impact.

c. Enhanced Deployment and Rollout: Rapid startup times facilitate smoother deployment and rollout of updates or new features, reducing downtime and minimizing disruption to users. In environments where continuous deployment is essential, such as microservices architectures, Rust’s fast startup times can be a significant advantage.

d. Better Recovery from Failures: In the event of a crash or other failure, applications with fast startup times can recover more quickly, minimizing the impact on users and reducing the risk of cascading failures in dependent systems.

e. Flexible Autoscaling: Rust’s fast startup times enable more dynamic and efficient autoscaling strategies, allowing applications to scale in response to changing demand with minimal latency. This can result in better performance during periods of high traffic and cost savings during periods of low demand.

f. Edge Computing: In edge computing scenarios, where resources are limited and latency is a critical factor, fast startup times can make a significant difference. Rust’s fast startup times enable developers to build responsive, efficient edge applications that can quickly process and respond to local data.

**Additional Rust Advantages**

Rust boasts several unique features that set it apart from other programming languages, contributing to its growing popularity among developers. One such feature is its distinct approach to handling values, references, and mutable references. This ensures predictability in control flow and eliminates surprises regarding whether passed variables are mutated or not, resulting in more maintainable and understandable code.  
For example, let’s examine the following Golang code:

func calcSomething(ls \*largeStruct) int32

In this case, we pass a large struct by reference rather than by value to optimize performance. However, the function name suggests that “ls” should not be mutated, even though it is technically allowed.  
In contrast, Rust provides a way to pass a read-only reference, ensuring that “ls” remains unaltered:

fn calc\_something(ls: &LargeStruct) i32

Rust also excels in its implementation of traits and generics, which allow for code reusability and abstraction. Traits define a set of behaviors that can be shared among different data types, while generics enable the creation of functions and data structures that can work with various types without the need for code duplication. The combination of traits and generics promotes clean, modular, and extensible code, making it easier to maintain and evolve over time.

Furthermore, Rust’s compiler messages are remarkably descriptive and helpful, surpassing those found in many other languages. These messages aid developers in pinpointing and resolving issues during the development process, ultimately enhancing overall productivity.

**Conclusion**

Throughout this blog post, we have explored various aspects of Rust that make it a compelling choice for backend development. We delved into the significance of performance, emphasizing that it plays a critical role in customer satisfaction, latency reduction, and cost savings. We also discussed Rust’s approach to error handling, which enforces explicit and robust error management, leading to more reliable applications.

In addition, we examined Rust’s thread safety, a key feature that ensures safe concurrency and prevents data races, resulting in efficient and secure multi-threaded applications. We highlighted the benefits of fast startup times, which are crucial for horizontal scaling and the optimal functioning of FaaS platforms. Lastly, we touched upon several unique Rust advantages, such as its approach to handling values, references, and mutable references, as well as its implementation of traits and generics for code reusability and abstraction.

In conclusion, Rust is a highly viable option for backend development when performance, error handling, thread safety, fast startup times, and other unique advantages are highly valued. By choosing Rust, developers can create efficient, secure, and maintainable applications that meet the ever-evolving demands of modern technology.