**Using Rust at a startup: A cautionary tale**

[[Matt Welsh](https://mdwdotla.medium.com/?source=post_page-----42ab823d9454--------------------------------)](https://mdwdotla.medium.com/?source=post_page-----42ab823d9454--------------------------------)

[Matt Welsh](https://mdwdotla.medium.com/?source=post_page-----42ab823d9454--------------------------------)

·

Follow

10 min read

·

Nov 23, 2022

2.8K

49

*Rust is awesome, for certain things. But think twice before picking it up for a startup that needs to move fast.*



All of the art for this post was generated using DALL-E.

I hesitated writing this post, because I don’t want to start, or get into, a holy war over programming languages. (Just to get the flame bait out of the way, Visual Basic is the best language ever!) But I’ve had a number of people ask me about my experience with Rust and whether they should pick up Rust for their projects. So, I’d like to share some of the pros and cons that I see of using Rust in a startup setting, where moving fast and scaling teams is really important.

I want to be clear that I am a fan of Rust *for certain things*. This post isn’t about how Rust is bad as a language or anything of the sort. What I do want to talk about, however, is how using Rust will almost certainly involve a nontrivial productivity hit that could be a major factor if you are trying to move fast. Weigh carefully whether the velocity impact is worth the benefits of the language for your company and product.

Right up front, I should say that **Rust is very good at what it’s designed to do**, and if your project needs the specific benefits of Rust (a systems language with high performance, super strong typing, no need for garbage collection, etc.) then Rust is a great choice. But I think that Rust is often used in situations where it’s not a great fit, and teams pay the price of Rust’s complexity and overhead without getting much benefit.

My primary experience from Rust comes from working with it for a little more than 2 years at a previous startup. This project was a cloud-based SaaS product that is, more-or-less, a conventional CRUD app: it is a set of microservices that provide a REST and gRPC API endpoint in front of a database, as well as some other back-end microservices (themselves implemented in a combination of Rust and Python). Rust was used primarily because a couple of the founders of the company were Rust experts. Over time, we grew the team considerably (increasing the engineering headcount by nearly 10x), and the size and complexity of the codebase grew considerably as well.

As the team and codebase grew, I felt that, over time, we were paying an increasingly heavy tax for continuing to use Rust. Development was sometimes sluggish, launching new features took longer than I would have expected, and the team was feeling a real productivity hit from that early decision to use Rust. Rewriting the code in another language would have, in the long run, made development much more nimble and sped up delivery time, but finding the time for the major rewrite work would have been exceedingly difficult. So we were kind of stuck with Rust unless we decided to bite the bullet and rewrite a large amount of the code.

Rust is supposed to be the best thing since sliced bread, so why was it not working so well for us?



**Rust has a huge learning curve.**

I’ve worked in dozens of languages in my career, and with few exceptions most modern, procedural languages (C++, Go, Python, Java, etc.) all very similar in terms of their basic concepts. Each language has its differences but usually it’s a matter of learning a few key patterns that differ across languages and then one can be productive pretty quickly. With Rust, though, one needs to learn *entirely new ideas* — things like lifetimes, ownership, and the borrow checker. These are not familiar concepts to most people working in other common languages, and there is a pretty steep learning curve, even for experienced programmers.

Some of those “new” ideas are, of course, present in other languages — especially functional ones — but Rust brings them into a “mainstream” language setting, and hence will be new to many Rust newcomers.

Despite being some of the smartest and most experienced developers I had worked with, many people on the team (myself included) struggled to understand the canonical ways to do certain things in Rust, how to grok the often arcane error messages from the compiler, or how to understand how key libraries worked (more on this below). We started having weekly “learn Rust” sessions for the team to help share knowledge and expertise. This was all a significant drain on the team’s productivity and morale as everyone felt the slow rate of development.

As a comparison point of what it looks like to adopt a new language on a software team, one of my teams at Google was one of the first to switch entirely from C++ to Go, and it took no more than about two weeks before the entire 15-odd-person team was quite comfortably coding in Go for the first time. With Rust, even after months of working daily in the language, most people on the team never felt fully competent. A number of devs told me they were often embarrassed that it was taking longer than they expected for their features to land and that they were spending so long trying to wrap their heads around Rust.

**There are other ways to fix the problems that Rust is trying to solve.**

As mentioned above, the service we were building was a fairly straightforward CRUD app. The expected load on this service was going to be on the order no more than a few queries per second, max, through the lifetime of this particular system. The service was a frontend to a fairly elaborate data-processing pipeline that could take many hours to run, so the service itself was not expected to be a performance bottleneck. There was no particular concern that a conventional language like Python would have any trouble delivering good performance. There were no special safety or concurrency needs beyond what any web-facing service needs to deal with. The only reason we were using Rust was because the original authors of the system were Rust experts, not because it was an especially good fit for building this kind of service.

**Rust has made the decision that safety is more important than developer productivity.**This is the right tradeoff to make in many situations — like building code in an OS kernel, or for memory-constrained embedded systems — but I don’t think it’s the right tradeoff in all cases, especially not in startups where velocity is crucial. I am a pragmatist. I would *much* rather have my team sink time into debugging the occasional memory leak or type error for code written in, say, Python or Go, than have everyone on the team suffer a 4x productivity hit for using a language designed to avoid these problems entirely.

As I mentioned above, my team at Google built a service, entirely in Go, that over time grew to supporting more than 800 million users and something like 4x the QPS of Google Search at its peak. I can count on one hand the number of times we hit a problem that was caused by Go’s type system or garbage collector in the years building and running this service. Basically, **the problems that Rust is designed to avoid can be solved in other ways**— by good testing, good linting, good code review, and good monitoring. Of course, not all software projects have this luxury, so I can imagine that Rust may be a good choice in those other situations.



**You will have a hard time hiring Rust developers.**

We hired a ton of people during my time at this company, but only about two or three of the 60+ people that joined the engineering team had previous experience with Rust. This was not for want of trying to find Rust devs — they just aren’t out there. (By the same token we were hesitant to hire people who *only* wanted to code in Rust, since I think that’s a bad expectation to set in a startup setting where language and other technology choices need to be made in an agile way.) This paucity of Rust dev talent will change over time, as Rust becomes more mainstream, but building around Rust on the assumption you’ll be able to hire people who already know it seems risky.

Another secondary factor is that using Rust will almost certainly lead to a schism between the people on the team who know Rust and those who don’t. Because we had chosen an “esoteric” programming language for this service, the other engineers in the company who might have otherwise been helpful in building features, debugging production issues, and so forth were largely unable to help because they couldn’t make heads or tails of the Rust codebase. This lack of fungibility in the engineering team can be a real liability when you’re trying to move fast and harness the combined strengths of everyone on the team. In my experience, people generally have little difficulty moving between languages like C++ and Python, but Rust is new enough, and complex enough, that it presents a barrier to people working together.

**Libraries and documentation are immature.**

This is a problem that (I hope!) will be fixed over time, but compared to, say, Go, Rust’s library and documentation ecosystem are incredibly immature. Now, Go had the benefit that it was developed and supported by an entire dedicated team at Google before it was released to the world, so docs and the libraries were fairly polished. Rust, by comparison, has long felt like a work in progress. The docs for a lot of popular libraries are pretty sparse, and one often needs to read the *source code* of a given library to understand how to use it. This is bad.

Rust apologists on the team would often say things like “async/await are still really new” and “yeah the docs for that library are lacking” but these shortcomings impacted the team pretty significantly. We made a huge mistake early on by adopting Actix as the web framework for our service, a decision that led to tremendous amounts of pain and suffering as we ran into bugs and issues buried deep in the library that nobody could figure out how to fix. (To be fair, this was a few years ago and maybe things have improved by now.)

Of course, this kind of immaturity is not really specific to Rust, but it does amount to a tax that your team has to pay. No matter how great the core language documentation and tutorials are, if you can’t figure out how to use the libraries, it doesn’t much matter (unless you’re planning to write everything from scratch, of course).

**Rust makes roughing out new features very hard.**

I don’t know about anyone else, but at least for me, when I’m building a new feature I usually don’t have all the data types, APIs, and other fine details worked out up front. I’m often just farting out code trying to get some basic idea working and checking whether my assumptions about how things should work are more-or-less correct. Doing this in, say, Python is extremely easy, because you can play fast and loose with things like typing and not worry if certain code paths are broken while you rough out your idea. You can go back later and make it all tidy and fix all the type errors and write all the tests.

In Rust, this kind of “draft coding” is very difficult, because the compiler **can and will complain about every goddamn thing that does not pass type and lifetime checking**— as it is explicitly designed to do. This makes perfect sense when you need to build your final, production-ready implementation, but absolutely sucks when you’re trying to cruft something together to test an idea or get a basic foundation in place. The unimplemented! macro is helpful to a point, but still requires that everything typechecks up and down the stack before you can even compile.

What really bites is when you need to change the type signature of a load-bearing interface and find yourself spending hours changing every place where the type is used only to see if your initial stab at something is feasible. And then redoing all of that work when you realize you need to change it again.



**What is Rust good at?**

There are definitely things I like about Rust, and features from Rust that I’d love to have in other languages. The match syntax is great. The Option, Result, and Error traits are really powerful, and the ? operator is an elegant way of handling errors. Many of these ideas have counterparts in other languages, but Rust’s approach with them is particularly elegant.

I would absolutely use Rust for projects that need a high level of performance and safety and for which I was not terribly worried about the need to rapidly evolve major parts of the code with a whole team that is growing fast. For individual projects, or very small (say, 2–3 person) teams, Rust would likely be just fine. Rust is a great choice for things like kernel modules, firmware, game engines, etc. where performance and safety are paramount, and in situations where it may be hard to do really thorough testing prior to shipping.

Okay, now that I’ve sufficiently pissed off half of the readership of Hacker News, I guess now is as good a time as any to announce the topic of my next article: Why nano is the superior text editor. See you next time!