Tick Tock: Maintaining Time

Intro

Hello! Today I'll be talking about my experience maintaining the most widely used library for handling date and time in Rust, the `time` crate, as well as providing a brief history and overview of the crate itself.

■ Background

First, some background. While this predates my involvement with the `time` crate, it may surprise you to know that `time` was originally part of Rust's standard library, but was removed in the language's first 1.0 `alpha` release. However, the crate was still maintained by the same folks maintaining the standard library. This didn't last long, ■ and by the following year the crate was officially unmaintained.

Because I had nothing to do with the crate at this point in time, I won't be touching on the maintenance of the 0.1 series. In fact, I wasn't even aware of the crate's existence because I hadn't started using Rust until a couple months after it was marked unmaintained.

But by pure coincidence, today marks seven years to the day since I requested to take over maintenance of the `time` crate, and to my amazement (still!), the answer was "yes". A few months after this request, ■ I had made my first release, kicking off the 0.2 series. This was a complete rewrite of the crate, even having a different git root. Some methods were carried over for compatibility, but that was overwhelmingly not the case.

Overall, I would say that the 0.2 series was ■ a great learning experience. While I had written software that was used by others, I had never written a *library* used by more than few thousand people. ■ Having performed a thorough survey of existing libraries, not just in Rust but in many other languages, I was generally able to learn from the mistakes of others regarding API design. That's not to say the 0.2 series was perfect — far from it. One of the major reasons for releasing the 0.3 series was

■ the quick number of deprecations that had accumulated due to oversights or not knowing the Rust ecosystem well enough. Two major cases where this was true was the assumption of UTC when a UTC offset was not known, which violates a basic principle to never assume nontrivial information, and ■ the primary methods for constructing various types panicking on invalid input rather than returning an error. This was patched by adding a second set of APIs, but that is quite messy both for implementation and for users.

■ The 0.3 Series

The 0.3 series isn't a ground-up rewrite; the core types from 0.2 stayed. But July 2021 delivered a breaking release that settled every outstanding debt at once. The 0.1 compatibility layer was removed, every panicking constructor was replaced with a `Result`-returning one, every function that silently assumed a UTC offset was either removed or renamed. Type and method names were standardized across the board by following Rust's API guidelines. `#![no_std]` support landed. `Instant` got a guaranteed representation. Many methods became `const fn` where the compiler allowed. It was the kind of release that accumulates every breaking change the project needed and ships them together, rather than sprinkling them across many incremental updates.

But here's the thing: it's never been a "ship and forget" series. Since that initial release, ■ I've shipped over fifty patch versions, and the pace hasn't slowed. The compiler stabilizes something new? Wait a while (which is important), bump MSRV and ■ tighten the implementation, but be sure that in doing so it either improves the situation for downstream users or improves maintainability. The MSRV is never bumped just because. ■ Security report lands in my inbox? That's my top priority for the crate and a patch will ship as soon as possible. ■ My goal is to do it right, and if I can't do it right now, do it ■ safely in a way that lets me come back to it later. ■ That's why every deprecation in 0.3 comes with a migration path, and why the next breaking change will be the result of years of careful, semver-compatible preparation rather than an off-the-cuff decision. The work I'll be describing in the rest of this talk — the compiler tricks, the performance work, the spin-off crates, even the language proposals — all of it has landed within this single 0.3 series, one patch release at a time for over five years.

■ Types Overview

But what exactly does the `time` crate provide? It is thorough, though not exhaustive. Starting off with what I would consider the "core" types, ■ we have `Month` and `Weekday` which are both `enum`s and should be self explanatory.

Similar to the standard library, ■ we have a `SignedDuration` type representing seconds and nanoseconds, but unlike the standard library's equivalent it can be negative, which is useful for representing the output of arithmetic.

Next up is a ■ `Timestamp`, which is a relatively recent addition. This type is particularly useful for situations where the month, day, etc. don't need to be immediately known, such as when storing a point in time in a database.

On to the most used types, we have ■ `Date` and `Time`. Both are what they sounds like; `Date` is in the Gregorian calendar (the one you're likely most familiar with) with the caveat that we pretend that system has always been in use. `Time` has nanosecond precision. If you want to combine the two, ■ you can use `PlainDateTime`. Note that a `PlainDateTime` does not have the concept of a UTC offset or time zone, so it does not represent a specific point in time.

If you need that, ■ there is a `UtcOffset` type with second-level precision and ■ an `OffsetDateTime` type that combines a `PlainDateTime` with a `UtcOffset` to represent a point in time. However, a fixed UTC offset is not the same as a time zone — that's being worked on but isn't yet publicly available. As an optimization, ■ there is also `UtcDateTime` which is equivalent to an `OffsetDateTime` that is known to be UTC. This provides better performance as it can skip a number of checks and calculations.

Another recent addition ■ is three iterator types, which let you iterate over `Date`s, `Month`s, and `Weekday`s. All three are well optimized, overriding trait methods to avoid unnecessary computation.

For formatting and parsing, ■ there are specific types for the formatting string, something that has evolved quite a bit due to being non-exhaustive. I'll cover formatting and parsing in more detail later, but for now I'll say that the low-level details of older versions are exposed for advanced use cases.

Finally as far as types, ■ there are numerous error types, plus some other miscellaneous and helper types that aren't worth delving into.

■ Helper Macros

Now that you've seen the types themselves, how do you actually create the values? A number of helper macros are provided.

■ Starting with constructing a `Date`, the year-month-day format is used, though you can also use the day of year or the week number and weekday. ■ For `Time`, it's either the 12 or 24 hour clock with precision to the nanosecond. ■ `PlainDateTime` simply combines the two. ■ `UtcOffset` uses the signed hour, with minutes and seconds being optional. ■ `OffsetDateTime` combines a `PlainDateTime` and `UtcOffset`. ■ `UtcDateTime` uses the same syntax as a `PlainDateTime`. Last for the main types is ■ `Timestamp`, which can be either the timestamp itself in seconds ■ or the same syntax as a `UtcDateTime`.

Note that the `datetime!` macro generates either a `PlainDateTime` or an `OffsetDateTime` depending on whether an offset is provided. This is because macros operate on the raw tokens, so it's possible to determine this and alter the output relatively easily.

All of these macros are validated at compile time, so if it compiles, you know the value is valid. The output is truly zero cost, as it is *guaranteed* to be evaluated using const eval.

■ Spin-Off Crates

Those macros are guaranteed zero-cost, but not everything I've built for `time` belongs inside `time` itself. Whether spun off for third-party utility or developed with that in mind from the start, these crates show that `time` isn't a monolith and can split off parts when appropriate.

I'll start off with ■ the `standback` crate, which is no longer maintained. It used to provide APIs from newer versions of the standard library for older compiler versions, allowing a lower MSRV to be used while still taking advantage of newer

library features. For folks familiar with JavaScript, you probably know this as a polyfill.

There is ■ also `num_threads`, which is a tiny crate that lets you know how many threads are running in the current process. That's it! It was needed for soundness and is still relied upon in one specific situation.

■ `powerfmt` is a crate that makes it easier to support padding and alignment when implementing the `Display` trait. It was used for all types in `time` that implement `Display`, but these trait impls are now deprecated because I found it was more efficient to use a pre-allocated buffer and work from there. With that said, `powerfmt` is still maintained.

■ `num-conv` is another lightweight crate that performs integer-to-integer conversions without using `as` casts. This allows for self-documenting method calls and prevents silent truncation of values. Using this crate has caught at least one bug in `time`, and it is useful enough that it is currently being uplifted into the standard library.

The biggest success in terms of spin-off crates has got to be ■ `deranged`, which provides ranged integers. I use it to ensure that values like the clock hour are within range, enforcing the requirement at the type system — the only way to create such a value is a checked or static constructor or `unsafe`. Once const generics get more features, `deranged` will be even more powerful, able to statically ensure even more invariants.

Finally, there are ■ `zoneinfo` and `zoneinfo-static`, which are crates that have not yet been released. `zoneinfo` will provide a reader and writer for TZif files, which are the binary format used by the time zone database. It will also provide the low-level types that contain all information necessary to represent a time zone. `zoneinfo-static` will provide a precompiled version of the time zone database, which can be used on systems like Windows that don't have a copy available.

■ Dependency Tree

While talking about other crates, it's worth taking a look at what dependencies are

used by `time`. What I'm covering here is the dependency tree for Unix, as it is slightly different on other platforms.

Starting off, we have the three crates that are part of the main repository: `time`, `time-macros`, and `time-core`. `time-macros` is optional, which is indicated by the lack of a black border. That is to say that you don't *have* to use it, but it's there if you want it. `time-core` exists in order to share code between `time` and `time-macros`, which also improves compile times a tiny amount.

Aside from these three main crates, ■ there are the spin-off crates that I have already described. `powerfmt` is the only one of these where its usage in `time` is deprecated, though the crate itself is not. The `num_threads` crate is needed to ensure soundness when refreshing the `TZ` environment variable, which is only done when the user requests it. Because the system UTC offset is optional, so is the dependency.

But we still need to be able to obtain the system UTC offset in the first place, which is what ■ I pull in the `libc` dependency for.

Finally, ■ we pull in three optional dependencies for third party interop: `quickcheck`, `serde_core`, and `rand`; these can be enabled independently. `rand` spans three versions (0.8, 0.9, 0.10) because 0.8 was the most recent when I started the 0.3 series. All of these have their own dependencies, which I'm not showing here.

That's the current dependency tree, but what about the future? It's not going to grow. The first and most obvious change is that ■ `powerfmt` will be removed because the same functionality is provided more efficiently and can't easily be moved into a separate crate.

Beyond that, the ■ `num-conv` crate is being uplifted into the standard library; once that is stabilized, I can remove that dependency too.

Longer term, it's *possible* we could get a function in the standard library to remove the `num_threads` dependency, and I hope we'll eventually get ranged integers in the language. ■ With those two deps gone, the tree would be quite minimal. I also plan to move more into `time-core` once `rustdoc` stabilizes `cfg` annotations, and

I'd *like to* split formatting and parsing into a separate crate, plus there will be an integral crate for time zone database integration. These last few changes are speculative, but the result will be smaller, more organized, and permit faster compilation.

■ Formatting and Parsing (pt. 1)

One area that may eventually be its own crate — as I just mentioned — is formatting and parsing. For `time` 0.3, I made a massive change there. Instead of using format descriptions equivalent to those from C, I created a new syntax that is more readable, composable, flexible, and extensible.

■ Here's an example. In C, you might use a format description like this *(show screen)*. I don't know about you, but that is not at all clear. If you're reviewing code, would you know what this means without having to look it up? I certainly wouldn't. In `time`, the same ■ would be written this way *(show screen)*. Yes, it's longer. But it's also immediately apparent what on earth you are doing, which I feel is far more important.

By using this different syntax, I added the ability to have various modifiers (the `repr:short` part in the example). This lets you choose case sensitivity when parsing, padding, and the representation of the value itself, among other things. It's not exhaustive, so there's always the possibility of adding more.

There are also a handful of well-known formats that are supported, ■ specifically ISO 8601, RFC 2822, and RFC 3339. All are zero-sized types that have formatting and parsing hand-written for performance, with ISO 8601 even being configurable to support various options permitted by the standard.

Honestly, I *hate* implementing well-known formats. They are an absolute pain, typically dealing with weird edge cases, deprecations, and cross-referencing all of that three levels deep. To give you an example of what is *technically* permitted by RFC 2822, ■ take a look at this. *(pause)* If a parser does not handle this, it is not technically spec-compliant. For some bizarre reason, the spec allows for arbitrary whitespace, control characters, null bytes, and even comments — and the comments can be *nested*. You can't even do that in many programming languages!

Yet it is something that is permitted, so I handle it correctly.

■ Formatting and Parsing (pt. 2)

What would you call something that accepts input, lexes it, parses it to a syntax tree, lowers it to an intermediate representation, performs optimizations, and generates output? Yeah, ■ the `time` crate is a compiler — or more specifically, the parser for format descriptions is.

At runtime, ■ you'd call the `parse_borrowed` or `parse_owned` function along with the version of the format description that you want to use. Being the runtime parser, they return a `Result`. For some components, additional allocation is necessary, which is why you need the `owned` variant.

To avoid the downsides, there is ■ the `format_description!` macro, which operates entirely at compile time. As with the functions, it is versioned, but the syntax for the format description is the same. For version 2, it returns a slice of items, but ■ for version 3 it returns an opaque type to allow for further improvements down the line. As with the constructor macros before, you'll get a compile error if the format description is invalid.

■ Let's take a look at what the version 3 macro expands to. The first thing that is obvious is that it's placed within a `const` block, guaranteeing everything is evaluated at compile time with zero runtime overhead.

■ After a pair of trivial imports, we start building the format description. ■ At the top level is a borrowed compound item, meaning no allocation is necessary. Because it's a compound, ■ it accepts a slice as a parameter. Within the slice, ■ the year is split into multiple variants — this lets the compiler better optimize based on which paths are reachable. All other modifiers ■ are in the field of that variant, which is the default here. After the year comes ■ a literal dash, ■ then the month, similarly split into multiple variants, ■ another dash, ■ and the day, which is a single variant.

All of this is ■ then converted into the opaque `FormatDescriptionV3` type. That call isn't a simple wrapper — it also runs an internal `const fn` to compute the

maximum length of a formatted value, which is stored so formatting can pre-allocate the full buffer instead of allocating repeatedly. And what's *not* shown here is that the macro also has 9 optimizing passes, all unobservable to the user other than the improved runtime performance.

■ Pushing the Compiler

Being able to determine the maximum width of a format description at compile time is great, but being able to do this without resorting to hackery in a macro is even better. That's just one of the ways that I have pushed the compiler to do more work for me.

In the previous slide, you may have noticed that I called a `default` method inside of a `const` block. But traits can't be const! ■ What I do is define an inherent method with the same type signature, letting the compiler resolve to that instead of the trait method. Once const trait impls are stabilized, I can remove the inherent methods without it being a breaking change.

■ Another way I've pushed the compiler is by using *both* named and anonymous generics on the same function. While unusual, this was necessary to avoid declaring a unit type twice. ■ I was able to eliminate magic constants *and* have it be generic over the type of the constant. Internally I use a second function with a named generic to obtain the constant from the trait impl of the anonymous type. It makes intent clearer, particularly when the same value has multiple meanings. If the values do not divide evenly, a descriptive error message is output.

■ One situation that I took advantage of an explicit statement of behavior being unspecified is ■ when I used the configuration of ISO 8601 to statically ensure that the configuration was compatible with the type. In this example, it's known that the configuration requires a date to be present, but the type being formatted (a `Time`) does not provide that information. Because of that, a const assertion is raised, and this will never be compiled.

■ Regarding ISO 8601, the configuration itself is my favorite hack. const generics are currently limited to only primitive types. However, ■ I wanted to use a `struct`. After some thinking, I realized that I could have the user call an `encode` method on

the config, which would return a `u128`. That is then passed as a const parameter, and the `decode` method is called internally to extract the information back out, storing it as an associated constant. This only works for types that are at most 16 bytes and you'll need to manually write the encoder and decoder, but it was worth the effort to ensure that the config was thorough while remaining a zero-sized type. Once const generic ADTs are stabilized, the `encode` method will be a no-op, with the type of the generic being changed to the config itself.

■ Finding Relevant Algorithms

Pushing the compiler catches misuse at compile time, but it can't fix incorrect math. Date and time computation is notoriously hard, but that doesn't mean you have to reinvent the wheel. For most algorithms I've needed, ■ there are already published solutions. They aren't always the easiest to find, but it is quite the rabbit hole to go down.

Sometimes, though, there simply isn't an algorithm out there. When that's the case, I've designed my own algorithms. ■ They start out quite simple and due to naïveté are generally not the most efficient. But it's *good enough*, and that's what matters at first. Eventually, I want to improve a number of these, and ■ in one situation I developed one from first principles. The QR code links to my write-up, but long story short: I took a known pattern in the calendar and got an approximation that worked for most values. With refinement and mathematical tricks, I got a branchless, integer-only algorithm that remained `const`-compatible, more than doubled throughput, and is provably correct for all values. Others have stated that this algorithm is novel as far as they are aware.

■ With that said, ■ just because a problem is *solved* does not mean that it is *done*. The simplest example of this is determining whether a given year is a leap year. The textbook case that follows the centuries-old rules is written here, but it turns out that is actually quite inefficient. Researchers have been able to improve on that quite a bit; the `time` crate has had a number of implementations of this function over the years, starting with the approach on screen, going through a handful of different approaches before finally ■ settling on an algorithm a few months ago. With a minor modification, I was able to improve some cases further, landing on

what is now the fastest known algorithm for this problem.

■ There is a similar story with the mapping of dates to integers and back. The first implementation was wrong for years before the Julian epoch. I found a paper proven correct for all values, adapted it to my internal representation, and the author even updated the paper to include my version. That held as fastest until Ben Joffe created an even faster implementation.

■ With that, I'd like to call out Peter Baum, Howard Hinnant, Cassio Neri, Lorenz Schneider, and Ulrich Drepper for their contributions to this space. And especially Ben Joffe, whose work has replaced multiple implementations with faster ones this year, and whose research I hope will eventually be included as a general-purpose LLVM optimization.

■ Performance (pt. 1)

Aside from algorithmic changes, a lot of work has gone into improving the performance of pre-existing code. This has been done in a number of ways — adding fast paths, removing unnecessary allocations, leveraging compiler hints, and in some cases rewriting the code to be slightly different in order to let the compiler optimize it better.

To give an idea of how successful this effort has been, I've got a number of benchmarks to show you. These measure elapsed time, not throughput, so smaller is better. For consistency, I'm showing relative rather than absolute change. ■ Starting off with RFC 3339, formatting now takes half the time, with parsing being a third faster. Remember that bizarre example of valid RFC 2822 from earlier? ■ By adding a fast path for the typical case and marking the deprecated case as cold, parsing is now five times faster. Formatting has a similar speedup to RFC 3339. If you're ■ handling format descriptions at runtime, that's four times as fast in large part due to reorganizing the internals of the parser to be more friendly to the compiler. But ■ formatting a value using that format description is also quite a bit faster thanks to preallocating the output buffer among other smaller changes.

The biggest wins are ■ with the `Display` implementations. `Date` is four times faster, `UtcOffset` five, `Time` six, `PlainDateTime` and `UtcDateTime` eight, and

`OffsetDateTime` eleven. This was similarly done by preallocating the buffer and writing into that before copying over the output, rather than formatting directly into the output which is far more expensive.

On top of preallocations, I took the code from `itoa` (which by the way will land on stable soon) and adapted it to be even more efficient for my specific use case. For an 11× speedup, it was worth it.

■ Performance (pt. 2)

One of the areas I sped up was formatting subsecond values. This formats it only to the necessary precision, so there are no trailing zeros unless it's the only digit. By leveraging compiler optimizations, I was able to write this in an extremely efficient, branchless way that compiles down to SIMD *and* even outperforms the same code when written using portable SIMD on nightly.

■ Starting off, we know that we have at most nine digits to format, so we store that in a struct. The `repr` will make more sense later, but for now what's important is that the digits are all adjacent. First off, ■ I call into a helper method to obtain the digits as pairs. Once we have all of the digits as strings, ■ they're loaded into the `Digits` struct. In reality, the compiler does this all in one go. So far, nothing is particularly unusual, we've obtained the nine digits and stored them consecutively. If we didn't want to get rid of trailing zeros, we'd be essentially done. But next is where the magic happens.

■ Loading the 2nd through 9th digits into a `u64`, an XOR is performed. What that does is test if any given byte is equal to zero. Note that if `Digits` weren't aligned to 8 bytes, loading into a `u64` could require a copy; instead it is absolutely free because we specified the `repr`. So now we have a bitmask, but how does that help us? ■ We can determine the number of digits to truncate by pulling the leading zeros and dividing by the number of bits in a byte. ■ Obtaining the length is trivial — just subtract from the maximum.

Now we know how long the string is, and we have the digits in consecutive memory from before, so ■ all that's left is to get a pointer to the first digit and create a string from that. As we constructed the buffer within the function, a

custom struct is used to avoid lifetime issues; it's essentially the same as a string reference with a statically known maximum length.

By ensuring that data is laid out in memory in a specific way, I was able to eliminate copies when constructing the `u64`, let the compiler optimize *both* the bitmask creation and determination of how many digits to truncate into SIMD, so all eight digits are processed at the same time. Overall, this code runs approximately twice as fast as the previous implementation.

■ Niche Value Optimization

On to one of the simpler optimizations that I have done: niche value optimization. Due to the way data is laid out, some bits or values are inherently unused. For example, I store a `Date` as a 32-bit integer, with the year being the high 22 bits, the next bit being a flag for if it's a leap year, and the final 9 bits to store the day of the year. However, the day of the year is guaranteed to be between 1 and 366 inclusive, so the value of the entire `i32` necessarily cannot be zero. For this reason, I changed the representation from a `i32` to a `NonZero<i32>`, so if you have an `Option<Date>`, it is the same size as `Date` itself.

■ A similar story is true with `Time`. ■ There are three fields that are `u8` and one that's `u32`. Due to alignment, this requires 8 bits of padding. The compiler doesn't make any guarantees about what those bits are, so it can't make any optimizations based on them. However, *I* can guarantee the layout, which has immeasurable performance overhead. The way to do this is actually quite simple. ■ Create an `enum` with a specified `repr` and then store the single variant. The compiler now has knowledge of 254 values that can never be used and will optimize layouts accordingly. When a `Time` is stored within another struct, the compiler still has that knowledge and can pass through the niche values.

■ CI/CD (pt. 1)

Niche optimizations like that only matter if you can ship them without breaking anything. ■ That's where CI comes in, which is a critical part of maintaining a library like `time`. Being able to push a commit and have it thoroughly tested in just

a couple minutes is necessary for rapid development. Years ago, ■ tests were split into statically-known sets that finished in approximately the same time, but as more feature flags were added, ■ the splits became unbalanced.

A few months ago I decided to tweak the approach slightly. ■ Instead of determining the splits by feature flags, generate the list of tests and then split them programmatically. ■ That cut CI from 20 minutes down to six and a half. As it turns out, ■ there were a couple of trivial changes, one of which I could simply ask for: ■ raising GitHub's limit of 20 concurrent runners. Just a day after reaching out, my limit was bumped to 30. By pure chance, ■ it was around this time that GitHub also added native parallel jobs within a single runner, letting me reorder some things to run in the background. With everything combined, ■ CI now takes just four and a half minutes.

Publishing was similarly slow — I had a matrix that eventually took 80 minutes and required me to monitor it. ■ I sharded that too, bringing it down to 20 minutes. Publishing is now fully automated through GitHub and prohibits manual publication. Only crates where the version number has changed are published; the rest are deliberately skipped.

■ CI/CD (pt. 2)

So what does my CI look like now? It starts by ■ computing the shards, which is functionally instantaneous. At the same time, ■ a sysroot for miri is prepared, as there's no need to rebuild it for each miri runner. ■ A miscellaneous runner handles publishing docs, formatting, linting, type-checking benchmarks, doc tests, code coverage, and cross-compilation.

While that's happening, ■ a test matrix runs all tests on stable and MSRV across Windows, Linux, and macOS — six runners total. ■ Then a separate "check targets" matrix spawns six shards each for stable and MSRV, validating a large set of feature-flag combinations on many targets so builds don't break unexpectedly.

Also run on every push is ■ the entire test suite under miri, split into 16 shards. It started at 20 minutes, but sharding and the pre-built sysroot brought it down to a minute and 40 seconds. Once that completes, ■ a small runner is spawned to

delete the miri sysroot, with another job outputting the overall pass/fail when everything is done.

From start to finish this takes about four and a half minutes, verifying that the code is formatted, lint-clean, documented, tested on all major platforms, MSRV-safe, sound, and compiles across thousands of feature-flag combinations.

This setup catches regressions early, prevents unsoundness, and ensures that `main` can always be published.

■ It's bug free...right?

So CI ensures that there's no bugs, right? I wish! ■ It can't catch what isn't tested. To give an example, ■ the division implementation for `SignedDuration` was wrong for *years* and was only caught because I added a semi-automated property test. I knew I had based that implementation on the standard library, and sure enough it was wrong there too. In fact it was wrong for as long as it has existed — well before Rust 1.0.

■ But the largest breakages were not entirely my fault. When Rust 1.80 was released, it included a major type inference regression that was caught during the crater run, but due to process failures it wasn't reverted before it landed on stable. Separately, `time` 0.3.48 broke downstream crates due to a decade-old compiler bug. Neither was detectable with any extant tooling, so short of my own crater run, I couldn't have known about the risk.

■ Security reports are the top priority. I have gotten a handful of reports over the years, starting with a difficult-to-encounter segfault triggered by a race of environment variables. Initially, I stripped the functionality, but eventually I was able to make a small change that resolved the issue while keeping the ability to update the environment variable behind a separate function with soundness checks.

More recently, there was a report of a potential denial of service attack vector via unbounded recursion, which was disclosed privately and fixed within days. A bit over a month ago I was informed of a bug in some unreleased code discovered

during an LLM audit. I confirmed the validity and patched it before it was even released. When doing quote-unquote "weird" things, it's easy to run into undefined behavior, and ■ it's inherently difficult to detect. UB can only be caught when it's actually encountered, so having thorough testing is critical and is something I am still working on improving.

■ Language Changes

My experience maintaining `time` isn't limited to just the crate itself. Because of shortcomings and bugs that I've run into, ■ I have authored multiple RFCs to improve the language itself.

■ The first was for `#[derive(Default)]` on `enum`s. I scaled it back in order to avoid concerns about derived bounds, but this is something that's used all the time and I am happy to say that I authored the RFC, implemented it, and stabilized it.

■ The second RFC proposed was for restrictions. If you've ever used the sealed trait pattern, there is now native syntax for it. Similarly, you can restrict field mutability to a given path. Both of these are available on nightly — ■ *please* test them and provide feedback! We'd like to stabilize them soon, but there's still a nontrivial question about syntax and we want to know what people think.

■ Related to that, `unsafe` fields was initially proposed by me and is also implemented on nightly. This would make it possible to mark a field itself as having a safety invariant such that accessing it in any way is `unsafe`.

■ Default field values was similarly written in large part by me and lets you provide defaults right in the struct declaration, so construction can omit them — cutting down on manual `Default` impls and many builders. The RFC is accepted and on nightly.

■ Finally, there's an open RFC to allow logical operations ('and', 'or', and 'not') in `cfg` expressions. This is currently waiting on the language team to make a decision, but I think it would make a ton of sense to make `cfg` more consistent with the rest of the language.

■ Near Future

Those language improvements will help with what's to come. For years, I have been tracking over 60 action items — some menial, others requiring dozens of hours of highly-technical and skilled work. That list has shrunk significantly, to only a few medium-to-large items remaining.

Among them ■ is a thorough performance audit, which I'm sure you could gather is well underway. Ensuring that code is as fast as possible is a never-ending task, but one that teaches me a significant amount about compilers and even CPUs at the hardware level.

One that I haven't gotten around to yet ■ is improving the documentation. It's functional today, but I want to add real-world examples guiding users to best practices rather than hoping they find the right method and struggle if they don't.

I'd also like ■ to add a `TimeSpan` type that will be similar to a `SignedDuration`, but dealing with variable-length units like months and years. And if you want to add that repeatedly, you'd ■ be able to use an `Interval` type that would do this in a performant manner.

The final major implementation ■ is a `ZonedDateTime` type. This will be a true time zone-aware type that will handle daylight saving time, legislation changing the time zone entirely, and other edge cases. It's well underway — I have around 20,000 lines of code written for it. But since I don't personally have a use case for it, it's being written solely for the benefit of others, and because it's very time-consuming, it's not a top priority. ■ Unless there is funding for time zone database integration, the extension will almost certainly be licensed under AGPL with commercial licensing available.

But for format descriptions, users would be well-served by something more specific than the page where I largely dump the syntax with diagrams and prose. On that front, ■ I have already written a roughly 30 page specification for the format description syntax, semantics, and behavior. It still needs to be reviewed and copy-edited, but it'll eventually be the go-to resource for anyone who needs that extra level of detail or wants to use that syntax in their own project, perhaps

not even in Rust.

Once all of this is complete, which will be a significant milestone, I will make the breaking changes I have planned, ensuring that the API is as good as it can be, fully-featured, and with a long-term stability plan, ■ release the capstone version 1.0. And hopefully in years to come, I'll be able to come back here and tell you about what a success the effort was.

■ Outro

And that's what I have for today! Hopefully what you came for made the cut; I know it was very fast paced in order to squeeze in as much as possible. Sponsorship is always appreciated and can help avoid AGPL-licensing that extension. If you want to learn more, feel free to reach out however you'd like. My contact info is on screen and easily accessible on my GitHub. Thank you.