Written by Piotr Sarnacki

on October 11, 2026

I'm sorry, but you still have to think

DHH, the creator of the Ruby on Rails framework, decided to rewrite Campfire Once from Ruby on Rails to Rust. Or rather ask a clanker to do it for him, as he can't stand reading or writing the Rust code himself. After that he added rewrites in other languages, like Elixir and Go.

This is very interesting to me, as it's a very good insight into the results of someone using AI agents without reading the code. In this case, that someone has also a lot of prior programming experience. And the results are... well, not very encouraging if you are hoping you can stop reading the code, or stop thinking altogether anytime soon. There are multiple problems with the rewrites, but let's start with non-functional differences cause they nicely show something that a lot of people seem to not realize: if your prompt is not specific enough, many decisions are a coin flip. This is because a whole lot of questions don't have a single correct answer. Do we want to keep backwards compatibility? Do we care more about latency or throughput? How much memory can the system use under load? Is loosing new content notifications after a crash acceptable? You may not care about them, or at least some of them, but they will be implicitly answered when an LLM implements what you think you want.

If you look into the rewrites closer, you will quickly see that they treat backwards compatibility and other constraints differently. The Rust version doesn't maintain 100% backwards compatibility, for example it drops the CSRF token so that caching is easier. It also dropped Redis in favour of in-process queues, for example for notifications. Elixir rewrite is much closer to the Rails original. This already makes the comparison pretty much useless, as these differences are language independent. It's not like a clanker heard "rewrite in Elixir" and chose to keep 100% backwards compatibility because of the language that has been used. But it gets even better!

If you looked at the code, and yeah, I know, we should not be reading code anymore, you would quickly notice a lot of the things are not great. For example, I've seen people complaining that Elixir's version got a single process sequentially processing all of the SQL queries, even if these were reads that could be running concurrently. That's a fair complaint, but DHH seems to think it's not the best look for Elixir that the clanker couldn't write performant code. Not sure if I agree when I look at the Rust version. Cause in the Rust version some of the database operations are not async, which in some

cases may be worse. You see, in Rust, when using an async runtime, the scheduling is not preemptive, but rather cooperative. If a task doesn't yield, no other task can run on the same worker thread. What this means in practice, is that the time spent in the task should be as short as possible. For example, when you run an SQL query that takes 100ms, you don't want all the other tasks to wait as the task issuing the query is waiting anyway. Thus, you should ideally use an async I/O operation that yields to the runtime while waiting for a response from the database. In the Rust rewrite some queries run on worker threads, but some run as blocking operations in async tasks. Same with locks. When using an async runtime, the safest bet is to use async locks like `tokio::sync::Mutex`. It's fine to use a non-async version if you're sure that the lock is held for a very short time, but if you hold a sync lock for 10ms, all the tasks on the same thread wait for it, while the blocking task itself is not doing any work. Thus, I regret to inform you, that no, coding is not likely "solved", and you still need to know what you're doing.

Going further, it turned out that slop benchmarks are also, well, slop, as they only measure throughput, disregarding other properties of the system. For example, Zach Daniels measured new post notifications delivery rate under load, and discovered a 1% successful delivery rate in the Rust version under heavy load. Not a great look. Benchmarks are hard. But even the improved benchmark may not be actually testing what you want to test, depending on the characteristics of your system. You see, when you test a system you may want to emphasize different properties under load. Both DHH's and Zach's benchmarks were closed-loop benchmarks, so they were testing "how many requests can the system handle in a specific amount of time?". The test was using N clients, each sending a new message as soon as it gets a response for the previous one. In many cases, though, the increased load on the system may come from a big number of users performing an action at the same time, who will not wait for other users to finish their actions. In this case you would prefer constant arrival rate, ie. you want to send requests at a constant rate rather than making the rate depend on how fast the system can respond. If you want to check how reliable the events delivery is, you would ideally want to compare the same number of events. Here, if you look at the results, it shows that Rust delivered 1% of notifications out of 6-7k, whereas Elixir delivered 100% of notifications out of ~1.7k. What it shows is that the Elixir version handles backpressure better, but this only holds as long as the number of requests don't overload the system.

But let's leave benchmarks just for a moment and talk about trade-offs, cause programming is all about trade-offs. Sure, there are situations where a tool or a solution is clearly better, without any downsides, but it's quite rare, especially when you get to complex systems that have to be reliable. I've seen quite a lot of people from the Elixir community coming to various conclusions after they've seen the 1% delivery rate of the Rust version, without really trying to understand why it happens. The general consensus? Elixir is just better at concurrency! In Rust the scheduler is cooperative, how can you even live like this? I like Elixir, and I've successfully used it in production, but it's not a silver bullet. Yes, Elixir (or other BEAM based languages) are very good at concurrency, and admittedly writing concurrent code in Elixir is, in general, easier than in Rust, but it comes at a cost of not having low level control, higher memory usage, and oftentimes, speed. There is a reason why rustler exists. And proclaiming language's superiority based on a single metric without knowing the root cause, may be misleading.

Remember how I mentioned the closed-loop stress test shows just one of the properties of the system as Rust processed ~4 times more requests and thus had to handle more events sent to the client through a WebSocket? I've rerun the test with constant delivery rate using the DHH's Rust version and Zach's Elixir version with various fixes. At 100 POSTs/s the clients received ~14% events in Rust. In Elixir it was ~60%. Still better, right? Not so fast! At this traffic level Rust didn't have any HTTP errors. Elixir timed out on ~23% of HTTP POST requests. And what about latency? The worst event delivery latency was close to 180s. In Rust when the deliveries are lagging, clients get disconnected. On reconnect, a browser client would fetch the latest messages, which largely invalidates the need for the missed events. What do you think is better UX: the client silently reconnecting in the background and fetching the new updates, or waiting for an update about a new message for 3 minutes? Which just shows that, again, a single metric doesn't tell the whole story.

Getting back to trade-offs. Do you know *why* the Rust version drops so many messages under heavy load? It uses `tokio::sync::broadcast` channel to broadcast events to connected clients. An event informing about a new chat message may need to be sent to multiple connections, so that makes total sense. One of the properties of the broadcast channel is that it's bounded by a set capacity. If a receiver can't handle the messages fast enough, the receiver will get a `RecvError::Lagged` error (refer to the docs to learn more about lagging). The broadcast capacity in the Rust rewrite was set to 256. In Elixir, the `GenServer` process is handling events delivery and by default `GenServer` mailboxes don't have a limit. A single line change in Rust:

```
- stream_capacity: 256
+ stream_capacity: 16384
```

increases the delivery rate at 100 req/s from ~14% to ~90%. It's better than Elixir now, isn't it? Not really. I think that this version is actually worse because in this case it's better to fail fast and force the client to reconnect rather than handle things extremely slowly. You know what else changed after raising the capacity? The pMAX event delivery latency went up from 11s to >130s, similarly to how Elixir behaved, which I think is strictly worse than disconnecting the lagging clients. If you ask me, even 11s is too long and if the receiver can't pass the event faster than that, it's better to drop the event. Which shows how important it is to set sensible constraints in the system, and that, in fact, Elixir isn't just automatically fixing all of the concurrency issues out of the box. Also, if you don't set bounds yourself, you will likely run into external limits. During the 100 req/s stress test the Elixir version reached 1.8GB memory usage. Another thing to ponder on: is it better to drop some messages or get OOM killed? Trade-offs all the way. Elixir is great, but it won't magically solve all of your problems. No matter the language you're using, you have to think about failure modes and trade-offs.

So what have we learned today? You still have to think critically. It's good to know what you're doing. Don't make hasty assumptions based on a single metric. If you want a reliable system you should know when to fail. Also, benchmarks are hard.

If you like this post please consider following me on Twitter.

Top

© 2021 Aaran Xu. Made with Zola using the Tale-Zola theme.