**Reliable software engineering with Rust**

[[Egor Dezhic](https://medium.com/@edezhic?source=post_page-----5bb4553b5d54--------------------------------)](https://medium.com/@edezhic?source=post_page-----5bb4553b5d54--------------------------------)

[[Globant](https://medium.com/globant?source=post_page-----5bb4553b5d54--------------------------------)](https://medium.com/globant?source=post_page-----5bb4553b5d54--------------------------------)

[Egor Dezhic](https://medium.com/@edezhic?source=post_page-----5bb4553b5d54--------------------------------)

·

Follow

Published in

[Globant](https://medium.com/globant?source=post_page-----5bb4553b5d54--------------------------------)

·

17 min read

·

Jun 8

381

2



A sample of Rust code by [M. Zakiyuddin Munziri](https://unsplash.com/fr/@zakiego?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

We all want to build reliable software that makes users happy and keeps our heads free from late-night production crashes on Fridays. But every engineer already knows some stack that can handle the users’ needs while tests can check the correctness, so why would anyone bother learning Rust?

You might have already heard that it’s a complicated language requiring considerable onboarding time. For me and many other devs, it wasn’t love at first sight. Memory safety didn’t sound like something I missed in my projects and Rust looks kinda boring in many other aspects.

Why is there so much hype around it, then? I hope this post will clarify why so many devs from all backgrounds and tech companies are getting addicted to it.

**My background**

I’ve been programming for about 12 years, starting with casual HTML and simple algorithmic problems in Pascal, but since then worked with JS, PHP, Python, and a bit of C#, C++, Go, and others. For the past two years, I’ve been learning and practicing Rust, and now I believe it stands out above all the other general-purpose programming languages I’ve encountered.

I don’t think that it is a panacea, though. For example, when I need to create a casual blog, I just pick up [WordPress](https://wordpress.com/) and be done in 15 mins; when I need some heavily interactive web UI, I just take [React](https://react.dev/) with whatever libs I might find for the task at hand; when I need to experiment with machine learning models, I’m definitely using [PyTorch](https://pytorch.org/). But, when I need to build something non-trivial, where performance and reliability are of concern, or the whole architecture is quite complex, **Rust is my go-to choice**.

**Rust’s background**

Many languages like C++ allow you to squeeze every bit and cycle from the available hardware, but this freedom also leaves a lot of room for potential bugs and security holes. Other languages like Java do the opposite and focus on runtime checks and memory management to provide reliability, but they sacrifice performance and consume significantly more resources.

Some projects like [Zig](https://ziglang.org/), [GraalVM](https://www.graalvm.org/), [Mojo](https://www.modular.com/mojo), and [AssemblyScript](https://www.assemblyscript.org/) try to add the missing pieces for the existing languages to compensate to some extent but achieve only partial success because of the abundance of legacy code. Such code was written with different runtime/compiler assumptions in mind and in some cases is incompatible with growing expectations and requirements. **Rust was designed from scratch to provide both performance and correctness at the same time**. And because it was a completely new language, its creators weren’t constrained by legacy support and were able to combine the best features from many languages into something convenient and familiar.

It started as a personal research project at Mozilla around 2006 because internet safety was their major concern. And since the web assumes execution of potentially unsafe code, it is important to minimize memory-error exploits. Rust reached its first stable release in 2015, so it’s pretty young by programming language standards. But, it has already outgrown lots of more mature languages, is the most loved and wanted one according to [Stack Overflow surveys](https://survey.stackoverflow.co/2022/) for many years in a row, and is the fastest growing according to the [PYPL index](https://pypl.github.io/PYPL.html).

It’s important to note here that Rust maintainers weren’t delusional to think that devs will just rewrite whole projects in Rust, so they made sure that interop with C, JS, Python, and other languages is painless. Also, they didn’t try to make some shiny unique, opinionated thing. Still, they accumulated common patterns and best practices from many languages in a way that feels familiar and intuitive to the broadest public. The result is a modern PL with extreme **performance, reliability, and compatibility**.

Interestingly, Rust still hasn’t drawn much attention outside of geeky platforms. It’s rarely mentioned among popular IT resources and has a pretty little share of outsourced/freelance gigs, despite being endorsed by lots and lots of huge names in the programming world. At least for now, it looks more like a long bet, but a pretty solid one in my view, so let’s move to why I believe in it *so much*.

**Under the hood**

The core feature of Rust is the deeply integrated borrow checker — memory usage verification system that makes sure that you are never accessing a variable that has already been recycled, not mutating a value concurrently from different threads without the proper checks, not trying to access a collection member outside of its contents and overall makes many types of memory bugs and vulnerabilities impossible to compile.

It often seems like a hostile party breaker when you’re just getting started, but it’s actually a great friend once you get to know this guy closer. To be precise, it doesn’t completely block you from writing dangerous code, but in most cases, it forces you to be explicit about such things and makes manual inspection of the unsafe pieces much easier. This also allows you to be much more confident in 3rd party dependencies just because they are compiled successfully.

Rust is most often compared to C because it provides the ability to control the hardware on the lowest levels, but I don’t think that such an analogy fairly appreciates all the effort that the compiler makes to hide that complexity by inferring types and lifetimes automatically. It’s still more verbose than most other typed languages and especially duck-typed ones, but this verbosity mostly comes from the error handling, null safety, and type casting **best practices**.

Ideologically it’s neither object-oriented nor functional, but has primitives from both worlds and allows any approach without enforcing one or the other. If anything, I think Rust is a correctness-oriented language, which might be problematic for pet project prototyping, but extremely valuable for mission-critical services. Worth noting that Rust isn’t the first language focused on reliability: there are [Haskell](https://www.haskell.org/), [OCaml](https://ocaml.org/), and others, but they are pretty exotic for most developers and have their own quirks. Also, Rust is very nice for multithreaded and concurrent programming.

Another primary focus of Rust is on *zero-cost abstractions* — types that don’t actually allocate any memory when the code is compiled and running. Mostly they are used as markers for other types which instruct the compiler how these types should be treated. Their purpose is to provide **safe** interfaces over highly optimized but potentially dangerous practices *without adding runtime overhead*.

Official Rust distribution packs the compiler, a standard cross-platform library, straightforward package management, an extensible build system, built-in tooling for tests and benchmarks, comprehensive documentation, and even several books for starters and advanced users. This combination and the developer experience with this tooling easily beat every other language known to me.

Build tools and std are available for a very wide range of target platforms, including lots of exotic ones like [Fuchsia](https://fuchsia.dev/) and [watchOS](https://en.wikipedia.org/wiki/WatchOS), but I think one of them deserves very special attention.

**WebAssembly**

While initially created as a companion for JS to run heavy workloads in the browsers, [WASM](https://webassembly.org/) became a lot more than that. It’s supposed to run on the client side with low-level control of the hardware, so it requires significant security measures and was created with almost air-gapped sandboxing in mind.

Thanks to the minimalistic security-first design WASM stands out among another assembly, bytecode, and binary formats as an extremely safe cross-platform compilation target. While it totally deserves a separate post I wanted to include it here because the **Rust ecosystem has great support for it**.

Due to its history and low-level rambling, it was hard for me to grasp the potential that it has initially. But now it’s hard for me to imagine a context where it can’t be beneficial. For example, the founder of Docker believes that there wouldn’t even have been a need for Docker if WASM and [WASI](https://wasi.dev/) (WebAssembly System Interface) existed at the time. And unlike usual Linux-based Docker images, WASM containers can be a lot smaller, consume significantly less resources, cold start a lot faster, provide even harder sandboxing, and have smaller potential attack surface.

Also, recently [Wasmer](https://wasmer.io/), a major player in the wasm ecosystem, announced [WASIX](https://wasix.org/) — an extension to WASI that adds core POSIX-compatible interfaces to WASM, and beside the spec, they provide build tools with first-class support for Rust and a polyfill library to work in the browsers. It allows you to run even such complex apps as the [CPython](https://en.wikipedia.org/wiki/CPython) interpreter, so the remaining WebAssembly’s limitations are being steadily wiped out.

It can also serve as a lighter-weight replacement for Java bytecode in many cases, with much broader language support and no vendor lock-in. It enables User Defined Functions written in any language for the databases. It allows you to safely run untrusted binaries. The potential here is just mind-blowing to me.

[WASM runtimes](https://github.com/appcypher/awesome-wasm-runtimes) can be even **smaller than 100kb**, and many languages support compilation into WASM to some extent. With this combination of features, you can go as far as cross-platform embedded development in pretty much any language!

**Above the hood**

Quite likely that by this point you might consider Rust as something cool on its own, especially if compiled into WASM, but real-world development doesn’t happen in a vacuum — we need an ecosystem of libraries and frameworks to be able to provide solutions in meaningful time spans. That’s something I didn’t actually expect from Rust when I was getting started because most young languages suffer from this problem and often quietly die out without accumulating enough packages to become actually useful.

However, that’s where safety, reliability, maintainability, and performance played a critical role. This combination of features proved to be *extremely* valuable for the big tech, so they invest heavily in the Rust ecosystem. It’s hard to overestimate how big of a deal this is and the only language with similar impact that comes to mind is Java. But even that isn’t the whole story — enthusiasts with all kinds of experiences are learning Rust as their first language with low-level control because it’s so much easier to get into than C/C++ and others.

All of that together provides an almost unprecedented boost to the Rust community in many directions. While most interesting things are happening with below-application-level software, these things are getting easy to use from your application-level code with reasonable defaults while also providing facilities to tweak *anything* in your stack without leaving the comfort of Rust.

I want to briefly present some usage scenarios and crates(Rust packages) for different specializations to highlight at least a few interesting representatives of the ecosystem.

**Backend**

Lots of performant server frameworks: [Actix Web](https://actix.rs/) for extreme throughput, [Rocket](https://rocket.rs/) for extreme simplicity, [warp](https://github.com/seanmonstar/warp) for the fans of the functional approach, [Salvo](https://salvo.rs/), as well as many others. But my personal favorite is [Axum](https://github.com/tokio-rs/axum) because it has a macro-free API and can be easily detached from the [Tokio](https://tokio.rs/) runtime to run inside the browser’s [Service Worker](https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API), which allows sharing server and client request processing codebase.

Thanks to Rust, you can have both **<1ms cold starts** for serverless functions and **reliable** long-running services depending on your needs. Deployments are simple because you can easily pack the whole thing into a single native or a WASM binary.

**GUI**

For web-based projects, you can compile your Rust code into a WebAssembly module and use [wasm-bindgen](https://rustwasm.github.io/wasm-bindgen/) to generate Rust<->TS/JS bindings. You can also transpile and bundle custom TS/JS with [SWC](https://swc.rs/), compile SASS with [grass](https://github.com/connorskees/grass), add [Tailwind](https://tailwindcss.com/) classes with [railwind](https://github.com/pintariching/railwind), and other things. Many of them can be quickly set up with [Rspack](https://github.com/web-infra-dev/rspack). Surprisingly, despite its reputation as a system-level programming language, Rust has all the required tools for the usual FE development.

Or, if you want a nice all-in-one package you can try [Yew](https://yew.rs/) or [Dioxus](https://dioxuslabs.com/docs/nightly/guide/en/index.html). Conceptually they’re pretty similar to react-based full-stack frameworks like NextJS, so they will feel familiar to lots of devs. Also, you can use [Tauri](https://tauri.app/) to pack your apps into native binaries that connect to the built-in webviews. Together with [PWA](https://pwrs.app/) capabilities, I think this is currently the best way to build apps that work both on the web and natively. Other popular options include [iced](https://iced.rs/) and [egui](https://github.com/emilk/egui) which are native-first but iced can work in the browser too.

**Gaming**

Rust doesn’t have a massive gamedev ecosystem like C++, but it is often considered as a cleaner and simpler language, and it has great support for the most promising and modern GPU programming standard — [WebGPU](https://en.wikipedia.org/wiki/WebGPU). [Bevy](https://bevyengine.org/) is the most popular choice thanks to its clean APIs and because it already runs inside the latest stable Chrome versions.

Unfortunately, I have very little experience with game development so there isn’t much I can suggest, but [arewegameyet.rs](https://arewegameyet.rs/) contains a whole bunch of libs, frameworks, and engines that you can check out.

**Data**

While Rust isn’t particularly popular among data engineers yet, efficient and reliable memory handling is at the core of this discipline, so the future looks quite bright. One project that looks especially promising to me is [sled](https://github.com/spacejam/sled) and its new WIP storage engine — [Komora project](https://github.com/komora-io). Also, [GlueSQL](https://github.com/gluesql/gluesql) might grow into a solid alternative to SQLite someday. However, these are pretty raw and not really ready for production.

On the other hand, there are multiple mature crates with interesting features: [SQLx](https://github.com/launchbadge/sqlx) provides build-time verification of your queries, [SeaORM](https://www.sea-ql.org/SeaORM/) provides an ORM on top of SQLx, [Diesel](https://diesel.rs/) is also a popular ORM but with a bit different approach to correctness checking, and just regular drivers for most popular DBs. For heavier data pipelines you can try [Databend](https://github.com/datafuselabs/databend) — an open-source alternative to [Snowflake](https://www.snowflake.com/) and [ClickHouse](https://clickhouse.com/).

Together with WASM, you can get another cool capability — user-defined functions that are performant, safe, and written in the same language as the rest of your app. For example, libSQL focuses on WASM support as one of its core goals, probably because they realize how efficient, portable, and future-proof this approach is.

**Infrastructure**

For consumers Rust is great because single binaries are easy to configure for deployment, especially with WASM, has wide platform support, a simple build process, and thanks to its efficiency customers end up with minimal bills. If you need more than a couple of services most likely you’re dealing with K8s, so you might find [kube-rs](https://github.com/kube-rs/kube) handy. Also, [kdash](https://github.com/kdash-rs/kdash) might be an interesting simpler alternative to more mature and complex cluster monitoring solutions.

For providers Rust delivers top-tier performance and security, so pretty much every major and sometimes even small cloud is investing in Rust development these days. One of the recent exciting projects in this space is [kuasar](https://kuasar.io/) — efficient container runtime that supports multiple sandboxes like [QEMU](https://www.qemu.org/) and [WasmEdge](https://wasmedge.org/), as well as a number of others, with a roadmap to get all the remaining major ones soon. Another ambitious project here is [nanocl](https://github.com/nxthat/nanocl) - basically a kubernetes replacement in Rust focused on simplicity for developers.

**Operating system**

Just like in the clouds — performance and safety are the core requirements here, so Rust is gaining a lot of attention for Linux and Windows development. With the Linux kernel it’s especially remarkable because no other language except C was ever accepted there, even C++, until the recent Rust adoption. I don’t know whether it’s used in the core of MacOS/iOS, but Apple is definitely using it for multiple projects.

On a more exotic, and in my view even more exciting side there is [Redox](https://www.redox-os.org/) — Unix-like OS written in Rust. It’s not just a toy or a proof-of-concept project, but a [real running one](https://www.redox-os.org/screens/) with over 8 years of development. Of course, it can’t replace Linux or others in most cases yet, but it’s designed to have hardened security, higher performance, and better stability from the start. If this project will get more support I believe with time we can get an actual competitor to Linux.

Also, worth reiterating here that the Rust standard library is [supported on almost all operating systems and architectures](https://doc.rust-lang.org/nightly/rustc/platform-support.html).

**Embedded**

Up until recently, C was basically the only language here, with a couple of projects to bring Python, JS, and some others to relatively powerful devices. But Rust creators [targeted this use case](https://www.rust-lang.org/what/embedded) from the very beginning and it’s a solid alternative now that supports most embedded architectures.

The main advantage is, again, reliability. It’s quite common for such devices to be required to run for months or even years without crashes under heavy memory and performance constraints. And Rust’s compiler helps developers to catch lots of bugs in the build process, while also allowing them to interop with existing C code and even inject custom assembly instructions when needed.

This [repo with curated resources](https://github.com/rust-embedded/awesome-embedded-rust) looks very solid if you’re looking for crates and learning materials, so make sure to check it out if you’re working or willing to get into this field.

**Machine learning**

Nowadays ML development is almost a monopoly in the hands of Nvidia and their CUDA library. While this status quo is pretty solid right now, it’s quite obvious that it won’t last forever and Rust has great support for something that can bring ML to a wide range of hardware — WebGPU. And it’s not just theory — multiple ML runtimes like [ONNX Runtime](https://github.com/microsoft/onnxruntime) and [webgpu-torch](https://github.com/praeclarum/webgpu-torch) are already moving there, so we should expect lower entry barriers and simpler deployments in the near future.

Another Rust application in this space that I find exciting — **code generation**. ML models can already produce significant amounts of meaningful code, but they aren’t reliable and won’t be in the observable future, so we need safeguards against bugs in such code and right now it’s usually just manual inspection. However, the Rust compiler can take some of the verification burden from devs and streamline this process a bit. Also, if you’ll compile it into WASM you can also safely run it without fear that your model accidentally generated malware that might corrupt your machines. After creating tests with acceptance criteria you can safely generate tons of plausible solution candidates, let the Rust compiler check for many problems right away, and then run these tests in the WASM sandboxes to check for correctness, and repeat the process until you get the proper solution.

**Blockchain**

Rust is a great fit here because blockchains themselves and their smart contracts are extremely sensitive to performance and correctness. Suboptimal performance or a vulnerability can easily lead to the loss of lots of $$$. [Solana](https://solana.com/), [Polkadot](https://polkadot.network/), and many others have already adopted Rust for their own development as well as Rust-based contracts.

Also, WASM is widely considered the future for contracts for the already mentioned reasons as well as support for development from multiple languages. Some blockchains can already run WASM smart contracts.

**Test automation**

First of all, the default Rust tooling already has solid support for tests, benchmarks, runnable examples, and even docs with runnable code. The whole language is heavily focused on correctness and tests are no exception.

If the built-in tooling is not enough for your needs there are crates like [thirtyfour](https://github.com/stevepryde/thirtyfour) — spec-compliant selenium driver, [goose](https://github.com/tag1consulting/goose) for load testing, [wiremock](https://github.com/LukeMathWalker/wiremock-rs) for HTTP request mocking, and many others.

**Security**

Enhanced security is probably the main reason why Rust was created in the first place. According to multiple sources and estimations, the cause of up to **80% of vulnerabilities is bad memory handling**! So even by just rewriting existing C/C++ code in Rust, you can patch a lot of holes. If you combine that with WASM’s sandboxing you can get extremely solid security guarantees for relatively small development and performance costs.

Among cryptography-related crates, the most famous is [Rustls](https://github.com/rustls/rustls) — an OpenSSL alternative written in Rust for better security, portability, and devex. There are also plenty of cryptographic primitives made by [RustCrypto](https://github.com/RustCrypto) group.

For network monitoring purposes there is [Sniffnet](https://www.sniffnet.net/). For other security needs, Rust provides low-level control and high portability, but the ecosystem isn’t huge as of yet. It’s also a pretty nice fit for malware development like described in the [Black Hat Rust book](https://github.com/skerkour/black-hat-rust), but I don’t recommend getting into that except for learning purposes.

**Everything everywhere all at once**

The coolest thing — usually real-world IT solutions rely on at least several disciplines mentioned above and with Rust you can have them all in one language to share types, business logic, and even engineers. Such uniformity also enables much easier overall integration than having, let’s say, a React frontend, a Python backend, some data pipelines in Java, and Go code that manages the deployments. This gives Rust a significant advantage over most PLs except C/C++, but I think Rust wins over them in many other aspects and especially developer experience.

**Want to try the rusty hoodie?**

You’ve probably heard that Rust isn’t trivial and might be hard to learn, but it’s not entirely true. It’s hard to become an expert in Rust because that will require a deep understanding of how the whole software stack works starting from individual processor instructions. But, most devs don’t need to go this far, just like most JS and Python devs don’t need to know all the internals of their runtimes.

Usual Rust development cycle has major differences from PLs like JS or PHP. Rust is a compiled language and its compiler is one of the strictest. Unlike development in interpreted languages where you usually write a piece of code and run it to see what happens, with Rust you’ll spend a lot more time trying to build the code until there are no more compilation errors. But don’t worry, **Rust compiler errors are truly amazing and often contain tips on how to fix them**.

This build process might seem like an obstacle at first, but once your program is compiled it already means that all the types are compatible and memory is handled meaningfully. Compilation itself is relatively slow because of these checks, but it’s incremental and these checks save your time from chasing many types of bugs. I’ve never before experienced writing so much code between runs for manual inspection as with Rust. However, it might take hours to get used to it, so don’t worry if your productivity will seem low at the beginning.

Google for crates and check out relevant discussions on Reddit. There aren’t many app boilerplates around, but you can easily compose the stack for your needs based on examples and docs. Interestingly, the ecosystem is extremely wide and I often stumbled upon libraries for use cases where I haven’t initially expected anything from Rust, like for example frontend development. But often you won’t find any information about them except for their GitHub and doc pages.

And the last tip for now — *don’t worry about writing the optimal code right away*. It might be tempting since Rust provides all the tools for that but remember — premature optimization is the root of all evil © [xkcd](https://www.explainxkcd.com/wiki/index.php/1691:_Optimization). Even naive Rust implementation will likely be magnitudes times faster than in many other languages. Just leave some comments around and you’ll be able to easily come back and refactor or optimize it later. Thanks to strict types and memory management it’s relatively easy to mess with even spaghetti multithreaded concurrent code, so you won’t be left with dark places in the codebase that everyone is too afraid to touch.

**Becoming a rustacean**

Rustacean is a common term for software engineers with Rust experience. They are usually born between the pages of [The Rust Book](https://doc.rust-lang.org/book/); their natural habitats are [Docs.rs](https://docs.rs/), Github, Reddit, and Twitter, and more often than not they are pretty friendly geeky creatures. They are using [Crates.io](https://crates.io/) to get nutrition and their favorite snacks can be found inside the examples folders in the crates’ repos. But beware — some of them are so rusty that they become salty if they encounter specimens of other kinds.

First, you’ll need to get some rust on your metals. The most common way to do that is by using [rustup](https://rustup.rs/) — toolchain installer and manager. It will include the compiler, package manager, linter, and other things. Basically a one-stop-shop for Rust tooling. The excellent quality of Rust documentation, compiler error messages, and community support together with an easy setup thanks to rustup lowers the bar for entering this rustacean territory.

Next, you’ll need the development environment. Personally, I recommend [VSCode](https://code.visualstudio.com/), especially if you are coding a web Rust+TypeScript project because it has a built-in TS type checker. Then I’d highly recommend installing the [rust-analyzer](https://rust-analyzer.github.io/) extension to get rich language support, and, by the way, it also integrates well with Vim, Emacs, and others.

Alternatively, you can quickly spin up a [Replit](https://replit.com/@replit/Rust) instance with the same setup and start coding even from your smartphone. Don’t worry — basic repl is free. Just press “Use Template” to fork it and you’re good to go.

If you want your project rebuilt and rerun after every change you should try [cargo-watch](https://crates.io/crates/cargo-watch). Also, since Rust is a great fit for the development of CLI tools there is an abundance of them for all kinds of needs.

Last but not least — just start building anything. Get used to the compiler and ownership rules. Get deeper intuition of smart pointers. Don’t try to understand all the Rust features right away — focus on the stuff that you need for your project, eg async/futures if you’re building something concurrent like web servers, and gradually expand your understanding. And unless you have solid C/C++ experience — don’t expect to get used to Rust in a couple of hours. Give it some time, it’s worth it. Good luck & have fun!

**Some learning resources**

* [The Rust Book](https://doc.rust-lang.org/book/) — official introduction, widely considered the best for newcomers
* [Comprehensive Rust](https://google.github.io/comprehensive-rust/welcome.html) — introduction by Google with a focus on Android development
* [Learn Rust in Y Minutes](https://learnxinyminutes.com/docs/rust/) — shallow yet quick introduction
* [How to Learn Rust](https://learning.accelerant.dev/how-to-learn-rust) — a new course made by a famous rustacean
* [rust-learning](https://github.com/ctjhoa/rust-learning) — a huge collection of all kinds of Rust resources in multiple languages